An Introduction to Computer Networks

Release 1.9.10

Peter L Dordal

February 19, 2018

CONTENTS

0 Preface 3
- 0.1 Licensing . . . 3
- 0.2 Classroom Use . . . 4
- 0.3 Progress Notes . . . 6
- 0.4 Technical considerations . . . 6
- 0.5 Recent Changes . . . 7

1 An Overview of Networks 11
- 1.1 Layers . . . 11
- 1.2 Data Rate, Throughput and Bandwidth . . . 12
- 1.3 Packets . . . 12
- 1.4 Datagram Forwarding . . . 13
- 1.5 Topology . . . 16
- 1.6 Routing Loops . . . 17
- 1.7 Congestion . . . 18
- 1.8 Packets Again . . . 19
- 1.9 LANs and Ethernet . . . 19
- 1.10 IP - Internet Protocol . . . 22
- 1.11 DNS . . . 27
- 1.12 Transport . . . 28
- 1.13 Firewalls . . . 32
- 1.14 Some Useful Utilities . . . 32
- 1.15 IETF and OSI . . . 34
- 1.16 Berkeley Unix . . . 37
- 1.17 Epilog . . . 37
- 1.18 Exercises . . . 37

2 Ethernet 43
- 2.1 10-Mbps Classic Ethernet . . . 43
- 2.2 100 Mbps (Fast) Ethernet . . . 54
- 2.3 Gigabit Ethernet . . . 55
- 2.4 Ethernet Switches . . . 56
- 2.5 Spanning Tree Algorithm and Redundancy . . . 59
- 2.6 Virtual LAN (VLAN) . . . 64
- 2.7 Software-Defined Networking . . . 65
- 2.8 Epilog . . . 72

2.9 Exercises . . . 72

3 Other LANs **79**
3.1 Virtual Private Networks . . . 79
3.2 Carrier Ethernet . . . 80
3.3 Token Ring . . . 81
3.4 Virtual Circuits . . . 82
3.5 Asynchronous Transfer Mode: ATM . . . 86
3.6 Adventures in Radioland . . . 88
3.7 Wi-Fi . . . 92
3.8 WiMAX and LTE . . . 112
3.9 Fixed Wireless . . . 116
3.10 Epilog . . . 118
3.11 Exercises . . . 119

4 Links **125**
4.1 Encoding and Framing . . . 125
4.2 Time-Division Multiplexing . . . 130
4.3 Epilog . . . 135
4.4 Exercises . . . 135

5 Packets **137**
5.1 Packet Delay . . . 137
5.2 Packet Delay Variability . . . 140
5.3 Packet Size . . . 141
5.4 Error Detection . . . 143
5.5 Epilog . . . 148
5.6 Exercises . . . 148

6 Abstract Sliding Windows **153**
6.1 Building Reliable Transport: Stop-and-Wait . . . 153
6.2 Sliding Windows . . . 157
6.3 Linear Bottlenecks . . . 161
6.4 Epilog . . . 168
6.5 Exercises . . . 168

7 IP version 4 **173**
7.1 The IPv4 Header . . . 174
7.2 Interfaces . . . 176
7.3 Special Addresses . . . 177
7.4 Fragmentation . . . 178
7.5 The Classless IP Delivery Algorithm . . . 180
7.6 IPv4 Subnets . . . 182
7.7 Network Address Translation . . . 187
7.8 DNS . . . 191
7.9 Address Resolution Protocol: ARP . . . 196
7.10 Dynamic Host Configuration Protocol (DHCP) . . . 199
7.11 Internet Control Message Protocol . . . 201

7.12 Unnumbered Interfaces . . . 205
7.13 Mobile IP . . . 206
7.14 Epilog . . . 207
7.15 Exercises . . . 208

8 IP version 6 **211**
8.1 The IPv6 Header . . . 212
8.2 IPv6 Addresses . . . 213
8.3 Network Prefixes . . . 215
8.4 IPv6 Multicast . . . 216
8.5 IPv6 Extension Headers . . . 217
8.6 Neighbor Discovery . . . 220
8.7 IPv6 Host Address Assignment . . . 224
8.8 Globally Exposed Addresses . . . 228
8.9 ICMPv6 . . . 229
8.10 IPv6 Subnets . . . 230
8.11 Using IPv6 and IPv4 Together . . . 231
8.12 IPv6 Examples Without a Router . . . 235
8.13 IPv6 Connectivity via Tunneling . . . 237
8.14 IPv6-to-IPv4 Connectivity . . . 240
8.15 Epilog . . . 242
8.16 Exercises . . . 242

9 Routing-Update Algorithms **245**
9.1 Distance-Vector Routing-Update Algorithm . . . 246
9.2 Distance-Vector Slow-Convergence Problem . . . 250
9.3 Observations on Minimizing Route Cost . . . 252
9.4 Loop-Free Distance Vector Algorithms . . . 254
9.5 Link-State Routing-Update Algorithm . . . 256
9.6 Routing on Other Attributes . . . 259
9.7 Epilog . . . 261
9.8 Exercises . . . 261

10 Large-Scale IP Routing **267**
10.1 Classless Internet Domain Routing: CIDR . . . 267
10.2 Hierarchical Routing . . . 269
10.3 Legacy Routing . . . 270
10.4 Provider-Based Routing . . . 270
10.5 Geographical Routing . . . 275
10.6 Border Gateway Protocol, BGP . . . 275
10.7 Epilog . . . 293
10.8 Exercises . . . 294

11 UDP Transport **299**
11.1 User Datagram Protocol – UDP . . . 299
11.2 Trivial File Transport Protocol, TFTP . . . 311
11.3 Fundamental Transport Issues . . . 313
11.4 Other TFTP notes . . . 318

11.5 Remote Procedure Call (RPC) 320
11.6 Epilog 324
11.7 Exercises 324

12 TCP Transport **329**
12.1 The End-to-End Principle 330
12.2 TCP Header 330
12.3 TCP Connection Establishment 332
12.4 TCP and WireShark 336
12.5 TCP Offloading 338
12.6 TCP simplex-talk 338
12.7 TCP state diagram 343
12.8 TCP Old Duplicates 348
12.9 TIMEWAIT 349
12.10 The Three-Way Handshake Revisited 350
12.11 Anomalous TCP scenarios 352
12.12 TCP Faster Opening 353
12.13 Path MTU Discovery 354
12.14 TCP Sliding Windows 354
12.15 TCP Delayed ACKs 355
12.16 Nagle Algorithm 356
12.17 TCP Flow Control 356
12.18 Silly Window Syndrome 356
12.19 TCP Timeout and Retransmission 357
12.20 KeepAlive 358
12.21 TCP timers 359
12.22 Variants and Alternatives 359
12.23 Epilog 368
12.24 Exercises 368

13 TCP Reno and Congestion Management **373**
13.1 Basics of TCP Congestion Management 374
13.2 Slow Start 378
13.3 TCP Tahoe and Fast Retransmit 383
13.4 TCP Reno and Fast Recovery 384
13.5 TCP NewReno 387
13.6 Selective Acknowledgments (SACK) 389
13.7 TCP and Bottleneck Link Utilization 390
13.8 Single Packet Losses 393
13.9 TCP Assumptions and Scalability 394
13.10 TCP Parameters 395
13.11 Epilog 395
13.12 Exercises 396

14 Dynamics of TCP Reno **401**
14.1 A First Look At Queuing 401
14.2 Bottleneck Links with Competition 402
14.3 TCP Fairness with Synchronized Losses 410

14.4 Notions of Fairness . . . 417
14.5 TCP Reno loss rate versus `cwnd` . . . 418
14.6 TCP Friendliness . . . 421
14.7 AIMD Revisited . . . 423
14.8 Active Queue Management . . . 425
14.9 The High-Bandwidth TCP Problem . . . 429
14.10 The Lossy-Link TCP Problem . . . 431
14.11 The Satellite-Link TCP Problem . . . 431
14.12 Epilog . . . 432
14.13 Exercises . . . 432

15 Newer TCP Implementations **441**
15.1 Choosing a TCP on linux . . . 441
15.2 High-Bandwidth Desiderata . . . 444
15.3 RTTs . . . 445
15.4 A Roadmap . . . 445
15.5 Highspeed TCP . . . 445
15.6 TCP Vegas . . . 448
15.7 FAST TCP . . . 451
15.8 TCP Westwood . . . 453
15.9 TCP Illinois . . . 455
15.10 Compound TCP . . . 456
15.11 TCP Veno . . . 458
15.12 TCP Hybla . . . 459
15.13 DCTCP . . . 459
15.14 H-TCP . . . 462
15.15 TCP CUBIC . . . 463
15.16 TCP BBR . . . 467
15.17 Epilog . . . 471
15.18 Exercises . . . 472

16 Network Simulations: ns-2 **477**
16.1 The ns-2 simulator . . . 477
16.2 A Single TCP Sender . . . 479
16.3 Two TCP Senders Competing . . . 491
16.4 TCP Loss Events and Synchronized Losses . . . 506
16.5 TCP Reno versus TCP Vegas . . . 516
16.6 Wireless Simulation . . . 518
16.7 Epilog . . . 524
16.8 Exercises . . . 524

17 The ns-3 Network Simulator **527**
17.1 Installing and Running ns-3 . . . 527
17.2 A Single TCP Sender . . . 528
17.3 Wireless . . . 536
17.4 Exercises . . . 542

18 Mininet **543**

18.1 Installing Mininet . . . 544
18.2 A Simple Mininet Example . . . 546
18.3 Multiple Switches in a Line . . . 547
18.4 IP Routers in a Line . . . 550
18.5 IP Routers With Simple Distance-Vector Implementation . . . 552
18.6 TCP Competition: Reno vs Vegas . . . 555
18.7 TCP Competition: Reno vs BBR . . . 560
18.8 Linux Traffic Control (tc) . . . 560
18.9 OpenFlow and the POX Controller . . . 563
18.10 Exercises . . . 575

19 Queuing and Scheduling **579**
19.1 Queuing and Real-Time Traffic . . . 579
19.2 Traffic Management . . . 580
19.3 Priority Queuing . . . 580
19.4 Queuing Disciplines . . . 581
19.5 Fair Queuing . . . 582
19.6 Applications of Fair Queuing . . . 594
19.7 Hierarchical Queuing . . . 597
19.8 Hierarchical Weighted Fair Queuing . . . 599
19.9 Token Bucket Filters . . . 605
19.10 Applications of Token Bucket . . . 610
19.11 Token Bucket Queue Utilization . . . 611
19.12 Hierarchical Token Bucket . . . 613
19.13 Fair Queuing / Token Bucket combinations . . . 615
19.14 Epilog . . . 617
19.15 Exercises . . . 618

20 Quality of Service **623**
20.1 Net Neutrality . . . 624
20.2 Where the Wild Queues Are . . . 624
20.3 Real-time Traffic . . . 625
20.4 Integrated Services / RSVP . . . 627
20.5 Global IP Multicast . . . 628
20.6 RSVP . . . 633
20.7 Differentiated Services . . . 637
20.8 RED with In and Out . . . 641
20.9 NSIS . . . 641
20.10 Comcast Congestion-Management System . . . 642
20.11 Real-time Transport Protocol (RTP) . . . 643
20.12 Multi-Protocol Label Switching (MPLS) . . . 648
20.13 Epilog . . . 650
20.14 Exercises . . . 650

21 Network Management and SNMP **653**
21.1 Network Architecture . . . 655
21.2 SNMP Basics . . . 655
21.3 SNMP Naming and OIDs . . . 656

21.4 MIBs . . . 659
21.5 SNMPv1 Data Types . . . 660
21.6 ASN.1 Syntax and SNMP . . . 660
21.7 SNMP Tables . . . 661
21.8 SNMP Operations . . . 666
21.9 MIB Browsing . . . 671
21.10 MIB-2 . . . 672
21.11 SNMPv1 communities and security . . . 680
21.12 SNMP and ASN.1 Encoding . . . 682
21.13 SNMPv2 . . . 684
21.14 Table Row Creation . . . 695
21.15 SNMPv3 . . . 704
21.16 Exercises . . . 714

22 Security **717**
22.1 Code-Execution Intrusion . . . 718
22.2 Stack Buffer Overflow . . . 719
22.3 Heap Buffer Overflow . . . 727
22.4 Network Intrusion Detection . . . 733
22.5 Cryptographic Goals . . . 734
22.6 Secure Hashes . . . 735
22.7 Shared-Key Encryption . . . 739
22.8 Diffie-Hellman-Merkle Exchange . . . 748
22.9 Public-Key Encryption . . . 750
22.10 SSH and TLS . . . 755
22.11 IPsec . . . 771
22.12 RSA Key Examples . . . 774
22.13 Exercises . . . 777

23 Bibliography **781**

24 Selected Solutions **783**
24.1 Solutions for *An Overview of Networks* . . . 783
24.2 Solutions for *Ethernet* . . . 784
24.3 Solutions for *Other LANs* . . . 785
24.4 Solutions for *Links* . . . 786
24.5 Solutions for *Packets* . . . 786
24.6 Solutions for *Sliding Windows* . . . 788
24.7 Solutions for *IPv4* . . . 789
24.8 Solutions for *Routing-Update Algorithms* . . . 790
24.9 Solutions for *Large-Scale IP Routing* . . . 791
24.10 Solutions for *UDP* . . . 791
24.11 Solutions for *TCP Reno* . . . 792
24.12 Solutions for *Dynamics of TCP Reno* . . . 792
24.13 Solutions for *Mininet* . . . 793

Indices and tables **795**

Bibliography 797

Index 805

Peter L Dordal
Department of Computer Science
Loyola University Chicago

Contents:

"No man but a blockhead ever wrote, except for money." - Samuel Johnson

The textbook world is changing. On the one hand, open source software and creative-commons licensing have been great successes; on the other hand, unauthorized PDFs of popular textbooks are widely available, and it is time to consider flowing with rather than fighting the tide. Hence this open-access textbook, released for free under the Creative Commons license described below. *Mene, mene, tekel pharsin*.

Perhaps the last straw, for me, was patent 8195571 for a roundabout method to force students to purchase textbooks. (A simpler strategy might be to include the price of the book in the course.) At some point, faculty have to be advocates for their students rather than, well, *Hirudinea*.

This is not to say that I have anything against for-profit publishing. It is just that this particular book does not – and will not – belong to that category; the online edition will always be free. In this it is in good company: there is Wikipedia, there is an increasing number of other open online textbooks out there, and there is the entire open-source world. Although the open-source-software and open-textbook models are not completely parallel, they are quite similar.

The market inefficiencies of traditional publishing are sobering: the return to authors of advanced textbooks is usually modest. Costs to users are also quite high: both the direct cost of purchasing a book, and the lost-opportunity cost of not having access to a book. (None of this is meant to imply there will never be a print edition; when I started this project it seemed inconceivable that a print publisher would ever agree to having the online edition remain free, but times are changing.)

The official book website (potentially subject to change) is intronetworks.cs.luc.edu. The book is available there as online html, as a zipped archive of html files, in .pdf format, and in other formats as may prove useful.

0.1 Licensing

This text is released under the Creative Commons license Attribution-NonCommercial-NoDerivs. This text is like a conventional book, in other words, except that it is free. You may copy the work and distribute it to others for any noncommercial use, but all reuse requires attribution.

Creation of derivative works also requires permission. It is not entirely clear, however, what would be considered a derivative work, beyond the traditional examples of abridgment and translation. Any supplemental materials like exams, slides or coverage of additional topics would be new, independent works, and would require no permission. Even the inclusion in such supplements of modest amounts of material from this book would have a strong claim to Fair Use. In the open-source software world, the right to make derivative works is exercised whenever the software is modified, but it is hard to see how this applies to textbooks. The bottom line is that if you have a situation you're concerned about in this regard, let me know and I'll probably be happy to grant permission.

The work may not be used for *commercial* purposes without permission. Free permission is likely to be granted for use and distribution of all or part of the work in for-profit and commercial training programs, provided there is no direct charge to recipients for the work and provided the free nature of the work is made

clear to recipients (*eg* by including this preface). However, such permission should always be requested. Alternatively, participants in commercial programs may be instructed to download the work individually.

The Creative Commons license does not precisely spell out what constitutes "noncommercial" use. The author considers any sale of this book, even by a non-profit organization and even if the price just covers expenses, to be commercial use.

Some of the chapters contain source code; this is licensed under the Apache 2.0 license. Some code files (those which are derivative works) contain an official Apache license statement; others do not.

0.2 Classroom Use

This book is meant as a serious and more-or-less thorough text for an introductory college or graduate course in computer networks, carefully researched, with consistent notation and style, and complete with diagrams and exercises. I have also tried to rethink the explanations of many protocols and algorithms, with the goal of making them easier to understand. My intent is to create a text that covers to a reasonable extent *why* the Internet is the way it is, to avoid the endless dreary focus on TLA's (Three-Letter Acronyms), and to remain not *too* mathematical. For the last, I have avoided calculus, linear algebra, and, for that matter, quadratic terms (though some inequalities do sneak in at times). That said, the book includes a large number of back-of-the-envelope calculations – in settings as concrete as I could make them – illustrating various networking concepts.

Overall, I tried to find a happy medium between practical matters and underlying principles. My goal has been to create a book that is useful to a broad audience, including those interested in network management, in high-performance networking, in software development, or just in how the Internet is put together.

One of the best ways to gain insight into why a certain design choice was made is to look at a few alternative implementations. To that end, this book includes coverage of some topics one may never encounter in practice, but which may be useful as points of comparison. These topics arguably include ATM (*3.5 Asynchronous Transfer Mode: ATM*), SCTP (*12.22.2 SCTP*) and even 10 Mbps Ethernet (*2.1 10-Mbps Classic Ethernet*).

The book can also be used as a networks supplement or companion to other resources for a variety of other courses that overlap to some greater or lesser degree with networking. At Loyola, this book has been used – sometimes coupled with a second textbook – in courses in computer security, network management, telecommunications, and even introduction-to-computing courses for non-majors. Another possibility is an alternative or nontraditional presentation of networking itself. It is when used in concert with other works, in particular, that this book's being free is of marked advantage.

Finally, I hope the book may also be useful as a reference work. To this end, I have attempted to ensure that the indexing and cross-referencing is sufficient to support the drop-in reader. Similarly, obscure or specialized notation is kept to a minimum.

Much is sometimes made, in the world of networking textbooks, about **top-down** versus **bottom-up** sequencing. This book is not really either, although the chapters are mostly numbered in bottom-up fashion. Instead, the first chapter provides a relatively complete overview of the LAN, IP and transport network layers (along with a few other things), allowing subsequent chapters to refer to all network layers without forward reference, and, more importantly, allowing the chapters to be covered in a variety of different orders. As a practical matter, when I use this text to teach Loyola's Introduction to Computer Networks course, I cover the IP/routing and TCP material more or less in parallel.

A distinctive feature of the book is the extensive coverage of TCP: TCP dynamics, newer versions of TCP such as TCP Cubic and BBR TCP, and chapters on using the ns-2 simulator and the Mininet emulator. This has its roots in a longstanding goal to find better ways to present competition and congestion in the classroom. Another feature is the detailed chapter on queuing disciplines.

One thing this book makes little attempt to cover in detail is the application layer; the token example included is SNMP. While SNMP actually makes a pretty good example of a self-contained application, my recommendation to instructors who wish to cover more familiar examples is to combine this text with the appropriate application documentation.

Although the book is continuously updated, I try very hard to ensure that all editions are classroom-compatible. To this end, section renumbering is avoided to the extent practical, and *existing exercises are never renumbered*. This is an essential feature for a textbook that is often updated mid-semester, and a useful feature for any textbook that is updated at all. New exercises are regularly inserted, but with fractional (floating point) numbers. Existing integral exercise numbers have been given a trailing .0, to reduce confusion between exercise 12.0, say, and 12.5.

For those interested in using the book for a "traditional" networks course, I with some trepidation offer the following set of core material. In solidarity with those who prefer alternatives to a bottom-up ordering, I emphasize that this represents a *set* and not a *sequence*.

- *1 An Overview of Networks*
- Selected sections from *2 Ethernet*, particularly switched Ethernet
- Selected sections from *3.7 Wi-Fi*
- Selected sections from *5 Packets*
- *6 Abstract Sliding Windows*
- *7 IP version 4* and/or *8 IP version 6*
- Selected sections from *9 Routing-Update Algorithms*, probably including the distance-vector algorithm
- Selected sections from *10 Large-Scale IP Routing*
- *11 UDP Transport*
- *12 TCP Transport*
- *13 TCP Reno and Congestion Management*

With some care in the topic-selection details, the above can be covered in one semester along with a survey of selected important network applications, or the basics of network programming, or the introductory configuration of switches and routers, or coverage of additional material from this book, or some other set of additional topics. Of course, non-traditional networks courses may focus on a quite different sets of topics.

Instructors who make use of this book in a course, as either a primary or a secondary text, are strongly encouraged to let me know, as this helps support continued work on the book. Comments – from anyone – on clarity, completeness, consistency and correctness are also much appreciated. I can be contacted at pld AT cs.luc.edu, or via the book comment form.

Peter Dordal

Shabbona, Illinois

0.3 Progress Notes

Edition 1.0 was declared complete as of March 2014; the current edition is 1.9.10.

The intronetworks.cs.luc.edu website carries both edition 1.0 and also the current 1.9.10 edition.

As of February 2018 I am very close to edition 2.0.

At this point I am actively seeking reviewers – either for style or for technical accuracy.

0.4 Technical considerations

The book was prepared in reStructuredText using the linux Sphinx package, which can produce multiple formats from the same source. That said, the primary format is **html**. The table-of-contents sidebar and the text sidebars work best there. The html version also provides a "Quick search" box, though it only works for all-alphabetic strings; strings with hyphens such as "wi-fi" and "Diffie-Hellman" fail. The index is an effective alternative.

This book uses a modest set of unicode special characters. Unfortunately, some of these characters are not universally available in all browsers. The comma-separated characters in the first line, below, appear to have the most limited support.

⟨,⟩,≃,α,β,γ,λ,ϕ,τ,ϱ,Δ,⌒,⟶,⟵,⟷
(,),≈,α,β,γ,λ,φ,τ,ρ,Δ,^,–→,←–,←→
µ,√,∞,⩽,⩾,×,÷,±,–,≠,→,←,–,│,┌,┐,└,┘,┬,┴,├,┤,┼

The characters above should look roughly as they do in the following image (the first line is the one most likely to fail):

⟨,⟩,≃,α,β,γ,λ,ϕ,τ,ρ,Δ,⌒,⟶,⟵,⟷
(,),≈,α,β,γ,λ,φ,τ,ρ,Δ,^,–→,←–,←→
µ,√,∞,≤,≥,×,÷,±,–,≠,→,←,–,│,┌,┐,└,┘,┬,┴,├,┤,┼

If they do not, there are two options for browser-based viewing. If the second and third rows above display successfully, there is a **unicode-safer** version of the book (both online and zipped) available at intronetworks.cs.luc.edu that has the characters in the first row above replaced by those in the second row.

The other alternative is to add an appropriate font. Generally Firefox and Internet Explorer display the necessary characters out of the box, but Chrome does not. The Chrome situation can usually be fixed, at least on "real" computers, by adding a font and then tweaking the Chrome font settings. I have had good

luck with Symbola (at shapecatcher.com/unicodefonts.html and other places). To install the font, extract the .ttf file and double-click on it. Then to adjust Chrome, go to Settings → Show advanced settings → Customize fonts (button), and change at a minimum the default Sans-serif font to Symbola. Then restart Chrome.

Unfortunately, adding fonts to (non-rooted) Android devices continues to be very difficult. Worse, Android often fails to display even a box symbol "□" in the place of missing characters.

If no available browser properly displays the symbols above, I recommend the pdf format. The unicode-safer version, however, *should* work on most systems.

At some point I hope to figure out how to handle this font situation a little better using Javascript. This turns out, however, not to be straightforward, and progress has been slow.

The diagrams in the body of the text have now all been migrated to the vector-graphics .svg format, although a few diagrams rendered with line-drawing characters appear in the exercises. Most browsers now (2018) appear to support zooming in on .svg images, which is a significant step forward.

0.5 Recent Changes

February 19, 2018 (ver 1.9.10): Expanded content on QUIC (*12.22.4 QUIC Revisited*).

January 5, 2018 (ver 1.9.9): Corrections to *15.5 Highspeed TCP*, added *15.10 Compound TCP*, *8.14 IPv6-to-IPv4 Connectivity* and NAT64, updates to *20.6.1 A CDN Alternative to IntServ* and *3.7 Wi-Fi*, miscellaneous other updates.

October 27, 2017 (ver 1.9.8): multiple corrections to *15.16 TCP BBR*; some updated exercises.

October 3, 2017 (ver 1.9.7): Added `tbf` and `htb` examples (*18.8 Linux Traffic Control (tc)*) to *18 Mininet*; miscellaneous exercise updates, and more on CDNs (*20.6.1 A CDN Alternative to IntServ*).

September 4, 2017 (ver 1.9.6): Added a webserver example (*18.3.1 Running a webserver*) to *18 Mininet*; updates to *7.8 DNS*.

August 27, 2017 (ver 1.9.5): Expanded examples in the chapter on Mininet (*18 Mininet*)

July 27, 2017 (ver 1.9.4): Clarified the role of `priority` in *2.7.1 OpenFlow Switches*.

July 22, 2017 (ver 1.9.3): Significant revisions to *2.7.2 Learning Switches in OpenFlow*; other minor changes.

July 11, 2017 (ver 1.9.2): A section in the Mininet chapter on the Pox controller (*18.9 OpenFlow and the POX Controller*).

July 7, 2017 (ver 1.9.1): A new (partially completed) chapter on Mininet (*18 Mininet*).

Jan 27, 2017 (ver 1.9.0): New sections on SEND (*8.6.4 Security and Neighbor Discovery*), DCCP (*11.1.2 DCCP*), TLS programming (*22.10.3 A TLS Programming Example*), bufferbloat (*13.7.1 Bufferbloat*), CoDel (*14.8.5 CoDel*) and BBR TCP (*15.16 TCP BBR*). There is also added content in *3.7.5 Wi-Fi Security*.

Oct 31, 2016 (ver 1.8.27): Introduction to TFTP (*11.2 Trivial File Transport Protocol, TFTP*) moved before *11.3 Fundamental Transport Issues*; a minor adjustment to the TCP state diagram (*12.7 TCP state diagram*) for transitions out of FIN_WAIT_1.

Oct 3, 2016 (ver 1.8.26): minor changes to the text, but a significant and hopefully useful change to the way the table-of-contents sidebar works: expanding the sidebar puts the table of contents in the current

viewport, and clicking on a contents link then collapses the sidebar.

Sept 27, 2016 (ver 1.8.25): several new exercises and other updates

Sept 16, 2016 (ver 1.8.24): Improved integration between the exercise sections and *24 Selected Solutions*; additional corrections and clarifications to *3.7.1 Wi-Fi and Collisions*, *2.4.1 Ethernet Learning Algorithm*, and other places.

Aug 9, 2016 (ver 1.8.23): More corrections and clarifications to SNMP (*21 Network Management and SNMP*); more on IPv6 extension headers (*8.5 IPv6 Extension Headers*) and OpenFlow (*2.7.1 OpenFlow Switches*).

Jul 21, 2016 (ver 1.8.22): Corrections and clarifications to SNMP (*21 Network Management and SNMP*)

Jun 14, 2016 (ver 1.8.21): Added material on *2.7 Software-Defined Networking* and *1.5.1 Traffic Engineering* (also scattered about in other sections), and updates to *22 Security*.

Apr 4, 2016 (ver 1.8.20): clarifications and new diagrams in *9 Routing-Update Algorithms*.

Mar 30, 2016 (ver 1.8.19): miscellaneous updates, including to TCP RST processing in *12.3 TCP Connection Establishment*.

Feb 29, 2016 (ver 1.8.18): Revisions to *8 IP version 6*, including updates related to *8.2.1 Interface identifiers*. All line drawings are now in .svg format.

Jan 20, 2016 (ver 1.8.17): New section on IPsec (*22.11 IPsec*), and other updates. Some diagrams are now in .svg format (*eg* in chapters *13 TCP Reno and Congestion Management* and *22 Security*). The .pdf version also takes advantage of the .svg format.

Dec 31, 2015 (ver 1.8.16): Updates to *10.6.7 BGP Relationships*, *3.7.5 Wi-Fi Security*, *6.2 Sliding Windows*, and other miscellaneous changes.

Dec 3, 2015 (ver 1.8.15): Technical corrections to some exercises.

Nov 24, 2015 (ver 1.8.14): Multiple small updates and clarifications, including to several exercises.

Oct 18, 2015 (ver 1.8.13): Extensive revisions to *8 IP version 6*.

Oct 14, 2015 (ver 1.8.12): Minor fixes and clarifications of some exercises.

Oct 11, 2015 (ver 1.8.11): Solutions to a few of the exercises (those marked with ◇) are now provided, in *24 Selected Solutions*. Hopefully this section will continue to expand. There is a correction to *10.6.7.1 BGP No-Valley Theorem* and a reorganization of the exposition; there are also several minor changes to *8 IP version 6*.

Sep 9, 2015 (ver 1.8.10): Miscellaneous clarifications; a paragraph on classless routing in *1.10 IP - Internet Protocol*.

Aug 16, 2015 (ver 1.8.09): Sections on *4.2.3 Optical Transport Network* and *11.1.4 netcat*; miscellaneous updates.

Jul 23, 2015 (ver 1.8.08): Multiple changes to *2 Ethernet* and *3.7 Wi-Fi*; other changes as well.

Jun 16, 2015 (ver 1.8.07): Corrections to *6.3.4 Simple Packet-Based Sliding-Windows Implementation* and additions to *7.11 Internet Control Message Protocol* and *8.9 ICMPv6*.

May 29, 2015 (ver 1.8.06): Section on RSA factoring (*22.9.1.2 Factoring RSA Keys*), fixed typos in *1.6 Routing Loops* and *11.3 Fundamental Transport Issues*.

May 24, 2015 (ver 1.8.05): Added discussion of the Logjam attack (*22.8 Diffie-Hellman-Merkle Exchange*).

May 22, 2015 (ver 1.8.04): Several additions to the wireless sections, including MIMO antennas (*3.7.3 Multiple Spatial Streams*) and LTE (*3.8 WiMAX and LTE*). Wireless LANs are now moved to the end of the chapter *3 Other LANs*.

May 1, 2015 (ver 1.8.03): This book is now available via IPv6! See *8.11 Using IPv6 and IPv4 Together*.

Also, corrections to exactly-once semantics in *11.5 Remote Procedure Call (RPC)*.

Apr 26, 2015 (ver 1.8.02): Numerous corrections and clarifications, and new sections on *12.22.1 MPTCP* and *12.22.2 SCTP*.

Mar 19, 2015: Added unicode-safer version (above), to support reading on most Android devices.

Mar 14, 2015: Expanded and revised chapter *8 IP version 6*, now including tunnel-broker IPv6 connections.

Mar 3, 2015: New section on *7.8 DNS*.

Feb 23, 2015: certificate pinning, sidebar on Superfish in *22.10.2.1 Certificate Authorities*.

Feb 15, 2015: New material on IPv6, in particular *8.11 Using IPv6 and IPv4 Together*.

Jan, 2015: The chapter *22 Security* is largely finished.

Somewhere there might be a field of interest in which the order of presentation of topics is well agreed upon.

Computer networking is not it.

There are many interconnections in the field of networking, as in most technical fields, and it is difficult to find an order of presentation that does not involve endless "forward references" to future chapters; this is true even if – as is done here – a largely bottom-up ordering is followed. I have therefore taken here a different approach: this first chapter is a summary of the essentials – LANs, IP and TCP – across the board, and later chapters expand on the material here.

Local Area Networks, or **LANs**, are the "physical" networks that provide the connection between machines within, say, a home, school or corporation. LANs are, as the name says, "local"; it is the **IP**, or Internet Protocol, layer that provides an abstraction for connecting multiple LANs into, well, the Internet. Finally, **TCP** deals with transport and connections and actually sending user data.

This chapter also contains some important other material. The section on **datagram forwarding**, central to packet-based switching and routing, is essential. This chapter also discusses packets generally, congestion, and sliding windows, but those topics are revisited in later chapters. Firewalls and network address translation are also covered here and not elsewhere.

1.1 Layers

These three topics – LANs, IP and TCP – are often called **layers**; they constitute the Link layer, the Internetwork layer, and the Transport layer respectively. Together with the Application layer (the software you use), these form the "**four-layer model**" for networks. A layer, in this context, corresponds strongly to the idea of a programming interface or library, with the understanding that a given layer communicates directly only with the two layers immediately above and below it. An application hands off a chunk of data to the TCP library, which in turn makes calls to the IP library, which in turn calls the LAN layer for actual delivery. An application does *not* interact directly with the IP and LAN layers at all.

The LAN layer is in charge of actual delivery of packets, using LAN-layer-supplied addresses. It is often conceptually subdivided into the "physical layer" dealing with, *eg*, the analog electrical, optical or radio signaling mechanisms involved, and above that an abstracted "logical" LAN layer that describes all the digital – that is, non-analog – operations on packets; see *2.1.4 The LAN Layer*. The physical layer is generally of direct concern only to those designing LAN hardware; the kernel software interface to the LAN corresponds to the logical LAN layer.

Application
Transport
IP
Logical LAN
Physical LAN

This LAN physical/logical division gives us the Internet **five-layer model**. This is less a formal hierarchy as an *ad hoc* classification method. We will return to this below in *1.15 IETF and OSI*, where we will also introduce two more rather obscure layers that complete the **seven**-layer model.

1.2 Data Rate, Throughput and Bandwidth

Any one network connection – *eg* at the LAN layer – has a **data rate**: the rate at which bits are transmitted. In some LANs (*eg* Wi-Fi) the data rate can vary with time. **Throughput** refers to the overall effective transmission rate, taking into account things like transmission overhead, protocol inefficiencies and perhaps even competing traffic. It is generally measured at a higher network layer than the data rate.

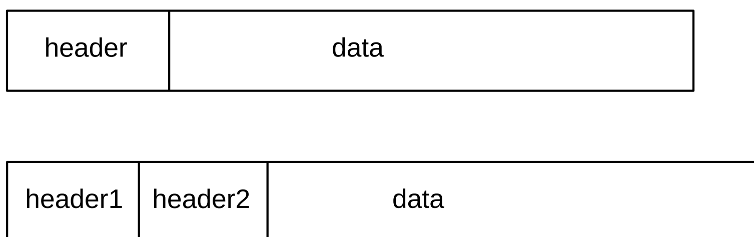
The term **bandwidth** can be used to refer to either of these, though we here use it mostly as a synonym for data rate. The term comes from radio transmission, where the width of the frequency band available is proportional, all else being equal, to the data rate that can be achieved.

In discussions about TCP, the term **goodput** is sometimes used to refer to what might also be called "application-layer throughput": the amount of usable data delivered to the receiving application. Specifically, retransmitted data is counted only once when calculating goodput but might be counted twice under some interpretations of "throughput".

Data rates are generally measured in kilobits per second (Kbps) or megabits per second (Mbps); the use of the lower-case "b" here denotes bits. In the context of data rates, a kilobit is 10^3 bits (not 2^{10}) and a megabit is 10^6 bits. Somewhat inconsistently, we follow the tradition of using KB and MB to denote data *volumes* of 2^{10} and 2^{20} bytes respectively, with the upper-case B denoting bytes. The newer abbreviations KiB and MiB would be more precise, but the consequences of confusion are modest.

1.3 Packets

Packets are modest-sized buffers of data, transmitted as a unit through some shared set of links. Of necessity, packets need to be prefixed with a **header** containing delivery information. In the common case known as **datagram forwarding**, the header contains a destination **address**; headers in networks using so-called **virtual-circuit** forwarding contain instead an identifier for the *connection*. Almost all networking today (and for the past 50 years) is packet-based, although we will later look briefly at some "circuit-switched" options for voice telephony.



Single and multiple headers

At the LAN layer, packets can be viewed as the imposition of a buffer (and addressing) structure on top of low-level serial lines; additional layers then impose additional structure. Informally, packets are often referred to as **frames** at the LAN layer, and as **segments** at the Transport layer.

The maximum packet size supported by a given LAN (*eg* Ethernet, Token Ring or ATM) is an intrinsic attribute of that LAN. Ethernet allows a maximum of 1500 bytes of data. By comparison, TCP/IP packets originally often held only 512 bytes of data, while early Token Ring packets could contain up to 4KB of data. While there are proponents of very large packet sizes, larger even than 64KB, at the other extreme the ATM (Asynchronous Transfer Mode) protocol uses 48 bytes of data per packet, and there are good reasons for believing in modest packet sizes.

One potential issue is how to forward packets from a large-packet LAN to (or through) a small-packet LAN; in later chapters we will look at how the IP (or Internet Protocol) layer addresses this.

Generally each layer adds its own header. Ethernet headers are typically 14 bytes, IP headers 20 bytes, and TCP headers 20 bytes. If a TCP connection sends 512 bytes of data per packet, then the headers amount to 10% of the total, a not-unreasonable overhead. For one common Voice-over-IP option, packets contain 160 bytes of data and 54 bytes of headers, making the header about 25% of the total. Compressing the 160 bytes of audio, however, may bring the data portion down to 20 bytes, meaning that the headers are now 73% of the total; see *20.11.4 RTP and VoIP*.

In datagram-forwarding networks the appropriate header will contain the address of the destination and perhaps other delivery information. Internal nodes of the network called **routers** or **switches** will then try to ensure that the packet is delivered to the requested destination.

The concept of packets and packet switching was first introduced by Paul Baran in 1962 (*[PB62]*). Baran's primary concern was with network survivability in the event of node failure; existing centrally switched protocols were vulnerable to central failure. In 1964, Donald Davies independently developed many of the same concepts; it was Davies who coined the term "packet".

It is perhaps worth noting that packets are buffers built of 8-bit *bytes*, and all hardware today agrees what a byte is (hardware agrees *by convention* on the order in which the bits of a byte are to be transmitted). 8-bit bytes are universal now, but it was not always so. Perhaps the last great non-byte-oriented hardware platform, which did indeed overlap with the Internet era broadly construed, was the DEC-10, which had a 36-bit word size; a word could hold five 7-bit ASCII characters. The early Internet specifications introduced the term **octet** (an 8-bit byte) and required that packets be sequences of octets; non-octet-oriented hosts had to be able to convert. Thus was chaos averted. Note that there are still byte-oriented data issues; as one example, binary integers can be represented as a sequence of bytes in either *big-endian* or *little-endian* byte order (*11.1.5 Binary Data*). **RFC 1700** specifies that Internet protocols use big-endian byte order, therefore sometimes called network byte order.

1.4 Datagram Forwarding

In the datagram-forwarding model of packet delivery, packet headers contain a destination address. It is up to the intervening switches or routers to look at this address and get the packet to the correct destination.

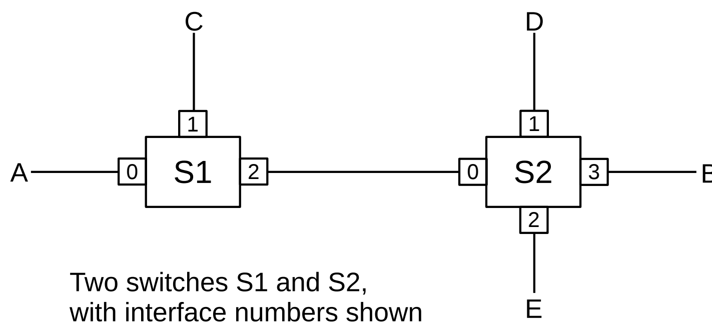
In datagram forwarding this is achieved by providing each switch with a **forwarding table** of ⟨destination,next_hop⟩ pairs. When a packet arrives, the switch looks up the destination address (presumed globally unique) in its forwarding table and finds the **next_hop** information: the immediate-neighbor address to which – or interface by which – the packet should be forwarded in order to bring it one step closer

to its final destination. The next_hop value in a forwarding table is a single entry; each switch is responsible for only one step in the packet's path. However, if all is well, the network of switches will be able to deliver the packet, one hop at a time, to its ultimate destination.

The "destination" entries in the forwarding table do not have to correspond exactly with the packet destination addresses, though in the examples here they do, and they do for Ethernet datagram forwarding. However, for IP routing, the table "destination" entries will correspond to **prefixes** of IP addresses; this leads to a huge savings in space. The fundamental requirement is that the switch can perform a lookup operation, using its forwarding table and the destination address in the arriving packet, to determine the next hop.

Just how the forwarding table is built is a question for later; we will return to this for Ethernet switches in *2.4.1 Ethernet Learning Algorithm* and for IP routers in *9 Routing-Update Algorithms*. For now, the forwarding tables may be thought of as created through initial configuration.

In the diagram below, switch S1 has interfaces 0, 1 and 2, and S2 has interfaces 0,1,2,3. If A is to send a packet to B, S1 must have a forwarding-table entry indicating that destination B is reached via its interface 2, and S2 must have an entry forwarding the packet out on interface 3.



Two switches S1 and S2, with interface numbers shown

A complete forwarding table for S1, using interface numbers in the `next_hop` column, would be:

S1	
destination	next_hop
A	0
C	1
B	2
D	2
E	2

The table for S2 might be as follows, where we have consolidated destinations A and C for visual simplicity.

S2	
destination	next_hop
A,C	0
D	1
E	2
B	3

In the network diagrammed above, all links are point-to-point, and so each interface corresponds to the unique immediate neighbor reached by that interface. We can thus replace the interface entries in the

`next_hop` column with the name of the corresponding **neighbor**. For human readers, using neighbors in the `next_hop` column is usually much more readable. S1's table can now be written as follows (with consolidation of the entries for B, D and E):

S1	
destination	next_hop
A	A
C	C
B,D,E	S2

A central feature of datagram forwarding is that each packet is forwarded "in isolation"; the switches involved do not have any awareness of any higher-layer logical connections established between endpoints. This is also called **stateless** forwarding, in that the forwarding tables have no per-connection state. **RFC 1122** put it this way (in the context of IP-layer datagram forwarding):

> To improve robustness of the communication system, gateways are designed to be stateless, forwarding each IP datagram independently of other datagrams. As a result, redundant paths can be exploited to provide robust service in spite of failures of intervening gateways and networks.

The fundamental alternative to datagram forwarding is **virtual circuits**, *3.4 Virtual Circuits*. In virtual-circuit networks, each router maintains state about each connection passing through it; different connections can be routed differently. If packet forwarding depends, for example, on per-connection information – *eg* both TCP port numbers – it is not datagram forwarding. (That said, it arguably still *is* datagram forwarding if web traffic – to TCP port 80 – is forwarded differently than all other traffic, because that rule does not depend on the specific connection.)

Datagram forwarding is sometimes allowed to use other information beyond the destination address. In theory, IP routing can be done based on the destination address and some **quality-of-service** information, allowing, for example, different routing to the same destination for high-bandwidth bulk traffic and for low-latency real-time traffic. In practice, most Internet Service Providers (ISPs) ignore user-provided quality-of-service information in the IP header, except by prearranged agreement, and route only based on the destination.

By convention, switching devices acting at the LAN layer and forwarding packets based on the LAN address are called **switches** (or, in earlier days, bridges), while such devices acting at the IP layer and forwarding on the IP address are called **routers**. Datagram forwarding is used both by Ethernet switches and by IP routers, though the destinations in Ethernet forwarding tables are individual nodes while the destinations in IP routers are entire *networks* (that is, sets of nodes).

In IP routers within end-user sites it is common for a forwarding table to include a catchall **default** entry, matching any IP address that is nonlocal and so needs to be routed out into the Internet at large. Unlike the consolidated entries for B, D and E in the table above for S1, which likely would have to be implemented as actual separate entries, a default entry is a single record representing where to forward the packet if no other destination match is found. Here is a forwarding table for S1, above, with a default entry replacing the last three entries:

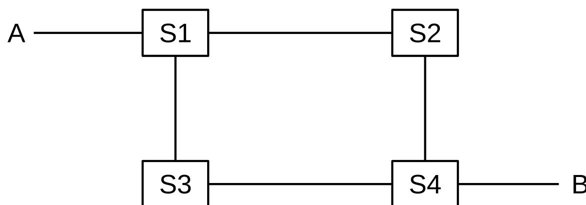
S1	
destination	next_hop
A	0
C	1
default	2

Default entries make sense only when we can tell by looking at an address that it does not represent a nearby node. This is common in IP networks because an IP address encodes the destination network, and routers generally know all the local networks. It is however rare in Ethernets, because there is generally no correlation between Ethernet addresses and locality. If S1 above were an Ethernet switch, and it had some means of knowing that interfaces 0 and 1 connected directly to individual hosts, not switches – and S1 knew the addresses of these hosts – then making interface 2 a default route would make sense. In practice, however, Ethernet switches do not know what kind of device connects to a given interface.

1.5 Topology

In the network diagrammed in the previous section, there are no loops; graph theorists might describe this by saying the network graph is **acyclic**, or is a **tree**. In a loop-free network there is a unique path between any pair of nodes. The forwarding-table algorithm has only to make sure that every destination appears in the forwarding tables; the issue of choosing between alternative paths does not arise.

However, if there are no loops then there is no **redundancy**: any broken link will result in partitioning the network into two pieces that cannot communicate. All else being equal (which it is not, but never mind for now), redundancy is a good thing. However, once we start including redundancy, we have to make decisions among the multiple paths to a destination. Consider, for a moment, the following network:



Should S1 list S2 or S3 as the next_hop to B? Both paths A–S1–**S2**–S4–B and A–S1–**S3**–S4–B get there. There is no right answer. Even if one path is "faster" than the other, taking the slower path is not exactly wrong (especially if the slower path is, say, less expensive). Some sort of protocol must exist to provide a mechanism by which S1 can make the choice (though this mechanism might be as simple as choosing to route via the first path discovered to the given destination). We also want protocols to make sure that, if S1 reaches B via S2 and the S2–S4 link fails, then S1 will switch over to the still-working S1–S3–S4–B route.

As we shall see, many LANs (in particular Ethernet) prefer "tree" networks with no redundancy, while IP has complex protocols in support of redundancy (*9 Routing-Update Algorithms*).

1.5.1 Traffic Engineering

In some cases the decision above between routes A–S1–S2–S4–B and A–S1–S3–S4–B might be of material significance – perhaps the S2–S4 link is slower than the others, or is more congested. We will use the term **traffic engineering** to refer to any intentional selection of one route over another, or any elevation of the priority of one class of traffic. The route selection can either be directly intentional, through configuration, or can be implicit in the selection or tuning of algorithms that then make these route-selection choices automatically. As an example of the latter, the algorithms of *9.1 Distance-Vector Routing-Update Algorithm* build forwarding tables on their own, but those tables are greatly influenced by the administrative assignment of link costs.

With pure datagram forwarding, used at either the LAN or the IP layer, the path taken by a packet is determined solely by its destination, and traffic engineering is limited to the choices made between alternative paths. We have already, however, suggested that datagram forwarding can be extended to take quality-of-service information into account; this may be used to have voice traffic – with its relatively low bandwidth but intolerance for delay – take an entirely different path than bulk file transfers. Alternatively, the network manager may simply assign voice traffic a higher priority, so it does not have to wait in queues behind file-transfer traffic.

The quality-of-service information may be set by the end-user, in which case an ISP may wish to recognize it only for designated users, which in turn means that the ISP will implicitly use the traffic source when making routing decisions. Alternatively, the quality-of-service information may be set by the ISP itself, based on its best guess as to the application; this means that the ISP may be using packet size, port number (*1.12 Transport*) and other contents as part of the routing decision. For some explicit mechanisms supporting this kind of routing, see *9.6 Routing on Other Attributes*.

At the LAN layer, traffic-engineering mechanisms are historically limited, though see *2.7 Software-Defined Networking*. At the IP layer, more strategies are available; see *20 Quality of Service*.

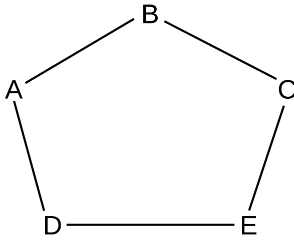
1.6 Routing Loops

A potential drawback to datagram forwarding is the possibility of a **routing loop**: a set of entries in the forwarding tables that cause some packets to circulate endlessly. For example, in the previous picture we would have a routing loop if, for (nonexistent) destination C, S1 forwarded to S2, S2 forwarded to S4, S4 forwarded to S3, and S3 forwarded to S1. A packet sent to C would not only not be delivered, but in circling endlessly it might easily consume a large majority of the bandwidth. Routing loops typically arise because the creation of the forwarding tables is often "distributed", and there is no global authority to detect inconsistencies. Even when there is such an authority, temporary routing loops can be created due to notification delays.

Routing loops can also occur in networks where the underlying link topology is loop-free; for example, in the previous diagram we could, again for destination C, have S1 forward to S2 and S2 forward back to S1. We will refer to such a case as a **linear** routing loop.

All datagram-forwarding protocols need some way of detecting and avoiding routing loops. Ethernet, for example, avoids nonlinear routing loops by disallowing loops in the underlying network topology, and avoids linear routing loops by not having switches forward a packet back out the interface by which it arrived. IP provides for a one-byte "Time to Live" (TTL) field in the IP header; it is set by the sender and decremented by 1 at each router; a packet is discarded if its TTL reaches 0. This limits the number of times a wayward packet can be forwarded to the initial TTL value, typically 64.

In datagram routing, a switch is responsible only for the next hop to the ultimate destination; if a switch has a complete path in mind, there is no guarantee that the next_hop switch or any other downstream switch will continue to forward along that path. Misunderstandings can potentially lead to routing loops. Consider this network:



D might feel that the best path to B is D–E–C–B (perhaps because it believes the A–D link is to be avoided). If E similarly decides the best path to B is E–D–A–B, and if D and E both choose their next_hop for B based on these best paths, then a linear routing loop is formed: D routes to B via E and E routes to B via D. Although each of D and E have identified a usable *path*, that path is not in fact followed. Moral: successful datagram routing requires cooperation and a consistent view of the network.

1.7 Congestion

Switches introduce the possibility of congestion: packets arriving faster than they can be sent out. This can happen with just two interfaces, if the inbound interface has a higher bandwidth than the outbound interface; another common source of congestion is traffic arriving on multiple inputs and all destined for the same output.

Whatever the reason, if packets are arriving for a given outbound interface faster than they can be sent, a queue will form for that interface. Once that queue is full, packets will be **dropped**. The most common strategy (though not the only one) is to drop any packets that arrive when the queue is full.

The term "congestion" may refer either to the point where the queue is just beginning to build up, or to the point where the queue is full and packets are lost. In their paper *[CJ89]*, Chiu and Jain refer to the first point as the **knee**; this is where the slope of the load vs throughput graph flattens. They refer to the second point as the **cliff**; this is where packet losses may lead to a precipitous decline in throughput. Other authors use the term **contention** for knee-congestion.

In the Internet, most packet losses are due to congestion. This is not because congestion is especially bad (though it can be, at times), but rather that other types of losses (*eg* due to packet corruption) are insignificant by comparison.

When to Upgrade?

Deciding when a network really *does* have insufficient bandwidth is not a technical issue but an economic one. The number of customers may increase, the cost of bandwidth may decrease or customers may simply be willing to pay more to have data transfers complete in less time; "customers" here can be external or in-house. Monitoring of links and routers for congestion can, however, help determine exactly what *parts* of the network would most benefit from upgrade.

We emphasize that the presence of congestion does *not* mean that a network has a shortage of bandwidth. Bulk-traffic senders (though not real-time senders) attempt to send as fast as possible, and congestion is simply the network's **feedback** that the maximum transmission rate has been reached. For further discussion, including alternative definitions of longer-term congestion, see *[BCL09]*.

Congestion *is* a sign of a problem in real-time networks, which we will consider in *20 Quality of Service*. In these networks losses due to congestion must generally be kept to an absolute minimum; one way to achieve this is to limit the acceptance of new connections unless sufficient resources are available.

1.8 Packets Again

Perhaps the core justification for packets, Baran's concerns about node failure notwithstanding, is that the same link can carry, at different times, different packets representing traffic to different destinations and from different senders. Thus, packets are the key to supporting **shared transmission lines**; that is, they support the **multiplexing** of multiple communications channels over a single cable. The alternative of a separate physical line between every pair of machines grows prohibitively complex very quickly (though **virtual circuits** between every pair of machines in a datacenter are not uncommon; see *3.4 Virtual Circuits*).

From this shared-medium perspective, an important packet feature is the maximum packet size, as this represents the maximum time a sender can send before other senders get a chance. The alternative of unbounded packet sizes would lead to prolonged network unavailability for everyone else if someone downloaded a large file in a single 1 Gigabit packet. Another drawback to large packets is that, if the packet is corrupted, the entire packet must be retransmitted; see *5.3.1 Error Rates and Packet Size*.

When a router or switch receives a packet, it (generally) reads in the entire packet before looking at the header to decide to what next node to forward it. This is known as **store-and-forward**, and introduces a **forwarding delay** equal to the time needed to read in the entire packet. For individual packets this forwarding delay is hard to avoid (though some switches do implement **cut-through** switching to begin forwarding a packet before it has fully arrived), but if one is sending a long train of packets then by keeping multiple packets *en route* at the same time one can essentially eliminate the significance of the forwarding delay; see *5.3 Packet Size*.

Total packet delay from sender to receiver is the sum of the following:

- **Bandwidth delay**, *ie* sending 1000 Bytes at 20 Bytes/millisecond will take 50 ms. This is a per-link delay.
- **Propagation delay** due to the speed of light. For example, if you start sending a packet right now on a 5000-km cable across the US with a propagation speed of 200 m/µsec (= 200 km/ms, about 2/3 the speed of light in vacuum), the first bit will not arrive at the destination until 25 ms later. The bandwidth delay then determines how much after that the entire packet will take to arrive.
- **Store-and-forward delay**, equal to the sum of the bandwidth delays out of each router along the path
- **Queuing delay**, or waiting in line at busy routers. At bad moments this can exceed 1 sec, though that is rare. Generally it is less than 10 ms and often is less than 1 ms. Queuing delay is the only delay component amenable to reduction through careful engineering.

See *5.1 Packet Delay* for more details.

1.9 LANs and Ethernet

A **local-area network**, or LAN, is a system consisting of

- physical links that are, ultimately, serial lines
- common interfacing hardware connecting the hosts to the links
- protocols to make everything work together

We will explicitly assume that every LAN node is able to communicate with every other LAN node. Sometimes this will require the cooperation of intermediate nodes acting as switches.

Far and away the most common type of LAN is Ethernet, originally described in a 1976 paper by Metcalfe and Boggs *[MB76]*. Ethernet's popularity is due to low cost more than anything else, though the primary reason Ethernet cost is low is that high demand has led to manufacturing economies of scale.

The original Ethernet had a bandwidth of 10 Mbps (megabits per second; we will use lower-case "b" for bits and upper-case "B" for bytes), though nowadays most Ethernet operates at 100 Mbps and gigabit (1000 Mbps) Ethernet (and faster) is widely used in server rooms. (By comparison, as of this writing (2015) the data transfer rate to a typical faster hard disk is about 1000 Mbps.) Wireless ("Wi-Fi") LANs are gaining popularity, and in many settings have supplanted wired Ethernet to end-users.

Many early Ethernet installations were unswitched; each host simply tapped in to one long primary cable that wound through the building (or floor). In principle, two stations could then transmit at the same time, rendering the data unintelligible; this was called a **collision**. Ethernet has several design features intended to minimize the bandwidth wasted on collisions: stations, before transmitting, check to be sure the line is idle, they monitor the line *while* transmitting to detect collisions during the transmission, and, if a collision is detected, they execute a random backoff strategy to avoid an immediate recollision. See *2.1.5 The Slot Time and Collisions*. While Ethernet collisions definitely reduce throughput, in the larger view they should perhaps be thought of as a part of a remarkably inexpensive shared-access mediation protocol.

In unswitched Ethernets every packet is received by every host and it is up to the network card in each host to determine if the arriving packet is addressed to that host. It is almost always possible to configure the card to forward *all* arriving packets to the attached host; this poses a security threat and "password sniffers" that surreptitiously collected passwords via such eavesdropping used to be common.

Password Sniffing

In the fall of 1994 at Loyola University I remotely changed the root password on several CS-department unix machines at the other end of campus, using telnet. I told no one. Within two hours, someone else logged into one of these machines, using the new password, from a host in Europe. Password sniffing was the likely culprit.
Two months later was the so-called "Christmas Day Attack" (*12.10.1 ISNs and spoofing*). One of the hosts used to launch this attack was Loyola's hacked apollo.it.luc.edu. It is unclear the degree to which password sniffing played a role in that exploit.

Due to both privacy and efficiency concerns, almost all Ethernets today are fully switched; this ensures that each packet is delivered only to the host to which it is addressed. One advantage of switching is that it effectively eliminates most Ethernet collisions; while in principle it replaces them with a **queuing** issue, in practice Ethernet switch queues so seldom fill up that they are almost invisible even to network managers (unlike IP router queues). Switching also prevents host-based eavesdropping, though arguably a better solution to this problem is encryption. Perhaps the more significant tradeoff with switches, historically, was that Once Upon A Time they were expensive and unreliable; tapping directly into a common cable was dirt cheap.

Ethernet addresses are six bytes long. Each Ethernet card (or **network interface**) is assigned a (supposedly) unique address at the time of manufacture; this address is burned into the card's ROM and is called the card's **physical** address or **hardware** address or **MAC** (Media Access Control) address. The first three bytes of the physical address have been assigned to the manufacturer; the subsequent three bytes are a serial number assigned by that manufacturer.

By comparison, IP addresses are assigned administratively by the local site. The basic advantage of having addresses in hardware is that hosts automatically know their own addresses on startup; no manual configuration or server query is necessary. It is not unusual for a site to have a large number of identically configured workstations, for which all network differences derive ultimately from each workstation's unique Ethernet address.

The network interface continually monitors all arriving packets; if it sees any packet containing a destination address that matches its own physical address, it grabs the packet and forwards it to the attached CPU (via a CPU interrupt).

Ethernet also has a designated **broadcast address**. A host sending to the broadcast address has its packet received by every other host on the network; if a switch receives a broadcast packet on one port, it forwards the packet out every other port. This broadcast mechanism allows host A to contact host B when A does not yet know B's physical address; typical broadcast queries have forms such as "Will the designated server please answer" or (from the ARP protocol) "will the host with the given IP address please tell me your physical address".

Traffic addressed to a particular host – that is, not broadcast – is said to be **unicast**.

Because Ethernet addresses are assigned by the hardware, knowing an address does not provide any direct indication of where that address is located on the network. In switched Ethernet, the switches must thus have a forwarding-table record for each individual Ethernet address on the network; for extremely large networks this ultimately becomes unwieldy. Consider the analogous situation with postal addresses: Ethernet is somewhat like attempting to deliver mail using social-security numbers as addresses, where each postal worker is provided with a large catalog listing each person's SSN together with their physical location. Real postal mail is, of course, addressed "hierarchically" using ever-more-precise specifiers: state, city, zipcode, street address, and name / room#. Ethernet, in other words, does not scale well to "large" sizes.

Switched Ethernet works quite well, however, for networks with up to 10,000-100,000 nodes. Forwarding tables with size in that range are straightforward to manage.

To forward packets correctly, switches must know where all active destination addresses in the LAN are located; traditional Ethernet switches do this by a passive **learning** algorithm. (IP routers, by comparison, use "active" protocols, and some newer Ethernet switches take the approach of *2.7 Software-Defined Networking*.) Typically a host physical address is entered into a switch's forwarding table when a packet from that host is first *received*; the switch notes the packet's arrival interface and *source* address and assumes that the same interface is to be used to deliver packets back to that sender. If a given destination address has not yet been seen, and thus is not in the forwarding table, Ethernet switches still have the backup delivery option of **flooding**: forwarding the packet to everyone by treating the destination address like the broadcast address, and allowing the host Ethernet cards to sort it out. Since this broadcast-like process is not generally used for more than one packet (after that, the switches will have learned the correct forwarding-table entries), the risks of excessive traffic and of eavesdropping are minimal.

The ⟨host,interface⟩ forwarding table is often easier to think of as ⟨host,next_hop⟩, where the next_hop node is whatever switch or host is at the immediate other end of the link connecting to the given interface. In a fully switched network where each link connects only two interfaces, the two perspectives are equivalent.

1.10 IP - Internet Protocol

To solve the scaling problem with Ethernet, and to allow support for other types of LANs and point-to-point links as well, the **Internet Protocol** was developed. Perhaps the central issue in the design of IP was to support universal connectivity (everyone can connect to everyone else) in such a way as to allow scaling to enormous size (in 2013 there appear to be around ~10^9 nodes, although IP should work to 10^{10} nodes or more), without resulting in unmanageably large forwarding tables (currently the largest tables have about 300,000 entries.)

In the early days, IP networks were considered to be "internetworks" of basic networks (LANs); nowadays users generally ignore LANs and think of the Internet as one large (virtual) network.

To support universal connectivity, IP provides a global mechanism for **addressing and routing**, so that packets can actually be delivered from any host to any other host. IP addresses (for the most-common version 4, which we denote **IPv4**) are 4 bytes (32 bits), and are part of the **IP header** that generally follows the Ethernet header. The Ethernet header only stays with a packet for one hop; the IP header stays with the packet for its entire journey across the Internet.

An essential feature of IPv4 (and IPv6) addresses is that they can be divided into a **network** part (a prefix) and a **host** part (the remainder). The "legacy" mechanism for designating the IPv4 network and host address portions was to make the division according to the first few bits:

first few bits	first byte	network bits	host bits	name	application
0	0-127	8	24	class A	a few very large networks
10	128-191	16	16	class B	institution-sized networks
110	192-223	24	8	class C	sized for smaller entities

For example, the original IP address allocation for Loyola University Chicago was 147.126.0.0, a class B. In binary, 147 is **10**010011.

IP addresses, unlike Ethernet addresses, are **administratively assigned**. Once upon a time, you would get your Class B network prefix from the Internet Assigned Numbers Authority, or IANA (they now delegate this task), and then you would in turn assign the host portion in a way that was appropriate for your local site. As a result of this administrative assignment, an IP address usually serves not just as an **endpoint identifier** but also as a **locator**, containing embedded location information (at least in the sense of location within the IP-address-assignment hierarchy, which may not be geographical). Ethernet addresses, by comparison, are endpoint identifiers but *not* locators.

The Class A/B/C definition above was spelled out in 1981 in **RFC 791**, which introduced IP. Class D was added in 1986 by **RFC 988**; class D addresses must begin with the bits 1110. These addresses are for **multicast**, that is, sending an IP packet to every member of a set of recipients (ideally without actually transmitting it more than once on any one link).

Nowadays the division into the network and host bits is dynamic, and can be made at different positions in the address at different levels of the network. For example, a small organization might receive a /27 address block (1/8 the size of a class-C /24) from its ISP, *eg* 200.1.130.96/**27**. The ISP routes to the organization based on this /27 prefix. At some higher level, however, routing might be based on the prefix 200.1.128/**18**; this might, for example, represent an address block assigned to the ISP (note that the first 18 bits of 200.1.130.x match 200.1.128; the first two bits of 128 and 130, taken as 8-bit quantities, are "10"). The network/host division point is *not* carried within the IP header; routers negotiate this division point

when they negotiate the next_hop forwarding information. We will return to this in *7.5 The Classless IP Delivery Algorithm*.

The network portion of an IP address is sometimes called the **network number** or **network address** or **network prefix**. As we shall see below, most forwarding decisions are made using only the network prefix. The network prefix is commonly denoted by setting the host bits to zero and ending the resultant address with a slash followed by the number of network bits in the address: *eg* 12.0.0.0/8 or 147.126.0.0/16. Note that 12.0.0.0/8 and 12.0.0.0/9 represent different things; in the latter, the second byte of any host address extending the network address is constrained to begin with a 0-bit. An anonymous block of IP addresses might be referred to only by the slash and following digit, *eg* "we need a /22 block to accommodate all our customers".

All hosts with the same network address (same network bits) are said to be on the same **IP network** and *must be located together on the same LAN*; as we shall see below, if two hosts share the same network address then they will assume they can reach each other directly via the underlying LAN, and if they cannot then connectivity fails. A consequence of this rule is that outside of the site *only the network bits need to be looked at to route a packet to the site*.

Usually, all hosts (or more precisely all network interfaces) on the same physical LAN share the same network prefix and thus are part of the same IP network. Occasionally, however, one LAN is divided into multiple IP networks.

Each individual LAN technology has a **maximum packet size** it supports; for example, Ethernet has a maximum packet size of about 1500 bytes but the once-competing Token Ring had a maximum of 4 KB. Today the world has largely standardized on Ethernet and almost entirely standardized on Ethernet packet-size limits, but this was not the case when IP was introduced and there was real concern that two hosts on separate large-packet networks might try to exchange packets too large for some small-packet intermediate network to carry.

Therefore, in addition to routing and addressing, the decision was made that IP must also support **fragmentation**: the division of large packets into multiple smaller ones (in other contexts this may also be called **segmentation**). The IP approach is not very efficient, and IP hosts go to considerable lengths to avoid fragmentation. IP does require that packets of up to 576 bytes be supported, and so a common legacy strategy was for a host to limit a packet to at most 512 user-data bytes whenever the packet was to be sent via a router; packets addressed to another host on the same LAN could of course use a larger packet size. Despite its limited use, however, fragmentation is essential conceptually, in order for IP to be able to support large packets without knowing anything about the intervening networks.

IP is a **best effort** system; there are no IP-layer acknowledgments or retransmissions. We ship the packet off, and hope it gets there. Most of the time, it does.

Architecturally, this best-effort model represents what is known as **connectionless** networking: the IP layer does not maintain information about endpoint-to-endpoint connections, and simply forwards packets like a giant LAN. Responsibility for creating and maintaining connections is left for the next layer up, the TCP layer. Connectionless networking is *not* the only way to do things: the alternative could have been some form **connection-oriented** internetworking, in which routers *do* maintain state information about individual connections. Later, in *3.4 Virtual Circuits*, we will examine how virtual-circuit networking can be used to implement a connection-oriented approach; virtual-circuit switching is the primary alternative to datagram switching.

Connectionless (IP-style) and connection-oriented networking each have advantages. Connectionless networking is conceptually more reliable: if routers do not hold connection state, then they cannot *lose* con-

nection state. The path taken by the packets in some higher-level connection can easily be dynamically rerouted. Finally, connectionless networking makes it hard for providers to bill by the connection; once upon a time (in the era of dollar-a-minute phone calls) this was a source of mild astonishment to many new users. (This was not always a given; the paper *[CK74]* considers, among other things, the possibility of per-packet accounting.)

The primary advantage of connection-oriented networking, on the other hand, is that the routers are then much better positioned to accept **reservations** and to make **quality-of-service guarantees**. This remains something of a sore point in the current Internet: if you want to use Voice-over-IP, or **VoIP**, telephones, or if you want to engage in video conferencing, your packets will be treated by the Internet core just the same as if they were low-priority file transfers. There is no "priority service" option.

The most common form of IP packet loss is router queue overflows, representing network congestion. Packet losses due to packet corruption are rare (*eg* less than one in 10^4; perhaps much less). But in a connectionless world a large number of hosts can simultaneously attempt to send traffic through one router, in which case queue overflows are hard to avoid.

Although we will often assume, for simplicity, that routers have a fixed input queue size, the reality is often a little more complicated. See *14.8 Active Queue Management* and *19 Queuing and Scheduling*.

1.10.1 IP Forwarding

IP routers use datagram forwarding, described in *1.4 Datagram Forwarding* above, to deliver packets, but the "destination" values listed in the forwarding tables are network prefixes – representing entire LANs – instead of individual hosts. The goal of IP forwarding, then, becomes delivery to the correct LAN; a separate process is used to deliver to the final host once the final LAN has been reached.

The entire point, in fact, of having a network/host division within IP addresses is so that **routers need to list only the network prefixes** of the destination addresses in their IP forwarding tables. This strategy is *the* key to IP scalability: it saves large amounts of forwarding-table space, it saves time as smaller tables allow faster lookup, and it saves the bandwidth and overhead that would be needed for routers to keep track of individual addresses. To get an idea of the forwarding-table space savings, there are currently (2013) around a billion hosts on the Internet, but only 300,000 or so networks listed in top-level forwarding tables.

With IP's use of network prefixes as forwarding-table destinations, matching an actual packet address to a forwarding-table entry is no longer a matter of simple equality comparison; routers must compare appropriate prefixes.

IP forwarding tables are sometimes also referred to as "routing tables"; in this book, however, we make at least a token effort to use "forwarding" to refer to the packet forwarding process, and "routing" to refer to mechanisms by which the forwarding tables are maintained and updated. (If we were to be completely consistent here, we would use the term "forwarding loop" rather than "routing loop".)

Now let us look at an example of how IP forwarding (or routing) works. We will assume that all network nodes are either **hosts** – user machines, with a single network connection – or **routers**, which do packet-forwarding only. Routers are not directly visible to users, and always have at least two different network interfaces representing different networks that the router is connecting. (Machines can be both hosts and routers, but this introduces complications.)

Suppose A is the sending host, sending a packet to a destination host D. The IP header of the packet will contain D's IP address in the "destination address" field (it will also contain A's own address as the "source

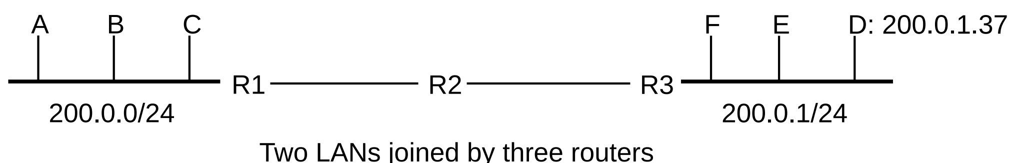
address"). The first step is for A to determine whether D is on the same LAN as itself or not; that is, whether D is **local**. This is done by looking at the network part of the destination address, which we will denote by D_{net}. If this net address is the same as A's (that is, if it is equal numerically to A_{net}), then A figures D is on the same LAN as itself, and can use direct LAN delivery. It looks up the appropriate physical address for D (probably with the **ARP** protocol, *7.9 Address Resolution Protocol: ARP*), attaches a LAN header to the packet in front of the IP header, and sends the packet straight to D via the LAN.

If, however, A_{net} and D_{net} do *not* match – D is **non-local** – then A looks up a router to use. Most ordinary hosts use only one router for all non-local packet deliveries, making this choice very simple. A then forwards the packet to the router, again using direct delivery over the LAN. The IP destination address in the packet remains D in this case, although the LAN destination address will be that of the router.

When the router receives the packet, it strips off the LAN header but leaves the IP header with the IP destination address. It extracts the destination D, and then looks at D_{net}. The router first checks to see if any of *its* network interfaces are on the same LAN as D; recall that the router connects to at least one additional network besides the one for A. If the answer is yes, then the router uses direct LAN delivery to the destination, as above. If, on the other hand, D_{net} is not a LAN to which the router is connected directly, then the router consults its internal forwarding table. This consists of a list of networks each with an associated next_hop address. These ⟨net,next_hop⟩ tables compare with switched-Ethernet's ⟨host,next_hop⟩ tables; the former type will be smaller because there are many fewer nets than hosts. The next_hop addresses in the table are chosen so that the router can always reach them via direct LAN delivery via one of its interfaces; generally they are other routers. The router looks up D_{net} in the table, finds the next_hop address, and uses direct LAN delivery to get the packet to that next_hop machine. The packet's IP header remains essentially unchanged, although the router most likely attaches an entirely new LAN header.

The packet continues being forwarded like this, from router to router, until it finally arrives at a router that is connected to D_{net}; it is then delivered by that final router directly to D, using the LAN.

To make this concrete, consider the following diagram:



Two LANs joined by three routers

With Ethernet-style forwarding, R2 would have to maintain entries for each of A,B,C,D,E,F. With IP forwarding, R2 has just two entries to maintain in its forwarding table: 200.0.0/24 and 200.0.1/24. If A sends to D, at 200.0.1.37, it puts this address into the IP header, notes that 200.0.0 ≠ 200.0.1, and thus concludes D is not a local delivery. A therefore sends the packet to its router R1, using LAN delivery. R1 looks up the destination network 200.0.1 in its forwarding table and forwards the packet to R2, which in turn forwards it to R3. R3 now sees that it *is* connected directly to the destination network 200.0.1, and delivers the packet via the LAN to D, by looking up D's physical address.

In this diagram, IP addresses for the ends of the R1–R2 and R2–R3 links are not shown. They could be assigned global IP addresses, but they could also use "private" IP addresses. Assuming these links are point-to-point links, they might not actually need IP addresses at all; we return to this in *7.12 Unnumbered Interfaces*.

One can think of the network-prefix bits as analogous to the "zip code" on postal mail, and the host bits as

analogous to the street address. The internal parts of the post office get a letter to the right zip code, and then an individual letter carrier (the LAN) gets it to the right address. Alternatively, one can think of the network bits as like the area code of a phone number, and the host bits as like the rest of the digits. Newer protocols that support different net/host division points at different places in the network – sometimes called **hierarchical routing** – allow support for addressing schemes that correspond to, say, zip/street/user, or areacode/exchange/subscriber.

The Invertebrate Internet

The backbone is not as essential as it once was. Once Upon A Time, all traffic between different providers passed through the backbone. The legacy backbone still exists, but today it is also common for traffic from large providers such as Google to take a backbone-free path; such providers connect (or "peer") directly with large residential ISPs such as Comcast. Google refers to this as their "Edge Network"; see peering.google.com and also *10.6.6.1 MED values and traffic engineering*.

We will refer to the Internet **backbone** as those IP routers that specialize in large-scale routing on the commercial Internet, and which generally have forwarding-table entries covering all public IP addresses; note that this is essentially a business definition rather than a technical one. We can revise the table-size claim of the previous paragraph to state that, while there are many *private* IP networks, there are about 300,000 visible to the backbone. A forwarding table of 300,000 entries is quite feasible; a table a hundred times larger is not, let alone a thousand times larger.

IP routers at non-backbone sites generally know all locally assigned network prefixes, *eg* 200.0.0/24 and 200.0.1/24 above. If a destination does not match any locally assigned network prefix, the packet needs to be routed out into the Internet at large; for typical non-backbone sites this almost always this means the packet is sent to the ISP that provides Internet connectivity. Generally the local routers will contain a catchall **default** entry covering all nonlocal networks; this means that the router needs an explicit entry only for locally assigned networks. This greatly reduces the forwarding-table size. The Internet backbone can be approximately described, in fact, as those routers that do *not* have a default entry.

For most purposes, the Internet can be seen as a combination of end-user LANs together with point-to-point links joining these LANs to the backbone, point-to-point links also tie the backbone together. Both LANs and point-to-point links appear in the diagram above.

Just how routers build their ⟨destnet,next_hop⟩ forwarding tables is a major topic itself, which we cover in *9 Routing-Update Algorithms*. Unlike Ethernet, IP routers do *not* have a "flooding" delivery mechanism as a fallback, so the tables must be constructed in advance. (There is a limited form of IP broadcast, but it is basically intended for reaching the local LAN only, and does not help at all with delivery in the event that the destination network is unknown.)

Most forwarding-table-construction algorithms used on a set of routers under common management fall into either the **distance-vector** or the **link-state** category; these are described in *9 Routing-Update Algorithms*. Routers *not* under common management – that is, neighboring routers belonging to different organizations – exchange information through the Border Gateway Protocol, BGP (*10 Large-Scale IP Routing*). BGP allows routing decisions to be based on a fusion of "technical" information (which sites are reachable at all, and through where) together with "policy" information representing legal or commercial agreements: which outside routers are "preferred", whose traffic an ISP will carry even if it isn't to one of the ISP's customers, *etc*.

Most common residential "routers" involve **network address translation** in addition to packet forwarding.

See *7.7 Network Address Translation*.

1.10.2 The Future of IPv4

As mentioned earlier, allocation of blocks of IP addresses is the responsibility of the Internet Assigned Numbers Authority. IANA long ago delegated the job of allocating network prefixes to individual sites; they limited themselves to handing out /8 blocks (class A blocks) to the five **regional registries**, which are

- ARIN – North America
- RIPE – Europe, the Middle East and parts of Asia
- APNIC – East Asia and the Pacific
- AfriNIC – most of Africa
- LACNIC – Central and South America

As of the end of January 2011, the IANA finally ran out of /8 blocks. There is a table at http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml of all IANA assignments of /8 blocks; examination of the table shows all have now been allocated.

In September 2015, ARIN ran out of its pool of IPv4 addresses. Most of ARIN's customers are ISPs, which can now obtain new IPv4 addresses only by buying unused address blocks from other organizations.

A few months after the IANA pool ran out, Microsoft purchased 666,624 IP addresses (2604 Class-C blocks) in a Nortel bankruptcy auction for $7.5 million. By a year later, IP-address prices appeared to have retreated only slightly. It is possible that the market for IPv4 address blocks will continue to develop; alternatively, this turn of events may accelerate implementation of IPv6, which has 128-bit addresses.

An IPv4 address price in the range of $10 is unlikely to have much impact in residential Internet access, where annual connection fees are often $600. Large organizations use NAT (*7.7 Network Address Translation*) extensively, leading to the need for only a small number of globally visible addresses. The IPv4 address shortage does not even seem to have affected wireless networking. It does, however, lead to inefficient routing tables, as a site that once had a single /20 address block – and thus a single backbone forwarding-table entry – might now be spread over more than a hundred /27 blocks and concomitant forwarding entries.

1.11 DNS

IP addresses are hard to remember (nearly impossible in IPv6). The **domain name system**, or DNS (*7.8 DNS*), comes to the rescue by creating a way to convert hierarchical text names to IP addresses. Thus, for example, one can type `www.luc.edu` instead of 147.126.1.230. Virtually all Internet software uses the same basic library calls to convert DNS names to actual addresses.

One thing DNS makes possible is changing a website's IP address while leaving the name alone. This allows moving a site to a new provider, for example, without requiring users to learn anything new. It is also possible to have several different DNS names resolve to the same IP address, and – through some modest trickery – have the http (web) server at that IP address handle the different DNS names as completely different websites.

DNS is hierarchical and distributed. In looking up `cs.luc.edu` four different DNS servers may be queried: for the so-called "DNS root zone", for `edu`, for `luc.edu` and for `cs.luc.edu`. Searching a hierarchy can be cumbersome, so DNS search results are normally cached locally. If a name is not found in the cache, the lookup may take a couple seconds. The DNS hierarchy need have nothing to do with the IP-address hierarchy.

1.12 Transport

The IP layer gets packets from one node to another, but it is not well-suited to transport. First, IP routing is a "best-effort" mechanism, which means packets can and do get lost sometimes. Additionally, data that does arrive can arrive out of order. Finally, IP only supports sending to a specific host; normally, one wants to send to a given application running on that host. Email and web traffic, or two different web sessions, should not be commingled!

The Transport layer is the layer above the IP layer that handles these sorts of issues, often by creating some sort of *connection* abstraction. Far and away the most popular mechanism in the Transport layer is the Transmission Control Protocol, or **TCP**. TCP extends IP with the following features:

- **reliability**: TCP numbers each packet, and keeps track of which are lost and retransmits them after a timeout. It holds early-arriving out-of-order packets for delivery at the correct time. Every arriving data packet is acknowledged by the receiver; timeout and retransmission occurs when an acknowledgment packet isn't received by the sender within a given time.
- **connection-orientation**: Once a TCP connection is made, an application sends data simply by writing to that connection. No further application-level addressing is needed. TCP connections are managed by the operating-system kernel, not by the application.
- **stream-orientation**: An application using TCP can write 1 byte at a time, or 100KB at a time; TCP will buffer and/or divide up the data into appropriate sized packets.
- **port numbers**: these provide a way to specify the receiving application for the data, and also to identify the sending application.
- **throughput management**: TCP attempts to maximize throughput, while at the same time not contributing unnecessarily to network **congestion**.

TCP endpoints are of the form ⟨host,port⟩; these pairs are known as **socket addresses**, or sometimes as just **sockets** though the latter refers more properly to the operating-system objects that receive the data sent to the socket addresses. **Servers** (or, more precisely, server applications) *listen* for connections to sockets they have opened; the **client** is then any endpoint that *initiates* a connection to a server.

When you enter a host name in a web browser, it opens a TCP connection to the server's port 80 (the standard web-traffic port), that is, to the server socket with socket-address ⟨server,80⟩. If you have several browser tabs open, each might connect to the *same* server socket, but the connections are distinguishable by virtue of using separate ports (and thus having separate socket addresses) on the *client* end (that is, your end).

A busy server may have thousands of connections to its port 80 (the web port) and hundreds of connections to port 25 (the email port). Web and email traffic are kept separate by virtue of the different ports used. All those clients to the same port, though, are kept separate because each comes from a unique ⟨host,port⟩ pair. A TCP connection is determined by the ⟨host,port⟩ socket address at *each* end; traffic on different connections does not intermingle. That is, there may be multiple independent connections to ⟨www.luc.edu,80⟩.

This is somewhat analogous to certain business telephone numbers of the "*operators are standing by*" type, which support multiple callers at the same time to the same toll-free number. Each call to that number is answered by a different operator (corresponding to a different cpu process), and different calls do not "overhear" each other.

TCP uses the **sliding-windows algorithm**, *6 Abstract Sliding Windows*, to keep multiple packets *en route* at any one time. The **window size** represents the number of packets simultaneously in transit (TCP actually keeps track of the window size in bytes, but packets are easier to visualize). If the window size is 10 packets, for example, then at any one time 10 packets are in transit (perhaps 5 data packets and 5 returning acknowledgments). Assuming no packets are lost, then as each acknowledgment arrives the window "slides forward" by one packet. The data packet 10 packets ahead is then sent, to maintain a total of 10 packets on the wire. For example, consider the moment when the ten packets 20-29 are in transit. When ACK[20] is received, the number of packets outstanding drops to 9 (packets 21-29). To keep 10 packets in flight, Data[30] is sent. When ACK[21] is received, Data[31] is sent, and so on.

Sliding windows minimizes the effect of store-and-forward delays, and propagation delays, as these then only count once for the entire windowful and not once per packet. Sliding windows also provides an automatic, if partial, brake on congestion: the queue at any switch or router along the way cannot exceed the window size. In this it compares favorably with **constant-rate** transmission, which, if the available bandwidth falls below the transmission rate, always leads to overflowing queues and to a significant percentage of dropped packets. Of course, if the window size is too large, a sliding-windows sender may also experience dropped packets.

The ideal window size, at least from a throughput perspective, is such that it takes one round-trip time to send an entire window, so that the next ACK will always be arriving just as the sender has finished transmitting the window. Determining this ideal size, however, is difficult; for one thing, the ideal size varies with network load. As a result, TCP approximates the ideal size. The most common TCP strategy – that of so-called TCP Reno – is that the window size is slowly raised until packet loss occurs, which TCP takes as a sign that it has reached the limit of available network resources. At that point the window size is reduced to half its previous value, and the slow climb resumes. The effect is a "sawtooth" graph of window size with time, which oscillates (more or less) around the "optimal" window size. For an idealized sawtooth graph, see *13.1.1 The Somewhat-Steady State*; for some "real" (simulation-created) sawtooth graphs see *16.4.1 Some TCP Reno cwnd graphs*.

While this window-size-optimization strategy has its roots in attempting to maximize the available bandwidth, it also has the effect of greatly limiting the number of packet-loss events. As a result, TCP has come to be the Internet protocol charged with reducing (or at least managing) **congestion** on the Internet, and – relatedly – with ensuring **fairness** of bandwidth allocations to competing connections. Core Internet routers – at least in the classical case – essentially have no role in enforcing congestion or fairness restrictions at all. The Internet, in other words, places responsibility for congestion avoidance cooperatively into the hands of end users. While "cheating" is possible, this cooperative approach has worked remarkably well.

While TCP is ubiquitous, the **real-time** performance of TCP is not always consistent: if a packet is lost, the receiving TCP host will not turn over anything further to the receiving application until the lost packet has been retransmitted successfully; this is often called **head-of-line blocking**. This is a serious problem for sound and video applications, which can discretely handle modest losses but which have much more difficulty with sudden large delays. A few lost packets ideally should mean just a few brief voice dropouts (pretty common on cell phones) or flicker/snow on the video screen (or just reuse of the previous frame); both of these are better than pausing completely.

The basic alternative to TCP is known as **UDP**, for User Datagram Protocol. UDP, like TCP, provides port

numbers to support delivery to multiple endpoints within the receiving host, in effect to a specific process on the host. As with TCP, a UDP socket consists of a ⟨host,port⟩ pair. UDP also includes, like TCP, a checksum over the data. However, UDP omits the other TCP features: there is no connection setup, no lost-packet detection, no automatic timeout/retransmission, and the application must manage its own packetization. This simplicity should not be seen as all negative: the absence of connection setup means data transmission can get started faster, and the absence of lost-packet detection means there is no head-of-line blocking. See *11 UDP Transport*.

The Real-time Transport Protocol, or **RTP**, sits above UDP and adds some additional support for voice and video applications.

1.12.1 Transport Communications Patterns

The two "classic" traffic patterns for Internet communication are these:

- Interactive or bursty communications such as via ssh or telnet, with long idle times between short bursts
- Bulk file transfers, such as downloading a web page

TCP handles both of these well, although its congestion-management features apply only when a large amount of data is in transit at once. Web browsing is something of a hybrid; over time, there is usually considerable burstiness, but individual pages now often exceed 1 MB.

To the above we might add **request/reply** operations, *eg* to query a database or to make DNS requests. TCP is widely used here as well, though most DNS traffic still uses UDP. There are periodic calls for a new protocol specifically addressing this pattern, though at this point the use of TCP is well established. If a *sequence* of request/reply operations is envisioned, a single TCP connection makes excellent sense, as the connection-setup overhead is minimal by comparison. See also *11.5 Remote Procedure Call (RPC)* and *12.22.2 SCTP*.

This century has seen an explosion in **streaming video** (*20.3.2 Streaming Video*), in lengths from a few minutes to a few hours. Streaming radio stations might be left playing indefinitely. TCP generally works well here, assuming the receiver can get, say, a minute ahead, buffering the video that has been received but not yet viewed. That way, if there is a dip in throughput due to congestion, the receiver has time to recover. Buffering works a little less well for streaming radio, as the listener doesn't want to get too far behind, though ten seconds is reasonable. Fortunately, audio bandwidth is smaller.

Another issue with streaming video is the bandwidth demand. Most streaming-video services attempt to estimate the available throughput, and then *adapt* to that throughput by changing the video resolution (*20.3 Real-time Traffic*).

Typically, video streaming operates on a start/stop basis: the sender pauses when the receiver's playback buffer is "full", and resumes when the playback buffer drops below a certain threshold.

If the video (or, for that matter, voice audio) is *interactive*, there is much less opportunity for stream buffering. If someone asks a simple question on an Internet telephone call, they generally want an answer more or less immediately; they do not expect to wait for the answer to make it through the other party's stream buffer. 200 ms of buffering is noticeable. Here we enter the realm of genuine real-time traffic (*20.3 Real-time Traffic*). UDP is often used to avoid head-of-line blocking. Lower bandwidth helps; voice-grade communications traditionally need only 8 KB/sec, less if compression is used. On the other hand, there may

be constraints on the *variation* in delivery time (known as *jitter*; see *20.11.3 RTP Control Protocol* for a specific numeric interpretation). Interactive video, with its much higher bandwidth requirements, is more difficult; fortunately, users seem to tolerate the common pauses and freezes.

Within the Transport layer, essentially all network connections involve a **client** and a **server**. Often this pattern is repeated at the Application layer as well: the client contacts the server and initiates a login session, or browses some web pages, or watches a movie. Sometimes, however, Application-layer exchanges fit the **peer-to-peer** model better, in which the two endpoints are more-or-less co-equals. Some examples include

- Internet telephony: there is no benefit in designating the party who place the call as the "client"
- Message passing in a CPU cluster, often using *11.5 Remote Procedure Call (RPC)*
- The routing-communication protocols of *9 Routing-Update Algorithms*. When router A reports to router B we might call A the client, but over time, as A and B report to one another repeatedly, the peer-to-peer model makes more sense.
- So-called peer-to-peer file-sharing, where individuals exchange files with other individuals (and as opposed to "cloud-based" file-sharing in which the "cloud" is the server).

RFC 5694 contains additional discussion of peer-to-peer patterns.

1.12.2 Content-Distribution Networks

Sites with an extremely large volume of content to distribute often turn to a specialized communication pattern called a content-distribution network or **CDN**. To reduce the amount of long-distance traffic, or to reduce the round-trip time, a site replicates its content at multiple data centers (also called *Points of Presence* (PoPs), *nodes* or *access points*). When a user makes a request (*eg* for a web page or a video), the request is routed to the nearest data center, and the content is delivered from there.

CDN Mapping

For a geographical map of the servers in the NetFlix CDN as of 2016, see *[BCTCU16]*. The map was created solely through end-user measurements. Most of the servers are in North and South America and Europe.

Large web pages typically contain both *static* content and also individualized *dynamic* content. On a typical Facebook page, for example, the videos and javascript might be considered static, while the individual wall posts might be considered dynamic. The CDN may serve up only the static content, leaving the dynamic content to come from a centralized server. Alternatively, the dynamic content may be replicated at each CDN node as well, though this introduces some real-time coordination issues. If dynamic content is not replicated, the CDN may include private high-speed links between its nodes, allowing for rapid low-congestion delivery to any node. Alternatively, CDN nodes may simply communicate using the public Internet. Finally, the CDN may (or may not) be configured to support fast *interactive* traffic between nodes, *eg* teleconferencing traffic, as is outlined in *20.6.1 A CDN Alternative to IntServ*.

Organizations can create their own CDNs, but often turn to specialized CDN providers, who often combine their CDN services with website-hosting services.

In principle, all that is needed to create a CDN is a multiplicity of data centers, each with its own connection to the Internet; private links between data centers are also common. In practice, many CDN providers also

try to build direct connections with the ISPs that serve their customers; the Google Edge Network above does this. This can improve performance and reduce traffic costs; we will return to this in *10.6.6.1 MED values and traffic engineering*.

1.13 Firewalls

One problem with having a program on your machine listening on an open TCP port is that someone may connect and then, using some flaw in the software on your end, do something malicious to your machine. Damage can range from the unintended downloading of personal data to compromise and takeover of your entire machine, making it a distributor of viruses and worms or a steppingstone in later break-ins of other machines.

A strategy known as **buffer overflow** (*22.2 Stack Buffer Overflow*) has been the basis for a great many total-compromise attacks. The idea is to identify a point in a server program where it fills a memory buffer with network-supplied data without careful length checking; almost any call to the C library function `gets(buf)` will suffice. The attacker then crafts an oversized input string which, when read by the server and stored in memory, overflows the buffer and overwrites subsequent portions of memory, typically containing the stack-frame pointers. The usual goal is to arrange things so that when the server reaches the end of the currently executing function, control is returned not to the calling function but instead to the attacker's own payload code located within the string.

A **firewall** is a mechanism to block connections deemed potentially risky, *eg* those originating from outside the site. Generally ordinary workstations do not ever need to accept connections from the Internet; client machines instead *initiate* connections to (better-protected) servers. So blocking incoming connections works reasonably well; when necessary (*eg* for games) certain ports can be selectively unblocked.

The original firewalls were built into routers. Incoming traffic to servers was often blocked unless it was sent to one of a modest number of "open" ports; for non-servers, typically all inbound connections were blocked. This allowed internal machines to operate reasonably safely, though being unable to accept incoming connections is sometimes inconvenient.

Nowadays per-host firewalls – in addition to router-based firewalls – are common: you can configure your workstation not to accept inbound connections to most (or all) ports regardless of whether software on the workstation requests such a connection. Outbound connections can, in many cases, also be prevented.

The typical home router implements something called network-address translation (*7.7 Network Address Translation*), which, in addition to conserving IPv4 addresses, also provides firewall protection.

1.14 Some Useful Utilities

There exists a great variety of useful programs for probing and diagnosing networks. Here we list a few of the simpler, more common and available ones; some of these are addressed in more detail in subsequent chapters. Some of these, like `ping`, are generally present by default; others will have to be installed from somewhere.

ping

`Ping` is useful to determine if another machine is accessible, *eg*

```
ping www.cs.luc.edu
ping 147.126.1.230
```

See *7.11 Internet Control Message Protocol* for how it works. Sometimes `ping` fails because the necessary packets are blocked by a firewall.

ifconfig, **ipconfig**, **ip**

To find your own IP address you can use `ipconfig` on Windows, `ifconfig` on linux and Macintosh systems, or the newer `ip addr list` on linux. The output generally lists all active interfaces but can be restricted to selected interfaces if desired. The `ip` command in particular can do many other things as well. The Windows command `netsh interface ip show config` also provides IP addresses.

nslookup, **dig** and **host**

This trio of programs, all developed by the Internet Systems Consortium, are all used for DNS lookups. They differ in convenience and options. The oldest is `nslookup`, the one with the most options (by a rather wide margin) is `dig`, and the newest and arguably most convenient for normal usage is `host`.

```
nslookup intronetworks.cs.luc.edu

Non-authoritative answer:
Name:   intronetworks.cs.luc.edu
Address: 162.216.18.28


dig intronetworks.cs.luc.edu

...
;; ANSWER SECTION:
intronetworks.cs.luc.edu. 86400 IN      A       162.216.18.28
...

host intronetworks.cs.luc.edu

intronetworks.cs.luc.edu has address 162.216.18.28
intronetworks.cs.luc.edu has IPv6 address 2600:3c03::f03c:91ff:fe69:f438
```

See *7.8.1 nslookup*.

traceroute

This lists the route from you to a remote host:

```
traceroute intronetworks.cs.luc.edu

1  147.126.65.1 (147.126.65.1)  0.751 ms  0.753 ms  0.783 ms
2  147.126.95.54 (147.126.95.54)  1.319 ms  1.286 ms  1.253 ms
3  12.31.132.169 (12.31.132.169)  1.225 ms  1.231 ms  1.193 ms
4  cr83.cgcil.ip.att.net (12.123.7.46)  4.983 ms cr84.cgcil.ip.att.net (12.123.7.170)  4.8
5  cr83.cgcil.ip.att.net (12.123.7.46)  4.926 ms  4.904 ms  4.888 ms
6  cr1.cgcil.ip.att.net (12.122.99.33)  5.043 ms cr2.cgcil.ip.att.net (12.122.132.109)  5
7  gar13.cgcil.ip.att.net (12.122.132.121)  3.879 ms  18.347 ms ggr4.cgcil.ip.att.net (12
8  chi-b21-link.telia.net (213.248.87.253)  2.344 ms  2.305 ms  2.409 ms
9  nyk-bb2-link.telia.net (80.91.248.197)  24.065 ms nyk-bb1-link.telia.net (213.155.136.
```

```
10  nyk-b3-link.telia.net (62.115.112.255)  23.557 ms  23.548 ms nyk-b3-link.telia.net (80
11  netaccess-tic-133837-nyk-b3.c.telia.net (213.248.99.90)  23.957 ms  24.382 ms  24.164 m
12  0.e1-4.tbr1.mmu.nac.net (209.123.10.101)  24.922 ms  24.737 ms  24.754 ms
13  207.99.53.42 (207.99.53.42)  24.024 ms  24.249 ms  23.924 ms
```

The last router (and `intronetworks.cs.luc.edu` itself) don't respond to the traceroute packets, so the list is not quite complete. The Windows `tracert` utility is functionally equivalent. See *7.11.1 Traceroute and Time Exceeded* for further information.

Traceroute sends, by default, three probes for each router. Sometimes the responses do not all come back from the same router, as happened above at routers 4, 6, 7, 9 and 10. Router 9 sent back three distinct responses.

On linux systems the mtr command may be available as an alternative to traceroute; it repeats the traceroute at one-second intervals and generates cumulative statistics.

route and **netstat**

The commands `route`, `route print` (Windows), `ip route show` (linux), and `netstat -r` (all systems) display the host's local IP forwarding table. For workstations not acting as routers, this includes the route to the default router and, usually, not much else. The default route is sometimes listed as destination 0.0.0.0 with netmask 0.0.0.0 (equivalent to 0.0.0.0/0).

The command `netstat -a` shows the existing TCP connections and open UDP sockets.

netcat

The `netcat` program, often called `nc`, allows the user to create TCP or UDP connections and send lines of text back and forth. It is seldom included by default. See *11.1.4 netcat* and *12.6.2 netcat again*.

WireShark

This is a convenient combination of packet capture and packet analysis, from wireshark.org. See *12.4 TCP and WireShark* and *8.11 Using IPv6 and IPv4 Together* for examples.

WireShark was originally named Etherreal. An earlier command-line-only packet-capture program is tcp-dump.

WireShark is the only non-command-line program listed here. It is sometimes desired to monitor packets on a remote system. If X-windows is involved (*eg* on linux), this can be done by logging in from ones local system using `ssh -X`, which enables X-windows forwarding, and then starting `wireshark` (or perhaps `sudo wireshark`) from the command line. Tcpdump is, of course, another alternative.

1.15 IETF and OSI

The Internet protocols discussed above are defined by the **Internet Engineering Task Force**, or IETF (under the aegis of the **Internet Architecture Board**, or IAB, in turn under the aegis of the **Internet Society**, ISOC). The IETF publishes "Request For Comment" or **RFC** documents that contain all the formal Internet standards; these are available at http://www.ietf.org/rfc.html (note that, by the time a document appears here, the actual comment-requesting period is generally long since closed). The five-layer model is closely associated with the IETF, though is not an official standard.

RFC standards sometimes allow modest flexibility. With this in mind, **RFC 2119** declares official understandings for the words MUST and SHOULD. A feature labeled with MUST is "an absolute requirement for the specification", while the term SHOULD is used when

> there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

The original **ARPANET** network was developed by the US government's Defense Advanced Research Projects Agency, or DARPA; it went online in 1969. The National Science Foundation began NSFNet in 1986; this largely replaced ARPANET. In 1991, operation of the NSFNet backbone was turned over to ANSNet, a private corporation. The ISOC was founded in 1992 as the NSF continued to retreat from the networking business.

Hallmarks of the IETF design approach were David Clark's declaration

> We reject: kings, presidents and voting.
>
> We believe in: rough consensus and running code.

and RFC Editor Jon Postel's aphorism

> Be liberal in what you accept, and conservative in what you send.

Postel's aphorism has come in for criticism in recent years, especially with regard to cryptographic protocols. To be fair, however, Postel wrote this in an era when protocol specifications sometimes failed to fully spell out the rules in every possible situation.

There is a persistent – though false – notion that the distributed-routing architecture of IP was due to a US Department of Defense mandate that the original ARPAnet be able to survive a nuclear attack. In fact, the developers of IP seemed unconcerned with this. However, Paul Baran did write, in his 1962 paper outlining the concept of packet switching, that

> If [the number of stations] is made sufficiently large, it can be shown that highly survivable system structures can be built – even in the thermonuclear era.

In 1977 the International Organization for Standardization, or **ISO**, founded the Open Systems Interconnection project, or **OSI**, a process for creation of new network standards. OSI represented an attempt at the creation of networking standards independent of any individual government.

The OSI project is today perhaps best known for its **seven-layer** networking model: between Transport and Application were inserted the **Session** and **Presentation** layers. The Session layer was to handle "sessions" between applications (including the graceful closing of Transport-layer connections, something included in TCP, and the re-establishment of "broken" Transport-layer connections, which TCP could sorely use), and the Presentation layer was to handle things like defining universal data formats (*eg* for binary numeric data, or for non-ASCII character sets), and eventually came to include compression and encryption as well.

Data presentation and session management are important concepts, but it has not proved necessary, or even particularly useful, to make them into true layers, in the sense that a layer communicates directly only with the layers adjacent to it. (SSL/TLS, *22.10.2 TLS*, might be an example of a true layer providing encryption; applications read and write data directly to the SSL/TLS endpoint, which in turn manages the TCP connection.) Generally what happens is that the Application layer manages its own Transport connections, and then reads and writes data directly from and to the Transport layer. The application then uses conventional libraries for Presentation actions such as encryption, compression and format translation. Similarly, applications typically implement their own Session actions for handling broken Transport connections, or

for multiplexing streams of data over a single Transport connection. Version 2 of the HTTP protocol, for example, contains a subprotocol for managing multiple streams; this is generally regarded as part of the Application layer.

OSI has its own version of IP and TCP. The IP equivalent is **CLNP**, the ConnectionLess Network Protocol, although OSI also defines a connection-*oriented* protocol CMNS. The TCP equivalent is TP4; OSI also defines TP0 through TP3 but those are for connection-oriented networks.

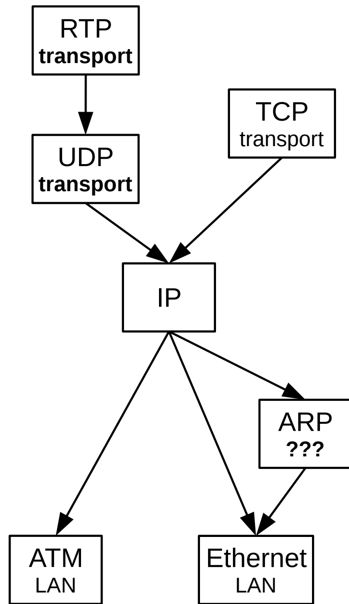
It seems clear that the primary reasons the OSI protocols failed in the marketplace were their ponderous bureaucracy for protocol management, their principle that protocols be completed before implementation began, and their insistence on rigid adherence to the specifications to the point of non-interoperability. In contrast, the IETF had (and still has) a "two working implementations" rule for a protocol to become a "Draft Standard". From **RFC 2026**:

> A specification from which at least *two independent and interoperable implementations* from different code bases have been developed, and for which sufficient successful operational experience has been obtained, may be elevated to the "Draft Standard" level. [emphasis added]

This rule has often facilitated the discovery of protocol design weaknesses early enough that the problems could be fixed. The OSI approach is a striking failure for the "waterfall" design model, when competing with the IETF's cyclic "prototyping" model. However, it is worth noting that the IETF has similarly been unable to keep up with rapid changes in html, particularly at the browser end; the OSI mistakes were mostly evident only in retrospect.

Trying to fit protocols into specific layers is often both futile and irrelevant. By one perspective, the Real-Time Protocol RTP lives at the Transport layer, but just above the UDP layer; others have put RTP into the Application layer. Parts of the RTP protocol resemble the Session and Presentation layers. A key component of the IP protocol is the set of various router-update protocols; some of these freely use higher-level layers. Similarly, tunneling might be considered to be a Link-layer protocol, but tunnels are often created and maintained at the Application layer.

A sometimes-more-successful approach to understanding "layers" is to view them instead as parts of a **protocol graph**. Thus, in the following diagram we have two protocol sublayers within the transport layer (UDP and RTP), and one protocol (ARP) not easily assigned to a layer.



1.16 Berkeley Unix

Though not officially tied to the IETF, the Berkeley Unix releases became *de facto* reference implementations for most of the TCP/IP protocols. 4.1BSD (BSD for Berkeley Software Distribution) was released in 1981, 4.2BSD in 1983, 4.3BSD in 1986, 4.3BSD-Tahoe in 1988, 4.3BSD-Reno in 1990, and 4.4BSD in 1994. Descendants today include FreeBSD, OpenBSD and NetBSD. The TCP implementations TCP Tahoe and TCP Reno (*13 TCP Reno and Congestion Management*) took their names from the corresponding 4.3BSD releases.

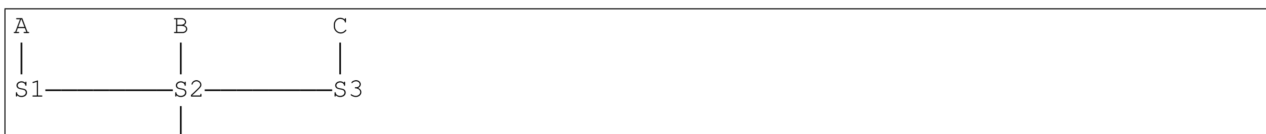
1.17 Epilog

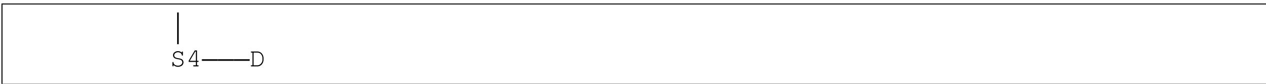
This completes our tour of the basics. In the remaining chapters we will expand on the material here.

1.18 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercise 2.5 is distinct, for example, from exercises 2.0 and 3.0. Exercises marked with a ♢ have solutions or hints at 24.1 Solutions for An Overview of Networks.

1.0. Give forwarding tables for each of the switches S1-S4 in the following network with destinations A, B, C, D. For the next_hop column, give the *neighbor* on the appropriate link rather than the interface number.

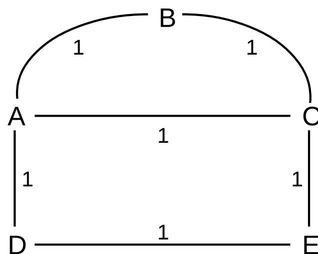




2.0. Give forwarding tables for each of the switches S1-S4 in the following network with destinations A, B, C, D. Again, use the neighbor form of next_hop rather than the interface form. Try to keep the route to each destination as short as possible. What decision has to be made in this exercise that did not arise in the preceding exercise?

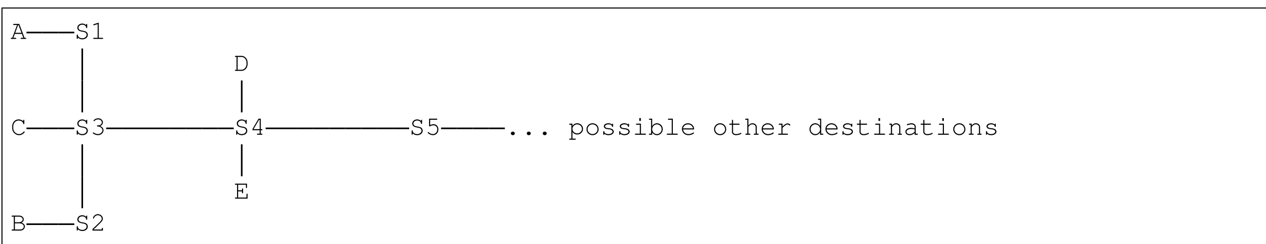
2.5. In the network of the previous exercise, suppose that destinations directly connected to an immediate neighbor are always reached via that neighbor; *eg* S1's forwarding table will always include ⟨B,S2⟩ and ⟨D,S4⟩. This leaves only routes to the diagonally opposite nodes undetermined (*eg* S1 to C). Show that, no matter which next_hop entries are chosen for the diagonally opposite destinations, no routing loops can ever be formed. (Hint: the number of links to any diagonally opposite switch is always 2.)

2.7.◇ Give forwarding tables for each of the switches A-E in the following network. Destinations are A-E themselves. Keep all route lengths the minimum possible (one hop for an immediate neighbor, two hops for everything else). If a destination is an immediate neighbor, you may list its next_hop as **direct** or **local** for simplicity. Indicate destinations for which there is more than one choice for next_hop.



3.0. Consider the following arrangement of switches and destinations. Give forwarding tables (in neighbor form) for S1-S4 that include **default** forwarding entries; *the default entries should point toward S5*. The default entries will thus automatically forward to the "possible other destinations" shown below right.

Eliminate all table entries that are implied by the default entry (that is, if the default entry is to S3, eliminate all other entries for which the next hop is S3).



4.0. Four switches are arranged as below. The destinations are S1 through S4 themselves.

(a). Give the forwarding tables for S1 through S4 assuming packets to adjacent nodes are sent along the connecting link, and packets to diagonally opposite nodes are sent clockwise.

(b). Give the forwarding tables for S1 through S4 assuming the S1–S4 link is not used at all, not even for S1⟷S4 traffic.

5.0. Suppose we have switches S1 through S4; the forwarding-table destinations are the switches themselves. The tab

S2: ⟨S1,S1⟩ ⟨S3,S3⟩ ⟨S4,S3⟩
S3: ⟨S1,S2⟩ ⟨S2,S2⟩ ⟨S4,S4⟩

From the above we can conclude that S2 must be directly connected to both S1 and S3 as its table lists them as next_hops; similarly, S3 must be directly connected to S2 and S4.

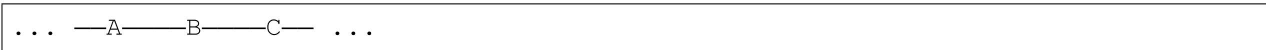
(a). The given tables are *consistent* with the network diagrammed in exercise 4.0. Are the tables also consistent with a network in which S1 and S4 are *not* directly connected? If so, give such a network; if not, explain why S1 and S4 must be connected.

(b). Now suppose S3's table is changed to the following. Find a network layout consistent with these tables in which S1 and S4 are not directly connected. Do not add additional switches.

S3: ⟨**S1,S4**⟩ ⟨S2,S2⟩ ⟨S4,S4⟩

While the table for S4 is not given, you may assume that forwarding does work correctly. However, you should *not* assume that paths are the shortest possible; in particular, you should not assume that each switch will always reach its directly connected neighbors by using the direct connection.

6.0. (a) Suppose a network is as follows, with the only path from A to C passing through B:



Explain why a single routing loop cannot include both A and C. Hint: if the loop involves destination D, how does B forward to D?

(b). Suppose a routing loop follows the path A—S_1—S_2— ... —S_n—A, where none of the S_i are equal to A. Show that all the S_i must be distinct. (A corollary of this is that any routing loop created by datagram-forwarding either involves forwarding back and forth between a pair of adjacent switches, or else involves an actual graph cycle in the network topology; linear loops of length greater than 1 are impossible.)

7.0 Consider the following arrangement of switches:

Suppose S1-S6 have the forwarding tables below. For each of the following destinations, suppose a packet is sent to the destination *from S1*.

(a). A
(b). B
(c). C
(d).◇ D
(e). E
(f). F

Give the switches the packet will pass through, including the initial switch S1, up until the final switch S10-S12.

S1: (A,S4), (B,S2), (C,S4), (D,S2), (E,S2), (F,S4)

S2: (A,S5), (B,S5), (D,S5), (E,S3), (F,S3)

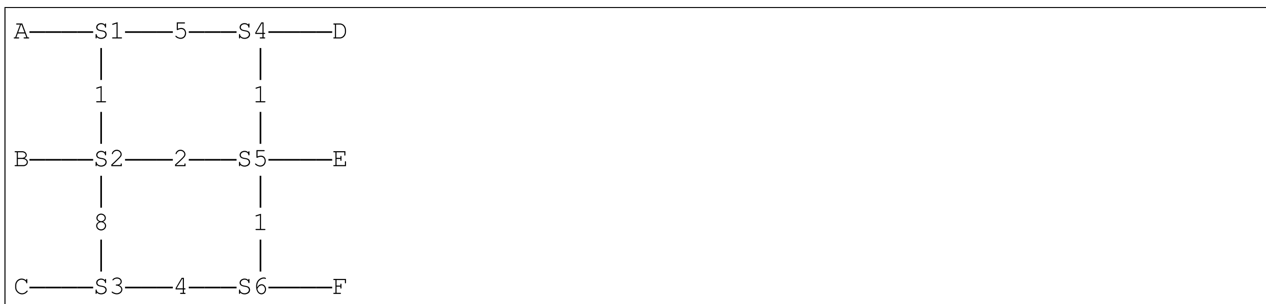
S3: (B,S6), (C,S2), (E,S6), (F,S6)

S4: (A,S10), (C,S5), (E,S10), (F,S5)

S5: (A,S6), (B,S11), (C,S6), (D,S6), (E,S4), (F,S2)

S6: (A,S3), (B,S12), (C,S12), (D,S12), (E,S5), (F,S12)

7.5 Suppose a set of nodes A-F and switches S1-S6 are connected as shown.



The links between switches are labeled with **weights**, which are used by some routing applications. The weights represent the cost of using that link. You are to find the path through S1-S6 with lowest total cost (that is, with smallest sum of weights), for each of the following transmissions. For example, the lowest-cost path from A to E is A–S1–S2–S5–E, for a total cost of 1+2=3; the alternative path A–S1–S4–S5–E has total cost 5+1=6.

(a).♢ A→F
(b). A→D
(c). A→C

(d). Give the complete forwarding table for S**2**, where all routes are selected for lowest total cost.

8.0 In exercise 7.0, the routes taken by packets A-D are reasonably direct, but the routes for E and F are rather circuitous.

(a). Assign weights to the seven links S1–S2, S2–S3, S1–S4, S2–S5, S3–S6, S4–S5 and S5–S6, as in exercise 7.5, so that destination E's route in exercise 7.0 becomes the optimum (lowest total link weight) path.
(b). Assign weights to the seven links that make destination F's route in exercise 7.0 optimal. (This will be a different set of weights from part (a).)

Hint: you can do this by assigning a weight of 1 to all links *except* to one or two "bad" links; the "bad" links get a weight of 10. In each of (a) and (b) above, the route taken will be the route that avoids all the "bad" links. You must treat (a) entirely differently from (b); there is no assignment of weights that can account for both routes.

9.0 Suppose we have the following three Class C IP networks, joined by routers R1–R4. There is no connection to the outside Internet. Give the forwarding table for each router. For networks directly connected to a router (*eg* 200.0.1/24 and R1), include the network in the table but list the next hop as **direct** or **local**.

We now turn to a deeper analysis of the ubiquitous Ethernet LAN protocol. Current user-level Ethernet today (2013) is usually 100 Mbps, with Gigabit and 10 Gigabit Ethernet standard in server rooms and backbones, but because the potential for collisions makes Ethernet speeds scale in odd ways, we will start with the 10 Mbps formulation. While the 10 Mbps speed is obsolete, and while even the Ethernet collision mechanism is largely obsolete, collision management itself continues to play a significant role in wireless networks.

The original Ethernet specification was the 1976 paper of Metcalfe and Boggs, *[MB76]*. The data rate was 10 megabits per second, and all connections were made with coaxial cable instead of today's twisted pair. The authors described their architecture as follows:

> We cannot afford the redundant connections and dynamic routing of store-and-forward packet switching to assure reliable communication, so we choose to achieve reliability through simplicity. We choose to make the shared communication facility passive so that the failure of an active element will tend to affect the communications of only a single station.

Classic Ethernet was indeed simple, and – mostly – passive. In its most basic form, the Ethernet medium was one long piece of coaxial cable, onto which stations could be connected via **taps**. If two stations happened to transmit at the same time – most likely because they were both waiting for a third station to finish – their signals were lost to the resultant **collision**. The only active components besides the stations were **repeaters**, originally intended simply to make end-to-end joins between cable segments.

Repeaters soon evolved into multiport devices, allowing the creation of arbitrary tree (that is, loop-free) topologies. At this point the standard wiring model shifted from one long cable, snaking from host to host, to a "star" network, where each host connected directly to a central multipoint repeater. This shift allowed for the replacement of expensive coaxial cable by the much-cheaper twisted pair; links could not be as long, but they did not need to be.

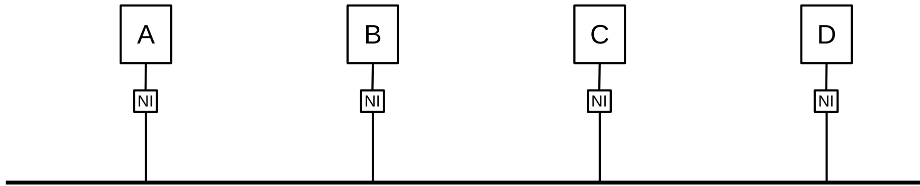
Repeaters, which forwarded collisions, soon gave way to **switches**, which did not (*2.4 Ethernet Switches*). Switches thus partitioned an Ethernet into disjoint **collision domains**, or physical Ethernets, through which collisions could propagate; an aggregation of physical Ethernets connected by switches was then sometimes known as a **virtual** Ethernet. Collision domains became smaller and smaller, eventually down to individual links and then vanishing entirely.

Throughout all these changes, Ethernet never implemented true redundant connections, in that at any one instant the topology was always required to be loop-free. However, Ethernet did adopt a mechanism by which idle backup links can quickly be placed into service after a primary link fails; *2.5 Spanning Tree Algorithm and Redundancy*.

2.1 10-Mbps Classic Ethernet

Originally, Ethernet consisted of a long piece of cable (possibly spliced by **repeaters**). When a station transmitted, the data went everywhere along that cable. Such an arrangement is known as a **broadcast bus**; all packets were, at least at the physical layer, broadcast onto the shared medium and could be seen, theoretically, by all other nodes. Logically, however, most packets would appear to be transmitted point-to-point, not broadcast. This was because between each station CPU and the cable there was a peripheral device

(that is, a card) known as a **network interface**, which would take care of the details of transmitting and receiving. The network interface would (and still does) decide when a received packet should be forwarded to the host, via a CPU interrupt.



Whenever two stations transmitted at the same time, the signals would **collide**, and interfere with one another; both transmissions would fail as a result. Proper handling of collisions was an essential part of the access-mediation strategy for the shared medium. In order to minimize collision loss, each station implemented the following:

1. Before transmission, wait for the line to become quiet
2. While transmitting, continually monitor the line for signs that a collision has occurred; if a collision is detected, cease transmitting
3. If a collision occurs, use a backoff-and-retransmit strategy

These properties can be summarized with the **CSMA/CD** acronym: Carrier Sense, Multiple Access, Collision Detect. (The term "carrier sense" was used by Metcalfe and Boggs as a synonym for "signal sense"; there is no literal carrier frequency to be sensed.) It should be emphasized that collisions are a normal event in Ethernet, well-handled by the mechanisms above.

IEEE 802 Network Standards

The IEEE network standards all begin with 802: 802.3 is Ethernet, 802.11 is Wi-Fi, 802.16 is WiMAX, and there are many others. One sometimes encounters the claim that 802 represents the date of an early meeting: February 1980. However, the IEEE has a continuous stream of standards (with occasional gaps): 799: Handling and Disposal of Transformer PCBs, 800: D-C Aircraft Rotating Machines, 803: Recommended Practice for Unique Identification in Power Plants, *etc*.

Classic Ethernet came in version 1 [1980, DEC-Intel-Xerox], version 2 [1982, DIX], and **IEEE 802.3**. There are some minor electrical differences between these, and one rather substantial packet-format difference, below. In addition to these, the Berkeley Unix trailing-headers packet format was used for a while.

There were three physical formats for 10 Mbps Ethernet cable: thick coax (10BASE-5), thin coax (10BASE-2), and, last to arrive, twisted pair (10BASE-T). Thick coax was the original; economics drove the successive development of the later two. The cheaper twisted-pair cabling eventually almost entirely displaced coax, at least for host connections.

The original specification included support for **repeaters**, which were in effect signal amplifiers although they might attempt to clean up a noisy signal. Repeaters processed each bit individually and did no buffering. In the telecom world, a repeater might be called a **digital regenerator**. A repeater with more than two ports was commonly called a **hub**; hubs allowed branching and thus much more complex topologies.

It was the rise of hubs that enabled **star topologies** in which each host connects directly to the hub rather than to one long run of coax. This in turn enabled twisted-pair cable: while this supported maximum runs of about 100 meters, versus the 500 meters of thick coax, each run simply had to go from the host to the central hub in the wiring closet. This was much more convenient than having to snake coax all around the building. A hub failure would bring the network down, but hubs proved largely reliable.

Bridges – later known as **switches** – came along a short time later. While repeaters act at the bit layer, a switch reads in and forwards an entire packet as a unit, and the destination address is consulted to determine to where the packet is forwarded. Except for possible collision-related performance issues, hubs and switches are interchangeable. Eventually, most wiring-closet hubs were replaced with switches.

Hubs propagate collisions; switches do not. If the signal representing a collision were to arrive at one port of a hub, it would, like any other signal, be retransmitted out all other ports. If a switch were to detect a collision one one port, no other ports would be involved; only packets received successfully are ever retransmitted out other ports.

Originally, switches were seen as providing interconnection ("bridging") between separate physical Ethernets, but later a switched Ethernet was seen as one large "virtual" Ethernet, composed of smaller collision domains. We return to switching below in *2.4 Ethernet Switches*.

In the original thick-coax cabling, connections were made via **taps**, often literally drilled into the coax central conductor. Thin coax allowed the use of T-connectors to attach hosts. Twisted-pair does not allow mid-cable attachment; it is only used for point-to-point links between hosts, switches and hubs. Mid-cable attachment, however, was always simply a way of avoiding the need for active devices like hubs and switches.

There is still a role for hubs today when one wants to monitor the Ethernet signal from A to B (*eg* for intrusion detection analysis), although some switches now also support a form of monitoring.

All three cable formats could interconnect, although only through repeaters and hubs, and all used the same 10 Mbps transmission speed. While twisted-pair cable is still used by 100 Mbps Ethernet, it generally needs to be a higher-performance version known as Category 5, versus the 10 Mbps Category 3.

Data in 10 Mbps Ethernets was transmitted using Manchester encoding; see *4.1.3 Manchester*. This meant that the electronics had to operate, in effect, at 20 Mbps. Faster Ethernets use different encodings.

2.1.1 Ethernet Packet Format

Here is the format of a typical Ethernet packet (DIX specification); it is still used for newer, faster Ethernets:

The destination and source addresses are 48-bit quantities; the type is 16 bits, the data length is variable up to a maximum of 1500 bytes, and the final CRC checksum is 32 bits. The checksum is added by the Ethernet hardware, never by the host software. There is also a preamble, not shown: a block of 1 bits followed by a 0, in the front of the packet, for synchronization. The type field identifies the next higher protocol layer; a few common type values are 0x0800 = IP, 0x8137 = IPX, 0x0806 = ARP.

The IEEE 802.3 specification replaced the type field by the length field, though this change never caught on. The two formats can be distinguished as long as the type values used are larger than the maximum Ethernet length of 1500 (or 0x05dc); the type values given in the previous paragraph all meet this condition.

The Ethernet maximum packet length of 1500 bytes worked well in the past, but can seem inconveniently small at 10 Gbit speeds. But 1500 bytes has become the *de facto* maximum packet size throughout the Internet, not just on Ethernet LANs; increasing it would be difficult. TCP TSO (*12.5 TCP Offloading*) is one alternative.

Each Ethernet card has a (hopefully unique) physical address in ROM; by default any packet sent to this address will be received by the board and passed up to the host system. Packets addressed to other physical addresses will be seen by the card, but ignored (by default). All Ethernet devices also agree on a broadcast address of all 1's: a packet sent to the broadcast address will be delivered to all attached hosts.

It is sometimes possible to change the physical address of a given card in software. It is almost universally possible to put a given card into **promiscuous mode**, meaning that all packets on the network, no matter what the destination address, are delivered to the attached host. This mode was originally intended for diagnostic purposes but became best known for the security breach it opens: it was once not unusual to find a host with network board in promiscuous mode and with a process collecting the first 100 bytes (presumably including userid and password) of every telnet connection.

2.1.2 Ethernet Multicast

Another category of Ethernet addresses is **multicast**, used to transmit to a *set* of stations; streaming video to multiple simultaneous viewers might use Ethernet multicast. The lowest-order bit in the first byte of an address indicates whether the address is physical or multicast. To receive packets addressed to a given multicast address, the host must inform its network interface that it wishes to do so; once this is done, any arriving packets addressed to that multicast address are forwarded to the host. The set of subscribers to a given multicast address may be called a **multicast group**. While higher-level protocols might prefer that the subscribing host also notifies some other host, *eg* the sender, this is not required, although that might be the easiest way to learn the multicast address involved. If several hosts subscribe to the same multicast address, then each will receive a copy of each multicast packet transmitted.

We are now able to list all cases in which a network interface forwards a received packet up to its attached host:

- if the destination address of the received packet maches the physical address of the interface
- if the destination address of the received packet is the broadcast address
- if the interface is in promiscuous mode
- if the destination address of the received packet is a multicast address and the host has told the network interface to accept packets sent to that multicast address

If switches (below) are involved, they must normally forward multicast packets on all outbound links, exactly as they do for broadcast packets; switches have no obvious way of telling where multicast subscribers might be. To avoid this, some switches do try to engage in some form of multicast filtering, sometimes by snooping on higher-layer multicast protocols. Multicast Ethernet is seldom used by IPv4, but plays a larger role in IPv6 configuration.

2.1.3 Ethernet Address Internal Structure

The second-to-lowest-order bit of a physical Ethernet address indicates whether that address is believed to be globally unique or if it is only locally unique; this is known as the **Universal/Local** bit. For real Ethernet physical addresses, the multicast and universal/local bits of the first byte should both be 0.

When (global) Ethernet IDs are assigned to physical Ethernet cards by the manufacturer, the first three bytes serve to indicate the manufacturer. They are allocated by the IEEE, and are officially known as *organizationally unique identifiers*. These can be looked up at any of several sites on the Internet to identify the manufacturer associated with any given Ethernet address; the official IEEE site is standards.ieee.org/develop/regauth/oui/public.html (OUIs must be entered here without colons).

As long as the manufacturer involved is diligent in assigning the second three bytes, every manufacturer-provided Ethernet address *should* be globally unique. Lapses, however, are not unheard of.

Ethernet addresses for *virtual* machines must be distinct from the Ethernet address of the host system, and may be (*eg* with so-called "bridged" configurations) as visible on the LAN as that host system's address. The first three bytes of virtual Ethernet addresses are often taken from the OUI assigned to the manufacturer whose card is being emulated; the last three bytes are then either set randomly or via configuration. In principle, the universal/local bit should be 1, as the address is only locally unique, but this is often ignored. It is entirely possible for virtual Ethernet addresses to be assigned so as to have some local meaning, though this appears not to be common.

2.1.4 The LAN Layer

The LAN layer, at its upper end, supplies to the network layer a mechanism for addressing a packet and sending it from one station to another. At its lower end, it handles interactions with the physical layer. The LAN layer covers packet addressing, delivery and receipt, forwarding, error detection, collision detection and collision-related retransmission attempts.

In IEEE protocols, the LAN layer is divided into the **media access control**, or MAC, sublayer and a higher **logical link control**, or LLC, sublayer for higher-level flow-control functions that today have moved largely to the transport layer. For example, the HDLC protocol (*4.1.5.1 HDLC*) supports sliding windows (*6.2 Sliding Windows*) as an option, as did the early X.25 protocol. ATM, *3.5 Asynchronous Transfer Mode: ATM*, also supports some higher-level functions, though not sliding windows.

Because the LLC layer is so often insignificant, and because the most well-known LAN-layer functions are in fact part of the MAC sublayer, it is common to identify the LAN layer with its MAC sublayer, especially for IEEE protocols where the MAC layer has official standing. In particular, LAN-layer addresses are perhaps most often called MAC addresses.

Generally speaking, much of the operation of the LAN/MAC layer takes place in the network card. Host systems (including drivers) are, for example, generally oblivious to collisions (although they may query the card for collision statistics). In some cases, *eg* with Wi-Fi rate scaling (*3.7.2 Dynamic Rate Scaling*), the host-system driver may get involved.

2.1.5 The Slot Time and Collisions

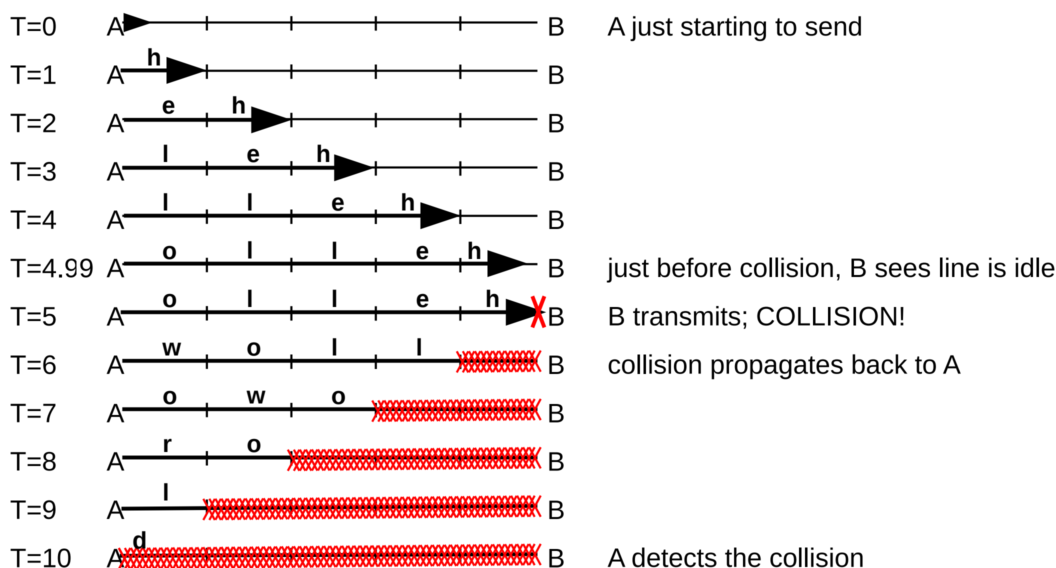
The **diameter** of an Ethernet is the maximum distance between any pair of stations. The actual total length of cable can be much greater than this, if, for example, the topology is a "star" configuration. The maximum allowed diameter, measured in bits, is limited to 232 (a sample "budget" for this is below). This makes the round-trip-time 464 bits. As each station involved in a collision discovers it, it transmits a special **jam signal** of up to 48 bits. These 48 jam bits bring the total above to 512 bits, or 64 bytes. The time to send these 512 bits is the **slot time** of an Ethernet; time intervals on Ethernet are often described in bit times but in conventional time units the slot time is 51.2 µsec.

The value of the slot time determines several subsequent aspects of Ethernet. If a station has transmitted for one slot time, then no collision can occur (unless there is a hardware error) for the remainder of that packet. This is because one slot time is enough time for any other station to have realized that the first station has started transmitting, so after that time they will wait for the first station to finish. Thus, after one slot time a station is said to have **acquired** the network. The slot time is also used as the basic interval for retransmission scheduling, below.

Conversely, a collision *can* be received, in principle, at any point up until the end of the slot time. As a result, Ethernet has a **minimum packet size**, equal to the slot time, *ie* 64 bytes (or 46 bytes in the data portion). A station transmitting a packet this size is assured that *if* a collision were to occur, the sender would detect it (and be able to apply the retransmission algorithm, below). Smaller packets might collide and yet the sender not know it, ultimately leading to greatly reduced throughput.

If we need to send less than 46 bytes of data (for example, a 40-byte TCP ACK packet), the Ethernet packet must be padded out to the minimum length. As a result, all protocols running on top of Ethernet need to provide some way to specify the actual data length, as it cannot be inferred from the received packet size.

As a specific example of a collision occurring as late as possible, consider the diagram below. A and B are 5 units apart, and the bandwidth is 1 byte/unit. A begins sending "helloworld" at T=0; B starts sending just as A's message arrives, at T=5. B has listened before transmitting, but A's signal was not yet evident. A doesn't discover the collision until 10 units have elapsed, which is twice the distance.



Here are typical maximum values for the delay in 10 Mbps Ethernet due to various components. These

are taken from the Digital-Intel-Xerox (DIX) standard of 1982, except that "point-to-point link cable" is replaced by standard cable. The DIX specification allows 1500m of coax with two repeaters and 1000m of point-to-point cable; the table below shows 2500m of coax and four repeaters, following the later IEEE 802.3 Ethernet specification. Some of the more obscure delays have been eliminated. Entries are one-way delay times, in bits. The maximum path may have four repeaters, and ten transceivers (simple electronic devices between the coax cable and the NI cards), each with its drop cable (two transceivers per repeater, plus one at each endpoint).

Ethernet delay budget

item	length	delay, in bits	explanation (c = speed of light)
coax	2500M	110 bits	23 meters/bit (.77c)
transceiver cables	500M	25 bits	19.5 meters/bit (.65c)
transceivers		40 bits, max 10 units	4 bits each
repeaters		25 bits, max 4 units	6+ bits each (DIX 7.6.4.1)
encoders		20 bits, max 10 units	2 bits each (for signal generation)

The total here is 220 bits; in a full accounting it would be 232. Some of the numbers shown are a little high, but there are also signal rise time delays, sense delays, and timer delays that have been omitted. It works out fairly closely.

Implicit in the delay budget table above is the "length" of a bit. The speed of propagation in copper is about $0.77\times c$, where $c=3\times 10^8$ m/sec = 300 m/µsec is the speed of light in vacuum. So, in 0.1 microseconds (the time to send one bit at 10 Mbps), the signal propagates approximately $0.77\times c\times 10^{-7}$ = 23 meters.

Ethernet packets also have a **maximum** packet size, of 1500 bytes. This limit is primarily for the sake of fairness, so one station cannot unduly monopolize the cable (and also so stations can reserve buffers guaranteed to hold an entire packet). At one time hardware vendors often marketed their own incompatible "extensions" to Ethernet which enlarged the maximum packet size to as much as 4KB. There is no technical reason, actually, not to do this, except compatibility.

The signal loss in any single segment of cable is limited to 8.5 db, or about 14% of original strength. Repeaters will restore the signal to its original strength. The reason for the per-segment length restriction is that Ethernet collision detection requires a strict limit on how much the remote signal can be allowed to lose strength. It is possible for a station to detect and reliably read very weak remote signals, but *not at the same time that it is transmitting locally*. This is exactly what must be done, though, for collision detection to work: remote signals must arrive with sufficient strength to be heard even while the receiving station is itself transmitting. The per-segment limit, then, has nothing to do with the overall length limit; the latter is set only to ensure that a sender is guaranteed of detecting a collision, even if it sends the minimum-sized packet.

2.1.6 Exponential Backoff Algorithm

Whenever there is a collision the exponential backoff algorithm – operating at the MAC layer – is used to determine when each station will retry its transmission. Backoff here is called *exponential* because the range from which the backoff value is chosen is doubled after every successive collision involving the same packet. Here is the full Ethernet transmission algorithm, including backoff and retransmissions:

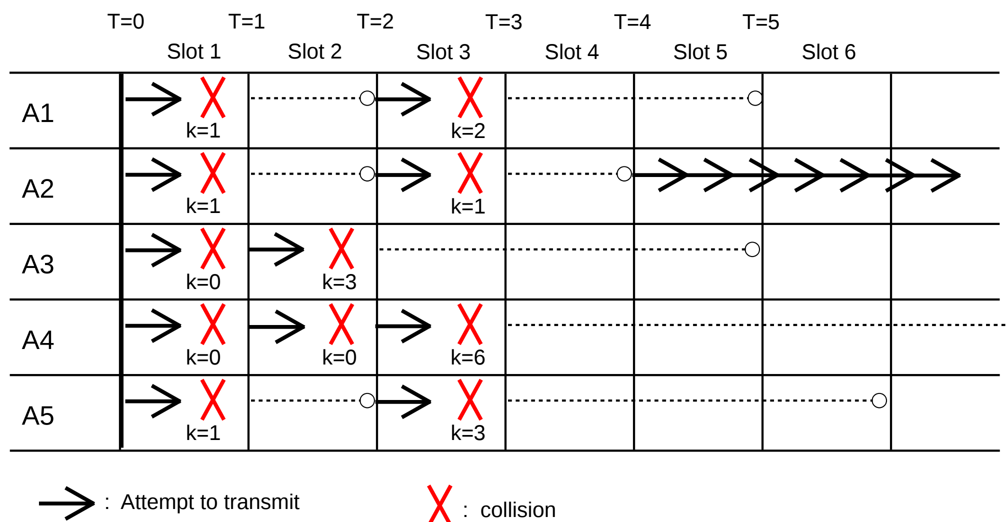
1. Listen before transmitting ("carrier detect")

2. If line is busy, wait for sender to stop and then wait an additional 9.6 microseconds (96 bits). One consequence of this is that there is always a 96-bit gap between packets, so packets do not run together.
3. Transmit while simultaneously monitoring for collisions
4. If a collision does occur, send the jam signal, and choose a **backoff time** as follows: For transmission N, $1 \leqslant N \leqslant 10$ (N=0 represents the original attempt), choose k randomly with $0 \leqslant k < 2^N$. Wait k slot times ($k \times 51.2$ µsec). Then check if the line is idle, waiting if necessary for someone else to finish, and then retry step 3. For $11 \leqslant N \leqslant 15$, choose k randomly with $0 \leqslant k < 1024$ ($= 2^{10}$)
5. If we reach N=16 (16 transmission attempts), give up.

If an Ethernet sender does not reach step 5, there is a very high probability that the packet was delivered successfully.

Exponential backoff means that if two hosts have waited for a third to finish and transmit simultaneously, and collide, then when N=1 they have a 50% chance of recollision; when N=2 there is a 25% chance, *etc*. When $N \geqslant 10$ the maximum wait is 52 milliseconds; without this cutoff the maximum wait at N=15 would be 1.5 seconds. As indicated above in the minimum-packet-size discussion, this retransmission strategy assumes that the sender is able to detect the collision while it is still sending, so it knows that the packet must be resent.

In the following diagram is an example of several stations attempting to transmit all at once, and using the above transmission/backoff algorithm to sort out who actually gets to acquire the channel. We assume we have five prospective senders A1, A2, A3, A4 and A5, all waiting for a sixth station to finish. We will assume that collision detection always takes one slot time (it will take much less for nodes closer together) and that the slot start-times for each station are synchronized; this allows us to measure time in slots. A solid arrow at the start of a slot means that sender began transmission in that slot; a red X signifies a collision. If a collision occurs, the backoff value k is shown underneath. A dashed line shows the station waiting k slots for its next attempt.



At T=0 we assume the transmitting station finishes, and all the Ai transmit and collide. At T=1, then, each of the Ai has discovered the collision; each chooses a random k<2. Let us assume that A1 chooses k=1, A2 chooses k=1, A3 chooses k=0, A4 chooses k=0, and A5 chooses k=1.

Those stations choosing k=0 will retransmit immediately, at T=1. This means A3 and A4 collide again, and at T=2 they now choose random k<4. We will Assume A3 chooses k=3 and A4 chooses k=0; A3 will try again at T=2+3=5 while A4 will try again at T=2, that is, now.

At T=2, we now have the original A1, A2, and A5 transmitting for the second time, while A4 trying again for the third time. They collide. Let us suppose A1 chooses k=2, A2 chooses k=1, A5 chooses k=3, and A4 chooses k=6 (A4 is choosing k<8 at random). Their scheduled transmission attempt times are now A1 at T=3+2=5, A2 at T=4, A5 at T=6, and A4 at T=9.

At T=3, nobody attempts to transmit. But at T=4, A2 is the only station to transmit, and so successfully seizes the channel. By the time T=5 rolls around, A1 and A3 will check the channel, that is, listen first, and wait for A2 to finish. At T=9, A4 will check the channel again, and also begin waiting for A2 to finish.

A maximum of 1024 hosts is allowed on an Ethernet. This number apparently comes from the maximum range for the backoff time as $0 \leqslant k < 1024$. If there are 1024 hosts simultaneously trying to send, then, once the backoff range has reached k<1024 (N=10), we have a good chance that one station will succeed in seizing the channel, that is; the minimum value of all the random k's chosen will be unique.

This backoff algorithm is not "fair", in the sense that the longer a station has been waiting to send, the lower its priority sinks. Newly transmitting stations with N=0 need not delay at all. The Ethernet capture effect, below, illustrates this unfairness.

2.1.7 Capture effect

The capture effect is a scenario illustrating the potential lack of fairness in the exponential backoff algorithm. The unswitched Ethernet must be fully busy, in that each of two senders always has a packet ready to transmit.

Let A and B be two such busy nodes, simultaneously starting to transmit their first packets. They collide. Suppose A wins, and sends. When A is finished, B tries to transmit again. But A has a second packet, and so A tries too. A chooses a backoff k<2 (that is, between 0 and 1 inclusive), but since B is on its second attempt it must choose k<4. This means A is favored to win. Suppose it does.

After that transmission is finished, A and B try yet again: A on its first attempt for its third packet, and B on its third attempt for its first packet. Now A again chooses k<2 but B must choose k<8; this time A is much more likely to win. Each time B fails to win a given backoff, its probability of winning the next one is reduced by about 1/2. It is quite possible, and does occur in practice, for B to lose *all* the backoffs until it reaches the maximum of N=16 attempts; once it has lost the first three or four this is in fact quite likely. At this point B simply discards the packet and goes on to the next one with N reset to 1 and k chosen from {0,1}.

The capture effect can be fixed with appropriate modification of the backoff algorithm; the Binary Logarithmic Arbitration Method (BLAM) was proposed in *[MM94]*. The BLAM algorithm was considered for the then-nascent 100 Mbps Fast Ethernet standard. But in the end a hardware strategy won out: Fast Ethernet supports "full-duplex" mode which is collision-free (see *2.2 100 Mbps (Fast) Ethernet*, below). While Fast Ethernet continues to support the original "half-duplex" mode, it was assumed that any sites concerned enough about performance to be worried about the capture effect would opt for full-duplex.

2.1.8 Hubs and topology

Ethernet hubs (multiport repeaters) change the topology, but not the fundamental constraints. Hubs enabled the model in which each station now had its own link to the wiring closet. Loops are still forbidden. The maximum diameter of an Ethernet consisting of multiple segments joined by hubs is still constrained by the round-trip-time, and the need to detect collisions before the sender has completed sending, as before. However, the network "diameter", or maximum distance between two hosts, is no longer synonymous with "total length". Because twisted-pair links are much shorter, about 100 meters, the diameter constraint is often immaterial.

2.1.9 Errors

Packets can have bits flipped or garbled by electrical noise on the cable; estimates of the frequency with which this occurs range from 1 in 10^4 to 1 in 10^6. Bit errors are not uniformly likely; when they occur, they are likely to occur in bursts. Packets can also be lost in hubs, although this appears less likely. Packets can be lost due to collisions only if the sending host makes 16 unsuccessful transmission attempts and gives up. Ethernet packets contain a 32-bit CRC error-detecting code (see *5.4.1 Cyclical Redundancy Check: CRC*) to detect bit errors. Packets can also be misaddressed by the sending host, or, most likely of all, they can arrive at the receiving host at a point when the receiver has no free buffers and thus be dropped by a higher-layer protocol.

2.1.10 CSMA persistence

A carrier-sense/multiple-access transmission strategy is said to be **nonpersistent** if, when the line is busy, the sender waits a randomly selected time. A strategy is **p-persistent** if, after waiting for the line to clear, the sender sends with probability p⩽1. Ethernet uses 1-persistence. A consequence of 1-persistence is that, if more than one station is waiting for line to clear, then when the line does clear a collision is certain. However, Ethernet then gracefully handles the resulting collision via the usual exponential backoff. If N stations are waiting to transmit, the time required for one station to win the backoff is linear in N.

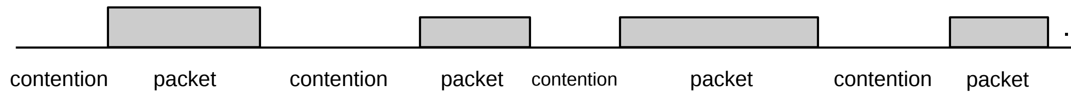
When we consider the Wi-Fi collision-handling mechanisms in *3.7 Wi-Fi*, we will see that collisions cannot be handled quite as cheaply: for one thing, there is no way to detect a collision in progress, so the entire packet-transmission time is wasted. In the Wi-Fi case, p-persistence is used with p<1.

An Ethernet broadcast storm was said to occur when there were too many transmission attempts, and most of the available bandwidth was tied up in collisions. A properly functioning classic Ethernet had an effective bandwidth of as much as 50-80% of the nominal 10Mbps capacity, but attempts to transmit more than this typically resulted in *successfully* transmitting a good deal less.

2.1.11 Analysis of Classic Ethernet

How much time does Ethernet "waste" on collisions? A paradoxical attribute of Ethernet is that raising the transmission-attempt rate on a busy segment can *reduce* the actual throughput. More transmission attempts can lead to longer **contention intervals** between packets, as senders use the transmission backoff algorithm to attempt to acquire the channel. What effective throughput can be achieved?

It is convenient to refer to the time between packet transmissions as the contention interval even if there is no actual contention, that is, even if the network is idle; we cannot tell if stations are not transmitting because they have nothing to send, or if they are simply waiting for their backoff timer to expire. Thus, a timeline for Ethernet always consists of alternating packet transmissions and contention intervals:



Ethernet packet transmissions alternating with contention intervals

As a first look at contention intervals, assume that there are N stations waiting to transmit at the start of the interval. It turns out that, if all follow the exponential backoff algorithm, we can expect O(N) slot times before one station successfully acquires the channel; thus, Ethernets are happiest when N is small and there are only a few stations simultaneously transmitting. However, multiple stations are not necessarily a severe problem. Often the number of slot times needed turns out to be about N/2, and slot times are short. If N=20, then N/2 is 10 slot times, or 640 bytes. However, one packet time might be 1500 bytes. If packet intervals are 1500 bytes and contention intervals are 640 byes, this gives an overall throughput of 1500/(640+1500) = 70% of capacity. In practice, this seems to be a reasonable upper limit for the throughput of classic shared-media Ethernet.

2.1.11.1 The ALOHA models

Another approach to analyzing the Ethernet contention interval is by using the ALOHA model that was a precursor to Ethernet. In the ALOHA model, stations transmit packets *without* listening first for a quiet line or monitoring the transmission for collisions (this models the situation of several ground stations transmitting to a satellite; the ground stations are presumed unable to see one another). Similarly, during the Ethernet contention interval, stations transmit one-slot packets under what are effectively the same conditions (we return to this below).

The ALOHA model yields roughly similar throughput values to the O(N) model of the previous section. We make, however, a rather artificial assumption: that there are a very large number of active senders, each transmitting at a very low rate. The model may thus have limited direct applicability to typical Ethernets.

To model the success rate of ALOHA, assume all the packets are the same size and let T be the time to send one (fixed-size) packet; T represents the Aloha slot time. We will find the transmission rate that optimizes throughput.

The core assumption of this model is that that a large number N of hosts are transmitting, each at a relatively low rate of s packets/slot. Denote by G the average number of transmission attempts per slot; we then have G = Ns. We will derive an expression for S, the average rate of *successful* transmissions per slot, in terms of G.

If two packets overlap during transmissions, both are lost. Thus, a successful transmission requires everyone else quiet for an interval of 2T: if a sender succeeds in the interval from t to t+T, then no other node can have tried to begin transmission in the interval t–T to t+T. The probability of one station transmitting during an interval of time T is G = Ns; the probability of the remaining N–1 stations all quiet for an interval of 2T is $(1–s)^{2(N–1)}$. The probability of a successful transmission is thus

$$S = Ns*(1–s)^{2(N–1)}$$

$= G(1–G/N)^{2N}$

Math Warning

Finding the limit of $G(1–G/N)^{2N}$ and finding the maximum of Ge^{-2G} realistically requires a little background in calculus. However, these are not central to applying the model.

As N gets large, the second line approaches Ge^{-2G}. The function $S = G e^{-2G}$ has a maximum at G=1/2, S=1/2e. The rate G=1/2 means that, on average, a transmission is attempted every other slot; this yields the maximum successful-transmission throughput of 1/2e. In other words, at this maximum attempt rate G=1/2, we expect about 2e–1 slot times worth of contention between successful transmissions. What happens to the remaining G–S unsuccessful attempts is not addressed by this model; presumably some higher-level mechanism (*eg* backoff) leads to retransmissions.

A given throughput S<1/2e may be achieved at either of two values for G; that is, a given success rate may be due to a comparable attempt rate or else due to a very high attempt rate with a similarly high failure rate.

2.1.11.2 ALOHA and Ethernet

The relevance of the Aloha model to Ethernet is that during one Ethernet slot time there is no way to detect collisions (they haven't reached the sender yet!) and so the Ethernet contention phase resembles ALOHA with an Aloha slot time T of 51.2 microseconds. Once an Ethernet sender succeeds, however, it continues with a full packet transmission, which is presumably many times longer than T.

The average length of the contention interval, at the maximum throughput calculated above, is 2e–1 slot times (from ALOHA); recall that our model here supposed many senders sending at very low individual rates. This is the minimum contention interval; with lower loads the contention interval is longer due to greater idle times and with higher loads the contention interval is longer due to more collisions.

Finally, let P be the time to send an entire packet in units of T; *ie* the average packet size in units of T. P is thus the length of the "packet" phase in the diagram above. The contention phase has length 2e–1, so the total time to send one packet (contention+packet time) is 2e–1+P. The useful fraction of this is, of course, P, so the effective maximum throughput is P/(2e–1+P).

At 10Mbps, T=51.2 microseconds is 512 bits, or 64 bytes. For P=128 bytes = 2*64, the effective bandwidth becomes 2/(2e-1+2), or 31%. For P=512 bytes=8*64, the effective bandwidth is 8/(2e+7), or 64%. For P=1500 bytes, the model here calculates an effective bandwidth of 80%.

These numbers are quite similar to our earlier values based on a small number of stations sending constantly.

2.2 100 Mbps (Fast) Ethernet

In all the analysis here of 10 Mbps Ethernet, what happens when the bandwidth is increased to 100 Mbps, as is done in the so-called Fast Ethernet standard? If the network physical diameter remains the same, then the round-trip time will be the same in microseconds but will be 10-fold larger measured in bits; this might mean a minimum packet size of 640 bytes instead of 64 bytes. (Actually, the minimum packet size might be somewhat smaller, partly because the "jam signal" doesn't have to become longer, and partly because some of the numbers in the 10 Mbps delay budget above were larger than necessary, but it would still be

large enough that a substantial amount of bandwidth would be consumed by padding.) The designers of Fast Ethernet felt this was impractical.

However, Fast Ethernet was developed at a time (~1995) when reliable switches (below) were widely available; the quote above at *2 Ethernet* from *[MB76]* had become obsolete. Large "virtual" Ethernet networks could be formed by connecting small physical Ethernets with switches, effectively eliminating the need to support large-diameter physical Ethernets. So instead of increasing the minimum packet size, the decision was made to ensure collision detectability by reducing the network diameter instead. The network diameter chosen was a little over 400 meters, with reductions to account for the presence of hubs. At 2.3 meters/bit, 400 meters is 174 bits, for a round-trip of 350 bits.

This 400-meter number, however, may be misleading: by far the most popular Fast Ethernet standard is 100BASE-TX which uses twisted-pair copper wire (so-called Category 5, or better), and in which any individual cable segment is limited to 100 meters. The maximum 100BASE-TX network diameter – allowing for hubs – is just over 200 meters. The 400-meter distance does apply to optical-fiber-based 100BASE-FX in half-duplex mode, but this is not common.

The 100BASE-TX network-diameter limit of 200 meters might seem small; it amounts in many cases to a single hub with multiple 100-meter cable segments radiating from it. In practice, however, such "star" configurations could easily be joined with switches. As we will see below in *2.4 Ethernet Switches*, switches partition an Ethernet into separate "collision domains"; the network-diameter rules apply to each domain separately but not to the aggregated whole. In a fully switched (that is, no hubs) 100BASE-TX LAN, each collision domain is simply a single twisted-pair link, subject to the 100-meter maximum length.

Fast Ethernet also introduced the concept of **full-duplex** Ethernet: two twisted pairs could be used, one for each direction. Full-duplex Ethernet is limited to paths not involving hubs, that is, to single **station-to-station** links, where a station is either a host or a switch. Because such a link has only two potential senders, and each sender has its own transmit line, full-duplex Ethernet is entirely **collision-free**.

Fast Ethernet uses 4B/5B encoding, covered in *4.1.4 4B/5B*. This means that the electronics have to handle 125 Mbps, versus the 200 Mbps if Manchester encoding were still used.

Fast Ethernet 100BASE-TX does not particularly support links between buildings, due to the maximum-cable-length limitation. However, fiber-optic point-to-point links are an effective alternative here, provided full-duplex is used to avoid collisions. We mentioned above that the coax-based 100BASE-FX standard allowed a maximum half-duplex run of 400 meters, but 100BASE-FX is much more likely to use full duplex, where the maximum cable length rises to 2,000 meters.

2.3 Gigabit Ethernet

The problem of scaling Ethernet to handle collision detection gets harder as the transmission rate increases. If we were continue to maintain the same 51.2 µsec slot time but raise the transmission rate to 1000 Mbps, the maximum network diameter would now be 20-40 meters. Instead of that, Gigabit Ethernet moved to a 4096-bit (512-byte, or 4.096 µsec) slot time, at least for the twisted-pair versions. Short frames need to be padded, but this padding is done by the hardware. Gigabit Ethernet 1000Base-T uses so-called PAM-5 encoding, below, which supports a special pad pattern (or symbol) that cannot appear in the data. The hardware pads the frame with these special patterns, and the receiver can thus infer the unpadded length as set by the host operating system.

Gigabit vs Disks

Once a network has reached Gigabit speed, the network is generally as fast as reading from or writing to a disk. Keeping data on another node no longer slows things down. This greatly expands the range of possibilities for constructing things like clustered databases.

However, the Gigabit Ethernet slot time is largely irrelevant, as full-duplex (bidirectional) operation is almost always supported. Combined with the restriction that each length of cable is a station-to-station link (that is, hubs are no longer allowed), this means that collisions simply do not occur and the network diameter is no longer a concern. (10 Gigabit Ethernet has officially abandoned any pretense of supporting collisions; everything *must* be full-duplex.)

There are actually multiple Gigabit Ethernet standards (as there are for Fast Ethernet). The different standards apply to different cabling situations. There are full-duplex optical-fiber formulations good for many miles (*eg* 1000Base-LX10), and even a version with a 25-meter maximum cable length (1000Base-CX), which would in theory make the original 512-bit slot practical.

The most common gigabit Ethernet over copper wire is 1000BASE-T (sometimes incorrectly referred to as 1000BASE-TX. While there exists a TX, it requires Category 6 cable and is thus seldom used; many devices labeled TX are in fact 1000BASE-T). For 1000BASE-T, all four twisted pairs in the cable are used. Each pair transmits at 250 Mbps, and each pair is *bidirectional*, thus supporting full-duplex communication. Bidirectional communication on a single wire pair takes some careful echo cancellation at each end, using a circuit known as a "hybrid" that in effect allows detection of the incoming signal by filtering out the outbound signal.

On any one cable pair, there are five signaling levels. These are used to transmit two-bit **symbols** at a rate of 125 symbols/µsec, for a data rate of 250 bits/µsec. Two-bit symbols in theory only require four signaling levels; the fifth symbol allows for some redundancy which is used for error detection and correction, for avoiding long runs of identical symbols, and for supporting a special pad symbol, as mentioned above. The encoding is known as 5-level pulse-amplitude modulation, or PAM-5. The target bit error rate (BER) for 1000BASE-T is 10^{-10}, meaning that the packet error rate is less than 1 in 10^6.

In developing faster Ethernet speeds, economics plays at least as important a role as technology. As new speeds reach the market, the earliest adopters often must take pains to buy cards, switches and cable known to "work together"; this in effect amounts to installing a proprietary LAN. The real benefit of Ethernet, however, is arguably that it *is* standardized, at least eventually, and thus a site can mix and match its cards and devices. Having a given Ethernet standard support existing cable is even more important economically; the costs of replacing inter-office cable often dwarf the costs of the electronics.

As Ethernet speeds continue to climb, it has become harder and harder for host systems to keep up. As a result, it is common for quite a bit of higher-layer processing to be offloaded onto the Ethernet hardware, for example, TCP checksum calculation. See *12.5 TCP Offloading*.

2.4 Ethernet Switches

Switches join separate physical Ethernets (or sometimes Ethernets and other kinds of networks). A switch has two or more Ethernet interfaces; when a packet is received on one interface it is retransmitted on one

or more other interfaces. Only valid packets are forwarded; collisions are **not** propagated. The term **collision domain** is sometimes used to describe the region of an Ethernet in between switches; a given collision propagates only within its collision domain. All the collision-detection rules, including the rules for maximum network diameter, apply only to collision domains, and not to the larger "virtual Ethernets" created by stringing collision domains together with switches.

Switch Costs

In the 1980's the author once installed a two-port 10-Mbps Ethernet switch (then called a "bridge") that cost $3000; *cf* the *[MB76]* quote at *2 Ethernet*. Today a wide variety of multiport 100-Mbps Ethernet switches are available for around $10, and almost all installed Ethernets are fully switched.

As we shall see below, a switched Ethernet offers much more resistance to eavesdropping than a non-switched (*eg* hub-based) Ethernet.

Like simpler unswitched Ethernets, the topology for a switched Ethernet is in principle required to be loop-free, although in practice, most switches support the spanning-tree loop-detection protocol and algorithm, *2.5 Spanning Tree Algorithm and Redundancy*, which automatically "prunes" the network topology to make it loop-free while allowing the pruned links to be placed back in service if a primary link fails.

While a switch does not propagate collisions, it must maintain a queue for each outbound interface in case it needs to forward a packet at a moment when the interface is busy; on (rare) occasion packets are lost when this queue overflows.

2.4.1 Ethernet Learning Algorithm

Traditional Ethernet switches use datagram forwarding as described in *1.4 Datagram Forwarding*; the trick is to build their forwarding tables without any cooperation from ordinary, non-switch hosts. Indeed, to the extent that a switch is to act as a drop-in replacement for a hub, it cannot count on cooperation from other *switches*.

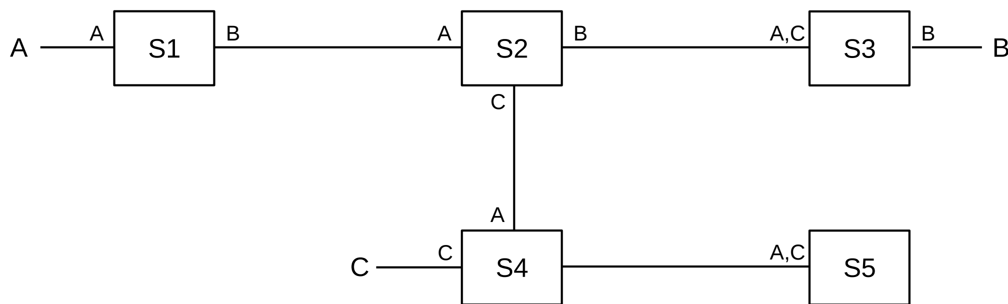
The solution is for the switch to start out with an empty forwarding table, and then incrementally build the table through a **learning** process. If a switch does not have an entry for a particular destination, it will **fall back to flooding**: it will forward the packet out every interface other than the one on which the packet arrived. This is sometimes also called "unknown unicast flooding"; it is equivalent to treating the destination as a broadcast address. The availability of fallback-to-flooding for unknown destinations is what makes it possible for Ethernet switches to learn their forwarding tables without any switch-to-switch or switch-to-host communication or coordination.

A switch learns address locations as follows: for each interface, the switch maintains a table of physical (MAC) addresses that have appeared as *source* addresses in packets arriving via that interface. The switch thus knows that to reach these addresses, if one of them later shows up as a *destination* address, the packet needs to be sent only via that interface. Specifically, when a packet arrives on interface I with source address S and destination unicast address D, the switch enters $\langle$S,I$\rangle$ into its forwarding table.

To actually deliver the packet, the switch also looks up the destination D in the forwarding table. If there is an entry $\langle$D,J$\rangle$ with J$\neq$I – that is, D is known to be reached via interface J – then the switch **forwards** the packet out interface J. If J=I, that is, the packet has arrived on the same interfaces by which the destination is reached, then the packet does not get forwarded at all; it presumably arrived at interface I only because

that interface was connected to a shared Ethernet segment that also either contained D or contained another switch that would bring the packet closer to D. If there is no entry for D, the switch must **flood** the packet out all interfaces J with J≠I; this represents the unknown-destination fallback to flooding. After a short while, the fallback-to-flooding alternative is needed less and less often, as switches learn where the active hosts are located. (However, in some switch implementations, forwarding tables also include timestamps, and entries are removed if they have not been used for, say, five minutes.)

If the destination address D is the broadcast address, or, for many switches, a multicast address, broadcast (flooding) is required. Some switches try to keep track of multicast groups, so as to forward multicast traffic only out interfaces with known subscribers; see *2.1.2 Ethernet Multicast*.



Five learning bridges after three packet transmissions

In the diagram above, each switch's tables are indicated by listing near each interface the destinations (identified by MAC addresses) known to be reachable by that interface. The entries shown are the result of the following packets:

- **A sends to B**; all switches learn where A is
- **B sends to A**; this packet goes directly to A; only S3, S2 and S1 learn where B is
- **C sends to B**; S4 does not know where B is so this packet goes to S5; S2 *does* know where B is so the packet does *not* go to S1.

It is worth observing that, at the application layer, hosts do not commonly identify one another by their MAC addresses. In an IPv4-based network, the use of ARP (*7.9 Address Resolution Protocol: ARP*) to translate from IPv4 to MAC addresses would introduce additional broadcasts, which would cause the above scenario to play out differently. See exercise 11.0.

Switches do *not* automatically discover directly connected neighbors; S1 does not learn about A until A transmits a packet.

Once all the switches have learned where all (or most of) the hosts are, each packet is forwarded rather than flooded. At this point packets are never sent on links unnecessarily; a packet from A to B only travels those links that lie along the (unique) path from A to B. (Paths must be unique because switched Ethernet networks cannot have loops, at least not active ones. If a loop existed, then a packet sent to an unknown destination would be forwarded around the loop endlessly.)

Switches have an additional privacy advantage in that traffic that does not flow where it does not need to flow is much harder to eavesdrop on. On an unswitched Ethernet, one host configured to receive all packets can eavesdrop on all traffic. Early Ethernets were notorious for allowing one unscrupulous station to capture, for instance, all passwords in use on the network. On a fully switched Ethernet, a host physically sees

only the traffic actually addressed to it; other traffic remains inaccessible. This switch-based eavesdropping protection is, however, potentially vulnerable to attackers flooding the network with fake source addresses, forcing switches into fallback-to-flooding mode.

CAM Table

On Cisco switches, the forwarding table is often called the CAM table, after the specialized high-speed content-addressable memory used to store it.

Typical large switches have room for a forwarding table with 10^4 - 10^5 entries, though fully switched networks at the upper end of this size range are not common. The main size limitations specific to switching are the requirement that the topology must be loop-free (thus disallowing duplicate paths which might otherwise provide redundancy), and that all broadcast traffic must always be forwarded everywhere. As a switched Ethernet grows, broadcast traffic comprises a larger and larger percentage of the total traffic, and the organization must at some point move to a routing architecture (*eg* as in *7.6 IPv4 Subnets*). A common recommendation is to have no more than 1000 hosts per LAN (or VLAN, *2.6 Virtual LAN (VLAN)*).

One of the differences between an inexpensive Ethernet switch and a pricier one is the degree of internal parallelism it can support. If three packets arrive simultaneously on ports 1, 2 and 3, and are destined for respective ports 4, 5 and 6, can the switch actually transmit the packets simultaneously? A simple switch likely has a single CPU and a single memory bus, both of which can introduce transmission bottlenecks. For commodity five-port switches, at most two simultaneous transmissions can occur; such switches can generally handle that degree of parallelism. It becomes harder as the number of ports increases, but at some point the need to support full parallel operation can be questioned; in many settings the majority of traffic involves one or two server or router ports. If a high degree of parallelism is in fact required, there are various architectures – known as **switch fabrics** – that can be used; these typically involve multiple simple processor elements.

2.5 Spanning Tree Algorithm and Redundancy

In theory, if you form a loop with Ethernet switches, any packet with destination not already present in the forwarding tables will circulate endlessly; some early switches would actually do this.

In practice, however, loops allow redundancy – if one link breaks there is still 100% connectivity – and so are desirable. As a result, Ethernet switches have incorporated a switch-to-switch protocol to construct a subset of the switch-connections graph that has no loops and yet allows reachability of every host, known in graph theory as a **spanning tree**. Once the spanning tree is built, links that are not part of the tree are disabled, even if they would represent the most efficient path between two nodes. If a link that is part of the spanning tree fails, partitioning the network, a new tree is constructed, and some formerly disabled links may now return to service.

One might ask, if switches can work together to negotiate a a spanning tree, whether they can also work together to negotiate loop-free forwarding tables for the original non-tree topology, thus keeping all links active. The difficulty here is not the switches' ability to coordinate, but the underlying Ethernet broadcast feature. As long as the topology has loops and broadcast is enabled, broadcast packets might circulate forever. And disabling broadcast is not a straightforward option; switches rely on the broadcast-based fallback-to-flooding strategy of *2.4.1 Ethernet Learning Algorithm* to deliver to unknown destinations.

However, we will return to this point in *2.7 Software-Defined Networking*. See also exercise 10.0.

The presence of hubs and other unswitched Ethernet **segments** slightly complicates the switch-connections graph. In the absence of these, the graph's nodes and edges are simply the hosts (including switches) and links of the Ethernet itself. If unswitched multi-host Ethernet segments are present, then each of these becomes a single node in the graph, with a graph edge to each switch to which it directly connects. (Any Ethernet switches not participating in the spanning-tree algorithm would be treated as hubs.)

Every switch has an ID, *eg* its smallest Ethernet address, and every edge that attaches to a switch does so via a particular, numbered interface. The goal is to disable redundant (cyclical) paths while retaining the ability to deliver to any segment. The algorithm is due to Radia Perlman, *[RP85]*.

The switches first elect a root node, *eg* the one with the smallest ID. Then, if a given segment connects to two switches that both connect to the root node, the switch with the shorter path to the root is used, if possible; in the event of ties, the switch with the smaller ID is used. The simplest measure of path cost is the number of hops, though current implementations generally use a cost factor inversely proportional to the bandwidth (so larger bandwidth has lower cost). Some switches permit other configuration here. The process is dynamic, so if an outage occurs then the spanning tree is recomputed. If the outage should partition the network into two pieces, both pieces will build spanning trees.

All switches send out regular messages on all interfaces called *bridge protocol data units*, or BPDUs (or "Hello" messages). These are sent to the Ethernet multicast address 01:80:c2:00:00:00, from the Ethernet physical address of the interface. (Note that Ethernet switches do not otherwise need a unique physical address for each interface.) The BPDUs contain

- The switch ID
- the ID of the node the switch believes is the root
- the path cost (*eg* number of hops) to that root

These messages are recognized by switches and are not forwarded naively. Switches process each message, looking for

- a switch with a lower ID than any the receiving switch has seen before (thus becoming the new root)
- a shorter path to the existing root
- an equal-length path to the existing root, but via a neighbor switch with a lower ID (the tie-breaker rule). If there are two ports that connect to that switch, the port number is used as an additional tie-breaker.

In a heterogeneous Ethernet we would also introduce a preference for *faster* paths, but we will assume here that all links have the same bandwidth.

When a switch sees a new root candidate, it sends BPDUs on all interfaces, indicating the distance. The switch includes the interface leading towards the root.

Once this process has stabilized, each switch knows

- its own path to the root
- which of its ports any further-out switches will be using to reach the root
- for each port, its directly connected neighboring switches

Now the switch can "prune" some (or all!) of its interfaces. It disables all interfaces that are not *en*abled by the following rules:

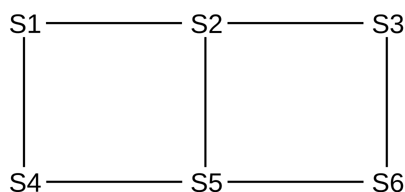
1. It enables the port via which it reaches the root
2. It enables any of its ports that further-out switches use to reach the root
3. If a remaining port connects to a segment to which other "segment-neighbor" switches connect as well, the port is enabled if the switch has the minimum cost to the root among those segment-neighbors, or, if a tie, the smallest ID among those neighbors, or, if two ports are tied, the port with the smaller ID.
4. If a port has no directly connected switch-neighbors, it presumably connects to a host or segment, and the port is enabled.

Rules 1 and 2 construct the spanning tree; if S3 reaches the root via S2, then Rule 1 makes sure S3's port towards S2 is open, and Rule 2 makes sure S2's corresponding port towards S3 is open. Rule 3 ensures that each network segment that connects to multiple switches gets a unique path to the root: if S2 and S3 are segment-neighbors each connected to segment N, then S2 enables its port to N and S3 does not (because 2<3). The primary concern here is to create a path for any *host* nodes on segment N; S2 and S3 will create their own paths via Rules 1 and 2. Rule 4 ensures that any "stub" segments retain connectivity; these would include all hosts directly connected to switch ports.

2.5.1 Example 1: Switches Only

We can simplify the situation somewhat if we assume that the network is **fully switched**: each switch port connects to another switch or to a (single-interface) host; that is, no repeater hubs (or coax segments!) are in use. In this case we can dispense with Rule 3 entirely.

Any switch ports directly connected to a host can be identified because they are "silent"; the switch never receives any BPDU messages on these interfaces because hosts do not send these. All these host port ends up enabled via Rule 4. Here is our sample network, where the switch numbers (*eg* 5 for S5) represent their IDs; no hosts are shown and interface numbers are omitted.



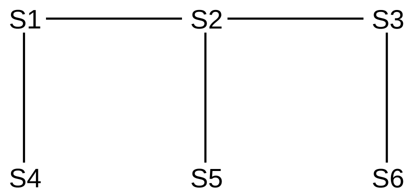
S1 has the lowest ID, and so becomes the root. S2 and S4 are directly connected, so they will enable the interfaces by which they reach S1 (Rule 1) while S1 will enable its interfaces by which S2 and S4 reach it (Rule 2).

S3 has a unique lowest-cost route to S1, and so again by Rule 1 it will enable its interface to S2, while by Rule 2 S2 will enable its interface to S3.

S5 has two choices; it hears of equal-cost paths to the root from both S2 and S4. It picks the lower-numbered neighbor S2; the interface to S4 will never be enabled. Similarly, S4 will never enable its interface to S5.

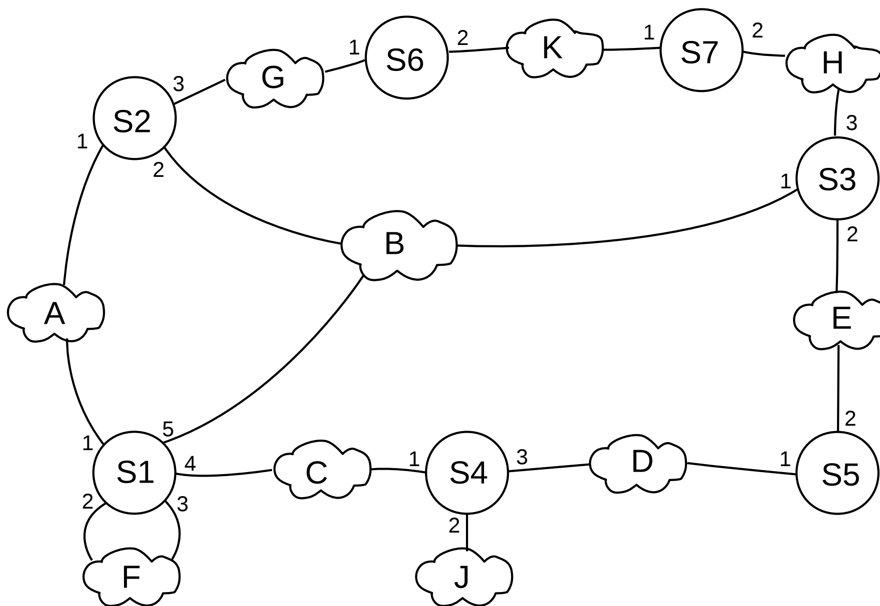
Similarly, S6 has two choices; it selects S3.

After these links are enabled (strictly speaking it is interfaces that are enabled, not links, but in all cases here either both interfaces of a link will be enabled or neither), the network in effect becomes:



2.5.2 Example 2: Switches and Segments

As an example involving switches that may join via unswitched Ethernet segments, consider the following network; S1, S2 and S3, for example, are all segment-neighbors via their common segment B. As before, the switch numbers represent their IDs. The letters in the clouds represent network segments; these clouds may include multiple hosts. Note that switches have no way to detect these hosts; only (as above) other switches.



Eventually, all switches discover S1 is the root (because 1 is the smallest of {1,2,3,4,5,6}). S2, S3 and S4 are one (unique) hop away; S5, S6 and S7 are two hops away.

Algorhyme

I think that I shall never see
a graph more lovely than a tree.
A tree whose crucial property
is loop-free connectivity.
A tree that must be sure to span
so packet can reach every LAN.
First, the root must be selected.
By ID, it is elected.
Least-cost paths from root are traced.
In the tree, these paths are placed.
A mesh is made by folks like me,
then bridges find a spanning tree.

Radia Perlman

For the switches one hop from the root, Rule 1 enables S2's port 1, S3's port 1, and S4's port 1. Rule 2 enables the corresponding ports on S1: ports 1, 5 and 4 respectively. Without the spanning-tree algorithm S2 could reach S1 via port 2 as well as port 1, but port 1 has a smaller number.

S5 has two equal-cost paths to the root: S5⟶S4⟶S1 and S5⟶S3⟶S1. S3 is the switch with the lower ID; its port 2 is enabled and S5 port 2 is enabled.

S6 and S7 reach the root through S2 and S3 respectively; we enable S6 port 1, S2 port 3, S7 port 2 and S3 port 3.

The ports still disabled at this point are S1 ports 2 and 3, S2 port 2, S4 ports 2 and 3, S5 port 1, S6 port 2 and S7 port 1.

Now we get to Rule 3, dealing with how segments (and thus their hosts) connect to the root. Applying Rule 3,

- We do not enable S2 port 2, because the network (B) has a direct connection to the root, S1
- We do enable S4 port 3, because S4 and S5 connect that way and S4 is closer to the root. This enables connectivity of network D. We do not enable S5 port 1.
- S6 and S7 are tied for the path-length to the root. But S6 has smaller ID, so it enables port 2. S7's port 1 is not enabled.

Finally, Rule 4 enables S4 port 2, and thus connectivity for host J. It also enables S1 port 2; network F has two connections to S1 and port 2 is the lower-numbered connection.

All this port-enabling is done using only the data collected during the root-discovery phase; there is no additional negotiation. The BPDU exchanges continue, however, so as to detect any changes in the topology.

If a link is disabled, it is not used even in cases where it would be more efficient to use it. That is, traffic from F to B is sent via B1, D, and B5; it never goes through B7. IP routing, on the other hand, uses the

"shortest path". To put it another way, all spanning-tree Ethernet traffic goes through the root node, or along a path to or from the root node.

The traditional (IEEE 802.1D) spanning-tree protocol is relatively slow; the need to go through the tree-building phase means that after switches are first turned on no normal traffic can be forwarded for ~30 seconds. Faster, revised protocols have been proposed to reduce this problem.

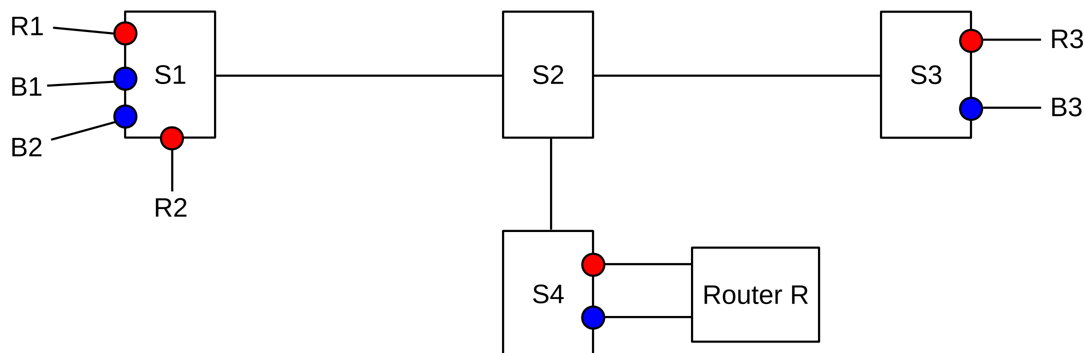
Another issue with the spanning-tree algorithm is that a rogue switch can announce an ID of 0, thus likely becoming the new root; this leaves that switch well-positioned to eavesdrop on a considerable fraction of the traffic. One of the goals of the Cisco "Root Guard" feature is to prevent this. Another goal of this and related features is to put the spanning-tree topology under some degree of administrative control. One likely wants the root switch, for example, to be geographically at least somewhat centered, and for the high-speed backbone links to be preferred to slow links.

2.6 Virtual LAN (VLAN)

What do you do when you have different people in different places who are "logically" tied together? For example, for a while the Loyola University CS department was split, due to construction, between two buildings.

One approach is to continue to keep LANs local, and use IP routing between different subnets. However, it is often convenient (printers are one reason) to configure workgroups onto a single "virtual" LAN, or **VLAN**. A VLAN looks like a single LAN, usually a single Ethernet LAN, in that all VLAN members will see broadcast packets sent by other members and the VLAN will ultimately be considered to be a single IP *subnet* (*7.6 IPv4 Subnets*). Different VLANs are ultimately connected together, but likely only by passing through a single, central IP router.

VLANs can be visualized and designed by using the concept of coloring. We logically assign all nodes on the same VLAN the same color, and switches forward packets accordingly. That is, if S1 connects to red machines R1 and R2 and blue machines B1 and B2, and R1 sends a broadcast packet, then it goes to R2 but not to B1 or B2. Switches must, of course, be told the color of each of their ports.



One network of switches S1-S4 divided into two VLANs, red and blue

In the diagram above, S1 and S3 each have both red and blue ports. The switch network S1-S4 will deliver traffic only when the source and destination ports are the same color. Red packets can be forwarded to the

blue VLAN *only* by passing through the router R, entering R's red port and leaving its blue port. R may apply firewall rules to restrict red–blue traffic.

When the source and destination ports are on the same switch, nothing needs to be added to the packet; the switch can keep track of the color of each of its ports. However, switch-to-switch traffic must be additionally tagged to indicate the source. Consider, for example, switch S1 above sending packets to S3 which has nodes R3 (red) and B3 (blue). Traffic between S1 and S3 must be tagged with the color, so that S3 will know to what ports it may be delivered. The IEEE **802.1Q** protocol is typically used for this packet-tagging; a 32-bit "color" tag is inserted into the Ethernet header after the source address and before the type field. The first 16 bits of this field is 0x8100, which becomes the new Ethernet type field and which identifies the frame as tagged.

Double-tagging is possible; this would allow an ISP to have one level of tagging and its customers to have another level.

2.7 Software-Defined Networking

The Spanning-Tree mechanism has been effective in allowing large Ethernets to contain redundant "backup" links, normally "suspended" but which can be returned to service promptly if any of the "primary" links should fail. But what about the possibility of actually making first-class use of *all* links? How can we avoid disabling links? In a high-performance datacenter, disabled links are a wasted resource.

As we indicated earlier, the central difficulty here is in forwarding packets to unknown destinations: Ethernet switches traditionally rely on fallback-to-flooding here, and safely using flooding (broadcast) in the presence of a loop topology is quite difficult (see, for example, "reliable flooding" in *9.5 Link-State Routing-Update Algorithm*).

The strategy of Software-Defined Networking, or **SDN**, is to move away from the traditional distributed Ethernet learning algorithm, and instead to place the forwarding mechanism of each participating switch under the aegis of a **controller**, in such a way that forwarding and redundancy can coexist. The controller can be a single node on the network, or can be a distributed set of nodes. The controller then manages the forwarding tables of the switches.

To handle legitimate broadcast traffic, the controller can, at startup, probe the switches to determine their layout, and, from this, construct a suitable spanning tree. The switches can then be instructed to forward broadcast traffic only along the links of this spanning tree. Links that are not part of the spanning tree can still be used for forwarding to known destinations, however, unlike conventional switches using the spanning tree algorithm.

Typically, if a switch sees a packet addressed to an unknown destination, it reports it to the controller, which then must figure out what to do next. One option is to have traffic to unknown destinations forwarded along the same spanning tree used for broadcast traffic. This allows fallback-to-flooding to coexist safely with the full use of loop topologies.

Switches are sometimes configured to report new *source* addresses to the controller, so that the controller can tell all the other switches the best route to that new source.

SDN controllers can be configured as simple firewalls, disallowing forwarding between selected pairs of nodes for security reasons. For example, if a data center has customers A and B, each with multiple nodes,

then it is possible to configure the network so that no node belonging to customer A can send packets to a node belonging to customer B. See also the following section.

At many sites, the SDN implementation is based on standardized modules. However, controller software can also be developed locally, allowing very local control of network functionality. This control, rather than the ability to combine loop topologies with Ethernet, is arguably SDN's most important feature. See *[FRZ13]*.

2.7.1 OpenFlow Switches

At the heart of SDN is the ability of controllers to tell switches how to forward packets. We next look at the packet-forwarding architecture for **OpenFlow** switches; OpenFlow is a specific SDN standard created by the Open Networking Foundation. See *[MABPPRST08]* and the OpenFlow switch specification (2015 version).

OpenFlow forwarding is built around one or more **flow tables**. The primary components of a flow-table entry are a set of **match fields** and a set of packet-response instructions, or **actions**, if the match succeeds. T Some common forms for the latter include

- drop the packet
- forward the packet out a specified single interface
- flood the packet out a *set* of interfaces
- forward the packet to the controller
- modify some field of the packet
- match the packet at another (higher-numbered) flow table

The match fields *can*, of course, be a single entry for the destination Ethernet address. But it can also include any other packet bit-field, and can include the ingress interface number. For example, the forwarding can be done entirely (or partially) on IP addresses rather than Ethernet addresses, thus allowing the OpenFlow switch to act as a so-called Layer 3 switch (*7.6.3 Subnets versus Switching*), that is, resembling an IP router. Matching can be by the destination IP address and the destination TCP port, allowing separate forwarding for different TCP protocols. In *9.6 Routing on Other Attributes* we define *policy-based routing*; arbitrary such routing decisions can be implemented using OpenFlow switches. In SDN settings the policy-based-routing abilities are sometimes used to segregate real-time traffic and large-volume "elephant" flows. In the `l2_pairs.py` example of the following section, matching is done on both Ethernet source and destination addresses.

Flow tables bear a rough similarity to forwarding tables, with the match fields corresponding to destinations and the actions corresponding to the next_hop. In simple cases, the match field contains a single destination address and the action is to forward out the corresponding switch port.

Normally, OpenFlow switches handle broadcast packets by flooding them; that is, by forwarding them out all interfaces other than the arrival interface. It is possible, however, to set the NO_FLOOD attribute on specific interfaces, which means that packets designated for flooding (broadcast or otherwise) will not be sent out on those interfaces. This is typically how spanning trees for broadcast traffic are implemented (see *18.9.6 l2_multi.py* for a Mininet example). An interface marked NO_FLOOD, however, may still be used for unicast traffic. Generally, broadcast flooding does not require a flow-table entry.

Match fields are also assigned a **priority** value. In the event that a packet matches two or more flow-table entries, the entry with the highest priority wins. The **table-miss** entry is the entry with no match fields (thereby matching every packet) and with priority 0. Often the table-miss entry's action is to forward the packet to the controller, although a packet that matches no entry is simply dropped.

Flow-table instructions can also involve modifying ("mangling") packets. One Ethernet-layer application might be VLAN coloring (*2.6 Virtual LAN (VLAN)*); at the IPv4 layer, this could be used to decrement the TTL and update the checksum (*7.1 The IPv4 Header*).

In addition to match fields and instructions, flow tables also include counters, flags, and a last_used time. The latter allows flows to be removed if no matching packets have been seen for a while. The counters allow the OpenFlow switch to implement Quality-of-Service constraints – *eg* bandwidth limiting – on the traffic.

2.7.2 Learning Switches in OpenFlow

Suppose we want to implement a standard Ethernet learning switch (*2.4.1 Ethernet Learning Algorithm*). The obvious approach is to use flows matching only on the destination address. But we encounter a problem because, by default, packets are reported to the controller only when there is no flow-entry match. Suppose switch S sees a packet from host B to host A and reports it to the controller, which installs a flow entry in S matching destination B (much as a real learning switch would do). If a packet now arrives at S from a third host C to B, it would simply be forwarded, as it would match the B flow entry, and therefore would not be reported to the controller. This means the controller would never learn about address C, and would never install a flow entry for C.

One straightforward alternative approach that avoids this problem is to match on Ethernet ⟨destaddr,srcaddr⟩ **pairs**. If a packet from A to B arrives at switch S and does not match any existing flow entry at S, it is reported to the controller, which now learns that A is reached via the port by which the packet arrived at S.

In the network above, suppose A sends a packet to B (or broadcasts a packet meant for B), and the flow table of S is empty. S will report the packet to the controller (not shown in the diagram), which will send it back to S to be flooded. However, the controller will also record that A can be reached from S via port 1.

Next, suppose B responds. When this packet from B arrives at S, there are still no flow-table entries, and so S again reports the packet to the controller. But this time the controller knows, because it learned from the first packet, that S can reach A via port 1. The controller also now knows, from the just-arrived packet, that B can be reached via port 2. Knowing both of these forwarding rules, the controller now installs two flow-table entries in S:

> dst=B,src=A: forward out port 2
>
> dst=A,src=B: forward out port 1

If a packet from a third host **C** now arrives at S, addressed to B, it will not be forwarded, even though its destination address matches the first rule above, as its source address does not match A. It will instead be sent to the controller (and ultimately be flooded). When B responds to C, the controller will install rules for dst=C,src=B and dst=B,src=C. If the packet from C were not reported to the controller – perhaps because

S had a flow rule for dst=B only – then the controller would never learn about C, and would never be in a position to install a flow rule for reaching C.

The pairs approach to OpenFlow learning is pretty much optimal, if a single flow-entry table is available. The problem with this approach is that it does not scale well; with 10,000 addresses on the network, we will need 100,000,000 flowtable-entry pairs to describe all the possible forwarding. This is prohibitive.

We examine a real implementation (in Python) of the pairs approach in *18.9.2 l2_pairs.py*, using the Mininet network emulator and the Pox controller (*18 Mininet*).

A more compact approach is to use **multiple flow tables**: one for matching the destination, and one for matching the source. In this version, the controller never has to remember partial forwarding information, as the controller in the version above had to do after receiving the first packet from A. When a packet arrives, it is matched against the first table, and any actions specified by the match are carried out. One of the actions may be a request to repeat the match against the second table, which may lead to a second set of actions. We repeat the A→B, B→A example above, letting T_0 denote the first table and T_1 denote the second.

Initially, before any packets are seen, the controller installs the following low-priority match rules in S:

T_0: match nothing: flood, send to T_1

T_1: match nothing: send to controller

These are in effect *default* rules: because there are no packet fields to match, they match all packets. The low priority ensures that better-matching rules are always used when available.

When the packet from A to B arrives, the T_0 rule above means the packet is flooded to B, while the T_1 rule means the packet is sent to the controller. The controller then installs the following rules in S:

T_0: match dst=A: forward via port 1, send to T_1

T_1: match src=A: do nothing

Now B sends its reply to A. The first rule above matches, so the packet is forwarded by S to A, and is resubmitted to T_1. The T_1 rule immediately above, however, does *not* match. The only match is to the original default rule, and the packet is sent to the controller. The controller then installs another two rules in S:

T_0: match dst=B: forward via port 2, send to T_1

T_1: match src=B: do nothing

At this point, as A and B continue to communicate, the T_0 rules ensure proper forwarding, while the T_1 rules ensure that no more packets from this flow are sent to the controller.

Note that the controller always installs the same address in the T_0 table and the T_1 table, so the list of addresses present in these two tables will always be identical. The T_0 table always matches destinations, though, while the T_1 table matches source addresses.

The Mininet/Pox version of this appears in *18.9.3 l2_nx.py*.

Another application for multiple flow tables involves switches that make quality-of-service prioritization decisions. A packet's destination would be found using the first flow table, while its priority would be found by matching to the second table. The packet would then be forwarded out the port determined by the first table, using the priority determined by the second table. Like building a learning switch, this can be done with a single table by listing all combinations of ⟨destaddr,priority⟩, but sometimes that's too many entries.

We mentioned above that SDN controllers can be used as firewalls. At the Ethernet-address level this is tedious to configure, but by taking advantage of OpenFlow's understanding of IP addresses, it is straightforward, for example, to block traffic between different IP subnets, much like a router might do. OpenFlow also allows blocking all such traffic *except* that between specific pairs of hosts using specific protocols. For example, we might want customer A's web servers to be able to communicate with A's database servers using the standard TCP port, while still blocking all other web-to-database traffic.

2.7.3 Other OpenFlow examples

After emulating a learning switch, perhaps the next most straightforward OpenFlow application, conceptually, is the support of Ethernet topologies that contain loops. This can be done quite generically; the controller does not need any special awareness of the network topology.

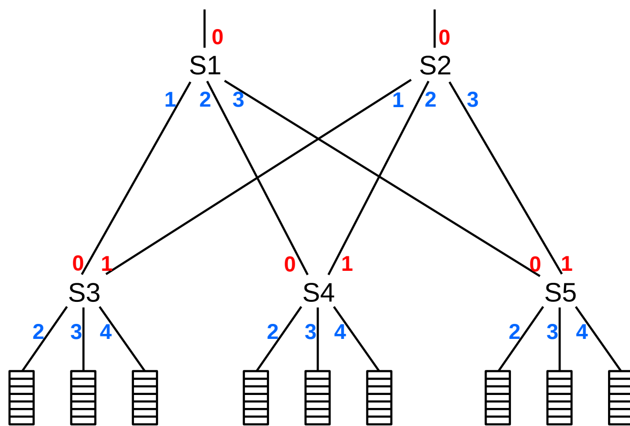
On startup, switches are instructed by the controller to report their neighboring switches. With this information the controller is then able to form a complete map of the switch topology. (One way to implement this is for the controller to order each switch to send a special marked packet out each of its ports, and then for the receiving switches to report these packets back to the controller.) Once the controller knows the switch topology, it can calculate a spanning tree, and then instruct each switch that flooded packets should be sent out only via ports that correspond to links on the spanning tree.

Once the location of a destination host has been learned by the controller (that is, the controller learns which switch the host is directly connected to), the controller calculates the shortest (or lowest-cost, if the application supports differential link costs) path from each switch to that host. The controller then instructs each switch how to forward to that host. Forwarding will likely use links that are not part of the spanning tree, unlike traditional Ethernet switches.

We outline an implementation of this strategy in *18.9.6 l2_multi.py*.

2.7.3.1 Interconnection Fabric

The previous Ethernet-loop example is quite general; it works for any switch topology. Many other OpenFlow applications require that the controller contains some prior knowledge of the switch topology. As an example of this, we present what we will refer to as an **interconnection fabric**. This is the S1-S5 network illustrated below, in which every upper (S1-S2) switch connects directly to every lower (S3-S5) switch. The bottom row in the diagram represents server racks, as interconnection fabrics are very common in datacenters. (For a real-world datacenter example, see here, although real-world interconnection fabrics are often joined using routing rather than switching.) The red and blue numbers identify the switch ports.



The first two rows here contain many loops, eg S1–S3–S2–S4–S1 (omitting the S3-S5 row, and having S1 and S2 connect directly to the server racks, does not eliminate loops). In the previous example we described how we could handle loops in a switched network by computing a spanning tree and then allowing packet flooding only along this spanning tree. This is certainly possible here, but if we allow the spanning-tree algorithm to prune the loops, we will lose most of the parallelism between the S1-S2 and S3-S5 layers; see exercise 8.5. This generic spanning-tree approach is not what we want.

If we use IP routing at S1 through S5, as in *9 Routing-Update Algorithms*, we then need the three clusters of server racks below S3, S4 and S5 to be on three *separate* IP subnets (*7.6 IPv4 Subnets*). While this is always technically possible, it can be awkward, if the separate subnets function for most other purposes as a single unit.

One OpenFlow approach is to assume that the three clusters of server racks below S3-S4-S5 form a *single* IP subnet. We can then configure S1-S5 with OpenFlow so that traffic *from* the subnet is always forwarded upwards while traffic *to* the subnet is always forwarded downwards.

But an even simpler solution – one not requiring any knowledge of the server subnet – is to use OpenFlow to configure the switches S1-S5 so that unknown-destination traffic entering on a red (upper) port is flooded out only on the blue (lower) ports, and vice-versa. This eliminates loops by ensuring that all traffic goes through the interconnection fabric either upwards-only or downwards-only. After the destination server below S3-S5 has replied, of course, S1 or S2 will learn to which of S3-S5 it should forward future packets to that server.

This example works the way it does because the topology has a particular property: once we eliminate paths that both enter and leave S1 or S2 via blue nodes, or that enter and leave S3, S4 and S5 via red nodes, there is a unique path between any input port (red upper port) and any output port (towards the server racks). From there, it is easy to avoid loops. Given a more general topology, on the other hand, in which unique paths *cannot* be guaranteed by such a rule, the OpenFlow controller has to *choose* a path. This in turn generally entails path discovery, shortest-path selection and loop avoidance, as in the previous example.

2.7.3.2 Load Balancer

The previous example was quite *local*, in that all the OpenFlow actions are contained within the interconnection fabric. As a larger-scale (and possiby more typical) special-purpose OpenFlow example, we next describe how to achieve **server load-balancing** via SDN; that is, users are connected transparently to one of

several identical servers. Each server handles only its assigned fraction of the total load. For this example, the controller must not only have knowledge of the topology, but also of the implementation goal.

To build the load-balancer, imagine that the SDN controller is in charge of, in the diagram of the previous section, all switches in the interconnection fabric and also all switches among the server racks below. At this point, we configure all the frontline servers within the server racks identically, including *giving them all identical IPv4 addresses*. When an incoming TCP connection request arrives, the controller picks a server (perhaps using round robin, perhaps selecting the server with the lowest load) and sets up OpenFlow forwarding rules so all traffic on that TCP connection arriving from the outside world is sent to the designated server, and vice-versa. Different servers with the same IPv4 address are not allowed to talk directly with one another at all, thereby averting chaos. The lifetime of the OpenFlow forwarding rule can be adjusted as desired, *eg* to match the typical lifetime of a user session.

When the first TCP packet of a connection arrives at the selected server, the server may or may not need to use ARP to figure out the appropriate internal LAN address to which to send its reply. Sometimes ARP needs to be massaged a bit to work acceptably in an environment in which some hosts have the same IPv4 address.

At no point is the fact that multiple servers have been assigned the same IPv4 address directly exposed: not to other servers, not to internal routers, and not to end users. (Servers never initiate outbound connections to users, so there is no risk of two servers contacting the same user.)

For an example of this sort of load balancing implemented in Mininet and Pox, see *18.9.5 loadbalance31.py*.

The identical frontline servers might need to access a common internal database cluster. This can be implemented by assigning each server a second IPv4 address for this purpose, not shared with other servers, or by using the common public-facing IPv4 address and a little more OpenFlow cleverness in setting up appropriate forwarding rules. If the latter approach is taken, it is now in principle possible that two servers would connect to the database using the same TCP port, by coincidence. This *would* expose the identical IPv4 addresses, and the SDN controllers would have to take care to ensure that this did not happen. One approach, if supported, would be to have the OpenFlow switches "mangle" the server IPv4 addresses or ports, as is done with NAT (*7.7 Network Address Translation*).

There are also several "traditional" strategies for implementing load balancing. For example, one can give each server its own IPv4 address but then use round-robin DNS (*7.8 DNS*) to assign different users to different servers. Alternatively, one can place a device called a *load balancer* at the front of the network that assigns incoming connection requests to an internal server and then takes care of setting up the appropriate forwarding. Forwarding can be at the IP layer (that is, via routing), or at the TCP layer, or at the application layer. The load balancer can be thought of as NAT-like (*7.7 Network Address Translation*) in that it maintains a table of associations between external-user connections and a internal servers; once a user connects, the association with the chosen server remains in place for a period of time. One advantage of the SDN approach described here is that the individual front-line servers need no special configuration; all of the load-sharing awareness is contained within the SDN network. Furthermore, the SDN switches do virtually no additional work beyond ordinary forwarding; they need only involve the controller when the first new TCP packet of each connection arrives.

2.8 Epilog

Ethernet dominates the LAN layer, but is not one single LAN protocol: it comes in a variety of speeds and flavors. Higher-speed Ethernet seems to be moving towards fragmenting into a range of physical-layer options for different types of cable, but all based on switches and point-to-point linking; different Ethernet types can be interconnected only with switches. Once Ethernet finally abandons physical links that are bi-directional (half-duplex links), it will be collision-free and thus will no longer need a minimum packet size.

Other wired networks have largely disappeared (or have been renamed "Ethernet"). Wireless networks, however, are here to stay, and for the time being at least have inherited the original Ethernet's collision-management concerns.

2.9 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercise 2.5 is distinct, for example, from exercises 2.0 and 3.0. Exercises marked with a ♢ have solutions or hints at 24.2 Solutions for Ethernet.

1.0. Simulate the contention period of five Ethernet stations that all attempt to transmit at T=0 (presumably when some sixth station has finished transmitting), in the style of the diagram in *2.1.6 Exponential Backoff Algorithm*. Assume that time is measured in slot times, and that exactly one slot time is needed to detect a collision (so that if two stations transmit at T=1 and collide, and one of them chooses a backoff time k=0, then that station will transmit again at T=2). Use coin flips or some other source of randomness.

2.0. Suppose we have Ethernet switches S1 through S3 arranged as below; each switch uses the learning algorithm of *2.4 Ethernet Switches*. All forwarding tables are initially empty.

(a). If A sends to B, which switches see this packet?

(b). If B then replies to A, which switches see this packet?

(c). If C then sends to B, which switches see this packet?

(d). If C then sends to D, which switches see this packet?

2.7.♢ Suppose we have the Ethernet switches S1 through S4 arranged as below. All forwarding tables are empty; each switch uses the learning algorithm of *2.4 Ethernet Switches*.

Now suppose the following packet transmissions take place:

- A sends to D
- D sends to A
- A sends to B
- B sends to D

For each switch S1-S4, list what source addresses (*eg* A,B,C,D) it has seen (and thus what nodes it has learned the location of).

3.0. Repeat the previous exercise (2.7), with the same network layout, except that instead the following packet transmissions take place:

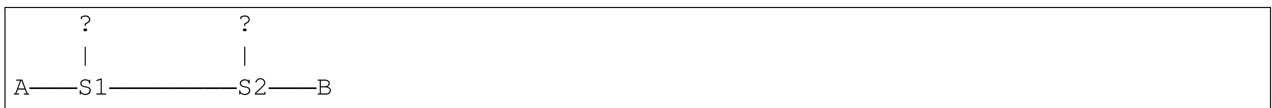
- A sends to B
- B sends to A
- C sends to B
- D sends to A

For each switch, list what source addresses (*eg* A,B,C,D) it has seen (and thus what nodes it has learned the location of).

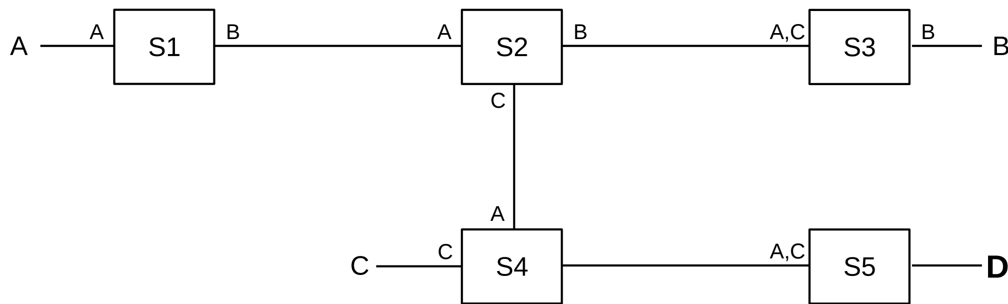
4.0. In the switched-Ethernet network below, find two packet transmissions so that, when a third transmission A⟶D occurs, the packet *is* seen by B (that is, it is flooded out all ports by S2), but is *not* similarly seen by C (because it is forwarded to D, not flooded, by S3). All forwarding tables are initially empty, and each switch uses the learning algorithm of *2.4 Ethernet Switches*.

Hint: Destination D must be in S3's forwarding table, but must not be in S2's. So there must have been a packet sent by D that was seen by S3 but not by S2.

5.0. Given the Ethernet network with learning switches below, with (disjoint) unspecified parts represented by ?, explain why it is impossible for a packet sent from A to B to be forwarded by S1 directly to S2, but to be flooded by S2 out all of S2's other ports.

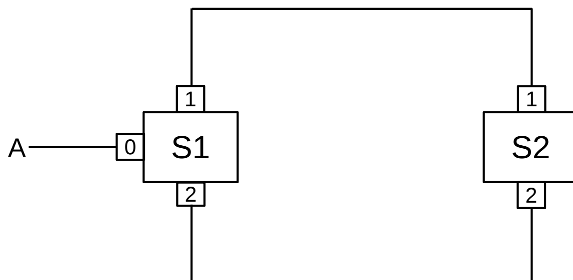


6.0. In the diagram below, from *2.4.1 Ethernet Learning Algorithm*, suppose node D is connected to S5. Now, with the tables as shown by the labels in the diagram (that is, S5 knows about A and C, *etc*), D sends to B.



Which switches will see this D→B packet, and thus learn about D? Of these switches, which do *not* already know where B is and will use fallback-to-flooding?

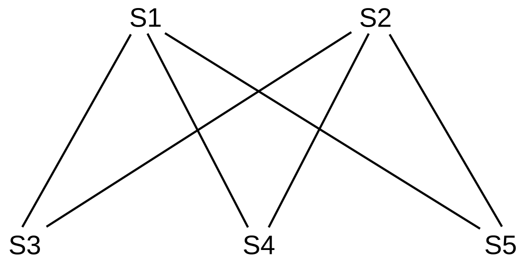
7.0. Suppose two Ethernet switches are connected in a loop as follows; S1 and S2 have their interfaces 1 and 2 labeled. These switches do *not* use the spanning-tree algorithm.



Suppose A attempts to send a packet to destination B, which is unknown. S1 will therefore flood the packet out interfaces 1 and 2. What happens then? How long will A's packet circulate?

8.0. The following network is like that of *2.5.1 Example 1: Switches Only*, except that the switches are numbered differently. Again, the ID of switch Sn is n, so S1 will be the root. Which links end up "pruned" by the spanning-tree algorithm, and why? Diagram the network formed by the surviving links.

8.5. Consider the network below, consisting of just the first two rows from the datacenter diagram in *2.7.1 OpenFlow Switches*:



(a).◊ Give network of surviving links after application of the spanning-tree algorithm. Assume the ID of switch Sn is n. In this network, what is the path of traffic from S2 to S5?

(b). Do the same as part (a) except assuming S4 has ID 0, and so will be the root, while the ID for the other Sn remains n. What will be the path of traffic from S1 to S5?

9.0. Suppose you want to develop a new protocol so that Ethernet switches participating in a VLAN all keep track of the VLAN "color" associated with every destination in their forwarding tables. Assume that each switch knows which of its ports (interfaces) connect to other switches and which may connect to hosts, and in the latter case knows the color assigned to that port.

(a). Suggest a way by which switches might propagate this destination-color information to other switches.

(b). What must be done if a port formerly reserved for connection to another switch is now used for a host?

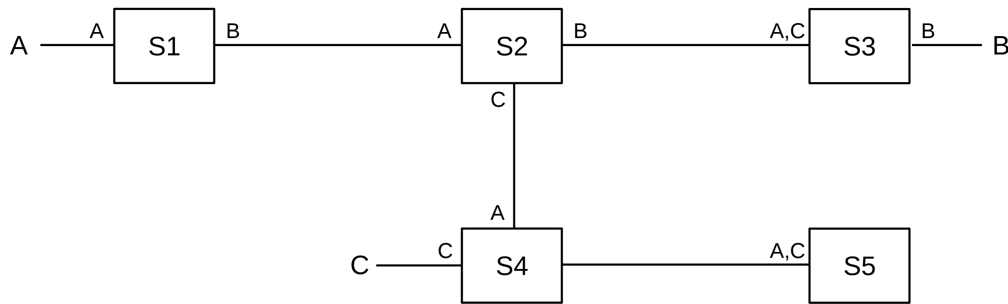
10.0. (This exercise assumes some familiarity with Distance-Vector routing as in *9 Routing-Update Algorithms*.)

(a). Suppose switches are able to identify the non-switch hosts that are **directly connected**, that is, reachable without passing through another switch. Explain how the algorithm of *9.1 Distance-Vector Routing-Update Algorithm* could be used to construct optimal Ethernet forwarding tables even if loops were present in the network topology.

(b). Suppose switches are allowed to "mark" packets; all packets are initially unmarked. Give a mechanism that allows switches to detect which non-switch hosts are directly connected.

(c). Explain why Ethernet broadcast (and multicast) would still be a problem.

11.0. Consider the scenario from *2.4.1 Ethernet Learning Algorithm*:

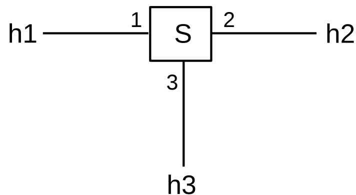


Five learning bridges after three packet transmissions

- **A sends to B**
- **B sends to A**
- **C sends to B**

Now suppose that, before each packet transmission above, the sender first sends a **broadcast** packet, and the destination then sends a unicast reply packet (this is roughly the ARP protocol, used to translate from IPv4 addresses to Ethernet physical addresses, *7.9 Address Resolution Protocol: ARP*). After the three transmissions listed above, what destinations do the switches S1-S5 have in their forwarding tables?

12.0.◊ Consider the following arrangement of three hosts h1, h2, h3 and one OpenFlow switch S with ports 1, 2 and 3 and controller C (not shown)



Four packets are then transmitted:

(a). h1→h2
(b). h2→h1
(c). h3→h1
(d). h2→h3

Assume that S reports to C all packets with **unknown destination**, that is, all packets for which S does not have a forwarding entry for that packet's destination. Packet reports include the source and destination addresses and the arrival port. On receiving a report, if the source address is previously unknown then C installs on S a forwarding-table entry for that source address. At that point S uses its forwarding table (including any new entries) to forward the packet, if a suitable entry exists. Otherwise S floods the packet as usual.

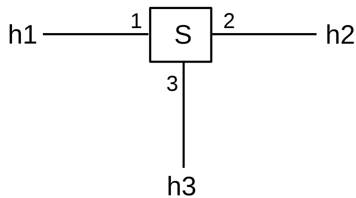
For the four packets above, indicate

1. whether S reports the packet to C

2. if so, any new forwarding entry C installs on S
3. whether S is then able to forward the packet using its table, or must fall back to flooding.

(If S does not report the packet to C then S must have had a forwarding-table entry for that destination, and so S is able to forward the packet normally.)

12.2. Consider again the arrangement of exercise 12.0 of three hosts h1, h2, h3 and one OpenFlow switch S with ports 1, 2 and 3 and controller C (not shown)



The same four packets are transmitted:

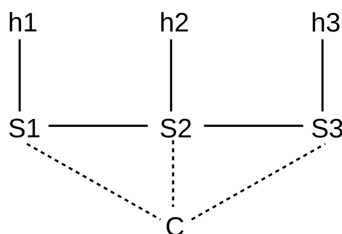
(a). h1→h2
(b). h2→h1
(c). h3→h1
(d). h2→h3

This time, assume that S reports to C all packets with unknown destination **or unknown source** (that is, S does not have a forwarding entry for either the packet's source or destination address). For the four packets above, indicate

1. whether S reports the packet to C
2. if so, any new forwarding entry C installs on S
3. whether S is then able to forward the packet using its table, or must fall back to flooding.

As before, packet reports include the source and destination addresses and the arrival port. On receiving a report, if the source address is previously unknown then C installs on S a forwarding-table entry for that source address. At that point S uses its forwarding table (including any new entries) to forward the packet, if a suitable entry exists. Otherwise S floods the packet as usual. Again, if S does not report a packet to C then S must have had a forwarding-table entry for that destination, and so is able to forward the packet normally.

13.0 Consider the following arrangement of three switches S1-S3, three hosts h1-h3 and one OpenFlow controller C.



As with exercise 12.0, assume that the switches report packets to C only if they do not already have a forwarding-table entry for the packet's destination. After each report, C installs a forwarding-table entry on the reporting switch for reaching the packet's source address via the arrival port. At that point the switch floods the packet (as the destination must not have been known). If a switch can forward a packet without reporting to C, no new forwarding entries are installed.

Packets are now sent as follows:

h1→h2

h2→h1

h1→h3

h3→h1

h2→h3

h3→h2

At the end, what are the forwarding tables on S1♢, S2 and S3?

14.0 Here are the switch rules for the multiple-flow-table example in *2.7.2 Learning Switches in OpenFlow*:

Table	match field	match action	no-match default
T_0	destaddr	forward and send to T_1	flood and send to T_1
T_1	srcaddr	do nothing	send to controller

Give a similar table where the matches are *reversed*; that is, T_0 matches the srcaddr field and T_1 matches the destaddr field.

In the wired era, one could get along quite well with nothing but Ethernet and the occasional long-haul point-to-point link joining different sites. However, there are important alternatives out there. Some, like token ring, are mostly of historical importance; others, like virtual circuits, are of great conceptual importance but – so far – of only modest day-to-day significance.

And then there is wireless. It would be difficult to imagine contemporary laptop networking, let alone mobile devices, without it. In both homes and offices, Wi-Fi connectivity is the norm. Mobile networking is ubiquitous. A return to being tethered by wires is almost unthinkable.

3.1 Virtual Private Networks

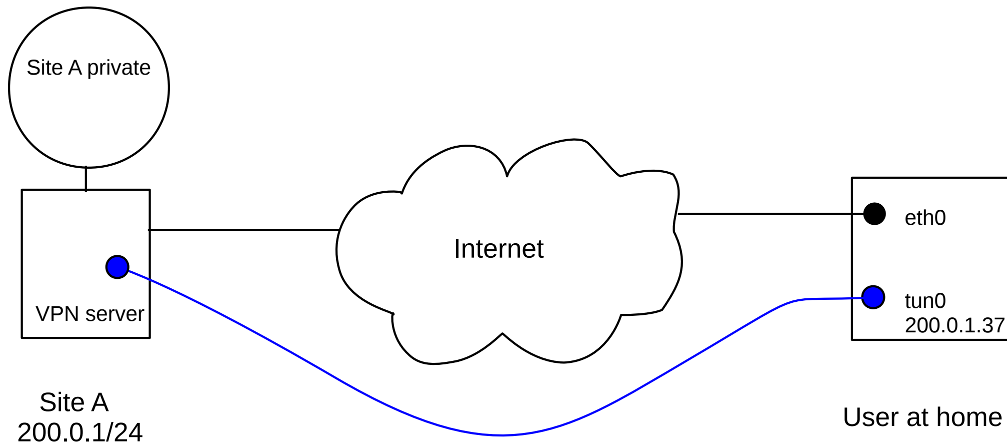
Suppose you want to connect to your workplace network from home. Your workplace, however, has a security policy that does not allow "outside" IP addresses to access essential internal resources. How do you proceed, without leasing a dedicated telecommunications line to your workplace?

A virtual private network, or **VPN**, provides a solution; it supports creation of **virtual links** that join far-flung nodes via the Internet. Your home computer creates an ordinary Internet connection (TCP or UDP) to a workplace **VPN server** (IP-layer packet encapsulation can also be used, and avoids the timeout problems sometimes created by sending TCP packets within another TCP stream; see *7.13 Mobile IP*). Each end of the connection is typically associated with a software-created **virtual network interface**; each of the two virtual interfaces is assigned an IP address. (Virtual interfaces are not essential; VPNs created with IPsec, *22.11 IPsec*, generally omit them.) When a packet is to be sent along the virtual link, it is actually encapsulated and sent along the original Internet connection to the VPN server, wending its way through the commodity Internet; this process is called **tunneling**. To all intents and purposes, the virtual link behaves like any other physical link.

Tunneled packets are often encrypted as well as encapsulated, though that is a separate issue. One relatively easy-to-implement example of a tunneling mechanism is to treat a TCP home-workplace connection as a serial line and send packets over it back-to-back, using PPP with HDLC; see *4.1.5.1 HDLC* and **RFC 1661** (though this can lead to the above-mentioned TCP-in-TCP timeout problems).

At the workplace side, the virtual network interface in the VPN server is attached to a router or switch; at the home user's end, the virtual network interface can now be assigned an *internal* workplace IP address. The home computer is now, for all intents and purposes, part of the internal workplace network.

In the diagram below, the user's regular Internet connection is via hardware interface `eth0`. A connection is established to Site A's VPN server; a virtual interface `tun0` is created on the user's machine which appears to be a direct link to the VPN server. The `tun0` interface is assigned a Site-A IP address. Packets sent via the `tun0` interface in fact travel over the original connection via `eth0` and the Internet.



After the VPN is set up, the home host's `tun0` interface appears to be locally connected to Site A, and thus the home host is allowed to connect to the private area within Site A. The home host's forwarding table will be configured so that traffic to Site A's private addresses is routed via interface `tun0`.

VPNs are also commonly used to connect entire remote offices to headquarters. In this case the remote-office end of the tunnel will be at that office's local router, and the tunnel will carry traffic for all the workstations in the remote office.

Other applications of VPNs include trying to appear geographically to be at another location, and bypassing firewall rules blocking specific TCP or UDP ports.

To improve security, it is common for the residential (or remote-office) end of the VPN connection to use the VPN connection as the default route for all traffic *except* that needed to maintain the VPN itself. This may require a so-called **host-specific** forwarding-table entry at the residential end to allow the packets that carry the VPN tunnel traffic to be routed correctly via `eth0`. This routing strategy means that potential intruders cannot access the residential host – and thus the workplace internal network – through the original residential Internet access. A consequence is that if the home worker downloads a large file from a non-workplace site, it will travel first to the workplace, then back out to the Internet via the VPN connection, and finally arrive at the home.

To improve congestion response, IP packets are sometimes marked by routers that are experiencing congestion; see *14.8.2 Explicit Congestion Notification (ECN)*. If such marking is done to the outer, encapsulating, packet, and the marks are not transferred at the remote endpoint of the VPN to the inner, encapsulat*ed*, packet, then the marks are lost. Congestion response may suffer. **RFC 6040** spells out a proper re-marking strategy in general; **RFC 7296** defines re-marking for IPsec (*22.11 IPsec*). Older VPN protocols, however, may not support congestion re-marking.

3.2 Carrier Ethernet

Carrier Ethernet is a leased-line point-to-point link between two sites, where the subscriber interface at each end of the line looks like Ethernet (in some flavor). The physical path in between sites, however, need not

have anything to do with Ethernet; it may be implemented however the carrier wishes. In particular, it will be (or at least appear to be) full-duplex, it will be collision-free, and its length may far exceed the maximum permitted by any IEEE Ethernet standard.

Bandwidth can be purchased in whatever increments the carrier has implemented. The point of carrier Ethernet is to provide a layflooder of abstraction between the customers, who need only install a commodity Ethernet interface, and the provider, who can upgrade the link implementation at will without requiring change at the customer end.

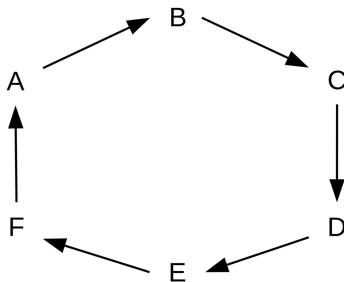
In a sense, carrier Ethernet is similar to the widespread practice of provisioning residential DSL and cable routers with an Ethernet interface for customer interconnection; again, the actual link technologies may not look anything like Ethernet, but the interface will.

A carrier Ethernet connection looks like a virtual VPN link, but runs on top of the provider's internal network rather than the Internet at large. Carrier Ethernet connections often provide the primary Internet connectivity for one endpoint, unlike Internet VPNs which assume both endpoints already have full Internet connectivity.

3.3 Token Ring

A significant part of the previous chapter was devoted to classic Ethernet's collision mechanism for supporting shared media access. After that, it may come as a surprise that there is a simple multiple-access mechanism that is not only **collision-free**, but which supports **fairness** in the sense that if N stations wish to send then each will receive 1/N of the opportunities.

That method is **Token Ring**. Actual implementations come in several forms, from Fiber-Distributed Data Interface (FDDI) to so-called "IBM Token Ring". The central idea is that stations are connected in a ring:



Packets will be transmitted in one direction (clockwise in the ring above). Stations in effect forward most packets around the ring, although they can also remove a packet. (It is perhaps more accurate to think of the forwarding as representing the default cable connectivity; *non*-forwarding represents the station's momentarily breaking that connectivity.)

When the network is idle, all stations agree to forward a special, small packet known as a *token*. When a station, say A, wishes to transmit, it must first wait for the token to arrive at A. Instead of forwarding the token, A then transmits its own packet; this travels around the network and is then removed by A. At that point (or in some cases at the point when A finishes transmitting its data packet) A then forwards the token.

In a small ring network, the ring circumference may be a small fraction of one packet. Ring networks become "large" at the point when some packets may be entirely in transit on the ring. Slightly different

solutions apply in each case. (It is also possible that the physical ring exists only within the token-ring switch, and that stations are connected to that switch using the usual point-to-point wiring.)

If all stations have packets to send, then we will have something like the following:

- A waits for the token
- A sends a packet
- A sends the token to B
- B sends a packet
- B sends the token to C
- C sends a packet
- C sends the token to D
- ...

All stations get an equal number of chances to transmit, and no bandwidth is wasted on collisions. (A station constantly sending smaller packets will send the same number of packets as a station constantly sending larger packets, but the bandwidth will be smaller in proportion to the smaller packet size.)

One problem with token ring is that when stations are powered off it is *essential* that the packets continue forwarding; this is usually addressed by having the default circuit configuration be to keep the loop closed. Another issue is that some station has to watch out in case the token disappears, or in case a duplicate token appears.

Because of fairness and the lack of collisions, IBM Token Ring was once considered to be the premium LAN mechanism. As such, Token Ring hardware commanded a substantial price premium. But due to Ethernet's combination of lower hardware costs and higher bitrates (even taking collisions into account), the latter eventually won out.

There was also a much earlier collision-free hybrid of 10 Mbps Ethernet and Token Ring known as **Token Bus**: an Ethernet physical network (often linear) was used with a token-ring-like protocol layer above that. Stations were physically connected to the (linear) Ethernet but were assigned identifiers that logically arranged them in a (virtual) ring. Each station had to wait for the token and only then could transmit a packet; after that it would send the token on to the next station in the virtual ring. As with "real" Token Ring, some mechanisms need to be in place to monitor for token loss.

Token Bus Ethernet never caught on. The additional software complexity was no doubt part of the problem, but perhaps the real issue was that it was not necessary.

3.4 Virtual Circuits

Before we can get to our final LAN example, ATM, we need to detour briefly through virtual circuits.

The Road Not Taken

A close reading of Robert Frost's poem referenced here reveals that the supposed great difference between the two roads exists only in the narrator's retrospective imaginings; the roads were in fact "really about the same". Perhaps this would also apply to datagram and virtual-circuit forwarding, though see below on per-connection billing.

Virtual circuits are The Road Not Taken by IP.

Virtual-circuit switching (or routing) is an alternative to datagram switching, which was introduced in Chapter 1. In datagram switching, routers know the next_hop to each destination, and packets are addressed by *destination*. In virtual-circuit switching, routers know about end-to-end *connections*, and packets are "addressed" by a connection ID.

Before any packets can be sent, a connection needs to be established first. For that connection, the route is computed and then each link along the path is assigned a connection ID, traditionally called the **VCI**, for Virtual Circuit Identifier. In most cases, VCIs are only *locally* unique; that is, the same connection may use a different VCI on each link. The lack of global uniqueness makes VCI allocation much simpler. Although the VCI keeps changing along a path, the VCI can still be thought of as identifying the connection. To send a packet, the host marks the packet with the VCI assigned to the host–router1 link.

Packets arrive at (and depart from) switches via one of several **ports**, which we will assume are numbered beginning at 0. Switches maintain a **connection table** indexed by ⟨VCI,port⟩ pairs; unlike a forwarding table, the connection table has a record of every connection through that switch at that particular moment. As a packet arrives, its inbound VCI_{in} and inbound port_{in} are looked up in this table; this yields an outbound $\langle\text{VCI}_{\text{out}},\text{port}_{\text{out}}\rangle$ pair. The VCI field of the packet is then *rewritten* to VCI_{out}, and the packet is sent via port_{out}.

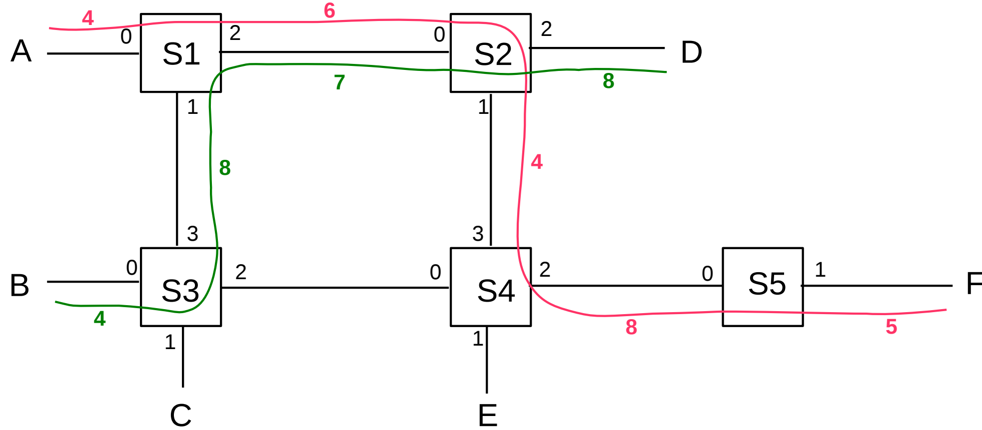
Note that typically there is no source address information included in the packet (although the sender can be identified from the connection, which can be identified from the VCI at any point along the connection). Packets are identified by connection, not destination. Any node along the path (including the endpoints) can in principle look up the connection and figure out the endpoints.

Note also that each switch must rewrite the VCI. Datagram switches never rewrite addresses (though they do update hopcount/TTL fields). The advantage to this rewriting is that VCIs need be unique only for a given link, greatly simplifying the naming. Datagram switches also do not make use of a packet's arrival interface.

As an example, consider the network below. Switch ports are numbered 0,1,2,3. Two paths are drawn in, one from A to F in red and one from B to D in green; each link is labeled with its VCI number in the same color.

We will construct virtual-circuit connections between

- A and F (shown above in red)
- A and E
- A and C
- B and D (shown above in green)
- A and F again (a separate connection)

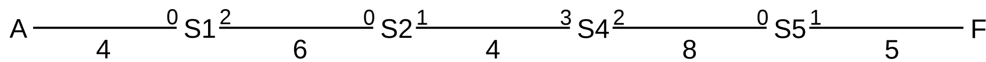


The following VCIs have been chosen for these connections. The choices are made more or less randomly here, but in accordance with the requirement that they be unique to each link. Because links are generally taken to be bidirectional, a VCI used from S1 to S3 cannot be reused from S3 to S1 until the first connection closes.

- A to F: A—4—S1—6—S2—4—S4—8—S5—5—F; this path goes from S1 to S4 via S2
- A to E: A—5—S1—6—S3—3—S4—8—E; this path goes, for no particular reason, from S1 to S4 via S3, the opposite corner of the square
- A to C: A—6—S1—7—S3—3—C
- B to D: B—4—S3—8—S1—7—S2—8—D
- A to F: A—7—S1—8—S2—5—S4—9—S5—2—F

One may verify that on any one link no two different paths use the same VCI.

We now construct the actual ⟨VCI,port⟩ tables for the switches S1-S4, from the above; the table for S5 is left as an exercise. Note that either the ⟨VCI_{in},$port_{in}$⟩ or the ⟨VCI_{out},$port_{out}$⟩ can be used as the key; we cannot have the same pair in both the in columns and the out columns. It may help to display the port numbers for each switch, as in the upper numbers in following diagram of the above red connection from A to F (lower numbers are the VCIs):



Switch **S1**:

VCI_{in}	$port_{in}$	VCI_{out}	$port_{out}$	connection
4	0	6	2	A⟶F #1
5	0	6	1	A⟶E
6	0	7	1	A⟶C
8	1	7	2	B⟶D
7	0	8	2	A⟶F #2

Switch **S2**:

VCI_{in}	$port_{in}$	VCI_{out}	$port_{out}$	connection
6	0	4	1	A⟶F #1
7	0	8	2	B⟶D
8	0	5	1	A⟶F #2

Switch **S3**:

VCI_{in}	$port_{in}$	VCI_{out}	$port_{out}$	connection
6	3	3	2	A⟶E
7	3	3	1	A⟶C
4	0	8	3	B⟶D

Switch **S4**:

VCI_{in}	$port_{in}$	VCI_{out}	$port_{out}$	connection
4	3	8	2	A⟶F #1
3	0	8	1	A⟶E
5	3	9	2	A⟶F #2

The namespace for VCIs is small, and compact (*eg* contiguous). Typically the VCI and port bitfields can be concatenated to produce a ⟨VCI,Port⟩ composite value small enough that it is suitable for use as an array index. VCIs work best as *local* identifiers. IP addresses, on the other hand, need to be globally unique, and thus are often rather sparsely distributed.

Virtual-circuit switching offers the following advantages:

- connections can get quality-of-service guarantees, because the switches are aware of connections and can reserve capacity at the time the connection is made
- headers are smaller, allowing faster throughput
- headers are small enough to allow efficient support for the very small packet sizes that are optimal for voice connections. ATM packets, for instance, have 48 bytes of data; see below.

Datagram forwarding, on the other hand, offers these advantages:

- Routers have less state information to manage.
- Router crashes and partial connection state loss are not a problem.
- If a router or link is disabled, rerouting is easy and does not affect any connection state. (As mentioned in Chapter 1, this was Paul Baran's primary concern in his 1962 paper introducing packet switching.)
- Per-connection billing is very difficult.

The last point above may once have been quite important; in the era when the ARPANET was being developed, typical daytime long-distance rates were on the order of $1/minute. It is unlikely that early TCP/IP protocol development would have been as fertile as it was had participants needed to justify per-minute billing costs for every project.

It is certainly possible to do virtual-circuit switching with globally unique VCIs – say the concatenation of source and destination IP addresses and port numbers. The IP-based RSVP protocol (*20.6 RSVP*) does exactly this. However, the fast-lookup and small-header advantages of a compact namespace are then lost.

Multi-Protocol Label Switching (*20.12 Multi-Protocol Label Switching (MPLS)*) is another IP-based application of virtual circuits.

Note that virtual-circuit switching does *not* suffer from the problem of idle channels still consuming resources, which is an issue with circuits using time-division multiplexing (*eg* shared T1 lines)

3.5 Asynchronous Transfer Mode: ATM

ATM is a network mechanism intended to accommodate real-time traffic as well as bulk data transfer. We present ATM here as a LAN layer, for which it is still sometimes used, but it was originally proposed as a replacement for the IP layer as well, and, to an extent, the Transport layer. These broader plans were not greeted with universal enthusiasm within the IETF. When used as a LAN layer, IP packets are transmitted over ATM as in *3.5.1 ATM Segmentation and Reassembly*.

A distinctive feature of ATM is its small packet size. ATM has its roots in the telephone industry, and was therefore particularly intended to support voice. A significant source of delay in voice traffic is the packet **fill time**: at DS0 speeds (64 Kbps), voice data accumulates at 8 bytes/ms. If we are sending 1KB packets, this means voice is delayed by about 1/8 second, meaning in turn that when one person stops speaking, the earliest they can hear the other's response is 1/4 second later. Slightly smaller levels of voice delay can introduce an annoying echo. Smaller packets reduce the fill time and thus the delay: when voice is sent over IP (VoIP), one common method is to send 160 bytes every 20 ms.

ATM took this small-packet strategy even further: packets have 48 bytes of data, plus 5 bytes of header. Such small packets are often called *cells*. To manage such a small header, virtual-circuit routing is a necessity. IP packets of such small size would likely consume more than 50% of the bandwidth on headers, if the LAN header were included.

Aside from reduced voice fill-time, other benefits to small cells are reduced store-and-forward delay and minimal queuing delay, at least for high-priority traffic. Prioritizing traffic and giving precedence to high-priority traffic is standard, but high-priority traffic is never allowed to *interrupt* transmission already begun of a low-priority packet. If you have a high-priority voice cell, and someone else has a 1500-byte packet just started, your cell has to wait about 30 cell times, because 1500 bytes is about 30 cells. However, if their low-priority traffic is instead made up of 30 cells, you have only to wait for their first cell to finish; the delay is 1/30 as much.

ATM also made the decision to require **fixed-size** cells. The penalty for one partially used cell among many is small. Having a fixed cell size simplifies hardware design, and, in theory, allows it easier to design for parallelism.

Unfortunately, the designers of ATM also chose to mandate **no cell reordering**. This means cells can use a smaller sequence-number field, but also makes parallel switches much harder to build. A typical parallel switch design might involve distributing incoming cells among any of several input queues; the queues would then handle the VCI lookups in parallel and forward the cells to the appropriate output queues. With such an architecture, avoiding reordering is difficult. It is not clear to what extent the no-reordering decision was related to the later decline of ATM in the marketplace.

ATM cells have 48 bytes of data and a 5-byte header. The header contains up to 28 bits of VCI information, three "type" bits, one **cell-loss priority**, or CLP, bit, and an 8-bit checksum over the header only. The VCI is divided into 8-12 bits of Virtual Path Identifier and 16 bits of Virtual Channel Identifier, the latter supposedly for customer use to separate out multiple connections between two endpoints. Forwarding is by full switching only, and there is no mechanism for physical (LAN) broadcast.

3.5.1 ATM Segmentation and Reassembly

Due to the small packet size, ATM defines its own mechanisms for segmentation and reassembly of larger packets. Thus, individual ATM links in an IP network are quite practical. These mechanisms are called **ATM Adaptation Layers**, and there are four of them: AALs 1, 2, 3/4 and 5 (AAL 3 and AAL 4 were once separate layers, which merged). AALs 1 and 2 are used only for voice-type traffic; we will not consider them further.

The ATM segmentation-and-reassembly mechanism defined here is intended to apply only to large *data*; no cells are ever further subdivided. Furthermore, segmentation is always applied at the point where the data enters the network; reassembly is done at exit from the ATM path. IPv4 fragmentation, on the other hand, applies conceptually to IP packets, and may be performed by routers within the network.

For AAL 3/4, we first define a high-level "wrapper" for an IP packet, called the CS-PDU (Convergence Sublayer - Protocol Data Unit). This prefixes 32 bits on the front and another 32 bits (plus padding) on the rear. We then chop this into as many 44-byte chunks as are needed; each chunk goes into a 48-byte ATM payload, along with the following 32 bits worth of additional header/trailer:

- 2-bit **type** field:
  - 10: begin new CS-PDU
  - 00: continue CS-PDU
  - 01: end of CS-PDU
  - 11: single-segment CS-PDU
- 4-bit sequence number, 0-15, good for catching up to 15 dropped cells
- 10-bit MessageID field
- CRC-10 checksum.

We now have a total of 9 bytes of header for 44 bytes of data; this is more than 20% overhead. This did not sit well with the IP-over-ATM community (such as it was), and so AAL 5 was developed.

AAL 5 moved the checksum to the CS-PDU and increased it to 32 bits from 10 bits. The MID field was discarded, as no one used it, anyway (if you wanted to send several different types of messages, you simply created several virtual circuits). A bit from the ATM header was taken over and used to indicate:

- 1: start of new CS-PDU
- 0: continuation of an existing CS-PDU

The CS-PDU is now chopped into 48-byte chunks, which are then used as the entire body of each ATM cell. With 5 bytes of header for 48 bytes of data, overhead is down to 10%. Errors are detected by the CS-PDU CRC-32. This also detects lost cells (impossible with a per-cell CRC!), as we no longer have any cell sequence number.

For both AAL3/4 and AAL5, **reassembly** is simply a matter of stringing together consecutive cells **in order of arrival**, starting a new CS-PDU whenever the appropriate bits indicate this. For AAL3/4 the receiver has to strip off the 4-byte AAL3/4 headers; for AAL5 the receiver has to verify the CRC-32 checksum once all cells are received. Different cells from different virtual circuits can be jumbled together on the ATM

"backbone", but on any one virtual circuit the cells from one higher-level packet must be sent one right after the other.

A typical IP packet divides into about 20 cells. For AAL 3/4, this means a total of 200 bits devoted to CRC codes, versus only 32 bits for AAL 5. It might seem that AAL 3/4 would be more reliable because of this, but, paradoxically, it was not! The reason for this is that errors are *rare*, and so we typically have one or at most two per CS-PDU. Suppose we have only a single error, *ie* a single cluster of corrupted bits small enough that it is likely confined to a single cell. In AAL 3/4 the CRC-10 checksum will fail to detect that error (that is, the checksum of the corrupted packet will by chance happen to equal the checksum of the original packet) with probability $1/2^{10}$. The AAL 5 CRC-32 checksum, however, will fail to detect the error with probability $1/2^{32}$. Even if there are enough errors that two cells are corrupted, the two CRC-10s together will fail to detect the error with probability $1/2^{20}$; the CRC-32 is better. AAL 3/4 is more reliable only when we have errors in at least four cells, at which point we might do better to switch to an error-*correcting* code.

Moral: one checksum over the entire message is often better than multiple shorter checksums over parts of the message.

3.6 Adventures in Radioland

For the remainder of this chapter we leave wires (and fiber) behind, and contemplate the transmission of packets via radio, freeing nodes from their cable tethers. Wi-fi (*3.7 Wi-Fi*) and mobile wireless (*3.8 WiMAX and LTE*) are now ubiquitous. But radio is not quite like wire, and wireless transmission of packets brings several changes.

3.6.1 Privacy

It's hard to tap into wired Ethernet, especially if you are locked out of the building. But anyone can receive wireless transmissions, often from a considerable distance. The data breach at TJX Corporation was achieved by attackers parking outside a company building and pointing a directional antenna at it; encryption was used but it was weak (see *22 Security* and *22.7.7 Wi-Fi WEP Encryption Failure*). Similarly, Internet café visitors generally don't want other patrons to read their email. Radio communication needs strong encryption.

3.6.2 Collisions

Ethernet-like **collision detection** is no longer feasible over radio. This has to do with the relative signal strength of the remote signal at the local transmitter. Along a wire-based Ethernet the remote signal might be as weak as 1/100 of the transmitted signal but that 1% received signal is still detectable *during* transmission. However, with radio the remote signal might easily be as little as 1/1,000,000 of the transmitted signal (-60 dB), as measured at the transmitting station, and it is simply overwhelmed during transmission.

As a result, wireless protocols must be constructed appropriately. We will look at how Wi-Fi handles this in its most common mode of operation in *3.7.1 Wi-Fi and Collisions*. Wi-Fi also supports its PCF mode (*3.7.7 Wi-Fi Polling Mode*) that involves fewer (but not zero) collisions through the use of central-point **polling**. Finally, WiMAX and LTE switch from polling to **scheduling** to further reduce collisions, though the potential for collisions is still inevitable when new stations join the network.

It is also worth pointing out that, while an Ethernet collision affects every station in the physical Ethernet (the "collision domain"), wireless collisions are **local**, occuring at the receiver. Two stations can transmit at the same time, and in range of one another, but without a collision! This can happen if each of the relevant *receivers* is in range of only one of the two transmitting stations. As an example, suppose three stations are arranged linearly, A–C–B, with the A–C and C–B distances just under the maximum effective range. When A and B both transmit there is indeed a collision at C. But when C and B transmit simultaneously, A may receive C's signal just fine, as B's is too weak to interfere.

3.6.3 Hidden Nodes

In wireless communication, two nodes A and B that are not in range of one another – and thus cannot detect one another – may still have their signals interfere at a third node C. This creates an additional complication to collision handling. See *3.7.1.4 Hidden-Node Problem*.

3.6.4 Band Width

To radio engineers, "band width" means the frequency range used by a signal, not the data transmission rate. No information can be conveyed using a single frequency; even signaling by switching a carrier frequency off and on at a low rate "blurs" the carrier into a band of nonzero width.

In keeping with this we will for the remainder of this chapter use the term "data rate" for what we have previously called "bandwidth". We will use the terms "channel width" or "width of the frequency band" for the frequency range.

All else being equal, the data rate achievable with a radio signal is proportional to the channel width. The constant of proportionality is limited by the Shannon-Hartley theorem: the maximum data rate divided by the width of the frequency band is $\log_2(1+\text{SNR})$, where SNR is the **signal to noise** power ratio. Noise here is assumed to have a specific statistical form known as *Gaussian white noise*. If SNR is 127, for example, and the width of the frequency band is 1 MHz, then the maximum theoretical data rate is 7 Mbps, where $7 = \log_2(128)$. If the signal power S drops by about half so SNR=63, the data rate falls to 6 Mbps, as $6 = \log_2(64)$; the relationship between signal power and data rate is logarithmic.

3.6.4.1 OFDM

The actual data rate achievable, for a given channel width and SNR, depends on the signal encoding, or **modulation**, mechanism. Most newer modulation mechanisms use "orthogonal frequency-division multiplexing", **OFDM**, or some variant.

A central feature of OFDM is that one wider frequency band is divided into multiple narrow subchannels; each subchannel then carries a proportional fraction of the total information signal, modulated onto a subchannel-specific carrier. All the subchannels can be allocated to one transmission at a time (time-division multiplexing, *4.2 Time-Division Multiplexing*), or disjoint sets of subchannels can be allocated to different transmissions that can then proceed (at proportionally lower data rates) in parallel. The latter is known as *frequency-division multiplexing*.

In many settings OFDM comes reasonably close to the Shannon-Hartley limit. Perhaps more importantly, OFDM also performs reasonably well with *multipath interference*, below, which is endemic in urban and building-interior environments with their many reflective surfaces. Multipath interference is, however, not

necessarily comparable to the Gaussian noise assumed by the Shannon-Hartley theorem. We will not address further technical details of OFDM here, except to note that implementation usually requires some form of digital signal processing.

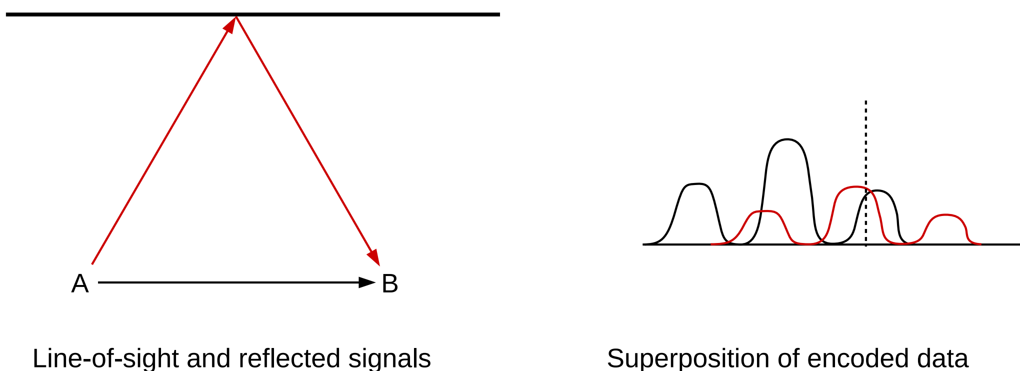
3.6.5 Cost

Another fundamental issue is that everyone shares the same radio spectrum. For mobile wireless providers, this constraint has driven prices to remarkable levels; the 2014-15 FCC AWS-3 auction raised almost $45 billion for 65 MHz (usable throughout the entire United States). This works out to somewhat over $2 per megahertz per phone. The corresponding issue for Wi-Fi users in a dense area is that all the available Wi-Fi bandwidth may be in use. Inside busy buildings one can often see dozens of Wi-Fi access points competing for the same Wi-Fi channel; the result is that no user will be getting close to the nominal data rates of *3.7 Wi-Fi*.

Higher data rates require wider frequency bands. To reduce costs in the face of fixed demand, the usual strategy is to make the coverage zones smaller, either by reducing power (and adding more access points as appropriate), or by using directional antennas, or both.

3.6.6 Multipath

While a radio signal generally covers a wide area – even with ordinary directional antennas – it does so in surprisingly non-uniform ways. A signal may reach a receiver through a line-of-sight path and also several reflected paths, possibly of varying length. In addition to reflection, the signal may be subject to reflection-like *scattering* and *diffraction*. All of this together is known as **multipath interference** (or, if analog audio is involved, multipath *distortion*; in the analog TV era this was *ghosting*).

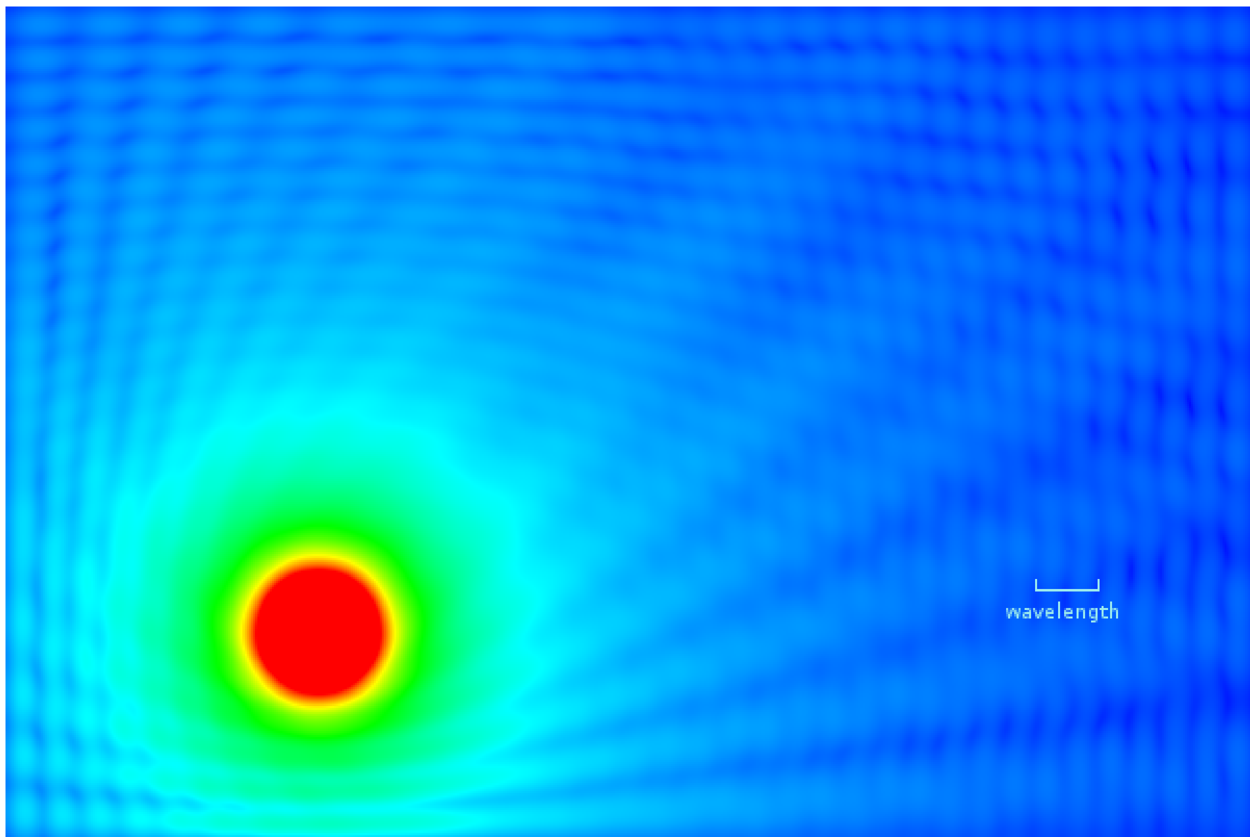


Line-of-sight and reflected signals

Superposition of encoded data

The picture above shows two transmission paths from A to B. The respective carrier paths may interfere with or supplement one another. The longer delay of the reflecting path (red) wil also delay its encoded signal. The result, shown at right, is that the line-of-sight and reflected data symbols may overlap and interfere with each other; this is known as **intersymbol interference**. Multipath interference may even change the meaning of the data symbol as seen by the receiver; for example, the red and black low data-signal peaks above at the point of the vertical dashed line may sum together so as to be received as a higher peak (assuming the underlying carriers are in sync).

Multipath interference tends to lead to wide fluctuations in signal intensity with a period of about half a wavelength;



Signal-intensity map (simulated) in a room with walls with 40% reflectivity

The picture above is from a mathematical simulation intended to illustrate multipath fading. The walls of the room reflect 40% of the signal from the transmitter located in the orange ball at the lower left. The transmitter transmits an unmodulated carrier signal, which may be reflected off the walls any number of times; at any point in the room the total signal intensity is the sum over all possible reflection paths. On the right-hand side, the small-scale blue ripples represent the received carrier strength variation due to multipath interference between the line-of-sight and all the reflected paths. Note that the ripple size is about half a wavelength.

In comparison to this simulated intensity map, real walls tend to have a lower reflectivity, real rooms are not two-dimensional, and real carriers are modulated. However, real rooms also introduce scattering, diffraction and shadowing from objects within, and significant (3× to 10×) multipath-fading signal-strength variations are common in actual wireless settings.

Multipath fading can be either **flat** – affecting all frequencies more or less equally – or **selective** – affecting some frequencies differently than others. It is quite possible for an OFDM channel (*3.6.4.1 OFDM*) to encounter selective fading of only some of its subchannel frequencies.

Generally, multipath interference is a problem that engineers go to great lengths to overcome. However, as we shall see in *3.7.3 Multiple Spatial Streams*, multipath interference can sometimes be put to positive use by allowing almost-adjacent antennas to transmit and receive independent signals, thus increasing the

effective throughput.

For an alternative example of multipath interference in which the signal strength has no ripples, see exercise 13.0.

3.6.7 Power

If you are cutting the network cable and replacing it with wireless, there is a good chance you will also want to cut the power cable as well and replace it with batteries. This tends to make power consumption a very important issue. The Wi-Fi standard has provisions for minimizing power usage by allowing a device to "doze" most of the time, waking periodically to check if any packets are ready to be sent to it (see *3.7.4.1 Joining a Network*). The 6LoWPAN project (IPv6 Low-power Wireless Personal Area Network) is intended to support very low-power devices; see **RFC 4919** and **RFC 6282**.

3.6.8 Tangle

Wireless is also used simply to replace cords and their attendant tangle, and, of course, the problem of incompatible connectors. The low-power **Bluetooth** wireless standard is commonly used as a cable alternative for things like computer mice and telephone headsets. Bluetooth is also a low-power network; for many applications the working range is about 10 meters. **ZigBee** is another low-power small-scale network.

3.7 Wi-Fi

Wi-Fi is a trademark of the Wi-Fi Alliance denoting any of several IEEE wireless-networking protocols in the 802.11 family, specifically 802.11a, 802.11b, 802.11g, 802.11n, and 802.11ac. (Strictly speaking, these are all *amendments* to the original 802.11 standard, but they are also *de facto* standards in their own right.) Like classic Ethernet, Wi-Fi must deal with **collisions**; unlike Ethernet, however, Wi-Fi is unable to detect collisions in progress, complicating the backoff and retransmission algorithms. See *3.6.2 Collisions* above.

Unlike any wired LAN protocol we have considered so far, in addition to normal data packets Wi-Fi also uses **control** and **management** packets that exist entirely within the Wi-Fi LAN layer; these are not initiated by or delivered to higher network layers. Control packets are used to compensate for some of the infelicities of the radio environment, such as the lack of collision detection. Putting radio-essential control and management protocols within the Wi-Fi layer means that the IP layer can continue to interact with the Wi-Fi LAN exactly as it did with Ethernet; no changes are required.

Wi-Fi is designed to interoperate freely with Ethernet at the logical LAN layer. Wi-Fi MAC (physical) addresses have the same 48-bit size as Ethernet's and the same internal structure (*2.1.3 Ethernet Address Internal Structure*). They also share the same namespace: one is never supposed to see an Ethernet and a Wi-Fi interface with the same address. As a result, data packets can be forwarded by switches between Ethernet and Wi-Fi; in many respects a Wi-Fi LAN attached to an Ethernet LAN looks like an extension of the Ethernet LAN. See *3.7.4 Access Points*.

Microwave Ovens and Wi-Fi

The impact of a running microwave oven on Wi-Fi signals is quite evident if the oven is between the sender and receiver. For other configurations the effect may vary. Most ovens transmit only during one half of the A/C cycle, that is, they are on 1/60 sec and then off 1/60 sec; this may allow intervening transmission time. See also here.

Generally, Wi-Fi uses the 2.4 GHz **ISM** (Industrial, Scientific and Medical) band used also by microwave ovens, though 802.11a uses a 5 GHz band, 802.11n supports that as an option and the new 802.11ac has returned to using 5 GHz exclusively. The 5 GHz band has reduced ability to penetrate walls, often resulting in a lower effective range (though in offices and multi-unit housing this can be an advantage).

Wi-Fi radio spectrum is usually **unlicensed**, meaning that no special permission is needed to transmit but also that others may be trying to use the same frequency band simultaneously; the availability of unlicensed channels in the 5 GHz band continues to evolve.

The table below summarizes the different Wi-Fi versions. All data bit rates assume a single spatial stream; channel widths are nominal.

IEEE name	maximum bit rate	frequency	channel width
802.11a	54 Mbps	5 GHz	20 MHz
802.11b	11 Mbps	2.4 GHz	20 MHz
802.11g	54 Mbps	2.4 GHz	20 MHz
802.11n	65-150 Mbps	2.4/5 GHz	20-40 MHz
802.11ac	78-867 Mbps	5 GHz	20-160 MHz

The maximum bit rate is seldom achieved in practice. The *effective* bit rate must take into account, at a minimum, the time spent in the collision-handling mechanism. More significantly, all the Wi-Fi variants above use dynamic rate scaling, below; the bit rate is reduced up to tenfold (or more) in environments with higher error rates, which can be due to distance, obstructions, competing transmissions or radio noise. All this means that, as a practical matter, getting 150 Mbps out of 802.11n requires optimum circumstances; in particular, no competing senders and unimpeded line-of-sight transmission. 802.11n lower-end performance can be as little as 10 Mbps, though 40-100 Mbps (for a 40 MHz channel) may be more typical.

The 2.4 GHz ISM band is divided by international agreement into up to 14 officially designated (and mostly adjacent) channels, each about 5 MHz wide, though in the United States use may be limited to the first 11 channels. The 5 GHz band is similarly divided into 5 MHz channels. One Wi-Fi sender, however, needs several of these official channels; the typical 2.4 GHz 802.11g transmitter uses an actual frequency range of up to 22 MHz, or up to five official channels. As a result, to avoid signal overlap Wi-Fi use in the 2.4 GHz band is often restricted to official channels 1, 6 and 11. The end result is that there are generally only three available Wi-Fi bands in the 2.4 GHz range, and so Wi-Fi transmitters can and do interact with and interfere with each other.

There are almost 200 5 MHz channels in the 5 GHz band. The United States requires users of the this band to avoid interfering with weather and military applications in the same frequency range; this may involve careful control of transmission power and so-called “dynamic frequency selection” to choose channels with little interference. Even so, there are many more channels than at 2.4 GHz; the larger number of channels is one of the reasons (arguably the primary reason) that 802.11ac can run faster (below).

Wi-Fi designers can improve throughput through a variety of techniques, including

1. improved radio modulation techniques

2. improved error-correcting codes
3. smaller guard intervals between symbols
4. increasing the channel width
5. allowing multiple spatial streams via multiple antennas

The first two in this list seem now to be largely tapped out; OFDM modulation (*3.6.4.1 OFDM*) is close enough to the Shannon-Hartley limit that there is limited room for improvement. The third reduces the range (because there is less protection from multipath interference) but may increase the data rate by ~10%. The largest speed increases are obtained the last two items in the list.

The channel width is increased by adding additional 5 MHz channels. For example, the 65 Mbps bit rate above for 802.11n is for a nominal frequency range of 20 MHz, comparable to that of 802.11g. However, in areas with minimal competition from other signals, 802.11n supports using a 40 MHz frequency band; the bit rate then goes up to 135 Mbps (or 150 Mbps if a smaller guard interval is used). This amounts to using two of the three available 2.4 GHz Wi-Fi bands. Similarly, the wide range in 802.11ac bit rates reflects support for using channel widths ranging from 20 MHz up to 160 MHz (32 5-MHz official channels).

Using multiple spatial streams is the newest data-rate-improvement technique; see *3.7.3 Multiple Spatial Streams*.

For all the categories in the table above, additional bits are used for error-correcting codes. For 802.11g operating at 54 Mbps, for example, the actual raw bit rate is (4/3)×54 = 72 Mbps, sent in symbols consisting of six bits as a unit.

3.7.1 Wi-Fi and Collisions

We looked extensively at the 10 Mbps Ethernet collision-handling mechanisms in *2.1 10-Mbps Classic Ethernet*, only to conclude that with switches and full-duplex links, Ethernet collisions are rapidly becoming a thing of the past. Wi-Fi, however, has brought collisions back from obscurity. An Ethernet sender will discover a collision, if one occurs, during the first slot time, by monitoring for faint interference with its own transmission. However, as mentioned in *3.6.2 Collisions*, Wi-Fi transmitting stations simply cannot detect collisions in progress. If another station transmits at the same time, a Wi-Fi sender will see nothing amiss although its signal will not be received. While there *is* a largely-collision-free mode for Wi-Fi operation (*3.7.7 Wi-Fi Polling Mode*), it is not commonly used, and collision management has a significant impact on ordinary Wi-Fi performance.

3.7.1.1 Link-Layer ACKs

The first problem with Wi-Fi collisions is even detecting them. Because of the inability to detect collisions directly, the Wi-Fi protocol adds **link-layer ACK packets**, at least for unicast transmission. These ACKs are our first example of Wi-Fi **control** packets and are unrelated to the higher-layer TCP ACKs.

The reliable delivery of these link-layer ACKs depends on careful timing. There are three time intervals applicable (numeric values here are for 802.11b/g in the 2.4 GHz band). The value we here call IFS is more formally known as DIFS (D for “distributed”; see *3.7.7 Wi-Fi Polling Mode*).

- slot time: 20 µsec

- IFS, the "normal" InterFrame Spacing: 50 µsec
- SIFS, the *short* IFS: 10 µsec

For comparison, note that the RTT between two Wi-Fi stations 100 meters apart is less than 1 µsec. At 11 Mbps, one IFS time is enough to send about 70 bytes; at 54 Mbps it is enough to send almost 340 bytes.

Once a station has received a data packet addressed to it, it waits for time SIFS and sends its ACK. At this point in time the receiver will be the only station authorized to send, because, as we will see in the next section, all other stations (including those on someone else's overlapping Wi-Fi) will be required to wait the longer IFS period following the end of the previous data transmission. These other stations will see the ACK before the IFS time has elapsed and will thus not interfere with it (though see exercise 4.0).

If a packet is involved in a collision, the receiver will send no ACK, so the sender will know something went awry. Unfortunately, the sender will *not* be able to tell whether the problem was due to a collision, or electromagnetic interference, or signal blockage, or excessive distance, or the receiver's being powered off. But as a collision is usually the most likely cause, and as assuming the lost packet was involved in a collision results in, at worst, a slight additional delay in retransmission, a collision will always be assumed.

Link-Layer ACKs contain no information – such as a sequence number – that identifies the packet being acknowledged. These ACKs simply acknowledge the most recent transmission, the one that ended one SIFS earlier. In the Wi-Fi context, this is unambiguous. It may be compared, however, to *6.1 Building Reliable Transport: Stop-and-Wait*, where at least one bit of packet sequence numbering is required.

3.7.1.2 Collision Avoidance and Backoff

The Ethernet collision-management algorithm was known as CSMA/CD, where CD stood for Collision Detection. The corresponding Wi-Fi mechanism is CSMA/**CA**, where CA stands for Collision **Avoidance**. A collision is presumed to have occurred if the link-layer ACK is not received. As with Ethernet, there is an exponential-backoff mechanism as well, though it is scaled somewhat differently.

Any sender wanting to send a new data packet waits the IFS time after first sensing the medium to see if it is idle. If no other traffic is seen in this interval, the station may then transmit immediately. However, if other traffic *is* sensed, the sender must do an exponential backoff even for its first transmission attempt; other stations, after all, are likely also waiting, and avoiding an initial collision is strongly preferred.

The initial backoff is to choose a random $k<2^5 = 32$ (recall that classic Ethernet in effect chooses an initial backoff of $k<2^0 = 1$; *ie* k=0). The prospective sender then waits k slot times. While waiting, the sender continues to monitor for other traffic; if any other transmission is detected, then the sender "suspends" the backoff-wait clock. The clock resumes when the other transmission has completed and one followup idle interval of length IFS has elapsed.

Note that, under these rules, data-packet senders *always* wait for at least one idle interval of length IFS before sending, thus ensuring that they never collide with an ACK sent after an idle interval of only SIFS.

On an Ethernet, if two stations are waiting for a third to finish before they transmit, they will both transmit as soon as the third is finished and so there will always be an initial collision. With Wi-Fi, because of the larger initial k<32 backoff range, such initial collisions are unlikely.

If a Wi-Fi sender believes there has been a collision, it retries its transmission, after doubling the backoff range to 64, then 128, 256, 512, 1024 and again 1024. If these seven attempts all fail, the packet is discarded and the sender starts over.

In one slot time, radio signals move 6,000 meters; the Wi-Fi slot time – unlike that for Ethernet – has nothing to do with the physical diameter of the network. As with Ethernet, though, the Wi-Fi slot time represents the fundamental unit for backoff intervals.

Finally, we note that, unlike Ethernet collisions, Wi-Fi collisions are a local phenomenon: if A and B transmit simultaneously, a collision occurs at node C only if the signals of A and B are both strong enough at C to interfere with one another. It is possible that a collision occurs at station C midway between A and B, but not at station D that is close to A. We return to this below in *3.7.1.4 Hidden-Node Problem*.

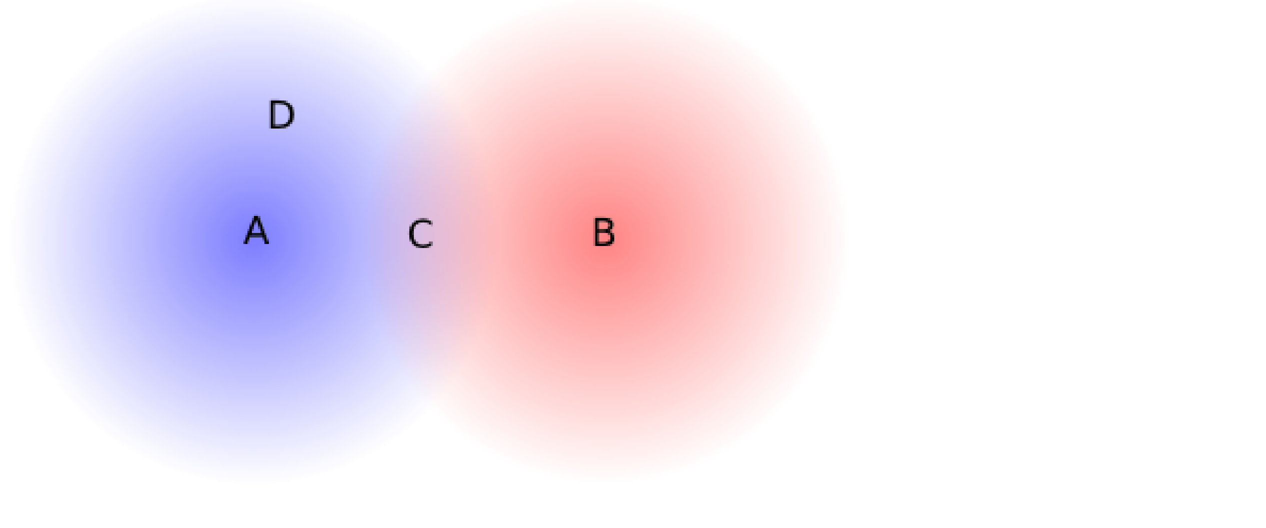
3.7.1.3 Wi-Fi RTS/CTS

Wi-Fi stations optionally also use a request-to-send/clear-to-send (**RTS/CTS**) protocol, again negotiated with designated control packets. Usually this is used only for larger data packets; often, the RTS/CTS "threshold" (the size of the largest packet *not* sent using RTS/CTS) is set (as part of the Access Point configuration, *3.7.4 Access Points*) to be the maximum packet size, effectively disabling this feature. The idea behind RTS/CTS is that a large packet that is involved in a collision represents a significant waste of potential throughput; for large packets, we should ask first.

The RTS control packet – which is small – is sent through the normal procedure outlined above; this packet includes the identity of the destination and the size of the data packet the station desires to transmit. The destination station then replies with CTS after the SIFS wait period, effectively preventing any other transmission after the RTS. The CTS packet also contains the data-packet size. The original sender then waits for SIFS after receiving the CTS, and sends the packet. If all other stations can hear both the RTS and CTS messages, then once the RTS and CTS are sent successfully no collisions should occur during packet transmission, again because the only idle times are of length SIFS and other stations should be waiting for time IFS.

3.7.1.4 Hidden-Node Problem

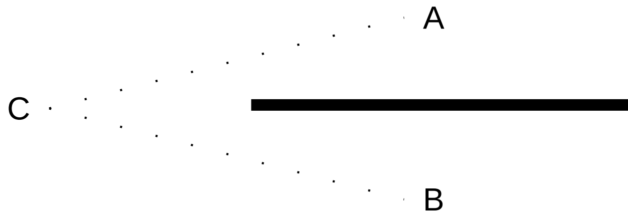
Consider the diagram below. Each station has a 100-meter range. Stations A and B are 150 meters apart and so cannot hear one another at all; each is 75 meters from C. If A is transmitting and B senses the medium in preparation for its own transmission, as part of collision avoidance, then B will conclude that the medium is idle and will go ahead and send.



However, C is within range of both A and B. If A and B transmit simultaneously, then from C's perspective a collision occurs. C receives nothing usable. We will call this a **hidden-node collision** as the senders A and B are hidden from one another; the general scenario is known as the **hidden-node problem**.

Note that node D receives only A's signal, and so no collision occurs at D.

The hidden-node problem can also occur if A and B cannot receive one another's transmissions due to a physical obstruction such as a radio-impermeable wall:



One of the rationales for the RTS/CTS protocol is the prevention of hidden-node collisions. Imagine that, instead of transmitting its data packet, A sends an RTS packet, and C responds with CTS. B has not heard the RTS packet from A, but *does* hear the CTS from C. A will begin transmitting after a SIFS interval, but B will not hear A's transmission. However, B will still wait, because the CTS packet contained the data-packet size and thus, implicitly, the length of time all other stations should remain idle. Because RTS packets are quite short, they are much less likely to be involved in collisions themselves than data packets.

3.7.1.5 Wi-Fi Fragmentation

Conceptually related to RTS/CTS is Wi-Fi **fragmentation**. If error rates or collision rates are high, a sender can send a large packet as multiple fragments, each receiving its own link-layer ACK. As we shall see in *5.3.1 Error Rates and Packet Size*, if bit-error rates are high then sending several smaller packets often leads to fewer total transmitted bytes than sending the same data as one large packet.

Wi-Fi packet fragments are reassembled by the receiving node, which may or may not be the final destination.

As with the RTS/CTS threshold, the fragmentation threshold is often set to the size of the maximum packet. Adjusting the values of these thresholds is seldom necessary, though might be appropriate if monitoring revealed high collision or error rates. Unfortunately, it is essentially impossible for an individual station to distinguish between reception errors caused by collisions and reception errors caused by other forms of noise, and so it is hard to use reception statistics to distinguish between a need for RTS/CTS and a need for fragmentation.

3.7.2 Dynamic Rate Scaling

Wi-Fi senders, if they detect transmission problems, are able to reduce their transmission bit rate in a process known as **rate scaling** or **rate control**. The idea is that lower bit rates will have fewer noise-related errors, and so as the error rate becomes unacceptably high – perhaps due to increased distance – the sender should fall back to a lower bit rate. For 802.11g, the standard rates are 54, 48, 36, 24, 18, 12, 9 and 6 Mbps. Senders attempt to find the transmission rate that maximizes throughput; for example, 36 Mbps with a packet loss

rate of 25% has an effective throughput of 36 × 75% = 27 Mbps, and so is better than 24 Mbps with no losses.

Senders may update their bit rate on a per-packet basis; senders may also choose different bit rates for different recipients. For example, if a sender sends a packet and receives no confirming link-layer ACK, the sender may fall back to the next lower bit rate. The actual bit-rate-selection algorithm lives in the particular Wi-Fi driver in use; different nodes in a network may use different algorithms.

The earliest rate-scaling algorithm was Automatic Rate Fallback, or ARF, *[KM97]*. The rate decreases after two consecutive transmission failures (that is, the link-layer ACK is not received), and increases after ten transmission successes.

A significant problem for rate scaling is that a packet loss may be due either to low-level random noise (white noise, or thermal noise) or to a collision (which is also a form of noise, but less random); only in the first case is a lower transmission rate likely to be helpful. If a larger number of collisions is experienced, the longer packet-transmission times caused by the lower bit rate may *increase* the frequency of hidden-node collisions. In fact, a *higher* transmission rate (leading to shorter transmission times) may help; enabling the RTS/CTS protocol may also help.

Signal Strength

Most Wi-Fi drivers report the received signal strength. Newer drivers use the IEEE Received Channel Power Indicator convention; the RCPI is an 8-bit integer proportional to the absolute power received by the antenna as measured in decibel-milliwatts (dBm). Wi-Fi values range from -10 dBm to -90 dBm and below. For comparison, the light from the star Polaris delivers about -97 dBm to one eye on a good night; Venus typically delivers about -73 dBm. A GPS satellite might deliver -127 dBm to your phone. (Inspired by Wikipedia on DBm.)

A variety of newer rate-scaling algorithms have been proposed; see *[JB05]* for a summary. One, Receiver-Based Auto Rate (RBAR, *[HVB01]*), attempts to incorporate the **signal-to-noise** ratio into the calculation of the transmission rate. This avoids the confusion introduced by collisions. Unfortunately, while the signal-to-noise ratio has a strong theoretical correlation with the transmission **bit-error rate**, most Wi-Fi radios will report to the host system the **received signal strength**. This is not the same as the signal-to-noise ratio, which is harder to measure. As a result, the RBAR approach has not been quite as effective in practice as might be hoped.

With the Collision-Aware Rate Adaptation algorithm (CARA, *[KKCQ06]*), a transmitting station attempts (among other things) to infer that its packet was lost to a collision rather than noise if, after one SIFS interval following the end of its packet transmission, no link-layer ACK has been received *and* the channel is still busy. This will detect collisions only when the colliding packet is *longer* than the station's own packet, and only when the hidden-node problem isn't an issue.

Because the actual data in a Wi-Fi packet may be sent at a rate not every participant is close enough to receive correctly, every Wi-Fi transmission begins with a brief preamble at the minimum bit rate. Link-layer ACKs, too, are sent at the minimum bit rate.

3.7.3 Multiple Spatial Streams

The latest innovation in improving Wi-Fi (and other wireless) data rates is to support multiple simultaneous data streams, through an antenna technique known as multiple-input-multiple-output, or **MIMO**. To use N streams, both sender and receiver must have N antennas; all the antennas use the same frequency channels but each transmitter antenna sends a different data stream. At first glance, any significant improvement in throughput might seem impossible, as the antenna elements in the respective sending and receiving groups are each within about half a wavelength of each other; indeed, in clear space MIMO is not possible.

The reason MIMO works in most everyday settings is that it puts multipath interference to positive use. Consider again at the right-hand side of the final image of *3.6.6 Multipath*, in which the signal strength varies according to the blue ripples; the peaks and valleys have a period of about half a wavelength. We will assume initially that the signal strength is low enough that reception in the darkest blue areas is no longer viable; a single antenna with the misfortune to be in one of these "dead zones" may receive nothing.

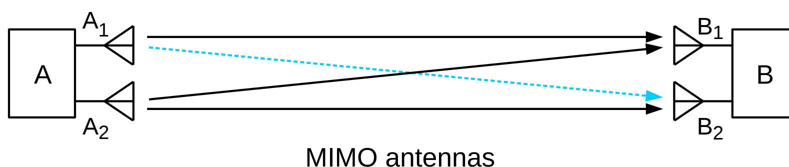
We will start with two simpler cases: SIMO (single-input-multiple-output) and MISO (multiple-input-single-output). In SIMO, the receiver has multiple antennas; in MISO, the transmitter. Assume for the moment that the multiple-antenna side has two antennas. In the simplest implementation of SIMO, the receiver picks the stronger of the two received signals and uses that alone; as long as at least one antenna is not in a "dead zone", reception is successful. With two antennas under half a wavelength apart, the odds are that at least one of them will be located outside a dead zone, and will receive an adequate signal.

Similarly, in simple MISO, the transmitter picks whichever of its antennas that gets a stronger signal to the receiver. The receiver is unlikely to be in a dead zone for *both* transmitter antennas. Note that for MISO the sender must get some feedback from the receiver to know which antenna to use.

We can do quite a bit better if signal-processing techniques are utilized so the two sender or two receiver antennas can be used simultaneously (though this complicates the mathematics considerably). Such signal-processing is standard in 802.11n and above; the Wi-Fi header, to assist this process, includes added management packets and fields for reporting MIMO-related information. One station may, for example, send the other a sequence of *training symbols* for discerning the response of the antenna system.

MISO with these added techniques is sometimes called **beamforming**: the sender coordinates its multiple antennas to maximize the signal reaching one particular receiver.

In our simplistic description of SIMO and MIMO above, in which only one of the multiple-antenna-side antennas is actually used, we have suggested that the idea is to improve marginal reception. At least one antenna on the multiple-antenna side can successfully communicate with the single antenna on the other side. **MIMO**, on the other hand, can be thought of as applying when transmission conditions are quite good all around, and every antenna on one side can reach every antenna on the other side. The key point is that, in an environment with a significant degree of multipath interference, the antenna-to-antenna paths may all be *independent*, or *uncorrelated*. At least one receiving antenna must be, from the perspective of at least one transmitting antenna, in a multipath-interference "gray zone" of reduced signal strength.



MIMO antennas

As a specific example, consider the diagram above, with two sending antennas A_1 and A_2 at the left and two receiving antennas B_1 and B_2 at the right. Antenna A_1 transmits signal S_1 and A_2 transmits S_2. There are thus four physical signal paths: A_1-to-B_1, A_1-to-B_2, A_2-to-B_1 and A_2-to-B_2. If we assume that the signal along the A_1-to-B_2 path (dashed and blue) arrives with half the strength of the other three paths (solid and black), then we have

signal received by B_1: $S_1 + S_2$
signal received by B_2: $S_1/2 + S_2$

From these, B can readily solve for the two independent signals S_1 and S_2. These signals are said to form two **spatial streams**, though the spatial streams are abstract and do not correspond to any of the four physical signal paths.

The antennas are each more-or-less omnidirectional; the signal-strength variations come from multipath interference and not from physical aiming. Similarly, while the diagonal paths A_1-to-B_2 and A_2-to-B_1 are slightly longer than the horizontal paths A_1-to-B_1 and A_2-to-B_2, the difference is not nearly enough to allow B to solve for the two signals.

In practice, overall data-rate improvement over a single antenna can be considerably less than a factor of 2 (or than N, the number of antennas at each end).

The 802.11n standard allows for up to four spatial streams, for a theoretical maximum bit rate of 600 Mbps. 802.11ac allows for up to eight spatial streams, for an even-more-theoretical maximum of close to 7 Gbps. MIMO support is sometimes described with an $A \times B \times C$ notation, *eg* $3 \times 3 \times 2$, where A and B are the number of transmitting and receiving antennas and $C \leqslant \min(A,B)$ is the number of spatial streams.

3.7.4 Access Points

There are two standard Wi-Fi configurations: **infrastructure** and **ad hoc**. The former involves connection to a designated **access point**; the latter includes individual Wi-Fi-equipped nodes communicating informally. For example, two laptops can set up an ad hoc connection to transfer data at a meeting. Ad hoc connections are often used for very simple networks *not* providing Internet connectivity. Complex ad hoc networks are, however, certainly possible; see *3.7.8 MANETs*.

The **infrastructure** configuration is much more common. Stations in an infrastructure network communicate directly *only* with their access point, which, in turn, communicates with the outside world. If Wi-Fi nodes B and C share access point AP, and B wishes to send a packet to C, then B first forwards the packet to AP and AP then forwards it to C. While this introduces a degree of inefficiency, it does mean that the access point and its associated nodes automatically act as a true LAN: every node can reach every other node. (It is also often the case that most traffic is between Wi-Fi nodes and the outside world.) In an ad hoc network, by comparison, it is common for two nodes to be able to reach each other only by forwarding through an intermediate third node; this is in fact a form of the hidden-node scenario.

Wi-Fi access points are generally identified by their **SSID** ("Service Set IDentifier"), an administratively defined human-readable string such as "linksys" or "loyola". Ad hoc networks also have SSIDs; these are generated pseudorandomly at startup and look like (but are not) 48-bit MAC addresses.

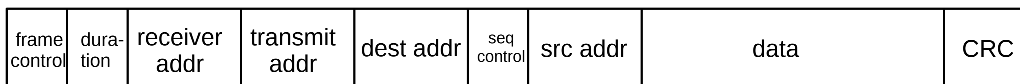
Portable Access Points

Being a Wi-Fi access point is a very specific job; Wi-Fi-enabled "station" devices like phones and workstations do not generally act as access points. However, it is often possible to for a station device to become an access point if the access-point mode is supported by the underlying radio hardware, and if suitable drivers can be found. The linux `hostapd` package is one option. The FCC may or may not bestow its blessing.

Many access points can support multiple SSIDs simultaneously. For example, an access point might support SSID "guest" with limited authentication (below), and also SSID "secure" with much stronger authentication.

Finally, Wi-Fi is by design completely interoperable with Ethernet; if station A is associated with access point AP, and AP also connects via (cabled) Ethernet to station B, then if A wants to send a packet to B it sends it using AP as the Wi-Fi destination but with B also included in the header as the "actual" destination. Once it receives the packet by wireless, AP acts as an Ethernet switch and forwards the packet to B. While this forwarding is transparent to senders, the Ethernet and Wi-Fi LAN header formats are quite different.

Ethernet



Wi-Fi Data

The above diagram illustrates an Ethernet header and the Wi-Fi header for a typical data packet (not using Wi-Fi quality-of-service features). The Ethernet type field usually moves to an IEEE Logical Link Control header in the Wi-Fi region labeled "data". The receiver and transmitter addresses are the MAC addresses of the nodes receiving and transmitting the (unicast) packet; these may each be different from the ultimate destination and source addresses. If station B wants to send a packet to station C in the same network, the source and destination are B and C but the transmitter and receiver are B and the access point. In infrastructure mode either the receiver or transmitter address is always the access point; in typical situations either the receiver is the destination or the sender is the transmitter. In ad hoc mode, if LAN-layer routing is used then all four addresses may be distinct; see *3.7.8.1 Routing in MANETs*.

3.7.4.1 Joining a Network

To join the network, an individual station must first discover its access point, and must **associate** and then **authenticate** to that access point before general communication can begin. (Older forms of authentication – so-called "open" authentication and the now-deprecated WEP authentication – came before association, but newer authentication protocols such as WPA, WPA-Personal and WPA-Enterprise (*3.7.5 Wi-Fi Security*) come after.) We can summarize the stages in the process as follows:

- scanning (or active probing)

- open-authentication and association
- true authentication
- DHCP (getting an IP address, *7.10 Dynamic Host Configuration Protocol (DHCP)*)

The association and authentication processes are carried out by an exchange of special **management packets**, which are confined to the Wi-Fi LAN layer. Occasionally stations may re-associate to their Access Point, *eg* if they wish to communicate some status update.

Access points periodically broadcast their SSID in special **beacon** packets (though for pseudo-security reasons the SSID in the beacon packets can be suppressed). Beacon packets are one of several Wi-Fi-layer-only **management packets**; the default beacon-broadcast interval is 100 ms. These broadcasts allow stations to see a list of available networks; the beacon packets also contain other Wi-Fi network parameters such as radio-modulation parameters and available data rates.

Another use of beacons is to support the power-management **doze** mode. Some stations may elect to enter this power-conservation mode, in which case they inform the access point, record the announced beacon-transmission time interval and then wake up briefly to receive each beacon. Beacons, in turn, each contain a list (in a compact bitmap form) of each dozing station for which the access point has a packet to deliver.

Ad hoc networks have beacon packets as well; all nodes participate in the regular transmission of these via a distributed algorithm.

A connecting station may either wait for the next scheduled beacon, or send a special **probe-request** packet to elicit a beacon-like probe-response packet. These operations may involve listening to or transmitting on multiple radio channels, sequentially, as the station does not yet know the correct channel to use. Unconnected stations often send probe-request packets at regular intervals, to keep abreast of available networks; it is these probe packets that allow tracking by the station's MAC address. See *3.7.4.2 MAC Address Randomization*.

Once the beacon is received, the station initiates an **association** process. There is still a vestigial open-authentication process that comes before association, but once upon a time this could also be "shared WEP key" authentication (below). Later, support for a wide range of authentication protocols was introduced, via the 802.1X framework; we return to this in *3.7.5 Wi-Fi Security*. For our purposes here, we will include open authentication as part of the association process.

Wi-Fi Drivers

Even in 2015, 100%-open-source Wi-Fi drivers are available only for selected hardware, and even then not all operations may be supported. Something as simple in principle as changing ones source Wi-Fi MAC address is sometimes not possible, though see *3.7.4.2 MAC Address Randomization*. Using multiple MAC addresses for a host plus embedded virtual machines is another problematic case.

In open authentication the station sends an authentication request to the access point and the access point replies. About all the station needs to know is the SSID of the access point, though it is usually possible to configure the access point to restrict admission to stations with MAC (physical) addresses on a pre-determined list. Stations sometimes evade MAC-address checking by changing their MAC address to an acceptable one, though some Wi-Fi drivers do not support this.

Because the SSID plays something of the role of a password here, some Wi-Fi access points are configured so that beacon packets does not contain the SSID; such access points are said to be **hidden**. Unfortunately,

access points hidden this way are easily unmasked: first, the SSID is sent in the clear by any other stations that need to authenticate, and second, an attacker can often transmit forged deauthentication or disassociation requests to force legitimate stations to retransmit the SSID. (See "management frame protection" in *3.7.5 Wi-Fi Security* for a fix to this last problem.)

The shared-WEP-key authentication was based on the (obsolete) WEP encryption mechanism (*3.7.5 Wi-Fi Security*). It involved a challenge-response exchange by which the station proved to the access point that it knew the shared WEP key. Actual WEP encryption would then start slightly later.

Once the open-authentication step is done, the next step in an infrastructure network is for the station to **associate** to the access point. This involves an association request from station to access point, and an association response in return. The primary goal of the association exchange is to ensure that the access point knows (by MAC address) what stations it can reach. This tells the access point how to deliver packets to the associating station that come from other stations or the outside world. Association is not necessary in an ad hoc network.

The entire connection process (including secure authentication, below, and DHCP, *7.10 Dynamic Host Configuration Protocol (DHCP)*), often takes rather longer than expected, sometimes several seconds. See *[PWZMTQ17]* for a discussion of some of the causes. Some station and access-point pairs appear not to work as well together as other pairs.

3.7.4.2 MAC Address Randomization

Most Wi-Fi-enabled devices are configured to transmit Wi-Fi **probe requests** at regular intervals (and on all available channels), at least when not connected. These probe requests identify available Wi-Fi networks, but they also reveal the device's MAC address. This allows sites such as stores to track customers by their device. To prevent such tracking, some devices now support **MAC address randomization**, proposed in *[GG03]*: the use at appropriate intervals of a new MAC address randomly selected by the device.

Probe requests are generally sent when the device is not joined to a network. To prevent tracking via probe requests, the simplest approach is to change the MAC address used for probes at regular, frequent intervals. A device might even change its MAC address on every probe.

Changing the MAC address used for actually *joining* a network is also important to prevent tracking, but introduces some complications. **RFC 7844** suggests these options for selecting new random addresses:

- At regular **time intervals**
- **Per connection**: each time the device connects to a Wi-Fi network, it will select a new MAC address
- **Per network**: like the above, except that if the device reconnects to the same network (identified by SSID), it will use the same MAC address

The first option, changing the joined MAC address at regular time intervals, breaks things. First, it will likely result in assignment of a new IP address to the device, terminating all existing connections. Second, many sites still authenticate – at least in part – based on the MAC address. The per-connection option prevents the first problem. The per-network option prevents both, but allows a site at which the device actually joins the network to track repeat connections. (Configuring the device to "forget" the connection between successive joins will usually prevent this, but may not be convenient.)

Another approach to the tracking problem is to disable probe requests entirely, except on explicit demand.

Wi-Fi MAC address randomization is, unfortunately, not a complete barrier to device tracking; there are other channels through which devices may leak information. For example, probe requests also contain device-capability data known as Information Elements; these values are often distinctive enough that they allow at least partial fingerprinting. Additionally, it is possible to track many Wi-Fi devices based on minute variations in the modulated signals they transmit. MAC address randomization does nothing to prevent such "radiometric identification". Access points can also impersonate other popular access points, and thus trick devices into initiating a connection with their real MAC addresses. See *[BBGO08]* and *[VMCCP16]* for these and other examples.

Finally, MAC address randomization may have applications for Ethernet as well as Wi-Fi. For example, in the original IPv6 specification, IPv6 addresses embedded the MAC address, and thus introduced the possibility of tracking a device by its IPv6 address. MAC address randomization can prevent this form of tracking as well. However, other techniques implementable solely in the IPv6 layer appear to be more popular; see *8.2.1 Interface identifiers*.

3.7.4.3 Wi-Fi Roaming

Large installations with multiple access points can create "roaming" access by assigning all the access points the same SSID. An individual station will stay with the access point with which it originally associated until the signal strength falls below a certain level (as determined by the station), at which point it will seek out other access points with the same SSID and with a stronger signal. In this way, a large area can be carpeted with multiple Wi-Fi access points, so as to look like one large Wi-Fi domain. The access points must all be connected via a wired LAN, known as the **distribution system**. At any one time, a station may be associated to only one access point. In 802.11 terminology, a multiple-access-point configuration is known as an "extended service set" or **ESS**.

In order for such a large-area network to work, traffic *to* a wireless station, *eg* B, must find that station's current access point, *eg* AP. This is a job for the distribution system. If the distribution system is a switched Ethernet supporting the usual learning mechanism (*2.4 Ethernet Switches*), one simple approach is to handle this the same way as, in a wired Ethernet, traffic finds a laptop that has been unplugged, carried to a new building, and plugged in again. Suppose B is a wireless node that has been exchanging packets via the distribution system with wired node C (perhaps a router connecting B to the Internet). When B moves to a new access point, all it has to do is send any packet over the LAN to C, and the Ethernet switches on the path from B to C will then learn the route through the switched Ethernet from C back to B's new AP, and thus to B. It is also possible for B's new AP to send this switch-updating packet.

This process may leave other switches in the distribution system still holding in their forwarding tables the old location for B. This is not terribly serious, as it will be fixed for any one switch as soon as B sends a packet to a destination reached by that switch. The problem can be avoided entirely if, after moving, B (or, again, its new AP) sends out an Ethernet broadcast packet.

The IEEE 802.11r amendment is an effort to standardize fast handoffs from one access point to another within the same ESS. It allows the station and new access point to reuse the same pairwise master keys (below) that had been negotiated between the station and the old access point. It also slightly streamlines the dissociation/reassociation process. Transitions must still be initiated by the station.

Some mobile station devices are not very good about initiating handoffs to another access point, however. Such devices may cling to their original access point well past the distance at which the original signal ceases to provide reasonable bandwidth, as long as it does not vanish entirely. Some access points designed for ESS

use attempt to overcome this, by communicating amongst themselves and detecting when a station's signal is weak enough that a handoff would be appropriate. One approach involves having the original access point initiate a dissociation. At that point the station will reconnect to the ESS but hopefully will now connect to an access point within the ESS that has a stronger signal. Another approach involves having the access points all use the same MAC address, so they are indistinguishable. Whichever access point receives the strongest signal from the station is the one used to transmit *to* the station.

3.7.5 Wi-Fi Security

Unencrypted Wi-Fi traffic is visible to anyone nearby with an appropriate receiver; this eavesdropping zone can be expanded by use of a larger antenna. Because of this, Wi-Fi security is important, and Wi-Fi supports several types of traffic encryption.

The original – and now obsolete – Wi-Fi encryption standard was Wired-Equivalent Privacy, or **WEP**. It involved a 5-byte key, later sometimes extended to 13 bytes. The encryption algorithm was based on RC4, *22.7.4.1 RC4*. The key was a **pre-shared key**, manually configured into each station.

Because of the specific way WEP made use of the RC4 cipher, it contained a fatal (and now-classic) flaw. Bytes of the key could be could be "broken" – that is, guessed – sequentially. Knowing bytes 0 through i–1 would allow an attacker to guess byte i with a relatively small amount of data, and so on through the entire key. See *22.7.7 Wi-Fi WEP Encryption Failure* for details.

WEP was replaced with Wi-Fi Protected Access, or **WPA**. This used the so-called TKIP encryption algorithm that, like WEP, was ultimately based on RC4, but which was immune to the sequential attack that made WEP so vulnerable. WPA was later replaced by WPA2 as part of the IEEE 802.11i amendment, which uses the presumptively stronger AES encryption (*22.7.2 Block Ciphers*). WPA2 encryption is believed to be quite secure, although there was a vulnerability in the associated Wi-Fi Protected Setup protocol. In the 802.11i standard, WPA2 is known as the **robust security network** protocol. Access points supporting WPA or WPA2 declare this in their beacon and probe-response packets; these packets also include a list of acceptable ciphers.

WPA2 (and WPA) comes in two flavors: **WPA2-Personal** and **WPA2-Enterprise**. These use the same AES encryption, but differ in how keys are managed. WPA2-Personal, appropriate for many smaller sites, uses a pre-shared master key, known as the PSK. This key must be entered into the Access Point (ideally not over the air) and into each connecting station. The key is usually a secure hash (*22.6 Secure Hashes*) of a passphrase. The use of a common key for multiple stations makes changing the key, or revoking the key for a particular user, difficult.

3.7.5.1 Four-way handshake

In any secure Wi-Fi authentication protocol, the station must authenticate to the access point *and* the access point must authenticate to the station; without the latter part, stations might inadvertently connect to rogue access points, which would then likely gain at least partial access to private data. This bidirectional authentication is achieved through the so-called **four-way handshake**, which also generates a **session key**, known as the pairwise *transient* key or **PTK**, that is independent of the master key. Compromise of the PTK should not allow an attacker to determine the master key. In WPA2-Personal, the master key is the pre-shared key (PSK); in WPA2-Enterprise, below, the master key is the negotiated PMK. The four-way handshake begins immediately after association and, for WPA2-Enterprise, the selection of the PMK.

Both station and access point begin by each selecting a random string, called a **nonce**, typically 32 bytes long. The PTK will be a secure hash of the master key, both nonces, and both MAC addresses. The first packet of the four-way handshake is sent by the access point to the station, and contains its nonce, unencrypted.

At this point the station has enough information to compute the PTK; in the second message of the handshake it now sends its own nonce to the access point. The nonce is again sent in the clear, but this second message also includes a digital signature based on the PTK. This signature is sometimes called a Message Integrity Code, or MIC, and in the 802.11i standard is officially named *Michael*. It is calculated in a manner similar to the HMAC mechanism of *22.6.1 Secure Hashes and Authentication*. Upon receipt of the station's nonce, the access point too is able to compute the PTK. With the PTK now in hand, the access point verifies the attached signature. If it checks out, that proves to the access point that the station did in fact know the master key, as a valid signature could not have been constructed without it. The station has now authenticated itself to the access point.

For the third step, the access point, now also in possession of the PTK, sends a signed message to the station; this message includes a starting sequence number for future message numbering. When this message is received and verified, the access point has authenticated itself to the station. The fourth and final step is simply an acknowledgment from the client.

Four-way-handshake packets are sent in the EAPOL format, described in the following section. This format can be used to identify the handshake packets in WireShark scans.

3.7.5.2 WPA2-Enterprise

The **WPA2-Enterprise** alternative allows each station to have its own separate key. In fact, it largely separates the encryption mechanisms from the Wi-Fi protocols, allowing sites great freedom in choosing the former. Despite the "enterprise" in the name, it is also well suited for smaller sites. WPA2-Enterprise is based rather closely on the 802.1X framework, which supports arbitrary authentication protocols as plug-in modules.

The keys are all held by a single common system known as the **authentication server**, usually unrelated to the access point. The client node (that is, the Wi-Fi station) is known as the **supplicant**, and the access point is known as the **authenticator**.

To begin the authentication process, the supplicant contacts the authenticator using the Extensible Authentication Protocol, or **EAP**, with what amounts to a request to authenticate to that access point. EAP is a generic message framework meant to support multiple specific types of authentication; see **RFC 3748** and **RFC 5247**. The EAP request is forwarded to the authentication server, which may exchange (via the authenticator) several challenge/response messages with the supplicant. No secret credentials should be sent in the clear.

EAP is usually used in conjunction with the RADIUS (Remote Authentication Dial-In User Service) protocol (**RFC 2865**), which is a specific (but flexible) authentication-server protocol. WPA2-Enterprise is sometimes known as 802.1X mode, EAP mode or RADIUS mode (though WPA2-Personal is also based on 802.1X, and uses EAP in its four-way handshake).

EAP communication takes place before the supplicant is given an IP address; thus, a mechanism must be provided to support exchange of EAP packets between supplicant and authenticator. This mechanism is known as EAPOL, for EAP Over LAN. EAP messages between the authenticator and the authentication

server, on the other hand, can travel via IP; in fact, sites may choose to have the authentication server hosted remotely. Specific protocols using the EAP/RADIUS framework often use packet formats other than EAPOL, but EAPOL will be used in the concluding four-way handshake.

Once the authentication server (*eg* RADIUS server) is set up, specific per-user authentication methods can be entered. This can amount to ⟨username,password⟩ pairs (below), or some form of security certificate, or sometimes both. The authentication server will generally allow different encryption protocols to be used for different supplicants, thus allowing for the possibility that there is not a common protocol supported by all stations.

In WPA2-Enterprise, the access point no longer needs to know anything about what authentication protocol is actually used; it is simply the middleman forwarding EAP packets between the supplicant and the authentication server. In particular, the access point does not need to support any specific authentication protocol. The access point allows the supplicant to connect to the network once it receives permission to do so from the authentication server.

At the end of the authentication process, the supplicant and the authentication server will, as part of that process, also have established a shared secret. In WPA2-Enterprise terminology this is known as the **pairwise master key** or PMK. The authentication server then communicates the PMK securely to the access point (using any standard protocol; see *22.10 SSH and TLS*). The next step is for the supplicant and the access point to negotiate their session key. This is done using the four-way-handshake mechanism of the previous section, with the PMK as the master key. The resultant PTK is, as with WPA2-Personal, used as the session key.

WPA2-Enterprise authentication typically does require that the access point have an IP address, in order to be able to contact the authentication server. An access point using WPA2-Personal authentication does not need an IP address, though it may have one simply to enable configuration.

3.7.5.3 Enabling WPA2-Enterprise

Configuring a Wi-Fi network to use WPA2-Enterprise authentication is relatively straightforward, as long as an authentication server running RADIUS is available. We here give an outline of setting up WPA2-Enterprise authentication using FreeRADIUS (version 2.1.12, 2018). We want to enable per-user passwords, but *not* per-user certificates. Passwords will be stored on the server using SHA-1 hashing (*22.6 Secure Hashes*). This is not necessarily strong enough for production use; see *22.6.2 Password Hashes* for other options. Because passwords will be hashed, the client will have to communicate the actual password to the authentication server; authentication methods such as those in *22.6.3 CHAP* are not an option.

The first step is to set up the access point. This is generally quite straightforward; WPA2-Enterprise is supported even on inexpensive access points. After selecting the option to enable WPA2-Enterprise security, we will need to enter the IP address of the authentication server, and also a "shared secret" password for authenticating messages between the access point and the server (see *22.6.1 Secure Hashes and Authentication* for message-authentication techniques).

Configuration of the RADIUS server is a bit more complex, as both RADIUS and EAP are both quite general; both were developed long before 802.1X, and both are used in many other settings as well. Because we have decided to use hashed passwords – which implies the client station will send the plaintext password to the authentication server – we will need to use an authentication method that creates an encrypted tunnel. The Protected EAP method is well-suited here; it encrypts its traffic using TLS (*22.10.2 TLS*, though here

without TCP). (There is also an EAP TLS method, using TLS directly and traditionally requiring client-side certificates, and a TTLS method, for Tunneled TLS.)

Within the PEAP encrypted tunnel, we want to use plaintext password authentication. Here we want the Password Authentication Protocol, PAP, which basically just asks for the username and password. FreeRADIUS does not allow PAP to run directly within PEAP, so we interpose the Generic Token Card protocol, GTC. (There is no "token card" device anywhere in sight, but GTC is indeed quite generic.)

We enable all these things by editing the `eap.conf` file so as to contain the following entries:

```
default_eap_type = peap

...

peap {
        default_eap_type = gtc
        ...
}

...

gtc {
        auth_type = PAP
        ...
}
```

The next step is to create a (username, hashed_password) credential pair on the server. To keep things simple, we will store credentials in the `users` file. The username will be "alice", with password "snorri". Per the FreeRADIUS rules, we need to convert the password to its SHA-1 hash, encoded using base64. There are several ways to do this; we will here make use of the OpenSSL command library:

```
echo -n "snorri" | openssl dgst -binary -sha1 | openssl base64
```

This returns the string `7E6FbhrN2TYOkrBti+8W8weC2W8=` which we then enter into the `users` file as follows:

```
alice   SHA1-Password := "7E6FbhrN2TYOkrBti+8W8weC2W8="
```

Other options include `Cleartext-Password`, `MD5-Password` and `SSHA1-Password`, with the latter being for salted passwords (which are recommended).

With this approach, Alice will have difficulty changing her password, unless she is administrator of the authentication server. This is not necessarily worse than WPA2-Personal, where Alice shares her password with other users. However, if we want to support user-controlled password changing, we can configure the RADIUS server to look for the (username, hashed_password) credentials in a database instead of the `users` file. It is then relatively straightforward to create a web-based interface for allowing users to change their passwords.

Now, finally, we try to connect. Any 802.1X client should ask for the username and password, before communication with the authentication server begins. Some may also ask for a preferred authentication method (though our RADIUS server here is only offering one), an optional certificate (which we are not using), or an "anonymous identity", which is a way for a client to specify a particular authentication server if there are several. If all goes well, connection should be immediate. If not, FreeRADIUS has an authentication-testing

tool, and copious debugging output.

3.7.5.4 Encryption Coverage

Originally, encryption covered only the data packets. A common attack involved forging management packets, *eg* to force stations to disassociate from their access point. Sometimes this was done continuously as a denial-of-service attack; it might also be done to force a station to reassociate and thus reveal a hidden SSID, or to reveal key information to enable a brute-force decryption attack.

The 2009 IEEE 802.11w amendment introduced the option for a station and access point to negotiate **management frame protection**, which encrypts (and digitally signs) essential management packets exchanged *after* the authentication phase is completed. This includes those station-to-access-point packets requesting deauthentication or disassociation, effectively preventing the above attacks. However, management frame protection is (as of 2015) seldom enabled by default by consumer-grade Wi-Fi access points, even when data encryption is in effect.

3.7.6 Wi-Fi Monitoring

Again depending on ones driver, it is sometimes possible to monitor all Wi-Fi traffic on a given channel. Special tools exist for this, including aircrack-ng and kismet, but often plain WireShark will suffice if one can get the underlying driver into so-called "monitor" mode. On linux systems the command `iwconfig wlan0 mode monitor` should do this (where `wlan0` is the name of the wireless network interface). It may be necessary to first kill other processes that have the `wlan0` interface open, *eg* with `service NetworkManager stop`. It may also be necessary to bring the interface down, with `ifconfig wlan0 down`, in which case the interface needs to be brought back `up` after entering monitor mode. Finally, the receive channel can be set with, *eg*, `iwconfig wlan0 channel 6`. (On some systems the interface name may change after the transition to monitor mode.)

After the mode and channel are set, Wireshark will report the 802.11 management-frame headers, and also the so-called radiotap header containing information about the transmission data rate, channel, and received signal strength.

One useful experiment is to begin monitoring and then to power up a Wi-Fi enabled device. The WireShark display filter `wlan.addr ==` *device-MAC-address* helps focus on the relevant packets(or, better yet, the capture filter `ether host` *device-MAC-address*). The WireShark screenshot below is an example.

```
 7 0.084938505   SamsungE_03:ef:ad   Broadcast           802.11   158 Probe Request, SN=23, FN=0, Flags=........C
 8 0.086005244   Cisco-Li_d1:24:40   SamsungE_03:ef:ad   802.11   139 Probe Response, SN=1230, FN=0, Flags=......
 9 0.115989487   SamsungE_03:ef:ad   Cisco-Li_d1:24:40   802.11    75 Authentication, SN=24, FN=0, Flags=........
10 0.116954409   Cisco-Li_d1:24:40   SamsungE_03:ef:ad   802.11    72 Authentication, SN=1231, FN=0, Flags=......
11 0.118707312   SamsungE_03:ef:ad   Cisco-Li_d1:24:40   802.11   131 Association Request, SN=25, FN=0, Flags=...
12 0.119764782   Cisco-Li_d1:24:40   SamsungE_03:ef:ad   802.11    88 Association Response, SN=1232, FN=0, Flags=
13 0.120082784   Cisco-Li_d1:24:40   SamsungE_03:ef:ad   EAPOL    165 Key (Message 1 of 4)
14 0.148433526   SamsungE_03:ef:ad   Cisco-Li_d1:24:40   EAPOL    189 Key (Message 2 of 4)
15 0.151498719   Cisco-Li_d1:24:40   SamsungE_03:ef:ad   EAPOL    191 Key (Message 3 of 4)
16 0.154637132   SamsungE_03:ef:ad   Cisco-Li_d1:24:40   EAPOL    165 Key (Message 4 of 4)
```

we see node SamsungE_03:3f:ad broadcast a probe request, which is answered by the access point Cisco-Li_d1:24:40. The next two packets represent the open-authentication process, followed by two packets representing the association process. The last four packets, of type `EAPOL`, represent the WPA2-Personal four-way authentication handshake.

3.7.7 Wi-Fi Polling Mode

Wi-Fi also includes a "polled" mechanism, where one station (the Access Point) determines which stations are allowed to send. While it is not often used, it has the potential to greatly reduce collisions, or even eliminate them entirely. This mechanism is known as "Point Coordination Function", or PCF, versus the collision-oriented mechanism which is then known as "Distributed Coordination Function". The PCF name refers to the fact that in this mode it is the Access Point that is in charge of coordinating which stations get to send when.

The PCF option offers the potential for regular traffic to receive improved throughput due to fewer collisions. However, it is often seen as intended for real-time Wi-Fi traffic, such as voice calls over Wi-Fi.

The idea behind PCF is to schedule, at regular intervals, a contention-free period, or **CFP**. During this period, the Access Point may

- send Data packets to any receiver
- send **Poll** packets to any receiver, allowing that receiver to reply with its own data packet
- send a combination of the two above (not necessarily to the same receiver)
- send management packets, including a special packet marking the end of the CFP

None of these operations can result in a collision (unless an unrelated but overlapping Wi-Fi domain is involved).

Stations receiving data from the Access Point send the usual ACK after a SIFS interval. A data packet from the Access Point addressed to station B may also carry, piggybacked in the Wi-Fi header, a Poll request to another station C; this saves a transmission. Polled stations that send data will receive an ACK from the Access Point; this ACK may be combined in the same packet with the Poll request to the next station.

At the end of the CFP, the regular "contention period" or CP resumes, with the usual CSMA/CA strategy. The time interval between the start times of consecutive CFP periods is typically 100 ms, short enough to allow some real-time traffic to be supported.

During the CFP, all stations normally wait only the Short IFS, SIFS, between transmissions. This works because normally there is only one station designated to respond: the Access Point or the polled station. However, if a station is polled and has nothing to send, the Access Point waits for time interval **PIFS** (PCF Inter-Frame Spacing), of length midway between SIFS and IFS above (our previous IFS should now really be known as DIFS, for DCF IFS). At the expiration of the PIFS, any non-Access-Point station that happens to be unaware of the CFP will continue to wait the full DIFS, and thus will not transmit. An example of such a CFP-unaware station might be one that is part of an entirely different but overlapping Wi-Fi network.

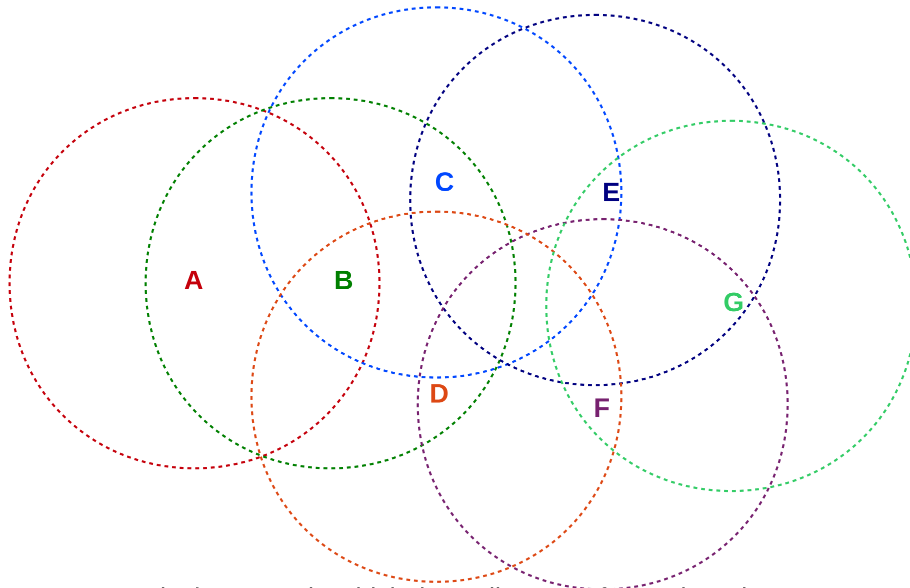
The Access Point generally maintains a **polling list** of stations that wish to be polled during the CFP. Stations request inclusion on this list by an indication when they associate or (more likely) reassociate to the Access Point. A polled station with nothing to send simply remains quiet.

PCF mode is not supported by many lower-end Wi-Fi routers, and often goes unused even when it is available. Note that PCF mode is collision-free, *so long as no other Wi-Fi access points are active and within range*. While the standard has some provisions for attempting to deal with the presence of other Wi-Fi networks, these provisions are somewhat imperfect; at a minimum, they are not always supported by other access points. The end result is that polling is not quite as useful as it might be.

3.7.8 MANETs

The MANET acronym stands for **mobile ad hoc network**; in practice, the term generally applies to ad hoc wireless networks of sufficient complexity that some internal routing mechanism is needed to enable full connectivity. The term **mesh network** is also used. While MANETs can use any wireless mechanism, we will assume here that Wi-Fi is used.

MANET nodes communicate by radio signals with a finite range, as in the diagram below.



Typical MANET in which the radio range for each node is represented by a circle around that node. A can reach G either by the route A—B—C—E—G or by A—B—D—F—G.

Each node's radio range is represented by a circle centered about that node. In general, two MANET nodes may be able to communicate only by relaying packets through intermediate nodes, as is the case for nodes A and G in the diagram above.

In the field, the radio range of each node may not be very circular at all, due to among other things signal reflection and blocking from obstructions. An additional complication arises when the nodes (or even just obstructions) are moving in real time (hence the "mobile" of MANET); this means that a working route may stop working a short time later. For this reason, and others, routing within MANETs is a good deal more complex than routing in an Ethernet. A switched Ethernet, for example, is required to be loop-free, so there is never a choice among multiple alternative routes.

Note that, without successful LAN-layer routing, a MANET does not have full node-to-node connectivity and thus does not meet the definition of a LAN given in *1.9 LANs and Ethernet*. With either LAN-layer or IP-layer routing, one or more MANET nodes may serve as gateways to the Internet.

Note also that MANETs in general do not support broadcast or multicast, unless the forwarding of broadcast and multicast messages throughout the MANET is built in to the routing mechanism. This can complicate the operation of IPv4 and IPv6 networks, even assuming that the MANET routing mechanism replaces the need for broadcast/multicast protocols like IPv4's ARP (*7.9 Address Resolution Protocol: ARP*) and IPv6's Neighbor Discovery (*8.6 Neighbor Discovery*) that otherwise play important roles in local packet delivery.

For example, the common IPv4 address-assignment mechanism we will describe in *7.10 Dynamic Host Configuration Protocol (DHCP)* relies on broadcast and so often needs adaptation. Similarly, IPv6 relies on multicast for several ancillary services, including address assignment (*8.7.3 DHCPv6*) and duplicate address detection (*8.7.1 Duplicate Address Detection*).

Finally, we observe that while MANETs are of great theoretical interest, their practical impact has been modest; they are almost unknown, for example, in corporate environments. They appear most useful in emergency situations, rural settings, and settings where the conventional infrastructure network has failed or been disabled.

3.7.8.1 Routing in MANETs

Routing in MANETs can be done either at the LAN layer, using physical addresses, or at the IP layer with some minor bending (below) of the rules.

Either way, nodes must find out about the existence of other nodes, and appropriate routes must then be selected. Route selection can use any of the mechanisms we describe later in *9 Routing-Update Algorithms*.

Routing at the LAN layer is much like routing by Ethernet switches; each node will construct an appropriate forwarding table. Unlike Ethernet, however, there may be multiple paths to a destination, direct connectivity between any particular pair of nodes may come and go, and negotiation may be required even to determine which MANET nodes will serve as forwarders.

Routing at the IP layer involves the same issues, but at least IP-layer routing-update algorithms have always been able to handle multiple paths. There are some minor issues, however. When we initially presented IP forwarding in *1.10 IP - Internet Protocol*, we assumed that routers made their decisions by looking only at the network prefix of the address; if another node had the same network prefix it was assumed to be reachable directly via the LAN. This model usually fails badly in MANETs, where direct reachability has nothing to do with addresses. At least within the MANET, then, a modified forwarding algorithm must be used where *every* address is looked up in the forwarding table. One simple way to implement this is to have the forwarding tables contain only **host-specific** entries as were discussed in *3.1 Virtual Private Networks*.

Multiple routing algorithms have been proposed for MANETs. Performance of a given algorithm may depend on the following factors:

- The size of the network
- Whether some nodes have agreed to serve as routers
- The degree of node mobility, especially of routing-node mobility if applicable
- Whether the nodes are under common administration, and thus may agree to defer their own transmission interests to the common good
- per-node storage and power availability

3.8 WiMAX and LTE

WiMAX and LTE are both wireless network technologies suitable for data connections to mobile (and sometimes stationary) devices.

WiMAX is an IEEE standard, 802.16; its original name is WirelessMAN (for Metropolitan Area Network), and this name appears intermittently in the IEEE standards. In its earlier versions it was intended for stationary subscribers (802.16d), but was later expanded to support mobile subscribers (802.16e). The stationary-subscriber version is often used to provide residential Internet connectivity, in both urban and rural areas.

LTE (the acronym itself stands for Long Term Evolution) is a product of the mobile telecom world; it was designed for mobile subscribers from the beginning. Its official name – at least for its radio protocols – is **Evolved UTRA**, or E-UTRA, where UTRA in turn stands for UMTS Terrestrial Radio Access. UMTS stands for Universal Mobile Telecommunications System, a core mobile-device data-network mechanism with standards dating from the year 2000.

4G Capacity

A medium-level wireless data plan often comes with a 5 GB monthly cap. At the 100 Mbps 4G data rate, that allotment can be downloaded in under six minutes. Data rate isn't everything.

Both LTE and the mobile version of WiMAX are often marketed as **fourth generation** (or 4G) networking technology. The ITU has a specific definition for 4G developed in 2008, officially named IMT-Advanced and including a 100 Mbps download rate to moving devices and a 1 Gbps download rate to more-or-less-stationary devices. Neither WiMAX nor LTE quite qualified technically, but to marketers that was no impediment. In any event, in December 2010 the ITU issued a statement in which it "recognized that [the term 4G], while undefined, may also be applied to the forerunners of these technologies, LTE and WiMax". So-called Advanced LTE and WiMAX2 are true IMT-Advanced protocols.

As in *3.6.4 Band Width* we will use the term "data rate" for what is commonly called "bandwidth" to avoid confusion with the radio-specific meaning of the latter term.

WiMAX can use unlicensed frequencies, like Wi-Fi, but its primary use is over licensed radio spectrum; LTE is used almost exclusively over licensed spectrum.

WiMAX and LTE both support a number of options for the width of the frequency band; the wider the band, the higher the data rate. Downlink (base station to subscriber) data rates can be well over 100 Mbps (uplink rates are usually smaller). Most LTE bands are either in the range 700-900 MHz or are above 1700 MHz; the lower frequencies tend to be better at penetrating trees and walls.

Like Wi-Fi, WiMAX and LTE **subscriber stations** connect to a central access point. The WiMAX standard prefers the term **base station** which we will use henceforth for both protocols; LTE officially prefers the term "evolved NodeB" or eNB.

The coverage radius for LTE and mobile-subscriber WiMAX might be one to ten kilometers, versus less (sometimes much less) than 100 meters for Wi-Fi. Stationary-subscriber WiMAX can operate on a larger scale; the coverage radius can be several tens of kilometers. As distances increase, the data rate is reduced.

Large-radius base stations are typically mounted in towers; smaller-radius base-stations, generally used only in areas densely populated with subscribers, may use lower antennas integrated discretely into the local architecture. Subscriber stations are not expected to be able to hear other stations; they interact only with the base station.

3.8.1 Uplink Scheduling

As distances increase, the subscriber-to-base RTT becomes non-negligible. At 10 kilometers, this RTT is 66 µsec, based on the speed of light of about 300 m/µsec. At 100 Mbps this is enough time to send 800 bytes, making it a priority to reduce the number of RTTs. To this end, it is no longer practical to use Wi-Fi-style collisions to resolve access contention; it is not even practical to use the Wi-Fi PCF mode of *3.7.7 Wi-Fi Polling Mode* because polling requires additional RTTs. Instead, WiMAX and LTE rely on base-station-regulated **scheduling** of transmissions.

The base station has no difficulty scheduling downlink transmissions, from base to subscriber: the base station simply sends the packets sequentially (or in parallel on different sets of subcarriers if OFDM is used). If beamforming MISO antennas are used, or multiple physically directional antennas, the base station will take this into account.

It is the uplink transmissions – from subscriber to base – that are more complicated to coordinate. Once a subscriber station completes the **network entry** process to connect to a base station (*3.8.3 Network Entry*), it is assigned regular transmission slots, including times and frequencies. These transmission slots may vary in size over time; the base station may regularly issue new transmission schedules. Each subscriber station is told in effect that it may transmit on its assigned frequencies starting at an assigned time and for an assigned length; LTE lengths start at 1 ms and WiMAX lengths at 2 ms. The station synchronizes its clock with that of the base station as part of the network entry process.

Each subscriber station is scheduled to transmit so that one transmission finishes arriving at the base station just before the next station's same-frequency transmission begins arriving. Only minimal "guard intervals" need be included between consecutive transmissions. Two (or more) consecutive uplink transmissions may in fact be "in the air" simultaneously, as far-away stations need to begin transmitting early so their signals will arrive at the base station at the expected time.

The diagram above illustrates this for stations separated by relatively large physical distances (as may be typical for long-range WiMAX). This strategy for uplink scheduling eliminates the full RTT that Wi-Fi polling mode (*3.7.7 Wi-Fi Polling Mode*) entails.

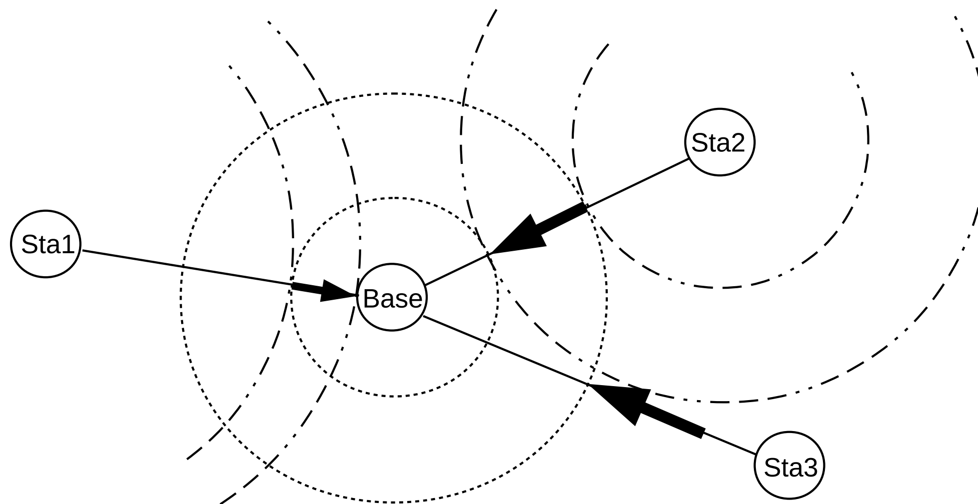
Scheduled timeslots may be periodic (as is would be appropriate for voice) or may occur at varying intervals. Quality-of-Service requests may also enter into the schedule; LTE focuses on end-to-end QoS while WiMAX focuses on subscriber-to-base QoS.

When a station has data to send, it may include in its next scheduled transmission a request for a longer transmission interval; if the request is granted, the station may send its data (or at least some of its data) in its *next* scheduled transmission slot. When a station is done transmitting, its timeslot may shrink back to the minimum, and may be scheduled less frequently as well, but it does not disappear. Stations without data to send remain connected to the base station by sending "empty" messages during these slots.

3.8.2 Ranging

The uplink scheduling of the previous section requires that each subscriber station know the distance to the base station. If a subscriber station is to transmit so that its message arrives at the base station at a certain time, it must actually begin transmission early by an amount equal to the one-way station-to-base propagation delay. This distance/delay measurement process is called **ranging**.

Ranging can be accomplished through any RTT measurement. Any base-station delay in replying, once a



Three packets in transit from stations Sta1, Sta2 and Sta3. The packets propagate outwards from the stations at the speed of light, like ripples, spatially confined between two concentric dot-dash circles (circles around Sta3 are not shown). The packet portion along the straight line from the station to the Base is represented as a heavy arrow. The three packets will arrive at Base sequentially and without overlap.

subscriber message is received, simply needs to be subtracted from the total RTT. Of course, that base-station delay needs also to be communicated back to the subscriber.

The distance to the base station is used not only for the subscriber station's transmission timing, but also to determine its power level; signals from each subscriber station, no matter where located, should arrive at the base station with about the same power.

3.8.3 Network Entry

The scheduling process eliminates the potential for collisions between normal data transmissions. But there remains the issue of initial entry to the network. If a **handoff** is involved, the new base station can be informed by the old base station, and send an appropriate schedule to the moving subscriber station. But if the subscriber station was just powered on, or is arriving from an area without LTE/WiMAX coverage, potential transmission collisions are unavoidable. Fortunately, network entry is infrequent, and so collisions are even less frequent.

A subscriber station begins the network-entry connection process to a base station by listening for the base station's transmissions; these message streams contain regular management messages containing, among other things, information about available data rates in each direction. Also included in the base station's message stream is information about when network-entry attempts can be made.

In WiMAX these entry-attempt timeslots are called **ranging intervals**; the subscriber station waits for one of these intervals and sends a "range-request" message to the base station. These ranging intervals are open to all stations attempting network entry, and if another station transmits at the same time there will be a collision. An Ethernet/Wi-Fi-like exponential-backoff process is used if a collision does occur.

In LTE the entry process is known as **RACH**, for Random Access CHannel. The base station designates certain 1 ms timeslots for network entry. During one of these slots an entry-seeking subscriber chooses at random one of up to 64 predetermined **random access preambles** (some preambles may be reserved for a second, contention-free form of RACH), and transmits it. The 1-ms timeslot corresponds to 300 kilometers, much larger than any LTE cell, so the fact that the subscriber does not yet know its distance to the base does not matter.

The preambles are mathematically "orthogonal", in such a way that as long as no two RACH-participating subscribers choose the same preamble, the base station can decode overlapping preambles and thus receive the *set* of all preambles transmitted during the RACH timeslot. The base station then sends a reply, listing the preambles received and, in effect, an initial schedule indexed by preamble of when each newly entering subscriber station can transmit actual data. This reply is sent to a special temporary multicast address known as a *radio network temporary identifier*, or RNTI, as the base station does not yet know the actual identity of any new subscriber. Those identities are learned as the new subscribers transmit to the base station according to this initial schedule.

A collision occurs when two LTE subscriber stations have the misfortune of choosing the same preamble in the same RACH timeslot, in which case the chosen preamble will not appear in the initial schedule transmitted by the base station. As for WiMAX, collisions are rare because network entry is rare. Subscribers experiencing a collision try again during the next RACH timeslot, choosing at random a new preamble.

For both WiMAX and LTE, network entry is the only time when collisions can occur; afterwards, all subscriber-station transmissions are scheduled by the base station.

If there is no collision, each subscriber station is able to use the base station's initial-response transmission to make its first ranging measurement. Subscribers must have a ranging measurement in hand before they can send any scheduled transmission.

3.8.4 Mobility

There are some significant differences between stationary and mobile subscribers. First, mobile subscribers will likely expect some sort of **handoff** from one base station to another as the subscriber moves out of range of the first. Second, moving subscribers mean that the base-to-subscriber ranging information may change rapidly; see exercise 7.0. If the subscriber does not update its ranging information often enough, it may transmit too early or too late. If the subscriber is moving fast enough, the Doppler effect may also alter frequencies.

3.9 Fixed Wireless

This category includes all wireless-service-provider systems where the subscriber's location does not change. Often, but not always, the subscriber will have an outdoor antenna for improved reception and range. Fixed-wireless systems can involve relay through satellites, or can be **terrestrial**.

3.9.1 Terrestrial Wireless

Terrestrial wireless – also called terrestrial broadband or fixed-wireless broadband – involves direct (non-satellite) radio communication between subscribers and a central access point. Access points are usually

tower-mounted and serve multiple subscribers, though single-subscriber point-to-point "microwave links" also exist. A multi-subscriber access point may serve an area with radius up to several tens of miles, depending on the technology, though more common ranges are under ten miles. WiMAX 802.16d is one form of terrestrial wireless, but there are several others. Frequencies may be either licensed or unlicensed. Unlicensed frequency bands are available at around 900 MHz, 2.4 GHz, and 5 GHz. Nominally all three bands require that line-of-sight transmission be used, though that requirement becomes stricter as the frequency increases. Lower frequencies tend to be better at "seeing" through trees and other obstructions.

Trees vs Signal

Photo of the author attempting to improve his 2.4 GHz terrestrial-wireless signal via tree trimming.

Terrestrial fixed wireless was originally popularized for rural areas, where residential density is too low for economical cable connections. However, some fixed-wireless ISPs now operate in urban areas, often using

WiMAX. One advantage of terrestrial fixed-wireless in remote areas is that the antennas covers a much smaller geographical area than a satellite, generally meaning that there is more data bandwidth available per user and the cost per megabyte is much lower.

Outdoor subscriber antennas often use a parabolic dish to improve reception; sizes range from 10 to 50 cm in diameter. The size of the dish may depend on the distance to the central tower.

While there are standardized fixed-wireless systems, such as WiMAX, there are also a number of proprietary alternatives, including systems from Trango and Canopy. Fixed-wireless systems might, in fact, be considered one of the last bastions of proprietary LAN protocols. This lack of standardization is due to a variety of factors; two primary ones are the relatively modest overall demand for this service and the the fact that most antennas need to be professionally installed by the ISP to ensure that they are "properly mounted, aligned, grounded and protected from lightning".

3.9.2 Satellite Internet

An extreme case of fixed wireless is **satellite Internet**, in which signals pass through a satellite in geosynchronous orbit (35,786 km above the earth's surface). Residential customers have parabolic antennas typically from 70 to 100 cm in diameter, larger than those used for terrestrial wireless but smaller than the dish antennas used at access points. The geosynchronous satellite orbit means that the antennas need to be pointed only once, at installation. Transmitter power is typically 1-2 watts, remarkably low for a signal that travels 35,786 km.

The primary problem associated with satellite Internet is very long RTTs. The the speed-of-light round-trip propagation delay is about 500 ms to which must be added queuing delays for the often-backlogged access point (my own personal experience suggested that RTTs of close to 1,000 ms were the norm). These long delays affect real-time traffic such as VoIP and gaming, but as we shall see in *14.11 The Satellite-Link TCP Problem* bulk TCP transfers also perform poorly with very long RTTs. To provide partial compensation for the TCP issue, many satellite ISPs provide some sort of "acceleration" for bulk downloads: a web page, for example, would be downloaded rapidly by the access point and streamed to the satellite and back down to the user via a proprietary mechanism. Acceleration, however, cannot help interactive connections such as VPNs.

Another common feature of satellite Internet is a low daily utilization cap, typically in the hundreds of megabytes. Utilization caps are directly tied to the cost of maintaining satellites, but also to the fact that one satellite covers a great deal of ground, and so its available capacity is shared by a large number of users.

The delay issues associated with satellite Internet would go away if satellites were in so-called low-earth orbits, a few hundred km above the earth. RTTs would then be comparable with terrestrial Internet. Fixed-direction antennas could no longer be used. A large number of satellites would need to be launched to provide 24-hour coverage even at one location. To data (2016), such a network of low-earth satellites has been proposed, but not yet launched.

3.10 Epilog

Along with a few niche protocols, we have focused primarily here on wireless and on virtual circuits. Wireless, of course, is enormously important: it is the enabler for mobile devices, and has largely replaced traditional Ethernet for home and office workstations.

While it is sometimes tempting (in the IP world at least) to write off ATM as a niche technology, virtual circuits are a serious conceptual alternative to datagram forwarding. As we shall see in *20 Quality of Service*, IP has problems handling real-time traffic, and virtual circuits offer a solution. The Internet has so far embraced only small steps towards virtual circuits (such as MPLS, *20.12 Multi-Protocol Label Switching (MPLS)*), but they remain a tantalizing strategy.

3.11 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercise 4.5 is distinct, for example, from exercises 4.0 and 5.0. Exercises marked with a ♢ have solutions or hints at 24.3 Solutions for Other LANs.

1.0. Suppose remote host A uses a VPN connection to connect to host B, with IP address 200.0.0.7. A's normal Internet connection is via device `eth0` with IP address 12.1.2.3; A's VPN connection is via device `ppp0` with IP address 10.0.0.44. Whenever A wants to send a packet via `ppp0`, it is encapsulated and forwarded over the connection to B at 200.0.0.7.

(a). Suppose A's IP forwarding table is set up so that all traffic to 200.0.0.7 uses `eth0` and all traffic to anywhere else uses `ppp0`. What happens if an intruder M attempts to open a connection to A at 12.1.2.3? What route will packets from A to M take?

(b). Suppose A's IP forwarding table is (mis)configured so that *all* outbound traffic uses `ppp0`. Describe what will happen when A tries to send a packet.

2.0. Suppose remote host A wishes to use a TCP-based VPN connection to connect to host B, with IP address 200.0.0.7. However, the VPN software is not available for host A. Host A is, however, able to run that software on a virtual machine V *hosted by* A; A and V have respective IP addresses 10.0.0.1 and 10.0.0.2 on the virtual network connecting them. V reaches the outside world through network address translation (*7.7 Network Address Translation*), with A acting as V's NAT router. When V runs the VPN software, it forwards packets addressed to B the usual way, through A using NAT. Traffic to any other destination it encapsulates over the VPN.

Can A configure its IP forwarding table so that it can make use of the VPN? If not, why not? If so, how? (If you prefer, you may assume V is a physical host connecting to a second interface on A; A still acts as V's NAT router.)

3.0. Token Bus was a proprietary Ethernet-based network. It worked like Token Ring in that a small token packet was sent from one station to the next in agreed-upon order, and a station could transmit only when it had just received the token.

(a). If the data rate is 10 Mbps and the token is 64 bytes long (the 10-Mbps Ethernet minimum packet size), what is the average wait to receive the token on an idle network with 40 stations? (The average number of stations the token must pass through is 40/2 = 20.) Ignore the propagation delay and the gap Ethernet requires between packets.

(b)◇. Sketch a protocol by which stations can sort themselves out to decide the order of token transmission; that is, an order of the stations S_0 ... S_{n-1} where station S_i sends the token to station $S_{(i+1) \bmod n}$.

4.0. A seemingly important part of the IEEE 801.11 Wi-Fi standard is that stations do not transmit when another station is transmitting; this is meant to reduce collisions. And yet the standard states "transmission of the ACK frame shall commence after a SIFS period, without regard to the busy/idle state of the medium"; that is, the ACK sender does *not* listen first for an idle network.

Give a scenario in which the transmission of an ACK while the medium is *not* idle does *not* result in a collision! That is, station A has just finished transmitting a packet to station C, but before C can begin sending its ACK, another station B starts transmitting. Hint: this is another example of the hidden-node problem, *3.7.1.4 Hidden-Node Problem*, with station C again the "middle" station. Recall also that simultaneous transmission results in a collision only if some node fails to be able to read either signal as a result. (Also note that, if C does *not* send its ACK, despite B, the packet just sent from A has to all intents and purposes been lost.)

4.5.◇ Give an example of a three-sender hidden-node collision (*3.7.1.4 Hidden-Node Problem*); that is, three nodes A, B and C, no two of which can see one another, where all can reach a fourth node D. Can you do this for more than three sending nodes?

5.0. Suppose the average contention interval in a Wi-Fi network (802.11g) is 64 SlotTimes. The average packet size is 1 KB, and the data rate is 54 Mbps. At that data rate, it takes about $(8\times1024)/54$ = 151 µsec to transmit a packet.

(a). How long is the average contention interval, in µsec?

(b)◇. What fraction of the total potential bandwidth is lost to contention? (See *2.1.11 Analysis of Classic Ethernet* for a similar example).

6.0. WiMAX and LTE subscriber stations are not expected to hear one another at all. For Wi-Fi non-access-point stations in an infrastructure (access-point) setting, on the other hand, listening to other non-access-point transmissions is encouraged.

(a). List some ways in which Wi-Fi non-access-point stations in an infrastructure (access-point) network do sometimes respond to packets sent by other non-access-point stations. The responses need not be in the form of transmissions.

(b). Explain why Wi-Fi stations cannot be *required* to respond as in part (a).

7.0. Suppose WiMAX subscriber stations can be moving, at speeds of up to 33 meters/sec (the maximum allowed under 802.16e).

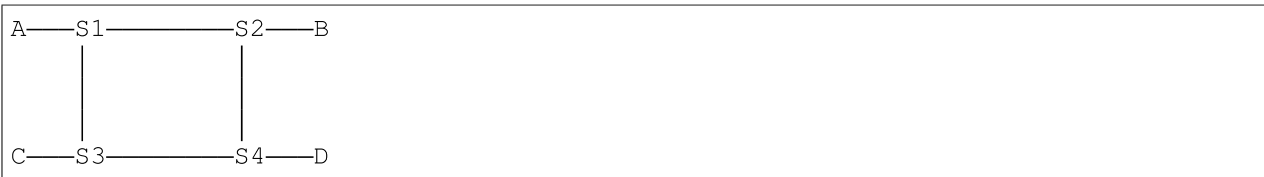
(a). How much earlier (or later) can one subscriber packet arrive? Assume that the ranging process updates the station's propagation delay once a minute. The speed of light is about 300 meters/µsec.

(b). With 5000 senders per second, how much time out of each second must be spent on "guard intervals" accommodating the early/late arrivals above? You will need to double the time from part (a), as the base station cannot tell whether the signal from a moving subscriber will arrive earlier or later.

8.0. *[SM90]* contained a proposal for sending IP packets over ATM as N cells as in AAL-5, followed by one cell containing the XOR of all the previous cells. This way, the receiver can recover from the loss of any one cell. Suppose N=20 here; with the SM90 mechanism, each packet would require 21 cells to transmit; that is, we always send 5% more. Suppose the *cell* loss-rate is p (presumably very small). If we send 20 cells without the SM90 mechanism, we have a probability of about 20p that any one cell will be lost, and we will have to retransmit the entire 20 again. This gives an average retransmission amount of about 20p extra packets. For what value of p do the with-SM90 and the without-SM90 approaches involve about the same total number of cell transmissions?

9.0. In the example in *3.4 Virtual Circuits*, give the VCI table for switch S5.

10.0. Suppose we have the following network:



The virtual-circuit switching tables are below. Ports are identified by the node at the other end. Identify all the connections. Give the path for each connection and the VCI on each link of the path.

Switch **S1**:

VCI_{in}	$port_{in}$	VCI_{out}	$port_{out}$
1	A	2	S3
2	A	2	S2
3	A	3	S2

Switch **S2**:

VCI_{in}	$port_{in}$	VCI_{out}	$port_{out}$
2	S4	1	B
2	S1	3	S4
3	S1	4	S4

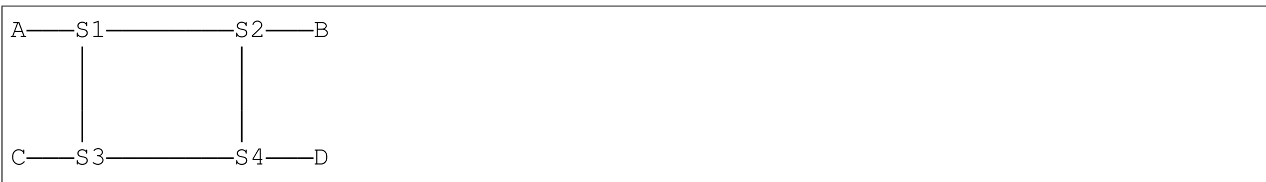
Switch **S3**:

VCI_{in}	$port_{in}$	VCI_{out}	$port_{out}$
2	S1	2	S4
3	S4	2	C

Switch **S4**:

VCI_{in}	$port_{in}$	VCI_{out}	$port_{out}$
2	S3	2	S2
3	S2	3	S3
4	S2	1	D

10.5◇ We have the same network as the previous exercise:



The virtual-circuit switching tables are below. Ports are identified by the node at the other end. Identify all the connections. Give the path for each connection and the VCI on each link of the path.

Switch **S1**:

VCI_{in}	$port_{in}$	VCI_{out}	$port_{out}$
1	A	2	S2
3	S3	2	A

Switch **S2**:

VCI_{in}	$port_{in}$	VCI_{out}	$port_{out}$
2	S1	3	S4
1	B	2	S4

Switch **S3**:

VCI_{in}	$port_{in}$	VCI_{out}	$port_{out}$
2	S1	2	S4
1	S4	3	S1

Switch **S4**:

VCI_{in}	$port_{in}$	VCI_{out}	$port_{out}$
3	S2	2	D
2	S2	1	S3

11.0. Suppose we have the following network:

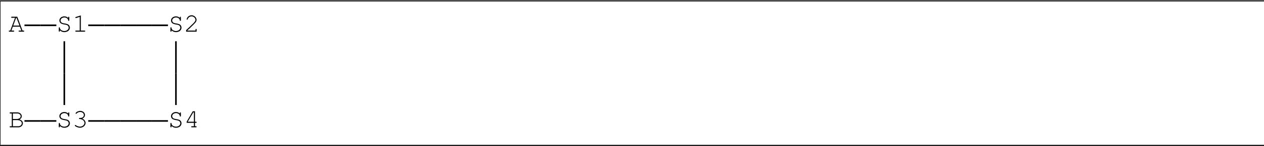
Give virtual-circuit switching tables for the following connections. Route via a shortest path.

(a). A–D
(b). C–B, via S4
(c). B–D
(d). A–D, via whichever of S2 or S3 was *not* used in part (a)

12.0. Below is a set of switches S1 through S4. Define VCI-table entries so the virtual circuit from A to B follows the path

$$A \longrightarrow S1 \longrightarrow S2 \longrightarrow S4 \longrightarrow S3 \longrightarrow S1 \longrightarrow S2 \longrightarrow S4 \longrightarrow S3 \longrightarrow B$$

That is, each switch is visited *twice*.



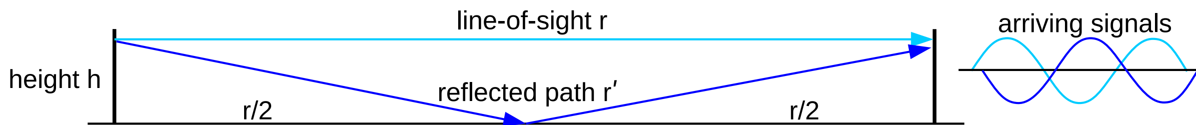
13.0. In this exercise we outline the **two-ray ground** model of wireless transmission in which the signal power is inversely proportional to the fourth power of the distance, rather than following the usual inverse-square law. Some familiarity with trigonometric (or complex-exponential) manipulations is necessary.

Suppose the signal near the transmitter is A $\sin(2\pi ft)$, where A is the amplitude, f the frequency and t the time. Signal power is proportional to A^2. At distance $r \geqslant 1$, the amplitude is reduced by a factor of 1/r (so the power is reduced by $1/r^2$) and the signal is delayed by a time r/c, where c is the speed of light, giving

$$(A/r)\sin(2\pi f(t - r/c)) = (A/r)\sin(2\pi ft - 2\pi\lambda r)$$

where λ = f/c is the wavelength.

The received signal is the superposition of the line-of-sight signal path and its reflection from the ground, as in the following diagram:



Sender and receiver are shown at equal heights above the ground, for simplicity. We assume 100% ground reflectivity (this is reasonable for very shallow angles). The phase of the ground signal is reversed 180° by the reflection, and then is delayed slightly more by the slightly longer path.

(a). Find a formula for the length of the reflected-signal path r', in terms of r and h. Eliminate the square root, using the approximation $(1+x)^{1/2} \simeq 1 + x/2$ for small x. (You will need to factor r out of the square-root expression for r' first.)

(b). Simplify the *difference* (because of the 180° reflection phase-reversal) of the line-of-sight and reflected-signal paths. Use the approximation $\sin(X) - \sin(Y) = 2 \sin((X-Y)/2) \cos((X+Y)/2) \simeq (X-Y) \cos((X+Y)/2) \simeq (X-Y) \cos(X)$, for $X \simeq Y$ (or else use complex exponentials, noting sin(X) is the real part of i e^{iX}). You may assume $r'-r$ is smaller than the wavelength λ. Start with $(A/r)\sin(2\pi ft - 2\pi\lambda r) - (A/r')\sin(2\pi ft - 2\pi\lambda r')$; it helps to isolate the $r \rightarrow r'$ change to one subexpression at a time by writing this as follows (adding and subtracting the identical middle terms):

$$((A/r)\sin(2\pi ft - 2\pi\lambda r) - (A/r')\sin(2\pi ft - 2\pi\lambda r)) + ((A/r')\sin(2\pi ft - 2\pi\lambda r) - (A/r')\sin(2\pi ft - 2\pi\lambda r'))$$

$$= (A/r - A/r')\sin(2\pi ft - 2\pi\lambda r) \quad + \quad (A/r')(\sin(2\pi ft - 2\pi\lambda r) - \sin(2\pi ft - 2\pi\lambda r'))$$

(c). Show that the approximate amplitude of this difference is proportional to $1/r^2$, making the relative power proportional to $1/r^4$.

14.0. In the four-way handshake of *3.7.5 Wi-Fi Security*, suppose station B (for Bad) records the successful handshake of station A and the access point. A then leaves the network, and B attempts a **replay attack**: B

uses A's packets in the handshake. At exactly what point does the handshake break down?

At the lowest (logical) level, network links look like serial lines. In this chapter we address how packet structures are built on top of serial lines, via encoding and framing. Encoding determines how bits and bytes are represented on a serial line; framing allows the receiver to identify the beginnings and endings of packets.

We then conclude with the high-speed serial lines offered by the telecommunications industry, T-carrier and SONET, upon which almost all long-haul **point-to-point** links that tie the Internet together are based.

4.1 Encoding and Framing

A typical serial line is ultimately a stream of *bits*, not bytes. How do we identify byte boundaries? This is made slightly more complicated by the fact that, beneath the logical level of the serial line, we generally have to avoid transmitting long runs of identical bits, because the receiver may simply lose count; this is the **clock synchronization** problem (sometimes called the clock recovery problem). This means that, one way or another, we cannot always just send the desired bits sequentially; for example, extra bits are often inserted to break up long runs. Exactly how we do this is the **encoding** mechanism.

Once we have settled the transmission of bits, the next step is to determine how the receiver identifies the start of each new packet. Ethernet packets are separated by physical gaps, but for most other link mechanisms packets are sent end-to-end, with no breaks. How we tell when one packet stops and the next begins is the **framing** problem.

To summarize:

- encoding: correctly recognizing all the bits in a stream
- framing: correctly recognizing packet boundaries

These are related, though not the same.

For long (multi-kilometer) electrical serial lines, in addition to the clock-related serial-line requirements we also want the average voltage to be zero; that is, we want no DC component. We will mostly concern ourselves here, however, only with lines short enough for this not to be a major concern.

4.1.1 NRZ

NRZ (Non-Return to Zero) is perhaps the simplest encoding; it corresponds to direct bit-by-bit transmission of the 0's and 1's in the data. We have two signal levels, **lo** and **hi**, we set the signal to one or the other of these depending on whether the data bit is 0 or 1, as in the diagram below. Note that in the diagram the signal bits have been aligned with the *start* of the pulse representing that signal value.

0 0 1 0 1 1 0 1 1 1 1 0 0 0 0 0 1

NRZ Encoding: 1 = hi, 0 = lo

NRZ replaces an earlier RZ (Return to Zero) encoding, in which hi and lo corresponded to +1 and -1, and between each pair of pulses corresponding to consecutive bits there was a brief return to the 0 level.

One drawback to NRZ is that we cannot distinguish between 0-bits and a signal that is simply idle. However, the more serious problem is the lack of **synchronization**: during long runs of 0's or long runs of 1's, the receiver can "lose count", *eg* if the receiver's clock is running a little fast or slow. The receiver's clock can and does resynchronize whenever there is a **transition** from one level to the other. However, suppose bits are sent at one per µs, the sender sends five 1-bits in a row, and the receiver's clock is running 10% fast. The signal sent is a 5-µs hi pulse, but when the pulse ends the receiver's clock reads 5.5 µs due to the clock speedup. Should this represent five 1-bits or six 1-bits?

4.1.2 NRZI

An alternative that helps here (though not obviously at first) is **NRZI**, or NRZ Inverted. In this encoding, we represent a 0-bit as no change, and a 1-bit as a **transition** from lo to hi or hi to lo:

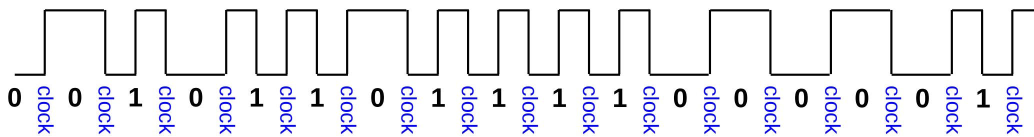
0 0 1 0 1 1 0 1 1 1 1 0 0 0 0 0 1

NRZI Encoding: 1 = transition, 0 = no transition

Now there is a signal transition aligned above every 1-bit; a 0-bit is represented by the lack of a transition. This solves the synchronization problem for runs of 1-bits, but does nothing to address runs of 0-bits. However, NRZI can be combined with techniques to minimize runs of 0-bits, such as 4B/5B (below).

4.1.3 Manchester

Manchester encoding sends the data stream using NRZI, with the addition of a **clock transition** between each pair of consecutive data bits. This means that the signaling rate is now double the data rate, *eg* 20 MHz for 10Mbps Ethernet (which does use Manchester encoding). The signaling is as if we doubled the bandwidth and inserted a 1-bit between each pair of consecutive data bits, removing this extra bit at the receiver:



Manchester Encoding: NRZI alternating with clock transitions

All these transitions mean that the longest the clock has to "count" is 1 bit-time; clock synchronization is essentially solved, at the expense of the doubled signaling rate.

4.1.4 4B/5B

In 4B/5B encoding, for each 4-bit "nybble" of data we actually transmit a designated 5-bit **symbol**, or code, selected to have "enough" 1-bits. A symbol in this sense is a digital or analog transmission unit that decodes to a set of data bits; the data bits are not transmitted individually. (The transmission of symbols rather than individual bits is nearly universal for high-performance links, including all forms of Ethernet faster than 10Mbps and all Wi-Fi links.)

Specifically, every 5-bit symbol used by 4B/5B has at most one leading 0-bit and at most two trailing 0-bits. The 5-bit symbols corresponding to the data are then sent with NRZI, where runs of 1's are safe. Note that the worst-case run of 0-bits has length three. Note also that the signaling rate here is 1.25 times the data rate. 4B/5B is used in 100-Mbps Ethernet, *2.2 100 Mbps (Fast) Ethernet*. The mapping between 4-bit data values and 5-bit symbols is fixed by the 4B/5B standard:

data	symbol	data	symbol
0000	11110	1011	10111
0001	01001	1100	11010
0010	10100	1101	11011
0011	10101	1110	11100
0100	01010	1111	11101
0101	01011	IDLE	11111
0110	01110	HALT	00100
0111	01111	START	10001
1000	10010	END	01101
1001	10011	RESET	00111
1010	10110	DEAD	00000

There are more than sixteen possible symbols; this allows for some symbols to be used for signaling rather than data. IDLE, HALT, START, END and RESET are shown above, though there are others. These can be used to include control and status information without fear of confusion with the data. Some combinations of control symbols do lead to up to four 0-bits in sequence; HALT and RESET have two leading 0-bits.

10-Mbps and 100-Mbps Ethernet pads short packets up to the minimum packet size with 0-bytes, meaning that the next protocol layer has to be able to distinguish between padding and actual 0-byte data. Although 100-Mbps Ethernet uses 4B/5B encoding, it does not make use of special non-data symbols for packet padding. Gigabit Ethernet uses PAM-5 encoding (*2.3 Gigabit Ethernet*), and *does* use special non-data symbols (inserted by the hardware) to pad packets; there is thus no ambiguity at the receiving end as to where the data bytes ended.

The choice of 5-bit symbols for 4B/5B is in principle arbitrary; note however that for data from 0100 to 1101 we simply insert a 1 in the fourth position, and in the last two we insert a 0 in the fourth position. The first four symbols (those with the most zeroes) follow no obvious pattern, though.

4.1.5 Framing

How does a receiver tell when one packet stops and the next one begins, to keep them from running together? We have already seen the following techniques for addressing this *framing* problem: determining where packets end:

- Interpacket gaps (as in Ethernet)
- 4B/5B and special bit patterns

Putting a length field in the header would also work, in principle, but seems not to be widely used. One problem with this technique is that restoring order after desynchronization can be difficult.

There is considerable overlap of framing with encoding; for example, the existence of non-data bit patterns in 4B/5B is due to an attempt to solve the encoding problem; these special patterns can also be used as unambiguous frame delimiters.

4.1.5.1 HDLC

HDLC (High-level Data Link Control) is a general link-level packet format used for a number of applications, including Point-to-Point Protocol (PPP) (which in turn is used for PPPoE – PPP over Ethernet – which is how a great many Internet subscribers connect to their ISP), and Frame Relay, still used as the low-level protocol for delivering IP packets to many sites via telecommunications lines. HDLC supports the following two methods for frame separation:

- HDLC over asynchronous links: byte stuffing
- HDLC over synchronous links: bit stuffing

The basic encapsulation format for HDLC packets is to begin and end each frame with the byte 0x7E, or, in binary, 0111 1110. The problem is that this byte may occur in the data as well; we must make sure we don't misinterpret such a data byte as the end of the frame.

Asynchronous serial lines are those with some sort of start/stop indication, typically between bytes; such lines tend to be slower. Over this kind of line, HDLC uses the byte 0x7D as an escape character. Any data bytes of 0x7D and 0x7E are escaped by preceding them with an additional 0x7D. (Actually, they are transmitted as 0x7D followed by (original_byte xor 0x20).) This strategy is fundamentally the same as that used by C-programming-language character strings: the string delimiter is **"** and the escape character is ****. Any occurrences of **"** or **** within the string are escaped by preceding them with ****.

Over synchronous serial lines (typically faster than asynchronous), HDLC generally uses **bit stuffing**. The underlying bit encoding involves, say, the reverse of NRZI, in which transitions denote 0-bits and lack of transitions denote 1-bits. This means that long runs of 1's are now the problem and runs of 0's are safe.

Whenever five consecutive 1-bits appear in the data, *eg* 011111, a 0-bit is then inserted, or "stuffed", by the transmitting hardware (regardless of whether or not the next data bit is also a 1). The HDLC frame byte of

0x7E = 0111 1110 thus can never appear as encoded data, because it contains six 1-bits in a row. If we had 0x7E in the data, it would be transmitted as 0111 11**0**10.

The HDLC receiver knows that

- six 1-bits in a row marks the end of the packet
- when five 1-bits in a row are seen, followed by a 0-bit, the 0-bit is removed

Example:

Data: 011110 0111110 01111110
Sent as: 011110 011111**0**0 011111**0**10 (stuffed bits in **bold**)

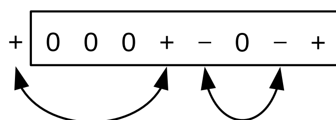
Note that bit stuffing is used by HDLC to solve two unrelated problems: the synchronization problem where long runs of the same bit cause the receiver to lose count, and the framing problem, where the transmitted bit pattern 0111 1110 now represents a flag that can never be mistaken for a data byte.

4.1.5.2 B8ZS

While insertion of an occasional extra bit or byte is no problem for data delivery, it is anathema to voice engineers; extra bits upset the precise 64 Kbps DS-0 rate. As a result, long telecom lines prefer encodings that, like 4B/5B, do not introduce timing fluctuations. Very long (electrical) lines also tend to require encodings that guarantee a long-term *average* voltage level of 0 (versus 0.5 if half the bits are 1 v and half are 0 v in NRZ); that is, the signal must have no DC component.

The **AMI** (Alternate Mark Inversion) technique eliminates the DC component by using three voltage levels, nominally +1, 0 and -1; this ternary encoding is also known as **bipolar**. Zero bits are encoded by the 0 voltage, while 1-bits take on alternating values of +1 and -1 volts. Thus, the bits 011101 might be encoded as 0,+1,-1,+1,0,-1, or, more compactly, 0+–+0–. Over a long run, the +1's and the –1's cancel out.

Plain AMI still has synchronization problems with long runs of 0-bits. The solution used on North American T1 lines (1.544 Mbps) is known as **B8ZS**, for bipolar with 8-zero substitution. The sender replaces any run of 8 zero bits with a special bit-pattern, either 000+–0–+ or 000–+0+–. To decide which, the sender checks to see if the previous 1-bit sent was +1 or –1; if the former, the first pattern is substituted, if the latter then the second pattern is substituted. Either way, this leads to two instances of violation of the rule that consecutive 1-bits have opposite sign. For example, if the previous bit were +, the receiver sees



B8ZS Encoding. The bits in the box were originally all zeros.
Arrows link 1-bit alternating-sign violations

This double-violation is the clue to the receiver that the special pattern is to be removed and replaced with the original eight 0-bits.

4.2 Time-Division Multiplexing

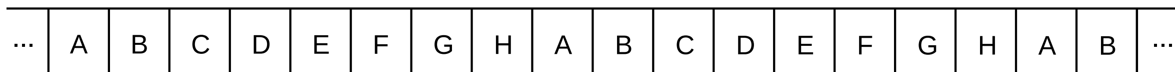
Classical **circuit switching** means a separate wire for each connection. This is still in common use for residential telephone connections: each subscriber has a dedicated wire to the Central Office. But a separate physical line for each connection is not a solution that scales well.

Once upon a time it was not uncommon to link computers with serial lines, rather than packet networks. This was most often done for file transfers, but telnet logins were also done this way. The problem with this approach is that the line had to be dedicated to one application (or one user) at a time.

Packet switching naturally implements multiplexing (sharing) on links; the demultiplexer is the destination address. Port numbers allow demultiplexing of multiple streams to same destination host.

There are other ways for multiple channels to share a single wire. One approach is **frequency-division multiplexing**, or putting each channel on a different carrier frequency. Analog cable television did this. Some fiber-optic protocols also do this, calling it **wavelength**-division multiplexing.

But perhaps the most pervasive alternative to packets is the voice telephone system's **time division multiplexing**, or TDM, sometimes prefixed with the adjective **synchronous**. The idea is that we decide on a number of channels, N, and the length of a timeslice, T, and allow each sender to send over the channel for time T, with the senders taking turns in round-robin style. Each sender gets to send for time T at regular intervals of NT, thus receiving 1/N of the total bandwidth. The timeslices consume no bandwidth on headers or addresses, although sometimes there is a small amount of space dedicated to maintaining synchronization between the two endpoints. Here is a diagram of sending with N=8:



Time-Division Multiplexing

Note, however, that if a sender has nothing to send, its timeslice cannot be used by another sender. Because so much data traffic is **bursty**, involving considerable idle periods, TDM has traditionally been rejected for data networks.

4.2.1 T-Carrier Lines

TDM, however, works extremely well for voice networks. It continues to work when the timeslice T is small, when packet-based approaches fail because the header overhead becomes unacceptable. Consider for a moment the telecom Digital Signal hierarchy. A single digitized voice line in North America is one 8-bit sample every 1/8,000 second, or 64 Kbps; this is known as a **DS0** channel. A **T1** line – the lowest level of the **T-carrier** hierarchy and known at the logical level as a **DS1** line – represents 24 DS0 lines multiplexed via TDM, where each channel sends a single byte at a time. Thus, every 1/8,000 of a second a T1 line carries 24 bytes of user data, one byte per channel (plus one *bit* for framing), for a total of 193 bits. This gives a raw line speed of 1.544 Mbps.

Note that the per-channel frame size here is a single byte. There is no efficient way to send single-byte *packets*. The advantage to the single-byte approach is that it greatly reduces the latency across the line. The biggest source of delay in packet-based digital voice lines is the packet **fill time** at the sender's end: the

sender generates voice data at a rate of 8 bytes/ms, and a packet cannot be sent until it is full. For a 1 KB packet, that's about a quarter second. For standard Voice-over-IP or **VoIP** channels, RTP is used with 160 bytes of data sent every 20 ms; for ATM, a 48-byte packet is sent every 6 ms. But the fill-time delay for a call sent over a T1 line is 0.125 ms, which is negligible (to be fair, 6 ms and even 20 ms turn out to be pretty negligible in terms of call quality). The T1 one-byte-at-a-time strategy also means that T1 multiplexers need to do essentially no buffering, which might have been important back in 1962 when T-carrier was introduced.

The next most common T-carrier / Digital Signal line is perhaps T3/DS3; this represents the TDM multiplexing of 28 DS1 signals. The problem is that some individual DS1s may run a little slow, so an elaborate **pulse stuffing** protocol has been developed. This allows extra bits to be inserted at specific points, if necessary, in such a way that the original component T1s can be exactly recovered even if there are clock irregularities. The pulse-stuffing solution did not scale well, and so T-carrier levels past T3 were very rarely used.

While T-carrier was originally intended as a way of bundling together multiple DS0 channels on a single high-speed line, it also allows providers to offer leased digital point-to-point links with data rates in almost any multiple of the DS0 rate.

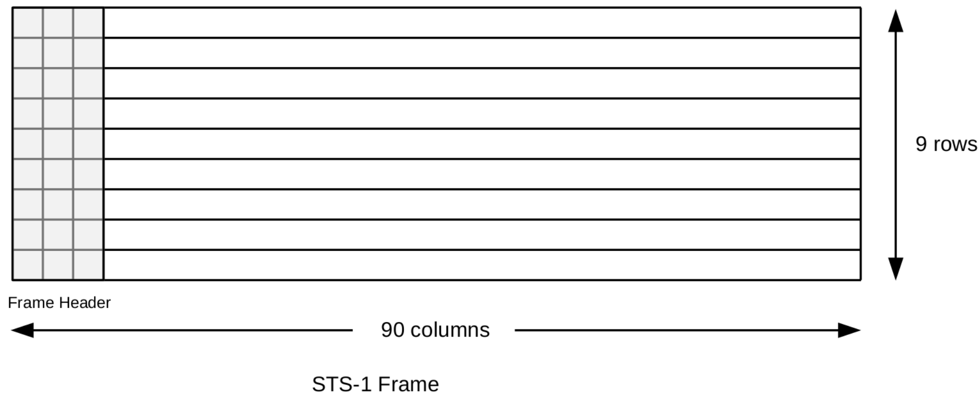
4.2.2 SONET

SONET stands for Synchronous Optical NETwork; it is the telecommunications industry's standard mechanism for very-high-speed TDM over optical fiber. While there is now flexibility regarding the the "optical" part, the "synchronous" part is taken quite seriously indeed, and SONET senders and receivers all use very precisely synchronized clocks (often atomic). The actual bit encoding is NRZI.

Due to the frame structure, below, the longest possible run of 0-bits is ~250 bits (~30 bytes), but is usually much less. Accurate reception of 250 0-bits requires a clock accurate to within (at a minimum) one part in 500, which is potentially within reach. However, SONET also has a "bit-scrambling" feature, involving XOR with a fixed bit pattern, to ensure in most cases that there is a 1-bit every byte or less.

The primary reason for SONET's accurate clocking, however, is not the clock-synchronization problem as we have been using the term, but rather the problem of demultiplexing and remultiplexing multiple component bitstreams in a setting in which some of the streams may run slow. One of the primary design goals for SONET was to allow such multiplexing without the need for "pulse stuffing", as is used in the Digital Signal hierarchy. SONET tributary streams are in effect not *allowed* to run slow (although SONET does provide for occasional very small byte slips, below). Furthermore, as multiple SONET streams are demultiplexed at a switching center and then remultiplexed into new SONET streams, synchronization means that none of the streams falls behind or gets ahead.

The basic SONET format is known as STS-1. Data is organized as a 9x90 byte grid. The first 3 bytes of each row (that is, the first three columns) form the frame header. Frames are not addressed; SONET is a point-to-point protocol and a node sends a continuous sequence of frames to each of its neighbors. When the frames reach their destination, in principle they need to be fully demultiplexed for the data to be forwarded on. In practice, there are some shortcuts to full demultiplexing.



STS-1 Frame

The actual bytes sent are scrambled: the data is XORed with a standard, fixed pseudorandom pattern before transmission. This introduces many 1-bits, on which clock resynchronization can occur, with a high degree of probability.

There are two other special columns in a frame, each guaranteed to contain at least one 1-bit, so the maximum run of data bytes is limited to ~30; this is thus the longest run of possible 0's.

The first two bytes of each frame are 0xF628. SONET's frame-synchronization check is based on verifying these byte values at the start of each frame. If the receiver is ever desynchronized, it begins a frame re-synchronization procedure: the receiver searches for those 0xF628 bytes at regular 810-byte (6480-bit) spacing. After a few frames with 0xF628 in the right place, the receiver is "very sure" it is looking at the synchronization bytes and not at a data-byte position. Note that there is no evident byte boundary to a SONET frame, so the receiver must check for 0xF628 beginning at every *bit* position.

SONET frames are transmitted at a rate of 8,000 frames/second. This is the canonical byte sampling rate for standard voice-grade ("DS0", or 64 Kbps) lines. Indeed, the classic application of SONET is to transmit multiple DS0 voice calls using TDM: within a frame, each data byte position is given over to one voice channel. The same byte position in consecutive frames constitutes one byte every 1/8000 seconds. The basic STS-1 data rate of 51.84 Mbps is exactly 810 bytes/frame $\times$ 8 bits/byte $\times$ 8000 frames/sec.

To a customer who has leased a SONET-based channel to transmit data, a SONET link looks like a very fast bitstream. There are several standard ways of encoding data packets over SONET. One is to encapsulate the data as ATM cells, and then embed the cells contiguously in the bitstream. Another is to send IP packets encoded in the bitstream using HDLC-like bit stuffing, which means that the SONET bytes and the IP bytes may no longer correspond. The advantage of HDLC encoding is that it makes SONET re-synchronization vanishingly infrequent. Most IP backbone traffic today travels over SONET links.

Within the 9$\times$90-byte STS-1 frame, the payload envelope is the 9$\times$87 region nominally following the three header columns; this payload region has its own three reserved columns meaning that there are 84 columns (9$\times$84 bytes) available for data. This 9$\times$87-byte payload envelope can "float" within the physical 9$\times$90-byte frame; that is, if the input frames are running slow then the output physical frames can be transmitted at the correct rate by letting the payload frames slip "backwards", one byte at a time. Similarly, if the input frames are arriving slightly too fast, they can slip "forwards" by up to one byte at a time; the extra byte is stored in a reserved location in the three header columns of the 9$\times$90 physical frame.

Faster SONET streams are made by multiplexing slower ones. The next step up is STS-3, an STS-3 frame is three STS-1 frames, for 9$\times$270 bytes. STS-3 (or, more properly, the physical layer for STS-3) is also called OC-3, for Optical Carrier. Beyond STS-3, faster lines are multiplexed combinations of four of the

next-slowest lines. Here are some of the higher levels:

STS	STM	line rate
STS-1	STM-0	51.84 Mbps
STS-3	STM-1	155.52 Mbps
STS-12	STM-4	622.08 Mbps (=12*51.84, exactly)
STS-48	STM-16	2.48832 Gbps
STS-192	STM-64	9.953 Gbps
STS-768	STM-256	39.8 Gbps

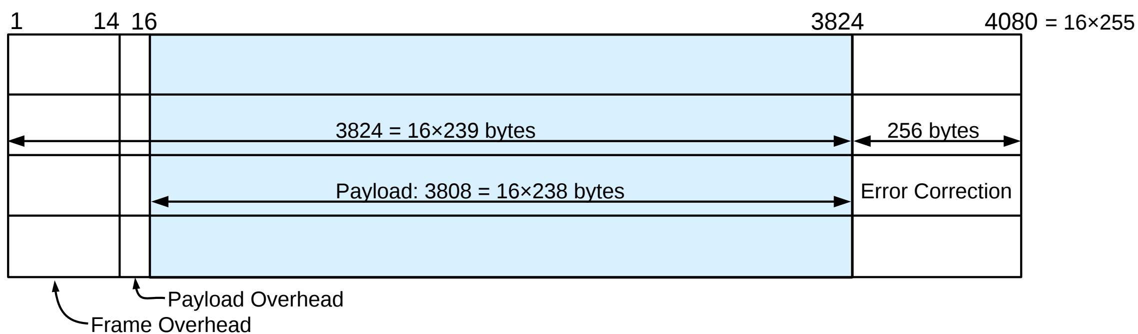
Usable capacity is typically 84/90 of the above line rates, as six of the 90 columns of an STS-1 frame are for overhead.

SONET provides a wide variety of leasing options at various bandwidths. High-volume customers can lease an entire STS-1 or larger unit. Alternatively, the 84 data columns of an STS-1 frame can be divided into seven **virtual tributary** groups, each of twelve columns; these groups can be leased individually or in multiples, or be further divided into as few as three columns (which works out to be just over the T1 data rate).

4.2.3 Optical Transport Network

The Optical Transport Network, or OTN, is an ITU specification for data transmission over optical fiber; the primary standard is G.709. A preliminary version of G.709 was published in 1988, but version 1.0 was released in 2001. OTN abandons SONET's voice-oriented frame rate of 8000 frames/sec; while OTN is still widely used for voice, transmission no longer quite so perfectly fits the time-division-multiplexing model.

The standard OTN frame is as diagrammed below; each frame is arranged in four rows. It can be helpful to view the 4,080 columns as divided into 255 16-byte-wide "supercolumns": one for the frame and payload overhead, 238 for the payload itself, and 16 for error correction.



OTN Frame Format: 4080 columns × 4 rows

The portion available for carrying customer data is highlighted in blue; unlike SONET, the payload portion of a frame is not allowed to "float". The 256 columns at the end are for error-correcting codes (*5.4.2 Error-Correcting Codes*); the addition of such error correction (below) is a major advantage of OTN over SONET.

OTN comes in three primary rates, OTN1, OTN2 and OTN3; there are also a few variants. The OTN1 rate is chosen so that a SONET STS-48/STM-16 stream – 2.48832 Gbps – exactly fits in the blue payload portion of the frame. This means that the OTN1 line rate must be (255/238) $\times$ 2.48832 $\simeq$ 2.6661 Gbps. The frame rate, in turn, is therefore about 20.42 frames/ms.

Four OTN1 streams can be multiplexed into a single OTN2 stream; four OTN2 streams can be multiplexed into a single OTN3 stream. The frame layout does not change, however; it is the frame rate that increases. The data rate does not increase by an exact multiple of 4; the OTN2 rate is chosen so that four OTN1 payloads *and an additional 16 columns per frame* can be carried in the payload portion of the OTN2 frames. The additional 16 columns serve to specify details about the interleaving of the four OTN1 frames; this leaves 3792 = 16×237 columns for the OTN1-stream data. The end result is that the OTN2 stream must carry data at a line rate of 4×238/237 times the OTN1 rate, or 4×255/237 times the STS-48/STM-16 rate. This works out to be about 10.709 Gbps.

In order to handle multiplexing without the need for T-carrier-style pulse stuffing, OTN streams must have a long-term bit-rate accuracy of ±20 parts per million, which works out to be one byte every 3-4 frames. The frame-overhead region takes care of the slack.

The last 256 columns of each frame are devoted to error correction; SONET has nothing comparable. By default, these columns are used for so-called Reed-Solomon codes; specifically, in a formulation where 16 bytes of codes are used for each 239 bytes of payload (this is exactly consistent with the relative sizes of the payload and error-correction columns). Such codes can correct up to 8 bytes of error. They can be used to increase the length of cable runs before regeneration is needed. Alternatively, their use can reduce the bit error rate as much as a hundredfold, which will be important in *14.9 The High-Bandwidth TCP Problem*.

OTN also includes, at the physical layer, extensive support for **dense wavelength-division multiplexing** (DWDM, a form of frequency-division multiplexing), meaning that multiple independent OTN streams can be carried on relatively close wavelengths. This greatly increases the overall capacity of a given run of fiber. DWDM works at a lower network layer than the frame organization outlined above, and in principle SONET could implement DWDM as well. In practice, though, DWDM has been integrated with OTN from the beginning.

4.2.4 Other Optical Fiber

Optical Fiber and Lightning

One of the advantages of of optical fiber (particularly at mountaintop observatories) is its resistance to damage and interference from lightning. Some fiber-optic cables do, however, have metal jackets to add strength or to resist animals; lightning resistance must be researched carefully.

SONET and OTN are primarily, though not exclusively, used by telecommunications carriers. There are also multiple fiber-optic alternatives for smaller-scale operations. These are often part of the Ethernet family although they may have little except their bitrate in common with one another or with Ethernet over copper wire. Another standard that supports optical fiber links is so-called "Fibre Channel", though that too also now supports copper.

Distances supported by fiber-optic Ethernet range from hundreds of meters to tens of kilometers. Generally, the longer-haul links require the use of full duplex to avoid the collision-detection (slot time) requirement that a sender continue to transmit for one full Ethernet-segment RTT, but full duplex is common at high Ethernet speeds even for short links. For 100 Mbps Ethernet, fiber-optic standards include 100BASE-FX, 100BASE-SX, 100BASE-LX and 100BASE-BX. The latter supports distances up to 40 km; the limiting factor is the need for signal regeneration. 1000-Gbit Ethernet optical standards include 1000BASE-SX, 1000BASE-LX, 1000BASE-BX and 1000BASE-EX; the latter again supports up to 40 km. These forms

of Ethernet are often deployed in residential "fiber Internet" connections, although in some cases the last hundred meters or so may still involve copper.

At speeds of 10 Gbps, long-range optical fiber alternatives include 10GBASE-ER and 10GBASE-ZR, supporting 40 and 80 km respectively; 10GBASE-ZR is based on SONET STM-64 standards. It is worth noting that fiber is often used for short links as well at 1 Gbps and 10 Gbps speeds; at 10 Gbps some copper-link standards support distances of only a few meters. As of 2014, several 10-Gbps Ethernet physical-layer standards come from industry consortiums rather than the IEEE.

4.3 Epilog

This completes our discussion of common physical links. Perhaps the main takeaway point is that transmitting bits over any distance is not quite as simple as it may appear; simple NRZ transmission is *not* effective.

4.4 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of future exercises. Exercise 3.5 is distinct, for example, from exercises 3.0 and 4.0. Exercises marked with a ♢ have solutions or hints at *24.4 Solutions for Links*.

1.0. What is encoded by the following NRZI signal? The first two bits are shown.

```
0    1
```

2.0. Argue that sending 4 0-bits via NRZI requires a clock accurate to within 1 part in 8. Assume that the receiver resynchronizes its clock whenever a 1-bit transition is received, but that otherwise it attempts to sample a bit in the middle of the bit's timeslot.

3.0.(a) What bits are encoded by the following Manchester-encoded sequence?

(b). Why is there no ambiguity as to whether the first transition is a clock transition or a data (1-bit) transition?

(c). Give an example of a signal pattern consisting of an NRZI encoding of 0-bits and 1-bits that does not contain two consecutive 0-bits and which is *not* a valid Manchester encoding of data. Such a pattern could thus could be used as a special non-data marker.

3.5.♢ What is the 4B/5B encoding for the 3-byte string "Net"? (Hint: 'N' is 0100 1110.)

4.0. What three ASCII letters (bytes) are encoded by the following 4B/5B pattern? (Be careful about upper-case vs lower-case.)

010110101001110101010111111110

5.0.(a) Suppose a device is forwarding SONET STS-1 frames. How much clock drift, as a percentage, on the incoming line would mean that the output payload envelopes must slip backwards by one byte per three physical frames? | (b). In *4.2.2 SONET* it was claimed that sending 250 0-bits required a clock accurate to within 1 part in 500. Describe how a SONET clock might meet the requirement of part (a) above, and yet fail at this second requirement. (Hint: in part (a) the requirement is a long-term average).

In this chapter we address a few abstract questions about packets, and take a close look at transmission times. We also consider how big packets should be, and how to detect transmission errors. These issues are independent of any particular set of protocols.

5.1 Packet Delay

There are several contributing sources to the delay encountered in transmitting a packet. On a LAN, the most significant is usually what we will call **bandwidth delay**: the time needed for a sender to get the packet onto the wire. This is simply the packet size divided by the bandwidth, after everything has been converted to common units (either all bits or all bytes). For a 1500-byte packet on 100 Mbps Ethernet, the bandwidth delay is 12,000 bits / (100 bits/µsec) = 120 µsec.

There is also **propagation delay**, relating to the propagation of the bits at the speed of light (for the transmission medium in question). This delay is the distance divided by the speed of light; for 1,000 m of Ethernet cable, with a signal propagation speed of about 230 m/µsec, the propagation delay is about 4.3 µsec. That is, if we start transmitting the 1500 byte packet of the previous paragraph at time T=0, then the first bit arrives at a destination 1,000 m away at T = 4.3 µsec, and the last bit is transmitted at 120 µsec, and the last bit arrives at T = 124.3 µsec.

Minimizing Delay

Back in the last century, gamers were sometimes known to take advantage of players with slow (as in dialup) links; an opponent could be eliminated literally before he or she could respond. As an updated take on this, some financial-trading firms have set up microwave-relay links between trading centers, say New York and Chicago, in order to reduce delay. In computerized trading, milliseconds count. A direct line of sight from New York to Chicago – which we round off to 1200 km – takes about 4 ms in air, where signals propagate at essentially the speed of light c = 300 km/ms. But fiber is slower; even an absolutely straight run would take 6 ms at glass fiber's propagation speed of 200 km/ms. In the presence of high-speed trading, this 2 ms savings is of considerable financial significance.

Bandwidth delay, in other words, tends to dominate within a LAN.

But as networks get larger, propagation delay begins to dominate. This also happens as networks get faster: bandwidth delay goes down, but propagation delay remains unchanged.

An important difference between bandwidth delay and propagation delay is that bandwidth delay is proportional to the amount of data sent while propagation delay is not. If we send two packets back-to-back, then the bandwidth delay is doubled but the propagation delay counts only once.

The introduction of switches leads to **store-and-forward delay**, that is, the time spent reading in the entire packet before any of it can be retransmitted. Store-and-forward delay can also be viewed as an additional bandwidth delay for the second link.

Finally, a switch may or may not also introduce **queuing delay**; this will often depend on competing traffic. We will look at this in more detail in *14 Dynamics of TCP Reno*, but for now note that a steady queuing delay (*eg* due to a more-or-less constant average queue utilization) looks to each sender more like propagation delay than bandwidth delay, in that if two packets are sent back-to-back and arrive that way at the queue, then the pair will experience only a single queuing delay.

5.1.1 Delay examples

Case 1: A———B

- Propagation delay is 40 µsec
- Bandwidth is 1 byte/µsec (1 MB/sec, 8 Mbit/sec)
- Packet size is 200 bytes (200 µsec bandwidth delay)

Then the total one-way transmit time for one packet is 240 µsec = 200 µsec + 40 µsec. To send two back-to-back packets, the time rises to 440 µsec: we add one more bandwidth delay, but not another propagation delay.

Case 2: A——————————B

Like the previous example except that the propagation delay is increased to 4 ms

The total transmit time for one packet is now 4200 µsec = 200 µsec + 4000 µsec. For two packets it is 4400 µsec.

Case 3: A———R———B

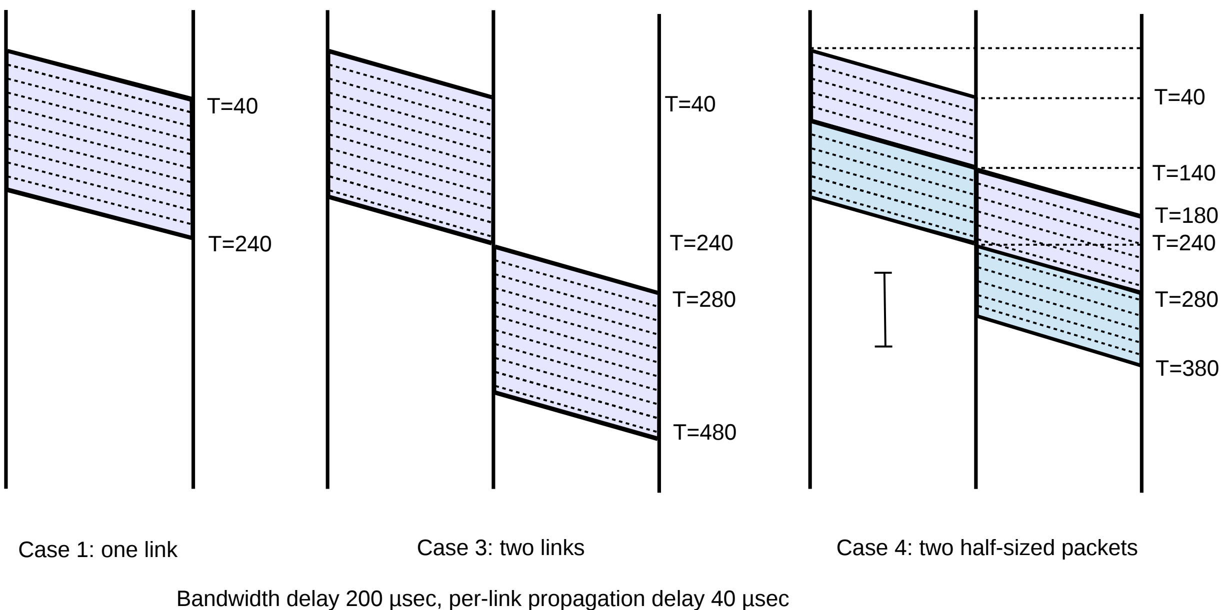
We now have two links, each with propagation delay 40 µsec; bandwidth and packet size as in Case 1.

The total transmit time for one 200-byte packet is now 480 µsec = 240 + 240. There are two propagation delays of 40 µsec each; A introduces a bandwidth delay of 200 µsec and R introduces a store-and-forward delay (or second bandwidth delay) of 200 µsec.

Case 4: A———R———B

The same as 3, but with data sent as two 100-byte packets

The total transmit time is now 380 µsec = 3x100 + 2x40. There are still two propagation delays, but there is only 3/4 as much bandwidth delay because the transmission of the first 100 bytes on the second link overlaps with the transmission of the second 100 bytes on the first link.



These **ladder diagrams** represent the full transmission; a snapshot state of the transmission at any one instant can be obtained by drawing a horizontal line. In the middle, case 3, diagram, for example, at no instant are both links active. Note that sending two smaller packets is faster than one large packet. We expand on this important point below.

Now let us consider the situation when the propagation delay is the most significant component. The cross-continental US roundtrip delay is typically around 50-100 ms (propagation speed 200 km/ms in cable, 5,000-10,000 km cable route, or about 3-6000 miles); we will use 100 ms in the examples here. At a bandwidth of 1.0 Mbps, 100ms is about 12 KB, or eight full-sized Ethernet packets. At this bandwidth, we would have four packets and four returning ACKs strung out along the path. At 1.0 Gbit/s, in 100ms we can send 12,000 KB, or 800 Ethernet packets, before the first ACK returns.

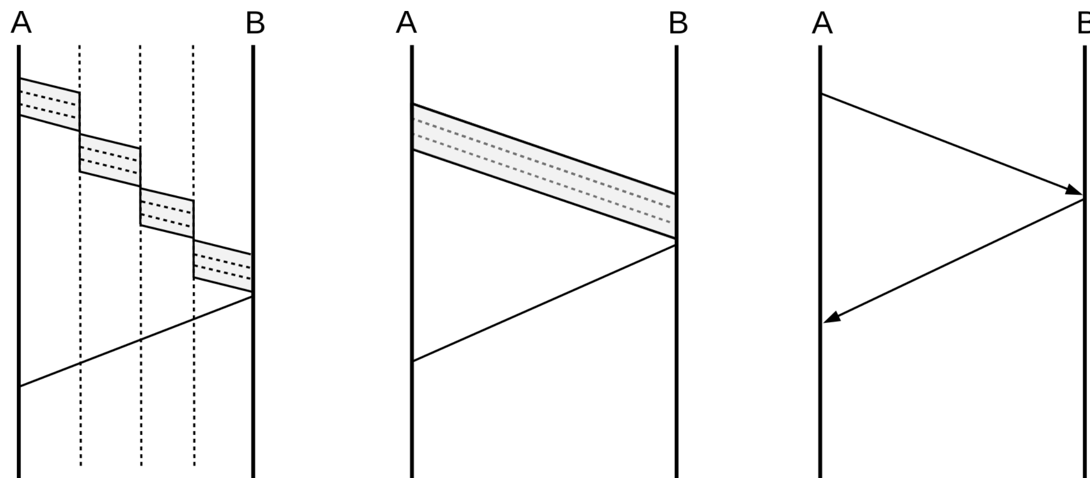
At most non-LAN scales, the delay is typically simplified to the **round-trip time**, or **RTT**: the time between sending a packet and receiving a response.

Different delay scenarios have implications for protocols: if a network is bandwidth-limited then protocols are easier to design. Extra RTTs do not cost much, so we can build in a considerable amount of back-and-forth exchange. However, if a network is delay-limited, the protocol designer must focus on minimizing extra RTTs. As an extreme case, consider wireless transmission to the moon (0.3 sec RTT), or to Jupiter (1 hour RTT).

At my home I formerly had satellite Internet service, which had a roundtrip propagation delay of ~600 ms. This is remarkably high when compared to purely terrestrial links.

When dealing with reasonably high-bandwidth "large-scale" networks (*eg* the Internet), to good approximation most of the non-queuing delay is propagation, and so bandwidth and total delay are effectively independent. Only when propagation delay is small are the two interrelated. Because propagation delay dominates at this scale, we can often make simplifications when diagramming. In the illustration below, A sends a data packet to B and receives a small ACK in return. In (a), we show the data packet traversing several switches; in (b) we show the data packet as if it were sent along one long unswitched link, and in (c) we introduce the idealization that bandwidth delay (and thus the width of the packet line) no longer matters.

(Most later ladder diagrams in this book are of this type.)



Long-distance packet transmission A→B and ACK B→A

(a) through switches (b) over a long cable (c) idealized

5.1.2 Bandwidth × Delay

The **bandwidth** × **delay** product (usually involving round-trip delay, or RTT), represents how much we can send before we hear anything back, or how much is "pending" in the network at any one time if we send continuously. Note that, if we use RTT instead of one-way time, then half the "pending" packets will be returning ACKs. Here are a few approximate values, where 100 ms can be taken as a typical inter-continental-distance RTT:

RTT	bandwidth	bandwidth × delay
1 ms	10 Mbps	1.2 KB
100 ms	1.5 Mbps	20 KB
100 ms	600 Mbps	8 MB
100 ms	1.5 Gbps	20 MB

5.2 Packet Delay Variability

For many links, the bandwidth delay and the propagation delay are rigidly fixed quantities, the former by the bandwidth and the latter by the speed of light. This leaves queuing delay as the major source of variability.

This state of affairs lets us define RTT_{noLoad} to be the time it takes to transmit a packet from A to B, and receive an acknowledgment back, with no queuing delay.

While this is often a reasonable approximation, it is not necessarily true that RTT_{noLoad} is always a fixed quantity. There are several possible causes for RTT variability. On Ethernet and Wi-Fi networks there is an initial "contention period" before transmission actually begins. Although this delay is related to waiting for other senders, it is not exactly queuing delay, and a packet may encounter considerable delay here even if it ends up being the first to be sent. For Wi-Fi in particular, the uncertainty introduced by collisions into packet delivery times – even with no other senders competing – can complicate higher-level delay measurements.

It is also possible that different packets are routed via slightly different paths, leading to (hopefully) minor variations in travel time, or are handled differently by different queues of a parallel-processing switch.

A link's bandwidth, too, can vary dynamically. Imagine, for example, a T1 link comprised of the usual 24 DS0 channels, in which all channels not currently in use by voice calls are consolidated into a single data channel. With eight callers, the data bandwidth would be cut by a third from 24 × DS0 to 16 × DS0. Alternatively, perhaps routers are allowed to *reserve* a varying amount of bandwidth for high-priority traffic, depending on demand, and so the bandwidth allocated to the best-effort traffic can vary. Perceived link bandwidth can also vary over time if packets are compressed at the link layer, and some packets are able to be compressed more than others.

Finally, if mobile nodes are involved, then the distance and thus the propagation delay can change. This can be quite significant if one is communicating with a wireless device that is being taken on a cross-continental road trip.

Despite these sources of fluctuation, we will usually assume that RTT_{noLoad} is fixed and well-defined, especially when we wish to focus on the queuing component of delay.

5.3 Packet Size

How big should packets be? Should they be large (*eg* 64 KB) or small (*eg* 48 bytes)?

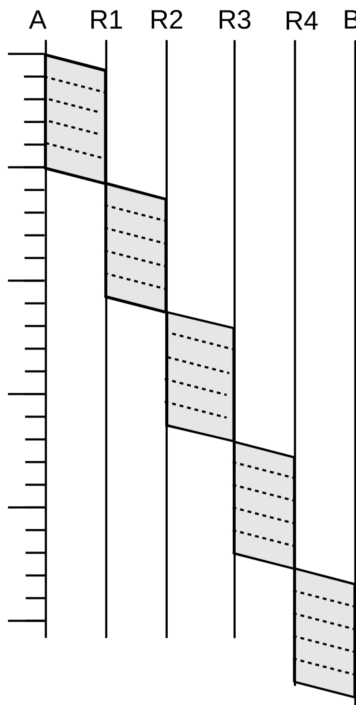
The Ethernet answer to this question had to do with equitable sharing of the line: large packets would not allow other senders timely access to transmit. In any network, this issue remains a concern.

On the other hand, large packets waste a smaller percentage of bandwidth on headers. However, in most of the cases we will consider, this percentage does not exceed 10% (the VoIP/RTP example in *1.3 Packets* is an exception).

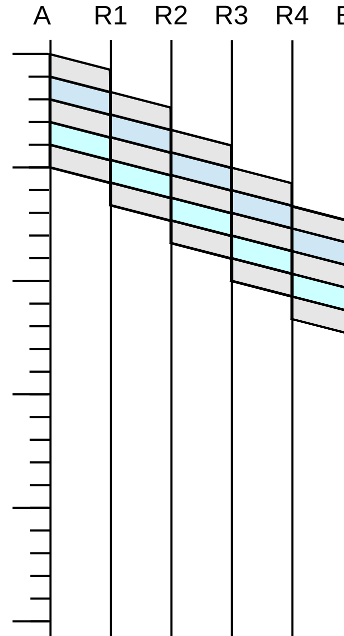
It turns out that if store-and-forward switches are involved, smaller packets have much better throughput. The links on either side of the switch can be in use simultaneously, as in Case 4 of *5.1.1 Delay examples*. This is a very real effect, and has put a damper on interest in support for IP "jumbograms". The ATM protocol (intended for both voice and data) pushes this to an extreme, with packets with only 48 bytes of data and 5 bytes of header.

As an example of this, consider a path from A to B with four switches and five links:

Suppose we send either one big packet or five smaller packets. The relative times from A to B are illustrated in the following figure:



One large packet over five links



Five smaller packets over five links

The point is that we can take advantage of parallelism: while the R4–B link above is handling packet 1, the R3–R4 link is handling packet 2 and the R2–R3 link is handling packet 3 and so on. The five smaller packets *would* have five times the header capacity, but as long as headers are small relative to the data, this is not a significant issue.

The sliding-windows algorithm, used by TCP, uses this idea as a continuous process: the sender sends a continual stream of packets which travel link-by-link so that, in the full-capacity case, all links may be in use at all times.

5.3.1 Error Rates and Packet Size

Packet size is also influenced, to a modest degree, by the transmission error rate. For relatively high error rates, it turns out to be better to send smaller packets, because when an error does occur then the entire packet containing it is lost.

> **Small error rates**
>
> Generally, if the bit error rate p is small, we can approximate the probability of error in an N-bit packet as $p \times N$, rather than working out the exact answer (assuming bit-error independence) of $1 - (1-p)^N$. This approximation works best if $p \times N$ is also small. For the 1000-bit example here with p=1/10,000, the exact value of the success rate is 90.4833% versus the $p \times N$ approximation of 90%. For the 10,000-bit packet, though, the $p \times N$ approximation predicts a 100% chance of error, which is not very helpful at all.

For example, suppose that 1 bit in 10,000 is corrupted, at random, so the probability that a single bit is

transmitted *correctly* is 0.9999 (this is much higher than the error rates encountered on real networks). For a 1000-bit packet, the probability that *every* bit in the packet is transmitted correctly is $(0.9999)^{1000}$, or about 90.5%. For a 10,000-bit packet the probability is $(0.9999)^{10,000} \simeq 37\%$. For 20,000-bit packets, the success rate is below 14%.

Now suppose we have 1,000,000 bits to send, either as 1000-bit packets or as 20,000-bit packets. Nominally this would require 1,000 of the smaller packets, but because of the 90% packet-success rate we will need to retransmit 10% of these, or 100 packets. Some of the retransmissions may also be lost; the total number of packets we expect to need to send is about 1,000/90% $\simeq$ 1,111, for a total of 1,111,000 bits sent. Next, let us try this with the 20,000-bit packets. Here the success rate is so poor that each packet needs to be sent on average *seven times*; lossless transmission would require 50 packets but we in fact need $7\times 50 = 350$ packets, or 7,000,000 bits.

Moral: choose the packet size small enough that most packets do not encounter errors.

To be fair, very large packets can be sent reliably on most cable links (*eg* TDM and SONET). Wireless, however, is more of a problem.

5.3.2 Packet Size and Real-Time Traffic

There is one other concern regarding excessive packet size. As we shall see in *20 Quality of Service*, it is common to commingle bulk traffic on the same links with real-time traffic. It is straightforward to give priority to the real-time traffic in such a mix, meaning that a router does not begin forwarding a bulk-traffic packet if there are any real-time packets waiting (we do need to be sure in this case that real-time traffic will not amount to so much as to starve the bulk traffic). However, once a bulk-traffic packet has begun transmission, it is impractical to interrupt it.

Therefore, one component of any maximum-delay bound for real-time traffic is the transmission time for the largest *bulk*-traffic packet; we will call this the **largest-packet delay**. As a practical matter, most IPv4 packets are limited to the maximum Ethernet packet size of 1500 bytes, but IPv6 has an option for so-called "jumbograms" up to 2 MB in size. Transmitting one such packet on a 100 Mbps link takes about 1/6 of a second, which is likely too large for happy coexistence with real-time traffic.

5.4 Error Detection

The basic strategy for packet error detection is to add some extra bits – formally known as an **error-detection code** – that will allow the receiver to determine if the packet has been corrupted in transit. A corrupted packet will then be discarded by the receiver; higher layers do not distinguish between lost packets and those never received. While packets lost due to bit errors occur much less frequently than packets lost due to queue overflows, it is essential that data be received accurately.

Intermittent packet errors generally fall into two categories: low-frequency bit errors due to things like cosmic rays, and *interference* errors, typically generated by nearby electrical equipment. Errors of the latter type generally occur in *bursts*, with multiple bad bits per packet. Occasionally, a malfunctioning network device will introduce bursty errors as well.

Networks v Refrigerators

At Loyola we once had a workstation used as a mainframe terminal that kept losing its connection. We eventually noticed that the connection dropped every time the office refrigerator kicked on. Sure enough, the cable ran directly behind the fridge; rerouting it solved the problem.

The simplest error-detection mechanism is a single parity bit; this will catch all one-bit errors. There is, however, no straightforward generalization to N bits! That is, there is no N-bit error code that catches all N-bit errors; see exercise 9.0.

The so-called **Internet checksum**, used by IP, TCP and UDP, is formed by taking the *ones-complement* sum of the 16-bit words of the message. Ones-complement is an alternative way of representing signed integers in binary; if one adds two positive integers and the sum does not overflow the hardware word size, then ones-complement and the now-universal *twos-complement* are identical. To form the ones-complement sum of 16-bit words A and B, first take the ordinary twos-complement sum A+B. Then, if there is an overflow bit, add it back in as low-order bit. Thus, if the word size is 4 bits, the ones-complement sum of 0101 and 0011 is 1000 (no overflow). Now suppose we want the ones-complement sum of 0101 and 1100. First we take the "exact" sum and get 1|0001, where the leftmost 1 is an overflow bit past the 4-bit wordsize. Because of this overflow, we add this bit back in, and get 0010.

The 4-bit ones-complement numeric representation has two forms for zero: 0000 and 1111 (it is straightforward to verify that any 4-bit quantity plus 1111 yields the original quantity; in twos-complement notation 1111 represents -1, and an overflow is guaranteed, so adding back the overflow bit cancels the -1 and leaves us with the original number). It is a fact that the ones-complement sum is never 0000 unless all bits of all the summands are 0; if the summands add up to zero by coincidence, then the actual binary representation will be 1111. This means that we can use 0000 in the checksum to represent "checksum not calculated", which the UDP protocol still allows over IPv4 for efficiency reasons. Over IPv6, UDP packets must include a calculated checksum (**RFC 2460**, §8.1).

Ones-complement

Long ago, before Loyola had any Internet connectivity, I wrote a primitive UDP/IP stack to allow me to use the Ethernet to back up one machine that did not have TCP/IP to another machine that did. We used "private" IP addresses of the form 10.0.0.x. I set as many header fields to zero as I could. I paid no attention to how to implement ones-complement addition; I simply used twos-complement, for the IP header only, and did not use a UDP checksum at all. Hey, it worked.
Then we got a real Class B address block 147.126.0.0/16, and changed IP addresses. My software no longer worked. It turned out that, in the original version, the IP header bytes were all small enough that when I added up the 16-bit words there were no carries, and so ones-complement was the same as twos-complement. With the new addresses, this was no longer true. As soon as I figured out how to implement ones-complement addition properly, my backups worked again.

Ones-complement addition has a few properties that make numerical calculations simpler. First, when finding the ones-complement sum of a series of 16-bit values, we can defer adding in the overflow bits until the end. Specifically, we can find the ones-complement sum of the values by adding them using ordinary (twos-complement) 32-bit addition, and then forming the ones-complement sum of the upper and lower 16-bit half-words. The upper half-word here represents the accumulated overflow. See exercise 10.0.

We can also find the ones-complement sum of a series of 16-bit values by concatenating them pairwise

into 32-bit values, taking the 32-bit ones-complement sum of these, and then, as in the previous paragraph, forming the ones-complement sum of the upper and lower 16-bit half-words.

Somewhat surprisingly, when calculating the 16-bit ones-complement sum of a series of bytes taken two at a time, it does not matter whether we convert the pairs of consecutive bytes to integers using big-endian or little-endian byte order (*11.1.5 Binary Data*). The overflow from the low-order bytes is added to the high-order bytes by virtue of ordinary carries in addition, and the overflow from the high-order bytes is added to the low-order bytes by the ones-complement rule. See exercise 10.5.

Finally, there is another way to look at the (16-bit) ones-complement sum: it is in fact the *remainder* upon dividing the message (seen as a very long binary number) by 2^{16} - 1, provided we replace a remainder of 0 with the equivalent ones-complement zero value consisting of sixteen 1-bits. This is similar to the decimal "casting out nines" rule: if we add up the digits of a base-10 number, and repeat the process until we get a single digit, then that digit is the remainder upon dividing the original number by 10-1 = 9. The analogy here is that the message is looked at as a very large number written in base-2^{16}, where the "digits" are the 16-bit words. The process of repeatedly adding up the "digits" until we get a single "digit" amounts to taking the ones-complement sum of the words. This remainder approach to ones-complement addition isn't very practical, but it does provide a useful way to analyze ones-complement checksums mathematically.

A weakness of any error-detecting code based on *sums* is that transposing words leads to the same sum, and the error is not detected. In particular, if a message is fragmented and the fragments are reassembled in the wrong order, the ones-complement sum will likely not detect it.

While some error-detecting codes are better than others at detecting certain kinds of systematic errors (for example, CRC, below, is usually better than the Internet checksum at detecting transposition errors), ultimately the effectiveness of an error-detecting code depends on its length. Suppose a packet P1 is corrupted *randomly* into P2, but still has its original N-bit error code EC(P1). This N-bit code will **fail** to detect the error that has occurred if EC(P2) is, *by chance*, equal to EC(P1). The probability that two *random* N-bit codes will match is $1/2^N$ (though a small random change in P1 might not lead to a uniformly distributed random change in EC(P1); see the tail end of the CRC section below).

This does not mean, however, that one packet in 2^N will be received incorrectly, as most packets are error-free. If we use a 16-bit error code, and only 1 packet in 100,000 is actually corrupted, then the rate at which corrupted packets will sneak by is only 1 in 100,000 × 65536, or about one in 6×10^9. If packets are 1500 bytes, you have a good chance (90+%) of accurately transferring a terabyte, and a 37% chance (1/e) at ten terabytes.

5.4.1 Cyclical Redundancy Check: CRC

The CRC error code is based on long division of polynomials; the use of polynomials tends to sound complicated but in fact it reduces all the long-division addition/subtraction operations to the much-simpler XOR operation. We treat the message, in binary, as a giant polynomial m(X), using the bits of the message as successive coefficients (*eg* 10011011 = $X^7 + X^4 + X^3 + X + 1$). We standardize a divisor polynomial p(X) of degree N (N=32 for CRC-**32** codes); the full specification of a given CRC code requires giving this polynomial. (A full specification also requires spelling out the bit order within bytes.) We append N 0-bits to m(X) (this is the polynomial X^Nm(X)), and divide the result by p(X). The "checksum" is the remainder r(X), of maximum degree N–1 (that is, N bits).

This is a reasonably secure hash against real-world network corruption, in that it is very hard for systematic errors to result in the same hash code. However, CRC is not secure against *intentional* corruption; given

msg1, there are straightforward algebraic means for tweaking the last bytes of msg2 so that so CRC(msg1) = CRC(msg2)

As an example of CRC, suppose that the CRC divisor is 1011 (making this a CRC-3 code) and the message is 0110 0001 1100. Here is the division:

```
             1110 0110
           ______________
1011      |  0110 0001 1100
              101 1
              ——— —
              011 10
               10 11
               —— ——
               01 010
                1 011
                — ———
                0 0011 11
                     10 11
                     —— ——
                     01 000
                      1 011
                      — ———
                      0 0110
```

The remainder, at the bottom, is 110; this is the CRC code. If we append the code to the message, which algebraically is $X^N m(X) + r(X)$, we get a polynomial which yields a remainder of zero upon division by p(X). As this appended message is what is actually transmitted, this slightly simplifies the receiver's job of validating the CRC code: the receiver just has to check that the remainder is zero.

CRC is easily implemented in hardware, using bit-shifting registers. Fast software implementations are also possible, usually involving handling the bits one byte at a time, with a precomputed lookup table with 256 entries.

If we randomly change *enough* bits in packet P1 to create P2, then CRC(P1) and CRC(P2) are effectively independent random variables, with probability of a match 1 in 2^N where N is the CRC length. However, if we change just a *few* bits then the change is *not* so random. In particular, for many CRC codes (that is, for many choices of the underlying polynomial p(X)), changing up to three bits in P1 to create a new message P2 guarantees that CRC(P1) $\neq$ CRC(P2). For the Internet checksum, this is not guaranteed even if we know only two bits were changed.

Finally, there are also **secure hashes**, such as MD-5 and SHA-1 and their successors (*22.6 Secure Hashes*). Nobody knows (or admits to knowing) how to produce two messages with same hash here. However, these secure-hash codes are generally not used in network error-correction as they are *much* slower to calculate than CRC; they are generally used only for secure authentication and other higher-level functions.

5.4.2 Error-Correcting Codes

If a link is noisy, we can add an error-*correction* code (also called *forward error correction*) that allows the receiver in many cases to figure out which bits are corrupted, and fix them. This has the effect of improving the bit error rate at a cost of reducing throughput. Error-correcting codes tend to involve many more bits than are needed for error detection. Typically, if a communications technology proves to have an unacceptably

high bit-error rate (such as wireless), the next step is to introduce an error-correcting code to the protocol. This generally reduces the "virtual" bit-error rate (that is, the error rate as corrected) to acceptable levels.

Perhaps the easiest error-correcting code to visualize is 2-D parity, for which we need $O(N^{1/2})$ additional bits. We take N×N data bits and arrange them into a square; we then compute the parity for every column, for every row, and for the entire square; this is 2N+1 extra bits. Here is a diagram with N=4, and with even parity; the column-parity bits (in blue) are in the bottom (fifth) row and the row-parity bits (also in blue) are in the rightmost (fifth) column. The parity bit for the entire 4×4 data square is the light-blue bit in the bottom right corner.

0	1	1	0	0
0	1	1	1	1
1	0	1	0	0
1	1	1	1	0
0	1	0	0	1

Now suppose one bit is corrupted; for simplicity, assume it is one of the data bits. Then exactly one column-parity bit will be incorrect, and exactly one row-parity bit will be incorrect. These two incorrect bits mark the column and row of the incorrect data bit, which we can then flip to the correct state.

We can make N large, but an essential requirement here is that there be only a single corrupted bit per square. We are thus likely either to keep N small, or to choose a different code entirely that allows correction of multiple bits. Either way, the addition of error-correcting codes can easily increase the size of a packet significantly; some codes double or even triple the total number of bits sent.

5.4.2.1 Hamming Codes

The Hamming code is another popular error-correction code; it adds O(log N) additional bits, though if N is large enough for this to be a material improvement over the $O(N^{1/2})$ performance of 2-D parity then errors must be very infrequent. If we have 8 data bits, let us number the bit positions 0 through 7. We then write each bit's position as a binary value between 000 and 111; we will call these the **position bits** of the given data bit. We now add four code bits as follows:

1. a parity bit over all 8 data bits
2. a parity bit over those data bits for which the first digit of the position bits is 1 (these are positions 4, 5, 6 and 7)
3. a parity bit over those data bits for which the second digit of the position bits is 1 (these are positions 010, 011, 110 and 111, or 2, 3, 6 and 7)
4. a parity bit over those data bits for which the third digit of the position bits is 1 (these are positions 001, 011, 101, 111, or 1, 3, 5 and 7)

We can tell whether or not an error has occurred by the first code bit; the remaining three code bits then tell us the respective three position bits of the incorrect bit. For example, if the #2 code bit above is correct, then

the first digit of the position bits is 0; otherwise it is one. With all three position bits, we have identified the incorrect data bit.

As a concrete example, suppose the data word is 10110010. The four code bits are thus

1. 0, the (even) parity bit over all eight bits
2. 1, the parity bit over the second half, 1011**0010**
3. 1, the parity bit over the bold bits: 10**11**00**10**
4. 1, the parity bit over these bold bits: 1**0**1**1**0**0**1**0**

If the received data+code is now 1011**1**010 0111, with the bold bit flipped, then the fact that the first code bit is wrong tells the receiver there was an error. The second code bit is also wrong, so the first bit of the position bits must be 1. The third code bit is right, so the second bit of the position bits must be 0. The fourth code bit is also right, so the third bit of the position bits is 0. The position bits are thus binary 100, or 4, and so the receiver knows that the incorrect bit is in position 4 (counting from 0) and can be flipped to the correct state.

5.5 Epilog

The issues presented here are perhaps not very glamorous, and often play a supporting, behind-the-scenes role in protocol design. Nonetheless, their influence is pervasive; we may even think of them as part of the underlying "physics" of the Internet.

As the early Internet became faster, for example, and propagation delay became the dominant limiting factor, protocols were often revised to limit the number of back-and-forth exchanges. A classic example is the Simple Mail Transport Protocol (SMTP), amended by **RFC 1854** to allow multiple commands to be sent together – called pipelining – instead of individually.

While there have been periodic calls for large-packet support in IPv4, and IPv6 protocols exist for "jumbograms" in excess of a megabyte, these are very seldom used, due to the store-and-forward costs of large packets as described in *5.3 Packet Size*.

Almost every LAN-level protocol, from Ethernet to Wi-Fi to point-to-point links, incorporates an error-detecting code chosen to reflect the underlying transportation reliability. Ethernet includes a 32-bit CRC code, for example, while Wi-Fi includes extensive error-correcting codes due to the noisier wireless environment. The Wi-Fi fragmentation option (*3.7.1.5 Wi-Fi Fragmentation*) is directly tied to *5.3.1 Error Rates and Packet Size*.

5.6 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercises marked with a ♢ have solutions or hints at 24.5 Solutions for Packets.

1.0. Suppose a link has a propagation delay of 20 µsec and a bandwidth of 2 bytes/µsec.

(a). How long would it take to transmit a 600-byte packet over such a link?

(b). How long would it take to transmit the 600-byte packet over two such links, with a store-and-forward switch in between?

2.0. Suppose the path from A to B has a single switch S in between: A──S──B. Each link has a propagation delay of 60 µsec and a bandwidth of 2 bytes/µsec.

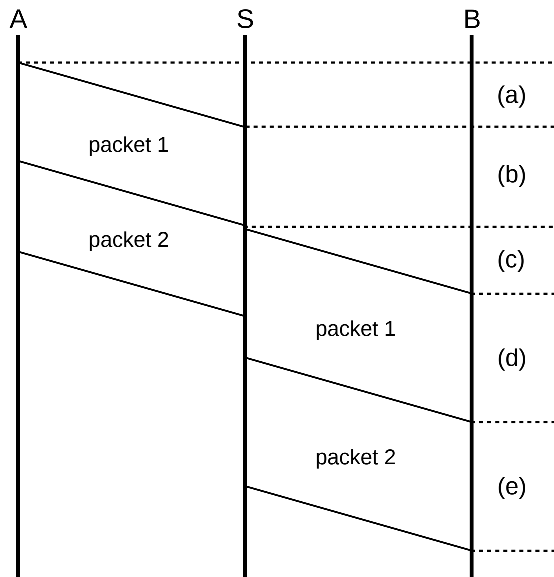
(a). How long would it take to send a single 600-byte packet from A to B?

(b). How long would it take to send two back-to-back 300-byte packets from A to B?

(c). How long would it take to send three back-to-back 200-byte packets from A to B?

3.0.◇ Repeat parts (a) and (b) of the previous exercise, except change the per-link propagation delay from 60 µsec to 600 µsec.

3.5. Suppose the path from A to B has a single switch S in between: A──S──B. The propagation delays on the A–S and S–B are 24 µsec and 35 µsec respectively. The per-packet bandwidth delays on the A–S and S–B links are 103 µsec and 157 µsec respectively. The ladder diagram below describes the sending of two consecutive packets from A to B. Label the time intervals (a) through (e) at the right edge, and give the total time for the packets to be sent.



4.0. Again suppose the path from A to B has a single switch S in between: A──S──B. The per-link bandwidth and propagation delays are as follows:

link	bandwidth	propagation delay
A—S	5 bytes/µsec	24 µsec
S—B	3 bytes/µsec	13 µsec

(a). How long would it take to send a single 600-byte packet from A to B?

(b). How long would it take to send two back-to-back 300-byte packets from A to B? Note that, because the S—B link is slower, packet 2 arrives at S from A before S has finished transmitting packet 1 to B.

5.0. Suppose in the previous exercise, the A–S link has the smaller bandwidth of 3 bytes/µsec and the S–B link has the larger bandwidth of 5 bytes/µsec. The propagation delays are unchanged. Now how long does it take to send two back-to-back 300-byte packets from A to B?

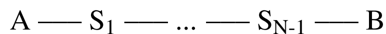
6.0. Suppose we have five links, A──R1──R2──R3──R4──B. Each link has a bandwidth of 100 bytes/ms. Assume we model the per-link propagation delay as 0.

(a). How long would it take a single 1500-byte packet to go from A to B?

(b). How long would it take five consecutive 300-byte packets to go from A to B?

The diagram in *5.3 Packet Size* may help.

7.0. Suppose there are N equal-bandwidth links on the path between A and B, as in the diagram below, and we wish to send M consecutive packets.



Let BD be the bandwidth delay of a single packet on a single link, and assume the propagation delay on each link is zero. Show that the total (bandwidth) delay is (M+N-1)×BD. Hint: the total time is the sum of the time A takes to begin transmitting the last packet, and the time that last packet (or any other packet) takes to travel from A to B. Show that the former is (M-1)×BD and the latter is N×BD. Note that no packets ever have to wait at any S_i because the ith packet takes exactly as long to arrive as the (i-1)th packet takes to depart.

8.0. Repeat the analysis in *5.3.1 Error Rates and Packet Size* to compare the probable total number of bytes that need to be sent to transmit 10^7 bytes using

(a). 1,000-byte packets

(b). 10,000-byte packets

Assume the bit error rate is 1 in 16×10^5, making the error rate per *byte* about 1 in 2×10^5.

9.0. In the text it is claimed "there is no N-bit error code that catches all N-bit errors" for N⩾2 (for N=1, a parity bit works). Prove this claim for N=2. Hint: pick a length M, and consider all M-bit messages with a *single* 1-bit. Any such message can be converted to any other with a 2-bit error. Show, using the Pigeonhole Principle, that for large enough M two messages m_1 and m_2 must have the same error code, that is, $e(m_1) = e(m_2)$. If this occurs, then the error code fails to detect the error that converted m_1 into m_2.

10.0. Consider the following four-bit numbers, with decimal values in parentheses:

1000 (8)

1011 (11)

1101 (13)
1110 (14)

The ones-complement sum of these can be found using the division method by treating these as a four-digit hex number 0x8bde and taking the remainder mod 15; the result is 1.

(a). Find this ones-complement sum via three 4-bit ones-complement additions. To get started, note that the (exact) sum of 1000 and 1011 is 1|0011, and adding the carry bit to the low-order 4 bits gives a ones-complement sum of the first pair of 0100.

(b). The exact (and 8-bit twos-complement) sum of the values above is 46, or 10|1110 in binary. Find the ones-complement sum of the values by taking this exact sum and then forming the ones-complement sum of the 4-bit high and low halves. Note that this is not the same as the twos-complement sum of the halves.

10.5. Let [a,b] denote a pair of bytes a and b. The 16-bit integer corresponding to [a,b] using big-endian conversion is a×256 + b; using little-endian conversion it is a + 256×b.

(a). Find the ones-complement sum of [200,150] and [90,230] by using big-endian conversion to the respective 16-bit integers 51,350 and 23,270. Convert back to two bytes, again using big-endian conversion, at the end.

(b). Do the same using little-endian conversion, in which case the 16-bit integers are 38,600 and 58,970.

11.0. Suppose a message is 110010101. Calculate the CRC-3 checksum using the polynomial $X^3 + 1$, that is, find the 3-bit remainder using divisor 1001.

12.0. The CRC algorithm presented above requires that we process one bit at a time. It is possible to do the algorithm N bits at a time (*eg* N=8), with a precomputed lookup table of size 2^N. Complete the steps in the following description of this strategy for N=3 and polynomial $X^3 + X + 1$, or 1011.

13.0. Consider the following set of bits sent with 2-D even parity; the data bits are in the 4×4 upper-left block and the parity bits are in the rightmost column and bottom row. Which bit is corrupted?

1	1	0	1	1
0	1	0	0	1
1	1	1	1	1
1	0	0	1	0
1	1	1	0	1

14.0. (a) Show that 2-D parity can *detect* any three errors.

(b). Find four errors that cannot be detected by 2-D parity.

(c). Show that that 2-D parity cannot *correct* all two-bit errors. Hint: put both bits in the same row or column.

15.0. Each of the following 8-bit messages with 4-bit Hamming code contains a single error. Correct the message.

(a)◇. 10100010 0111

(b). 10111110 1011

16.0 (a) What happens in 2-D parity if the corrupted bit is in the parity column or parity row?

(b). In the following 8-bit message with 4-bit Hamming code, there is an error in the *code* portion. How can this be determined?

> 1001 1110 0100

In this chapter we take a general look at how to build reliable data-transport layers on top of unreliable lower layers. This is achieved through a **retransmit-on-timeout** policy; that is, if a packet is transmitted and there is no acknowledgment received during the timeout interval then the packet is resent. As a class, protocols where one side implements retransmit-on-timeout are known as **ARQ** protocols, for Automatic Repeat reQuest.

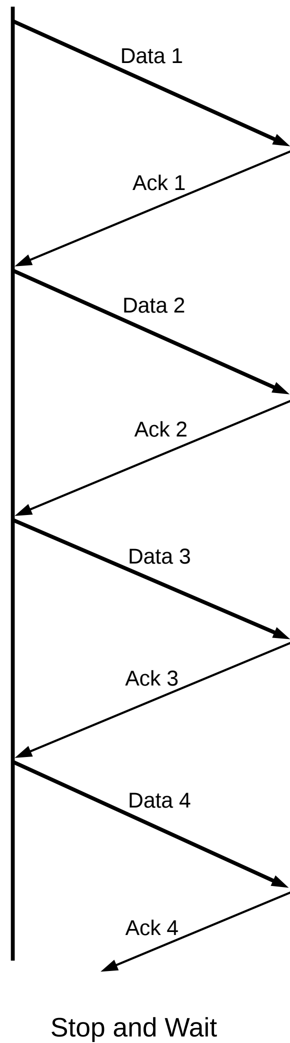
In addition to reliability, we also want to keep as many packets in transit as the network can support. The strategy used to achieve this is known as **sliding windows**. It turns out that the sliding-windows algorithm is also the key to managing congestion; we return to this in *13 TCP Reno and Congestion Management*.

The *End-to-End* principle, *12.1 The End-to-End Principle*, suggests that trying to achieve a reliable transport layer by building reliability into a lower layer is a misguided approach; that is, implementing reliability at the endpoints of a connection – as is described here – is in fact the correct mechanism.

6.1 Building Reliable Transport: Stop-and-Wait

Retransmit-on-timeout generally requires sequence numbering for the packets, though if a network path is guaranteed not to reorder packets then it is safe to allow the sequence numbers to wrap around surprisingly quickly (for stop-and-wait, a single-bit sequence number will work; see exercise 8.5). However, as the no-reordering hypothesis does not apply to the Internet at large, we will assume conventional numbering. Data[N] will be the Nth data packet, acknowledged by ACK[N].

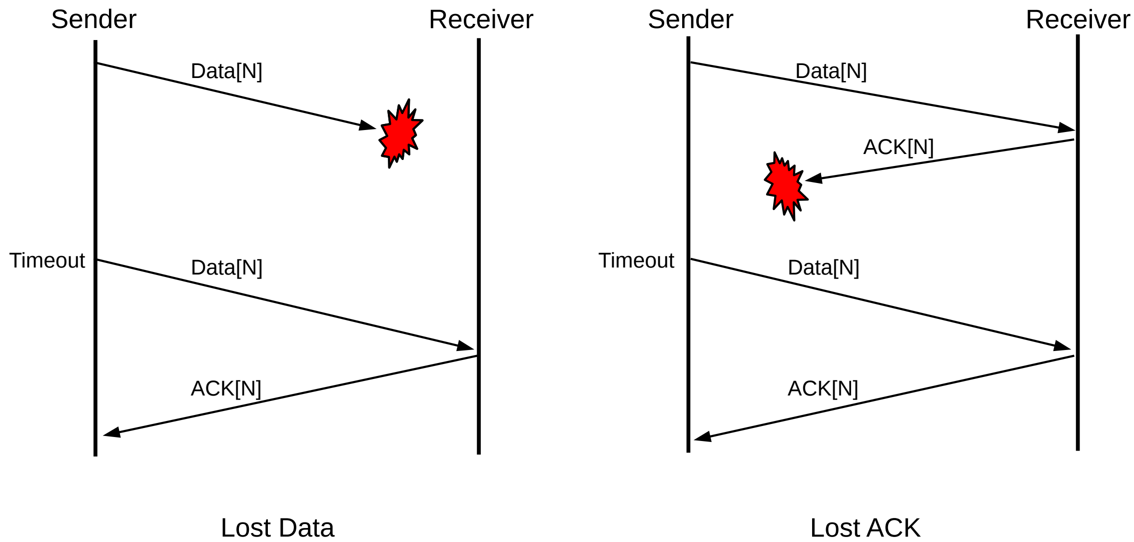
In the **stop-and-wait** version of retransmit-on-timeout, the sender sends only one outstanding packet at a time. If there is no response, the packet may be retransmitted, but the sender does not send Data[N+1] until it has received ACK[N]. Of course, the receiving side will not send ACK[N] until it has received Data[N]; *each* side has only one packet in play at a time. In the absence of packet loss, this leads to the following:



Stop and Wait

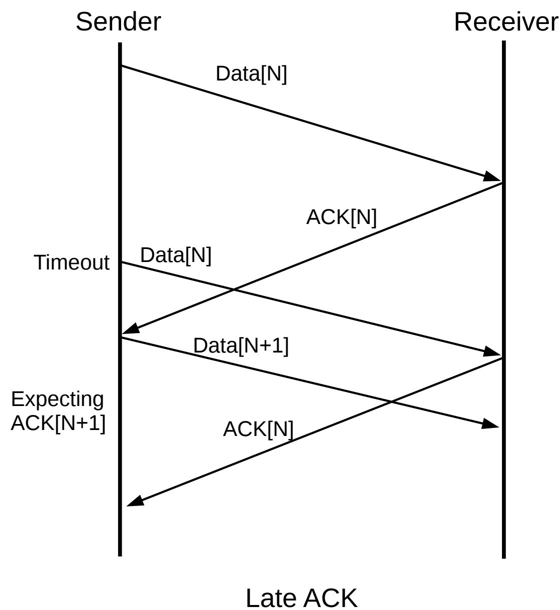
6.1.1 Packet Loss

Lost packets, however, are a reality. The left half of the diagram below illustrates a lost Data packet, where the sender is the host sending Data and the Receiver is the host sending ACKs. The receiver is not aware of the loss; it sees Data[N] as simply slow to arrive.



The right half of the diagram, by comparison, illustrates the case of a lost ACK. The receiver has received a *duplicate* Data[N]. We have assumed here that the receiver has implemented a **retransmit-on-duplicate** strategy, and so its response upon receipt of the duplicate Data[N] is to retransmit ACK[N].

As a final example, note that it is possible for ACK[N] to have been delayed (or, similarly, for the first Data[N] to have been delayed) longer than the timeout interval. Not every packet that times out is actually lost!



In this case we see that, after sending Data[N], receiving a delayed ACK[N] (rather than the expected ACK[N+1]) *must be considered a normal event*.

In principle, either side can implement retransmit-on-timeout if nothing is received. Either side can also implement retransmit-on-duplicate; this was done by the receiver in the second example above but *not* by the sender in the third example (the sender received a second ACK[N] but did not retransmit Data[N+1]).

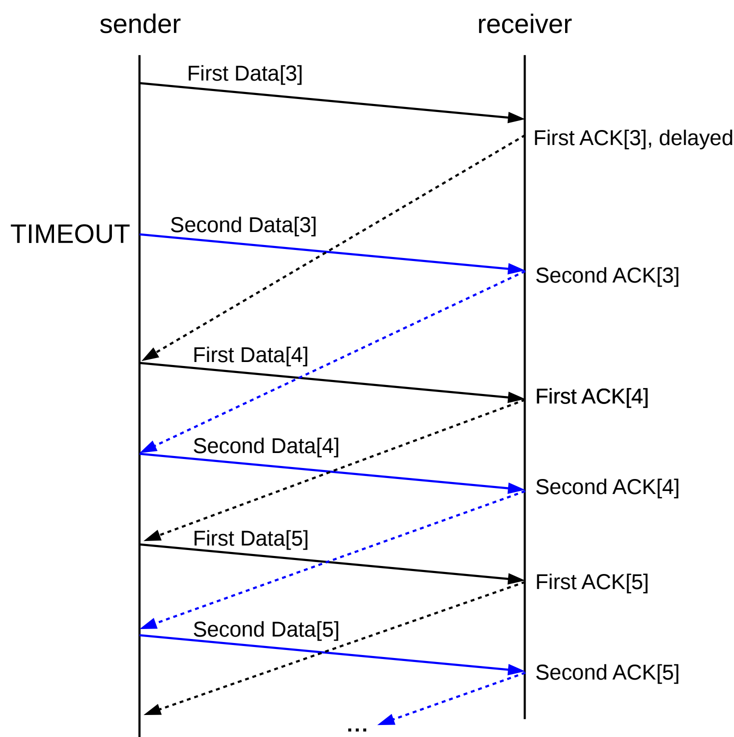
At least one side *must* implement retransmit-on-timeout; otherwise a lost packet leads to deadlock as the sender and the receiver both wait forever. The other side *must* implement at least one of retransmit-on-duplicate or retransmit-on-timeout; usually the former alone. If both sides implement retransmit-on-timeout with different timeout values, generally the protocol will still work.

6.1.2 Sorcerer's Apprentice Bug

Sorcerer's Apprentice

The Sorcerer's Apprentice bug is named for the legend in which the apprentice casts a spell on a broom to carry water, one bucket at a time. When the basin is full, the apprentice chops the broom in half, only to find both halves carrying water. See Disney's Fantasia, at around T = 5:35.

A strange thing happens if one side implements retransmit-on-timeout but *both* sides implement retransmit-on-duplicate, as can happen if the implementer takes the naive view that retransmitting on duplicates is "safer"; the moral here is that too much redundancy can be the Wrong Thing. Let us imagine that an implementation uses this strategy (with the sender retransmitting on timeouts), and that the initial ACK[3] is delayed until after Data[3] is retransmitted on timeout. In the following diagram, the only packet retransmitted due to timeout is the second Data[3]; all the other duplications are due to the bilateral retransmit-on-duplicate strategy.



All packets are sent *twice* from Data[3] on. The transfer completes normally, but takes double the normal bandwidth. The usual fix is to have one side (usually the sender) retransmit on timeout only. TCP does this; see *12.19 TCP Timeout and Retransmission*. See also exercise 1.5.

6.1.3 Flow Control

Stop-and-wait also provides a simple form of **flow control** to prevent data from arriving at the receiver faster than it can be handled. Assuming the time needed to process a received packet is less than one RTT, the stop-and-wait mechanism will prevent data from arriving too fast. If the processing time is slightly larger than RTT, all the receiver has to do is to wait to send ACK[N] until Data[N] has not only arrived but also been processed, and the receiver is *ready* for Data[N+1].

For modest per-packet processing delays this works quite well, but if the processing delays are long it introduces a new problem: Data[N] may time out and be retransmitted even though it has successfully been received; the receiver cannot send an ACK until it has finished processing. One approach is to have *two* kinds of ACKs: ACK_{WAIT}[N] meaning that Data[N] has arrived but the receiver is not yet ready for Data[N+1], and ACK_{GO}[N] meaning that the sender may now send Data[N+1]. The receiver will send ACK_{WAIT}[N] when Data[N] arrives, and ACK_{GO}[N] when it is done processing it.

Presumably we want the sender not to time out and retransmit Data[N] after ACK_{WAIT}[N] is received, as a retransmission would be unnecessary. This introduces a new problem: if the subsequent ACK_{GO}[N] is lost and neither side times out, the connection is deadlocked. The sender is waiting for ACK_{GO}[N], which is lost, and the receiver is waiting for Data[N+1], which the sender will not send until the lost ACK_{GO}[N] arrives. One solution is for the receiver to switch to a timeout model, perhaps until Data[N+1] is received.

TCP has a fix to the flow-control problem involving sender-side polling; see *12.17 TCP Flow Control*.

6.2 Sliding Windows

Stop-and-wait is reliable but it is not very efficient (unless the path involves neither intermediate switches nor significant propagation delay; that is, the path involves a single LAN link). Most links along a multi-hop stop-and-wait path will be idle most of the time. During a file transfer, ideally we would like zero idleness (at least along the slowest link; see *6.3 Linear Bottlenecks*).

We can improve overall throughput by allowing the sender to continue to transmit, sending Data[N+1] (and beyond) *without* waiting for ACK[N]. We cannot, however, allow the sender get *too* far ahead of the returning ACKs. Packets sent too fast, as we shall see, simply end up waiting in queues, or, worse, dropped from queues. If the links of the network have sufficient bandwidth, packets may also be dropped at the receiving end.

Now that, say, Data[3] and Data[4] may be simultaneously in transit, we have to revisit what ACK[4] means: does it mean that the receiver has received only Data[4], or does it mean both Data[3] and Data[4] have arrived? We will assume the latter, that is, ACKs are **cumulative**: ACK[N] cannot be sent until Data[K] has arrived for all $K \leq N$. With this understanding, if ACK[3] is lost then a later-arriving ACK[4] makes up for it; without it, if ACK[3] is lost the only recovery is to retransmit Data[3].

The sender picks a **window size**, winsize. The basic idea of sliding windows is that the sender is allowed to send this many packets before waiting for an ACK. More specifically, the sender keeps a state variable

last_ACKed, representing the last packet for which it has received an ACK from the other end; if data packets are numbered starting from 1 then initially last_ACKed = 0.

Window Size

In this chapter we will assume winsize does not change. TCP, however, varies winsize up and down with the goal of making it as large as possible without introducing congestion; we will return to this in *13 TCP Reno and Congestion Management*.

At any instant, the sender may send packets numbered last_ACKed + 1 through last_ACKed + winsize; this packet range is known as the **window**. Generally, if the first link in the path is not the slowest one, the sender will most of the time have sent all these.

If ACK[N] arrives with N > last_ACKed (typically N = last_ACKed+1), then the window *slides forward*; we set last_ACKed = N. This also increments the upper edge of the window, and frees the sender to send more packets. For example, with winsize = 4 and last_ACKed = 10, the window is [11,12,13,14]. If ACK[11] arrives, the window slides forward to [12,13,14,15], freeing the sender to send Data[15]. If instead ACK[13] arrives, then the window slides forward to [14,15,16,17] (recall that ACKs are cumulative), and three more packets become eligible to be sent. If there is no packet reordering and no packet losses (and every packet is ACKed individually) then the window will slide forward in units of one packet at a time; the next arriving ACK will always be ACK[last_ACKed+1].

Note that the rate at which ACKs are returned will always be exactly equal to the rate at which the slowest link is delivering packets. That is, if the slowest link (the "bottleneck" link) is delivering a packet every 50 ms, then the receiver will receive those packets every 50 ms and the ACKs will return at a rate of one every 50 ms. Thus, new packets will be sent at an average rate exactly matching the delivery rate; this is the sliding-windows **self-clocking** property. Self-clocking has the effect of reducing congestion by automatically reducing the sender's *rate* whenever the available fraction of the bottleneck bandwidth is reduced.

Below is a video of sliding windows in action, with winsize = 5. (A link is here, if the embedded video does not display properly, which will certainly be the case with non-html formats.) The nodes are labeled 0, 1 and 2. The second link, 1–2, has a capacity of five packets in transit either way, so one "flight" (windowful) of five packets can exactly fill this link. The 0–1 link has a capacity of one packet in transit either way. The video was prepared using the network animator, "nam", described further in *16 Network Simulations: ns-2*.

The first flight of five data packets leaves node 0 just after T=0, and leaves node 1 at around T=1 (in video time). Subsequent flights are spaced about seven seconds apart. The tiny packets moving leftwards from node 2 to node 0 represent ACKs; at the very beginning of the video one can see five returning ACKs from the previous windowful. At any moment (except those instants where packets have just been received) there are in principle five packets in transit, either being transmitted on a link as data, or being transmitted as an ACK, or sitting in a queue (this last does not happen in this video). Due to occasional video artifacts, in some frames not all the ACK packets are visible.

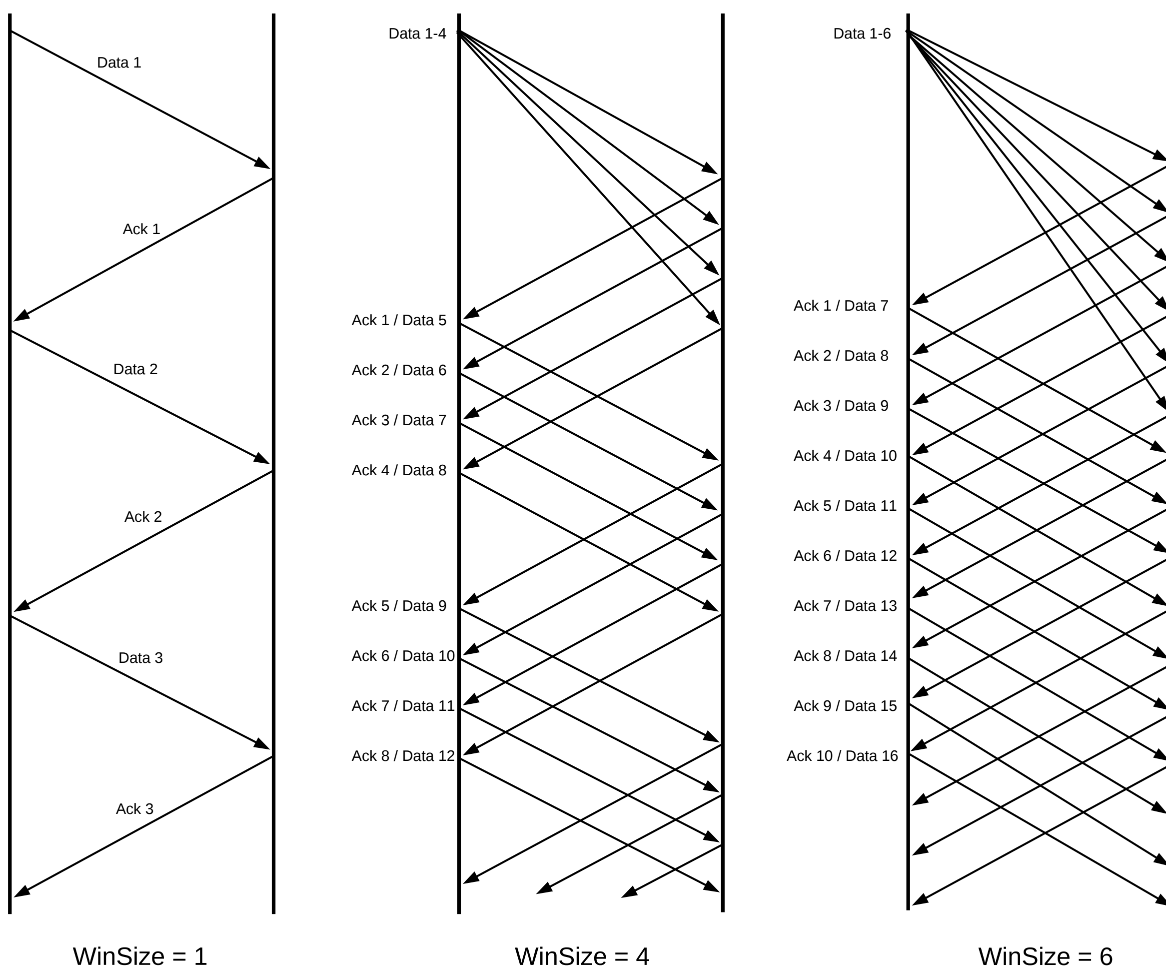
6.2.1 Bandwidth × Delay

As indicated previously (*5.1 Packet Delay*), the bandwidth × RTT product represents the amount of data that can be sent before the first response is received. It plays a large role in the analysis of transport protocols. In the literature the bandwidth×delay product is often abbreviated BDP.

The bandwidth × RTT product is generally the optimum value for the window size. There is, however, one catch: if a sender chooses winsize larger than this, then the RTT simply grows – due to queuing delays – to the point that bandwidth × RTT matches the chosen winsize. That is, a connection's own traffic can inflate RTT_{actual} to well above RTT_{noLoad}; see *6.3.1.3 Case 3: winsize = 6* below for an example. For this reason, a sender is often more interested in bandwidth × RTT_{noLoad}, or, at the very least, the RTT before the sender's own packets had begun contributing to congestion.

We will sometimes refer to the bandwidth × RTT_{noLoad} product as the **transit capacity** of the route. As will become clearer below, a winsize smaller than this means underutilization of the network, while a larger winsize means each packet spends time waiting in a queue somewhere.

Below are simplified diagrams for sliding windows with window sizes of 1, 4 and 6, each with a path bandwidth of 6 packets/RTT (so bandwidth × RTT = 6 packets). The diagram shows the initial packets sent as a burst; these then would be spread out as they pass through the bottleneck link so that, after the first burst, packet spacing is uniform. (Real sliding-windows protocols such as TCP generally attempt to avoid such initial bursts.)



Sliding Windows, bandwidth 6 packets/RTT

With winsize=1 we send 1 packet per RTT; with winsize=4 we always *average* 4 packets per RTT. To put this another way, the three window sizes lead to bottle-neck link utilizations of 1/6, 4/6 and 6/6 = 100%,

respectively.

While it is tempting to envision setting winsize to bandwidth × RTT, in practice this can be complicated; neither bandwidth nor RTT is constant. Available bandwidth can fluctuate in the presence of competing traffic. As for RTT, if a sender sets winsize too large then the RTT is simply inflated to the point that bandwidth × RTT matches winsize; that is, a connection's own traffic can inflate RTT_{actual} to well above RTT_{noLoad}. This happens even in the absence of competing traffic.

6.2.2 The Receiver Side

Perhaps surprisingly, sliding windows can work pretty well with the receiver assuming that winsize=1, even if the sender is in fact using a much larger value. Each of the receivers in the diagrams above receives Data[N] and responds with ACK[N]; the only difference with the larger *sender* winsize is that the Data[N] arrive faster.

If we are using the sliding-windows algorithm over single links, we may assume packets are never reordered, and a receiver winsize of 1 works quite well. Once switches are introduced, however, life becomes more complicated (though some links may do *link-level* sliding-windows for per-link throughput optimization).

If packet reordering is a possibility, it is common for the receiver to use the same winsize as the sender. This means that the receiver must be prepared to buffer a full window full of packets. If the window is [11,12,13,14,15,16], for example, and Data[11] is delayed, then the receiver may have to buffer Data[12] through Data[16].

Like the sender, the receiver will also maintain the state variable last_ACKed, though it will not be completely synchronized with the sender's version. At any instant, the receiver is willing to accept Data[last_ACKed+1] through Data[last_ACKed+winsize]. For any but the first of these, the receiver must buffer the arriving packet. If Data[last_ACKed+1] arrives, then the receiver should consult its buffers and send back the largest cumulative ACK it can for the data received; for example, if the window is [11-16] and Data[12], Data[13] and Data[15] are in the buffers, then on arrival of Data[11] the correct response is ACK[13]. Data[11] fills the "gap", and the receiver has now received everything up through Data[13]. The new receive window is [14-19], and as soon as the ACK[13] reaches the sender that will be the new send window as well.

6.2.3 Loss Recovery Under Sliding Windows

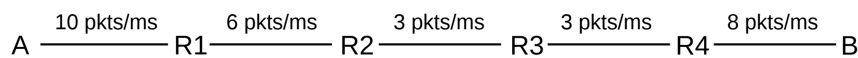
Suppose winsize = 4 and packet 5 is lost. It is quite possible that packets 6, 7, and 8 may have been received. However, the only (cumulative) acknowledgment that can be sent back is ACK[4]; the sender does not know how much of the windowful made it through. Because of the possibility that *only* Data[5] (or more generally Data[last_ACKed+1]) is lost, and because losses are usually associated with congestion, when we most especially do *not* wish to overburden the network, the sender will usually retransmit only the first lost packet, *eg* packet 5. If packets 6, 7, and 8 were also lost, then after retransmission of Data[5] the sender will receive ACK[5], and can assume that Data[6] now needs to be sent. However, if packets 6-8 did make it through, then after retransmission the sender will receive back ACK[8], and so will know 6-8 do not need retransmission and that the next packet to send is Data[9].

Normally Data[6] through Data[8] would time out shortly after Data[5] times out. After the first timeout, however, sliding windows protocols generally suppress further timeout/retransmission responses until recovery is more-or-less complete.

Once a full timeout has occurred, usually the sliding-windows process itself has ground to a halt, in that there are usually no packets remaining in flight. This is sometimes described as **pipeline drain**. After recovery, the sliding-windows process will have to start up again. Most implementations of TCP, as we shall see later, implement a mechanism ("fast recovery") for early detection of packet loss, before the pipeline has fully drained.

6.3 Linear Bottlenecks

Consider the simple network path shown below, with bandwidths shown in packets/ms. The minimum bandwidth, or **path bandwidth**, is 3 packets/ms.



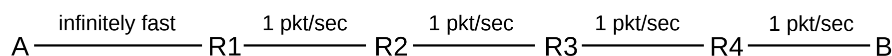
The slow links are R2–R3 and R3–R4. We will refer to the slowest link as the **bottleneck** link; if there are (as here) ties for the slowest link, then the first such link is the bottleneck. The bottleneck link is where the queue will form. If traffic is sent at a rate of 4 packets/ms from A to B, it will pile up in an ever-increasing queue at R2. Traffic will *not* pile up at R3; it arrives at R3 at the same rate by which it departs.

Furthermore, if sliding windows is used (rather than a fixed-*rate* sender), traffic will eventually not queue up at any router other than R2: data cannot reach B faster than the 3 packets/ms rate, and so B will not return ACKs faster than this rate, and so A will eventually not *send* data faster than this rate. At this 3 packets/ms rate, traffic will not pile up at R1 (or R3 or R4).

There is a significant advantage in speaking in terms of winsize rather than transmission *rate*. If A sends to B at any rate greater than 3 packets/ms, then the situation is unstable as the bottleneck queue grows without bound and there is no convergence to a steady state. There is no analogous instability, however, if A uses sliding windows, even if the winsize chosen is quite large (although a large-enough winsize will overflow the bottleneck queue). If a sender specifies a sending window size rather than a rate, then the network will converge to a steady state in relatively short order; if a queue develops it will be steadily replenished at the same rate that packets depart, and so will be of fixed size.

6.3.1 Simple fixed-window-size analysis

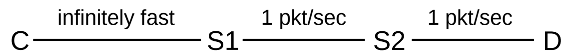
We will analyze the effect of window size on overall throughput and on RTT. Consider the following network path, with bandwidths now labeled in packets/**second**.



We will assume that in the backward B⟶A direction, all connections are infinitely fast, meaning zero delay; this is often a good approximation because ACK packets are what travel in that direction and they are negligibly small. In the A⟶B direction, we will assume that the A⟶R1 link is infinitely fast, but

the other four each have a bandwidth of 1 packet/second (and no propagation-delay component). This makes the R1⟶R2 link the **bottleneck link**; any queue will now form at R1. The "path bandwidth" is 1 packet/second, and the RTT is 4 seconds.

As a roughly equivalent alternative example, we might use the following:



with the following assumptions: the C–S1 link is infinitely fast (zero delay), S1⟶S2 and S2⟶D each take 1.0 sec bandwidth delay (so two packets take 2.0 sec, per link, etc), and ACKs also have a 1.0 sec bandwidth delay in the reverse direction.

In both scenarios, if we send one packet, it takes 4.0 seconds for the ACK to return, on an idle network. This means that the no-load delay, RTT_{noLoad}, is 4.0 seconds.

(These models will change significantly if we replace the 1 packet/sec bandwidth delay with a 1-second *propagation* delay; in the former case, 2 packets take 2 seconds, while in the latter, 2 packets take 1 second. See exercise 4.0.)

We assume a single connection is made; *ie* there is no competition. Bandwidth × delay here is 4 packets (1 packet/sec × 4 sec RTT)

6.3.1.1 Case 1: winsize = 2

In this case winsize < bandwidth×delay (where delay = RTT). The table below shows what is sent by A and each of R1-R4 for each second. Every packet is acknowledged 4 seconds after it is sent; that is, RTT_{actual} = 4 sec, equal to RTT_{noLoad}; this will remain true as the winsize changes by small amounts (*eg* to 1 or 3). Throughput is proportional to winsize: when winsize = 2, throughput is 2 packets in 4 seconds, or 2/4 = 1/2 packet/sec. During each second, two of the routers R1-R4 are idle. The overall path will have less than 100% utilization.

Time	A	R1	R1	R2	R3	R4	B
T	sends	queues	sends	sends	sends	sends	ACKs
0	1,2	2	1				
1			2	1			
2				2	1		
3					2	1	
4	3		3			2	1
5	4		4	3			2
6				4	3		
7					4	3	
8	5		5			4	3
9	6		6	5			4

Note the brief pile-up at R1 (the bottleneck link!) on startup. However, in the steady state, there is no queuing. Real sliding-windows protocols generally have some way of minimizing this "initial pileup".

6.3.1.2 Case 2: winsize = 4

When winsize=4, at each second all four slow links are busy. There is again an initial burst leading to a brief surge in the queue; RTT_{actual} for Data[4] is 7 seconds. However, RTT_{actual} for every subsequent packet is 4 seconds, and there are no queuing delays (and nothing in the queue) after T=2. The steady-state connection throughput is 4 packets in 4 seconds, *ie* 1 packet/second. Note that overall path throughput now equals the bottleneck-link bandwidth, so this is the best possible throughput.

T	A sends	R1 queues	R1 sends	R2 sends	R3 sends	R4 sends	B ACKs
0	1,2,3,4	2,3,4	1				
1		3,4	2	1			
2		4	3	2	1		
3			4	3	2	1	
4	5		5	4	3	2	1
5	6		6	5	4	3	2
6	7		7	6	5	4	3
7	8		8	7	6	5	4
8	9		9	8	7	6	5

At T=4, R1 has just finished sending Data[4] as Data[5] arrives from A; R1 can begin sending packet 5 immediately. No queue will develop.

Case 2 is the "congestion knee" of Chiu and Jain *[CJ89]*, defined here in *1.7 Congestion*.

6.3.1.3 Case 3: winsize = 6

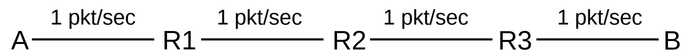
T	A sends	R1 queues	R1 sends	R2 sends	R3 sends	R4 sends	B ACKs
0	1,2,3,4,5,6	2,3,4,5,6	1				
1		3,4,5,6	2	1			
2		4,5,6	3	2	1		
3		5,6	4	3	2	1	
4	7	6,7	5	4	3	2	1
5	8	7,8	6	5	4	3	2
6	9	8,9	7	6	5	4	3
7	10	9,10	8	7	6	5	4
8	11	10,11	9	8	7	6	5
9	12	11,12	10	9	8	7	6
10	13	12,13	11	10	9	8	7

Note that packet 7 is sent at T=4 and the acknowledgment is received at T=10, for an RTT of 6.0 seconds. All later packets have the same RTT_{actual}. That is, the RTT has risen from RTT_{noLoad} = 4 seconds to 6 seconds. *Note that we continue to send one windowful each RTT*; that is, the throughput is still winsize/RTT, but RTT is now 6 seconds.

One might initially conjecture that if winsize is greater than the bandwidth×RTT_{noLoad} product, then the entire window cannot be in transit at one time. In fact this is not the case; the sender *does* usually have the entire window sent and in transit, but **RTT has been inflated** so it appears to the sender that winsize *equals* the bandwidth×RTT product.

In general, whenever winsize > bandwidth×RTT_{noLoad}, what happens is that the extra packets pile up at a router somewhere along the path (specifically, at the router in front of the bottleneck link). RTT_{actual} is inflated by queuing delay to winsize/bandwidth, where bandwidth is that of the bottleneck link; this means winsize = bandwidth×RTT_{actual}. Total throughput is equal to that bandwidth. Of the 6 seconds of RTT_{actual} in the example here, a packet spends 4 of those seconds being transmitted on one link or another because RTT_{noLoad}=4. The other two seconds, therefore, must be spent in a queue; there is no other place for packets wait. Looking at the table, we see that each second there are indeed two packets in the queue at R1.

If the bottleneck link is the very first link, packets may begin returning before the sender has sent the entire windowful. In this case we may argue that the full windowful has at least been queued by the sender, and thus has in this sense been "sent". Suppose the network, for example, is



where, as before, each link transports 1 packet/sec from A to B and is infinitely fast in the reverse direction. Then, if A sets winsize = 6, a queue of 2 packets will form at A.

6.3.2 RTT Calculations

We can make some quantitative observations of sliding windows behavior, and about queue utilization. First, we note that RTT_{noLoad} is the physical "travel" time (subject to the limitations addressed in *5.2 Packet Delay Variability*); any time in excess of RTT_{noLoad} is spent waiting in a queue somewhere. Therefore, the following holds regardless of competing traffic, and even for individual packets:

1. queue_time = RTT_{actual} – RTT_{noLoad}

When the bottleneck link is saturated, that is, is always busy, the number of packets actually in transit (not queued) somewhere along the path will always be bandwidth × RTT_{noLoad}.

Second, we always send one windowful per actual RTT, assuming no losses and each packet is individually acknowledged. This is perhaps best understood by consulting the diagrams above, but here is a simple non-visual argument: if we send Data[N] at time T_D, and ACK[N] arrives at time T_A, then RTT = T_A – T_D, by definition. At time T_A the sender is allowed to send Data[N+winsize], so during the RTT interval $T_D \leqslant T < T_A$ the sender must have sent Data[N] through Data[N+winsize-1]; that is, winsize many packets in time RTT. Therefore (whether or not there is competing traffic) we always have

2. throughput = winsize/RTT_{actual}

where "throughput" is the rate at which the connection is sending packets.

This relationship holds even if winsize or the bottleneck bandwidth changes suddenly, though in that case RTT_{actual} might change from one packet to the next, and the throughput here must be seen as a measurement averaged over the RTT of one specific packet. If the sender doubles its winsize, those extra packets will immediately end up in a queue somewhere (perhaps a queue at the sender itself, though this is why in examples it is often clearer if the first link has infinite bandwidth so as to prevent this). If the bottleneck bandwidth is cut in half without changing winsize, eventually the RTT must rise due to queuing. See exercise 12.0.

In the sliding windows *steady state*, where throughput and RTT_{actual} are reasonably constant, the average number of packets in the queue is just throughput×queue_time (where throughput is measured in packets/sec):

3. queue_usage = throughput × (RTT_{actual} – RTT_{noLoad})
   = winsize × (1 – $RTT_{noLoad}/RTT_{actual}$)

To give a little more detail making the averaging perhaps clearer, each packet spends time (RTT_{actual} – RTT_{noLoad}) in the queue, from equation 1 above. The total time spent by a windowful of packets is winsize × (RTT_{actual} – RTT_{noLoad}), and dividing this by RTT_{actual} thus gives the average number of packets in the queue over the RTT interval in question.

In the presence of competing traffic, the throughput referred to above is simply the connection's current share of the total bandwidth. It is the value we get if we measure the rate of returning ACKs. If there is *no* competing traffic and winsize is below the congestion knee – winsize < bandwidth × RTT_{noLoad} – then winsize is the limiting factor in throughput. Finally, if there is no competition and winsize ⩾ bandwidth × RTT_{noLoad} then the connection is using 100% of the capacity of the bottleneck link and throughput is equal to the bottleneck-link physical bandwidth. To put this another way,

4. RTT_{actual} = winsize/bottleneck_bandwidth

   queue_usage = winsize – bandwidth × RTT_{noLoad}

Dividing the first equation by RTT_{noLoad}, and noting that bandwidth × RTT_{noLoad} = winsize - queue_usage = transit_capacity, we get

5. **$RTT_{actual}/RTT_{noLoad}$ = winsize/transit_capacity** = (transit_capacity + queue_usage) / transit_capacity

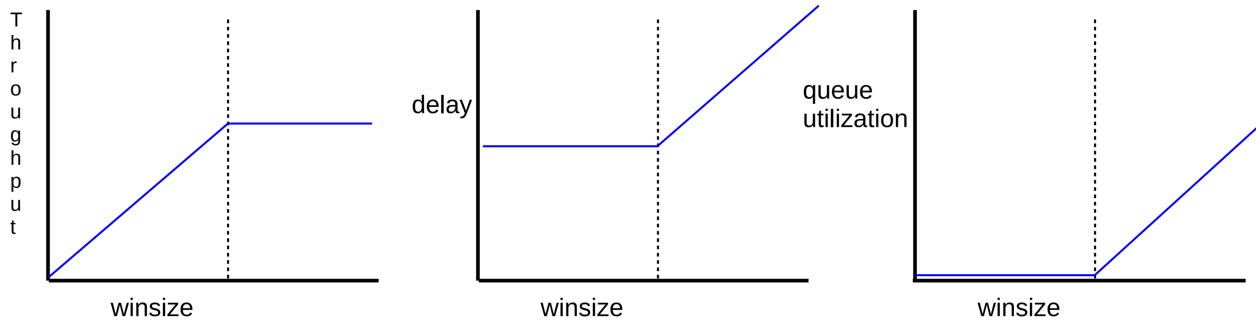
Regardless of the value of winsize, in the steady state the sender never sends faster than the bottleneck bandwidth. This is because the bottleneck bandwidth determines the rate of packets arriving at the far end, which in turn determines the rate of ACKs arriving back at the sender, which in turn determines the continued sending rate. This illustrates the self-clocking nature of sliding windows.

We will return in *14 Dynamics of TCP Reno* to the issue of bandwidth in the presence of competing traffic. For now, suppose a sliding-windows sender has winsize > bandwidth × RTT_{noLoad}, leading as above to a fixed amount of queue usage, and no competition. Then another connection starts up and competes for the bottleneck link. The first connection's *effective* bandwidth will thus decrease. This means that bandwidth × RTT_{noLoad} will decrease, and hence the connection's queue usage will increase.

6.3.3 Graphs at the Congestion Knee

Consider the following graphs of winsize versus

1. throughput
2. delay
3. queue utilization



Graphs of winsize versus throughput, delay and queue utilization.
Vertical dashed line represents winsize = bandwidth x no-load delay

The critical winsize value is equal to bandwidth × RTT_{noLoad}; this is known as the congestion **knee**. For winsize below this, we have:

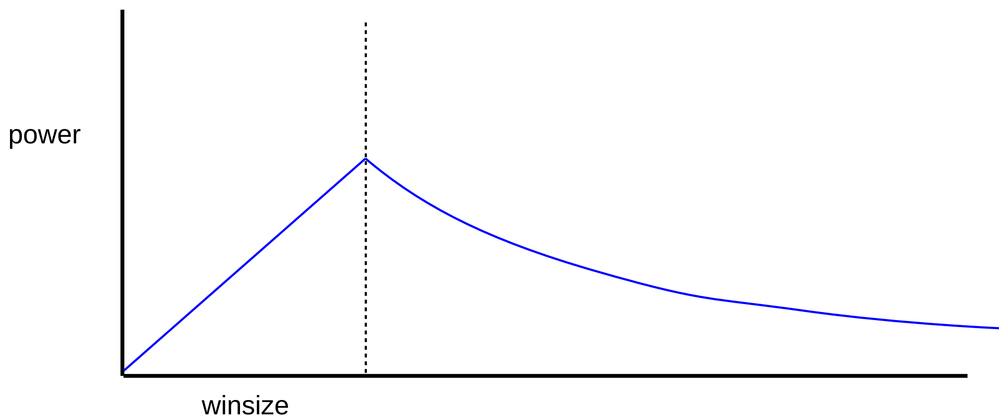
- throughput is proportional to winsize
- delay is constant
- queue utilization in the steady state is zero

For winsize larger than the knee, we have

- throughput is constant (equal to the bottleneck bandwidth)
- delay increases linearly with winsize
- queue utilization increases linearly with winsize

Ideally, winsize will be at the critical knee. However, the exact value varies with time: available bandwidth changes due to the starting and stopping of competing traffic, and RTT changes due to queuing. Standard TCP makes an effort to stay well *above* the knee much of the time, presumably on the theory that maximizing throughput is more important than minimizing queue use.

The **power** of a connection is defined to be throughput/RTT. For sliding windows below the knee, RTT is constant and power is proportional to the window size. For sliding windows above the knee, throughput is constant and delay is proportional to winsize; power is thus proportional to 1/winsize. Here is a graph, akin to those above, of winsize versus power:



6.3.4 Simple Packet-Based Sliding-Windows Implementation

Here is a pseudocode outline of the receiver side of a sliding-windows implementation, ignoring lost packets and timeouts. We abbreviate as follows:

> W: winsize
>
> LA: last_ACKed

Thus, the next packet expected is LA+1 and the window is [LA+1, ..., LA+W]. We have a data structure EarlyArrivals in which we can place packets that cannot yet be delivered to the receiving application.

Upon arrival of Data[M]:

```
if M⩽LA or M>LA+W, ignore the packet
if M>LA+1, put the packet into EarlyArrivals.
if M==LA+1:
    deliver the packet (that is, Data[LA+1]) to the application
    LA = LA+1 (slide window forward by 1)
    while (Data[LA+1] is in EarlyArrivals) {
        output Data[LA+1]
        LA = LA+1
    }
    send ACK[LA]
```

A possible implementation of EarlyArrivals is as an array of packet objects, of size W. We always put packet Data[M] into position M % W.

At any point between packet arrivals, Data[LA+1] is not in EarlyArrivals, but some later packets may be present.

For the sender side, we begin by sending a full windowful of packets Data[1] through Data[W], and setting LA=0. When ACK[M] arrives, LA<M⩽LA+W, the window slides forward from [LA+1...LA+W] to [M+1...M+W], and we are now allowed to send Data[LA+W+1] through Data[M+W]. The simplest case is M=LA+1.

Upon arrival of ACK[M]:

```
if M⩽LA or M>LA+W, ignore the packet
otherwise:
    set K = LA+W+1, the first packet just above the old window
    set LA = M, just below the bottom of the new window
    for (i=K; i⩽LA+W; i++) send Data[i]
```

Note that new ACKs may arrive while we are in the loop at the last line. We assume here that the sender stolidly sends what it may send and only after that does it start to process additional arriving ACKs. Some implementations may take a more asynchronous approach, perhaps with one thread processing arriving ACKs and incrementing LA and another thread sending everything it is allowed to send.

To add support for timeout and retransmission, each transmitted packet would need to be stored, together with the time it was sent. Periodically this collection of stored packets must then be scanned, looking for

packets for which `send_time + timeout_interval ⩽ current_time`; those packets get retransmitted. When a packet Data[N] is acknowledged (perhaps by an ACK[M] for M>N), it can be deleted.

6.4 Epilog

This completes our discussion of the sliding-windows algorithm in the abstract setting. We will return to concrete implementations of this in *11.4.1 TFTP and the Sorcerer* (stop-and-wait) and in *12.14 TCP Sliding Windows*; the latter is one of the most important mechanisms on the Internet.

6.5 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercise 1.5 is distinct, for example, from exercises 1.0 and 2.0. Exercises marked with a ♢ have solutions or hints at 24.6 Solutions for Sliding Windows.

1.0 Sketch a ladder diagram for stop-and-wait if Data[3] is lost the first time it is sent, assuming **no sender timeout** (but the sender retransmits on duplicate), and a *receiver* timeout of 2 seconds. Continue the diagram to the point where Data[4] is successfully transmitted. Assume an RTT of 1 second.

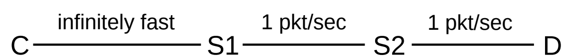
1.5 Re-draw the Sorcerer's Apprentice diagram of *6.1.2 Sorcerer's Apprentice Bug*, assuming the sender now does *not* retransmit on duplicates, though the receiver still does. ACK[3] is, as before, delayed until the sender retransmits Data[3].

2.0 Suppose a stop-and-wait receiver has an implementation flaw. When Data[1] arrives, ACK[1] and ACK[2] are sent, separated by a brief interval; after that, the receiver transmits ACK[N+1] when Data[N] arrives, rather than the correct ACK[N].

(a). Assume the sender responds to each ACK as soon as it arrives. Explain why the sender will not be able to detect this receiver problem until after it has sent its final packet, Data[M], and receives an unexpected ACK[M+1]. Hint: draw a diagram.

(b). Is there anything the transmitter can do to detect this receiver problem earlier?

2.5♢ Consider the alternative model of *6.3.1 Simple fixed-window-size analysis*:



(a). Using the formulas of *6.3.2 RTT Calculations*, calculate the steady-state queue usage for a window size of 6.

(b). Again for a window size of 6, create a table like those in *6.3.1 Simple fixed-window-size analysis* up through T=8 seconds.

3.0 Create a table as in *6.3.1 Simple fixed-window-size analysis* for the original A—–R1—–R2—–R3—–R4—–B network with winsize = 8. As in the text examples, assume 1 packet/sec bandwidth delay for the R1⟶R2, R2⟶R3, R3⟶R4 and R4⟶B links. The A–R1 link and all reverse links (from B to A) are infinitely fast. Carry out the table for 10 seconds.

4.0 Create a table as in *6.3.1 Simple fixed-window-size analysis* for a network A—–R1—–R2—–B. The A–R1 ink is infinitely fast; the R1–R2 and R2–B each have a 1-second **propagation** delay, in each direction, and zero *bandwidth* delay (that is, one packet takes 1.0 sec to travel from R1 to R2; two packets also take 1.0 sec to travel from R1 to R2). Assume winsize=6. Carry out the table for 8 seconds. Note that with zero bandwidth delay, multiple packets sent together will remain together until the destination; propagation delay behaves very differently from bandwidth delay!

5.0 Suppose RTT_{noLoad} = 4 seconds and the bottleneck bandwidth is 1 packet every 2 seconds.

(a). What window size is needed to remain just at the knee of congestion?

(b). Suppose winsize=6. What is the eventual value of RTT_{actual}?

(c). Again with winsize=6, how many packets are in the queue at the steady state?

6.0 Create a table as in *6.3.1 Simple fixed-window-size analysis* for a network A—–R1—–R2—–R3—–B. The A–R1 link is infinitely fast. The R1–R2 and R3–B links have a bandwidth delay of 1 packet/second with no additional propagation delay. The R2–R3 link has a bandwidth delay of 1 packet / 2 seconds, and no propagation delay. The reverse B⟶A direction (for ACKs) is infinitely fast. Assume winsize = 6.

(a). Carry out the table for 10 seconds. Note that you will need to show the queue for both R1 and R2.

(b). Continue the table at least partially until T=18, in sufficient detail that you can verify that RTT_{actual} for packet 8 is as calculated in exercise 5.0. To do this you will need more than 10 packets, but fewer than 16; the use of hex labels A, B, C for packets 10, 11, 12 is a convenient notation.

Hint: The column for "R2 sends" (or, more accurately, "R2 is in the process of sending") should look like this:

T	R2 sends
0	
1	1
2	1
3	2
4	2
5	3
6	3
...	...

7.0 Argue that, if A sends to B using sliding windows, and in the path from A to B the slowest link is *not* the first link out of A, then eventually A will have the entire window outstanding (except at the instant just after each new ACK comes in).

7.5◇ Suppose RTT_{noLoad} is 100 ms and the available bandwidth is 1,000 packets/sec. Sliding windows is used.

(a). What is the transit capacity for the connection?

(b). If RTT_{actual} rises to 130 ms (due to use of a larger winsize), how many packets are in a queue at any one time?

(c). If winsize increases by 50, what is RTT_{actual}?

8.0 Suppose RTT_{noLoad} is 50 ms and the available bandwidth is 2,000 packets/sec. Sliding windows is used for transmission.

(a). What window size is needed to remain just at the knee of congestion?

(b). If RTT_{actual} rises to 60 ms (due to use of a larger winsize), how many packets are in a queue at any one time?

(c). What value of winsize would lead to RTT_{actual} = 60 ms?

(d). What value of winsize would make RTT_{actual} rise to 100 ms?

8.5 Suppose stop-and-wait is used (winsize=1), and assume that while packets may be lost, *they are never reordered* (that is, if two packets P1 and P2 are sent in that order, and both arrive, then they arrive in that order). Show that at the point the receiver is waiting for Data[N], the only two packet arrivals possible are Data[N] and Data[N-1]. (A consequence is that, in the absence of reordering, stop-and-wait can make do with 1-bit packet sequence numbers.) Hint: if the receiver is waiting for Data[N], it must have just received Data[N-1] and sent ACK[N-1]. Also, once the sender has sent Data[N], it will never transmit a Data[K] with K<N.

⋆9.0◇ Suppose winsize=4 in a sliding-windows connection, and assume that while packets may be lost, *they are never reordered* (that is, if two packets P1 and P2 are sent in that order, and both arrive, then they arrive in that order). Show that if Data[8] is in the receiver's window (meaning that everything up through Data[4] has been received and acknowledged), then it is no longer possible for even a late Data[0] to arrive at the receiver. (A consequence of the general principle here is that – *in the absence of reordering* – we can replace the packet sequence number with (sequence_number) mod (2×winsize+1) without ambiguity.)

10.0 Suppose winsize=4 in a sliding-windows connection, and assume as in the previous exercise that while packets may be lost, they are never reordered. Give an example in which Data[8] is in the receiver's window (so the receiver has presumably sent ACK[4]), and yet Data[1] legitimately arrives. (Thus, the late-packet bound in the previous exercise is the best possible.)

11.0 Suppose the network is A───R1───R2───B, where the A–R1 ink is infinitely fast and the R1–R2 link has a bandwidth of 1 packet/second each way, for an RTT_{noLoad} of 2 seconds. Suppose also that A begins sending with winsize = 6. By the analysis in *6.3.1.3 Case 3: winsize = 6*, RTT should rise to

winsize/bandwidth = 6 seconds. Give the RTTs of the first eight packets. How long does it take for RTT to rise to 6 seconds?

12.0 In this exercise we look at the relationship between bottleneck bandwidth and winsize/RTT_{actual} when the former changes suddenly. Suppose the network is as follows

The A—R1 link is infinitely fast. The R1→R2 link has a 1 packet/sec bandwidth delay in the R1→R2 direction. The remaining links R2→R3 and R3→B have a 1 sec bandwidth delay in the direction indicated. ACK packets, being small, travel instantaneously from B back to A.

A sends to B using a winsize of three. Three packets P0, P1 and P2 are sent at times T=0, T=1 and T=2 respectively.

At T=3, P0 arrives at B. At this instant the R1→R2 bandwidth is suddenly halved to 1 packet / 2 sec; P3 is transmitted at T=3 and arrives at R2 at T=5. It will arrive at B at T=7.

(a). Complete the following table of packet arrival times

T	A sends	R1's queue	R1 sends	R2 sends	R3 sends	B recvs/ACKs
2	P2		P2	P1	P0	
3	P3		P3	P2	P1	P0
4	P4	P4	P3 cont		P2	P1
5	P5	P5	P4	P3		P2
6		P5	P4 cont		P3	
7	P6					P3
8						
9	P7					
10						
11	P8					

(b). For each of P2, P3, P4 and P5, calculate the througput given by winsize/RTT over the course of that packet's round trip. Obtain each packet's RTT from the completed table above.

(c). Once the steady state is reached in which RTT_{actual} = 6, how much time does each packet spend in transit? How much time does each packet spend in R1's queue?

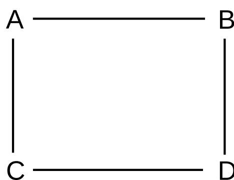
There are multiple LAN protocols below the IP layer and multiple transport protocols above, but IP itself stands alone. The Internet is essentially the IP Internet. If you want to run your own LAN protocol somewhere, or if you want to run your own transport protocol, the Internet backbone will still work just fine for you. But if you want to change the IP layer, you will encounter difficulty. (Just talk to the IPv6 people, or the IP-multicasting or IP-reservations groups.)

Currently the Internet uses (mostly, but no longer quite exclusively) IP version 4, with its 32-bit address size. As the Internet has run out of new large blocks of IPv4 addresses (*1.10 IP - Internet Protocol*), there is increasing pressure to convert to IPv6, with its 128-bit address size. Progress has been slow, however, and delaying tactics such as IPv4-address markets and NAT (*7.7 Network Address Translation*) – by which multiple hosts can share a single public IPv4 address – have allowed IPv4 to continue. Aside from the major change in address structure, there are relatively few differences in the routing models of IPv4 and IPv6. We will study IPv4 in this chapter and IPv6 in the following; at points where the IPv4/IPv6 difference doesn't much matter we will simply write "IP".

IPv4 (and IPv6) is, in effect, a universal **routing and addressing** protocol. Routing and addressing are developed together; every node has an IP address and every router knows how to handle IP addresses. IP was originally seen as a way to *inter*connect multiple LANs, but it may make more sense now to view IP as a virtual LAN overlaying all the physical LANs.

A crucial aspect of IP is its **scalability**. As the Internet has grown to ~10^9 hosts, the forwarding tables are not much larger than 10^5 (perhaps now $10^{5.5}$). Ethernet, in comparison, scales poorly.

Furthermore, IP, unlike Ethernet, offers excellent support for **multiple redundant links**. If the network below were an IP network, each node would communicate with each immediate neighbor via their shared direct link. If, on the other hand, this were an Ethernet network with the spanning-tree algorithm, then one of the four links would simply be disabled completely.



The IP network service model is to act like a giant LAN. That is, there are no acknowledgments; delivery is generally described as best-effort. This design choice is perhaps surprising, but it has also been quite fruitful.

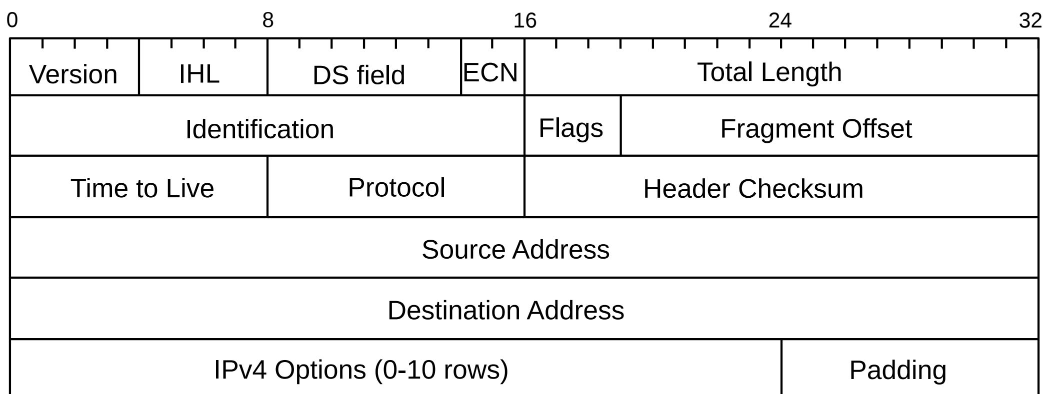
If you want to provide a universal service for delivering any packet anywhere, what else do you need besides routing and addressing? Every network (LAN) needs to be able to carry any packet. The protocols spell out the use of octets (bytes), so the only possible compatibility issue is that a packet is *too large* for a given network. IPv4 handles this by supporting **fragmentation**: a network may break a too-large packet up into units it can transport successfully. While IPv4 fragmentation is inefficient and clumsy, it does guarantee that any packet can potentially be delivered to any node. (Note, however, that IPv6 has given up on universal fragmentation; *8.5.4 IPv6 Fragment Header*.)

7.1 The IPv4 Header

The IPv4 Header needs to contain the following information:

- destination and source addresses
- indication of ipv4 versus ipv6
- a Time To Live (TTL) value, to prevent infinite routing loops
- a field indicating what comes next in the packet (*eg* TCP v UDP)
- fields supporting fragmentation and reassembly.

The header is organized as a series of 32-bit words as follows:



The IPv4 header, and basics of IPv4 protocol operation, were originally defined in **RFC 791**; some minor changes have since occurred. Most of these changes were documented in **RFC 1122**, though the DS field was defined in **RFC 2474** and the ECN bits were first proposed in **RFC 2481**.

The **Version** field is, for IPv4, the number 4: 0100. The **IHL** field represents the total IPv4 Header Length, in 32-bit words; an IPv4 header can thus be at most 15 words long. The base header takes up five words, so the IPv4 Options can consist of at most ten words. If one looks at IPv4 packets using a packet-capture tool that displays the packets in hex, the first byte will most often be 0x45.

The **Differentiated Services** (DS) field is used by the Differentiated Services suite to specify preferential handling for designated packets, *eg* those involved in VoIP or other real-time protocols. The **Explicit Congestion Notification** bits are there to allow routers experiencing congestion to mark packets, thus indicating to the sender that the transmission rate should be reduced. We will address these in *14.8.2 Explicit Congestion Notification (ECN)*. These two fields together replace the old 8-bit **Type of Service** field.

The **Total Length** field is present because an IPv4 packet may be smaller than the minimum LAN packet size (see Exercise 1) or larger than the maximum (if the IPv4 packet has been fragmented over several LAN packets. The IPv4 packet length, in other words, cannot be inferred from the LAN-level packet size. Because the Total Length field is 16 bits, the maximum IPv4 packet size is 2^{16} bytes. This is probably much too large, even if fragmentation were not something to be avoided (though see IPv6 "jumbograms" in *8.5.1 Hop-by-Hop Options Header*).

The second word of the header is devoted to fragmentation, discussed below.

The **Time-to-Live** (TTL) field is decremented by 1 at each router; if it reaches 0, the packet is discarded. A typical initial value is 64; it must be larger than the total number of hops in the path. In most cases, a value of 32 would work. The TTL field is there to prevent routing loops – always a serious problem should they occur – from consuming resources indefinitely. Later we will look at various IP routing-table update protocols and how they minimize the risk of routing loops; they do not, however, eliminate it. By comparison, Ethernet headers have no TTL field, but Ethernet also disallows cycles in the underlying topology.

The **Protocol** field contains a value to identify the contents of the packet body. A few of the more common values are

- 1: an ICMP packet, *7.11 Internet Control Message Protocol*
- 4: an encapsulated IPv4 packet, *7.13.1 IP-in-IP Encapsulation*
- 6: a TCP packet
- 17: a UDP packet
- 41: an encapsulated IPv6 packet, *8.13 IPv6 Connectivity via Tunneling*
- 50: an Encapsulating Security Payload, *22.11 IPsec*

A list of assigned protocol numbers is maintained by the IANA.

The **Header Checksum** field is the "Internet checksum" applied to the header only, not the body. Its only purpose is to allow the discarding of packets with corrupted headers. When the TTL value is decremented the router must update the header checksum. This can be done "algebraically" by adding a 1 in the correct place to compensate, but it is not hard simply to re-sum the 8 halfwords of the average header. The header checksum must also be updated when an IPv4 packet header is rewritten by a NAT router.

The **Source** and **Destination Address** fields contain, of course, the IPv4 addresses. These would normally be updated only by NAT firewalls.

The source-address field is supposed to be the sender's IPv4 address, but hardly any ISP checks that traffic they send out has a source address matching one of their customers, despite the call to do so in **RFC 2827**. As a result, IP **spoofing** – the sending of IP packets with a faked source address – is straightforward. For some examples, see *12.10.1 ISNs and spoofing*, and SYN flooding at *12.3 TCP Connection Establishment*.

IP-address spoofing also facilitates an all-too-common IP-layer **denial-of-service** attack in which a server is flooded with a huge volume of traffic so as to reduce the bandwidth available to legitimate traffic to a trickle. This flooding traffic typically originates from a large number of compromised machines. Without spoofing, even a lengthy list of sources can be blocked, but, with spoofing, this becomes quite difficult.

One IPv4 option is the **Record Route** option, in which routers are to insert their own IPv4 address into the IPv4 header option area. Unfortunately, with only ten words available, there is not enough space to record most longer routes (but see *7.11.1 Traceroute and Time Exceeded*, below). The **Timestamp** option is related; intermediate routers are requested to mark packets with their address and a local timestamp (to save space, the option can request only timestamps). There is room for only four ⟨address,timestamp⟩ pairs, but addresses can be **prespecified**; that is, the sender can include up to four IPv4 addresses and only those routers will fill in a timestamp.

Another option, now deprecated as security risk, is to support **source routing**. The sender would insert into the IPv4 header option area a list of IPv4 addresses; the packet would be routed to pass through each of those IPv4 addresses in turn. With **strict** source routing, the IPv4 addresses had to represent adjacent neighbors; no router could be used if its IPv4 address were not on the list. With **loose** source routing, the

listed addresses did not have to represent adjacent neighbors and ordinary IPv4 routing was used to get from one listed IPv4 address to the next. Both forms are essentially never used, again for security reasons: if a packet has been source-routed, it may have been routed outside of the at-least-somewhat trusted zone of the Internet backbone.

Finally, the IPv4 header was carefully laid out with memory alignment in mind. The 4-byte address fields are aligned on 4-byte boundaries, and the 2-byte fields are aligned on 2-byte boundaries. All this was once considered important enough that incoming packets were stored following two bytes of padding at the head of their containing buffer, so the IPv4 header, starting after the 14-byte Ethernet header, would be aligned on a 4-byte boundary. Today, however, the architectures for which this sort of alignment mattered have mostly faded away; alignment is a non-issue for ARM and Intel x86 processors.

7.2 Interfaces

IP addresses (both IPv4 and IPv6) are, strictly speaking, assigned not to hosts or nodes, but to **interfaces**. In the most common case, where each node has a single LAN interface, this is a distinction without a difference. In a room full of workstations each with a single Ethernet interface `eth0` (or perhaps `Ethernet adapter Local Area Connection`), we might as well view the IP address assigned to the interface as assigned to the workstation itself.

Each of those workstations, however, likely also has a **loopback** interface (at least conceptually), providing a way to deliver IP packets to other processes on the same machine. On many systems, the name "localhost" resolves to the IPv4 loopback address 127.0.0.1 (the IPv6 address ::1 is also used); see *7.3 Special Addresses*. Delivering packets to the loopback interface is simply a form of interprocess communication; a functionally similar alternative is named pipes.

Loopback delivery avoids the need to use the LAN at all, or even the need to *have* a LAN. For simple client/server testing, it is often convenient to have both client and server on the same machine, in which case the loopback interface is a convenient (and fast) standin for a "real" network interface. On unix-based machines the loopback interface represents a genuine logical interface, commonly named `lo`. On Windows systems the "interface" may not represent an actual operating-system entity, but this is of practical concern only to those interested in "sniffing" all loopback traffic; packets sent to the loopback address are still delivered as expected.

Workstations often have special other interfaces as well. Most recent versions of Microsoft Windows have a Teredo Tunneling pseudo-interface and an Automatic Tunneling pseudo-interface; these are both intended (when activated) to support IPv6 connectivity when the local ISP supports only IPv4. The Teredo protocol is documented in **RFC 4380**.

When VPN connections are created, as in *3.1 Virtual Private Networks*, each end of the logical connection typically terminates at a virtual interface (one of these is labeled `tun0` in the diagram of *3.1 Virtual Private Networks*). These virtual interfaces appear, to the systems involved, to be attached to a point-to-point link that leads to the other end.

When a computer hosts a virtual machine, there is almost always a virtual network to connect the host and virtual systems. The host will have a virtual interface to connect to the virtual network. The host may act as a NAT router for the virtual machine, "hiding" that virtual machine behind its own IP address, or it may act as an Ethernet switch, in which case the virtual machine will need an additional public IP address.

What's My IP Address?

This simple-seeming question is in fact not very easy to answer, if by "my IP address" one means the IP address assigned to the interface that connects directly to the Internet. One strategy is to find the address of the default router, and then iterate through all interfaces (*eg* with the Java `NetworkInterface` class) to find an IP address with a matching network prefix; a Python3 example of this approach appears in *18.5.1 Multicast Programming*. Unfortunately, finding the default router (to identify the primary interface) is hard to do in an OS-independent way, and even then this approach can fail if the Wi-Fi and Ethernet interfaces both are assigned IP addresses on the same network, but only one is actually connected.

A host with multiple "real" network interfaces is often said to be **multihomed** (though sometimes multihoming is intended to mean that the different interfaces are on different IP networks; that is, they have different IP prefixes). Many laptops, for example, have both an Ethernet interface and a Wi-Fi interface. Both of these can be used simultaneously, with different IP addresses assigned to each. On residential networks the two interfaces will often be on the same IP network (*eg* the same bridged Wi-Fi/Ethernet LAN); at more security-conscious sites the Ethernet and Wi-Fi interfaces are often on quite different IP networks (though see *7.9.5 ARP and multihomed hosts*).

Routers always have at least two interfaces on two separate IP networks. Generally this means a separate IP address for each interface, though some point-to-point interfaces can be used without being assigned any IP address (*7.12 Unnumbered Interfaces*).

Finally, it is usually possible to assign multiple IP addresses to a *single* interface. Sometimes this is done to allow two IP networks to share a single physical LAN; in this case the interface would be assigned one IP address for each IP network. Other times a single interface is assigned multiple IP addresses on the same IP network; this is often done so that one physical machine can act as a server (*eg* a web server) for multiple distinct IP addresses corresponding to multiple distinct domain names.

While it is important to be at least vaguely aware of all these special cases, we emphasize again that in most ordinary contexts each end-user workstation has one IP address that corresponds to a LAN connection.

7.3 Special Addresses

A few IPv4 addresses represent special cases.

While the standard IPv4 loopback address is 127.0.0.1, any IPv4 address beginning with 127 can serve as a loopback address. Logically they all represent the current host. Most hosts are configured to resolve the name "localhost" to 127.0.0.1. However, any loopback address – *eg* 127.255.37.59 – should work, *eg* with `ping`. For an example using 127.0.**1**.0, see *7.8 DNS*.

Private addresses are IPv4 addresses intended only for site internal use, *eg* either behind a NAT firewall or intended to have no Internet connectivity at all. If a packet shows up at any non-private router (*eg* at an ISP router), with a private IPv4 address as either source or destination address, the packet should be dropped. Three standard private-address blocks have been defined:

- 10.0.0.0/8
- 172.16.0.0/12

- 192.168.0.0/16

The last block is the one from which addresses are most commonly allocated by DHCP servers (*7.10.1 NAT, DHCP and the Small Office*) built into NAT routers.

Broadcast addresses are a special form of IPv4 address intended to be used in conjunction with LAN-layer broadcast. The most common forms are "broadcast to this network", consisting of all 1-bits, and "broadcast to network D", consisting of D's network-address bits followed by all 1-bits for the host bits. If you try to send a packet to the broadcast address of a remote network D, the odds are that some router involved will refuse to forward it, and the odds are even higher that, once the packet arrives at a router actually on network D, that router will refuse to broadcast it. Even addressing a broadcast to ones own network will fail if the underlying LAN does not support LAN-level broadcast (*eg* ATM).

The highly influential early Unix implementation Berkeley 4.2 BSD used 0-bits for the broadcast bits, instead of 1's. As a result, to this day host bits cannot be all 1-bits or all 0-bits in order to avoid confusion with the IPv4 broadcast address. One consequence of this is that a Class C network has 254 usable host addresses, not 256.

7.3.1 Multicast addresses

Finally, **IPv4 multicast addresses** remain as the last remnant of the Class A/B/C strategy: multicast addresses are Class D, with first byte beginning 1110 (meaning that the first byte is, in decimal, 224-239). Multicasting means delivering to a specified *set* of addresses, preferably by some mechanism more efficient than sending to each address individually. A reasonable goal of multicast would be that no more than one copy of the multicast packet traverses any given link.

Support for IPv4 multicast requires considerable participation by the backbone routers involved. For example, if hosts A, B and C each connect to different interfaces of router R1, and A wishes to send a multicast packet to B and C, then it is up to R1 to receive the packet, figure out that B and C are the intended recipients, and forward the packet *twice*, once for B's interface and once for C's. R1 must also keep track of what hosts have joined the **multicast group** and what hosts have left. Due to this degree of router participation, backbone router support for multicasting has not been entirely forthcoming. A discussion of IPv4 multicasting appears in *20 Quality of Service*.

7.4 Fragmentation

If you are trying to interconnect two LANs (as IP does), what else might be needed besides Routing and Addressing? IPv4 (and IPv6) explicitly assumes all packets are composed on 8-bit bytes (something not universally true in the early days of IP; to this day the RFCs refer to "octets" to emphasize this requirement). IP also defines bit-order within a byte, and it is left to the networking hardware to translate properly. Neither byte size nor bit order, therefore, can interfere with packet forwarding.

There is one more feature IPv4 must provide, however, if the goal is universal connectivity: it must accommodate networks for which the maximum packet size, or **Maximum Transfer Unit**, MTU, is smaller than the packet that needs forwarding. Otherwise, if we were using IPv4 to join Token Ring (MTU = 4KB, at least originally) to Ethernet (MTU = 1500B), the token-ring packets might be too large to deliver to the Ethernet side, or to traverse an Ethernet backbone *en route* to another Token Ring. (Token Ring, in its day, did commonly offer a configuration option to allow Ethernet interoperability.)

So, IPv4 must support fragmentation, and thus also reassembly. There are two potential strategies here: **per-link** fragmentation and reassembly, where the reassembly is done at the opposite end of the link (as in ATM), and **path** fragmentation and reassembly, where reassembly is done at the far end of the path. The latter approach is what is taken by IPv4, partly because intermediate routers are too busy to do reassembly (this is as true today as it was in 1981 when RFC 791 was published), partly because there is no absolute guarantee that all fragments will go to the same next-hop router, and partly because IPv4 fragmentation has always been seen as the strategy of last resort.

An IPv4 sender is supposed to use a different value for the **IDENT** field for different packets, at least up until the field wraps around. When an IPv4 datagram is fragmented, the fragments keep the same IDENT field, so this field in effect indicates which fragments belong to the same packet.

After fragmentation, the **Fragment Offset** field marks the start position of the data portion of this fragment within the data portion of the original IPv4 packet. Note that the start position can be a number up to 2^{16}, the maximum IPv4 packet length, but the FragOffset field has only 13 bits. This is handled by requiring the data portions of fragments to have sizes a multiple of 8 (three bits), and left-shifting the FragOffset value by 3 bits before using it.

As an example, consider the following network, where MTUs are excluding the LAN header:

Suppose A addresses a packet of 1500 bytes to B, and sends it via the LAN to the first router R1. The packet contains 20 bytes of IPv4 header and 1480 of data.

R1 fragments the original packet into two packets of sizes 20+976 = 996 and 20+504=544. Having 980 bytes of payload in the first fragment would fit, but violates the rule that the sizes of the data portions be divisible by 8. The first fragment packet has FragOffset = 0; the second has FragOffset = 976.

R2 *re*fragments the first fragment into three packets as follows:

- first: size = 20+376=396, FragOffset = 0
- second: size = 20+376=396, FragOffset = 376
- third: size = 20+224 = 244 (note 376+376+224=976), FragOffset = 752.

R2 refragments the second fragment into two:

- first: size = 20+376 = 396, FragOffset = 976+0 = 976
- second: size = 20+128 = 148, FragOffset = 976+376=1352

R3 then sends the fragments on to B, without reassembly.

Note that it would have been slightly more efficient to have fragmented into four fragments of sizes 376, 376, 376, and 352 in the beginning. Note also that the packet format is designed to handle fragments of different sizes easily. The algorithm is based on multiple fragmentation with reassembly only at the final destination.

Each fragment has its IPv4-header Total Length field set to the length of that fragment.

We have not yet discussed the three flag bits. The first bit is reserved, and must be 0. The second bit is the **Don't Fragment**, or DF, bit. If it is set to 1 by the sender then a router must *not* fragment the packet and must drop it instead; see *12.13 Path MTU Discovery* for an application of this. The third bit is set to 1 for all fragments *except* the final one (this bit is thus set to 0 if no fragmentation has occurred). The third bit tells the receiver where the fragments stop.

The receiver must take the arriving fragments and **reassemble** them into a whole packet. The fragments may not arrive in order – unlike in ATM networks – and may have unrelated packets interspersed. The reassembler must identify when different arriving packets are fragments of the same original, and must figure out how to reassemble the fragments in the correct order; both these problems were essentially trivial for ATM.

Fragments are considered to belong to the same packet if they have the same IDENT field and also the same source and destination addresses and same protocol.

As all fragment sizes are a multiple of 8 bytes, the receiver can keep track of whether all fragments have been received with a bitmap in which each bit represents one 8-byte fragment chunk. A 1 KB packet could have up to 128 such chunks; the bitmap would thus be 16 bytes.

If a fragment arrives that is part of a new (and fragmented) packet, a buffer is allocated. While the receiver cannot know the final size of the buffer, it can usually make a reasonable guess. Because of the FragOffset field, the fragment can then be stored in the buffer in the appropriate position. A new bitmap is also allocated, and a **reassembly timer** is started.

As subsequent fragments arrive, not necessarily in order, they too can be placed in the proper buffer in the proper position, and the appropriate bits in the bitmap are set to 1.

If the bitmap shows that all fragments have arrived, the packet is sent on up as a completed IPv4 packet. If, on the other hand, the reassembly timer expires, then all the pieces received so far are discarded.

TCP connections usually engage in **Path MTU Discovery**, and figure out the largest packet size they can send that will *not* entail fragmentation (*12.13 Path MTU Discovery*). But it is not unusual, for example, for UDP protocols to use fragmentation, especially over the short haul. In the Network File System (NFS) protocol, for example, UDP is used to carry 8KB disk blocks. These are often sent as a single 8+ KB IPv4 packet, fragmented over Ethernet to five full packets and a fraction. Fragmentation works reasonably well here because most of the time the packets do not leave the Ethernet they started on. Note that this is an example of fragmentation done by the *sender*, not by an intermediate router.

Finally, any given IP link may provide its own link-layer fragmentation and reassembly; we saw in *3.5.1 ATM Segmentation and Reassembly* that ATM does just this. Such link-layer mechanisms are, however, generally invisible to the IP layer.

7.5 The Classless IP Delivery Algorithm

Recall from Chapter 1 that any IPv4 address can be divided into a net portion IP_{net} and a host portion IP_{host}; the division point was determined by whether the IPv4 address was a Class A, a Class B, or a Class C. We also indicated in Chapter 1 that the division point was not always so clear-cut; we now present the delivery algorithm, for both hosts and routers, that does *not* assume a globally predeclared division point of the input IPv4 address into net and host portions. We will, for the time being, punt on the question of forwarding-table

lookup and assume there is a `lookup()` method available that, when given a destination address, returns the next_hop neighbor.

Instead of class-based divisions, we will assume that each of the IPv4 addresses assigned to a node's interfaces is configured with an associated length of the network prefix; following the slash notation of *1.10 IP - Internet Protocol*, if B is an address and the prefix length is $k = k_B$ then the prefix itself is B/k. As usual, an ordinary host may have only one IP interface, while a router will always have multiple interfaces.

Let D be the given IPv4 destination address; we want to decide if D is **local** or **nonlocal**. The host or router involved may have multiple IP interfaces, but for each interface the length of the network portion of the address will be known. For each network address B/k assigned to one of the host's interfaces, we compare the first k bits of B and D; that is, we ask if D **matches** B/k.

- If one of these comparisons yields a match, delivery is **local**; the host delivers the packet to its final destination via the LAN connected to the corresponding interface. This means looking up the LAN address of the destination, if applicable, and sending the packet to that destination via the interface.
- If there is no match, delivery is **nonlocal**, and the host passes D to the `lookup()` routine of the forwarding table and sends to the associated next_hop (which must represent a physically connected neighbor). It is now up to `lookup()` routine to make any necessary determinations as to how D might be split into D_{net} and D_{host}; the split *cannot* be made outside of `lookup()`.

The forwarding table is, abstractly, a set of network addresses – now also with lengths – each of the form B/k, with an associated next_hop destination for each. The `lookup()` routine will, in principle, compare D with each table entry B/k, looking for a match (that is, equality of the first $k = k_B$ bits). As with the local-delivery interfaces check above, the net/host division point (that is, k) will come from the table entry; it will not be inferred from D or from any other information borne by the packet. There is, in fact, no place in the IPv4 header to store a net/host division point, and furthermore different routers along the path may use different values of k with the same destination address D. Routers receive the prefix length /k for a destination B/k as part of the process by which they receive ⟨destination,next_hop⟩ pairs; see *9 Routing-Update Algorithms*.

In *10 Large-Scale IP Routing* we will see that in some cases multiple matches in the forwarding table may exist, *eg* 147.0.0.0/8 and 147.126.0.0/16. The **longest-match** rule will be introduced for such cases to pick the best match.

Here is a simple example for a router with immediate neighbors A-E:

destination	next_hop
10.3.0.0/16	A
10.4.1.0/24	B
10.4.2.0/24	C
10.4.3.0/24	D
10.3.37.0/24	E

The IPv4 addresses 10.3.67.101 and 10.3.59.131 both route to A. The addresses 10.4.1.101, 10.4.2.157 and 10.4.3.233 route to B, C and D respectively. Finally, 10.3.37.103 matches both A and E, but the E match is longer so the packet is routed that way.

The forwarding table may also contain a **default** entry for the next_hop, which it may return in cases when the destination D does not match any known network. We take the view here that returning such a default entry is a valid result of the routing-table `lookup()` operation, rather than a third option to the algorithm above; one approach is for the default entry to be the next_hop corresponding to the destination 0.0.0.0/0,

which does indeed match everything (use of this would definitely require the above longest-match rule, though).

Default routes are hugely important in keeping leaf forwarding tables small. Even backbone routers sometimes expend considerable effort to keep the network address prefixes in their forwarding tables as short as possible, through consolidation.

At a site with a single ISP and with no Internet customers (that is, which is not itself an ISP for others), the top-level forwarding table usually has a single external route: its default route to its ISP. If a site has more than one ISP, however, the top-level forwarding table can expand in a hurry. For example, Internet2 is a consortium of research sites with very-high-bandwidth internal interconnections, acting as a sort of "parallel Internet". Before Internet2, Loyola's top-level forwarding table had the usual single external default route. After Internet2, we in effect had a second ISP and had to divide traffic between the commercial ISP and the Internet2 ISP. The default route still pointed to the commercial ISP, but the top-level forwarding table now had to have an entry for every individual Internet2 site, so that traffic to any of these sites would be forwarded via the Internet2 ISP. See exercise 5.0.

Routers may also be configured to allow passing quality-of-service information to the `lookup()` method, as mentioned in Chapter 1, to support different routing paths for different kinds of traffic (*eg* bulk file-transfer versus real-time).

For a modest exception to the local-delivery rule described here, see below in *7.12 Unnumbered Interfaces*.

7.6 IPv4 Subnets

Subnets were the first step away from Class A/B/C routing: a large network (*eg* a class A or B) could be divided into smaller IPv4 networks called subnets. Consider, for example, a typical Class B network such as Loyola University's (originally 147.126.0.0/16); the underlying assumption is that any packet can be delivered via the underlying LAN to any internal host. This would require a rather large LAN, and would require that a single physical LAN be used throughout the site. What if our site has more than one physical LAN? Or is really too big for one physical LAN? It did not take long for the IP world to run into this problem.

Subnets were first proposed in **RFC 917**, and became official with **RFC 950**.

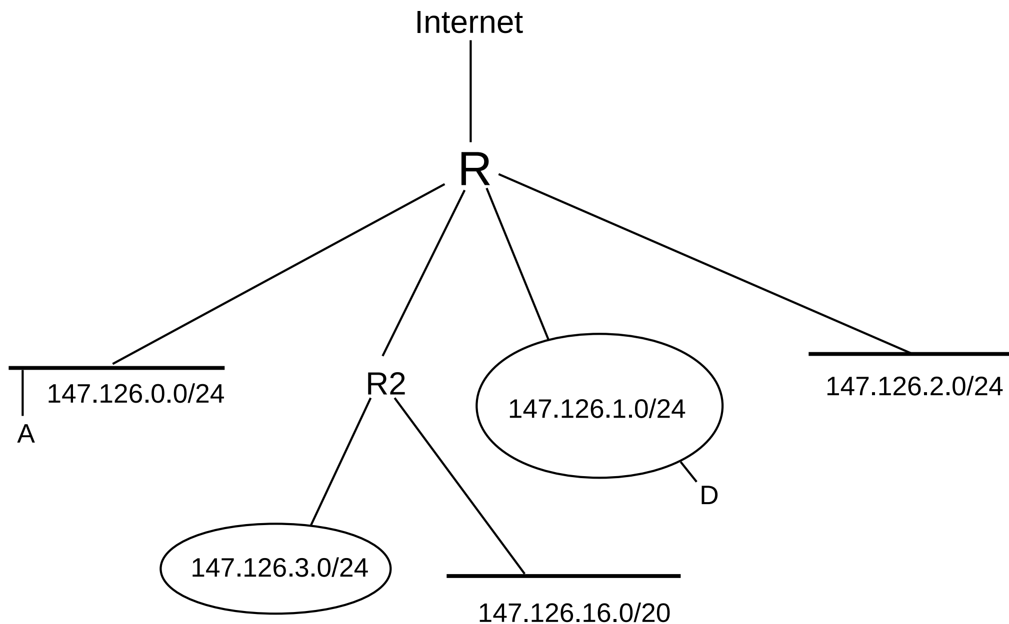
Getting a separate IPv4 network prefix for each subnet is bad for routers: the backbone forwarding tables now must have an entry for every subnet instead of just for every site. What is needed is a way for a site to appear to the outside world as a single IP network, but for further IP-layer routing to be supported *inside* the site. This is what subnets accomplish.

Subnets introduce **hierarchical routing**: first we route to the primary network, then inside that site we route to the subnet, and finally the last hop delivers to the host.

Routing with subnets involves in effect moving the IP_{net} division line rightward. (Later, when we consider CIDR, we will see the complementary case of moving the division line to the left.) For now, observe that moving the line rightward within a site does not affect the outside world at all; outside routers are not even aware of site-internal subnetting.

In the following diagram, the outside world directs traffic addressed to 147.126.0.0/16 to the router R. Internally, however, the site is divided into subnets. The idea is that traffic from 147.126.1.0/24 to 147.126.2.0/24 is routed, not switched; the two LANs involved may not even be compatible. Most of the subnets shown are

of size /24, meaning that the third byte of the IPv4 address has become part of the network portion of the subnet's address; one /20 subnet is also shown. RFC 950 would have disallowed the subnet with third byte 0, but having 0 for the subnet bits generally does work.



What we want is for the internal routing to be based on the extended network prefixes shown, while externally continuing to use only the single routing entry for 147.126.0.0/16.

To implement subnets, we divide the site's IPv4 network into some combination of physical LANs – the subnets –, and assign each a **subnet address**: an IPv4 network address which has the *site's* IPv4 network address as prefix. To put this more concretely, suppose the site's IPv4 network address is A, and consists of n network bits (so the site address may be written with the slash notation as A/n); in the diagram above, A/n = 147.126.0.0/16. A subnet address is an IPv4 network address B/k such that:

- The address B/k is within the site: the first n bits of B are the same as A/n's
- B/k extends A/n: k⩾n

An example B/k in the diagram above is 147.126.1.0/24. (There is a slight simplification here in that subnet addresses do not absolutely *have* to be prefixes; see below.)

We now have to figure out how packets will be routed to the correct subnet. For incoming packets we could set up some proprietary protocol at the entry router to handle this. However, the more complicated situation is all those existing internal hosts that, under the class A/B/C strategy, would still believe they can deliver via the LAN to any site host, when in fact they can now only do that for hosts on their own subnet. We need a more general solution.

We proceed as follows. For each subnet address B/k, we create a **subnet mask** for B consisting of k 1-bits followed by enough 0-bits to make a total of 32. We then make sure that every host and router in the site knows the subnet mask for every one of its *interfaces*. Hosts usually find their subnet mask the same way they find their IP address (by static configuration if necessary, but more likely via DHCP, below).

Hosts and routers now apply the IP delivery algorithm of the previous section, with the proviso that, if a

subnet mask for an interface is present, then the subnet mask is used to determine the number of address bits rather than the Class A/B/C mechanism. That is, we determine whether a packet addressed to destination D is deliverable locally via an interface with subnet address B/k and corresponding mask M by comparing D&M with B&M, where **&** represents bitwise AND; if the two match, the packet is local. This will generally involve a match of *more* bits than if we used the Class A/B/C strategy to determine the network portion of addresses D and B.

As stated previously, given an address D with no other context, we will *not* be able to determine the network/host division point in general (*eg* for outbound packets). However, that division point is not in fact what we need. All that *is* needed is a way to tell if a given destination host address D belongs to the current subnet, say B; that is, we need to compare the first k bits of D and B where k is the (known) length of B.

In the diagram above, the subnet mask for the /24 subnets would be 255.255.255.0; bitwise ANDing any IPv4 address with the mask is the same as extracting the first 24 bits of the IPv4 address, that is, the subnet portion. The mask for the /20 subnet would be 255.255.240.0 (240 in binary is 1111 0000).

In the diagram above none of the subnets overlaps or conflicts: the subnets 147.126.0.0/24 and 147.126.1.0/24 are disjoint. It takes a little more effort to realize that 147.126.16.0/20 does not overlap with the others, but note that an IPv4 address matches this network prefix only if the first four bits of the third byte are 0001, so the third byte itself ranges from decimal 32 to decimal 63 = binary 0001 1111.

Note also that if host A = 147.126.0.1 wishes to send to destination D = 147.126.1.1, and A is *not* subnet-aware, then delivery will fail: A will infer that the interface is a Class B, and therefore compare the first two bytes of A and D, and, finding a match, will attempt direct LAN delivery. But direct delivery is now likely impossible, as the subnets are not joined by a switch. Only with the subnet mask will A realize that its network is 147.126.0.0/**24** while D's is 147.126.1.0/24 and that these are not the same. A *would* still be able to send packets to its own subnet. In fact A would still be able to send packets to the outside world: it would realize that the destination in that case does not match 147.126.0.0/16 and will thus forward to its router. Hosts on other subnets would be the only unreachable ones.

Properly, the subnet address is the entire prefix, *eg* 147.126.65.0/24. However, it is often convenient to identify the subnet address with just those bits that represent the extension of the site IPv4-network address; we might thus say casually that the subnet address here is 65.

The class-based IP-address strategy allowed any host anywhere on the Internet to properly separate any address into its net and host portions. With subnets, this division point is now allowed to vary; for example, the address 147.126.65.48 divides into 147.126 | 65.48 outside of Loyola, but into 147.126.65 | 48 inside. This means that the net-host division is no longer an absolute property of addresses, but rather something that depends on where the packet is on its journey.

Technically, we also need the requirement that given any two subnet addresses of different, disjoint subnets, neither is a proper prefix of the other. This guarantees that if A is an IP address and B is a subnet address with mask M (so B = B&M), then A&M = B implies A does not match any other subnet. Regardless of the net/host division rules, we cannot possibly allow subnet 147.126.16.0/20 to represent one LAN while 147.126.16.0/24 represents another; the second subnet address block is a subset of the first. (We *can*, and sometimes do, allow the first LAN to correspond to everything in 147.126.16.0/20 that is not also in 147.126.16.0/24; this is the longest-match rule.)

The strategy above is actually a slight simplification of what the subnet mechanism actually allows: subnet address bits do not in fact have to be contiguous, and masks do not have to be a series of 1-bits followed by 0-bits. The mask can be *any* bit-mask; the subnet address bits are by definition those where there is a 1 in the mask bits. For example, we could at a Class-B site use the *fourth* byte as the subnet address, and the *third*

byte as the host address. The subnet mask would then be 255.255.0.255. While this generality was once sometimes useful in dealing with "legacy" IPv4 addresses that could not easily be changed, life is simpler when the subnet bits precede the host bits.

7.6.1 Subnet Example

As an example of having different subnet masks on different interfaces, let us consider the division of a class-C network into subnets of size 70, 40, 25, and 20. The subnet addresses will of necessity have different lengths, as there is not room for four subnets each able to hold 70 hosts.

- A: size 70
- B: size 40
- C: size 25
- D: size 20

Because of the different subnet-address lengths, division of a local IPv4 address LA into net versus host on subnets cannot be done in isolation, without looking at the host bits. However, that division is not in fact what we need. All that is needed is a way to tell if the local address LA belongs to a given subnet, say B; that is, we need to compare the first n bits of LA and B, where n is the length of B's subnet mask. We do this by comparing LA&M to B&M, where M is the mask corresponding to n. LA&M is not necessarily the same as LA_{net}, if LA actually belongs to one of the other subnets. However, if LA&M = B&M, then LA must belong subnet B, in which case LA&M is in fact LA_{net}.

We will assume that the site's IPv4 network address is 200.0.0.0/24. The first three bytes of each subnet address must match 200.0.0. Only some of the bits of the fourth byte will be part of the subnet address, so we will switch to binary for the last byte, and use both the /n notation (for total number of subnet bits) and also add a vertical bar | to mark the separation between subnet and host.

Example: 200.0.0.10 | 00 0000 / 26

Note that this means that the 0-bit following the 1-bit in the fourth byte is "significant" in that for a subnet to match, it must match this 0-bit exactly. The remaining six 0-bits are part of the host portion.

To allocate our four subnet addresses above, we start by figuring out just how many host bits we need in each subnet. Subnet sizes are always powers of 2, so we round up the subnets to the appropriate size. For subnet A, this means we need 7 host bits to accommodate 2^7 = 128 hosts, and so we have a single bit in the fourth byte to devote to the subnet address. Similarly, for B we will need 6 host bits and will have 2 subnet bits, and for C and D we will need 5 host bits each and will have 8-5=3 subnet bits.

We now start choosing non-overlapping subnet addresses. We have one bit in the fourth byte to choose for A's subnet; rather arbitrarily, let us choose this bit to be 1. This means that *every other subnet address* must have a 0 in the first bit position of the fourth byte, or we would have ambiguity.

Now for B's subnet address. We have two bits to work with, and the first bit must be 0. Let us choose the second bit to be 0 as well. If the fourth byte begins 00, the packet is part of subnet B, and the subnet addresses for C and D must therefore *not* begin 00.

Finally, we choose subnet addresses for C and D to be 010 and 011, respectively. We thus have

subnet	size	address bits in fourth byte	host bits in 4th byte	decimal range
A	128	1	7	128-255
B	64	00	6	0-63
C	32	010	5	64-95
D	32	011	5	96-127

As desired, none of the subnet addresses in the third column is a prefix of any other subnet address.

The end result of all of this is that routing is now **hierarchical**: we route on the site IP address to get to a site, and then route on the subnet address within the site.

7.6.2 Links between subnets

Suppose the Loyola CS department subnet (147.126.65.0/24) and a department at some other site, we will say 147.100.100.0/24, install a private link. How does this affect routing?

Each department router would add an entry for the other subnet, routing along the private link. Traffic addressed to the other subnet would take the private link. All other traffic would go to the default router. Traffic from the remote department to 147.126.6**4**.0/24 would take the long route, and Loyola traffic to 147.100.10**1**.0/24 would take the long route.

Subnet anecdote

A long time ago I was responsible for two hosts, abel and borel. One day I was informed that machines in computer lab 1 at the other end of campus could not reach borel, though they could reach abel. Machines in lab 2, *adjacent* to lab 1, however, could reach both borel and abel just fine. What was the difference?

It turned out that borel had a bad (/16 instead of /24) subnet mask, and so it was attempting local delivery to the labs. This *should* have meant it could reach neither of the labs, as both labs were on a different subnet from my machines; I was still perplexed. After considerably more investigation, it turned out that between abel/borel and the lab building was a **bridge-router**: a hybrid device that properly routed subnet traffic, but which also forwarded Ethernet packets directly, the latter feature apparently for the purpose of backwards compatibility. Lab 2 was connected directly to the bridge-router and thus appeared to be on the same LAN as borel, despite the apparently different subnet; lab 1 was connected to its own router R1 which in turn connected to the bridge-router. Lab 1 was thus, at the LAN level, isolated from abel and borel.

Moral 1: Switching and routing are both great ideas, alone. But switching mixed with routing is not.

Moral 2: Test thoroughly! The reason the problem wasn't noticed earlier was that previously borel communicated only with other hosts on its own subnet and with hosts outside the university entirely. Both of these worked with the bad subnet mask; it was different-subnet local hosts that were the problem.

How would nearby subnets at either endpoint decide whether to use the private link? Classical link-state or distance-vector theory (*9 Routing-Update Algorithms*) requires that they be able to compare the private-link route with the going-around-the-long-way route. But this requires a global picture of relative routing costs, which, as we shall see, almost certainly does not exist. The two departments are in different routing domains; if neighboring subnets at either end want to use the private link, then manual configuration is likely the only option.

7.6.3 Subnets versus Switching

A frequent network design question is whether to have many small subnets or to instead have just a few (or even only one) larger subnet. With multiple small subnets, IP **routing** would be used to interconnect them; the use of larger subnets would replace much of that routing with LAN-layer communication, likely Ethernet **switching**. Debates on this route-versus-switch question have gone back and forth in the networking community, with one aphorism summarizing a common view:

> Switch when you can, route when you must

This aphorism reflects the idea that switching is faster, cheaper and easier to configure, and that subnet boundaries should be drawn only where "necessary".

Ethernet switching equipment is indeed generally cheaper than routing equipment, for the same overall level of features and reliability. And traditional switching requires relatively little configuration, while to implement subnets not only must the subnets be created by hand but one must also set up and configure the routing-update protocols. However, the price difference between switching and routing is not always significant in the big picture, and the configuration involved is often straightforward.

Somewhere along the way, however, switching has acquired a reputation – often deserved – for being *faster* than routing. It is true that routers have more to do than switches: they must decrement TTL, update the header checksum, and attach a new LAN header. But these things are relatively minor: a larger reason many routers are slower than switches may simply be that they are inevitably *asked to serve as firewalls*. This means "deep inspection" of every packet, *eg* comparing every packet to each of a large number of firewall rules. The firewall may also be asked to keep track of connection state. All this drives down the forwarding rate, as measured in packets-per-second.

Traditional switching scales remarkably well, but it does have limitations. First, broadcast packets must be forwarded throughout a switched network; they do not, however, pass to different subnets. Second, LAN networks do not like redundant links (that is, loops); while one can rely on the spanning-tree algorithm to eliminate these, that algorithm too becomes less efficient at larger scales.

The rise of software-defined networking (*2.7 Software-Defined Networking*) has blurred the distinction between routing and switching. The term "Layer 3 switch" is sometimes used to describe routers that in effect do not support all the usual firewall bells and whistles. These are often SDN Ethernet switches (*2.7 Software-Defined Networking*) that are making forwarding decisions based on the contents of the IP header. Such streamlined switch/routers may also be able to do most of the hard work in specialized hardware, another source of speedup.

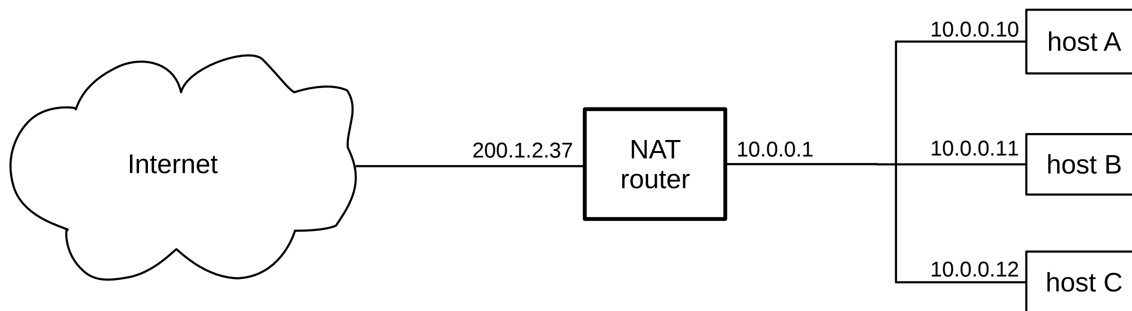
But SDN can do much more than IP-layer forwarding, by taking advantage of site-specific layout information. One application, of a switch hierarchy for traffic entering a datacenter, appears in *2.7.1 OpenFlow Switches*. Other SDN applications include enabling Ethernet topologies with loops, offloading large-volume flows to alternative paths, and implementing policy-based routing as in *9.6 Routing on Other Attributes*. Some SDN solutions involve site-specific programming, but others work more-or-less out of the box. Locations with switch-versus-route issues are likely to turn increasingly to SDN in the future.

7.7 Network Address Translation

What do you do if your ISP assigns to you a single IPv4 address and you have two computers? The solution is Network Address Translation, or **NAT**. NAT's ability to "multiplex" an arbitrarily large number of indi-

vidual hosts behind a single IPv4 address (or small number of addresses) makes it an important tool in the conservation of IPv4 addresses. It also, however, enables an important form of firewall-based security. It is documented in **RFC 3022**, where this is called NAPT, or Network Address **Port** Translation.

The basic idea is that, instead of assigning each host at a site a publicly visible IPv4 address, just one such address is assigned to a special device known as a NAT router. A NAT router sold for residential or small-office use is commonly simply called a "router", or (somewhat more precisely) a "residential gateway". One side of the NAT router connects to the Internet; the other connects to the site's internal network. Hosts on the internal network are assigned private IP addresses (*7.3 Special Addresses*), typically of the form or 192.168.x.y or 10.x.y.z. Connections to internal hosts that originate in the outside world are banned. When an internal machine wants to connect to the outside, the NAT router intercepts the connection, and forwards the connection's packets after rewriting the source address to make it appear they came from the NAT router's own IP address, shown below as 200.1.2.37.



The remote machine responds, sending its responses to the NAT router's public IPv4 address. The NAT router remembers the connection, having stored the connection information in a special forwarding table, and forwards the data to the correct internal host, rewriting the destination-address field of the incoming packets.

The NAT forwarding table also includes port numbers. That way, if two internal hosts attempt to connect to the same external host, the NAT router can tell which packets belong to which. For example, suppose internal hosts A and B each connect from port 3000 to port 80 on external hosts S and T, respectively. Here is what the NAT forwarding table might look like. No columns for the NAT router's own IPv4 addresses are needed; we shall let NR denote the router's external address.

remote host	remote port	outside source port	inside host	inside port
S	80	3000	A	3000
T	80	3000	B	3000

A packet to S from ⟨A,3000⟩ would be rewritten so that the source was ⟨NR,3000⟩. A packet from ⟨S,80⟩ addressed to ⟨NR,3000⟩ would be rewritten and forwarded to ⟨A,3000⟩. Similarly, a packet from ⟨T,80⟩ addressed to ⟨NR,3000⟩ would be rewritten and forwarded to ⟨B,3000⟩; the NAT table takes into account the source host and port as well as the destination.

Sometimes it is necessary for the NAT router to rewrite the internal-side port number as well; this happens if two internal hosts want to connect, each from the *same* port, to the same external host and port. For example, suppose B now opens a connection to ⟨S,80⟩, also from inside port 3000. This time the NAT router must remap the port number, because that is the only way to distinguish between packets from ⟨S,80⟩ back to A and to B. With B's second connection's internal port remapped from 3000 to 3001, the new table is

remote host	remote port	outside source port	inside host	inside port
S	80	3000	A	3000
T	80	3000	B	3000
S	80	3001	B	3000

The NAT router does not create TCP connections between itself and the external hosts; it simply forwards packets (with rewriting). The connection endpoints are still the external hosts S and T and the internal hosts A and B. However, NR might very well *monitor* the TCP connections to know when they have closed, and so can be removed from the table. For UDP connections, NAT routers typically remove the forwarding entry after some period of inactivity; see *11 UDP Transport*, exercise 14.0.

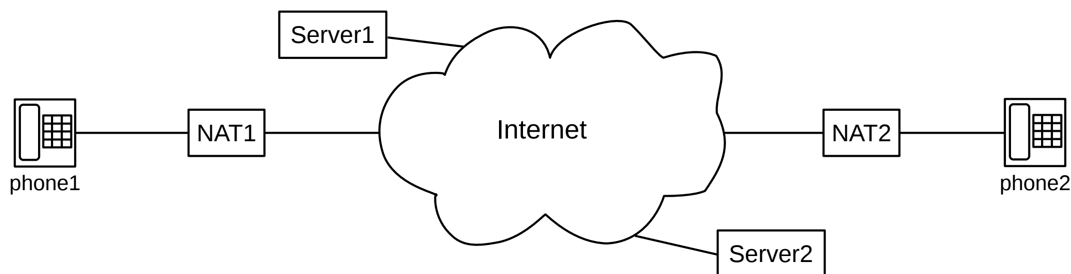
NAT still works for *some* traffic without port numbers, such as network pings, though the above table is then not quite the whole story. See *7.11 Internet Control Message Protocol*.

Done properly, NAT improves the security of a site, by making it impossible for an external host to probe or connect to any of the internal hosts. While this firewall feature is of great importance, essentially the same effect can be achieved without address translation, and *with* public IPv4 addresses for all internal hosts, by having the router refuse to forward incoming packets that are not part of existing connections. The router still needs to maintain a table like the NAT table above, in order to recognize such packets. The address translation itself, in other words, is not the source of the firewall security. That said, it is hard for a NAT router to "fail open"; *ie* to fail in a way that lets outside connections in. It is much easier for a non-NAT firewall to fail open.

For the common residential form of NAT router, see *7.10.1 NAT, DHCP and the Small Office*.

7.7.1 NAT Problems

NAT router's refusal to allow inbound connections is a source of occasional frustration. We illustrate some of these frustrations here, using Voice-over-IP (VoIP) and the call-setup protocol SIP (**RFC 3261**). The basic strategy is that each phone is associated with a remote **phone server**. These phone servers, because they have to be able to accept incoming connections from anywhere, must not be behind NAT routers. The phones themselves, however, usually will be:



For phone1 to call phone2, phone1 first contacts Server1, which then contacts Server2. So far, all is well. The final step is for Server2 to contact phone2, which, however, cannot be done normally as NAT2 allows no inbound connections.

One common solution is for phone2 to maintain a persistent connection to Server2 (and ditto for phone1 and Server1). By having these persistent phone-to-server connections, we can arrange for the phone to ring on incoming calls.

As a second issue, somewhat particular to the SIP protocol, is that it is common for server and phone to prefer to use UDP port 5060 at *both* ends. For a single internal phone, it is likely that port 5060 will pass through without remapping, so the phone will appear to be connecting from the desired port 5060. However, if there are two phones inside (not shown above), one of them will appear to be connecting to the server *from* an alternative port. The solution here is to have the server tolerate such port remapping.

VoIP systems run into a much more serious problem with NAT, however. Once the call between phone1 and phone2 is set up, the servers would prefer to step out of the loop, and have the phones exchange voice packets directly. The SIP protocol was designed to handle this by having each phone report to its respective server the UDP socket (⟨IP address,port⟩ pair) it intends to use for the voice exchange; the servers then report these phone sockets to each other, and from there to the opposite phones. This socket information is rendered incorrect by NAT, however, certainly the IP address and quite likely the port as well. If only one of the phones is behind a NAT firewall, it can initiate the voice connection to the other phone, but the other phone will see the voice packets arriving from a different socket than promised and will likely not recognize them as part of the call. If both phones are behind NAT firewalls, they will not be able to connect directly to one another at all. The common solution is for the VoIP server of a phone behind a NAT firewall to remain in the communications path, forwarding packets to its hidden partner. This works, but represents an unwanted server workload.

If a site wants to make it possible to allow external connections to hosts behind a NAT router or other firewall, one option is **tunneling**. This is the creation of a "virtual LAN link" that runs on top of a TCP connection between the end user and one of the site's servers; the end user can thus appear to be on one of the organization's internal LANs; see *3.1 Virtual Private Networks*. Another option is to "open up" a specific port: in essence, a static NAT-table entry is made connecting a specific port on the NAT router to a specific internal host and port (usually the same port). For example, all UDP packets to port 5060 on the NAT router might be forwarded to port 5060 on internal host A, even in the absence of any prior packet exchange. Gamers creating peer-to-peer game connections must also usually engage in some port-opening configuration. The Port Control Protocol (**RFC 6887**) is sometimes used for this.

NAT routers work very well when the communications model is of client-side TCP connections, originating from the inside and with public outside servers as destination. The NAT model works less well for peer-to-peer networking, as with the gamers above, where two computers, each behind a different NAT router, wish to establish a connection. Most NAT routers provide at least limited support for "opening" access to a given internal ⟨host,port⟩ socket, by creating a semi-permanent forwarding-table entry. See also *12.24 Exercises*, exercise 2.5.

NAT routers also often have trouble with UDP protocols, due to the tendency for such protocols to have the public server reply from a *different* port than the one originally contacted. For example, if host A behind a NAT router attempts to use TFTP (*11.2 Trivial File Transport Protocol, TFTP*), and sends a packet to port 69 of public server C, then C is likely to reply from some *new* port, say 3000, and this reply is likely to be dropped by the NAT router as there will be no entry there yet for traffic from ⟨C,3000⟩.

7.7.2 Middleboxes

Firewalls and NAT routers are sometimes classed as **middleboxes**: network devices that block, throttle or modify traffic beyond what is necessary for basic forwarding. Middleboxes play a very important role in network security, but they sometimes (as here with VoIP) break things. The word "middlebox" (versus "router" or "firewall") usually has a perjorative connotation; middleboxes have, in some circles, acquired a rather negative reputation.

NAT routers' interference with VoIP, above, is a direct consequence of their function: NAT handles connections from inside to outside quite well, but the NAT mechanism offers no support for connections from one inside to another inside. Sometimes, however, middleboxes block traffic when there is no technical reason to do so, simply because correct behavior has not been widely implemented. As an example, the SCTP protocol, *12.22.2 SCTP*, has seen very limited use despite some putative advantages over TCP, largely due to lack of NAT-router support. SCTP cannot be used by residential users because the middleboxes have not kept up.

A third category of middlebox-related problems is overzealous blocking in the name of security. SCTP runs into this problem as well, though not quite as universally: a few routers simply drop all SCTP packets because they represent an "unknown" – and therefore suspect – type of traffic. There is a place for this block-by-default approach. If a datacenter firewall blocks all inbound TCP traffic except to port 80 (the HTTP port), and if SCTP is not being used within the datacenter intentionally, it is hard to argue against blocking all inbound SCTP traffic. But if the frontline router for home or office users blocks all *outbound* SCTP traffic, then the users cannot use SCTP.

A consequence of overzealous blocking is that it becomes much harder to introduce new protocols. If a new protocol is blocked for even a small fraction of potential users, it is just not worth the effort. See also the discussion at *12.22.4 QUIC Revisited*; the design of QUIC includes several elements to mitigate middlebox problems.

For another example of overzealous blocking by middleboxes, with the added element of spoofed TCP RST packets, see the sidebar at *14.8.2 Explicit Congestion Notification (ECN)*.

7.8 DNS

The **Domain Name System**, DNS, is an essential companion protocol to IPv4; an overview can be found in **RFC 1034**. It is DNS that permits users the luxury of not needing to remember numeric IP addresses. Instead of 162.216.18.28, a user can simply enter intronetworks.cs.luc.edu, and DNS will take care of looking up the name and finding the corresponding address. DNS also makes it easy to move services from one server to another with a different IP address; as users will locate the service by DNS name and not by IP address, they do not need to be notified.

While a workstation can use TCP/IP without DNS, users have an almost impossible time finding anything, and so the core startup configuration of an Internet-connected workstation almost always includes the IP address of its DNS server (see *7.10 Dynamic Host Configuration Protocol (DHCP)* below for how startup configurations are often assigned).

Most DNS traffic today is over UDP, although a TCP option exists.

DNS is **distributed**, meaning that each domain is responsible for maintaining its own **DNS servers** to translate names to addresses. (DNS, in fact, is a classic example of a highly distributed database where each node maintains a relatively small amount of data.) It is **hierarchical** as well; for the DNS name `intronetworks.cs.luc.edu` the levels of the hierarchy are

- **edu**: the **top-level domain** (TLD) for educational institutions in the US
- **luc**: Loyola University Chicago
- **cs**: The Loyola Computer Science Department

- **intronetworks**: a hostname associated to a specific IP address

The hierarchy of DNS names (that is, the set of all names and prefixes of names) forms a tree, but it is not only leaf nodes that represent individual hosts. In the example above, luc.edu and cs.luc.edu happen to be valid hostnames as well.

Top-level domains are assigned by ICANN. The original top-level domains were the two-letter country-code domains (*eg* .us, .ca, .mx) and seven others: `.com`, `.net`, `.org`, `.int`, `.edu`, `.mil` and `.gov`. Now there are hundreds of non-country top-level domains.

The full tree of all DNS names and prefixes is divided administratively into **zones**: a zone is a subtree, minus any sub-subtrees that have been placed – by delegation – into their own zone. Each zone has its own root DNS name that is a prefix of every DNS name in the zone. For example, the `luc.edu` zone contains most of Loyola's DNS names, but `cs.luc.edu` has been spun off into its own zone. A zone cannot be the disjoint union of two subtrees.

Each zone has its own **authoritative nameservers** for the zone, which are charged with maintaining the records for that zone. Each zone must have at least two nameservers, for redundancy. IPv4 addresses are stored as so-called **A records**, for Address. Information about how to find sub-zones is stored as **NS records**. An authoritative nameserver need not be part of the organization that manages the zone, and a single server can be the authoritative nameserver for thousands of unrelated zones.

The **root nameservers** handle the zone that is the root of the DNS tree; that is, that is represented by the DNS name that is the empty string. The root nameservers contain only NS records, identifying the nameservers for all the immediate subzones. Each top-level domain is its own such subzone. The IP addresses of the root nameservers are widely distributed. Their DNS names (which are only of use if some DNS lookup mechanism is already present) are `a.root.servers.net` through `m.root-servers.net`. These names today correspond not to individual machines but to clusters of up to hundreds of servers.

We can now put together a first draft of a DNS lookup algorithm. To find the IP address of intronetworks.cs.luc.edu, a host first contacts a root nameserver (at a known address) to find the nameserver for the `edu` zone; this involves the retrieval of an NS record. The `edu` nameserver is then queried to find the nameserver for the `luc.edu` zone, which in turn supplies the NS record giving the address of the `cs.luc.edu` zone. This last has an A record for the actual host. (This example is carried out in detail below.)

DNS Policing

It is sometimes suggested that if a site is engaged in illegal activity or copyright infringement, such as thepiratebay.se, its domain name should be seized. The problem with this strategy is that it is straightforward for users to set up "nonstandard" nameservers (for example GNS) that continue to list the banned site.

This strategy has a defect in that it would send much too much traffic to the root nameservers. Instead, it is common for each organization or ISP to provide what we will call a **resolver** (though, confusingly, these are sometimes also known as "nameservers" or, more precisely, **non**-authoritative nameservers). A resolver is a nearby host charged with looking up DNS names on behalf of the site's users or customers, and returning corresponding addresses. It uses the algorithm of the previous paragraph as a fallback, but also keeps a large **cache** of all the domain names that have been requested. A lifetime for each cache entry is provided by that entry's authoritative nameserver; these lifetimes are typically on the order of several days. If I send a query to Loyola's resolver for `google.com`, it is almost certainly in the cache. If I send a query for the

misspelling googel.com, this may not be in the cache, but the `.com` top-level nameserver almost certainly *is* in the cache. From that nameserver my local resolver finds the nameserver for the `googel.com` zone, and from that finds the IP address of the `googel.com` host.

Applications almost always invoke DNS through library calls, such as Java's `InetAddress.getByName()`. The library forwards the query to the designated resolver. We will return to this in *11.1.3.3 The Client* and *12.6.1 The TCP Client*.

On unix-based systems, traditionally the IPv4 addresses of the local DNS resolvers were kept in a file `/etc/resolv.conf`. Typically this file was updated with the addresses of the current resolvers by DHCP (*7.10 Dynamic Host Configuration Protocol (DHCP)*), at the time the system received its IPv4 address. It is possible, though not common, to create special entries in `/etc/resolv.conf` so that queries about different domains are sent to different resolvers, or so that single-level hostnames have a domain name appended to them before lookup. On Windows, similar functionality can be achieved through settings on the DNS tab within the `Network Connections` applet.

Recent linux systems often run a small resolver locally (*eg* dnsmasq); such resolvers are sometimes also called DNS *forwarders*. The entry in `/etc/resolv.conf` is then an IPv4 address of `localhost` (sometimes 127.0.**1**.1 rather than 127.0.**0**.1). Such a local resolver would, of course, still need access to the addresses of external DNS resolvers.

If a system running a local resolver then runs internal virtual machines, it is usually possible to configure everything so that the virtual machines can be given an IPv4 address of the host system as their DNS resolver. For example, often virtual machines are assigned IPv4 addresses on a private subnet and connect to the outside world using NAT (*7.7 Network Address Translation*). In such a setting, the virtual machines are given the IPv4 address of the host system interface that connects to the private subnet. It is then necessary to ensure that, on the host system, the local resolver accepts queries sent not only to the designated loopback address but also to the host system's private-subnet address. (Generally, local resolvers do *not* accept requests arriving from externally visible addresses.)

At the DNS protocol layer, a DNS lookup query can be either **recursive** or **non-recursive**. If A sends to B a recursive query to resolve a given DNS name, then B takes over the job until it is finally able to return an answer to A. If The query is non-recursive, on the other hand, then if B is not an authoritative nameserver for the DNS name in question it returns either a failure notice or an NS record for the sub-zone that is the next step on the path.

Usually a DNS lookup of a hostname returns a single A record, representing a single IPv4 address. If a site has multiple servers that are entirely equivalent, however, it is possible to give them all the same hostname by configuring the authoritative nameserver to return, for the hostname in question, multiple A records listing, in turn, each of the server IPv4 addresses. This is sometimes known as **round-robin DNS**. It is a simple form of **load balancing**; see also *18.9.5 loadbalance31.py*. Consecutive queries to the nameserver should return the list of A records in different orders; ideally the same should also happen with consecutive queries to a local resolver that has the hostname in its cache. It is also common for a single server, with a single IPv4 address, to be identified by multiple DNS names; see the next section.

The Tor Project uses DNS-like names that end in ".onion". While these are not true DNS names in that they are not managed by the DNS hierarchy, they do work as such for Tor users; see **RFC 7686**. These names follow an unusual pattern: the next level of name is an 80-bit hash of the site's RSA public key (*22.9.1 RSA*), converted to sixteen ASCII bytes. For example, 3g2upl4pq6kufc4m.onion is apparently the Tor address for the search engine duckduckgo.com. Unlike DuckDuckGo, many sites try different RSA keys until they find one where at least some initial prefix of the hash looks more or less meaningful; for example,

nytimes2tsqtnxek.onion. Facebook got very lucky in finding an RSA key whose corresponding Tor address is facebookcorewwwi.onion (though it is sometimes said that *fortune is infatuated with the wealthy*). This naming strategy is a form of **cryptographically generated addresses**; for another example see *8.6.4 Security and Neighbor Discovery*. The advantage of this naming strategy is that you don't need a certificate authority (*22.10.2.1 Certificate Authorities*) to verify a site's RSA key; the site name does it for you.

7.8.1 nslookup

Let us trace a non-recursive lookup of intronetworks.cs.luc.edu, using the nslookup tool. Lines we type in nslookup's interactive mode begin below with the prompt ">"; the shell prompt is "#".

The first step is to look up the IP address of the root nameserver a.name-servers.net. We can do this with a regular call to nslookup, we can look this up in our nameserver's configuration files, or we can search for it on the Internet. The address is 198.41.0.4.

We now send our nonrecursive query to this address; the presence of the single hyphen in the command line below means that we want to use 198.41.0.4 as the nameserver rather than as the thing to be looked up:

```
# nslookup -norecurse - 198.41.0.4
> intronetworks.cs.luc.edu
*** Can't find intronetworks.cs.luc.edu: No answer
```

This is because by default nslookup asks for an A record. What we want is an NS record: the name of the next zone down to ask.

```
> set query=ns
> intronetworks.cs.luc.edu
edu   nameserver = a.edu-servers.net
...
a.edu-servers.net       internet address = 192.5.6.30
```

This is one of the nameservers for the .edu zone. (We also get back several other edu-servers.)

We send the next NS query to a.edu-servers.net:

```
# nslookup -query=ns -norecurse - 192.5.6.30
> intronetworks.cs.luc.edu
...
Authoritative answers can be found from:
luc.edu nameserver = bcdnswt1.it.luc.edu.
bcdnswt1.it.luc.edu    internet address = 147.126.64.64
```

(Again, we show only one of several luc.edu nameservers). We continue.

```
# nslookup -query=ns - -norecurse 147.126.64.64
> intronetworks.cs.luc.edu
...
Authoritative answers can be found from:
cs.luc.edu     nameserver = dns1.cs.luc.edu.
ns1.cs.luc.edu internet address = 147.126.2.44
```

We now ask this last nameserver, for the cs.luc.edu zone, for the A record:

```
# nslookup -query=A -norecurse - 147.126.2.44
> intronetworks.cs.luc.edu
...
intronetworks.cs.luc.edu       canonical name = linode1.cs.luc.edu.
Name:  linode1.cs.luc.edu
Address: 162.216.18.28
```

Here we get a **canonical name**, or CNAME, record. The server that hosts `intronetworks.cs.luc.edu` also hosts several other websites, with different names; for example, introcs.cs.luc.edu (at least as of 2015). This is known as **virtual hosting**. Rather than provide separate A records for each website name, DNS was set up to provide a `CNAME` record for each website name pointing to a single physical server name `linode1.cs.luc.edu`. Only one A record is then needed, for this server.

The `nslookup` request for an A record returned instead the CNAME record, together with the A record for that CNAME (this is the 162.216.18.28 above). This is done for convenience.

Finally, note that the IPv4 address here, 162.216.18.28, is unrelated to Loyola's own IPv4 address block 147.126.0.0/16. The server `linode1.cs.luc.edu` is managed by an external provider; there is no connection between the DNS name hierarchy and the IP address hierarchy.

7.8.2 Other DNS Records

Besides address lookups, DNS also supports a few other kinds of searches. The best known is probably **reverse DNS**, which takes an IP address and returns a name. This is slightly complicated by the fact that one IP address may be associated with multiple DNS names. What DNS does in this case is to return the canonical name, or CNAME; a given address can have only one CNAME.

Given an IPv4 address, say 147.126.1.230, the idea is to reverse it and append to it the suffix `in-addr.arpa`.

```
230.1.126.147.in-addr.arpa
```

There is a DNS name hierarchy for names of this form, with zones and authoritative servers. If all this has been configured – which it often is not, especially for user workstations – a request for the PTR record corresponding to the above should return a DNS hostname. In the case above, the name luc.edu is returned.

DNS also supports MX, or Mail eXchange, records, meant to map a domain name (which originally might not correspond to any hostname, and which today is likely to correspond to the name of a web server) to the hostname of a server that accepts email on behalf of the domain. In effect this allows an organization's domain name, *eg* `luc.edu`, to represent both a web server and, at a different IP address, an email server.

The DNS Security Extensions, DNSSEC, make it possible for authoritative nameservers to provide authenticated responses to DNS queries, by using digital signatures (*22.9 Public-Key Encryption*). DNSSEC defines several additional DNS record types; see **RFC 3833**, **RFC 4033** and, for the new record types, **RFC 4034**.

RFC 6891 outlines a framework for extensions to DNS, including new record types.

7.9 Address Resolution Protocol: ARP

If a host or router A finds that the destination IP address D = D_{IP} matches the network address of one of its interfaces, it is to deliver the packet via the LAN (probably Ethernet). This means looking up the LAN address (MAC address) D_{LAN} corresponding to D_{IP}. How does it do this?

One approach would be via a special server, but the spirit of early IPv4 development was to avoid such servers, for both cost and reliability issues. Instead, the **Address Resolution Protocol** (ARP) is used. This is our first protocol that takes advantage of the existence of LAN-level broadcast; on LANs without physical broadcast (such as ATM), some other mechanism (usually involving a server) must be used.

The basic idea of ARP is that the host A sends out a broadcast ARP query or "who-has D_{IP}?" request, which includes A's own IPv4 and LAN addresses. All hosts on the LAN receive this message. The host for whom the message is intended, D, will recognize that it should reply, and will return an ARP reply or "is-at" message containing D_{LAN}. Because the original request contained A_{LAN}, D's response can be sent directly to A, that is, unicast.

A broadcasts "who-has D"
D replies to A, unicast, and includes its LAN address

Additionally, all hosts maintain an **ARP cache**, consisting of ⟨IPv4,LAN⟩ address pairs for other hosts on the network. After the exchange above, A has $\langle D_{IP}, D_{LAN} \rangle$ in its table; anticipating that A will soon send it a packet to which it needs to respond, D also puts $\langle A_{IP}, A_{LAN} \rangle$ into its cache.

ARP-cache entries eventually expire. The timeout interval used to be on the order of 10 minutes, but linux systems now use a much smaller timeout (~30 seconds observed in 2012). Somewhere along the line, and probably related to this shortened timeout interval, repeat ARP queries about a *timed-out* entry are first sent **unicast**, not broadcast, to the previous Ethernet address on record. This cuts down on the total amount of broadcast traffic; LAN broadcasts are, of course, still needed for new hosts. The ARP cache on a linux system can be examined with the command `ip -s neigh`; the corresponding windows command is `arp -a`.

The above protocol is sufficient, but there is one further point. When A sends its broadcast "who-has D?" ARP query, all other hosts C check their own cache for an entry for **A**. If there *is* such an entry (that is, if A_{IP} is found there), then the value for A_{LAN} is updated with the value taken from the ARP message; if there is no pre-existing entry then no action is taken. This update process serves to avoid stale ARP-cache entries, which can arise is if a host has had its Ethernet interface replaced. (USB Ethernet interfaces, in particular, can be replaced very quickly.)

ARP is quite an efficient mechanism for bridging the gap between IPv4 and LAN addresses. Nodes generally find out neighboring IPv4 addresses through higher-level protocols, and ARP then quickly fills in the missing LAN address. However, in some Software-Defined Networking (*2.7 Software-Defined Networking*) environments, the LAN switches and/or the LAN controller may have knowledge about IPv4/LAN address correspondences, potentially making ARP superfluous. The LAN (Ethernet) switching network might in principle even know exactly how to route *via the LAN* to a given IPv4 address, potentially even making

LAN addresses unnecessary. At such a point, ARP may become an inconvenience. For an example of a situation in which it is necessary to work around ARP, see *18.9.5 loadbalance31.py*.

7.9.1 ARP Finer Points

Most hosts today implement **self-ARP**, or **gratuitous ARP**, on startup (or wakeup): when station A starts up it sends out an ARP query *for itself*: "who-has A?". Two things are gained from this: first, all stations that had A in their cache are now updated with A's most current A_{LAN} address, in case there was a change, and second, if an answer is received, then presumably some other host on the network has the same IPv4 address as A.

Self-ARP is thus the traditional IPv4 mechanism for **duplicate address detection**. Unfortunately, it does not always work as well as might be hoped; often only a single self-ARP query is sent, and if a reply is received then frequently the only response is to log an error message; the host may even continue using the duplicate address! If the duplicate address was received via DHCP, below, then the host is supposed to notify its DHCP server of the error and request a different IPv4 address.

RFC 5227 has defined an improved mechanism known as **Address Conflict Detection**, or ACD. A host using ACD sends out three ARP queries for its new IPv4 address, spaced over a few seconds and leaving the ARP field for the sender's IPv4 address filled with zeroes. This last step means that any other host with that IPv4 address in its cache will ignore the packet, rather than update its cache. If the original host receives no replies, it then sends out two more ARP queries for its new address, this time with the ARP field for the sender's IPv4 address filled in with the new address; this is the stage at which other hosts on the network will make any necessary cache updates. Finally, ACD requires that hosts that do detect a duplicate address must discontinue using it.

It is also possible for other stations to answer an ARP query on behalf of the actual destination D; this is called **proxy ARP**. An early common scenario for this was when host C on a LAN had a modem connected to a serial port. In theory a host D dialing in to this modem should be on a different subnet, but that requires allocation of a new subnet. Instead, many sites chose a simpler arrangement. A host that dialed in to C's serial port might be assigned IP address D_{IP}, from the same subnet as C. C would be configured to route packets to D; that is, packets arriving from the serial line would be forwarded to the LAN interface, and packets sent to C_{LAN} addressed to D_{IP} would be forwarded to D. But we also have to handle ARP, and as D is not actually on the LAN it will not receive broadcast ARP queries. Instead, C would be configured to answer on behalf of D, replying with $\langle D_{IP}, \mathbf{C}_{LAN} \rangle$. This generally worked quite well.

Proxy ARP is also used in Mobile IP, for the so-called "home agent" to intercept traffic addressed to the "home address" of a mobile device and then forward it (*eg* via tunneling) to that device. See *7.13 Mobile IP*.

One delicate aspect of the ARP protocol is that stations are required to respond to a **broadcast** query. In the absence of proxies this theoretically should not create problems: there should be only one respondent. However, there were anecdotes from the Elder Days of networking when a broadcast ARP query would trigger an avalanche of responses. The protocol-design moral here is that determining who is to respond to a broadcast message should be done with great care. (**RFC 1122** section 3.2.2 addresses this same point in the context of responding to broadcast ICMP messages.)

ARP-query implementations also need to include a timeout and some queues, so that queries can be resent if lost and so that a burst of packets does not lead to a burst of queries. A naive ARP algorithm without these might be:

> To send a packet to destination D_{IP}, see if D_{IP} is in the ARP cache. If it is, address the packet to D_{LAN}; if not, send an ARP query for D

To see the problem with this approach, imagine that a 32KB packet arrives at the IP layer, to be sent over Ethernet. It will be fragmented into 22 fragments (assuming an Ethernet MTU of 1500 bytes), all sent at once. The naive algorithm above will likely send an ARP query for *each* of these. What we need instead is something like the following:

> To send a packet to destination D_{IP}:
> If D_{IP} is in the ARP cache, send to D_{LAN} and return
> If not, see if an ARP query for D_{IP} is pending.
> > If it is, put the current packet in a queue for D.
> > If there is no pending ARP query for D_{IP}, start one,
> > > again putting the current packet in the (new) queue for D

We also need:

> If an ARP query for some C_{IP} times out, resend it (up to a point)
> If an ARP query for C_{IP} is answered, send off any packets in C's queue

7.9.2 ARP Security

Suppose A wants to log in to secure server S, using a password. How can B (for Bad) impersonate S?

Here is an ARP-based strategy, sometimes known as **ARP Spoofing**. First, B makes sure the real S is down, either by waiting until scheduled downtime or by launching a denial-of-service attack against S.

When A tries to connect, it will begin with an ARP "who-has S?". All B has to do is answer, "S is-at B". There is a trivial way to do this: B simply needs to set its own IP address to that of S.

A will connect, and may be convinced to give its password to B. B now simply responds with something plausible like "backup in progress; try later", and meanwhile use A's credentials against the real S.

This works even if the communications channel A uses is encrypted! If A is using the SSH protocol (*22.10.1 SSH*), then A will get a message that the other side's key has changed (B will present its own SSH key, not S's). Unfortunately, many users (and even some IT departments) do not recognize this as a serious problem. Some organizations – especially schools and universities – use personal workstations with "frozen" configuration, so that the filesystem is reset to its original state on every reboot. Such systems may be resistant to viruses, but in these environments the user at A will always get a message to the effect that S's credentials are not known.

7.9.3 ARP Failover

Suppose you have two front-line servers, A and B (B for Backup), and you want B to be able to step in if A freezes. There are a number of ways of achieving this, but one of the simplest is known as **ARP Failover**. First, we set $A_{IP} = B_{IP}$, but for the time being B does not use the network so this duplication is not a problem. Then, once B gets the message that A is down, it sends out an ARP query for A_{IP}, including B_{LAN} as the source LAN address. The gateway router, which previously would have had $\langle A_{IP},A_{LAN}\rangle$ in its ARP cache,

updates this to $\langle A_{IP},\mathbf{B}_{LAN}\rangle$, and packets that had formerly been sent to A will now go to B. As long as B is trafficking in stateless operations (*eg* html), B can pick up right where A left off.

7.9.4 Detecting Sniffers

Finally, there is an interesting use of ARP to detect Ethernet password sniffers (generally not quite the issue it once was, due to encryption and switching). To find out if a particular host A is in promiscuous mode, send an ARP "who-has A?" query. Address it not to the broadcast Ethernet address, though, but to some nonexistent Ethernet address.

If promiscuous mode is off, A's network interface will ignore the packet. But if promiscuous mode is on, A's network interface will pass the ARP request to A itself, which is likely then to answer it.

Alas, linux kernels reject at the ARP-software level ARP queries to physical Ethernet addresses other than their own. However, they do respond to faked Ethernet multicast addresses, such as ff:ff:ff:00:00:00 or ff:ff:ff:ff:ff:fe.

7.9.5 ARP and multihomed hosts

If host A has two interfaces `iface1` and `iface2` *on the same LAN*, with respective IP addresses A_1 and A_2, then it is common for the two to be used interchangeably. Traffic addressed to A_1 may be received via `iface2` and vice-versa, and traffic *from* A_1 may be sent via `iface2`. In **RFC 1122**, §3.3.4.2 this is known as the **weak end-system** model; the idea is that we should think of the IP addresses A_1 and A_2 as bound to A rather than to their respective interfaces.

In support of this model, ARP can usually be configured (in fact this is often the default) so that ARP requests for either IP address and received by either interface may be answered with either physical address. Usually all requests are answered with the physical address of the preferred (*ie* faster) interface.

As an example, suppose A has an Ethernet interface `eth0` with IP address 10.0.0.2 and a faster Wi-Fi interface `wlan0` with IP address 10.0.0.3 (although Wi-Fi interfaces are *not* always faster). In this setting, an ARP request "who-has 10.0.0.2" would be answered with `wlan0`'s physical address, and so all traffic to A, to either IP address, would arrive via `wlan0`. The `eth0` interface would go essentially unused. Similarly, though not due to ARP, traffic sent by A with source address 10.0.0.2 might depart via `wlan0`.

This situation is on linux systems adjusted by changing `arp_ignore` and `arp_announce` in `/proc/sys/net/ipv4/conf/all`.

The alternative, in which the two interfaces are kept strictly separate, is the **strong end-system** model.

7.10 Dynamic Host Configuration Protocol (DHCP)

DHCP is the most common mechanism by which hosts are assigned their IPv4 addresses. DHCP started as a protocol known as Reverse ARP (RARP), which evolved into BOOTP and then into its present form. It is documented in **RFC 2131**. Recall that ARP is based on the idea of someone broadcasting an ARP query for a host, containing the host's IPv4 address, and the host answering it with its LAN address. DHCP involves a host, at startup, broadcasting a query containing its *own* LAN address, and having a server reply telling the host what IPv4 address is assigned to it, hence the "Reverse ARP" name.

The DHCP response message is also likely to carry, piggybacked onto it, several other essential startup options. Unlike the IPv4 address, these additional network parameters usually do not depend on the specific host that has sent the DHCP query; they are likely constant for the subnet or even the site. In all, a typical DHCP message includes the following:

- IPv4 address
- subnet mask
- default router
- DNS Server

These four items are a standard **minimal network configuration**; in practical terms, hosts cannot function properly without them. Most DHCP implementations support the piggybacking of the latter three above, and a wide variety of other configuration values, onto the server responses.

Default Routers and DHCP

If you lose your default router, you cannot communicate. Here is something that used to happen to me, courtesy of DHCP:

1. I am connected to the Internet via Ethernet, and my default router is via my Ethernet interface
2. I connect to my institution's wireless network.
3. Their DHCP server sends me a new default router on the wireless network. However, this default router will only allow access to a tiny private network, because I have neglected to complete the "Wi-Fi network registration" process.
4. I therefore disconnect from the wireless network, and my wireless-interface default router goes away. However, my system does not automatically revert to my Ethernet default-router entry; DHCP does not work that way. As a result, I will have no router at all until the next scheduled DHCP lease renegotiation, and must fix things manually.

The DHCP server has a range of IPv4 addresses to hand out, and maintains a database of which IPv4 address has been assigned to which LAN address. Reservations can either be permanent or dynamic; if the latter, hosts typically renew their DHCP reservation periodically (typically one to several times a day).

7.10.1 NAT, DHCP and the Small Office

If you have a large network, with multiple subnets, a certain amount of manual configuration is inevitable. What about, however, a home or small office, with a single line from an ISP? A combination of NAT (*7.7 Network Address Translation*) and DHCP has made **autoconfiguration** close to a reality.

The typical home/small-office "router" is in fact a NAT router (*7.7 Network Address Translation*) coupled with an Ethernet switch, and usually also coupled with a Wi-Fi access point and a DHCP server. In this section, we will use the term "NAT router" to refer to this whole package. One specially designated port, the **external** port, connects to the ISP's line, and uses DHCP as a client to obtain an IPv4 address for that port. The other, **internal**, ports are connected together by an Ethernet switch; these ports as a group are connected to the external port using NAT translation. If wireless is supported, the wireless side is connected directly to the internal ports.

Isolated from the Internet, the internal ports can thus be assigned an arbitrary non-public IPv4 address block, *eg* 192.168.0.0/24. The NAT router typically contains a DCHP server, usually enabled by default, that will hand out IPv4 addresses to everything connecting from the internal side.

Generally this works seamlessly. However, if a second NAT router is also connected to the network (sometimes attempted to extend Wi-Fi range, in lieu of a commercial Wi-Fi repeater), one then has two operating DHCP servers on the same subnet. This often results in chaos, though is easily fixed by disabling one of the DHCP servers.

While omnipresent DHCP servers have made IPv4 autoconfiguration work "out of the box" in many cases, in the era in which IPv4 was designed the need for such servers would have been seen as a significant drawback in terms of expense and reliability. IPv6 has an autoconfiguration strategy (*8.7.2 Stateless Autoconfiguration (SLAAC)*) that does not require DHCP, though DHCPv6 may well end up displacing it.

7.10.2 DHCP and Routers

It is often desired, for larger sites, to have only one or two DHCP servers, but to have them support multiple subnets. Classical DHCP relies on broadcast, which isn't forwarded by routers, and even if it were, the DHCP server would have no way of knowing on what subnet the host in question was actually located.

This is generally addressed by **DHCP Relay** (sometimes still known by the older name BOOTP Relay). The router (or, sometimes, some other node on the subnet) receives the DHCP broadcast message from a host, and notes the subnet address of the arrival interface. The router then relays the DHCP request, together with this subnet address, to the designated DHCP Server; this relayed message is sent directly (unicast), not broadcast. Because the subnet address is included, the DHCP server can figure out the correct IPv4 address to assign.

This feature has to be specially enabled on the router.

7.11 Internet Control Message Protocol

The Internet Control Message Protocol, or ICMP, is a protocol for sending IP-layer error and status messages; it is defined in **RFC 792**. ICMP is, like IP, **host-to-host**, and so they are never delivered to a specific port, even if they are sent in response to an error related to something sent from that port. In other words, individual UDP and TCP connections do not receive ICMP messages, even when it would be helpful to get them.

ICMP messages are identified by an 8-bit **type** field, followed by an 8-bit subtype, or **code**. Here are the more common ICMP types, with subtypes listed in the description.

Type	Description
Echo Request	`ping` queries
Echo Reply	`ping` responses
Destination Unreachable	Destination **network** unreachable
	Destination **host** unreachable
	Destination **port** unreachable
	Fragmentation required but DF flag set
	Network administratively prohibited
Source Quench	Congestion control
Redirect Message	Redirect datagram for the **network**
	Redirect datagram for the **host**
	Redirect for TOS and network
	Redirect for TOS and host
Router Solicitation	Router discovery/selection/solicitation
Time Exceeded	TTL expired in transit
	Fragment reassembly time exceeded
Bad IP Header or Parameter	Pointer indicates the error
	Missing a required option
	Bad length
Timestamp Timestamp Reply	Like `ping`, but requesting a timestamp from the destination

The Echo and Timestamp formats are *queries*, sent by one host to another. Most of the others are all *error messages*, sent by a router to the sender of the offending packet. Error-message formats contain the IP header and next 8 bytes of the packet in question; the 8 bytes will contain the TCP or UDP port numbers. Redirect and Router Solicitation messages are informational, but follow the error-message format. Query formats contain a 16-bit *Query Identifier*, assigned by the query sender and echoed back by the query responder.

`ping` Packet Size

The author once had to diagnose a problem where pings were *almost* 100% successful, and yet file transfers failed immediately; this could have been the result of either a network fault or a file-transfer application fault. The problem turned out to be a failed network device with a very high bit-error rate: 1500-byte file-transfer packets were frequently corrupted, but ping packets, with a default size of 32-64 bytes, were mostly unaffected. If the bit-error rate is such that 1500-byte packets have a 50% success rate, 50-byte packets can be expected to have a 98% ($\simeq 0.5^{1/30}$) success rate. Setting the ping packet size to a larger value made it immediately clear that the network, and not the file-transfer application, was at fault.

ICMP is perhaps best known for Echo Request/Reply, on which the `ping` tool (*1.14 Some Useful Utilities*) is based. Ping remains very useful for network troubleshooting: if you can ping a host, then the network is reachable, and any problems are higher up the protocol chain. Unfortunately, ping replies are often blocked by many firewalls, on the theory that revealing even the existence of computers is a security risk. While this may sometimes be an appropriate decision, it does significantly impair the utility of ping.

Ping can be asked to include IP timestamps (*7.1 The IPv4 Header*) on linux systems with the `-T` option, and on Windows with `-s`.

Source Quench was used to signal that congestion has been encountered. A router that drops a packet due to congestion experience was encouraged to send ICMP Source Quench to the originating host. Generally the

TCP layer would handle these appropriately (by reducing the overall sending rate), but UDP applications never receive them. ICMP Source Quench did not quite work out as intended, and was formally deprecated by **RFC 6633**. (Routers can inform TCP connections of impending congestion by using the ECN bits.)

The Destination Unreachable type has a large number of subtypes:

- **Network unreachable**: some router had no entry for forwarding the packet, and no default route
- **Host unreachable**: the packet reached a router that was on the same LAN as the host, but the host failed to respond to ARP queries
- **Port unreachable**: the packet was sent to a UDP port on a given host, but that port was not open. TCP, on the other hand, deals with this situation by replying to the connecting endpoint with a `reset` packet. Unfortunately, the UDP Port Unreachable message is sent to the host, not to the application on that host that sent the undeliverable packet, and so is close to useless as a practical way for applications to be informed when packets cannot be delivered.
- **Fragmentation required but DF flag set**: a packet arrived at a router and was too big to be forwarded without fragmentation. However, the Don't Fragment bit in the IPv4 header was set, forbidding fragmentation.
- **Administratively Prohibited**: this is sent by a router that knows it can reach the network in question, but has configureintro to drop the packet and send back Administratively Prohibited messages. A router can also be configured to **blackhole** messages: to drop the packet and send back nothing.

In *12.13 Path MTU Discovery* we will see how TCP uses the ICMP message **Fragmentation required but DF flag set** as part of **Path MTU Discovery**, the process of finding the largest packet that can be sent *to a specific destination* without fragmentation. The basic idea is that we set the DF bit on some of the packets we send; if we get back this message, that packet was too big.

Some sites and firewalls block ICMP packets in addition to Echo Request/Reply, and for some messages one can get away with this with relatively few consequences. However, blocking **Fragmentation required but DF flag set** has the potential to severely affect TCP connections, depending on how Path MTU Discovery is implemented, and thus is not recommended. If ICMP filtering is contemplated, it is best to base block/allow decisions on the ICMP type, or even on the type and code. For example, most firewalls support rule sets of the form "allow ICMP destination-unreachable; block all other ICMP".

The **Timestamp** option works something like Echo Request/Reply, but the receiver includes its own local timestamp for the arrival time, with millisecond accuracy. See also the IP Timestamp option, *7.1 The IPv4 Header*, which appears to be more frequently used.

The type/code message format makes it easy to add new ICMP types. Over the years, a significant number of additional such types have been defined; a complete list is maintained by the IANA. Several of these later ICMP types were seldom used and eventually deprecated, many by **RFC 6918**.

ICMP packets are usually forwarded correctly through NAT routers, though due to the absence of port numbers the router must do a little more work. **RFC 3022** and **RFC 5508** address this. For ICMP queries, like ping, the ICMP Query Identifier field can be used to recognize the returning response. ICMP error messages are a little trickier, because there is no direct connection between the inbound error message and any of the previous outbound non-ICMP packets that triggered the response. However, the headers of the packet that triggered the ICMP error message are embedded in the body of the ICMP message. The NAT router can look at those embedded headers to determine how to forward the ICMP message (the NAT router must also rewrite the addresses of those embedded headers).

7.11.1 Traceroute and Time Exceeded

The traceroute program uses ICMP Time Exceeded messages. A packet is sent to the destination (often UDP to an unused port), with the TTL set to 1. The first router the packet reaches decrements the TTL to 0, drops it, and returns an ICMP Time Exceeded message. The sender now knows the first router on the chain. The second packet is sent with TTL set to 2, and the second router on the path will be the one to return ICMP Time Exceeded. This continues until finally the remote host returns something, likely ICMP Port Unreachable.

For an example of traceroute output, see *1.14 Some Useful Utilities*. In that example, the three traceroute probes for the Nth router are sometimes answered by two or even three different routers; this suggests routers configured to work in parallel rather than route changes.

Many routers no longer respond with ICMP Time Exceeded messages when they drop packets. For the distance value corresponding to such a router, traceroute reports `***`.

Traceroute assumes the path does not change. This is not always the case, although in practice it is seldom an issue.

Route Efficiency

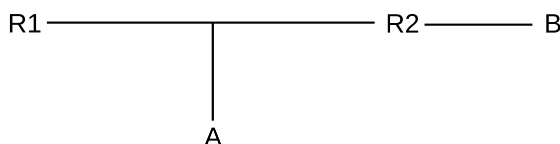
Once upon a time (~2001), traceroute showed that traffic from my home to the office, both in the Chicago area, went through the MAE-EAST Internet exchange point, outside of Washington DC. That inefficient route was later fixed. A situation like this is typically caused by two higher-level providers who did not negotiate sufficient Internet exchange points.

Traceroute to a nonexistent site works up to the point when the packet reaches the Internet “backbone”: the first router which does not have a default route. At that point the packet is not routed further (and an ICMP Destination Network Unreachable should be returned).

Traceroute also interacts somewhat oddly with routers using MPLS (see *20.12 Multi-Protocol Label Switching (MPLS)*). Such routers – most likely large-ISP internal routers – may continue to forward the ICMP Time Exceeded message on further towards its destination before returning it to the sender. As a result, the round-trip time measurements reported may be quite a bit larger than they should be.

7.11.2 Redirects

Most non-router hosts start up with an IPv4 forwarding table consisting of a single (default) router, discovered along with their IPv4 address through DHCP. ICMP Redirect messages help hosts learn of other useful routers. Here is a classic example:



A is configured so that its default router is R1. It addresses a packet to B, and sends it to R1. R1 receives the packet, and forwards it to R2. However, R1 also notices that R2 and A are on the same network, and so A could have sent the packet to R2 directly. So R1 sends an appropriate ICMP redirect message to A ("Redirect Datagram for the Network"), and A adds a route to B via R2 to its own forwarding table.

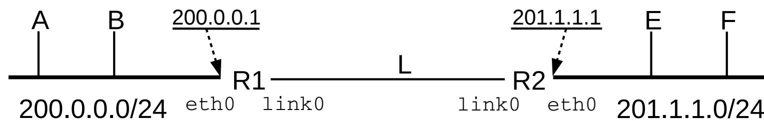
7.11.3 Router Solicitation

These ICMP messages are used by some router protocols to identify immediate neighbors. When we look at routing-update algorithms, *9 Routing-Update Algorithms*, these are where the process starts.

7.12 Unnumbered Interfaces

We mentioned in *1.10 IP - Internet Protocol* and *7.2 Interfaces* that some devices allow the use of point-to-point IP links without assigning IP addresses to the interfaces at the ends of the link. Such IP interfaces are referred to as **unnumbered**; they generally make sense only on routers. It is a firm requirement that the node (*ie* router) at each endpoint of such a link has at least one other interface that *does* have an IP address; otherwise, the node in question would be anonymous, and could not participate in the router-to-router protocols of *9 Routing-Update Algorithms*.

The diagram below shows a link L joining routers R1 and R2, which are connected to subnets 200.0.0.0/24 and 201.1.1.0/24 respectively. The endpoint interfaces of L, both labeled `link0`, are unnumbered.



Two LANs joined by an unnumbered link L

The endpoints of L could always be assigned private IPv4 addresses (*7.3 Special Addresses*), such as 10.0.0.1 and 10.0.0.2. To do this we would need to create a subnet; because the host bits cannot be all 0's or all 1's, the minimum subnet size is four (*eg* 10.0.0.0/30). Furthermore, the routing protocols to be introduced in *9 Routing-Update Algorithms* will distribute information about the subnet throughout the organization or "routing domain", meaning care must be taken to ensure that each link's subnet is unique. Use of unnumbered links avoids this.

If R1 were to *originate* a packet to be sent to (or forwarded via) R2, the standard strategy is for it to treat its `link0` interface as if it shared the IP address of its Ethernet interface `eth0`, that is, 200.0.0.1; R2 would do likewise. This still leaves R1 and R2 violating the IP local-delivery rule of *7.5 The Classless IP Delivery Algorithm*; R1 is expected to deliver packets via local delivery to 201.1.1.1 but has no interface that is assigned an IP address on the destination subnet 201.1.1.0/24. The necessary dispensation, however, is granted by **RFC 1812**. All that is necessary by way of configuration is that R1 be told R2 is a directly connected neighbor reachable via its `link0` interface. On linux systems this might be done with the `ip route` command on R1 as follows:

ip route

The linux `ip route` command illustrated here was tested on a virtual point-to-point link created with `ssh` and `pppd`; the link interface name was in fact `ppp0`. While the command appeared to work as advertised, it was only possible to create the link if endpoint IP addresses were assigned at the time of creation; these were then removed with `ip route del` and then re-assigned with the command shown here.

```
ip route add 201.1.1.1 dev link0
```

Because L is a point-to-point link, there is no destination LAN address and thus no ARP query.

7.13 Mobile IP

In the original IPv4 model, there was a strong if implicit assumption that each IP host would stay put. One role of an IPv4 address is simply as a unique endpoint identifier, but another role is as a **locator**: some prefix of the address (*eg* the network part, in the class-A/B/C strategy, or the provider prefix) represents something about where the host is physically located. Thus, if a host moves far enough, it may need a new address.

When laptops are moved from site to site, it is common for them to receive a new IP address at each location, *eg* via DHCP as the laptop connects to the local Wi-Fi. But what if we wish to support devices like smartphones that may remain active and communicating while moving for thousands of miles? Changing IP addresses requires changing TCP connections; life (and application development) might be simpler if a device had a single, unchanging IP address.

One option, commonly used with smartphones connected to some so-called “3G” networks, is to treat the phone’s data network as a giant wireless LAN. The phone’s IP address need not change as it moves within this LAN, and it is up to the phone provider to figure out how to manage LAN-level routing, much as is done in *3.7.4.3 Wi-Fi Roaming*.

But **Mobile IP** is another option, documented in **RFC 5944**. In this scheme, a mobile host has a permanent **home address** and, while roaming about, will also have a temporary **care-of address**, which changes from place to place. The care-of address might be, for example, an IP address assigned by a local Wi-Fi network, and which in the absence of Mobile IP would be *the* IP address for the mobile host. (This kind of care-of address is known as “co-located”; the care-of address can also be associated with some other device – known as a **foreign agent** – in the vicinity of the mobile host.) The goal of Mobile IP is to make sure that the mobile host is always reachable via its home address.

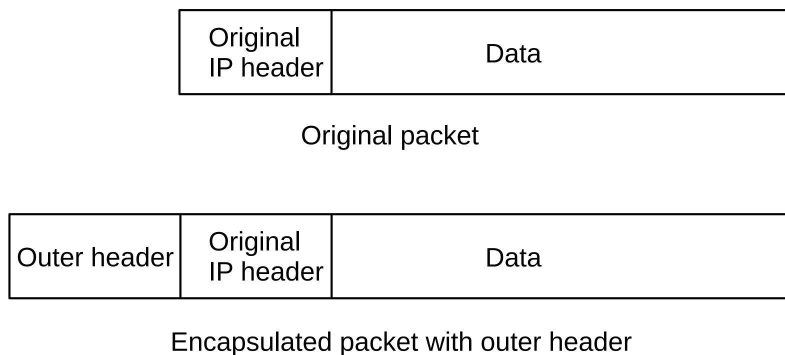
To maintain connectivity to the home address, a Mobile IP host needs to have a **home agent** back on the home network; the job of the home agent is to maintain an IP tunnel that always connects to the device’s current care-of address. Packets arriving at the home network addressed to the home address will be forwarded to the mobile device over this tunnel by the home agent. Similarly, if the mobile device wishes to send packets *from* its home address – that is, with the home address as IP source address – it can use the tunnel to forward the packet to the home agent.

The home agent may use proxy ARP (*7.9.1 ARP Finer Points*) to declare itself to be the appropriate destination on the home LAN for packets addressed to the home (IP) address; it is then straightforward for the home agent to forward the packets.

An **agent discovery** process is used for the mobile host to decide whether it is mobile or not; if it is, it then needs to notify its home agent of its current care-of address.

7.13.1 IP-in-IP Encapsulation

There are several forms of packet encapsulation that can be used for Mobile IP tunneling, but the default one is IP-in-IP encapsulation, defined in **RFC 2003**. In this process, the entire original IP packet (with header addressed to the home address) is used as data for a new IP packet, with a new IP header (the "outer" header) addressed to the care-of address.



A value of 4 in the outer-IP-header `Protocol` field indicates that IPv4-in-IPv4 tunneling is being used, so the receiver knows to forward the packet on using the information in the inner header. The MTU of the tunnel will be the original MTU of the path to the care-of address, minus the size of the outer header. A very similar mechanism is used for IPv6-in-IPv4 encapsulation (that is, with IPv6 in the inner packet), except that the outer IPv4 `Protocol` field value is now 41. See *8.13 IPv6 Connectivity via Tunneling*.

IP-in-IP encapsulation presents some difficulties for NAT routers. If two hosts A and B behind a NAT router send out encapsulated packets, the packets may differ only in the source IP address. The NAT router, upon receiving responses, doesn't know whether to forward them to A or to B. One partial solution is for the NAT router to support only one inside host sending encapsulated packets. If the NAT router knew that encapsulation was being used for Mobile IP, it might look at the home address in the inner header to determine the correct home agent to which to deliver the packet, but this is a big assumption. A fuller solution is outlined in **RFC 3519**.

7.14 Epilog

At this point we have concluded the basic mechanics of IPv4. Still to come is a discussion of how IP routers build their forwarding tables. This turns out to be a complex topic, divided into routing within single organizations and ISPs – *9 Routing-Update Algorithms* – and routing between organizations – *10 Large-Scale IP Routing*.

But before that, in the next chapter, we compare IPv4 with IPv6, now twenty years old but still seeing limited adoption. The biggest issue fixed by IPv6 is IPv4's lack of address space, but there are also several other less dramatic improvements.

7.15 Exercises

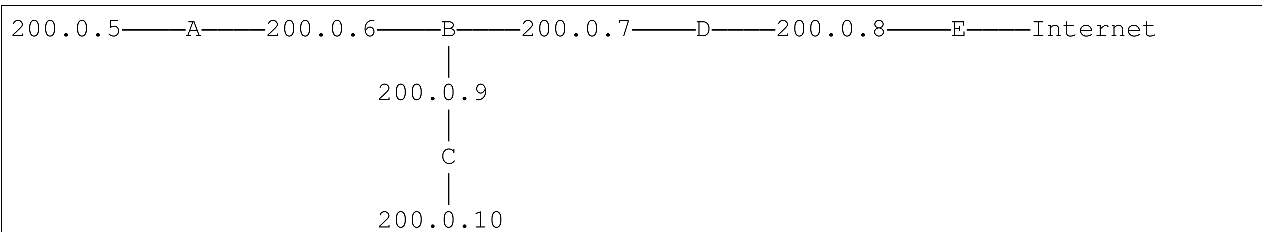
Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercise 6.5 is distinct, for example, from exercises 6.0 and 7.0. Exercises marked with a ♢ have solutions or hints at 24.7 Solutions for IPv4.

1.0. Suppose an Ethernet packet represents a TCP acknowledgment; that is, the packet contains an IPv4 header and a 20-byte TCP header but nothing else. Is the IPv4 packet here smaller than the Ethernet minimum-packet size, and, if so, by how much?

2.0. How can a receiving host tell if an arriving IPv4 packet is unfragmented? Hint: such a packet will be both the "first fragment" and the "last fragment"; how are these two states marked in the IPv4 header?

3.0. How long will it take the IDENT field of the IPv4 header to wrap around, if the sender host A sends a stream of packets to host B as fast as possible? Assume the packet size is 1500 bytes and the bandwidth is 600 Mbps.

4.0. The following diagram has routers A, B, C, D and E; E is the "border router" connecting the site to the Internet. All router-to-router connections are via Ethernet-LAN /24 subnets with addresses of the form 200.0.x. Give forwarding tables for each of A♢, B, C and D. Each table should include each of the listed subnets and also a **default** entry that routes traffic toward router E. Directly connected subnets may be listed with a next_hop of "direct".



5.0. (This exercise is an attempt at modeling Internet-2 routing.) Suppose sites S_1 ... S_n each have a single connection to the standard Internet, and each site S_i has a single IPv4 address block A_i. Each site's connection to the Internet is through a single router R_i; each R_i's default route points towards the standard Internet. The sites also maintain a separate, higher-speed network among themselves; each site has a single link to this separate network, also through R_i. Describe what the forwarding tables on each R_i will have to look like so that traffic from one S_i to another will always use the separate higher-speed network.

6.0. For each IPv4 network prefix given (with length), identify which of the subsequent IPv4 addresses are part of the same subnet.

(a). **10.0.130.0/23**: 10.0.130.23, 10.0.129.1, 10.0.131.12, 10.0.132.7

(b). **10.0.132.0/22**: 10.0.130.23, 10.0.135.1, 10.0.134.12, 10.0.136.7

(c). **10.0.64.0/18**: 10.0.65.13, 10.0.32.4, 10.0.127.3, 10.0.128.4

(d).♢ **10.0.168.0/21**: 10.0.166.1, 10.0.170.3, 10.0.174.5, 10.0.177.7

(e). **10.0.0.64/26**: 10.0.0.125, 10.0.0.66, 10.0.0.130, 10.0.0.62

6.5. Convert the following subnet masks to /k notation, and vice-versa:

(a).◊ 255.255.240.0
(b). 255.255.248.0
(c). 255.255.255.192
(d).◊ /20
(e). /22
(f). /27

7.0. Suppose that the subnet bits below for the following five subnets A-E all come from the beginning of the fourth byte of the IPv4 address; that is, these are subnets of a /24 block.

- A: 00
- B: 01
- C: 110
- D: 111
- E: 1010

(a). What are the sizes of each subnet, and the corresponding decimal ranges? Count the addresses with host bits all 0's or with host bits all 1's as part of the subnet.
(b). How many IPv4 addresses in the class-C block do not belong to any of the subnets A, B, C, D and E?

8.0. In *7.9 Address Resolution Protocol: ARP* it was stated that, in newer implementations, "repeat ARP queries about a timed out entry are first sent unicast", in order to reduce broadcast traffic. Suppose multiple unicast repeat-ARP queries for host A's IP address fail, but a followup broadcast query for A's address succeeds. What probably changed at host A?

9.0. Suppose A broadcasts an ARP query "who-has B?", receives B's response, and proceeds to send B a regular IPv4 packet. If B now wishes to reply, why is it likely that A will already be present in B's ARP cache? Identify a circumstance under which this can fail.

10.0. Suppose A broadcasts an ARP request "who-has B", but inadvertently lists the physical address of another machine C instead of its own (that is, A's ARP query has IP_{src} = A, but LAN_{src} = C). What will happen? Will A receive a reply? Will any other hosts on the LAN be able to send to A? What entries will be made in the ARP caches on A, B and C?

11.0. Suppose host A connects to the Internet via Wi-Fi. The default router is R_W. Host A now begins exchanging packets with a remote host B: A sends to B, B replies, *etc*. The exact form of the connection does not matter, except that TCP may not work.

(a). You now plug in A's Ethernet cable. The Ethernet port is assumed to be on a different subnet from the Wi-Fi (so that the strong and weak end-system models of *7.9.5 ARP and multihomed hosts* do not play a role here). Assume A automatically selects the new Ethernet connection as its default route, with router R_E. What happens to the original connection to A? Can packets still travel back and forth? Does the return address used for either direction change?

(b). You now *disconnect* A's Wi-Fi interface, leaving the Ethernet interface connected. What happens now to the connection to B? Hint: to what IP address are the packets from B being sent?

See also *9 Routing-Update Algorithms*, *9 Routing-Update Algorithms* exercise 13.0, and *12 TCP Transport*, exercise 13.0.

What has been learned from experience with IPv4? First and foremost, more than 32 bits are needed for addresses; the primary motive in developing IPv6 was the specter of running out of IPv4 addresses (something which, at the highest level, has already happened; see the discussion at the end of *1.10 IP - Internet Protocol*). Another important issue is that IPv4 requires (or used to require) a modest amount of effort at configuration; IPv6 was supposed to improve this.

By 1990 the IETF was actively interested in proposals to replace IPv4. A working group for the so-called "IP next generation", or IPng, was created in 1993 to select the new version; **RFC 1550** was this group's formal solicitation of proposals. In July 1994 the IPng directors voted to accept a modified version of the "Simple Internet Protocol", or SIP (unrelated to the Session Initiation Protocol, *20.11.4 RTP and VoIP*), as the basis for IPv6. The first IPv6 specifications, released in 1995, were **RFC 1883** (now **RFC 2460**, with updates) for the basic protocol, and **RFC 1884** (now **RFC 4291**, again with updates) for the addressing architecture.

SIP addresses were originally 64 bits in length, but in the month leading up to adoption as the basis for IPv6 this was increased to 128. 64 bits would probably have been enough, but the problem is less the actual number than the simplicity with which addresses can be allocated; the more bits, the easier this becomes, as sites can be given relatively large address blocks without fear of waste. A secondary consideration in the 64-to-128 leap was the potential to accommodate now-obsolete CLNP addresses (*1.15 IETF and OSI*), which were up to 160 bits in length, but compressible.

IPv6 has to some extent returned to the idea of a fixed division between network and host portions; for most IPv6 addresses, the first 64 bits is the network prefix (including any subnet portion) and the remaining 64 bits represents the host portion. The rule as spelled out in **RFC 2460**, in 1998, was that the 64/64 split would apply to all addresses except those beginning with the bits 000; those addresses were then held in reserve in the unlikely event that the 64/64 split ran into problems in the future. This was a change from 1995, when **RFC 1884** envisioned 48-bit host portions and 80-bit prefixes.

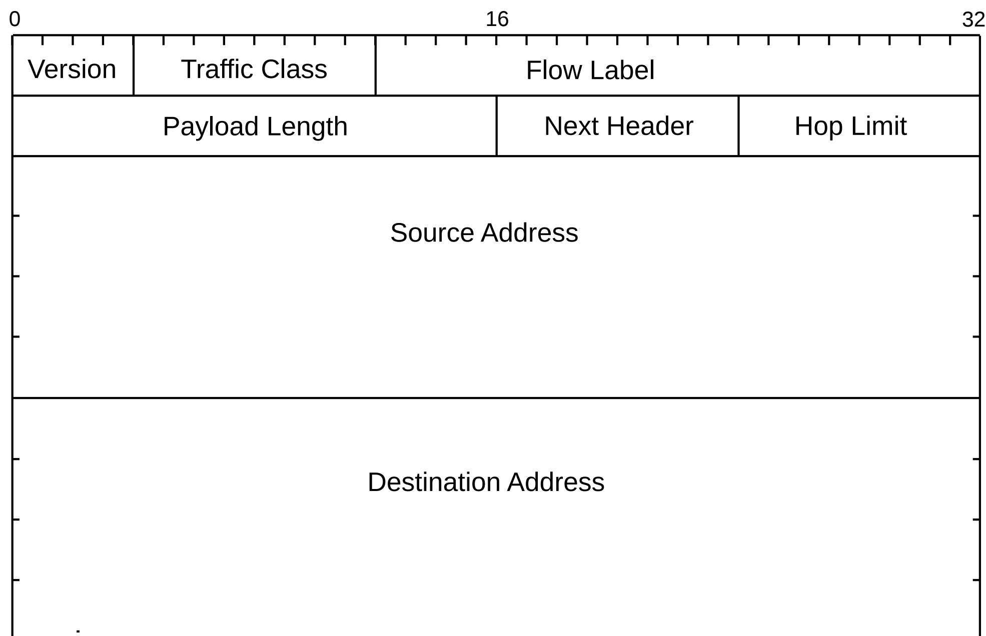
While the IETF occasionally revisits the issue, at the present time the 64/64 split seems here to stay; for discussion and justification, see **RFC 7421**. The 64/64 split is not automatic, however; there is no default prefix length as there was in IPv4. Thus, it is misleading to think of IPv6 as a return to something like IPv4's Class A/B/C addressing scheme. Router advertisements must always include the prefix length, and, when assigning IPv6 addresses manually, the /64 prefix length must be specified explicitly; see *8.12.3 Manual address configuration*.

High-level routing, however, can, as in IPv4, be done on prefixes of any length (usually that means lengths shorter than /64). Routing can also be done on different prefix lengths at different points of the network.

IPv6 is now twenty years old, and yet usage as of 2015 remains quite modest. However, the shortage in IPv4 addresses has begun to loom ominously; IPv6 adoption rates may rise quickly if IPv4 addresses begin to climb in price.

8.1 The IPv6 Header

The IPv6 **fixed header** is pictured below; at 40 bytes, it is twice the size of the IPv4 header. The fixed header is intended to support only what *every* packet needs: there is no support for fragmentation, no header checksum, and no option fields. However, the concept of **extension headers** has been introduced to support some of these as options; some IPv6 extension headers are described in *8.5 IPv6 Extension Headers*. Whatever header comes next is identified by the Next Header field, much like the IPv4 Protocol field. Some other fixed-header fields have also been renamed from their IPv4 analogues: the IPv4 TTL is now the IPv6 Hop_Limit (still decremented by each router with the packet discarded when it reaches 0), and the IPv4 DS field has become the IPv6 Traffic Class.



The Flow Label is new. **RFC 2460** states that it

> may be used by a source to label sequences of packets for which it requests special handling by the IPv6 routers, such as non-default quality of service or "real-time" service.

Senders not actually taking advantage of any quality-of-service options are supposed to set the Flow Label to zero.

When used, the Flow Label represents a sender-computed hash of the source and destination addresses, and perhaps the traffic class. Routers can use this field as a way to look up quickly any priority or reservation state for the packet. All packets belonging to the same flow should have the same Routing Extension header, *8.5.3 Routing Header*. The Flow Label will in general *not* include any information about the source and destination *port* numbers, except that only some of the connections between a pair of hosts may make use of this field.

A **flow**, as the term is used here, is *one-way*; the return traffic belongs to a different flow. Historically, the term "flow" has also been used at various other scales: a single bidirectional TCP connection, multiple *related* TCP connections, or even all traffic from a particular subnet (*eg* the "computer-lab flow").

8.2 IPv6 Addresses

IPv6 addresses are written in eight groups of four hex digits, with a-f preferred over A-F (**RFC 5952**). The groups are separated by colons, and have leading 0's removed, *eg*

> fedc:13:1654:310:fedc:bc37:61:3210

If an address contains a long run of 0's – for example, if the IPv6 address had an embedded IPv4 address – then when writing the address the string "::" should be used to represent however many blocks of 0000 as are needed to create an address of the correct length; to avoid ambiguity this can be used only once. Also, embedded IPv4 addresses may continue to use the "." separator:

> ::ffff:147.126.65.141

The above is an example of one standard IPv6 format for representing IPv4 addresses (see *8.11 Using IPv6 and IPv4 Together*). 48 bits are explicitly displayed; the :: means these are prefixed by 80 0-bits.

The IPv6 loopback address is ::1 (that is, 127 0-bits followed by a 1-bit).

Network address **prefixes** may be written with the "/" notation, as in IPv4:

> 12ab:0:0:cd30::/60

RFC 3513 suggested that initial IPv6 unicast-address allocation be initially limited to addresses beginning with the bits 001, that is, the 2000::/3 block (20 in binary is 0010 0000).

Generally speaking, IPv6 addresses consist of a 64-bit network prefix (perhaps including subnet bits) followed by a 64-bit "interface identifier". See *8.3 Network Prefixes* and *8.2.1 Interface identifiers*.

IPv6 addresses all have an associated **scope**, defined in **RFC 4007**. The scope of a unicast address is either **global**, meaning it is intended to be globally routable, or **link-local**, meaning that it will only work with directly connected neighbors (*8.2.2 Link-local addresses*). The loopback address is considered to have link-local scope. A few more scope levels are available for multicast addresses, *eg* "site-local" (**RFC 4291**). The scope of an IPv6 address is implicitly coded within the first 64 bits; addresses in the 2000::/3 block above, for example, have global scope.

Packets with local-scope addresses (*eg* link-local addresses) for either the destination or the source cannot be routed (the latter because a reply would be impossible).

Although addresses in the "unique local address" category of *8.3 Network Prefixes* officially have global scope, in a practical sense they still behave as if they had the now-officially-deprecated "site-local scope".

8.2.1 Interface identifiers

As mentioned earlier, most IPv6 addresses can be divided into a 64-bit network prefix and a 64-bit "host" portion, the latter corresponding to the "host" bits of an IPv4 address. These host-portion bits are known officially as the **interface identifier**; the change in terminology reflects the understanding that all IP addresses attach to interfaces rather than to hosts.

The original plan for the interface identifier was to derive it in most cases from the LAN address, though the interface identifier can also be set administratively. Given a 48-bit Ethernet address, the interface identifier based on it was to be formed by inserting 0xfffe between the first three bytes and the last three bytes, to get 64 bits in all. The seventh bit of the first byte (the Ethernet "universal/local" flag) was then set to 1.

The result of this process is officially known as the **Modified EUI-64 Identifier**, where EUI stands for Extended Unique Identifier; details can be found in **RFC 4291**. As an example, for a host with Ethernet address 00:a0:cc:24:b0:e4, the EUI-64 identifier would be 0**2**a0:ccff:fe24:b0e4 (the leading 00 becomes 02 when the seventh bit is turned on). At the time the EUI-64 format was proposed, it was widely expected that Ethernet MAC addresses would eventually become 64 bits in length.

EUI-64 interface identifiers turn out to introduce a major privacy concern: no matter where a (portable) host connects to the Internet – home or work or airport or Internet cafe – such an interface identifier always remains the same, and thus serves as a permanent host fingerprint. As a result, EUI-64 identifiers are now discouraged for personal workstations and mobile devices. (Some fixed-location hosts continue to use EUI-64 interface identifiers, or, alternatively, administratively assigned interface identifiers.)

RFC 7217 proposes an alternative: the interface identifier is a secure hash (*22.6 Secure Hashes*) of a “Net_Iface” parameter, the 64-bit IPv6 address prefix, and a host-specific secret key (a couple other parameters are also thrown into the mix, but they need not concern us here). The “Net_Iface” parameter can be the interface’s MAC address, but can also be the interface’s “name”, *eg* `eth0`. Interface identifiers created this way change from connection point to connection point (because the prefix changes), do not reveal the Ethernet address, and are randomly scattered (because of the key, if nothing else) through the 2^{64}-sized interface-identifier space. The last feature makes probing for IPv6 addresses effectively impossible; see exercise 6.0.

Interface identifiers as in the previous paragraph do not change unless the prefix changes, which normally happens only if the host is moved to a new network. In *8.7.2.1 SLAAC privacy* we will see that interface identifiers are often changed at regular intervals, for privacy reasons.

Finally, interface identifiers are often centrally assigned, using DHCPv6 (*8.7.3 DHCPv6*).

Remote probing for IPv6 addresses based on EUI-64 identifiers is much easier than for those based on RFC-7217 identifiers, as the former are not very random. If an attacker can guess the hardware vendor, and thus the first three bytes of the Ethernet address (*2.1.3 Ethernet Address Internal Structure*), there are only 2^{24} possibilities, down from 2^{64}. As the last three bytes are often assigned in serial order, considerable further narrowing of the search space may be possible. While it may amount to security through obscurity, keeping internal global IPv6 addresses hidden is often of practical importance.

Additional discussion of host-scanning in IPv6 networks can be found in **RFC 7707** and draft-ietf-opsec-ipv6-host-scanning-06.

8.2.2 Link-local addresses

IPv6 defines **link-local** addresses, with so-called link-local scope, intended to be used only on a single LAN and never routed. These begin with the 64-bit link-local prefix consisting of the ten bits 1111 1110 10 followed by 54 more zero bits; that is, fe80::/64. The remaining 64 bits are the interface identifier for the link interface in question, above. The EUI-64 link-local address of the machine in the previous section with Ethernet address 00:a0:cc:24:b0:e4 is thus fe80::2a0:ccff:fe24:b0e4.

The main applications of link-local addresses are as a “bootstrap” address for global-address autoconfiguration (*8.7.2 Stateless Autoconfiguration (SLAAC)*), and as an optional permanent address for routers. IPv6 routers often communicate with neighboring routers via their link-local addresses, with the understanding that these do not change when global addresses (or subnet configurations) change (**RFC 4861** §6.2.8). If EUI-64 interface identifiers are used then the link-local address does change whenever the Ethernet hard-

ware is replaced. However, if **RFC 7217** interface identifiers are used and that mechanism's "Net_Iface" parameter represents the interface name rather than its physical address, the link-local address can be constant for the life of the host. (When RFC 7217 is used to generate link-local addresses, the "prefix" hash parameter is the link-local prefix fe80::/64.)

A consequence of identifying routers to their neighbors by their link-local addresses is that it is often possible to configure routers so they do not even have global-scope addresses; for forwarding traffic and for exchanging routing-update messages, link-local addresses are sufficient. Similarly, many ordinary hosts forward packets to their default router using the latter's link-local address. We will return to router addressing in *8.13.2 Setting up a router* and *8.13.2.1 A second router*.

For non-Ethernet-like interfaces, *eg* tunnel interfaces, there may be no natural candidate for the interface identifier, in which case a link-local address *may* be assigned manually, with the low-order 64 bits chosen to be unique for the link in question.

When sending to a link-local address, one must separately supply somewhere the link's "zone identifier", often by appending a string containing the interface name to the IPv6 address, *eg* fe80::f00d:cafe**%eth0**. See *8.12.1 ping6* and *8.12.2 TCP connections using link-local addresses* for examples of such use of link-local addresses.

IPv4 also has true link-local addresses, defined in **RFC 3927**, though they are rarely used; such addresses are in the 169.254.0.0/16 block (not to be confused with the 192.168.0.0/16 private-address block). Other than these, IPv4 addresses always implicitly identify the link subnet by virtue of the network prefix.

Once the link-local address is created, it must pass the **duplicate-address detection** test before being used; see *8.7.1 Duplicate Address Detection*.

8.2.3 Anycast addresses

IPv6 also introduced **anycast** addresses. An anycast address might be assigned to each of a set of routers (in addition to each router's own unicast addresses); a packet addressed to this anycast address would be delivered to only one member of this set. Note that this is quite different from multicast addresses; a packet addressed to the latter is delivered to *every* member of the set.

It is up to the local routing infrastructure to decide which member of the anycast group would receive the packet; normally it would be sent to the "closest" member. This allows hosts to send to any of a set of routers, rather than to their designated individual default router.

Anycast addresses are not marked as such, and a node sending to such an address need not be aware of its anycast status. Addresses are anycast simply because the routers involved have been configured to recognize them as such.

8.3 Network Prefixes

We have been assuming that an IPv6 address, at least as seen by a host, is composed of a 64-bit network prefix and a 64-bit interface identifier. As of 2015 this remains a requirement; **RFC 4291** (IPv6 Addressing Architecture) states:

> For all unicast addresses, except those that start with the binary value 000, Interface IDs are required to be 64 bits long....

This /64 requirement is occasionally revisited by the IETF, but is unlikely to change for mainstream IPv6 traffic. This firm 64/64 split is a departure from IPv4, where the host/subnet division point has depended, since the development of subnets, on local configuration.

Note that while the net/interface (net/host) division point is fixed, routers may still use CIDR (*10.1 Classless Internet Domain Routing: CIDR*) and may still base forwarding decisions on prefixes shorter than /64.

As of 2015, all allocations for globally routable IPv6 prefixes are part of the 2000::/3 block.

IPv6 also defines a variety of specialized network prefixes, including the link-local prefix and prefixes for anycast and multicast addresses. For example, as we saw earlier, the prefix ::ffff:0:0/96 identifies IPv6 addresses with embedded IPv4 addresses.

The most important class of 64-bit network prefixes, however, are those supplied by a provider or other address-numbering entity, and which represent the first half of globally routable IPv6 addresses. These are the prefixes that will be visible to the outside world.

IPv6 customers will typically be assigned a relatively large block of addresses, *eg* /48 or /56. The former allows 64–48 = 16 bits for local "subnet" specification within a 64-bit network prefix; the latter allows 8 subnet bits. These subnet bits are – as in IPv4 – supplied through router configuration; see *8.10 IPv6 Subnets*. The closest IPv6 analogue to the IPv4 subnet mask is that all network prefixes are supplied to hosts with an associated length, although that length will almost always be 64 bits.

Many sites will have only a single externally visible address block. However, some sites may be multihomed and thus have multiple independent address blocks.

Sites may also have private **unique local address** prefixes, corresponding to IPv4 private address blocks like 192.168.0.0/16 and 10.0.0.0/8. They are officially called Unique Local Unicast Addresses and are defined in **RFC 4193**; these replace an earlier **site-local** address plan (and official site-local scope) formally deprecated in **RFC 3879** (though unique-local addresses are sometimes still informally referred to as site-local).

The first 8 bits of a unique-local prefix are 1111 1101 (fd00::/8). The related prefix 1111 1100 (fc00::/8) is reserved for future use; the two together may be consolidated as fc00::/7. The last 16 bits of a 64-bit unique-local prefix represent the subnet ID, and are assigned either administratively or via autoconfiguration. The 40 bits in between, from bit 8 up to bit 48, represent the **Global ID**. A site is to set the Global ID to a pseudorandom value.

The resultant unique-local prefix is "almost certainly" globally unique (and is considered to have **global scope** in the sense of *8.2 IPv6 Addresses*), although it is not supposed to be routed off a site. Furthermore, a site would generally not admit any packets from the outside world addressed to a destination with the Global ID as prefix. One rationale for choosing unique Global IDs for each site is to accommodate potential later mergers of organizations without the need for renumbering; this has been a chronic problem for sites using private IPv4 address blocks. Another justification is to accommodate VPN connections from other sites. For example, if I use IPv4 block 10.0.0.0/8 at home, and connect using VPN to a site also using 10.0.0.0/8, it is possible that my printer will have the same IPv4 address as their application server.

8.4 IPv6 Multicast

IPv6 has moved away from LAN-layer *broad*cast, instead providing a wide range of LAN-layer *multi*cast groups. (Note that LAN-layer multicast is often straightforward; it is general IP-layer multicast

(*20.5 Global IP Multicast*) that is problematic. See *2.1.2 Ethernet Multicast* for the Ethernet implementation.) This switch to multicast is intended to limit broadcast traffic in general, though many switches still propagate LAN multicast traffic everywhere, like broadcast.

An IPv6 multicast address is one beginning with the eight bits 1111 1111 (ff00::/8); numerous specific such addresses, and even classes of addresses, have been defined. For actual delivery, IPv6 multicast addresses correspond to LAN-layer (*eg* Ethernet) multicast addresses through a well-defined static correspondence; specifically, if x, y, z and w are the last four bytes of the IPv6 multicast address, in hex, then the corresponding Ethernet multicast address is 33:33:x:y:z:w (**RFC 2464**). A typical IPv6 host will need to join (that is, subscribe to) several Ethernet multicast groups.

The IPv6 multicast address with the broadest scope is **all-nodes**, with address ff02::1; the corresponding Ethernet multicast address is 33:33:00:00:00:01. This essentially corresponds to IPv4's LAN broadcast, though the use of LAN multicast here means that non-IPv6 hosts should not see packets sent to this address. Another important IPv6 multicast address is ff02::2, the **all-routers** address. This is meant to be used to reach all routers, and routers only; ordinary hosts do not subscribe.

Generally speaking, IPv6 nodes on Ethernets send LAN-layer **Multicast Listener Discovery** (MLD) messages to multicast groups they wish to start using; these messages allow multicast-aware Ethernet switches to optimize forwarding so that only those hosts that have subscribed to the multicast group in question will receive the messages. Otherwise switches are supposed to treat multicast like broadcast; worse, some switches may simply fail to forward multicast packets to destinations that have not explicitly opted to join the group.

8.5 IPv6 Extension Headers

In IPv4, the IP header contained a Protocol field to identify the next header; usually UDP or TCP. All IPv4 options were contained in the IP header itself. IPv6 has replaced this with a scheme for allowing an arbitrary chain of supplemental IPv6 headers. The IPv6 Next Header field *can* indicate that the following header is UDP or TCP, but can also indicate one of several IPv6 options. These optional, or extension, headers include:

- Hop-by-Hop options header
- Destination options header
- Routing header
- Fragment header
- Authentication header
- Mobility header
- Encapsulated Security Payload header

These extension headers must be processed in order; the recommended order for inclusion is as above. Most of them are intended for processing only at the destination host; the hop-by-hop and routing headers are exceptions.

8.5.1 Hop-by-Hop Options Header

This consists of a set of ⟨type,value⟩ pairs which are intended to be processed by each router on the path. A tag in the type field indicates what a router should do if it does not understand the option: drop the packet, or continue processing the rest of the options. The only Hop-by-Hop options provided by **RFC 2460** were for padding, so as to set the alignment of later headers.

RFC 2675 later defined a Hop-by-Hop option to support IPv6 **jumbograms**: datagrams larger than 65,535 bytes. The need for such large packets remains unclear, in light of *5.3 Packet Size*. IPv6 jumbograms are not meant to be used if the underlying LAN does not have an MTU larger than 65,535 bytes; the LAN world is not currently moving in this direction.

Because Hop-by-Hop Options headers must be processed by each router encountered, they have the potential to overburden the Internet routing system. As a result, **RFC 6564** strongly discourages new Hop-by-Hop Option headers, unless examination at every hop is essential.

8.5.2 Destination Options Header

This is very similar to the Hop-by-Hop Options header. It again consists of a set of ⟨type,value⟩ pairs, and the original **RFC 2460** specification only defined options for padding. The Destination header is intended to be processed at the destination, before turning over the packet to the transport layer.

Since **RFC 2460**, a few more Destination Options header types have been defined, though none is in common use. **RFC 2473** defined a Destination Options header to limit the nesting of tunnels, called the Tunnel Encapsulation Limit. **RFC 6275** defines a Destination Options header for use in Mobile IPv6. **RFC 6553**, on the Routing Protocol for Low-Power and Lossy Networks, or RPL, has defined a Destination (and Hop-by-Hop) Options type for carrying RPL data.

A complete list of Option Types for Hop-by-Hop Option and Destination Option headers can be found at www.iana.org/assignments/ipv6-parameters; in accordance with **RFC 2780**.

8.5.3 Routing Header

The original, or Type 0, Routing header contained a list of IPv6 addresses through which the packet should be routed. These did not have to be contiguous. If the list to be visited en route to destination D was ⟨R1,R2,...,Rn⟩, then this option header contained ⟨R2,R3,...,Rn,D⟩ with R1 as the initial destination address; R1 then would update this header to ⟨R1,R3,...,Rn,D⟩ (that is, the old destination R1 and the current next-router R2 were swapped), and would send the packet on to R2. This was to continue on until Rn addressed the packet to the final destination D. The header contained a Segments Left pointer indicating the next address to be processed, incremented at each Ri. When the packet arrived at D the Routing Header would contain the routing list ⟨R1,R3,...,Rn⟩. This is, in general principle, very much like IPv4 Loose Source routing. Note, however, that routers *between* the listed routers R1...Rn did not need to examine this header; they processed the packet based only on its current destination address.

This form of routing header was deprecated by **RFC 5095**, due to concerns about a traffic-amplification attack. An attacker could send off a packet with a routing header containing an alternating list of just two routers ⟨R1,R2,R1,R2,...,R1,R2,D⟩; this would generate substantial traffic on the R1–R2 link. **RFC 6275** and **RFC 6554** define more limited routing headers. **RFC 6275** defines a quite limited routing header to be used for IPv6 mobility (and also defines the IPv6 Mobility header). The **RFC 6554** routing header used for

RPL, mentioned above, has the same basic form as the Type 0 header described above, but its use is limited to specific low-power routing domains.

8.5.4 IPv6 Fragment Header

IPv6 supports limited IPv4-style fragmentation via the Fragment Header. This header contains a 13-bit Fragment Offset field, which contains – as in IPv4 – the 13 high-order bits of the actual 16-bit offset of the fragment. This header also contains a 32-bit Identification field; all fragments of the same packet must carry the same value in this field.

IPv6 fragmentation is done *only* by the original sender; routers along the way are not allowed to fragment or re-fragment a packet. Sender fragmentation would occur if, for example, the sender had an 8KB IPv6 packet to send via UDP, and needed to fragment it to accommodate the 1500-byte Ethernet MTU.

If a packet needs to be fragmented, the sender first identifies the **unfragmentable part**, consisting of the IPv6 fixed header and any extension headers that must accompany each fragment (these would include Hop-by-Hop and Routing headers). These unfragmentable headers are then attached to each fragment.

IPv6 also requires that every link on the Internet have an MTU of at least 1280 bytes beyond the LAN header; link-layer fragmentation and reassembly can be used to meet this MTU requirement (which is what ATM links (*3.5 Asynchronous Transfer Mode: ATM*) carrying IP traffic do).

Generally speaking, fragmentation should be avoided at the application layer when possible. UDP-based applications that attempt to transmit filesystem-sized (usually 8 KB) blocks of data remain persistent users of fragmentation.

8.5.5 General Extension-Header Issues

In the IPv4 world, many middleboxes (*7.7.2 Middleboxes*) examine not just the destination address but also the TCP port numbers; firewalls, for example, do this routinely to block all traffic except to a designated list of ports. In the IPv6 world, a middlebox may have difficulty *finding* the TCP header, as it must traverse a possibly lengthy list of extension headers. Worse, some of these extension headers may be newer than the middlebox, and thus unrecognized. Some middleboxes would simply drop packets with unrecognized extension headers, making the introduction of new such headers problematic.

RFC 6564 addresses this by requiring that all future extension headers use a common "type-length-value" format: the first byte indicates the extension-header's type and the second byte indicates its length. This facilitiates rapid traversal of the extension-header chain. A few older extension headers – for example the Encapsulating Security Payload header of **RFC 4303** – do not follow this rule; middleboxes must treat these as special cases.

RFC 2460 states

> With one exception [that is, Hop-by-Hop headers], extension headers are not examined or processed by any node along a packet's delivery path, until the packet reaches the node (or each of the set of nodes, in the case of multicast) identified in the Destination Address field of the IPv6 header.

Nonetheless, sometimes intermediate nodes do attempt to add extension headers. This can break Path MTU Discovery (*12.13 Path MTU Discovery*), as the sender no longer controls the total packet size.

RFC 7045 attempts to promulgate some general rules for the real-world handling of extension headers. For example, it states that, while routers are allowed to drop packets with certain extension headers, they may not do this simply because those headers are unrecognized. Also, routers *may* ignore Hop-by-Hop Option headers, or else process packets with such headers via a slower queue.

8.6 Neighbor Discovery

IPv6 Neighbor Discovery, or **ND**, is a set of related protocols that replaces several IPv4 tools, most notably ARP, ICMP redirects and most non-address-assignment parts of DHCP. The messages exchanged in ND are part of the ICMPv6 framework, *8.9 ICMPv6*. The original specification for ND is in **RFC 2461**, later updated by **RFC 4861**. ND provides the following services:

- Finding the local router(s) [*8.6.1 Router Discovery*]
- Finding the set of network address prefixes that can be reached via local delivery (IPv6 allows there to be more than one) [*8.6.2 Prefix Discovery*]
- Finding a local host's LAN address, given its IPv6 address [*8.6.3 Neighbor Solicitation*]
- Detecting duplicate IPv6 addresses [*8.7.1 Duplicate Address Detection*]
- Determining that some neighbors are now unreachable

8.6.1 Router Discovery

IPv6 routers periodically send **Router Advertisement** (RA) packets to the all-nodes multicast group. Ordinary hosts wanting to know what router to use can wait for one of these periodic multicasts, or can request an RA packet immediately by sending a **Router Solicitation** request to the all-routers multicast group. Router Advertisement packets serve to identify the routers; this process is sometimes called **Router Discovery**. In IPv4, by comparison, the address of the default router is usually piggybacked onto the DHCP response message (*7.10 Dynamic Host Configuration Protocol (DHCP)*).

These RA packets, in addition to identifying the routers, also contain a list of all network address prefixes in use on the LAN. This is "prefix discovery", described in the following section. To a first approximation on a simple network, prefix discovery supplies the network portion of the IPv6 address; on IPv4 networks, DHCP usually supplies the entire IPv4 address.

RA packets may contain other important information about the LAN as well, such as an agreed-on MTU.

These IPv6 router messages represent a change from IPv4, in which routers need not send anything besides forwarded packets. To become an IPv4 router, a node need only have IPv4 forwarding enabled in its kernel; it is then up to DHCP (or the equivalent) to inform neighboring nodes of the router. IPv6 puts the responsibility for this notification on the router itself: for a node to become an IPv6 router, in addition to forwarding packets, it "MUST" (**RFC 4294**) also run software to support Router Advertisement. Despite this mandate, however, the RA mechanism does not play a role in the forwarding process itself; an IPv6 network can run without Router Advertisements if every node is, for example, manually configured to know where the routers are and to know which neighbors are on-link. (We emphasize that manual configuration like this scales very poorly.)

On linux systems, the Router Advertisement agent is most often the radvd daemon. See *8.13 IPv6 Connectivity via Tunneling* below.

8.6.2 Prefix Discovery

Closely related to Router Discovery is the **Prefix Discovery** process by which hosts learn what IPv6 network-address prefixes, above, are valid on the network. It is also where hosts learn which prefixes are considered to be local to the host's LAN, and thus reachable at the LAN layer instead of requiring router assistance for delivery. IPv6, in other words, does *not* limit determination of whether delivery is local to the IPv4 mechanism of having a node check a destination address against each of the network-address prefixes assigned to the node's interfaces.

Even IPv4 allows two IPv4 network prefixes to share the same LAN (*eg* a private one 10.1.2.0/24 and a public one 147.126.65.0/24), but a consequence of IPv4 routing is that two such LAN-sharing subnets can only reach one another via a router on the LAN, even though they should in principle be able to communicate directly. IPv6 drops this restriction.

The Router Advertisement packets sent by the router should contain a complete list of valid network-address prefixes, as the **Prefix Information** option. In simple cases this list may contain a single globally routable 64-bit prefix corresponding to the LAN subnet. If a particular LAN is part of multiple (overlapping) physical subnets, the prefix list will contain an entry for each subnet; these 64-bit prefixes will themselves likely share a common site-wide prefix of length N<64. For multihomed sites the prefix list may contain multiple unrelated prefixes corresponding to the different address blocks. Finally, site-specific "unique local" IPv6 address prefixes may also be included.

Each prefix will have an associated **lifetime**; nodes receiving a prefix from an RA packet are to use it only for the duration of this lifetime. On expiration (and likely much sooner) a node must obtain a newer RA packet with a newer prefix list. The rationale for inclusion of the prefix lifetime is ultimately to allow sites to easily **renumber**; that is, to change providers and switch to a new network-address prefix provided by a new router. Each prefix is also tagged with a bit indicating whether it can be used for autoconfiguration, as in *8.7.2 Stateless Autoconfiguration (SLAAC)* below.

Each prefix also comes with a flag indicating whether the prefix is **on-link**. If set, then every node receiving that prefix is supposed to be on the same LAN. Nodes assume that to reach a neighbor sharing the same on-link address prefix, Neighbor Solicitation is to be used to find the neighbor's LAN address. If a neighbor shares an off-link prefix, a router must be used. The IPv4 equivalent of two nodes sharing the same on-link prefix is sharing the same subnet prefix. For an example of subnets with prefix-discovery information, see *8.10 IPv6 Subnets*.

Routers advertise off-link prefixes only in special cases; this would mean that a node is part of a subnet but cannot reach other members of the subnet directly. This may apply in some wireless settings, *eg* MANETs (*3.7.8 MANETs*) where some nodes on the same subnet are out of range of one another. It may also apply when using IPv6 Mobility (*7.13 Mobile IP*, **RFC 3775**).

8.6.3 Neighbor Solicitation

Neighbor Solicitation messages are the IPv6 analogues of IPv4 ARP requests. These are essentially queries of the form "who has IPv6 address X?" While ARP requests were broadcast, IPv6 Neighbor Solicitation messages are sent to the **solicited-node multicast address**, which at the LAN layer usually represents a

rather small multicast group. This address is ff02::0001:x.y.z.w, where x, y, z and w are the low-order 32 bits of the IPv6 address the sender is trying to look up. Each IPv6 host on the LAN will need to subscribe to all the solicited-node multicast addresses corresponding to its own IPv6 addresses (normally this is not too many).

Neighbor Solicitation messages are repeated regularly, but followup verifications are initially sent to the unicast LAN address on file (this is common practice with ARP implementations, but is optional). Unlike with ARP, other hosts on the LAN are not expected to eavesdrop on the initial Neighbor Solicitation message. The target host's response to a Neighbor Solicitation message is called **Neighbor Advertisement**; a host may also send these unsolicited if it believes its LAN address may have changed.

The analogue of Proxy ARP is still permitted, in that a node may send Neighbor Advertisements on behalf of another. The most likely reason for this is that the node receiving proxy services is a "mobile" host temporarily remote from the home LAN. Neighbor Advertisements sent as proxies have a flag to indicate that, if the real target does speak up, the proxy advertisement should be ignored.

Once a node (host or router) has discovered a neighbor's LAN address through Neighbor Solicitation, it continues to monitor the neighbor's continued reachability.

Neighbor Solicitation also includes Neighbor Unreachability Detection. Each node (host or router) continues to monitor its known neighbors; reachability can be inferred either from ongoing IPv6 traffic exchanges or from Neighbor Advertisement responses. If a node detects that a neighboring host has become unreachable, the original node may retry the multicast Neighbor Solicitation process, in case the neighbor's LAN address has simply changed. If a node detects that a neighboring *router* has become unreachable, it attempts to find an alternative path.

Finally, IPv4 ICMP Redirect messages have also been moved in IPv6 to the Neighbor Discovery protocol. These allow a router to tell a host that another router is better positioned to handle traffic to a given destination.

8.6.4 Security and Neighbor Discovery

In the protocols outlined above, received ND messages are trusted; this can lead to problems with nodes pretending to be things they are not. Here are two examples:

- A host can pretend to be a router simply by sending out Router Advertisements; such a host can thus capture traffic from its neighbors, and even send it on – perhaps selectively – to the real router.
- A host can pretend to be another host, in the IPv6 analog of ARP spoofing (*7.9.2 ARP Security*). If host A sends out a Neighbor Solicitation for host B, nothing prevents host C from sending out a Neighbor Advertisement claiming to be B (after previously joining the appropriate multicast group).

These two attacks can have the goal either of eavesdropping or of denial of service; there are also purely denial-of-service attacks. For example, host C can answer host B's DAD queries (below at *8.7.1 Duplicate Address Detection*) by claiming that the IPv6 address in question is indeed in use, preventing B from ever acquiring an IPv6 address. A good summary of these and other attacks can be found in **RFC 3756**.

These attacks, it is worth noting, can only be launched by nodes on the same LAN; they cannot be launched remotely. While this reduces the risk, though, it does not eliminate it. Sites that allow anyone to connect, such as Internet cafés, run the highest risk, but even in a setting in which all workstations are "locked down", a node compromised by a virus may be able to disrupt the network.

RFC 4861 suggested that, at sites concerned about these kinds of attacks, hosts might use the IPv6 Authentication Header or the Encapsulated Security Payload Header to supply digital signatures for ND packets (see *22.11 IPsec*). If a node is configured to require such checks, then most ND-based attacks can be prevented. Unfortunately, **RFC 4861** offered no suggestions beyond static configuration, which scales poorly and also rather completely undermines the goal of autoconfiguration.

A more flexible alternative is Secure Neighbor Discovery, or **SEND**, specified in **RFC 3971**. This uses public-key encryption (*22.9 Public-Key Encryption*) to validate ND messages; for the remainder of this section, some familiarity with the material at *22.9 Public-Key Encryption* may be necessary. Each message is digitally signed by the sender, using the sender's private key; the recipient can validate the message using the sender's corresponding public key. In principle this makes it impossible for one message sender to pretend to be another sender.

In practice, the problem is that public keys by themselves guarantee (if not compromised) only that the sender of a message is the same entity that previously sent messages using that key. In the second bulleted example above, in which C sends an ND message falsely claiming to be B, straightforward applications of public keys would prevent this *if* the original host A had previously heard from B, and trusted that sender to be the real B. But in general A would not know which of B or C was the real B. A cannot trust whichever host it heard from first, as it is indeed possible that C started its deception with A's very first query for B, beating B to the punch.

A common solution to this identity-guarantee problem is to create some form of "public-key infrastructure" such as **certificate authorities**, as in *22.10.2.1 Certificate Authorities*. In this setting, every node is configured to trust messages signed by the certificate authority; that authority is then configured to vouch for the identities of other nodes whenever this is necessary for secure operation. SEND implements its own version of certificate authorities; these are known as **trust anchors**. These would be configured to guarantee the identities of all routers, and perhaps hosts. The details are somewhat simpler than the mechanism outlined in *22.10.2.1 Certificate Authorities*, as the anchors and routers are under common authority. When trust anchors are used, each host needs to be configured with a list of their addresses.

SEND also supports a simpler public-key validation mechanism known as **cryptographically generated addresses**, or CGAs (**RFC 3972**). These are IPv6 interface identifiers that are secure hashes (*22.6 Secure Hashes*) of the host's public key (and a few other non-secret parameters). CGAs are an alternative to the interface-identifier mechanisms discussed in *8.2.1 Interface identifiers*. DNS names in the .onion domain used by TOR also use CGAs.

The use of CGAs makes it impossible for host C to successfully claim to be host B: only B will have the public key that hashes to B's address *and* the matching private key. If C attempts to send to A a neighbor advertisement claiming to be B, then C can sign the message with its own private key, but the hash of the corresponding public key will not match the interface-identifier portion of B's address. Similarly, in the DAD scenario, if C attempts to tell B that B's newly selected CGA address is already in use, then again C won't have a key matching that address, and B will ignore the report.

In general, CGI addresses allow recipients of a message to verify that the source address is the "owner" of the associated public key, without any need for a public-key infrastructure (*22.9.3 Trust and the Man in the Middle*). C *can* still pretend to be a router, using its own CGA address, because router addresses are not known by the requester beforehand. However, it is easier to protect routers using trust anchors as there are fewer of them.

SEND relies on the fact that finding two inputs hashing to the same 64-bit CGA is infeasible, as in general this would take about 2^{64} tries. An IPv4 analog would be impossible as the address host portion won't have

enough bits to prevent finding hash collisions via brute force. For example, if the host portion of the address has ten bits, it would take C about 2^{10} tries (by tweaking the supplemental hash parameters) until it found a match for B's CGA.

SEND has seen very little use in the IPv6 world, partly because IPv6 itself has seen such slow adoption, but also because of the perception that the vulnerabilities SEND protects against are difficult to exploit.

RA-guard is a simpler mechanism to achieve ND security, but one that requires considerable support from the LAN layer. Outlined in **RFC 6105**, it requires that each host connects directly to a switch; that is, there must be no shared-media Ethernet. The switches must also be fairly smart; it must be possible to configure them to know which ports connect to routers rather than hosts, and, in addition, it must be possible to configure them to block Router Advertisements from host ports that are *not* router ports. This is quite effective at preventing a host from pretending to be a router, and, while it assumes that the switches can do a significant amount of packet inspection, that is in fact a fairly common Ethernet switch feature. If Wi-Fi is involved, it does require that access points (which are a kind of switch) be able to block Router Advertisements; this isn't quite as commonly available. In determining which switch ports are connected to routers, **RFC 6105** suggests that there might be a brief initial learning period, during which all switch ports connecting to a device that *claims* to be a router are considered, permanently, to be router ports.

8.7 IPv6 Host Address Assignment

IPv6 provides two competing ways for hosts to obtain their full IP addresses. One is **DHCPv6**, based on IPv4's DHCP (*7.10 Dynamic Host Configuration Protocol (DHCP)*), in which the entire address is handed out by a DHCPv6 server. The other is **StateLess Address AutoConfiguration**, or SLAAC, in which the interface-identifier part of the address is generated locally, and the network prefix is obtained via prefix discovery. The original idea behind SLAAC was to support complete plug-and-play network setup: hosts on an isolated LAN could talk to one another out of the box, and if a router was introduced connecting the LAN to the Internet, then hosts would be able to determine unique, routable addresses from information available from the router.

In the early days of IPv6 development, in fact, DHCPv6 may have been intended only for address assignments to routers and servers, with SLAAC meant for "ordinary" hosts. In that era, it was still common for IPv4 addresses to be assigned "statically", via per-host configuration files. **RFC 4862** states that SLAAC is to be used when "a site is not particularly concerned with the exact addresses hosts use, so long as they are unique and properly routable."

SLAAC and DHCPv6 evolved to some degree in parallel. While SLAAC solves the autoconfiguration problem quite neatly, at this point DHCPv6 solves it just as effectively, and provides for greater administrative control. For this reason, SLAAC may end up less widely deployed. On the other hand, SLAAC gives hosts greater control over their IPv6 addresses, and so may end up offering hosts a greater degree of privacy by allowing endpoint management of the use of private and temporary addresses (below).

When a host first begins the Neighbor Discovery process, it receives a Router Advertisement packet. In this packet are two special bits: the M (managed) bit and the O (other configuration) bit. The M bit is set to indicate that DHCPv6 is available on the network for address assignment. The O bit is set to indicate that DHCPv6 is able to provide additional configuration information (*eg* the name of the DNS server) to hosts that are using SLAAC to obtain their addresses. In addition, each individual prefix in the RA packet has an A bit, which when set indicates that the associated prefix may be used with SLAAC.

8.7.1 Duplicate Address Detection

Whenever an IPv6 host obtains a unicast address – a link-local address, an address created via SLAAC, an address received via DHCPv6 or a manually configured address – it goes through a **duplicate-address detection** (DAD) process. The host sends one or more Neighbor Solicitation messages (that is, like an ARP query), as in *8.6 Neighbor Discovery*, asking if any other host has this address. If anyone answers, then the address is a duplicate. As with IPv4 ACD (*7.9.1 ARP Finer Points*), but *not* as with the original IPv4 self-ARP, the source-IP-address field of this NS message is set to a special "unspecified" value; this allows other hosts to recognize it as a DAD query.

Because this NS process may take some time, and because addresses are in fact almost always unique, **RFC 4429** defines an **optimistic DAD** mechanism. This allows limited use of an address before the DAD process completes; in the meantime, the address is marked as "optimistic".

Outside the optimistic-DAD interval, a host is not allowed to use an IPv6 address if the DAD process has failed. **RFC 4862** in fact goes further: if a host with an established address receives a DAD query for that address, indicating that some other host wants to use that address, then the original host should discontinue use of the address.

If the DAD process fails for an address based on an EUI-64 identifier, then some other node has the same Ethernet address and you have bigger problems than just finding a working IPv6 address. If the DAD process fails for an address constructed with the **RFC 7217** mechanism, *8.2.1 Interface identifiers*, the host is able to generate a new interface identifier and try again. A counter for the number of DAD attempts is included in the hash that calculates the interface identifier; incrementing this counter results in an entirely new identifier.

While DAD works quite well on Ethernet-like networks with true LAN-layer multicast, it may be inefficient on, say, MANETs (*3.7.8 MANETs*), as distant hosts may receive the DAD Neighbor Solicitation message only after some delay, or even not at all. Work continues on the development of improvements to DAD for such networks.

8.7.2 Stateless Autoconfiguration (SLAAC)

To obtain an address via SLAAC, defined in **RFC 4862**, the first step for a host is to generate its link-local address (above, *8.2.2 Link-local addresses*), appending the standard 64-bit link-local prefix fe80::/64 to its interface identifier (*8.2.1 Interface identifiers*). The latter is likely derived from the host's LAN address using either EUI-64 or the **RFC 7217** mechanism; the important point is that it is available without network involvement.

The host must then ensure that its newly configured link-local address is in fact unique; it uses DAD (above) to verify this. Assuming no duplicate is found, then at this point the host can talk to any other hosts on the same LAN, *eg* to figure out where the printers are.

The next step is to see if there is a router available. The host may send a Router Solicitation (RS) message to the all-routers multicast address. A router – if present – should answer with a Router Advertisement (RA) message that also contains a Prefix Information option; that is, a list of IPv6 network-address prefixes (*8.6.2 Prefix Discovery*).

As mentioned earlier, the RA message will mark with a flag those prefixes eligible for use with SLAAC; if no prefixes are so marked, then SLAAC should not be used. All prefixes will also be marked with a lifetime,

indicating how long the host may continue to use the prefix. Once the prefix expires, the host must obtain a new one via a new RA message.

The host chooses an appropriate prefix, stores the prefix-lifetime information, and appends the prefix to the front of its interface identifier to create what should now be a routable address. The address so formed must now be verified through the DAD mechanism above.

In the era of EUI-64 interface identifiers, it would in principle have been possible for the receiver of a packet to extract the sender's LAN address from the interface-identifier portion of the sender's SLAAC-generated IPv6 address. This in turn would allow bypassing the Neighbor Solicitation process to look up the sender's LAN address. This was never actually permitted, however, even before the privacy options below, as there is no way to be certain that a received address was in fact generated via SLAAC. With **RFC 7217**-based interface identifiers, LAN-address extraction is no longer even potentially an option.

A host using SLAAC may receive multiple network prefixes, and thus generate for itself multiple addresses. **RFC 6724** defines a process for a host to determine, when it wishes to connect to destination address D, which of its own multiple addresses to use. For example, if D is a unique-local address, not globally visible, then the host will likely want to choose a source address that is also unique-local. **RFC 6724** also includes mechanisms to allow a host with a permanent public address (possibly corresponding to a DNS entry, but just as possibly formed directly from an interface identifier) to prefer alternative "temporary" or "privacy" addresses for outbound connections. Finally, **RFC 6724** also defines the sorting order for multiple addresses representing the same destination; see *8.11 Using IPv6 and IPv4 Together*.

At the end of the SLAAC process, the host knows its IPv6 address (or set of addresses) and its default router. In IPv4, these would have been learned through DHCP along with the identity of the host's DNS server; one concern with SLAAC is that it originally did not provide a way for a host to find its DNS server. One strategy is to fall back on DHCPv6 for this. However, **RFC 6106** now defines a process by which IPv6 routers can include DNS-server information in the RA packets they send to hosts as part of the SLAAC process; this completes the final step of autoconfiguration.

How to get DNS names for SLAAC-configured IPv6 hosts into the DNS servers is an entirely separate issue. One approach is simply not to give DNS names to such hosts. In the NAT-router model for IPv4 autoconfiguration, hosts on the inward side of the NAT router similarly do not have DNS names (although they are also not reachable directly, while SLAAC IPv6 hosts would be reachable). If DNS names are needed for hosts, then a site might choose DHCPv6 for address assignment instead of SLAAC. It is also possible to figure out the addresses SLAAC would use (by identifying the host-identifier bits) and then creating DNS entries for these hosts. Finally, hosts can also use **Dynamic DNS** (**RFC 2136**) to update their own DNS records.

8.7.2.1 SLAAC privacy

A portable host that always uses SLAAC as it moves from network to network and always bases its SLAAC addresses on the EUI-64 interface identifier (or on any other static interface identifier) will be easy to track: its interface identifier will never change. This is one reason why the obfuscation mechanism of **RFC 7217** interface identifiers (*8.2.1 Interface identifiers*) includes the network *prefix* in the hash: connecting to a new network will then result in a new interface identifier.

Well before **RFC 7217**, however, **RFC 4941** introduced a set of **privacy extensions** to SLAAC: optional mechanisms for the generation of alternative interface identifiers, based as with RFC 7217 on pseudorandom generation using the original LAN-address-based interface identifier as a "seed" value.

RFC 4941 goes further, however, in that it supports **regular changes** to the interface identifier, to increase the difficulty of tracking a host over time even if it does not change its network prefix. One first selects a 128-bit secure-hash function F(), *eg* MD5 (*22.6 Secure Hashes*). New temporary interface IDs (IIDs) can then be calculated as follows

$(IID_{new}, seed_{new}) = F(seed_{old}, IID_{old})$

where the left-hand pair represents the two 64-bit halves of the 128-bit return value of F() and the arguments to F() are concatenated together. (The seventh bit of IID_{new} must also be set to 0; *cf 8.2.1 Interface identifiers* where this bit is set to 1.) This process is privacy-safe even if the initial IID is based on EUI-64.

The probability of two hosts accidentally choosing the same interface identifier in this manner is vanishingly small; the Neighbor Solicitation mechanism with DAD must, however, still be used to verify that the address is in fact unique within the host's LAN.

The privacy addresses above are to be used only for connections initiated by the client; to the extent that the host accepts incoming connections and so needs a "fixed" IPv6 address, the address based on the original EUI-64/RFC-7217 interface identifier should still be available. As a result, the RFC 7217 mechanism is still important for privacy even if the RFC 4941 mechanism is fully operational.

RFC 4941 stated that privacy addresses were to be disabled by default, largely because of concerns about frequently changing IP addresses. These concerns have abated with experience and so privacy addresses are often now automatically enabled. Typical address lifetimes range from a few hours to 24 hours. Once an address has "expired" it generally remains available but *deprecated* for a few temporary-address cycles longer.

DHCPv6 also provides an option for temporary address assignments, again to improve privacy, but one of the potential advantages of SLAAC is that this process is entirely under the control of the end system.

Regularly (*eg* every few hours, or less) changing the host portion of an IPv6 address should make external tracking of a host more difficult, at least if tracking via web-browser cookies is also somehow prevented. However, for a residential "site" with only a handful of hosts, a considerable degree of tracking may be obtained simply by observing the common 64-bit prefix.

For a general discussion of privacy issues related to IPv6 addressing, see **RFC 7721**.

8.7.3 DHCPv6

The job of a DHCPv6 server is to tell an inquiring host its network prefix(es) and also supply a 64-bit host-identifier, very similar to an IPv4 DHCPv4 server. Hosts begin the process by sending a DHCPv6 request to the All_DHCP_Relay_Agents_and_Servers multicast IPv6 address ff02::1:2 (versus the broadcast address for IPv4). As with DHCPv4, the job of a relay agent is to tag a DHCPv6 request with the correct current subnet, and then to forward it to the actual DCHPv6 server. This allows the DHCPv6 server to be on a different subnet from the requester. Note that the use of multicast does nothing to diminish the need for relay agents. In fact, the All_DHCP_Relay_Agents_and_Servers multicast address scope is limited to the current LAN; relay agents then forward to the actual DHCPv6 server using the *site*-scoped address All_DHCP_Servers.

Hosts using SLAAC to obtain their address can still use a special Information-Request form of DHCPv6 to obtain their DNS server and any other "static" DHCPv6 information.

Clients may ask for **temporary** addresses. These are identified as such in the "Identity Association" field of the DHCPv6 request. They are handled much like "permanent" address requests, except that the client may ask for a new temporary address only a short time later. When the client does so, a *different* temporary address will be returned; a repeated request for a permanent address, on the other hand, would usually return the same address as before.

When the DHCPv6 server returns a temporary address, it may of course keep a log of this address. The absence of such a log is one reason SLAAC may provide a greater degree of privacy. SLAAC also places control of the cryptographic mechanisms for temporary-address creation in the hands of the end user.

A DHCPv6 response contains a list (perhaps of length 1) of IPv6 addresses. Each separate address has an expiration date. The client must send a new request before the expiration of any address it is actually using.

In DHCPv4, the host portion of addresses typically comes from "address pools" representing small ranges of integers such as 64-254; these values are generally allocated consecutively. A DHCPv6 server, on the other hand, should take advantage of the enormous range (2^{64}) of possible host portions by allocating values more sparsely, through the use of pseudorandomness. This is to make it very difficult for an outsider who knows one of a site's host addresses to guess the addresses of other hosts, *cf 8.2.1 Interface identifiers*.

The Internet Draft draft-ietf-dhc-stable-privacy-addresses proposes the following mechanism by which a DHCPv6 server may generate the interface-identifier bits for the addresses it hands out; F() is a secure-hash function and its arguments are concatenated together:

F(prefix, client_DUID, IAID, DAD_counter, secret_key)

The prefix, DAD_counter and secret_key arguments are as in *8.7.2.1 SLAAC privacy*. The client_DUID is the string by which the client identifies itself to the DHCPv6 server; it may be based on the Ethernet address though other options are possible. The IAID, or Identity Association identifier, is a client-provided name for this request; different names are used when requesting temporary versus permanent addresses.

Some older DHCPv6 servers may still allocate interface identifiers in serial order; such obsolete servers might make the SLAAC approach more attractive.

8.8 Globally Exposed Addresses

Perhaps the most striking difference between a contemporary IPv4 network and an IPv6 network is that on the former, many hosts are likely to be "hidden" behind a NAT router (*7.7 Network Address Translation*). On an IPv6 network, on the other hand, *every* host may be globally visible to the IPv6 world (though NAT may still be used to allow connectivity to legacy IPv4 servers).

Legacy IPv4 NAT routers provide a measure of each of privacy, security and nuisance. Privacy in IPv6 can be handled, as above, through private or temporary addresses.

The degree of security provided via NAT is entirely due to the fact that all connections must be initiated from the inside; no packet from the outside is allowed through the NAT firewall unless it is a response to a packet sent from the inside. This feature, however, can also be implemented via a conventional firewall (IPv4 or IPv6), without address translation. Furthermore, given such a conventional firewall, it is then straightforward to modify it so as to support limited and regulated connections from the outside world as desired; an analogous modification of a NAT router is more difficult. (That said, a blanket ban on IPv6 connections from the outside can prove as frustrating as IPv4 NAT.)

Finally, one of the major reasons for hiding IPv4 addresses is that with IPv4 it is easy to map a /24 subnet by pinging or otherwise probing each of the 254 possible hosts; such mapping may reveal internal structure. In IPv6 such mapping is meant to be impractical as a /64 subnet has $2^{64} \simeq 18$ quintillion hosts (though see the randomness note in *8.2.1 Interface identifiers*). If the low-order 64 bits of a host's IPv6 address are chosen with sufficient randomness, finding the host by probing is virtually impossible; see exercise 6.0.

As for nuisance, NAT has always broken protocols that involve negotiation of new connections (*eg* TFTP, FTP, or SIP, used by VoIP); IPv6 should make these much easier to manage.

8.9 ICMPv6

RFC 4443 defines an updated version of the ICMP protocol for IPv6. As with the IPv4 version, messages are identified by 8-bit type and code (subtype) fields, making it reasonably easy to add new message formats. We have already seen the ICMP messages that make up Neighbor Discovery (*8.6 Neighbor Discovery*).

Unlike ICMPv4, ICMPv6 distinguishes between informational and error messages by the first bit of the type field. Unknown informational messages are simply dropped, while unknown error messages must be handed off, if possible, to the appropriate upper-layer process. For example, "[UDP] port unreachable" messages are to be delivered to the UDP sender of the undeliverable packet.

ICMPv6 includes an IPv6 version of Echo Request / Echo Reply, upon which the "ping6" command (*8.12.1 ping6*) is based; unlike with IPv4, arriving IPv6 echo-reply messages are delivered to the process that generated the corresponding echo request. The base ICMPv6 specification also includes formats for the error conditions below; this list is somewhat cleaner than the corresponding ICMPv4 list:

Destination Unreachable

In this case, one of the following numeric codes is returned:

0. **No route to destination**, returned when a router has no next_hop entry.
1. **Communication with destination administratively prohibited**, returned when a router *has* a next_hop entry, but declines to use it for policy reasons. Codes 5 and 6, below, are special cases of this situation; these more-specific codes are returned when appropriate.
2. **Beyond scope of source address**, returned when a router is, for example, asked to route a packet to a global address, but the return address is not, *eg* is unique-local. In IPv4, when a host with a private address attempts to connect to a global address, NAT is almost always involved.
3. **Address unreachable**, a catchall category for routing failure not covered by any other message. An example is if the packet was successfully routed to the last_hop router, but Neighbor Discovery failed to find a LAN address corresponding to the IPv6 address.
4. **Port unreachable**, returned when, as in ICMPv4, the destination host does not have the requested UDP port open.
5. **Source address failed ingress/egress policy**, see code 1.
6. **Reject route to destination**, see code 1.

Packet Too Big

This is like ICMPv4's "Fragmentation Required but DontFragment flag set"; IPv6 however has no router-based fragmentation.

Time Exceeded

This is used for cases where the Hop Limit was exceeded, and also where *source*-based fragmentation was used and the fragment-reassembly timer expired.

Parameter Problem

This is used when there is a malformed entry in the IPv6 header, an unrecognized Next Header type, or an unrecognized IPv6 option.

_node information:

8.9.1 Node Information Messages

ICMPv6 also includes **Node Information** (NI) Messages, defined in **RFC 4620**. One form of NI query allows a host to be asked directly for its name; this is accomplished in IPv4 via reverse-DNS lookups (*7.8.2 Other DNS Records*). Other NI queries allow a host to be asked for its other IPv6 addresses, or for its IPv4 addresses. Recipients of NI queries may be configured to refuse to answer.

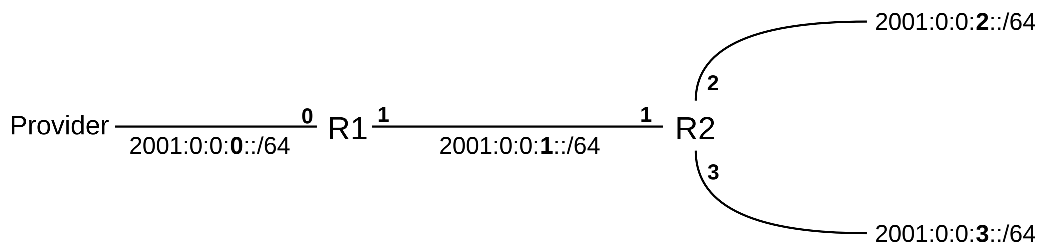
8.10 IPv6 Subnets

In the IPv4 world, network managers sometimes struggle to divide up a limited address space into a pool of appropriately sized subnets. In IPv6, this is much simpler: all subnets are of size /64, following the guidelines set out in *8.3 Network Prefixes*.

There is one common exception: **RFC 6164** permits the use of 127-bit prefixes at each end of a point-to-point link. The 128th bit is then 0 at one end and 1 at the other.

A site receiving from its provider an address prefix of size /56 can assign up to 256 /64 subnets. As with IPv4, the reasons for IPv6 subnetting are to join incompatible LANs, to press intervening routers into service as inter-subnet firewalls, or otherwise to separate traffic.

The diagram below shows a site with an external prefix of 2001::/62, two routers R1 and R2 with interfaces numbered as shown, and three internal LANS corresponding to three subnets 2001:0:0:**1**::/64, 2001:0:0:**2**::/64 and 2001:0:0:**3**::/64. The subnet 2001:0:0:**0**::/64 (2001::/64) is used to connect to the provider.



Interface 0 of R1 would be assigned an address from the /64 block 2001:0:0:**0**/64, perhaps 2001::2.

R1 will announce over its interface 1 – via router advertisements – that it has a *route* to ::/0, that is, it has the default route. It will also advertise via interface 1 the on-link prefix 2001:0:0:1::/64.

R2 will announce via interface 1 its routes to 2001:0:0:2::/64 and 2001:0:0:3::/64. It will also announce the default route on interfaces 2 and 3. On interface 2 it will advertise the on-link prefix 2001:0:0:2::/64, and on interface 3 the prefix 2001:0:0:3::/64. It could also, as a backup, advertise prefix 2001:0:0:1::/64 on its interface 1. On each subnet, only the subnet's on-link prefix is advertised.

Even a residential customer with only, say, two hosts and a router needs more than a single /64 address block. The customer's router has two interfaces, and these must be on different subnets. Two disjoint /64 blocks should work as well as a single /63, though would increase the size of the ISP's routing table.

In theory it is possible to squeeze a site with subnets onto a single /64 by converting the site's main router to a switch; all the customer's hosts now connect on an equal footing to the ISP. But this means abandoning any chance to use the router as a firewall, as described in *8.8 Globally Exposed Addresses*. For most users, this is too risky.

8.11 Using IPv6 and IPv4 Together

In this section we will assume that IPv6 connectivity exists at a site; if it does not, see *8.13 IPv6 Connectivity via Tunneling*.

If IPv6 coexists on a client machine with IPv4, in a so-called **dual-stack** configuration, which is used? If the client wants to connect using TCP to an IPv4-only website (or to some other network service), there is no choice. But what if the remote site also supports both IPv4 and IPv6?

The first step is the **DNS lookup**, triggered by the application's call to the appropriate address-lookup library procedure; in the Java stalk example of *11.1.3.3 The Client* we use `InetAddress.getByName()`. In the C language, address lookup is done with `getaddrinfo()` or (the now-deprecated) `gethostbyname()`. The DNS system on the client then contacts its DNS resolver and asks for the appropriate address record corresponding to the server name.

For IPv4 addresses, DNS maintains so-called "A" records, for "Address". The IPv6 equivalent is the "AAAA" record, for "Address four times longer". A dual-stack machine usually requests both. The Internet Draft draft-vavrusa-dnsop-aaaa-for-free proposes that, whenever a DNS server delivers an IPv4 A record, it also includes the corresponding AAAA record, much as IPv4 CNAME records are sent with piggybacked corresponding A records (*7.8.1 nslookup*). The DNS requests are sent to the client's pre-configured DNS-resolver address (probably set via DHCP).

IPv6 and this book

This book is, as of April 2015, available via IPv6. Within the `cs.luc.edu` DNS zone are defined the following:

- intronetworks: both A and AAAA records
- intronetworks6: AAAA records only
- intronetworks4: A records only

DNS itself can run over either IPv4 or IPv6. A DNS server (authoritative nameserver or just resolver) using only IPv4 can answer IPv6 AAAA-record queries, and a DNS server using only IPv6 can answer IPv4 A-record queries. Ideally each nameserver would eventually support both IPv4 and IPv6 for all queries, though it is common for hosts with newly enabled IPv6 connectivity to continue to use IPv4-only resolvers. See **RFC 4472** for a discussion of some operational issues.

Here is an example of DNS requests for A and AAAA records made with the `nslookup` utility from the command line. (In this example, the DNS resolver was contacted using IPv4.)

nslookup -query=A facebook.com

Name: facebook.com

Address: 173.252.120.6

nslookup -query=AAAA facebook.com

facebook.com has AAAA address 2a03:2880:2130:cf05:**face**:**b00c**:0:1

A few sites have IPv6-only DNS names. If the DNS query returns only an AAAA record, IPv6 must be used. One example in 2015 is ipv6.google.com. In general, however, IPv6-only names such as this are recommended only for diagnostics and testing. The primary DNS names for IPv4/IPv6 sites should have both types of DNS records, as in the Facebook example above (and as for google.com).

Java getByName()

The Java `getByName()` call may *not* abide by system-wide **RFC 6742**-style preferences; the Java Networking Properties documentation (2015) states that "the default behavior is to prefer using IPv4 addresses over IPv6 ones". This can be changed by setting the system property `java.net.preferIPv6Addresses` to `true`, using `System.setProperty()`.

If the client application uses a library call like Java's `InetAddress.getByName()`, which returns a *single* IP address, the client will then attempt to connect to the address returned. If an IPv4 address is returned, the connection will use IPv4, and similarly with IPv6. If an IPv6 address is returned and IPv6 connectivity is not working, then the connection will fail.

For such an application, the DNS resolver library thus effectively makes the IPv4-or-IPv6 decision. **RFC 6724**, which we encountered above in *8.7.2 Stateless Autoconfiguration (SLAAC)*, provides a configuration mechanism, through a small table of IPv6 prefixes and **precedence** values such as the following.

prefix	precedence	
::1/128	50	IPv6 loopback
::/0	40	"default" match
2002::/16	30	6to4 address; see sidebar in *8.13 IPv6 Connectivity via Tunneling*
::ffff:0:0/96	10	Matches embedded IPv4 addresses; see *8.3 Network Prefixes*
fc00::/7	3	unique-local plus reserved; see *8.3 Network Prefixes*

An address is assigned a precedence by looking it up in the table, using the longest-match rule (*10.1 Classless Internet Domain Routing: CIDR*); a list of addresses is then sorted in decreasing order of precedence. There is no entry above for link-local addresses, but by default they are ranked below global addresses. This can be changed by including the link-local prefix fe80::/64 in the above table and ranking it higher than, say, ::/0.

The default configuration is generally to prefer IPv6 if IPv6 is available; that is, if an interface has an IPv6

address that is (or should be) globally routable. Given the availability of both IPv6 and IPv4, a preference for IPv6 is implemented by assigning the prefix ::/0 – matching general IPv6 addresses – a higher precedence than that assigned to the IPv4-specific prefix ::ffff:0:0/96. This is done in the table above.

Preferring IPv6 does not always work out well, however; many hosts have IPv6 connectivity through tunneling that may be slow, limited or outright down. The precedence table can be changed to prefer IPv4 over IPv6 by raising the precedence for the prefix ::ffff:0.0.0.0/96 to a value higher than that for ::/0. Such system-wide configuration is usually done on linux hosts by editing `/etc/gai.conf` and on Windows via the `netsh` command; for example, `netsh interface ipv6 show prefixpolicies`.

We can see this systemwide IPv4/IPv6 preference in action using OpenSSH (see *22.10.1 SSH*), between two systems that each support both IPv4 and IPv6 (the remote system here is intronetworks.cs.luc.edu). With the IPv4-matching prefix precedence set high, connection is automatically via IPv4:

> /etc/gai.conf: `precedence ::ffff:0:0/96 100`
>
> ssh: `Connecting to intronetworks.cs.luc.edu [162.216.18.28] ...`

With the IPv4-prefix precedence set low, new connections use IPv6:

> /etc/gai.conf: `precedence ::ffff:0:0/96 10`
>
> ssh: `Connecting to intronetworks.cs.luc.edu [2600:3c03::f03c:91ff:fe69:f438] ...`

Applications can also use a DNS-resolver call that returns a *list* of all addresses matching a given hostname. (Often this list will have just two entries, for the IPv4 and IPv6 addresses, though round-robin DNS (*7.8 DNS*) can make the list much longer.) The C language `getaddrinfo()` call returns such a list, as does the Java `InetAddress.getAllByName()`. The **RFC 6724** preferences then determine the relative order of IPv4 and IPv6 entries in this list.

If an application requests such a list of all addresses, probably the most common strategy is to try each address in turn, according to the system-provided order. In the example of the previous paragraph, OpenSSH does in fact request a list of addresses, using `getaddrinfo()`, but, according to its source code, tries them in order and so usually connects to the first address on the list, that is, to the one preferred by the **RFC 6724** rules. Alternatively, an application might implement user-specified configuration preferences to decide between IPv4 and IPv6, though user interest in this tends to be limited (except, perhaps, by readers of this book).

The "**Happy Eyeballs**" algorithm, **RFC 8305**, offers a more nuanced strategy for deciding whether an application should connect using IPv4 or IPv6. Initially, the client might try the IPv6 address (that is, will send TCP SYN to the IPv6 address, *12.3 TCP Connection Establishment*). If that connection does not succeed within, say, 250 ms, the client would try the IPv4 address. 250 ms is barely enough time for the TCP handshake to succeed; it does not allow – and is not meant to allow – sufficient time for a retransmission. The client falls back to IPv4 well before the failure of IPv6 is certain.

IPv6 servers

As of 2015, the list of websites supporting IPv6 was modest, though the number has crept up since then. Some sites, such as apple.com and microsoft.com, require the "www" prefix for IPv6 availability. Networking providers are more likely to be IPv6-available. Sprint.com gets an honorable mention for having the shortest IPv6 address I found: 2600::aaaa.

A Happy-Eyeballs client is also encouraged to **cache** the winning protocol, so for the next connection the client will attempt to use only the protocol that was successful before. The cache timeout is to be on the order of 10 minutes, so that if IPv6 connectivity failed and was restored then the client can resume using it with only moderate delay. Unfortunately, if the Happy Eyeballs mechanism is implemented at the *application* layer, which is often the case, then the scope of this cache may be limited to the particular application.

As IPv6 becomes more mainstream, Happy Eyeballs implementations are likely to evolve towards placing greater confidence in the IPv6 option. One simple change is to increase the time interval during which the client waits for an IPv6 response before giving up and trying IPv4.

We can test for the Happy Eyeballs mechanism by observing traffic with WireShark. As a first example, we imagine giving our client host a unique-local IPv6 address (in addition to its automatic link-local address); recall that unique-local addresses are not globally routable. If we now were to connect to, say, google.com, and monitor the traffic using WireShark, we would see a DNS AAAA query (IPv6) for "google.com" followed immediately by a DNS A query (IPv4). The subsequent TCP SYN, however, would be sent only to the IPv4 address: the client host would know that its IPv6 unique-local address is not routable, and it is not even tried.

Next let us change the IPv6 address for the client host to `2000:dead:beef:cafe::2`, through manual configuration (*8.12.3 Manual address configuration*), and *without providing an actual IPv6 connection*. (We also manually specify a fake default router.) This address is part of the 2000::/3 block, and is *supposed* to be globally routable.

We now try two connections to `google.com`, TCP port 80. The first is via the Firefox browser.

*tap0 [Wireshark 1.12.1 (Git Rev Unknown from unknown)]

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help

Filter: dns or tcp.flags.syn == 1 | Expression... Clear Apply Save

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.2.5.19	147.126.68.1	DNS	74	Standard query 0xe6fe AAAA www.google.com
2	0.035038000	147.126.68.1	10.2.5.19	DNS	238	Standard query response 0xe6fe AAAA 2607:
3	0.035407000	10.2.5.19	147.126.68.1	DNS	74	Standard query 0xa324 A www.google.com
4	0.070073000	147.126.68.1	10.2.5.19	DNS	226	Standard query response 0xa324 A 216.58.2
8	0.071422000	2000:dead:beef:cafe::2	2607:f8b0:4009:80b::2004	TCP	94	38732→443 [SYN] Seq=0 Win=5760 Len=0 MSS=1
9	0.320561000	10.2.5.19	147.126.68.1	DNS	74	Standard query 0x1340 A www.google.com
10	0.349995000	147.126.68.1	10.2.5.19	DNS	226	Standard query response 0x1340 A 216.58.2
11	0.350343000	10.2.5.19	216.58.216.100	TCP	74	32879→443 [SYN] Seq=0 Win=5840 Len=0 MSS=1
12	0.375167000	216.58.216.100	10.2.5.19	TCP	74	443→32879 [SYN, ACK] Seq=0 Ack=1 Win=42540

We see two DNS queries, AAAA and A, in packets 1-4, followed by the first attempt (highlighted in orange) at T=0.071 to negotiate a TCP connection via IPv6 by sending a TCP SYN packet (*12.3 TCP Connection Establishment*) to the `google.com` IPv6 address 2607:f8b0:4009:80b::200e. Only 250 ms later, at T=0.321, we see a second DNS A-query (IPv4), followed by an ultimately successful connection attempt using IPv4 starting at T=0.350. This particular version of Firefox, in other words, has implemented the Happy Eyeballs dual-stack mechanism.

Now we try the connection using the previously mentioned OpenSSH application, using `-p 80` to connect to port 80. (This example was generated somewhat later; DNS now returns 2607:f8b0:4009:807::1004 as google.com's IPv6 address.)

Filter: dns or tcp.flags.syn == 1 Expression... Clear Apply Save

No.	Time	Source	Destination	Protocol	Length	New Column
4	0.000110000	10.2.5.19	147.126.68.1	DNS	70	Standard query 0x71e2 AAAA google.com
5	0.045496000	147.126.68.1	10.2.5.19	DNS	234	Standard query response 0x71e2 AAAA 2607:f8b
6	0.045776000	10.2.5.19	147.126.68.1	DNS	70	Standard query 0x4f43 A google.com
7	0.077999000	147.126.68.1	10.2.5.19	DNS	382	Standard query response 0x4f43 A 173.194.46.1
8	0.078490000	2000:dead:beef:cafe::2	2607:f8b0:4009:807::1004	TCP	94	40303→80 [SYN] Seq=0 Win=5760 Len=0 MSS=1440
9	3.077154000	2000:dead:beef:cafe::2	2607:f8b0:4009:807::1004	TCP	94	[TCP Retransmission] 40303→80 [SYN] Seq=0 Win=
14	9.078699000	2000:dead:beef:cafe::2	2607:f8b0:4009:807::1004	TCP	94	[TCP Retransmission] 40303→80 [SYN] Seq=0 Win=
19	21.08024500	10.2.5.19	173.194.46.105	TCP	74	40045→80 [SYN] Seq=0 Win=5840 Len=0 MSS=1460
20	21.13292400	173.194.46.105	10.2.5.19	TCP	74	80→40045 [SYN, ACK] Seq=0 Ack=1 Win=42540 Len=

We see two DNS queries, AAAA and A, in packets numbered 4 and 6 (pale blue); these are made by the client from its IPv4 address 10.2.5.19. Half a millisecond after the A query returns (packet 7), the client sends a TCP SYN packet to google.com's IPv6 address; this packet is highlighted in orange. This SYN packet is retransmitted 3 seconds and then 9 seconds later (in black), to no avail. After 21 seconds, the client gives up on IPv6 and attempts to connect to `google.com` at its IPv4 address, 173.194.46.105; this connection (in green) is successful. The long delay shows that Happy Eyeballs was not implemented by OpenSSH, which its source code confirms.

(The host initiating the connections here was running Ubuntu 10.04 LTS, from 2010. The ultimately failing TCP connection gives up after three tries over only 21 seconds; newer systems make more tries and take much longer before they abandon a connection attempt.)

8.12 IPv6 Examples Without a Router

In this section we present a few IPv6 experiments that can be done without an IPv6 connection and without even an IPv6 router. Without a router, we cannot use SLAAC or DHCPv6. We will instead use link-local addresses, which require the specification of the interface along with the address, and manually configured unique-local (*8.3 Network Prefixes*) addresses. One practical problem with link-local addresses is that application documentation describing how to include a specification of the interface is sometimes sparse.

8.12.1 ping6

The IPv6 analogue of the familiar `ping` command, used to send ICMPv6 Echo Requests, is `ping6` on linux and Mac systems and `ping -6` on Windows. The ping6 command supports an option to specify the interface; *eg* `-I eth0`; as noted above, this is mandatory when sending to link-local addresses. Here are a few ping6 examples:

ping6 ::1: This pings the host's loopback address; it should always work.

ping6 -I eth0 ff02::1: This pings the all-nodes multicast group on interface `eth0`. Here are two of the answers received:

- 64 bytes from fe80::3e97:eff:fe2c:2beb (this is the host I am pinging *from*)
- 64 bytes from fe80::2a0:ccff:fe24:b0e4 (a second linux host)

Answers were also received from a Windows machine and an Android phone. A VoIP phone – on the same subnet but supporting IPv4 only – remained mute, despite VoIP's difficulties with IPv4 NAT that would be avoided with IPv6. In lieu of the interface option `-I eth0`, the "zone-identifier" syntax **ping6 ff02::1%eth0** also usually works; see the following section.

ping6 -I eth0 fe80::2a0:ccff:fe24:b0e4: This pings the link-local address of the second linux host answering the previous query; again, the **%eth0** syntax should also work. The destination interface identifier here uses the now-deprecated EUI-64 format; note the "ff:fe" in the middle. Also note the flipped seventh bit of the two bytes 02a0; the destination has Ethernet address 00:a0:cc:24:b0:e4.

8.12.2 TCP connections using link-local addresses

The next experiment is to create a TCP connection. Some commands, like ping6 above, may provide for a way of specifying the interface as a command-line option. Failing that, **RFC 4007** defines the concept of a **zone identifier** that is appended to the IPv6 address, separated from it by a "%" character, to specify the link involved. On linux systems the zone identifier is most often the interface name, *eg* `eth0` or `ppp1`. Numeric zone identifiers are also used, in which case it represents the number of the particular interface in some designated list and can be called the **zone index**. On Windows systems the zone index for an interface can often be inferred from the output of the `ipconfig` command, which should include it with each link-local address. The use of zone identifiers is often restricted to literal (numeric) IPv6 addresses, perhaps because there is little demand for symbolic link-local addresses.

The following link-local address with zone identifier creates an ssh connection to the second linux host in the example of the preceding section:

`ssh fe80::2a0:ccff:fe24:b0e4`**%eth0**

That the ssh service is listening for IPv6 connections can be verified on that host by `netstat -a | grep -i tcp6`. That the ssh connection actually *used* IPv6 can be verified by, say, use of a network sniffer like WireShark (for which the filter expression `ipv6` or `ip.version == 6` is useful). If the connection fails, but ssh works for IPv4 connections and shows as listening in the tcp6 list from the netstat command, a firewall-blocked port is a likely suspect.

8.12.3 Manual address configuration

The use of manually configured addresses is also possible, for either global or unique-local (*ie* not connected to the Internet) addresses. However, without a router there can be no Prefix Discovery, *8.6.2 Prefix Discovery*, and this may create subtle differences.

The first step is to pick a suitable prefix; in the example below we use the unique-local prefix fd37:beef:cafe::/64 (though this particular prefix does *not* meet the randomness rules for unique-local prefixes). We could also use a globally routable prefix, but here we do not want to mislead any hosts about reachability.

Without a router as a source of Router Advertisements, we need some way to specify both the prefix and the prefix *length*; the latter can be thought of as corresponding to the IPv4 subnet mask. One might be forgiven for imagining that the default prefix length would be /64, given that this is the only prefix length generally allowed (*8.3 Network Prefixes*), but this is often not the case. In the commands below, the prefix length is included at the end as the /64. This usage is just slightly peculiar, in that in the IPv4 world the slash notation is most often used only with true prefixes, with all bits zero beyond the slash length. (The linux `ip` command also uses the slash notation in the sense here, to specify an IPv4 subnet mask, *eg* 10.2.5.37/24. The `ifconfig` and Windows `netsh` commands specify the IPv4 subnet mask the traditional way, *eg* 255.255.255.0.)

Hosts will usually assume that a prefix configured this way with a length represents an **on-link** prefix, meaning that neighbors sharing the prefix are reachable directly via the LAN.

We can now assign the low-order 64 bits manually. On linux this is done with:

- host1: `ip -6 address add fd37:beef:cafe::1/64 dev eth0`
- host2: `ip -6 address add fd37:beef:cafe::2/64 dev eth0`

Macintosh systems can be configured similarly except the name of the interface is probably `en0` rather than `eth0`. On Windows systems, a typical IPv6-address-configuration command is

```
netsh interface ipv6 add address "Local Area Connection" fd37:beef:cafe::1/64
```

Now on host1 the command

```
ssh fd37:beef:cafe::2
```

should create an ssh connection to host2, again assuming `ssh` on host2 is listening for IPv6 connections. Because the addresses here are not link-local, /etc/host entries may be created for them to simplify entry.

Assigning IPv6 addresses manually like this is *not* recommended, except for experiments.

On a LAN not connected to the Internet and therefore with no actual routing, it is nonetheless possible to start up a Router Advertisement agent (*8.6.1 Router Discovery*), such as **radvd**, with a manually configured /64 prefix. The RA agent will include this prefix in its advertisements, and reasonably modern hosts will then construct full addresses for themselves from this prefix using SLAAC. IPv6 can then be used within the LAN. If this is done, the RA agent should also be configured to announce only a meaningless route, such as ::/128, or else nodes may falsely believe the RA agent is providing full Internet connectivity.

8.13 IPv6 Connectivity via Tunneling

The best option for IPv6 connectivity is native support by ones ISP. In such a situation ones router should be sending out Router Advertisement messages, and from these all the hosts should discover how to reach the IPv6 Internet.

If native IPv6 support is not forthcoming, however, a short-term option is to connect to the IPv6 world using **packet tunneling** (less often, some other VPN mechanism is used). **RFC 4213** outlines the common **6in4** strategy of simply attaching an IPv4 header to the front of the IPv6 packet; it is very similar to the IPv4-in-IPv4 encapsulation of *7.13.1 IP-in-IP Encapsulation*.

There are several available providers for this service; they can be found by searching for "IPv6 tunnel broker". Some tunnel brokers provide this service at no charge.

6in4, 6to4

6**in**4 tunneling should not be confused with 6**to**4 tunneling, which uses the same encapsulation as 6in4 but which constructs a site's IPv6 prefix by embedding its IPv4 address: a site with IPv4 address **129**.3.5.7 gets IPv6 prefix 2002:**81**03:0507::/48 (129 decimal = 0x81). See **RFC 3056**. There is also a 6**over**4, **RFC 2529**.

The basic idea behind 6in4 tunneling is that the tunnel broker allocates you a /64 prefix out of its own address block, and agrees to create an IPv4 tunnel to you using 6in4 encapsulation. All your IPv6 traffic from the Internet is routed by the tunnel broker to you via this tunnel; similarly, IPv6 packets from your site reach the outside world using this same tunnel. The tunnel, in other words, is your link to an IPv6 router.

Generally speaking, the MTU of the tunnel must be at least 20 bytes less than the MTU of the physical interface, to allow space for the header. At the near end this requires a local configuration change; tunnel brokers often provide a way for users to set the MTU at the far end. Practical MTU values vary from a mandatory IPv6 minimum of 1280 to the Ethernet maximum of 1500–20 = 1480.

Setting up the tunnel does not involve creating a stateful connection. All that happens is that the tunnel client (*ie* your endpoint) and the broker record each other's IPv4 addresses, and agree to accept encapsulated IPv6 packets from one another provided these two endpoint addresses are used as source and destination. The tunnel at the client end is represented by an appropriate "virtual network interface", *eg* `sit0` or `gif0` or `IP6Tunnel`. Tunnel providers generally supply the basic commands necessary to get the tunnel interface configured and the MTU set.

Once the tunnel is created, the tunnel interface at the client end must be assigned an IPv6 address and then a (default) route. We will assume that the /64 prefix for the broker-to-client link is 2001:470:0:10::/64, with the broker at 2001:470:0:10::**1** and with the client to be assigned the address 2001:470:0:10::**2**. The address and route are set up on the client with the following commands (linux/mac/Windows respectively; interface names may vary, and some commands assume the interface represents a point-to-point link):

```
ip addr add 2001:470:0:10::2/64 dev sit1
ip route add ::/0 dev sit1

ifconfig gif0 inet6 2001:470:0:10::2 2001:470:0:10::1 prefixlen 128
route -n add -inet6 default 2001:470:0:10::1

netsh interface ipv6 add address IP6Tunnel 2001:470:0:10::2
netsh interface ipv6 add route ::/0 IP6Tunnel 2001:470:0:10::1
```

At this point the tunnel client should have full IPv6 connectivity! To verify this, one can use `ping6`, or visit IPv6-only versions of websites (*eg* intronetworks6.cs.luc.edu), or visit IPv6-identifying sites such as IsMyIPv6Working.com. Alternatively, one can often install a browser plugin to at least make visible whether IPv6 is used. Finally, one can use `netcat` with the `-6` option to force IPv6 use, following the HTTP example in *12.6.2 netcat again*.

There is one more potential issue. If the tunnel client is behind an IPv4 NAT router, that router must deliver arriving encapsulated 6in4 packets correctly. This can sometimes be a problem; encapsulated 6in4 packets are at some remove from the TCP and UDP traffic that the usual consumer-grade NAT router is primarily designed to handle. Careful study of the router forwarding settings may help, but sometimes the only fix is a newer router. A problem is particularly likely if two different inside clients attempt to set up tunnels simultaneously; see *7.13.1 IP-in-IP Encapsulation*.

8.13.1 IPv6 firewalls

It is strongly recommended that an IPv6 host **block new inbound connections** over its IPv6 interface (*eg* the tunnel interface), much as an IPv4 NAT router would do. Exceptions may be added as necessary for essential services (such as ICMPv6). Using the linux `ip6tables` firewall command, with IPv6-tunneled

interface `sit1`, this might be done with the following:

```
ip6tables --append INPUT --in-interface sit1 --protocol icmpv6 --jump ACCEPT
ip6tables --append INPUT --in-interface sit1 --match conntrack --ctstate ESTABLISHED,RELATE
ip6tables --append INPUT --in-interface sit1 --jump DROP
```

At this point the firewall should be tested by attempting to access inside hosts from the outside. At a minimum, ping6 from the outside to any global IPv6 address of any inside host should fail if the ICMPv6 exception above is removed (and should succeed if the ICMPv6 exception is restored). This can be checked by using any of several websites that send pings on request; such sites can be found by searching for "online ipv6 ping". There are also a few sites that will run a remote IPv6 TCP port scan; try searching for "online ipv6 port scan". See also exercise 7.0.

8.13.2 Setting up a router

The next step, if desired, is to set up the tunnel endpoint as a router, so other hosts at the client site can also enjoy IPv6 connectivity. For this we need a second /64 prefix; we will assume this is 2001:470:0:**2**0::/64 (note this is not an "adjacent" /64; the two /64 prefixes cannot be merged into a /63). Let R be the tunnel endpoint, with `eth0` its LAN interface, and let A be another host on the LAN.

We will use the linux `radvd` package as our Router Advertisement agent (*8.6.1 Router Discovery*). In the `radvd.conf` file, we need to say that we want the LAN prefix 2001:470:0:20::/64 advertised as on-link over interface eth0:

```
interface eth0 {
        ...
        prefix 2001:470:0:20::/64
        {
                AdvOnLink on;           # advertise this prefix as on-link
                AdvAutonomous on;       # allows SLAAC with this prefix
        };
};
```

If `radvd` is now started, other LAN hosts (*eg* A) will automatically get the prefix (and thus a full SLAAC address). `Radvd` will automatically share R's default route (::/0), taking it not from the configuration file but from R's routing table. (It may still be necessary to manually configure the IPv6 address of R's eth0 interface, *eg* as 2001:470:0:20::**1**.)

On the author's version of host A, the IPv6 route is now (with some irrelevant attributes not shown)

```
default via fe80::2a0:ccff:fe24:b0e4 dev eth0
```

That is, host A routes to R via the latter's **link-local** address, always guaranteed on-link, rather than via the subnet address.

If `radvd` or its equivalent is not available, the manual approach is to assign R and A each a /64 address:

On host R: `ip -6 address add 2001:470:0:20::1/64 dev eth0`

On host A: `ip -6 address add 2001:470:0:20::2/64 dev eth0`

Because of the "/64" here (*8.12.3 Manual address configuration*), R and A understand that they can reach each other via the LAN, and do so. Host A also needs to be told of the default route via R:

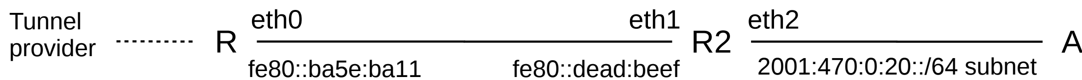
On host A: `ip -6 route add ::/0 via 2001:470:0:10::1 dev eth0`

Here we use the subnet address of R, but we could have used R's link-local address as well.

It is likely that A's `eth0` will also need its MTU configured, so that it matches that of R's virtual tunnel interface (which, recall, should be at least 20 bytes less than the MTU of R's physical outbound interface).

8.13.2.1 A second router

Now let us add a second router R2, as in the diagram below. The R—R2 link is via a separate Ethernet LAN, not a point-to-point link. The LAN with A is, as above, subnet 2001:470:0:20::/64.



In this case, it is R2 that needs to run the Router Advertisement agent (*eg* radvd). If this were an IPv4 network, the interfaces eth0 and eth1 on the R—R2 link would need IPv4 addresses from some new subnet (though the use of private addresses is an option). We can't use unnumbered interfaces (*7.12 Unnumbered Interfaces*), because the R—R2 connection is not a point-to-point link.

But with IPv6, we can configure the R—R2 routing to use only link-local addresses. Let us assume for mnemonic convenience these are as follows:

R's eth0: `fe80::ba5e:ba11`

R2's eth1: `fe80::dead:beef`

R2's forwarding table will have a default route with next_hop `fe80::ba5e:ba11` (R). Similarly, R's forwarding table will have an entry for destination subnet 2001:470:0:20::/64 with next_hop `fe80::dead:beef` (R2). Neither eth0 nor eth1 needs any other IPv6 address.

R2's eth2 interface will likely need a global IPv6 address, *eg* 2001:470:0:20::1 again. Otherwise R2 may not be able to determine that its eth2 interface is in fact connected to the 2001:470:0:20::/64 subnet.

One advantage of not giving eth0 or eth1 global addresses is that it is then impossible for an outside attacker to reach these interfaces directly. It also saves on subnets, although one hopes with IPv6 those are not in short supply. All routers at a site are likely to need, for management purposes, an IP address reachable throughout the site, but this does not have to be globally visible.

8.14 IPv6-to-IPv4 Connectivity

What happens if you switch to IPv6 completely, perhaps because your ISP (or phone provider) has run out of IPv4 addresses? Some of the time – hopefully more and more of the time – you will only need to talk to IPv6 servers. For example, the DNS names `facebook.com` and `google.com` each correspond to an IPv4 address, but also to an IPv6 address (above). But what do you do if you want to reach an IPv4-only server? Such servers are expected to continue operating for a long time to come. It is necesary to have some sort of centralized IPv6-to-IPv4 **translator**.

An early strategy was NAT-PT (**RFC 2766**). The translator was assigned a /96 prefix. The IPv6 host would append to this prefix the 32-bit IPv4 address of the destination, and use the resulting address to contact the IPv4 destination. Packets sent to this address would be delivered via IPv6 to the translator, which would translate the IPv6 header into IPv4 and then send the translated packet on to the IPv4 destination. As in IPv4 NAT (*7.7 Network Address Translation*), the reverse translation will typically involve TCP port numbers to resolve ambiguities. This approach requires the IPv6 host to be aware of the translator, and is limited to TCP and UDP (because of the use of port numbers). Due to these and several other limitations, NAT-PT was formally deprecated in **RFC 4966**.

Do you still have IPv4 service?

As of 2017, several phone providers have switched many of their customers to IPv6 while on their mobile-data networks. The change can be surprisingly inconspicuous. Connections to IPv4-only services still work just fine, courtesy of NAT64. About the only way to tell is to look up the phone's IP address.

The replacement protocol is **NAT64**, documented in **RFC 6146**. This is also based on address translation, and, as such, cannot allow connections initiated from IPv4 hosts to IPv6 hosts. It is, however, transparent to both the IPv6 and IPv4 hosts involved, and is not restricted to TCP (though only TCP, UDP and ICMP are supported by **RFC 6146**). It uses a special DNS variant, DNS64 (**RFC 6147**), as a companion protocol.

To use NAT64, an IPv6 client sends out its ordinary DNS query to find the addresses of the destination server. The DNS resolver (*7.8 DNS*) receiving the request must use DNS64. If the destination has only an IPv4 address, then the DNS resolver will return to the IPv6 client (as an AAAA record) a synthetic IPv6 address consisting of a prefix and the embedded IPv4 address of the server, much as in NAT-PT above (though multiple prefix-length options exist; see **RFC 6052**). The prefix belongs to the actual NAT64 translator; any packet addressed to an IPv6 address starting with the prefix will be delivered to the translator. There is no relationship between the NAT64 translator and the DNS64 resolver beyond the fact that the former's prefix is configured into the latter.

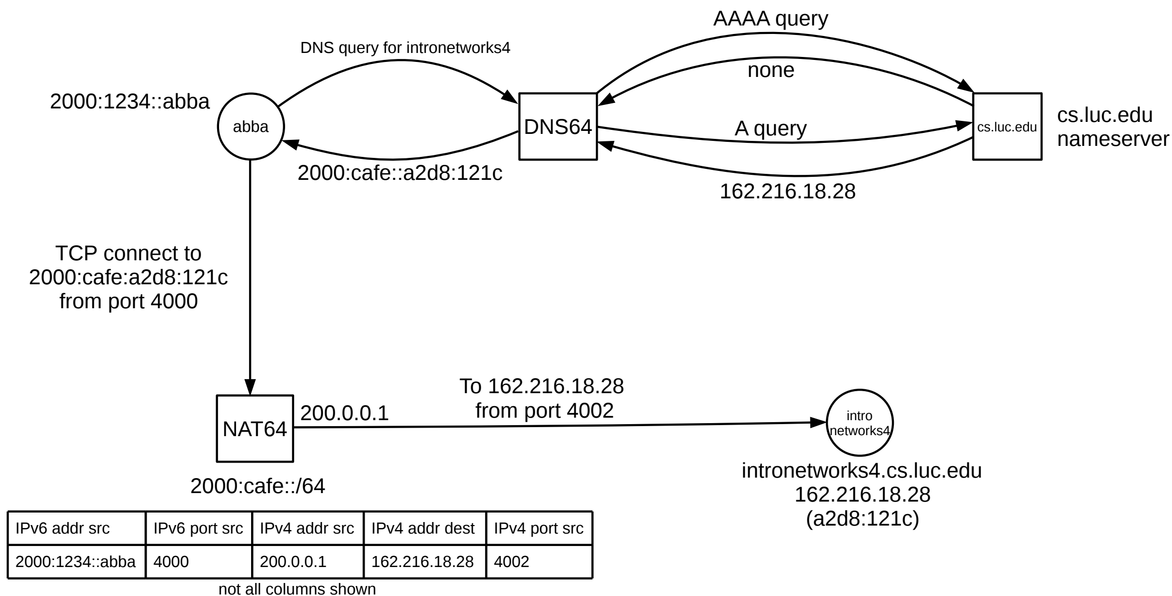
The IPv6 client now uses this synthetic IPv6 address to contact the IPv4 server. Its packets will be routed to the NAT64 translator itself, by virtue of the prefix, much as in NAT-PT. Upon receiving the first packet from the IPv6 client, the NAT64 translator will assign one of its IPv4 addresses to the new connection. As IPv4 addresses are in short supply, this pool of available IPv4 addresses may be small, so NAT64 allows one IPv4 address to be used by many IPv6 clients. To this end, the NAT64 translator will also (for TCP and UDP) establish a port mapping between the incoming IPv6 source port and a port number allocated by the NAT64 to ensure that traffic is uniquely reversable. As with IPv4 NAT, if two IPv6 clients try to contact the same IPv4 server using the same source ports, and are assigned the same NAT64 IPv4 address, then one of the clients will have its port number changed.

If an ICMP query is being sent, the Query Identifier is used in lieu of port numbers. To extend NAT64 to new protocols, an appropriate analog of port numbers must be identified, to allow demultiplexing of multiple connections sharing a single IPv4 address.

After the translation is set up, by creating appropriate table entries, the translated packet is sent on to the IPv4 server address that was embedded in the synthetic IPv6 address. The source address will be the assigned IPv4 address of the translator, and the source port will have been rewritten in accordance with the new port mapping. At this point packets can flow freely between the original IPv6 client and its IPv4 destination, with neither endpoint being aware of the translation (unless the IPv6 client carefully inspects the synthetic

address it receives via DNS64). A timer within the NAT64 translator will delete the association between the IPv6 and IPv4 addresses if the connection is not used for a while.

As an example, suppose the IPv6 client has address 2000:1234::abba, and is trying to reach *intronetworks4.cs.luc.edu* at TCP port 80. It contacts its DNS server, which finds no AAAA record but IPv**4** address 162.216.18.28 (in hex, a2d8:121c). It takes the prefix for its NAT64 translator, which we will assume is 2000:cafe::, and returns the synthetic address 2000:cafe::a2d8:121c.



IPv6 addr src	IPv6 port src	IPv4 addr src	IPv4 addr dest	IPv4 port src
2000:1234::abba	4000	200.0.0.1	162.216.18.28	4002

The IPv6 client now tries to connect to 2000:cafe::a2d8:121c, using source port 4000. The first packet arrives at the NAT64 translator, which assigns the connection the outbound IPv4 address of 200.0.0.1, and reassigns the source port on the IPv4 side to 4002. The new IPv4 packet is sent on to 162.216.18.28. The reply from *intronetworks4.cs.luc.edu* comes back, to ⟨200.0.0.1,4002⟩. The NAT64 translator looks this up and finds that this corresponds to ⟨2000:1234::abba,4000⟩, and forwards it back to the original IPv6 client.

8.15 Epilog

IPv4 has run out of large address blocks, as of 2011. IPv6 has reached a mature level of development. Most common operating systems provide excellent IPv6 support.

Yet conversion has been slow. Many ISPs still provide limited (to nonexistent) support, and inexpensive IPv6 firewalls to replace the ubiquitous consumer-grade NAT routers are just beginning to appear. Time will tell how all this evolves. However, while IPv6 has now been around for twenty years, top-level IPv4 address blocks disappeared much more recently. It is quite possible that this will prove to be just the catalyst IPv6 needs.

8.16 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises.

1.0. Each IPv6 address is associated with a specific solicited-node multicast address.

(a). Explain why, on a typical Ethernet, if the original IPv6 host address was obtained via SLAAC then the LAN multicast group corresponding to the host's solicited-node multicast addresses is likely to be small, in many cases consisting of one host only. (Packet delivery to small LAN multicast groups can be much more efficient than delivery to large multicast groups.)

(b). What steps might a DHCPv6 server take to ensure that, for the IPv6 addresses it hands out, the LAN multicast groups corresponding to the host addresses' solicited-node multicast addresses will be small?

2.0. If an attacker sends a large number of probe packets via IPv4, you can block them by blocking the attacker's IP address. Now suppose the attacker uses IPv6 to launch the probes; for each probe, the attacker changes the low-order 64 bits of the address. Can these probes be blocked efficiently? If so, what do you have to block? Might you also be blocking other users?

3.0. Suppose someone tried to implement ping6 so that, if the address was a link-local address and no interface was specified, the ICMPv6 Echo Request was sent out all non-loopback interfaces. Could the end result be different than conventional ping6 with the correct interface supplied? If so, how likely is this?

4.0. Create an IPv6 ssh connection as in *8.12 IPv6 Examples Without a Router*. Examine the connection's packets using WireShark or the equivalent. Does the TCP handshake (*12.3 TCP Connection Establishment*) look any different over IPv6?

5.0. Create an IPv6 ssh connection using manually configured addresses as in *8.12.3 Manual address configuration*. Again use WireShark or the equivalent to monitor the connection. Is DAD (*8.7.1 Duplicate Address Detection*) used?

6.0. An IPv6 fixed-header is 40 bytes. Taking this as the minimum packet size, how long will it take to send 10^{15} hosts (one quadrillion) probe packets to a site, if the bandwidth is 1 Gbps?

7.0. Suppose host A gets its IPv6 traffic through tunnel provider H, as in *8.13 IPv6 Connectivity via Tunneling*. To improve security, A blocks all packets that are not part of connections it has initiated, and makes no exception for ICMPv6 traffic. H is correctly configured to know the MTU of the A–H link. For (a) and (b), this MTU is 1280, the minimum allowed for IPv6. Much of the Internet, however, allows larger MTU values.

(a). If A attempts to send a larger-than-1280-byte IPv6 packet to remote host B, will A be informed of the resultant failure? Why or why not?

(b). Suppose B attempts to send a larger-than-1280-byte IPv6 packet to A. Will B receive an ICMPv6 `Packet Too Big` message? Why or why not?

(c). Now suppose the MTU of the A–H link is raised to 1400 bytes. Outline a scenario in which A sends a packet of size greater than 1280 bytes to remote host B, the packet is too big to make it all the way to B, and yet A receives no notification of this.

How do IP routers build and maintain their forwarding tables?

Ethernet bridges always have the option of fallback-to-flooding for unknown destinations, so they can afford to build their forwarding tables "incrementally", putting a host into the forwarding table only when that host is first seen as a *sender*. For IP, there is no fallback delivery mechanism: forwarding tables must be built *before* delivery can succeed. While manual table construction is possible, it is not practical.

In the literature it is common to refer to router-table construction as "routing algorithms". We will avoid that term, however, to avoid confusion with the fundamental datagram-forwarding algorithm; instead, we will call these "routing-update algorithms".

The two classes of algorithms we will consider here are **distance-vector** and **link-state**. In the distance-vector approach, often used at smaller sites, routers exchange information with their immediately neighboring routers; tables are built up this way through a sequence of such periodic exchanges. In the link-state approach, routers rapidly propagate information about the state of each link; all routers in the organization receive this link-state information and each one uses it to build and maintain a map of the entire network. The forwarding table is then constructed (sometimes on demand) from this map.

Both approaches assume that consistent information is available as to the **cost** of each link (*eg* that the two routers at opposite ends of each link know this cost, and agree on how the cost is determined). This requirement classifies these algorithms as **interior** routing-update algorithms: the routers involved are internal to a larger organization or other common administrative regime that has an established policy on how to assign link weights. The set of routers following a common policy is known as a **routing domain** or (from the BGP protocol) an **autonomous system**.

The simplest link-weight strategy is to give each link a cost of 1; link costs can also be based on bandwidth, propagation delay, financial cost, or administrative preference value. Careful assignment of link costs often plays a major role in herding traffic onto the faster or "better" links.

In the following chapter we will look at the Border Gateway Protocol, or BGP, in which no link-cost calculations are made. BGP is used to select routes that traverse other organizations, and financial rather than technical factors may therefore play the dominant role in making routing choices.

Generally, all these algorithms apply to IPv6 as well as IPv4, though specific protocols of course may need modification.

Finally, we should point out that from the early days of the Internet, routing was allowed to depend not just on the destination, but also on the "quality of service" (QoS) requested; thus, forwarding table entries are strictly speaking not ⟨destination, next_hop⟩ but rather ⟨destination, QoS, next_hop⟩. Originally, the Type of Service field in the IPv4 header (*7.1 The IPv4 Header*) could be used to specify QoS (often then called ToS). Packets could request low delay, high throughput or high reliability, and could be routed accordingly. In practice, the Type of Service field was rarely used, and was eventually taken over by the DS field and ECN bits. The first three bits of the Type of Service field, known as the precedence bits, remain available, however, and can still be used for QoS routing purposes (see the Class Selector PHB of *20.7 Differentiated Services* for examples of these bits). See also **RFC 2386**.

In much of the following, we are going to ignore QoS information, and assume that routing decisions are based only on the destination. See, however, the first paragraph of *9.5 Link-State Routing-Update*

Algorithm, and also *9.6 Routing on Other Attributes*.

9.1 Distance-Vector Routing-Update Algorithm

Distance-vector is the simplest routing-update algorithm, used by the Routing Information Protocol, or RIP. Version 2 of the protocol is specified in **RFC 2453**.

Routers identify their router neighbors (through some sort of neighbor-discovery mechanism), and add a third column to their forwarding tables representing the total **cost** for delivery to the corresponding destination. These costs are the "distance" of the algorithm name. Forwarding-table entries are now of the form ⟨destination,next_hop,cost⟩.

Costs are administratively assigned to each link, and the algorithm then calculates the total cost to a destination as the sum of the link costs along the path. The simplest case is to assign a cost of 1 to each link, in which case the total cost to a destination will be the number of links to that destination. This is known as the "hopcount" metric; it is also possible to assign link costs that reflect each link's bandwidth, or delay, or whatever else the network administrators wish. Thoughtful cost assignments are a form of traffic engineering and sometimes play a large role in network performance.

At this point, each router then **reports** the ⟨destination,cost⟩ portion of its table to its neighboring routers at regular intervals; these table portions are the "vectors" of the algorithm name. It does not matter if neighbors exchange reports at the same time, or even at the same rate.

Each router also monitors its continued connectivity to each neighbor; if neighbor N becomes unreachable then its reachability cost is set to infinity.

In a real IP network, actual destinations would be *subnets* attached to routers; one router might be directly connected to several such destinations. In the following, however, we will identify all a router's directly connected subnets with the router itself. That is, we will build forwarding tables to reach every *router*. While it is possible that one destination subnet might be reachable by two or more routers, thus breaking our identification of a router with its set of attached subnets, in practice this is of little concern. See exercise 4.0 for an example in which subnets are *not* identified with adjacent routers.

In *18.5 IP Routers With Simple Distance-Vector Implementation* we present a simplified working implementation of RIP using the Mininet network emulator.

9.1.1 Distance-Vector Update Rules

Let A be a router receiving a report $\langle D, c_D \rangle$ from neighbor N at cost c_N. Note that this means A can reach D *via N* with cost $c = c_D + c_N$. A updates its own table according to the following three rules:

1. **New destination**: D is a previously unknown destination. A adds ⟨D,N,c⟩ to its forwarding table.
2. **Lower cost**: D is a known destination with entry $\langle D, M, c_{old} \rangle$, but the new total cost c is less than c_{old}. A switches to the cheaper route, updating its entry for D to ⟨D,N,c⟩. It is possible that M=N, meaning that N is now reporting a cost decrease to D. (If $c = c_{old}$, A ignores the new report; see exercise 5.5.)
3. **Next_hop increase**: A has an existing entry $\langle D, \mathbf{N}, c_{old} \rangle$, and the new total cost c is *greater* than c_{old}. Because this is a cost increase from the neighbor N that A is currently using to reach D, A must incorporate the increase in its table. A updates its entry for D to ⟨D,N,c⟩.

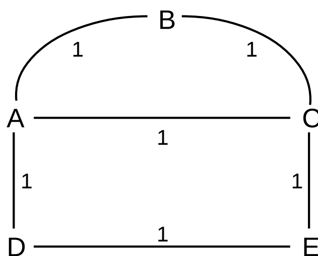
The first two rules are for new destinations and a shorter path to existing destinations. In these cases, the cost to each destination monotonically decreases (at least if we consider all unreachable destinations as being at cost ∞). Convergence is automatic, as the costs cannot decrease forever.

The third rule, however, introduces the possibility of instability, as a cost may also go up. It represents the **bad-news** case, in that neighbor N has learned that some link failure has driven up its own cost to reach D, and is now passing that "bad news" on to A, which routes to D *via* N.

The next_hop-increase case only passes bad news along; the very first cost increase must always come from a router discovering that a neighbor N is unreachable, and thus updating its cost to N to ∞. Similarly, if router A learns of a next_hop increase to destination D from neighbor B, then we can follow the next_hops back until we reach a router C which is either the originator of the cost=∞ report, or which has learned of an alternative route through one of the first two rules.

9.1.2 Example 1

For our first example, no links will break and thus only the first two rules above will be used. We will start out with the network below with empty forwarding tables; all link costs are 1.



After initial neighbor discovery, here are the forwarding tables. Each node has entries only for its directly connected neighbors:

A: ⟨B,B,1⟩ ⟨C,C,1⟩ ⟨D,D,1⟩
B: ⟨A,A,1⟩ ⟨C,C,1⟩
C: ⟨A,A,1⟩ ⟨B,B,1⟩ ⟨E,E,1⟩
D: ⟨A,A,1⟩ ⟨E,E,1⟩
E: ⟨C,C,1⟩ ⟨D,D,1⟩

Now let D report to A; it sends records ⟨A,1⟩ and ⟨E,1⟩. A ignores D's ⟨A,1⟩ record, but ⟨E,1⟩ represents a new destination; A therefore adds ⟨E,D,**2**⟩ to its table. Similarly, let A now report to D, sending ⟨B,1⟩ ⟨C,1⟩ ⟨D,1⟩ ⟨E,2⟩ (the last is the record we just added). D ignores A's records ⟨D,1⟩ and ⟨E,2⟩ but A's records ⟨B,1⟩ and ⟨C,1⟩ cause D to create entries ⟨B,A,2⟩ and ⟨C,A,2⟩. A and D's tables are now, in fact, complete.

Now suppose C reports to B; this gives B an entry ⟨E,C,2⟩. If C also reports to E, then E's table will have ⟨A,C,2⟩ and ⟨B,C,2⟩. The tables are now:

A: ⟨B,B,1⟩ ⟨C,C,1⟩ ⟨D,D,1⟩ ⟨E,D,2⟩
B: ⟨A,A,1⟩ ⟨C,C,1⟩ ⟨E,C,2⟩
C: ⟨A,A,1⟩ ⟨B,B,1⟩ ⟨E,E,1⟩
D: ⟨A,A,1⟩ ⟨E,E,1⟩ ⟨B,A,2⟩ ⟨C,A,2⟩

E: ⟨C,C,1⟩ ⟨D,D,1⟩ ⟨A,C,2⟩ ⟨B,C,2⟩

We have two missing entries: B and C do not know how to reach D. If A reports to B and C, the tables will be complete; B and C will each reach D via A at cost 2. However, the following sequence of reports might also have occurred:

- E reports to C, causing C to add ⟨D,E,2⟩
- C reports to B, causing B to add ⟨D,C,**3**⟩

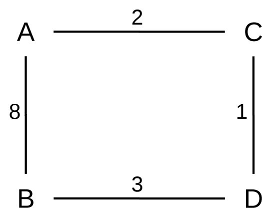
In this case we have 100% reachability but B routes to D via the longer-than-necessary path B–C–E–D. However, one more report will fix this: suppose A reports to B. B will received ⟨D,1⟩ from A, and will update its entry ⟨D,C,3⟩ to ⟨D,A,2⟩.

Note that A routes to E via D while E routes to A via C; this asymmetry was due to indeterminateness in the order of initial table exchanges.

If all link weights are 1, and if each pair of neighbors exchange tables once before any pair starts a second exchange, then the above process will discover the routes in order of length, *ie* the shortest paths will be the first to be discovered. This is not, however, a particularly important consideration.

9.1.3 Example 2

The next example illustrates link weights other than 1. The first route discovered between A and B is the direct route with cost 8; eventually we discover the longer A–C–D–B route with cost 2+1+3=6.



The initial tables are these:

A: ⟨C,C,2⟩ ⟨B,B,8⟩
B: ⟨A,A,8⟩ ⟨D,D,3⟩
C: ⟨A,A,2⟩ ⟨D,D,1⟩
D: ⟨B,B,3⟩ ⟨C,C,1⟩

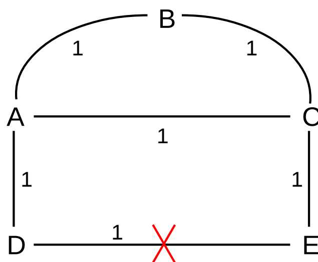
After A and C exchange, A has ⟨D,C,3⟩ and C has ⟨B,A,10⟩. After C and D exchange, C updates its ⟨B,A,10⟩ entry to ⟨B,D,4⟩ and D adds ⟨A,C,3⟩; D receives C's report of ⟨B,10⟩ but ignores it. Now finally suppose B and D exchange. D ignores B's route to A, as it has a better one. B, however, gets D's report ⟨A,3⟩ and updates its entry for A to ⟨A,D,6⟩. At this point the tables are as follows:

A: ⟨C,C,2⟩ ⟨B,B,8⟩ ⟨D,C,3⟩
B: ⟨A,D,6⟩ ⟨D,D,3⟩
C: ⟨A,A,2⟩ ⟨D,D,1⟩ ⟨B,D,4⟩
D: ⟨B,B,3⟩ ⟨C,C,1⟩ ⟨A,C,3⟩

We have two more things to fix before we are done: A has an inefficient route to B, and B has no route to C. The first will be fixed when C reports ⟨B,4⟩ to A; A will replace its route to B with ⟨B,C,6⟩. The second will be fixed when D reports to B; if A reports to B first then B will temporarily add the inefficient route ⟨C,A,10⟩; this will change to ⟨C,D,4⟩ when D's report to B arrives. If we look only at the A–B route, B discovers the lower-cost route to A once, first, C reports to D and, second, *after* that, D reports to B; a similar sequence leads to A's discovering the lower-cost route.

9.1.4 Example 3

Our third example will illustrate how the algorithm proceeds when a link **breaks**. We return to the first diagram, with all tables completed, and then suppose the D–E link breaks. This is the "bad-news" case: a link has broken, and is no longer available; this will bring the third rule into play.



We shall assume, as above, that A reaches E via D, but we will here assume – contrary to Example 1 – that C reaches D via A (see exercise 3.5 for the original case).

Initially, upon discovering the break, D and E update their tables to ⟨E,-,∞⟩ and ⟨D,-,∞⟩ respectively (whether or not they actually enter ∞ into their tables is implementation-dependent; we may consider this as equivalent to *removing* their entries for one another; the "-" as next_hop indicates there is no next_hop).

Eventually D and E will report the break to their respective neighbors A and C. A will apply the "bad-news" rule above and update its entry for E to ⟨E,-,∞⟩. We have assumed that C, however, routes to D via A, and so it will ignore E's report.

We will suppose that the next steps are for C to report to E and to A. When C reports its route ⟨D,2⟩ to E, E will add the entry ⟨D,C,3⟩, and will again be able to reach D. When C reports to A, A will add the route ⟨E,C,2⟩. The final step will be when A next reports to D, and D will have ⟨E,A,3⟩. Connectivity is restored.

9.1.5 Example 4

The previous examples have had a "global" perspective in that we looked at the entire network. In the next example, we look at how one specific router, R, responds when it receives a distance-vector report from its neighbor S. Neither R nor S nor we have any idea of what the entire network looks like. Suppose R's table is initially as follows, and the S–R link has cost 1:

destination	next_hop	cost
A	S	3
B	T	4
C	S	5
D	U	6

S now sends R the following report, containing only destinations and its costs:

destination	cost
A	2
B	3
C	5
D	4
E	2

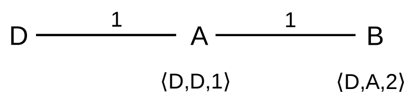
R then updates its table as follows:

destination	next_hop	cost	reason
A	S	3	No change; S probably sent this report before
B	T	4	No change; R's cost via S is tied with R's cost via T
C	S	6	Next_hop increase
D	S	5	Lower-cost route via S
E	S	3	New destination

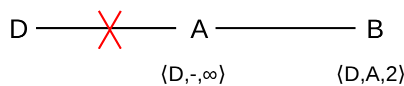
Whatever S's cost to a destination, R's cost to that destination via S is one greater.

9.2 Distance-Vector Slow-Convergence Problem

There is a significant problem with distance-vector table updates in the presence of broken links. Not only can routing loops form, but the loops can persist indefinitely! As an example, suppose we have the following arrangement, with all links having cost 1:



Now suppose the D–A link breaks:



If A immediately reports to B that D is no longer reachable (cost = ∞), then all is well. However, it is possible that B reports to A first, telling A that *it* has a route to D, with cost 2, which B still believes it has.

This means A now installs the entry ⟨D,B,3⟩. At this point we have what we called in *1.6 Routing Loops* a linear routing loop: if a packet is addressed to D, A will forward it to B and B will forward it back to A.

Worse, this loop will be with us a while. At some point A will report ⟨D,3⟩ to B, at which point B will update its entry to ⟨D,A,4⟩. Then B will report ⟨D,4⟩ to A, and A's entry will be ⟨D,B,5⟩, *etc*. This process is known as **slow convergence to infinity**. If A and B each report to the other once a minute, it will take 2,000 years for the costs to overflow an ordinary 32-bit integer.

9.2.1 Slow-Convergence Fixes

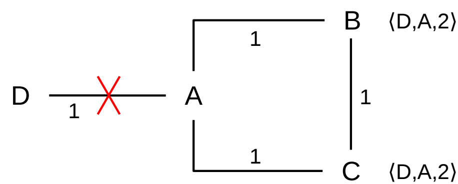
The simplest fix to this problem is to use a small value for infinity. Most flavors of the RIP protocol use infinity=16, with updates every 30 seconds. The drawback to so small an infinity is that no path through the network can be longer than this; this makes paths with weighted link costs difficult. Cisco IGRP uses a variable value for infinity up to a maximum of 256; the default infinity is 100.

There are several well-known other fixes:

9.2.1.1 Split Horizon

Under split horizon, if A uses N as its next_hop for destination D, then A simply does not report to N that it can reach D; that is, in preparing its report to N it first deletes all entries that have N as next_hop. In the example above, split horizon would mean B would never report to A about the reachability of D because A is B's next_hop to D.

Split horizon prevents all linear routing loops. However, there are other topologies where it cannot prevent loops. One is the following:



Suppose the A-D link breaks, and A updates to ⟨D,-,∞⟩. A then reports ⟨D,∞⟩ to B, which updates its table to ⟨D,-,∞⟩. But then, before A can also report ⟨D,∞⟩ to C, C reports ⟨D,2⟩ to B. B then updates to ⟨D,C,3⟩, and reports ⟨D,3⟩ back to A; neither this nor the previous report violates split-horizon. Now A's entry is ⟨D,B,4⟩. Eventually A will report to C, at which point C's entry becomes ⟨D,A,5⟩, and the numbers keep increasing as the reports circulate counterclockwise. The actual routing proceeds in the other direction, clockwise.

Split horizon often also includes **poison reverse**: if A uses N as its next_hop to D, then A in fact reports ⟨D,∞⟩ to N, which is a more definitive statement that A cannot reach D by itself. However, coming up with a scenario where poison reverse actually affects the outcome is not trivial.

9.2.1.2 Triggered Updates

In the original example, if A was first to report to B then the loop resolved immediately; the loop occurred if B was first to report to A. Nominally each outcome has probability 50%. Triggered updates means that any router should report immediately to its neighbors whenever it detects any change for the worse. If A reports first to B in the first example, the problem goes away. Similarly, in the second example, if A reports to both B and C before B or C report to one another, the problem goes away. There remains, however, a small window where B could send its report to A just as A has discovered the problem, before A can report to B.

9.2.1.3 Hold Down

Hold down is sort of a receiver-side version of triggered updates: the receiver does not use new alternative routes for a period of time (perhaps two router-update cycles) following discovery of unreachability. This gives time for bad news to arrive. In the first example, it would mean that when A received B's report $\langle$D,2$\rangle$, it would set this aside. It would then report $\langle$D,$\infty\rangle$ to B as usual, at which point B would now report $\langle$D,$\infty\rangle$ back to A, at which point B's earlier report $\langle$D,2$\rangle$ would be discarded. A significant drawback of hold down is that legitimate new routes are also delayed by the hold-down period.

These mechanisms for preventing slow convergence are, in the real world, quite effective. The Routing Information Protocol (RIP, **RFC 2453**) implements all but hold-down, and has been widely adopted at smaller installations.

However, the potential for routing loops and the limited value for infinity led to the development of alternatives. One alternative is the link-state strategy, *9.5 Link-State Routing-Update Algorithm*. Another alternative is Cisco's Enhanced Interior Gateway Routing Protocol, or EIGRP, *9.4.2 EIGRP*. While part of the distance-vector family, EIGRP is provably loop-free, though to achieve this it must sometimes suspend forwarding to some destinations while tables are in flux.

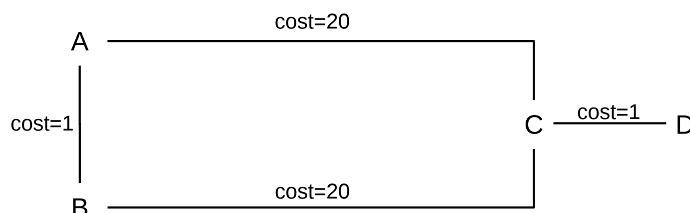
9.3 Observations on Minimizing Route Cost

Does distance-vector routing actually achieve minimum costs? For that matter, does each packet incur the cost its sender expects? Suppose node A has a forwarding entry $\langle$D,B,c$\rangle$, meaning that A forwards packets to destination D via next_hop B, and expects the total cost to be c. If A sends a packet to D, and we follow it on the actual path it takes, must the total link cost be c? If so, we will say that the network has **accurate costs**.

The answer to the accurate-costs question, as it turns out, is *yes* for the distance-vector algorithm, if we follow the rules carefully, and the network is stable (meaning that no routing reports are changing, or, more concretely, that every update report now circulating is based on the current network state); a proof is below. However, if there is a routing loop, the answer is of course no: the actual cost is now infinite. The answer would also be no if A's neighbor B has just switched to using a longer route to D than it last reported to A.

It turns out, however, that we seek the shortest route not because we are particularly trying to save money on transit costs; a route 50% longer would generally work just fine. (AT&T, back when they were the Phone Company, once ran a series of print advertisements claiming longer routes as a *feature*: if the direct path was congested, they could still complete your call by routing you the long way 'round.) However, we *are* guaranteed that if all routers seek the shortest route – and if the network is stable – then all paths are loop-free, because in this case the network will have accurate costs.

Here is a simple example illustrating the importance of global cost-minimization in preventing loops. Suppose we have a network like this one:



Now suppose that A and B use distance-vector but are allowed to choose the shortest route *to within 10%*. A would get a report from C that D could be reached with cost 1, for a total cost of 21. The forwarding entry via C would be ⟨D,C,21⟩. Similarly, A would get a report from B that D could be reached with cost 21, for a total cost of 22: ⟨D,B,22⟩. Similarly, B has choices ⟨D,C,21⟩ and ⟨D,A,22⟩.

If A and B both choose the minimal route, no loop forms. But if A and B both use the 10%-overage rule, they would be allowed to choose the other route: A could choose ⟨D,B,22⟩ and B could choose ⟨D,A,22⟩. If this happened, we would have a routing loop: A would forward packets for D to B, and B would forward them right back to A.

As we apply distance-vector routing, each router independently builds its tables. A router might have some notion of the path its packets would take to their destination; for example, in the case above A might believe that with forwarding entry ⟨D,B,22⟩ its packets would take the path A–B–C–D (though in distance-vector routing, routers do not particularly worry about the big picture). Consider again the accurate-cost question above. This *fails* in the 10%-overage example, because the actual path is now infinite.

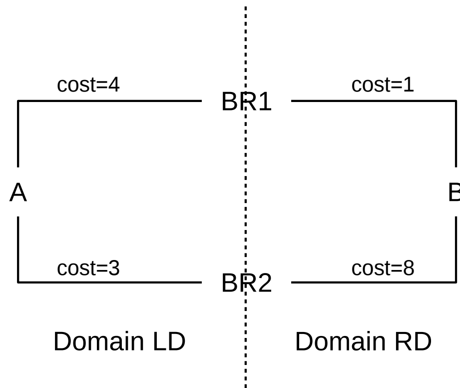
We now prove that, in distance-vector routing, the network will have accurate costs, provided

- each router selects what it believes to be the shortest path to the final destination, and
- the network is stable, meaning that further dissemination of any reports would not result in changes

To see this, suppose the actual route taken by some packet from source to destination, as determined by application of the distributed distance-vector algorithm, is longer than the cost calculated by the source. Choose an example of such a path with the **fewest number of links**, among all such paths in the network. Let S be the source, D the destination, and k the number of links in the actual path P. Let S's forwarding entry for D be ⟨D,N,c⟩, where N is S's next_hop neighbor.

To have obtained this route through the distance-vector algorithm, S must have received report ⟨D,c_D⟩ from N, where we also have the cost of the S–N link as c_N and $c = c_D + c_N$. If we follow a packet from N to D, it must take the same path P with the first link deleted; this sub-path has length k-1 and so, by our hypothesis that k was the length of the shortest path with non-accurate costs, the cost from N to D is c_D. But this means that the cost along path P, from S to D via N, must be $c_D + c_N = c$, contradicting our selection of P as a path longer than its advertised cost.

There is one final observation to make about route costs: any cost-minimization can occur only within a single routing domain, where full information about all links is available. If a path traverses multiple routing domains, each separate routing domain may calculate the optimum path traversing that domain. But these "local minimums" do not necessarily add up to a globally minimal path length, particularly when one domain calculates the minimum cost from one of its routers only to the other *domain* rather than to a router within that other domain. Here is a simple example. Routers BR1 and BR2 are the **border routers** connecting the domain LD to the left of the vertical dotted line with domain RD to the right. From A to B, LD will choose the shortest path to RD (not to B, because LD is not likely to have information about links within RD). This is the path of length 3 through BR2. But this leads to a total path length of 3+8=11 from A to B; the global minimum path length, however, is 4+1=5, through BR1.



In this example, domains LD and RD join at two points. For a route across two domains joined at only a single point, the domain-local shortest paths do add up to the globally shortest path.

9.4 Loop-Free Distance Vector Algorithms

It is possible for routing-update algorithms based on the distance-vector idea to eliminate routing loops – and thus the slow-convergence problem – entirely. We present brief descriptions of two such algorithms.

9.4.1 DSDV

DSDV, or **Destination-Sequenced Distance Vector**, was proposed in *[PB94]*. It avoids routing loops by the introduction of **sequence numbers**: each router will always prefer routes with the most recent sequence number, and bad-news information will always have a lower sequence number then the next cycle of corrected information.

DSDV was originally proposed for MANETs (*3.7.8 MANETs*) and has some additional features for traffic minimization that, for simplicity, we ignore here. It is perhaps best suited for wired networks and for small, relatively stable MANETs.

DSDV forwarding tables contain entries for every other reachable node in the system. One successor of DSDV, Ad Hoc On-Demand Distance Vector routing or AODV, allows forwarding tables to contain only those destinations in active use; a mechanism is provided for discovery of routes to newly active destinations. See *[PR99]* and **RFC 3561**.

Under DSDV, each forwarding table entry contains, in addition to the destination, cost and next_hop, the current sequence number for that destination. When neighboring nodes exchange their distance-vector reachability reports, the reports include these per-destination sequence numbers.

When a router R receives a report from neighbor N for destination D, and the report contains a sequence number larger than the sequence number for D currently in R's forwarding table, then R always updates to use the new information. The three cost-minimization rules of *9.1.1 Distance-Vector Update Rules* above are used only when the incoming and existing sequence numbers are equal.

Each time a router R sends a report to its neighbors, it includes a new value for its *own* sequence number, which it always increments by 2. This number is then entered into each neighbor's forwarding-table entry for R, and is then propagated throughout the network via continuing report exchanges. Any sequence number

originating this way will be even, and whenever another node's forwarding-table sequence number for R is even, then its cost for R will be finite.

Infinite-cost reports are generated in the usual way when former neighbors discover they can no longer reach one another; however, in this case each node increments the sequence number for its former neighbor by 1, thus generating an odd value. Any forwarding-table entry with infinite cost will thus always have an odd sequence number. If A and B are neighbors, and A's current sequence number is s, and the A–B link breaks, then B will start reporting A at cost ∞ with sequence number s+1 while A will start reporting its own new sequence number s+2. Any other node now receiving a report originating with B (with sequence number s+1) will mark A as having cost ∞, but will obtain a valid route to A upon receiving a report originating from A with new (and larger) sequence number s+2.

The triggered-update mechanism is used: if a node receives a report with some destinations newly marked with infinite cost, it will in turn forward this information immediately to its other neighbors, and so on. This is, however, not essential; "bad" and "good" reports are distinguished by sequence number, not by relative arrival time.

It is now straightforward to verify that the slow-convergence problem is solved. After a link break, if there is some alternative path from router R to destination D, then R will eventually receive D's latest even sequence number, which will be greater than any sequence number associated with any report listing D as unreachable. If, on the other hand, the break partitioned the network and there is no longer any path to D from R, then the highest sequence number circulating in R's half of the original network will be odd and the associated table entries will all list D at cost ∞. One way or another, the network will quickly settle down to a state where every destination's reachability is accurately described.

In fact, a stronger statement is true: not even transient routing loops are created. We outline a proof. First, whenever router R has next_hop N for a destination D, then N's sequence number for D must be greater than or equal to R's, as R must have obtained its current route to D from one of N's reports. A consequence is that all routers participating in a loop for destination D must have the same (even) sequence number s for D throughout. This means that the loop would have been created if only the reports with sequence number s were circulating. As we noted in *9.1.1 Distance-Vector Update Rules*, any application of the next_hop-increase rule must trace back to a broken link, and thus must involve an odd sequence number. Thus, the loop must have formed from the sequence-number-s reports by the application of the first two rules only. But this violates the claim in Exercise 10.0.

There is one drawback to DSDV: nodes may sometimes briefly switch to routes that are longer than optimum (though still correct). This is because a router is required to use the route with the newest sequence number, even if that route is longer than the existing route. If A and B are two neighbors of router R, and B is closer to destination D but slower to report, then every time D's sequence number is incremented R will receive A's longer route first, and switch to using it, and B's shorter route shortly thereafter.

DSDV implementations usually address this by having each router R keep track of the time interval between the *first* arrival at R of a new route to a destination D with a given sequence number, and the arrival of the *best* route with that sequence number. During this interval following the arrival of the first report with a new sequence number, R will use the new route, but will refrain from including the route in the reports it sends to its neighbors, anticipating that a better route will soon arrive.

This works best when the hopcount cost metric is being used, because in this case the best route is likely to arrive first (as the news had to travel the fewest hops), and at the very least will arrive soon after the first route. However, if the network's cost metric is unrelated to the hop count, then the time interval between first-route and best-route arrivals can involve multiple update cycles, and can be substantial.

9.4.2 EIGRP

EIGRP, or the **Enhanced Interior Gateway Routing Protocol**, is a once-proprietary Cisco distance-vector protocol that was released as an Internet Draft in February 2013. As with DSDV, it eliminates the risk of routing loops, even ephemeral ones. It is based on the "distributed update algorithm" (DUAL) of *[JG93]*. EIGRP is an actual protocol; we present here only the general algorithm. Our discussion follows *[CH99]*.

Each router R keeps a list of neighbor routers N_R, as with any distance-vector algorithm. Each R also maintains a data structure known (somewhat misleadingly) as its **topology table**. It contains, for each destination D and each N in N_R, an indication of whether N has reported the ability to reach D and, if so, the reported cost c(D,N). The router also keeps, for each N in N_R, the cost c_N of the link from R to N. Finally, the forwarding-table entry for any destination can be marked "passive", meaning safe to use, or "active", meaning updates are in process and the route is temporarily unavailable.

Initially, we expect that for each router R and each destination D, R's next_hop to D in its forwarding table is the neighbor N for which the following total cost is a minimum:

$$c(D,N) + c_N$$

Now suppose R receives a distance-vector report from neighbor N_1 that it can reach D with cost $c(D,N_1)$. This is processed in the usual distance-vector way, unless it represents an increased cost and N_1 is R's next_hop to D; this is the third case in *9.1.1 Distance-Vector Update Rules*. In this case, let C be R's current cost to D, and let us say that neighbor N of R is a **feasible** next_hop (feasible successor in Cisco's terminology) if N's cost to D (that is, c(D,N)) is strictly less than C. R then updates its route to D to use the feasible neighbor N for which $c(D,N) + c_N$ is a minimum. Note that this may not in fact be the shortest path; it is possible that there is another neighbor M for which $c(D,M)+c_M$ is smaller, but $c(D,M) \geqslant C$. However, because N's path to D is loop-free, and because $c(D,N) < C$, this new path through N must also be loop-free; this is sometimes summarized by the statement "one cannot create a loop by adopting a shorter route".

If no neighbor N of R is feasible – which would be the case in the D—A—B example of *9.2 Distance-Vector Slow-Convergence Problem*, then R invokes the "DUAL" algorithm. This is sometimes called a "diffusion" algorithm as it invokes a diffusion-like spread of table changes proceeding away from R.

Let C in this case denote the new cost from R to D as based on N_1's report. R marks destination D as "active" (which suppresses forwarding to D) and sends a special query to each of its neighbors, in the form of a distance-vector report indicating that its cost to D has now increased to C. The algorithm terminates when all R's neighbors reply back with their own distance-vector reports; at that point R marks its entry for D as "passive" again.

Some neighbors may be able to process R's report without further diffusion to other nodes, remain "passive", and reply back to R immediately. However, other neighbors may, like R, now become "active" and continue the DUAL algorithm. In the process, R may receive other queries that elicit its distance-vector report; as long as R is "active" it will report its cost to D as C. We omit the argument that this process – and thus the network – must eventually converge.

9.5 Link-State Routing-Update Algorithm

Link-state routing is an alternative to distance-vector. It is often – though certainly not always – considered to be the routing-update algorithm class of choice for networks that are "sufficiently large", such as those of ISPs.

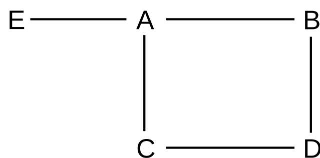
In distance-vector routing, each node knows a bare minimum of network topology: it knows nothing about links beyond those to its immediate neighbors. In the link-state approach, each node keeps a *maximum* amount of network information: a full map of all nodes and all links. Routes are then computed locally from this map, using the shortest-path-first algorithm. The existence of this map allows, in theory, the calculation of different routes for different quality-of-service requirements. The map also allows calculation of a new route as soon as news of the failure of the existing route arrives; distance-vector protocols on the other hand must wait for news of a new route after an existing route fails.

Link-state protocols distribute network map information through a modified form of broadcast of the status of each individual link. Whenever either side of a link notices the link has died (or if a node notices that a new link has become available), it sends out **link-state packets** (LSPs) that "flood" the network. This broadcast process is called **reliable flooding**. In general, broadcast mechanisms are not compatible with networks that have topological looping (that is, redundant paths); broadcast packets may circulate around the loop endlessly. Link-state protocols must be carefully designed to ensure that both every router sees every LSP, and also that no LSPs circulate repeatedly. (The acronym LSP is used by a link-state implementation known as IS-IS; the preferred acronym used by the Open Shortest Path First (OSPF) implementation is LSA, where A is for advertisement.) LSPs are sent immediately upon link-state changes, like triggered updates in distance-vector protocols except there is no "race" between "bad news" and "good news".

It is possible for ephemeral routing loops to exist; for example, if one router has received a LSP but another has not, they may have an inconsistent view of the network and thus route to one another. However, as soon as the LSP has reached all routers involved, the loop should vanish. There are no "race conditions", as with distance-vector routing, that can lead to persistent routing loops.

The link-state flooding algorithm avoids the usual problems of broadcast in the presence of loops by having each node keep a database of all LSP messages. The originator of each LSP includes its identity, information about the link that has changed status, and also a **sequence number**. Other routers need only keep in their databases the LSP packet with the largest sequence number; older LSPs can be discarded. When a router receives a LSP, it first checks its database to see if that LSP is old, or is current but has been received before; in these cases, no further action is taken. If, however, an LSP arrives with a sequence number not seen before, then in typical broadcast fashion the LSP is retransmitted over all links except the arrival interface.

As an example, consider the following arrangement of routers:



Suppose the A–E link status changes. A sends LSPs to C and B. Both these will forward the LSPs to D; suppose B's arrives first. Then D will forward the LSP to C; the LSP traveling C→D and the LSP traveling D→C might even cross on the wire. D will ignore the second LSP copy that it receives from C and C will ignore the second copy it receives from D.

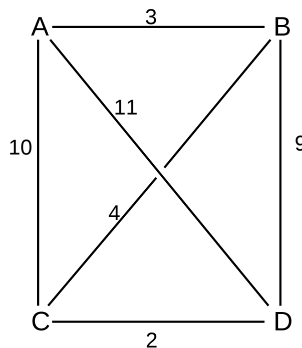
It is important that LSP sequence numbers not wrap around. (Protocols that *do* allow a numeric field to wrap around usually have a clear-cut idea of the "active range" that can be used to conclude that the numbering has wrapped rather than restarted; this is harder to do in the link-state context.) OSPF uses **lollipop sequence-numbering** here: sequence numbers begin at -2^{31} and increment to $2^{31}-1$. At this point they wrap around back to 0. Thus, as long as a sequence number is less than zero, it is guaranteed unique; at the same time,

routing will not cease if more than 2^{31} updates are needed. Other link-state implementations use 64-bit sequence numbers.

Actual link-state implementations often give link-state records a maximum lifetime; entries *must* be periodically renewed.

9.5.1 Shortest-Path-First Algorithm

The next step is to compute routes from the network map, using the shortest-path-first algorithm. Below is our example network; we are interested in the shortest paths from A to B, C and D.



The shortest path from A to D is A-B-C-D, which has cost 3+4+2=9.

We build the set **R** of all shortest-path routes iteratively. Initially, **R** contains only the 0-length route to the start node; one new destination and route is added to **R** at each stage of the iteration. At each stage we have a **current** node, representing the node most recently added to **R**. The initial **current** node is our starting node, in this case, A.

We will also maintain a set **T**, for tentative, of routes to other destinations. This is also initialized to empty.

At each stage, we find all nodes which are immediate neighbors of the **current** node and which do not already have routes in the set **R**. For each such node N, we calculate the length of the route from the start node to N that goes through the **current** node. We see if this is our first route to N, or if the route improves on any route to N already in **T**; if so, we add or update the route in **T** accordingly.

At the end of this process, we choose the shortest path in **T**, and move the route and destination node to **R**. The destination node of this shortest path becomes the next **current** node. Ties can be resolved arbitrarily, but note that, as with distance-vector routing, we must choose the minimum or else the accurate-costs property will fail.

We repeat this process until all nodes have routes in the set **R**.

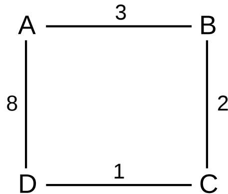
For the example above, we start with **current** = A and **R** = {⟨A,A,0⟩}. The set **T** will be {⟨B,B,3⟩, ⟨C,C,10⟩, ⟨D,D,11⟩}. The shortest entry is ⟨B,B,3⟩, so we move that to **R** and continue with **current** = B.

For the next stage, the neighbors of B without routes in **R** are C and D; the routes from A to these through B are ⟨C,B,7⟩ and ⟨D,B,12⟩. The former is an improvement on the existing **T** entry ⟨C,C,10⟩ and so replaces it; the latter is not an improvement over ⟨D,D,11⟩. **T** is now {⟨C,B,7⟩, ⟨D,D,11⟩}. The shortest route in **T** is that to C, so we move this node and route to **R** and set C to be **current**.

For the next stage, D is the only non-**R** neighbor; the path from A to D via C has entry ⟨D,B,9⟩, an improvement over the existing ⟨D,D,11⟩ in **T**. The only entry in **T** is now ⟨D,B,9⟩; this is the shortest and thus we move it to **R**.

We now have routes in **R** to all nodes, and are done.

Here is another example, again with links labeled with costs:



We start with **current** = A. At the end of the first stage, ⟨B,B,3⟩ is moved into **R**, **T** is {⟨D,D,12⟩}, and **current** is B. The second stage adds ⟨C,B,5⟩ to **T**, and then moves this to **R**; **current** then becomes C. The third stage introduces the route (from A) ⟨D,B,10⟩; this is an improvement over ⟨D,D,12⟩ and so replaces it in **T**; at the end of the stage this route to D is moved to **R**.

A link-state source node S computes the entire path to a destination D. But as far as the actual path that a packet sent by S will take to D, S has direct control only as far as the first hop N. While the accurate-cost rule we considered in distance-vector routing will still hold, the actual path taken by the packet may differ from the path computed at the source, in the presence of alternative paths of the same length. For example, S may calculate a path S–N–A–D, and yet a packet may take path S–N–B–D, so long as the N–A–D and N–B–D paths have the same length.

Link-state routing allows calculation of routes on demand (results are then cached), or larger-scale calculation. Link-state also allows routes calculated with quality-of-service taken into account, via straightforward extension of the algorithm above.

9.6 Routing on Other Attributes

There is sometimes a desire to route on packet attributes other than the destination, or the destination and QoS bits. For example, we might want to route packets based in part on the packet source, or on the TCP port number. This kind of routing is decidedly nonstandard, though it is often available, and often an important component of traffic engineering.

This option is often known as **policy-based routing**, because packets are routed according to attributes specified by local administrative policy. (This term should not be confused with BGP *routing policy* (*10.6 Border Gateway Protocol, BGP*), which means something quite different.)

Policy-based routing is not used frequently, but one routing decision of this type can have far-reaching effects. If an ISP wishes to route customer voice traffic differently from customer data traffic, for example, it need only apply policy-based routing to classify traffic at the point of entry, and send the voice traffic to its own router. After that, ordinary routers on the voice path and on the separate data path can continue the forwarding without using policy-based methods.

Sometimes policy-based routing is used to *mark* packets for special processing; this might mean different

routing further downstream or it might mean being sent along the same path as the other traffic but with preferential treatment. For two packet-marking strategies, see *20.7 Differentiated Services* and *20.12 Multi-Protocol Label Switching (MPLS)*.

On linux systems policy-based routing is part of the Linux Advanced Routing facility, often grouped with some advanced queuing features known as Traffic Control; the combination is referred to as LARTC.

As a simple example of what can be done, suppose a site has two links L1 and L2 to the Internet, with L1 the default route to the Internet. Perhaps L1 is faster and L2 serves more as a backup; perhaps L2 has been added to increase outbound capacity. A site may wish to route some outbound traffic via L2 for any of the following reasons:

- the traffic may involve protocols deemed lower in priority (*eg* email)
- the traffic may be real-time traffic that can benefit from reduced competition on L2
- the traffic may come from lower-priority senders; *eg* some customers within the site may be relegated to using L2 because they are paying less
- a few large-volume elephant flows may be offloaded from L1 to L2

In the first two cases, routing might be based on the destination port numbers; in the third, it might be based on the source IP address. In the fourth case, a site's classification of its elephant flows may have accumulated over time.

Note that nothing can be done in the *inbound* direction unless L1 and L2 lead to the same ISP, and even there any special routing would be at the discretion of that ISP.

The trick with LARTC is to be compatible with existing routing-update protocols; this would be a problem if the kernel forwarding table simply added columns for other packet attributes that neighboring non-LARTC routers knew nothing about. Instead, the forwarding table is split up into multiple ⟨dest, next_hop⟩ (or ⟨dest, QoS, next_hop⟩) tables. One of these tables is the **main** table, and is the table that is updated by routing-update protocols interacting with neighbors. Before a packet is forwarded, administratively supplied rules are consulted to determine which table to apply; these rules *are* allowed to consult other packet attributes. The collection of tables and rules is known as the **routing policy database**.

As a simple example, in the situation above the main table would have an entry ⟨default, L1⟩ (more precisely, it would have the IP address of the far end of the L1 link instead of L1 itself). There would also be another table, perhaps named **slow**, with a single entry ⟨default, L2⟩. If a rule is created to have a packet routed using the "slow" table, then that packet will be forwarded via L2. Here is one such linux rule, applying to traffic from host 10.0.0.17:

```
ip rule add from 10.0.0.17 table slow
```

Now suppose we want to route traffic to port 25 (the SMTP port) via L2. This is harder; linux provides no support here for routing based on port numbers. However, we can instead use the iptables mechanism to "mark" all packets destined for port 25, and then create a routing-policy rule to have such marked traffic use the slow table. The mark is known as the forwarding mark, or `fwmark`; its value is 0 by default. The `fwmark` is not actually part of the packet; it is associated with the packet only while the latter remains within the kernel.

```
iptables --table mangle --append PREROUTING \\
         --protocol tcp --dest-port 25 --jump MARK --set-mark 1
```

```
ip rule add fwmark 1 table slow
```

Consult the applicable man pages for further details.

The iptables mechanism can also be used to set the appropriate QoS bits – the IPv4 DS bits (*7.1 The IPv4 Header*) or the IPv6 Traffic Class bits (*8.1 The IPv6 Header*) – so that a single standard IP forwarding table can be used, though support for the IPv4 QoS bits is limited.

9.7 Epilog

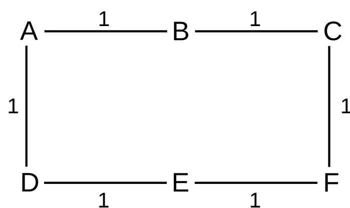
At this point we have concluded the basics of IP routing, involving routing within large (relatively) *homogeneous* organizations such as multi-site corporations or Internet Service Providers. Every router involved must agree to run the same protocol, and must agree to a uniform assignment of link costs.

At the very largest scales, these requirements are impractical. The next chapter is devoted to this issue of very-large-scale IP routing, *eg* on the global Internet.

9.8 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercise 2.5 is distinct, for example, from exercises 2.0 and 3.0. Exercises marked with a ♢ have solutions or hints at 24.8 Solutions for Routing-Update Algorithms.

1.0. Suppose the network is as follows, where distance-vector routing update is used. Each link has cost 1, and each router has entries in its forwarding table only for its immediate neighbors (so A's table contains ⟨B,B,1⟩, ⟨D,D,1⟩ and B's table contains ⟨A,A,1⟩, ⟨C,C,1⟩).

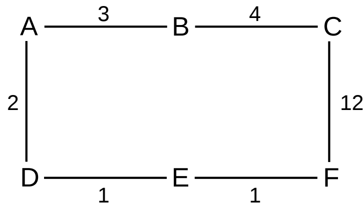


(a). Suppose each node creates a report from its initial configuration and sends that to each of its neighbors. What will each node's forwarding table be after this set of exchanges? The exchanges, in other words, are all conducted simultaneously; each node first sends out its own report and then processes the reports arriving from its two neighbors.

(b). What will each node's table be after the simultaneous-and-parallel exchange process of part (a) is repeated a second time?

Hint: you do not have to go through each exchange in detail; the only information added by an exchange is additional *reachability* information.

2.0. Now suppose the configuration of routers has the link weights shown below.



(a). As in the previous exercise, give each node's forwarding table after each node exchanges with its immediate neighbors simultaneously and in parallel.

(b). How many iterations of such parallel exchanges will it take before C learns to reach F via B; that is, before it creates the entry ⟨F,B,11⟩? Count the answer to part (a) as the first iteration.

2.5.◇ A router R has the following distance-vector table:

destination	cost	next hop
A	2	R1
B	3	R2
C	4	R1
D	5	R3

R now receives the following report from R1; the cost of the R–R1 link is 1.

destination	cost
A	1
B	2
C	4
D	3

Give R's updated table after it processes R1's report. For each entry that changes, give a brief explanation

3.0. A router R has the following distance-vector table:

destination	cost	next hop
A	5	R1
B	6	R1
C	7	R2
D	8	R2
E	9	R3

R now receives the following report from R1; the cost of the R–R1 link is 1.

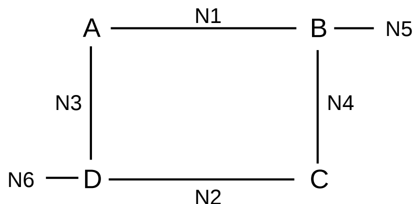
destination	cost
A	4
B	7
C	7
D	6
E	8
F	8

Give R's updated table after it processes R1's report. For each entry that changes, give a brief explanation, in the style of *9.1.5 Example 4*.

3.5. At the start of Example 3 (*9.1.4 Example 3*), we changed C's routing table so that it reached D via A instead of via E: C's entry ⟨D,E,2⟩ was changed to ⟨D,A,2⟩. This meant that C had a valid route to D at the start.

How might the scenario of Example 3 play out if C's table had not been altered? Give a sequence of reports that leads to correct routing between D and E.

4.0. In the following exercise, A-D are routers and the attached subnets N1-N6, which are the ultimate destinations, are shown explicitly. In the case of N1 through N4, the links *are* the subnets. Routers still exchange distance-vector reports with neighboring routers, as usual. In the tables requested below, if a router has a direct connection to a subnet, you may report the next_hop as "direct", *eg*, from A's table, ⟨N1,direct,0⟩

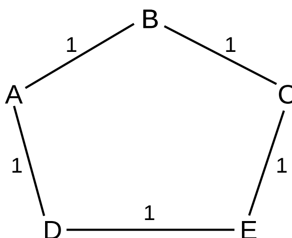


(a). Give the initial tables for A through D, before any distance-vector exchanges.

(b). Give the tables after each router A-D exchanges with its immediate neighbors simultaneously and in parallel.

(c). At the end of (b), what subnets are *not* known by what routers?

5.0. Suppose A, B, C, D and E are connected as follows. Each link has cost 1, and so each forwarding table is uniquely determined; B's table is ⟨A,A,1⟩, ⟨C,C,1⟩, ⟨D,A,2⟩, ⟨E,C,2⟩. Distance-vector routing update is used.



Now suppose the D–E link fails, and so D updates its entry for E to ⟨E,-,∞⟩.

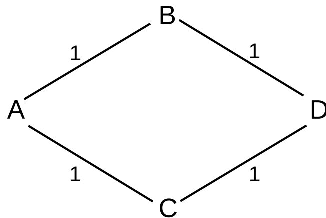
(a). Give A's table after D reports ⟨E,∞⟩ to A

(b). Give B's table after A reports to B

(c). Give A's table after B reports to A; note that B has an entry ⟨E,C,2⟩

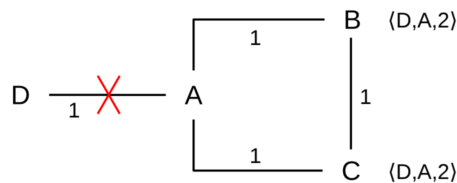
(d). Give D's table after A reports to D.

5.5. In the network below, A receives alternating reports about destination D from neighbors B and C. Suppose A uses a modified form of Rule 2 of *9.1.1 Distance-Vector Update Rules*, in which it updates its forwarding table whenever new cost c is less than *or equal to* c_{old}.



Explain why A's forwarding entry for destination D never stabilizes.

6.0. Consider the network in *9.2.1.1 Split Horizon*:, using distance-vector routing updates. B and C's table entries for destination D are shown. All link costs are 1.

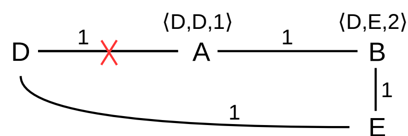


Suppose the D–A link breaks and then these update reports occur:

- A reports ⟨D,∞⟩ to B (as before)
- C reports ⟨D,2⟩ to B (as before)
- A now reports ⟨D,∞⟩ to C (instead of B reporting ⟨D,3⟩ to A)

(a). Give A, B and C's forwarding-table records for destination D, including the cost, after these three reports.
(b). What additional reports (a pair should suffice) will lead to the formation of the routing loop?
(c). What (single) additional report will eliminate the possibility of the routing loop?

7.0. Suppose the network of *9.2 Distance-Vector Slow-Convergence Problem* is changed to the following. Distance-vector update is used; again, the D–A link breaks.



(a). Explain why B's report back to A, after A reports ⟨D,-,∞⟩, is now valid.

(b). Explain why hold down (*9.2.1.3 Hold Down*) will delay the use of the new route A–B–E–D.

8.0. Suppose the routers are A, B, C, D, E and F, and all link costs are 1. The distance-vector forwarding tables for A and F are below. Give the network with the fewest links that is consistent with these tables. Hint: any destination reached at cost 1 is directly connected; if X reaches Y via Z at cost 2, then Z and Y must be directly connected.

A's table

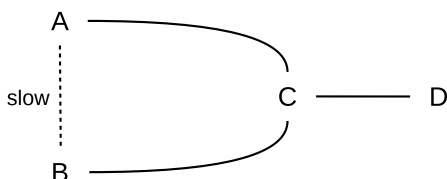
dest	cost	next_hop
B	1	B
C	1	C
D	2	C
E	2	C
F	3	B

F's table

dest	cost	next_hop
A	3	E
B	2	D
C	2	D
D	1	D
E	1	E

9.0. (a) Suppose routers A and B somehow end up with respective forwarding-table entries ⟨D,B,n⟩ and ⟨D,A,m⟩, thus creating a routing loop. Explain why the loop may be removed more quickly if A and B both use **poison reverse** with split horizon (*9.2.1.1 Split Horizon*), versus if A and B use split horizon only.

(b). Suppose the network looks like the following. The A–B link is extremely slow.



Suppose A and B send reports to each other advertising their routes to D, and immediately afterwards the C–D link breaks and C reports to A and B that D is unreachable. *After* those unreachability reports from C are processed, A and B's reports sent to each other before the break finally arrive. Explain why the network is now in the state described in part (a).

10.0. Suppose the distance-vector algorithm is run on a network and no links break (so by the last paragraph of *9.1.1 Distance-Vector Update Rules* the next_hop-increase rule is never applied).

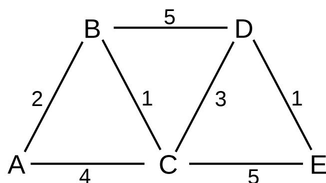
(a). Prove that whenever A is B's next_hop to destination D, then A's cost to D is strictly less than B's. Hint: assume that if this claim is true, then it remains true after any application of the rules in

9.1.1 Distance-Vector Update Rules. If the lower-cost rule is applied to B after receiving a report from A, resulting in a change to B's cost to D, then one needs to show A's cost is less than B's, and also B's new cost is less than that of any neighbor C that uses B as its next_hop to D.

(b). Use (a) to prove that no routing loops ever form.

11.0. It was mentioned in *9.5 Link-State Routing-Update Algorithm* that link-state routing might give rise to an ephemeral routing loop. Give a concrete scenario illustrating creation (and then dissolution) of such a loop.

12.0. Use the Shortest-Path-First algorithm to find the shortest path from A to E in the network below. Show the sets **R** and **T**, and the node **current**, after each step.



13.0. Suppose you take a laptop, plug it into an Ethernet LAN, *and* connect to the same LAN via Wi-Fi. From laptop to LAN there are now two routes. Which route will be preferred? How can you tell which way traffic is flowing? How can you configure your OS to prefer one path or another? (See also *7.9.5 ARP and multihomed hosts*, *7 IP version 4* exercise 11.0, and *12 TCP Transport* exercise 13.0.)

In the previous chapter we considered two classes of routing-update algorithms: distance-vector and link-state. Each of these approaches requires that participating routers have agreed first to a common protocol, and then to a common understanding of how link costs are to be assigned. We will address this further below in *10.6 Border Gateway Protocol, BGP*, but a basic problem is that if one site prefers the hop-count approach, assigning every link a cost of 1, while another site prefers to assign link costs in proportion to their bandwidth, then *meaningful* path cost comparisons between the two sites simply cannot be done.

The term **routing domain** is used to refer to a set of routers under common administration, using a common link-cost assignment. Another term for this is **autonomous system**. While use of a common routing-update protocol within the routing domain is not an absolute requirement – for example, some subnets may internally use distance-vector while the site's "backbone" routers use link-state – we can assume that all routers have a uniform view of the site's topology and cost metrics.

One of the things included in the term "large-scale" IP routing is the coordination of routing between multiple routing domains. Even in the earliest Internet there were multiple routing domains, if for no other reason than that how to measure link costs was (and still is) too unsettled to set in stone. However, another component of large-scale routing is support for **hierarchical** routing, above the level of subnets; we turn to this next.

10.1 Classless Internet Domain Routing: CIDR

CIDR is the mechanism for supporting hierarchical routing in the Internet backbone. Subnetting moves the network/host division line further rightwards; CIDR allows moving it to the left as well. With subnetting, the revised division line is visible only within the organization that owns the IP network address; subnetting is not visible outside. CIDR allows aggregation of IP address blocks in a way that *is* visible to the Internet backbone.

When CIDR was introduced in 1993, the following were some of the justifications for it, all relating to the increasing size of the backbone IP forwarding tables, and expressed in terms of the then-current Class A/B/C mechanism:

- The Internet is running out of Class B addresses (this happened in the mid-1990's)
- There are too many Class C's (the most numerous) for backbone forwarding tables to be efficient
- Eventually IANA (the Internet Assigned Numbers Authority) will run out of IP addresses (this happened in 2011)

Assigning non-CIDRed multiple Class C's in lieu of a single Class B would have helped with the first point in the list above, but made the second point worse.

Ironically, the current (2013) very tight market for IP address blocks is likely to lead to larger and larger backbone IP forwarding tables, as sites are forced to use multiple small address blocks instead of one large block.

By the year 2000, CIDR had essentially eliminated the Class A/B/C mechanism from the backbone Internet, and had more-or-less completely changed how backbone routing worked. You purchased an address block from a provider or some other IP address allocator, and it could be whatever size you needed, from /32 to /15.

What CIDR enabled is IP routing based on an address prefix of any length; the Class A/B/C mechanism of course used fixed prefix lengths of 8, 16 and 24 bits. Furthermore, CIDR allows different routers, at different levels of the backbone, to route on prefixes of different lengths. If organization P were allocated a /10 block, for example, then P could *suballocate* into /20 blocks. At the top level, routing to P would likely be based on the first 10 bits, while routing within P would be based on the first 20 bits.

CIDR was formally introduced by **RFC 1518** and **RFC 1519**. For a while there were strategies in place to support compatibility with non-CIDR-aware routers; these are now obsolete. In particular, it is no longer appropriate for large-scale routers to fall back on the Class A/B/C mechanism in the absence of CIDR information; if the latter is missing, the routing should fail.

One way to look at the basic strategy of CIDR is as a mechanism to consolidate multiple network blocks going to the same destination into a single entry. Suppose a router has four class C's all to the same destination:

200.7.0.0/24 ⟶ foo
200.7.1.0/24 ⟶ foo
200.7.2.0/24 ⟶ foo
200.7.3.0/24 ⟶ foo

The router can replace all these with the single entry

200.7.0.0/**22** ⟶ foo

It does not matter here if foo represents a single ultimate destination or if it represents four sites that just happen to be routed to the same next_hop.

It is worth looking closely at the arithmetic to see why the single entry uses /22. This means that the first 22 bits must match 200.7.0.0; this is all of the first and second bytes and the first six bits of the third byte. Let us look at the third byte of the network addresses above in binary:

200.7.000000 00.0/24 ⟶ foo
200.7.000000 01.0/24 ⟶ foo
200.7.000000 10.0/24 ⟶ foo
200.7.000000 11.0/24 ⟶ foo

The /24 means that the network addresses stop at the end of the third byte. The four entries above cover every possible combination of the last two bits of the third byte; for an address to match one of the entries above it suffices to begin 200.7 and then to have 0-bits as the first *six bits* of the third byte. This is another way of saying the address must match 200.7.0.0/22.

Most implementations actually use a bitmask, *eg* 255.255.252.0, rather than the number 22. Note 252 is, in binary, 1111 1100, with 6 leading 1-bits, so 255.255.252.0 has 8+8+6=22 1-bits followed by 10 0-bits.

The IP delivery algorithm of *7.5 The Classless IP Delivery Algorithm* still works with CIDR, with the understanding that the router's forwarding table can now have a network-prefix length associated with *any* entry. Given a destination D, we search the forwarding table for network-prefix destinations B/k until we

find a match; that is, equality of the first k bits. In terms of masks, given a destination D and a list of table entries ⟨prefix,mask⟩ = ⟨B[i],M[i]⟩, we search for i such that (D & M[i]) = B[i].

But what about the possibility of multiple matches? For subnets, avoiding this was the responsibility of the subnetting site, but responsibility for avoiding this with CIDR is much too distributed to be declared illegal by IETF mandate. Instead, CIDR introduced the **longest-match rule**: if destination D matches both B_1/k_1 and B_2/k_2, with $k_1 < k_2$, then the longer match B_2/k_2 match is to be used. (Note that if D matches two distinct entries B_1/k_1 and B_2/k_2 then either $k_1 < k_2$ or $k_2 < k_1$).

10.2 Hierarchical Routing

Strictly speaking, CIDR is simply a mechanism for routing to IP address blocks of any prefix length; that is, for setting the network/host division point to an arbitrary place within the 32-bit IP address.

However, by making this network/host division point *variable*, CIDR introduced support for routing on *different* prefix lengths at different places in the backbone routing infrastructure. For example, top-level routers might route on /8 or /9 prefixes, while intermediate routers might route based on prefixes of length 14. This feature of routing on fewer bits at one point in the Internet and more bits at another point is exactly what is meant by **hierarchical routing**.

We earlier saw hierarchical routing in the context of subnets: traffic might first be routed to a class-B site 147.126.0.0/16, and then, within that site, to subnets such as 147.126.1.0/24, 147.126.2.0/24, *etc*. But with CIDR the hierarchy can be much more flexible: the top level of the hierarchy can be much larger than the "customer" level, lower levels need not be administratively controlled by the higher levels (as is the case with subnets), and more than two levels can be used.

CIDR is an address-block-allocation *mechanism*; it does not directly speak to the kinds of *policy* we might wish to implement with it. Here are four possible applications; the latter two involve hierarchical routing:

- Application 1 (legacy): CIDR allows the allocation of multiple blocks of Class C, or fragments of a Class A, to a single customer, so as to require only a single forwarding-table entry for that customer
- Application 2 (legacy): CIDR allows opportunistic **aggregation** of routes: a router that sees the four 200.7.x.0/24 routes above in its table may consolidate them into a single entry.
- Application 3 (current): CIDR allows huge provider blocks, with suballocation by the provider. This is known as **provider-based** routing.
- Application 4 (hypothetical): CIDR allows huge regional blocks, with suballocation within the region, somewhat like the original scheme for US phone numbers with area codes. This is known as **geographical** routing.

Each of these has the potential to achieve a considerable reduction in the size of the backbone forwarding tables, which is arguably the most important goal here. Each involves using CIDR to support the creation of arbitrary-sized address blocks and then *routing to them as a single unit*. For example, the Internet backbone might be much happier if all its routers simply had to maintain a single entry ⟨200.0.0.0/8, R1⟩, versus 256 entries ⟨200.x.0.0/16, R1⟩ for every value of x. (As we will see below, this is still useful even if a few of the x's have a different next_hop.) Secondary CIDR goals include bringing some order to IP address allocation and (for the last two items in the list above) enabling a routing hierarchy that mirrors the actual flow of most traffic.

Hierarchical routing does introduce one new wrinkle: the routes chosen may no longer be globally optimal, at least if we also apply the routing-update algorithms hierarchically. Suppose, for example, at the top level forwarding is based on the first eight bits of the address, and all traffic to 200.0.0.0/8 is routed to router R1. At the second level, R1 then routes traffic (hierarchically) to 200.20.0.0/16 via R2. A packet sent to 200.20.1.2 by an independent router R3 might therefore pass through R1, *even if there were a lower-cost path R3→R4→R2 that bypassed R1*. The top-level forwarding entry ⟨200.0.0.0/8,R1⟩, in other words, may represent a simplification of the real situation. Prohibiting "back-door" routes like R3→R4→R2 is impractical (and would not be helpful either); customers are independent entities.

This non-optimal routing issue cannot happen if all routers agree upon one of the shortest-path mechanisms of *9 Routing-Update Algorithms*; in that case R3 would learn of the lower-cost R3→R4→R2 path. But then the potential hierarchical benefits of decreasing the size of forwarding tables would be lost. More seriously, complete global agreement of all routers on one common update protocol is simply not practical; in fact, one of the goals of hierarchical routing is to provide a workable alternative. We will return to this below in *10.4.3 Hierarchical Routing via Providers*.

10.3 Legacy Routing

Back in the days of NSFNet, the Internet backbone was a single routing domain. While most customers did not connect directly to the backbone, the intervening providers tended to be relatively compact, geographically – that is, *regional* – and often had a single primary routing-exchange point with the backbone. IP addresses were allocated to subscribers directly by the IANA, and the backbone forwarding tables contained entries for every site, even the Class C's.

Because the NSFNet backbone and the regional providers did not necessarily share link-cost information, routes were even at this early point not necessarily globally optimal; compromises and approximations were made. However, in the NSFNet model routers generally did find a reasonable approximation to the shortest path to each site referenced by the backbone tables. While the legacy backbone routing domain was not all-encompassing, if there *were* differences between two routes, at least the backbone portions – the longest components – would be identical.

10.4 Provider-Based Routing

In provider-based routing, large CIDR blocks are allocated to large-scale providers. The different providers each know how to route to one another. Subscribers (usually) obtain their IP addresses from within their providers' blocks; thus, traffic from the outside is routed first to the provider, and then, *within* the provider's routing domain, to the subscriber. We may even have a hierarchy of providers, so packets would be routed first to the large-scale provider, and eventually to the local provider. There may no longer be a central backbone; instead, multiple providers may each build parallel transcontinental networks.

Here is a simpler example, in which providers have *unique* paths to one another. Suppose we have providers P0, P1 and P2, with customers as follows:

- P0: customers A,B,C
- P1: customers D,E
- P2: customers F,G

We will also assume that each provider has an IP address block as follows:

- P0: 200.0.0.0/8
- P1: 201.0.0.0/8
- P2: 202.0.0.0/8

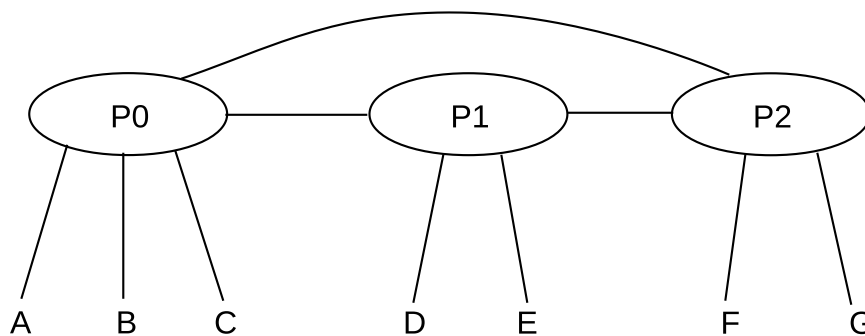
Let us now allocate addresses to the customers:

A: 200.0.0.0/16
B: 200.1.0.0/16
C: 200.2.16.0/20 (16 = 0001 0000)

D: 201.0.0.0/16
E: 201.1.0.0/16

F: 202.0.0.0/16
G: 202.1.0.0/16

The routing model is that packets are first routed to the appropriate provider, and then to the customer. While this model may not in general guarantee the shortest end-to-end path, it does in this case because each provider has a single point of interconnection to the others. Here is the network diagram:



With this diagram, P0's forwarding table looks something like this:

P0	
destination	next_hop
200.0.0.0/16	A
200.1.0.0/16	B
200.2.16.0/20	C
201.0.0.0/8	P1
202.0.0.0/8	P2

That is, P0's table consists of

- one entry for each of P0's own customers

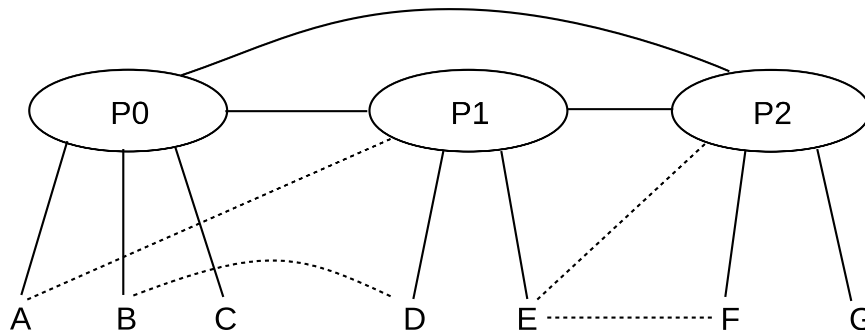
- one entry for each other provider

If we had 1,000,000 customers divided equally among 100 providers, then each provider's table would have only 10,099 entries: 10,000 for its own customers and 99 for the other providers. Without CIDR, each provider's forwarding table would have 1,000,000 entries.

CIDR enables hierarchical routing by allowing the routing decision to be made on different prefix lengths in different contexts. For example, when a packet is sent from D to A, P1 looks at the first 8 bits while P0 looks at the first 16 bits. Within customer A, routing might be made based on the first 24 bits.

Even if we have some additional "secondary" links, that is, additional links that do not create alternative paths between providers, the routing remains *relatively* straightforward. Shown here are the private customer-to-customer links C–D and E–F; these are likely used only by the customers they connect. Two customers, A and E, are **multihomed**; that is, they have connections to alternative providers: A–P1 and E–P2. (The term "multihomed" is often applied to any host with multiple network interfaces on different LANs, which includes any router; here we mean more specifically that there are multiple network interfaces connecting to different providers.)

Typically, though, while A and E may use their alternative-provider links all they want for *outbound* traffic, their respective *inbound* traffic would still go through their primary providers P0 and P1 respectively.



10.4.1 Internet Exchange Points

The long links joining providers in these diagrams are somewhat misleading; providers do not always like sharing long links and the attendant problems of sharing responsibility for failures. Instead, providers often connect to one another at **Internet eXchange Points** or IXPs; the link P0———P1 might actually be P0——IXP——P1, where P0 owns the left-hand link and P1 the right-hand. IXPs can either be third-party sites open to all providers, or private exchange points. The term "Metropolitan Area Exchange", or MAE, appears in the names of the IXPs MAE-East, originally near Washington DC, and MAE-West, originally in San Jose, California; each of these is now actually a *set* of IXPs. MAE in this context is now a trademark.

10.4.2 CIDR and Staying Out of Jail

Suppose we want to change providers. One way we can do this is to accept a new IP-address block from the new provider, and change all our IP addresses. The paper *Renumbering: Threat or Menace* [LKCT96] was frequently cited – at least in the early days of CIDR – as an intimation that such renumbering was inevitably

a Bad Thing. In principle, therefore, we would like to allow at least the option of *keeping* our IP address allocation while changing providers.

An address-allocation standard that did not allow changing of providers might even be a violation of the US Sherman Antitrust Act; see *American Society of Mechanical Engineers v Hydrolevel Corporation*, 456 US 556 (1982). The IETF thus had the added incentive of wanting to stay out of jail, when writing the CIDR standard so as to allow portability between providers (actually, antitrust violations usually involve civil penalties).

The CIDR **longest-match** rule turns out to be exactly what we (and the IETF) need. Suppose, in the diagrams above, that customer C wants to move from P0 to P1, and does not want to renumber. What routing changes need to be made? One solution is for P0 to add a route ⟨200.2.16.0/20, P1⟩ that routes all of C's traffic to P1; P1 will then forward that traffic on to C. P1's table will be as follows, and P1 will use the longest-match rule to distinguish traffic for its new customer C from traffic bound for P0.

P1	
destination	next_hop
200.0.0.0/8	P0
202.0.0.0/8	P2
201.0.0.0/16	D
201.1.0.0/16	E
200.2.16.0/20	C

This does work, but all C's inbound traffic except for that originating in P1 will now be routed through C's ex-provider P0, which as an *ex*-provider may not be on the best of terms with C. Also, the routing is inefficient: C's traffic from P2 is routed P2→P0→P1 instead of the more direct P2→P1.

A better solution is for *all* providers other than P1 to add the route ⟨200.2.16.0/20, P1⟩. While traffic to 200.0.0.0/8 otherwise goes to P0, this particular sub-block is instead routed by each provider to P1. The important case here is P2, as a stand-in for all other providers and their routers: P2 routes 200.0.0.0/8 traffic to P0 *except* for the block 200.2.16.0/20, which goes to P1.

Having every other provider in the world need to add an entry for C has the potential to cost some money, and, one way or another, C will be the one to pay. But at least there is a choice: C can consent to renumbering (which is not difficult if they have been diligent in using DHCP and perhaps NAT too), or they can pay to keep their old address block.

As for the second diagram above, with the various private links (shown as dashed lines), it is likely that the longest-match rule is *not* needed for these links to work. A's "private" link to P1 might only mean that

- A can send outbound traffic via P1
- P1 forwards A's traffic to A via the private link

P2, in other words, is still free to route to A via P0. P1 may not *advertise* its route to A to anyone else.

10.4.3 Hierarchical Routing via Providers

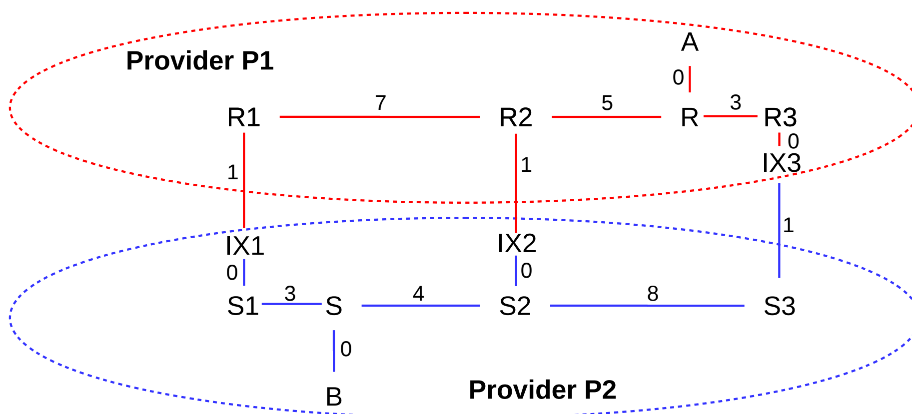
With provider-based routing, the route taken may no longer be end-to-end optimal; we have replaced the problem of finding an optimal route from A to B with the two problems of finding an optimal route from A to B's provider P, and then from P's entry point to B. This strategy mirrors the two-stage hierarchical routing

process of first routing on the address bits that identify the provider, and then routing on the address bits including the subscriber portion.

This two-stage strategy may not yield the same result as finding the globally optimal route. The result *will* be the same if B's customers can only be reached through P's single entry-point router RP, which models the situation that P and its customers look like a single site. However, either or both of the following can disrupt this model:

- There may be multiple entry-point routers into provider P's network, *eg* RP_1, RP_2 and RP_3, with different costs from A.
- P's customer B may have an alternative connection to the outside world via a different provider, as in the second diagram in *10.4 Provider-Based Routing*.

Consider the following example representing the first situation (the more important one in practice), in which providers P1 and P2 have three interconnection points IX1, IX2, IX3 (from Internet eXchange, *10.4.1 Internet Exchange Points*). Links are labeled with costs; we assume that P1's costs happen to be comparable with P2's costs.



The globally shortest path between A and B is via the R2–IX2–S2 crossover, with total length 5+1+0+4=10. However, traffic from A to B will be routed by P1 to its closest crossover to P2, namely the R3–IX3–S3 link. The total path is 3+0+1+8+4=16. Traffic from B to A will be routed by P2 via the R1–IX1–S1 crossover, for a length of 3+0+1+7+5=16. This routing strategy is sometimes called **hot-potato** routing; each provider tries to get rid of any traffic (the potatoes) as quickly as possible, by routing to the closest exit point.

Not only are the paths taken *inefficient*, but the A⟶B and B⟶A paths are now **asymmetric**. This can be a problem if forward and reverse timings are critical, or if one of P1 or P2 has significantly more bandwidth or less congestion than the other. In practice, however, route asymmetry is seldom important.

As for the route inefficiency itself, this also is not necessarily a significant problem; the primary reason routing-update algorithms focus on the *shortest* path is to guarantee that all computed paths are loop-free. As long as each half of a path is loop-free, and the halves do not intersect except at their common midpoint, these paths too will be loop-free.

The BGP "MED" value (*10.6.5.3 MULTI_EXIT_DISC*) offers an optional mechanism for P1 to agree that A⟶B traffic should take the r1–s1 crossover. This might be desired if P1's network were "better" and customer A was willing to pay extra to keep its traffic within P1's network as long as possible.

10.5 Geographical Routing

The classical alternative to provider-based routing is geographical routing; the archetypal model for this is the telephone area code system. A call from anywhere in the US to Loyola University's main switchboard, 773-274-3000, would traditionally be routed first to the 773 area code in Chicago. From there the call would be routed to the north-side 274 exchange, and from there to subscriber 3000. A similar strategy *can* be used for IP routing.

Geographical addressing has some advantages. Figuring out a good route to a destination is usually straightforward, and close to optimal in terms of the path physical distance. Changing providers never involves renumbering (though moving may). And approximate IP address geolocation (determining a host's location from its IP address) is automatic.

Geographical routing has some minor technical problems. First, routing may be inefficient between immediate neighbors A and B that happen to be split by a boundary for larger geographical areas; the path might go from A to the center of A's region to the center of B's region and then to B. Another problem is that some larger sites (*eg* large corporations) are themselves geographically distributed; if efficiency is the goal, each office of such a site would need a separate IP address block appropriate for its physical location.

But the real issue with geographical routing is apparently the business question of who carries the traffic. The provider-based model has a very natural answer to this: every link is owned by a specific provider. For geographical IP routing, my local provider might know at once from the prefix that a packet of mine is to be delivered from Chicago to San Francisco, but who will carry it there? My provider might have to enter into different traffic contracts for multiple different regions. If different local providers make different arrangements for long-haul packet delivery, the routing efficiency (at least in terms of table size) of geographical routing is likely lost. Finally, there is no natural answer for who should own those long inter-region links. It may be useful to recall that the present area-code system was created when the US telephone system was an AT&T monopoly, and the question of who carried traffic did not exist.

That said, the top five Regional Internet Registries represent geographical regions (usually continents), and provider-based addressing is *below* that level. That is, the IANA handed out address blocks to the geographical RIRs, and the RIRs then allocated address blocks to providers.

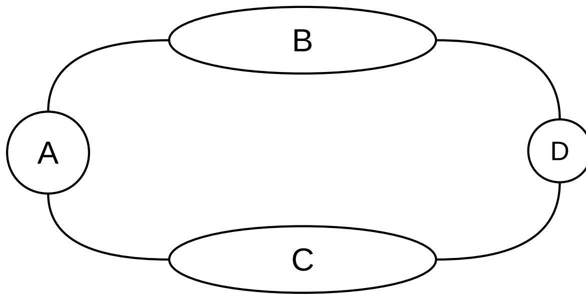
At the intercontinental level, geography does matter: some physical link paths are genuinely more expensive than other (shorter) paths. It is much easier to string terrestrial cable than undersea cable. However, within a continent physical distance does not always matter as much as might be supposed. Furthermore, a large geographically spread-out provider can always divide up its address blocks by region, allowing internal geographical routing to the correct region.

Here is a diagram of IP address allocation as of 2006: http://xkcd.com/195.

10.6 Border Gateway Protocol, BGP

In *9 Routing-Update Algorithms*, we considered **interior** routing-update protocols: those in which all the routers involved are under common management. That management can then dictate the routing-update protocol to be used, and also the rules for assigning per-link costs. For both Distance-Vector and Link State methods, the per-link cost played an essential role: by trying to minimize the cost, we were assured that no routing loops would be present in a stable network (*9.3 Observations on Minimizing Route Cost*).

But now consider the problem of **exterior** routing; that is, of choosing among routes that pass through independent organizations. In the diagram below, suppose that A, B, C and D are each managed independently; it may be useful to think of A, B and C as three ISPs and D as some destination.



Organization (or ISP) A has two routes to destination D – one via B and one via C – and must choose between them.

If A wanted to use one of the interior routing-update protocols to choose its path to D, it would face several purely technical problems. First, what if B uses distance-vector while C speaks only in link-state LSP messages? Second, what if B measures its path costs using the hopcount metric, while C assigns costs based on bandwidth, or congestion, or pecuniary considerations?

The mixing of unrelated metrics isn't necessarily useless: all that is required for the shortest-path-is-loop-free result mentioned above is that the two ends of each link agree on the cost assigned to that link. But apples-and-oranges comparison of different metrics *would* completely undermine the intended use of those metrics to influence the selection of which links should carry the most traffic. Sharing link-cost information without a common administrative policy to set those costs does not, in practical terms, make sense.

But A also faces a larger issue: to reach D it must rely on having its traffic carried by an *outsider* – either B or C. Outsiders are likely not inclined to offer this service without some form of compensation, either monetary or through reciprocal exchange. If A reaches an understanding with B on this matter of traffic carriage, then A does not want its traffic routed via C *even if that latter route is of lower technical cost*. If A is paying B, it is going to expect to use B. If A is not paying C, C is going to expect that A not use C.

The **Border Gateway Protocol**, or BGP, is assigned the job of handling exterior routing; that is, of handling exchange of routing information between neighboring independent organizations. The current version is BGP-4, documented in **RFC 4271**.

BGP's primary goal is to provide support for what are sometimes called **routing policies**; that is, for choosing routes based on *managerial* or *administrative* input. We address this in *10.6.4 BGP Filtering and Routing Policies*. (Routing policies have nothing to do with the policy-based routing described in *9.6 Routing on Other Attributes*, in which different packets with the same destination address may be routed differently because a site has a "policy" to take packet attributes other than destination into account. With BGP, once a site's policies to choose a route to a given destination are applied, all traffic to that destination takes that single route.)

Ultimately, the administrative input used by BGP very likely relates to who is paying what for the traffic carried. It is also possible, though less common, to use BGP to implement other preferences, such as for domestic traffic to remain within national boundaries.

The BGP term for a routing domain under coordinated administration, and using one consistent interior pro-

tocol and link-cost metric throughout, is **Autonomous System**, or AS. That said, all that is strictly required is that all BGP routers within an AS have the same consistent view of routing, and in fact some Autonomous Systems do run multiple routing protocols and may even use different metrics at different points. As indicated above, BGP does *not* support the exchange of link-cost information between Autonomous Systems.

Every non-leaf site (and some large leaf sites) has one or more **BGP speakers**: the routers that run BGP. If there is more than one, they must remain coordinated with one another so as to present a consistent view of the site's connections and advertisements; this coordination process is sometimes called **internal BGP** to distinguish it from the communication with neighboring Autonomous Systems. The latter process is then known as **external BGP**.

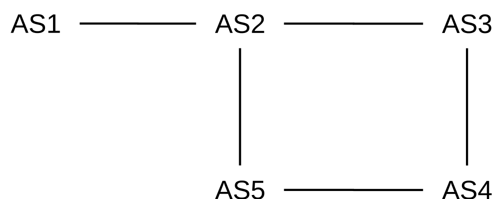
The BGP speakers of a site are often not the busy border routers that connect directly to the neighboring AS, though they are usually located near them and are often on the same subnet. Each interconnection point with a neighboring AS generally needs its own BGP speaker. Connections between BGP speakers of neighboring Autonomous Systems – sometimes called **BGP peers** – are generally configured administratively; they are not subject to a "neighbor discovery" process like that used by most interior routers.

The BGP speakers must maintain a database of all routes received, not just of the routes actually used. However, the speakers exchange with neighbors only the routes they (and thus their AS) use themselves; this is a firm BGP rule.

10.6.1 AS-paths

At its most basic level, BGP involves the exchange of lists of reachable destinations, like distance-vector routing without the distance information. But that strategy, alone, cannot detect routing loops. BGP solves the loop problem by having routers exchange, not just destination information, but also the entire path used to reach each destination. Paths including each router would be cumbersome; instead, BGP abbreviates the path to the list of AS's traversed. This is called the **AS-path**. This allows routers to make sure their routes do not traverse any AS more than once, and thus do not have loops.

As an example of this, consider the network below, in which we consider Autonomous Systems also to be destinations. Initially, we will assume that each AS discovers its immediate neighbors. AS3 and AS5 will then each advertise to AS4 their routes to AS2, but AS4 will have no reason at this level to prefer one route to the other (BGP does use the shortest AS-path as part of its tie-breaking rule, but, before falling back on that rule, AS4 is likely to have a *commercial* preference for which of AS3 and AS5 it uses to reach AS2).



Also, AS2 will advertise to AS3 its route to reach AS1; that advertisement will contain the AS-path ⟨AS2,AS1⟩. Similarly, AS3 will advertise this route to AS4 and then AS4 will advertise it to AS5.

When AS5 in turn advertises this AS1-route to AS2, it has the *potential* to create a loop. It does not, however, because it will include the entire AS-path ⟨AS5,AS4,AS3,AS2,AS1⟩ in the advertisement it sends to AS2.

AS2 will know not to use this route because it will see that it is a member of the AS-path. Thus, BGP is spared the kind of slow-convergence problem that traditional distance-vector approaches were subject to.

It is theoretically possible that the shortest path (in the sense, say, of the hopcount metric) from one host to another traverses some AS twice. If so, BGP will not allow this route.

AS-paths potentially add considerably to the size of the AS database. The number of paths a site must keep track of is proportional to the number of AS's, because there will be one AS-path to each destination AS. (Actually, an AS may have to record many times that many AS-paths, as an AS may hear of AS-paths that it elects not to use.) Typically there are several thousand AS's in the world. Let A be the number of AS's. Typically the average length of an AS-path is about log(A), although this depends on connectivity. The amount of memory required by BGP is

$$C \times A \times \log(A) + K \times N,$$

where C and K are constants.

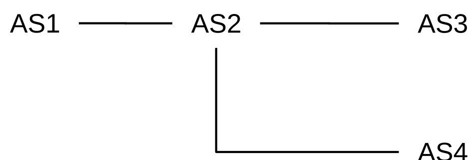
The other major goal of BGP is to allow **administrative** input to what, for interior routing, is largely a technical calculation (though an interior-routing administrator can set link costs). BGP is the interface between ISPs (and between ISPs and their larger customers), and can be used to implement *contractual* agreements made regarding which ISPs will carry other ISPs' traffic. If ISP2 tells ISP1 it has a route to destination D, but ISP1 chooses not to send traffic to ISP2, BGP can be used to implement this. Perhaps more likely, if ISP2 has a route to D but does not want ISP1 to use it until they pay for the privilege, BGP can be used to implement this as well.

Despite the exchange of AS-path information, temporary routing loops may still exist. This is because BGP may first decide to use a route and only then export the new AS-path; the AS on the other side may realize there is a problem as soon as the AS-path is received but by then the loop will have at least briefly been in existence. See the first example below in *10.6.8 Examples of BGP Instability*.

BGP's predecessor was EGP, which guaranteed loop-free routes by allowing only a single route to any AS, thus forcing the Internet into a tree topology, at least at the level of Autonomous Systems. The AS graph could contain no cycles or alternative routes, and hence there could be no redundancy provided by alternative paths. EGP also thus avoided having to make decisions as to the preferred path; there was never more than one choice. EGP was sometimes described as a reachability protocol; its only concern was whether a given network was reachable.

10.6.2 AS-Paths and Route Aggregation

There is some conflict between the goal of reporting precise AS-paths to each destination, and of consolidating as many address prefixes as possible into a single prefix (single CIDR block). Consider the following network:



Suppose AS2 has paths

path=⟨AS2⟩, destination 200.0.0/23
path=⟨AS2,AS3⟩, destination 200.0.2/24
path=⟨AS2,AS4⟩, destination 200.0.3/24

If AS2 wants to optimize address-block aggregation using CIDR, it may prefer to aggregate the three destinations into the single block 200.0.0/22. In this case there would be two options for how AS2 reports its routes to AS1:

- **Option 1**: report 200.0.0/22 with path ⟨AS2⟩. But this ignores the AS's AS3 and AS4! These are legitimately part of the AS-paths to some of the destinations within the block 200.0.0/22; loop detection could conceivably now fail.
- **Option 2**: report 200.0.0/22 with path ⟨AS2,AS3,AS4⟩, which is not a real path but which does include all the AS's involved. This ensures that the loop-detection algorithm works, but artificially inflates the length of the AS-path, which is used for certain tie-breaking decisions.

As neither of these options is desirable, the concept of the **AS-set** was introduced. A list of Autonomous Systems traversed in order now becomes an **AS-sequence**. In the example above, AS2 can thus report net 200.0.0/22 with

- AS-sequence=⟨AS2⟩
- AS-set={AS3,AS4}

AS2 thus both achieves the desired aggregation and also accurately reports the AS-path length.

The AS-path can in general be an arbitrary list of AS-sequence and AS-set parts, but in cases of simple aggregation such as the example here, there will be one AS-sequence followed by one AS-set.

RFC 6472 now recommends against using AS-sets entirely, and recommends that aggregation as above be avoided.

10.6.3 Transit Traffic

It is helpful to distinguish between two kinds of traffic, as seen from a given AS. **Local** traffic is traffic that either originates or terminates at that AS; this is traffic that "belongs" to that AS. At leaf sites (that is, sites that connect only to their ISP and not to other sites), all traffic is local.

The other kind of traffic is **transit** traffic; the AS is forwarding it along on behalf of some nonlocal party. For ISPs, most traffic is transit traffic. A large almost-leaf site might also carry a small amount of transit traffic for one particular related (but autonomous!) organization.

The decision as to whether to carry transit traffic is a classic example of an administrative choice, implemented by BGP's support for routing policies. Most real-world BGP configuration issues relate to the carriage (or non-carriage) of transit traffic.

10.6.4 BGP Filtering and Routing Policies

As stated above, one of the goals of BGP is to support **routing policies**; that is, routing based on managerial or administrative concerns in addition to technical ones. A BGP speaker may be aware of multiple routes to

a destination. To choose the one route that we will use, it may combine a mixture of optimization rules and policy rules. Some examples of policy rules might be:

- do not use AS13 as we have an adversarial relationship with them
- do not allow transit traffic

BGP implements policy through **filtering** rules – that is, rules that allow rejection of certain routes – at three different stages:

1. **Import filtering** is applied to the lists of routes a BGP speaker receives from its neighbors.
2. **Best-path selection** is then applied as that BGP speaker chooses which of the routes accepted by the first step it will actually use.
3. **Export filtering** is done to decide what routes from the previous step a BGP speaker will actually advertise. A BGP speaker can only advertise paths it uses, but does not have to advertise every such path.

While there are standard default rules for all these (accept everything imported, use simple tie-breakers, export everything), a site will usually implement at least some **policy rules** through this filtering process (*eg* "prefer routes through the ISP we have a contract with").

As an example of import filtering, a site might elect to ignore all routes from a particular neighbor, or to ignore all routes whose AS-path contains a particular AS, or to ignore temporarily all routes from a neighbor that has demonstrated too much recent "route instability" (that is, rapidly changing routes). Import filtering *can* also be done in the best-path-selection stage, by having the best-path-selection process ignore routes from selected neighbors.

BGP Breakdowns

In the real world, it sometimes happens that a small regional ISP is misconfigured to attempt to report to some high-level provider that it can reach, say, every site in the world. Export filtering on the part of the small ISP and best-path selection and import filtering on the part of the large ISP usually – though not always – catches this. Occasionally, such incidents represent malicious **BGP hijacking**.

The next stage is **best-path selection**, to pick the preferred routes from among all those just imported. The first step is to eliminate AS-paths with loops. Even if the neighbors have been diligent in not advertising paths with loops, an AS will still need to reject routes that contain itself in the associated AS-path.

The next step in the best-path-selection stage, generally the most important in BGP configuration, is to assign a **local_preference**, or weight, to each route received. An AS may have policies that add a certain amount to the local_preference for routes that use a certain AS, *etc*. Very commonly, larger sites will have preferences based on contractual arrangements with particular neighbors. Provider AS's, for example, will in general prefer routes learned from their customers, as these are "cheaper". A smaller ISP that connects to two larger ones might be paying to route the majority of its outbound traffic through a particular one of the two; its local_preference values will then *implement* this choice. After BGP calculates the local_preference value for every route, the routes with the best local_preference are then selected.

Domains are free to choose their local_preference rules however they wish. In principle this can involve rather strange criteria; for example, in *10.6.8 Examples of BGP Instability* we will consider an example

where AS1 prefers routes with AS-path ⟨AS3,AS2⟩ to the strictly shorter path ⟨AS2⟩. That example, however, demonstrates instability; domains are encouraged to set their rules in accordance with some standard principles, below, to avoid this.

Local_preference values are communicated internally via the LOCAL_PREF path attribute, below. They are not shared with other Autonomous Systems.

In the event of ties – two routes to the same destination with the same local_preference – a first tie-breaker rule is to prefer the route with the shorter AS-path. While this superficially resembles a shortest-path algorithm, the real work should have been done in administratively assigning local_preference values. The shorter-AS-path tie-breaker is perhaps best thought of as similar in spirit to the smaller-AS-number tie-breaker (although the sometimes-significant Multi-Exit-Discriminator tie-breaker, next, comes between them).

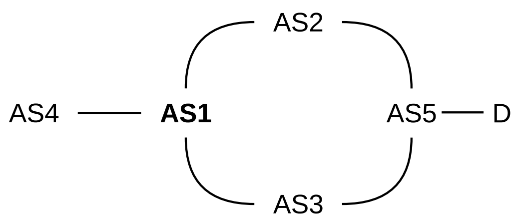
The final significant step of the route-selection phase is to apply the Multi-Exit-Discriminator value, *10.6.5.3 MULTI_EXIT_DISC*. A site may very well choose to ignore this value entirely.

Finally we get to the trivial tie-breaker rules, though if a tie-breaker rule assigns significant traffic to one AS over another then it may have economic consequences and shouldn't be considered "trivial". If this situation is detected, it would probably be addressed in the local-preferences phase. The trivial tie-breakers take into account the internal routing cost, the numeric value of the AS number, and the numeric value of the neighbor's IP address.

After the best-path-selection stage is complete, the BGP speaker has now selected the routes *it* will use. The final stage is to decide what rules will be **exported** to which neighbors. Only routes the BGP speaker will use – that is, routes that have made it to this point – can be exported; a site cannot route to destination D through AS1 but export a route claiming D can be reached through AS2.

It is at the export-filtering stage that an AS can enforce no-transit rules. If it does not wish to carry transit traffic to destination D, it will not advertise D to any of its AS-neighbors.

The export stage can lead to anomalies. Suppose, for example, that AS1 reaches D and AS5 via AS2, and announces this to AS4.



Later, we imagine, AS1 switches to reaching D via AS3, but is *forbidden by policy* to announce to AS4 any routes with AS-path containing AS3; such a policy is straightforward to implement via export filtering. Then AS1 must simply withdraw the announcement to AS4 that it could reach D at all, even though the route to D via AS2 is still there.

10.6.5 BGP Path attributes

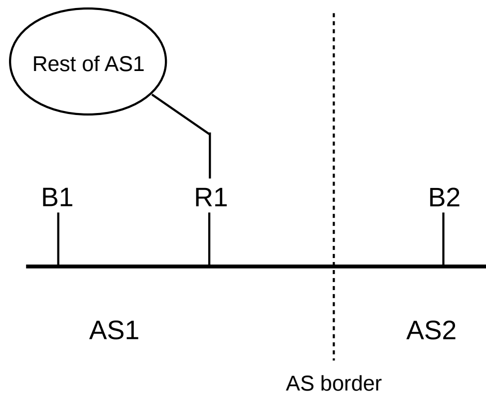
BGP supports the inclusion of various **path attributes** when exchanging routing information. Attributes exchanged with neighbors can be **transitive** or **non-transitive**; the difference is that if a neighbor AS does

not recognize a received path attribute then it should pass it along anyway if it is marked transitive, but not otherwise. Some path attributes are entirely **local**, that is, internal to the AS of origin. Other flags are used to indicate whether recognition of a path attribute is required or optional, and whether recognition can be partial or must be complete.

The AS-path itself is perhaps the most fundamental path attribute. Here are a few other common attributes:

10.6.5.1 NEXT_HOP

This mandatory external attribute allows BGP speaker B1 of AS1 to inform its BGP peer B2 of AS2 what actual router to use to reach a given destination. If B1, B2 and AS1's actual border router R1 are all on the same subnet, B1 will include R1's IP address as its NEXT_HOP attribute. If B1 is *not* on the same subnet as B2, it may not know R1's IP address; in this case it may include its own IP address as the NEXT_HOP attribute. Routers on AS2's side will then look up the "immediate next hop" they would use as the first step to reach B1, and forward traffic there. This should either be R1 or should lead to R1, which will then route the traffic properly (*not* necessarily on to B1).



10.6.5.2 LOCAL_PREF

If one BGP speaker in an AS has been configured with local_preference values, used in the best-path-selection phase above, it uses the LOCAL_PREF path attribute to share those preferences with all other BGP speakers at a site. In other words, once one BGP speaker has determined the local_preference value of a given route, the LOCAL_PREF attribute is used to distribute that value uniformly throughout the AS.

10.6.5.3 MULTI_EXIT_DISC

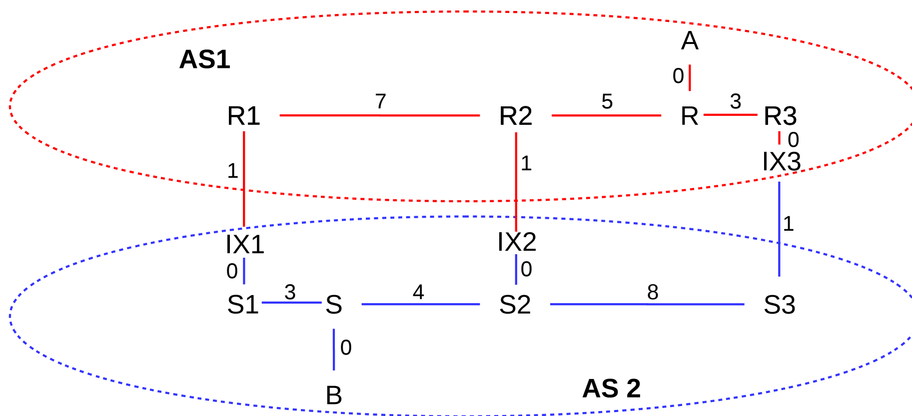
The Multi-Exit Discriminator, or **MED**, attribute allows one AS to learn something of the internal structure of another AS, *should it elect to do so*. Using the MED information provided by a neighbor has the potential to cause an AS to incur higher costs, as it may end up carrying traffic for longer distances internally; MED values received from a neighboring AS are therefore only recognized when there is an explicit administrative decision to do so.

Specifically, if an autonomous system AS1 has multiple links to neighbor AS2, then AS1 can, when advertising an internal destination D to AS2, have each of its BGP speakers provide associated MED values so

that AS2 can know which link AS1 would prefer that AS2 use to reach D. This allows AS2 to route traffic to D so that it is carried primarily by AS2 rather than by AS1. The alternative is for AS2 to use only the closest gateway to AS1, which means traffic is likely carried primarily by AS1.

MED values are considered late in the best-path-selection process; in this sense the use of MED values is a tie-breaker when two routes have the same local_preference.

As an example, consider the following network (from *10.4.3 Hierarchical Routing via Providers*, with providers now replaced by Autonomous Systems); the numeric values on links are their relative costs. We will assume that each site has three BGP speakers co-located at the exchange points IX1, IX2 and IX3.



In the absence of the MED, AS1 will send traffic from A to B via the R3–IX3–S3 link, and AS2 will return the traffic via S1–IX1–R1. These are the links that are closest to R and S, respectively, representing AS1 and AS2's desire to hand off the outbound traffic as quickly as possible.

However, AS1's BGP speakers at IX1, IX2 and IX3 can provide MED values to AS2 when advertising destination A, indicating a preference for AS2→AS1 traffic to use the rightmost link:

- IX1: destination A has MED 200
- IX2: destination A has MED 150
- IX3: destination A has MED 100

If this is done, and AS2 abides by this information, then AS2 will route traffic from B to A via IX3; that is, via the exchange point with the lowest MED value. Note the importance of fact that AS2 is allowed to ignore the MED; use of it may shift costs from AS1 to AS2!

The relative order of the MED values for R1 and R2 is irrelevant, unless the IX3 exchange becomes disabled, in which case the numeric MED values above would mean that AS2 should then prefer IX2 for reaching A.

We cannot use MED values to cause A–B traffic to take the path through IX2; that path has minimal cost only in the global sense, and the only way to achieve global cost minimization is for the two AS's to agree to use a common distance metric and a common metric-based routing algorithm, in effect becoming one AS. While AS1 does provide different numeric MED values for the three exchange points, they are used only in ranking precedence, not as numeric measures of cost (though they are sometimes derived from that).

In the example above, importing and using MED values *raises* AS2's costs, by causing it to route AS2-to-AS1 traffic so that it stays for a longer path within AS2's network. This is, in fact, almost always the case when using MED values. Why, then, would AS2 agree to this? One simple reason might be that AS2 and AS1 have, together, negotiated this arrangement; perhaps AS1 gives AS2 a break on interconnection ("peering") fees in exchange for AS2's accepting and using AS1's MED data. It is also possible that AS2's use of AS1's MED data may improve the quality of service AS2 can offer to its customers; we will return to an example of this in *10.6.6.1 MED values and traffic engineering*.

Also in the example above, the MED values are used to decide between multiple routes to the same destination that all pass through the *same* AS, namely AS1. Some BGP implementations allow the use of MED values to decide between different routes through different neighbor AS's. The different neighbors must all have the same local_preference values. For example, AS2 might connect to AS3 and AS4 and receive the following BGP information:

- AS3: destination A has MED 200
- AS4: destination A has MED 100

Assuming AS2 assigns the same local_preference to AS3 and AS4, it might be configured to use these MED values as the tie-breaker, and thus routing traffic to A via AS3. On Cisco routers, the **always-compare-med** command is used to create this behavior.

MED values are not intended to be used to communicate routing preferences to non-neighboring AS's.

Additional information on the use of MED values can be found in **RFC 4451**.

10.6.5.4 COMMUNITY

This is simply a tag to attach to routes. Routes can have multiple tags corresponding to membership in multiple communities. Some communities are defined globally; for example, NO_EXPORT and NO_ADVERTISE. A route marked with one of these two communities will not be shared further. Other communities may be relevant only to a particular AS.

The importance of communities is that they allow one AS to place some of its routes into specific categories when advertising them to another AS; the categories must have been created and recognized by the receiving AS. The receiving AS is not obligated to honor the community memberships, of course, but doing so has the effect of allowing the original AS to "configure itself" without involving the receiving AS in the process. Communities are often used, for example, by (large) customers of an ISP to request specific routing treatment.

A customer would have to find out from the provider what communities the provider defines, and what their numeric codes are. At that point the customer can place itself into the provider's community at will.

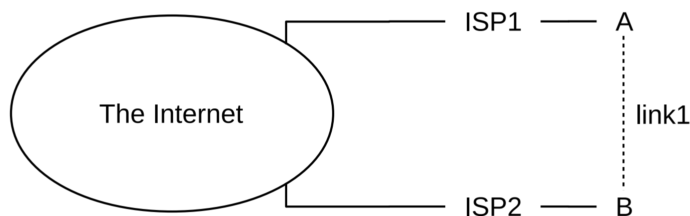
Here are some of the community values once supported by a no-longer-extant ISP that we shall call AS1. The full community value would have included AS1's AS-number.

value	action
90	set local_preference used by AS1 to 90
100	set local_preference used by AS1 to 100, the default
105	set local_preference used by AS1 to 105
110	set local_preference used by AS1 to 110
990	the route will not leave AS1's domain; equivalent to NO_EXPORT
991	route will only be exported to AS1's other customers

10.6.6 BGP and Traffic Engineering

BGP is *the* mechanism for inter-autonomous-system traffic engineering. The first-line tools are import and export filtering and best-path selection. For autonomous systems with multiple interconnection points, the Multi-Exit Discriminator above also may play a large role.

After establishing basic connectivity, perhaps the most important decision a site makes via its BGP configuration is whether or not it will accept **transit traffic**. As a first example of this, let us consider the case of configuring a private link, such as the dashed link1 below between "friendly" but unaffiliated sites A and B (link1 can be either a shared "real" link or a short "jumper" link within an Internet exchange point):



Suppose A exports its link1 route to B to its provider ISP1. Then ISP1 may in turn announce this route to the Internet at large, and so some or all of B's inbound traffic may be routed through ISP1 (paid by A) and through A itself. Similarly, B may end up paying to carry A's traffic if B exports its link1 route to A to ISP2.

Economically, carrying someone else's transit traffic not desirable unless you are compensated for it. The primary issue here is the use of the ISP1–A link by B and the ISP2–B link by A; use of the shared link1 *might* be a secondary issue depending on the relative bandwidths and A and B's understandings of appropriate uses for link1.

Two common options A and B might agree to regarding link1 are **no-transit** and **backup**.

For the **no-transit** option, A and B simply do not export the route to their respective ISPs at all. This is done via export filtering. If ISP1 does not know A can reach B, it will not send any of B's traffic to A.

For the **backup** option, the intent is that traffic to A will normally arrive via ISP1, but if the ISP1 link is down then A's traffic will be allowed to travel through ISP2 and B. To achieve this, A and B can export their link1-route to each other, but arrange for ISP1 and ISP2 respectively to assign this route a low local_preference value. As long as ISP1 hears of a route to B from *its* upstream provider, it will reach B that way, and will not advertise the existence of the link1 route to B; ditto ISP2. However, if the ISP2 route to B fails, then A's upstream provider will stop advertising any route to B, and so ISP1 will begin to use the link1 route to B and begin advertising it to the Internet. The link1 route will be the primary route to B until ISP2's service is restored.

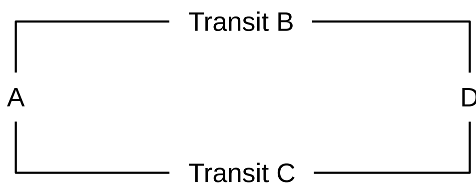
A and B must convince their respective ISPs to assign the link1 route a low local_preference; they cannot mandate this directly. However, if their ISPs recognize **community** attributes that, as above, allow customers to influence their local_preference value, then A and B can use this to create the desired local_preference.

To use the shared link for backup *outbound* traffic, A and B will need a way to send through one another if their own ISP link is down. If A detects that its ISP link is down, it can simply change its default route to point to B. One way to automate this is for A and B to view their default-route path (*eg* to 0.0.0.0/0) to be a concrete destination within BGP. ISP1 advertises this to A, using BGP, but so does B, and A has configured its import rules so B's route to 0.0.0.0/0 has a higher cost. Then A will route to 0.0.0.0/0 through ISP1 – that is, will use ISP1 as its default route – as long as it is available, and will switch to B when it is not.

A and B might also wish to use their shared private link for **load balancing**, but for this BGP offers limited help. If ISP1 and ISP2 both export routes to A, then A has lost all control over how other sites will prefer one to the other. A may be able to make one path artificially appear more expensive, perhaps by duplicating one of the ISPs in the AS-path. A might then be able to keep tweaking this cost until the inbound loads are comparable, but there is no guarantee (or even likelihood) this will be stable. Outbound load-balancing is up to A and B's respective internal routers.

Providers in the business of carrying transit traffic must also make decisions about exactly whose traffic they will carry; these decisions are again implemented with BGP. In the diagram below, two transit-providing Autonomous Systems B and C connect to individual sites (or regional ISPs) A and D.

In the diagram above, the left and right interconnections are shown taking place at Internet exchange points IXP1 and IXP2 (*10.4.1 Internet Exchange Points*). IXPs are typically where such interconnections take place but are not required; the essential topology is simply this:



B would like to make sure C does not attempt to save on its long-haul transit costs by forwarding A⟶D traffic over to B at IXP1, and D⟶A traffic over to B at IXP2. B avoids this problem by not advertising to C that it can reach A and D, and similarly with C. Transit providers are quite careful about not advertising reachability to any other AS for whom they do not intend to provide transit service, because to do so is likely to mean getting stuck with that traffic.

If B advertises to A that it can reach D, then A may accept that route, and send all its D-bound traffic via B, with C not involved at all. B is not likely to do this unless A pays for the privilege. If B and C *both* advertise to A that they can reach D, then A has a choice, which it will make via its best-path-selection rules. But in such a case A will want to be sure that it does not end up paying full price to both B and C to carry its traffic

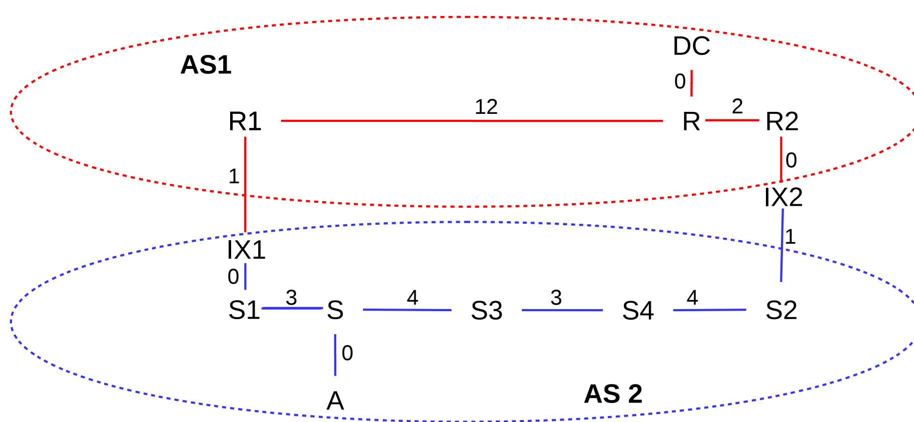
while using only one of them. Site A might, for example, agree to payment based on the actual volume of carried traffic, meaning that if it prefers B's route then it will pay only B.

It is quite possible that B advertises to A that it can reach D, but does not advertise to D that it can reach A. As we have seen, B advertises to A that it can reach D only if A has paid for this privilege; perhaps D prefers to do business with C rather than with B. In that case, A-to-D traffic would travel via B, while D-to-A traffic would travel via C.

In the unlikely event that B and C both advertise to one another at IXP1 their route to D, a routing loop may even be created. B might forward D-bound traffic to C while C forwards it back to B. But in that case B would state, in its next BGP advertisement to C at IXP1, that it reaches D via an AS-path that begins with C, and C would do similarly. B and C would then see themselves in the AS-paths they receive and would stop using these routes.

10.6.6.1 MED values and traffic engineering

Let us now address why an AS would bother with importing and using MED values, given that doing so will almost always increase the site's cost. Consider the following diagram of autonomous systems AS1 and AS2, with link costs shown:



Site DC in the diagram above is a data center that wants its user – at site A – to experience high-performance downloads. Perhaps DC delivers high-performance streaming video, and needs to minimize both congestion and packet losses. In order to achieve this superior quality, it builds a particularly robust network R1–R–R2, shown above as AS1.

A first step is to have AS1 connect (or peer) directly to customer networks such as AS2, rather than relying on the Internet backbone. Two such interconnection points are shown above, IX1 and IX2.

At this point, traffic from A to DC will take IX1 (on the shortest path from A to AS1), and so will travel most of the way in AS1. This is good, but traffic from A to DC is probably mostly acknowledgments; these are unlikely to benefit from the special network. The actual *data*, sent from DC to A, will take IX2, because that is AS1's shortest path to reach AS2. The data will thus travel most of the way in AS2, bypassing AS1's high-performance network. This is not what DC wants.

However, the picture changes if AS1 agrees to accept MED information from AS2 (and other providers).

If AS2 tells AS1 that AS2's preferred link for reaching A is via IX1, then traffic from DC to A will travel through R1 to IX1, and from there onto A. This keeps DC's *outbound* traffic in the AS1 network as long as possible, instead of handing it off to the other network of lower quality. This *is* what DC wants; this is why DC built the high-performance network.

Rather than building its own high-performance network, DC might simply contract with an existing high-performance network. That would make AS1's business model the following:

- peering with as many potential customer networks as possible
- importing and using the MED information from those networks
- advertising to potential customers like DC that their network will give DC's users a better experience

10.6.7 BGP Relationships

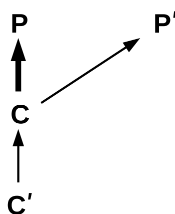
Arbitrarily complex policies may be created through BGP, and, as we shall see in the following section, convergence to a stable set of routes is not guaranteed. Nonconvergence does not mean distance-vector's "slow convergence to infinity", but rather a regular oscillation of routes among competing alternatives.

It turns out, however, that if some constraints are applied to the different AS-to-AS relationships, then better behavior is obtained. The paper *[LG01]* analyzed BGP networks in which each AS-to-AS relationship fit one of the following three business patterns, discussed further below:

1. Customer to provider (the most common pattern)
2. Peer to peer (*eg* two top-level providers mutually exchanging traffic)
3. Sibling to sibling (for very close AS-to-AS relationships)

A major consequence these relationships is the extent to which the autonomous systems involved accept one another's "non-customer" routes (below), and hence the extent to which they provide each other with transit services. We start with the most basic case, that of customer and provider.

If autonomous systems C and P have a **customer-to-provider** relationship, with C as the customer and P as the provider, then C is paying P to carry some or all of its traffic to the "outside world". P may not carry all such traffic, because C may also be a customer of another provider P′. C may also have its own sub-customers, such as C′:



In offering itself as a provider, P will export all the routes it has, from all sources, to C, in effect telling C "this is what I can reach". If C has no other providers it might accept these routes in the form of a single default-route entry pointing to P; if C has another provider P′ then it might accept some routes from P and some from P′.

Similarly, C will always export its own routes to P. If C has customers of its own, such as C′, then it will also export those routes to P. Collectively, we will say that C's own routes and the routes of its own customers and sub-customers are its **customer routes**.

But what about **non**-customer routes, *eg* routes learned from other providers? These C generally **does not** export. If C were to export to P a route to destination D that it learned from second provider P′, then C might end up providing transport service to P, carrying P's D-bound traffic to P′. As a **customer**, this is probably not what C intends.

To summarize, a provider **does** export its non-customer routes to its customer, but a customer generally **does not** export its non-customer routes to its providers. This rule is not, in the world of real business relationships, absolute; AS's may negotiate all sorts of special arrangements. A nominal customer might, for example, agree to provide transit service for some set of destinations, in exchange for a lower-priced rate for the handling of its other traffic. Nonetheless, the rule is largely accurate, and provides a helpful starting point to understanding customer-provider relationships. Below, in *10.6.7.1 BGP No-Valley Theorem*, we will in effect use this rule as a *definition* of customer-provider relationships.

Now let us consider a **peer-to-peer** relationship, which is a connection between two transit providers that have agreed to exchange all their customer traffic with each other; thus carrying transit traffic for one another. Often the idea is for the interconnection to be seen as equally valuable by both parties (*eg* because the parties exchange comparable volumes of traffic); in such a case the relationship would likely be "settlement-free", that is, involving no monetary exchange. If, however, the volume flow is significantly asymmetric then compensation can certainly be negotiated, making the relationship more like customer-to-provider.

As with customers and providers, two peers P1 and P2 each export all their customer routes to the other; that way, P2 knows it how to reach P1's customers and vice-versa. By doing this, P1 and P2 each carry transit traffic for their own customers.

Peers **do not**, however, generally export their non-customer routes, in either direction. If P1 learns of a route to destination D from another peer (or provider) P3, it does not export this to P2. If it were to do so, then P1 would carry non-customer transit traffic from P2 to P3. Instead, P2 is expected also to peer with P3, and learn of P3's route to D that way. Alternatively, P3 can become a customer of P1, and thus pay for P1's transit carriage of P3's traffic.

The so-called **tier-1** providers are those that are not customers of anyone; these represent the top-level "backbone" providers. Each tier-1 AS must, as a rule, peer with every other tier-1 AS, though AS's are free to negotiate exceptions.

Finally, some autonomous system relationships that do not fit the customer-to-provider or peer-to-peer patterns can be characterized as **sibling-to-sibling**. Siblings are ISPs that have a close relationship; often siblings are AS's that, due to mergers, are now part of the same organization. Siblings may also be nominal competitors who intend to use their mutual link as a cooperative backup, as in *10.6.6 BGP and Traffic Engineering*. Two siblings may or may not have the same upstream ISP as provider.

Siblings typically export everything to one another – both customer and non-customer routes – and thus **do** potentially use their connection for transit traffic in both directions (although they **may** rank routes through one another at low preference, so as to use the shared link only when nothing else is available).

We can summarize the three kinds of relationships in terms of how they export non-customer routes:

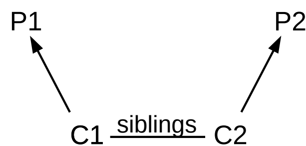
- in peer-to-peer relationships, non-customer routes are not exported in either direction.

- in customer-to-provider relationships, non-customer routes are exported only from the provider to the customer.
- in sibling-to-sibling relationships, non-customer routes are exported in both directions.

It is possible to make at least some inferences about BGP relationships from sites' actual export information, though accuracy is imperfect because sites may negotiate non-standard arrangements; see *[LG01]*.

In the real world, BGP sibling relationships are relatively rare, probably because they do not really fit the model of traffic carriage as a service. This may be fortunate, as sibling relationships, with universal and bidirectional route export, tend to introduce the greatest complexity. The non-convergence examples of *10.6.8 Examples of BGP Instability* all require sibling relationships.

One problematic sibling case is the following, in which P1 and P2 are providers for C1 and C2, respectively, and C1 and C2 are siblings:



Suppose P1 exports to C1 a route to destination D. C1 then exports it to sibling C2. If C2 treats this as a customer route, it will export it to P2, in which case C1 and C2 are now providing transit service to traffic from P2 bound for D.

Sibling relationships can be tamed considerably, however, if we adopt a requirement that collections of linked siblings act as a unit, keeping track of the original non-sibling source (that is, customer, provider or peer) of each route. Let us say that autonomous systems S and S′ are in the same **sibling family** if there is a chain of autonomous systems $S_0...S_n$ so that $S=S_0$, $S_n=S'$, and each consecutive S_{i-1} and S_i, $i\leqslant n$, are siblings. We can then define the following property:

> **Selective Export Property**: A sibling family satisfies this property if, whenever one member of the family learns of a route from a provider (respectively peer or customer) then all other members of the family treat the route as a provider (respectively peer or customer) route when deciding whether to export.

In other words, in the situation diagrammed above, in which C1 has learned of a route to D from its provider P1, C2 will also treat this route as a non-customer route and will not export it to P2.

In the real world, BGP relationships may not fit any of the above three categories, or else there may be many sibling relationships for which the selective-export property fails. However, quite often these relationships do hold to a useful degree.

We can also specialize the relationships to a particular set of destinations, or even to an individual destination; for example, autonomous systems C and P might be said to have a customer-to-provider relationship for destination D if C learned its route to D from a non-customer, does not export this route to P, and P does export to C its own route to D.

BGP certainly allows for complicated variations: if a regional provider is a customer of a large transit backbone, then the backbone might only announce routes listed in transit agreement (rather than all routes, as above). There is a supposition here that the regional provider has multiple connections, and has contracted with that particular transit backbone only for certain routes. But we can fit this into the classification above

either by restricting attention to the set of routes listed in the agreement, or by declaring that in principle the transit provider exports all routes, but the regional customer doesn't import the ones it hasn't paid for.

10.6.7.1 BGP No-Valley Theorem

A consequence of adherence to the above classification and attendant export rules is the **no-valley theorem** of *[LG01]*: Suppose every pair of adjacent AS's has a relationship described by the customer-provider, peer-to-peer or sibling rules above (now taken to be *definitions* of these three relationships). In addition, every sibling family abides by the selective-export property. Let $A=A_0$ be an autonomous system that has received a route to destination D with AS-path $\langle A_1,A_2,...,A_n\rangle$. **Then**: in this AS-path, there is at most one peer-to-peer link. Links to the left of the peer-to-peer link (that is, closer to A) are either customer→provider links or sibling→sibling links; that is, they are non-downwards. To the right of the peer-to-peer link, there are only provider→customer or sibling→sibling links; that is, these are non-upwards. If there is no peer-to-peer link, then we can still divide the AS-path into a non-downwards first part and a non-upwards second part.

Intuitively, autonomous systems on the right (non-upwards) part of the path export the route to D as a customer route. Autonomous systems on the left (non-downwards) part of the path export the route from provider to customer.

The no-valley theorem can be seen as an illustration of the power of the restrictions built into the customer-to-provider and peer-to-peer export rules.

We give an informal argument for the case in which the AS-path has no peer-to-peer link. First, note that BGP rules mean that each autonomous system AS_i in the path has received the route to D from neighbor AS_{i+1} with AS-path $\langle A_{i+1},...,A_n\rangle$.

If the no-valley theorem were to fail, then somewhere along the AS-path in order of increasing i we would have a downward link followed by, eventually, an upward link. Choose the largest i for which this arrangement appears, and let k be the position of the first subsequent upward link, so that

- A_i to A_{i+1} is provider-to-customer
- A_j to A_{j+1} is sibling-to-sibling for $i<j<k-1$
- A_{k-1} to A_k is customer-to-provider.

Then the route to D was acquired by A_{k-1} from its provider A_k, and so is a provider route. The set $\{A_{i+1},...,A_{k-1}\}$ is a sibling family, and so by the selective-export rule A_{i+1} also treats this route to D as a provider route. It therefore cannot export this non-customer route to different provider A_i, a contradiction.

For the case with a peer-to-peer edge, see exercise 12.0.

If the hypotheses of the no-valley theorem hold only for routes involving a particular destination or set of destinations, then the theorem is still true for those routes.

The hypotheses of the no-valley theorem are not quite sufficient to guarantee convergence of the BGP system to a stable set of routes. To ensure convergence in the case without sibling relationships, it is shown in *[GR01]* that the following simple local_preference rule suffices:

> If AS1 gets two routes r1 and r2 to a destination D, and the first AS of the r1 route is a customer of AS1, and the first AS of r2 is not, then r1 will be assigned a higher local_preference value than r2.

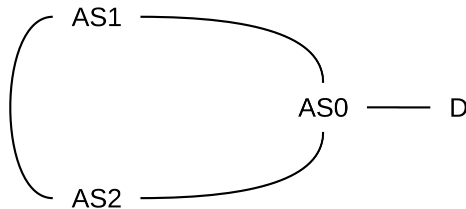
More complex rules exist that allow for cases when the local_preference values can be equal; one such rule states that strict inequality is only required when r2 is a provider route. Other straightforward rules handle the case of sibling relationships, *eg* by requiring that siblings have local_preference rules consistent with the use of their shared connection only for backup.

As a practical matter, whether or not actual BGP relationships are consistent with the rules above, arrangements resulting in actual BGP instability appear rare on the Internet.

10.6.8 Examples of BGP Instability

What if the "normal" rules regarding BGP preferences are *not* followed? It turns out that BGP allows genuinely unstable situations to occur; this is a consequence of allowing each AS a completely independent hand in selecting preference functions. Here are two simple examples, from *[GR01]*.

Example 1: A stable state exists, but convergence to it is not guaranteed. Consider the following network arrangement:



We assume AS1 prefers AS-paths to destination D in the following order:

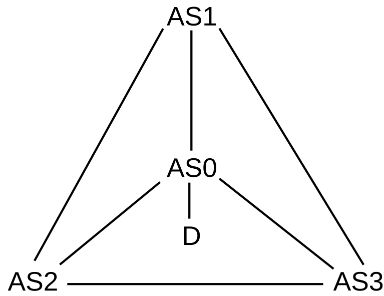
⟨AS2,AS0⟩, ⟨AS0⟩

That is, ⟨AS2,AS0⟩ is preferred to the direct path ⟨AS0⟩ (one way to express this preference might be "prefer routes for which the AS-PATH begins with AS2"; perhaps the AS1–AS0 link is more expensive). Similarly, we assume AS2 prefers paths to D in the order ⟨AS1,AS0⟩, ⟨AS0⟩. Both AS1 and AS2 start out using path ⟨AS0⟩; they advertise this to each other. As each receives the other's advertisement, they apply their preference order and therefore each switches to routing D's traffic to the other; that is, AS1 switches to the route with AS-path ⟨AS2,AS0⟩ and AS2 switches to ⟨AS1,AS0⟩. This, of course, causes a routing loop! However, as soon as they export these paths to one another, they will detect the loop in the AS-path and reject the new route, and so both will switch back to ⟨AS0⟩ as soon as they announce to each other the change in what they use.

This oscillation may continue indefinitely, as long as both AS1 and AS2 switch away from ⟨AS0⟩ at the same moment. If, however, AS1 switches to ⟨AS2,AS0⟩ while AS2 continues to use ⟨AS0⟩, then AS2 is "stuck" and the situation is stable. In practice, therefore, eventual convergence to a stable state is likely.

AS1 and AS2 might choose not to export their D-route to each other to avoid this instability. Because they do export this route to one another, they are siblings in the sense of the previous section.

Example 2: No stable state exists. This example is from *[VGE00]*. Assume that the destination D is attached to AS0, and that AS0 in turn connects to AS1, AS2 and AS3 as in the following diagram:



AS1-AS3 each have a direct route to AS0, but we assume each prefers the AS-path that takes their clockwise neighbor; that is, AS1 prefers ⟨AS3,AS0⟩ to ⟨AS0⟩; AS3 prefers ⟨AS2,AS0⟩ to ⟨AS0⟩, and AS2 prefers ⟨AS1,AS0⟩ to ⟨AS0⟩. This is a peculiar, but legal, example of input filtering.

Suppose all initially adopt AS-path ⟨AS0⟩, and advertise this, and AS1 is the first to look at the incoming advertisements. AS1 switches to the route ⟨AS3,AS0⟩, and announces this to AS2 and AS3.

At this point, AS2 sees that AS1 uses ⟨AS3,AS0⟩; if AS2 switches to AS1 then its path would be ⟨AS1,AS3,AS0⟩ rather than ⟨AS1,AS0⟩ and so it does not make the switch.

But AS3 *does* switch: it prefers ⟨AS2,AS0⟩ and this is still available. Once it makes this switch, and advertises it, AS1 sees that the route it had been using, ⟨AS3,AS0⟩, has become ⟨AS3,AS1,AS0⟩. At this point AS1 switches back to ⟨AS0⟩.

Now AS2 can switch to using ⟨AS1,AS0⟩, and does so. After that, AS3 finds it is now using ⟨AS2,AS1,AS0⟩ and it switches back to ⟨AS0⟩. This allows AS1 to switch to the longer route, and then AS2 switches back to the direct route, and then AS3 gets the longer route, then AS2 again, *etc*, forever rotating clockwise.

Because each of AS1, AS2 and AS3 export their route to D to both their neighbors, they must all be siblings of one another.

10.7 Epilog

CIDR was a deceptively simple idea. At first glance it is a straightforward extension of the subnet concept, moving the net/host division point to the left as well as to the right. But it has ushered in true hierarchical routing, most often provider-based. While CIDR was originally offered as a solution to some early crises in IPv4 address-space allocation, it has been adopted into the core of IPv6 routing as well.

Interior routing – using either distance-vector or link-state protocols – is neat and mathematical. Exterior routing with BGP is messy and arbitrary. Perhaps the most surprising thing about BGP is that the Internet works as well as it does, given the complexity of provider interconnections. The business side of routing *almost* never has an impact on ordinary users. To an extent, BGP works well because providers voluntarily limit the complexity of their filtering preferences, but that seems to be largely because the business relationships of real-world ISPs do not seem to require complex filtering.

10.8 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercise 5.5 is distinct, for example, from exercises 5.0 and 6.0. Exercises marked with a ♢ have solutions or hints at *24.9 Solutions for Large-Scale IP Routing*.

0.5.♢ Consider the following IP forwarding table that uses CIDR.

destination	next_hop
200.0.0.0/8	A
200.64.0.0/10	B
200.64.0.0/12	C
200.64.0.0/16	D

For each of the following IP addresses, indicate to what destination it is forwarded. 64 is 0x40, or 0100 0000 in binary.

(i) 200.63.1.1

(ii) 200.80.1.1

(iii) 200.72.1.1

(iv) 200.64.1.1

1.0. Consider the following IP forwarding table that uses CIDR. IP address bytes are in **hexadecimal** here, so each hex digit corresponds to four address bits. This makes prefixes such as /12 and /20 align with hex-digit boundaries. As a reminder of the hexadecimal numbering, ":" is used as the separator rather than "."

destination	next_hop
81:30:0:0/12	A
81:3c:0:0/16	B
81:3c:50:0/20	C
81:40:0:0/12	D
81:44:0:0/14	E

For each of the following IP addresses, give the next_hop for each entry in the table above that it matches. If there are multiple matches, use the longest-match rule to identify where the packet would be forwarded.

(i) 81:3b:15:49

(ii) 81:3c:56:14

(iii) 81:3c:85:2e

(iv) 81:4a:35:29

(v) 81:47:21:97

(vi) 81:43:01:c0

2.0. Consider the following IP forwarding table, using CIDR. As in exercise 1, IP address bytes are in **hexadecimal**, and ":" is used as the separator as a reminder.

destination	next_hop
00:0:0:0/2	A
40:0:0:0/2	B
80:0:0:0/2	C
C0:0:0:0/2	D

(a). To what next_hop would each of the following be routed? 63:b1:82:15, 9e:00:15:01, de:ad:be:ef

(b). Explain why every IP address is routed somewhere, even though there is no default entry. Hint: convert the first bytes to binary.

3.0. Give an IPv4 forwarding table – using CIDR – that will route all Class A addresses (first bit 0) to next_hop A, all Class B addresses (first two bits 10) to next_hop B, and all Class C addresses (first three bits 110) to next_hop C.

4.0. Suppose a router using CIDR has the following entries. Address bytes are in decimal except for the third byte, which is in **binary**.

destination	next_hop
37.149.0000 0000.0/18	A
37.149.0100 0000.0/18	A
37.149.1000 0000.0/18	A
37.149.1100 0000.0/18	B

If the next_hop for the last entry were also A, we could consolidate these four into a single entry 37.149.0.0/**16** → A. But with the final next_hop as B, how could these four be consolidated into *two* entries? You will need to assume the longest-match rule.

5.0. Suppose P, Q and R are ISPs with respective CIDR address blocks (with bytes in decimal) 51.0.0.0/8, 52.0.0.0/8 and 53.0.0.0/8. P then has customers A and B, to which it assigns address blocks as follows:

A: 51.10.0.0/16

B: 51.23.0.0/16

Q has customers C and D and assigns them address blocks as follows:

C: 52.14.0.0/16

D: 52.15.0.0/16

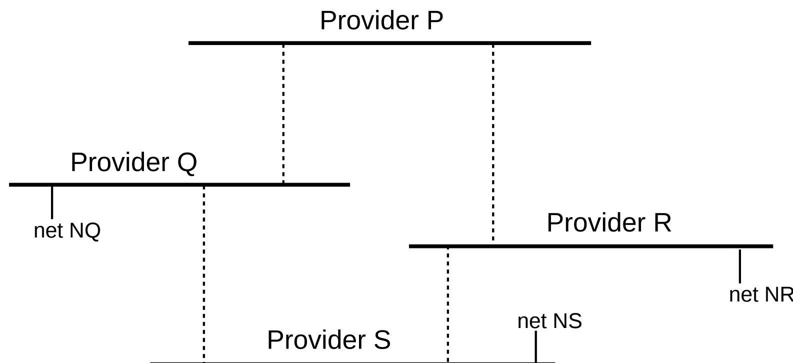
(a).♢ Give forwarding tables for P, Q and R assuming they connect to each other and to each of their own customers.

(b). Now suppose A switches from provider P to provider Q, and takes its address block with it. Give the changes to the forwarding tables for P, Q and R; the longest-match rule will be needed to resolve conflicts.

5.5 Let P, Q and R be the ISPs of exercise 5.0. This time, suppose customer C switches from provider Q to provider R. R will now have a new entry 52.14.0.0/16 → C. Give the changes to the forwarding tables of P and Q.

6.0. Suppose P, Q and R are ISPs as in exercise 5.0. This time, P and R do not connect directly; they route traffic to one another via Q. In addition, customer B is multihomed and has a secondary connection to provider R; customer D is also multihomed and has a secondary connection to provider P. R and P use these secondary connections to send to B and D respectively; however, these secondary connections are *not* advertised to other providers. Give forwarding tables for P, Q and R.

7.0. Consider the following network of providers P-S, all using BGP. The providers are the horizontal lines; each provider is its own AS.

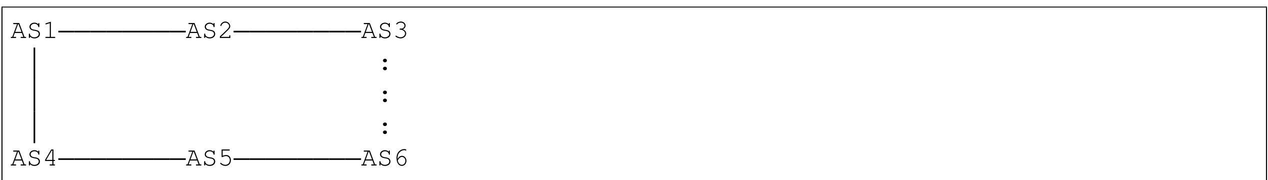


(a).◇ What routes to network NS will P receive, assuming each provider exports all its routes to its neighbors without filtering? For each route, list the AS-path.

(b). What routes to network NQ will P receive? For each route, list the AS-path.

(c). Suppose R now uses export filtering so as not to advertise any of its routes to P, though it does continue to advertise its routes to S. What routes to network NR will P receive, with AS-paths?

8.0. Consider the following network of Autonomous Systems AS1 through AS6, which double as destinations. When AS1 advertises itself to AS2, for example, the AS-path it provides is $\langle AS1 \rangle$.



(a). If neither AS3 nor AS6 exports their AS3–AS6 link to their neighbors AS2 and AS5 to the left, what routes will AS2 receive to reach AS5? Specify routes by AS-path.

(b). What routes will AS2 receive to reach AS6?

(c). Suppose AS3 exports to AS2 its link to AS6, but AS6 continues not to export the AS3–AS6 link to AS5. How will AS5 now reach AS3? How will AS2 now reach AS6? Assume that there are no local preferences in use in BGP best-path selection, and that the shortest AS-path wins.

9.0. Suppose that Internet routing in the US used geographical routing, and the first 12 bits of every IP address represent a geographical area similar in size to a telephone area code. Megacorp gets the prefix 12.34.0.0/16, based geographically in Chicago, and allocates subnets from this prefix to its offices in all 50 states. Megacorp routes all its internal traffic over its own network.

(a). Assuming all Megacorp traffic must enter and exit in Chicago, what is the route of traffic to and from the San Diego office to a client also in San Diego?

(b). Now suppose each office has its own link to a local ISP, but still uses its 12.34.0.0/16 IP addresses. Now what is the route of traffic between the San Diego office and its neighbor?

(c). Suppose Megacorp gives up and gets a separate geographical prefix for each office, *eg* 12.35.1.0/24 for San Diego and 12.37.3.0/24 for Boston. How must it configure its internal IP forwarding tables to ensure that its internal traffic is still routed entirely over its own network?

10.0. Suppose we try to use BGP's strategy of exchanging destinations plus paths as an interior routing-update strategy, perhaps replacing distance-vector routing. No costs or hop-counts are used, but routers attach to each destination a list of the routers used to reach that destination. Routers can also have route preferences, such as "prefer my link to B whenever possible".

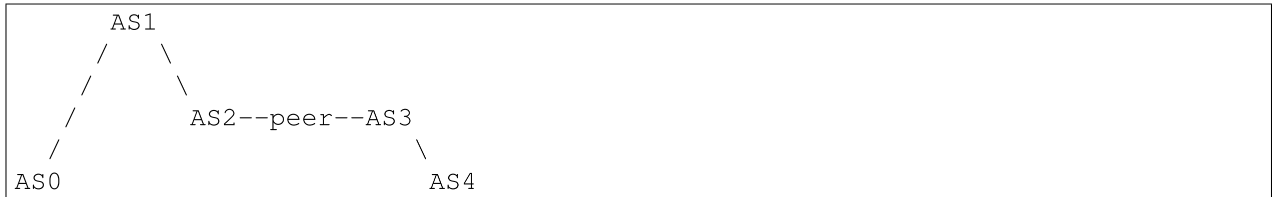
(a). Consider the network of *9.2 Distance-Vector Slow-Convergence Problem*:

The D–A link breaks, and B offers A what it thinks is its own route to D. Explain how exchanging path information prevents a routing loop here.

(b). Suppose the network is as below, and initially each router knows about itself and its immediately adjacent neighbors. What sequence of router announcements can lead to A reaching F via A→D→E→B→C→F, and what individual router preferences would be necessary? (Initially, for example, A would reach B directly; what preference might make it prefer A→D→E→B?)

(c). Explain why this method is equivalent to using the hopcount metric with either distance-vector or link-state routing, if routers are not allowed to have preferences and if the router-path length is used as a tie-breaker.

11.0. In the following AS-path from AS0 to AS4, with customers lower than providers, how far can a customer route of AS0 be exported towards AS4? How far can a customer route of AS4 be exported towards AS0?



12.0. Complete the proof of the no-valley theorem of *10.6.7 BGP Relationships* to include peer-to-peer links.

(a). Show that the existing argument also works if the A_i-to-A_{i+1} link was peer-to-peer rather than provider-to-customer, establishing that an upwards link cannot appear to the right of a peer-to-peer link.

(b). Show that the existing argument works if the A_{k-1}-to-A_k link was peer-to-peer rather than customer-to-provider, establishing that a downwards link cannot appear to the left of a peer-to-peer link.

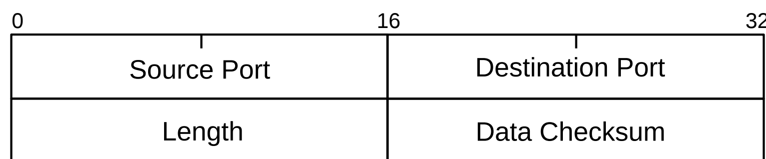
(c). Show that there cannot be two peer-to-peer links.

The standard transport protocols riding above the IP layer are **TCP** and **UDP**. As we saw in Chapter 1, UDP provides simple datagram delivery to remote sockets, that is, to ⟨host,port⟩ pairs. TCP provides a much richer functionality for sending data, but requires that the remote socket first be *connected*. In this chapter, we start with the much-simpler UDP, including the UDP-based Trivial File Transfer Protocol.

We also review some fundamental issues any transport protocol must address, such as lost final packets and packets arriving late enough to be subject to misinterpretation upon arrival. These fundamental issues will be equally applicable to TCP connections.

11.1 User Datagram Protocol – UDP

RFC 1122 refers to UDP as "almost a null protocol"; while that is something of a harsh assessment, UDP is indeed fairly basic. The two features it adds beyond the IP layer are **port numbers** and a **checksum**. The UDP header consists of the following:



The port numbers are what makes UDP into a real transport protocol: with them, an application can now connect to an individual server *process* (that is, the process "owning" the port number in question), rather than simply to a host.

UDP is **unreliable**, in that there is no UDP-layer attempt at timeouts, acknowledgment and retransmission; applications written for UDP must implement these. As with TCP, a UDP ⟨host,port⟩ pair is known as a **socket** (though UDP ports are considered a separate namespace from TCP ports). UDP is also **unconnected**, or stateless; if an application has opened a port on a host, any other host on the Internet may deliver packets to that ⟨host,port⟩ socket without preliminary negotiation.

> An old bit of Internet humor about UDP's unreliability has it that *if I send you a UDP joke, you might not get it*.

UDP packets use the 16-bit Internet **checksum** (*5.4 Error Detection*) on the data. While it is seldom done today, the checksum can be disabled and the field set to the all-0-bits value, which never occurs as an actual ones-complement sum. The UDP checksum covers the UDP header, the UDP data and also a "pseudo-IP header" that includes the source and destination IP addresses. If a NAT router rewrites an IP address or port, the UDP checksum must be updated.

UDP packets can be dropped due to queue overflows either at an intervening router or at the receiving host. When the latter happens, it means that packets are arriving faster than the receiver can process them. Higher-

level protocols that define ACK packets (*eg* UDP-based RPC, below) typically include some form of **flow control** to prevent this.

UDP is popular for "local" transport, confined to one LAN. In this setting it is common to use UDP as the transport basis for a **Remote Procedure Call**, or RPC, protocol. The conceptual idea behind RPC is that one host invokes a procedure on another host; the parameters and the return value are transported back and forth by UDP. We will consider RPC in greater detail below, in *11.5 Remote Procedure Call (RPC)*; for now, the point of UDP is that on a local LAN we can fall back on rather simple mechanisms for timeout and retransmission.

UDP is well-suited for "request-reply" semantics beyond RPC; one can use TCP to send a message and get a reply, but there is the additional overhead of setting up and tearing down a connection. DNS uses UDP, largely for this reason. However, if there is any chance that a sequence of request-reply operations will be performed in short order then TCP may be worth the overhead.

UDP is also popular for **real-time** transport; the issue here is head-of-line blocking. If a TCP packet is lost, then the receiving *host* queues any later data until the lost data is retransmitted successfully, which can take several RTTs; there is no option for the receiving *application* to request different behavior. UDP, on the other hand, gives the receiving application the freedom simply to ignore lost packets. This approach is very successful for voice and video, which are **loss-tolerant** in that small losses simply degrade the received signal slightly, but **delay-intolerant** in that packets arriving too late for playback might as well not have arrived at all. Similarly, in a computer game a lost position update is moot after any subsequent update. Loss tolerance is the reason the **Real-time Transport Protocol**, or RTP, is built on top of UDP rather than TCP. It is common for VoIP telephone calls to use RTP and UDP. See also the NoTCP Manifesto.

There is a dark side to UDP: it is sometimes the protocol of choice in flooding attacks on the Internet, as it is easy to send UDP packets with spoofed source address. See the Internet Draft draft-byrne-opsec-udp-advisory. That said, it is not especially hard to send TCP connection-request (SYN) packets with spoofed source address. It is, however, quite difficult to get TCP source-address spoofing to work for long enough that data is delivered to an application process; see *12.10.1 ISNs and spoofing*.

UDP also sometimes enables what are called **traffic amplification** attacks: the attacker sends a small message to a server, with spoofed source address, and the server then responds to the spoofed address with a much larger response message. This creates a larger volume of traffic to the victim than the attacker would be able to generate directly. One approach is for the server to limit the size of its response – ideally to the size of the client's request – until it has been able to verify that the client actually receives packets sent to its claimed IP address. QUIC uses this approach; see *12.22.4.4 Connection handshake and TLS encryption*.

11.1.1 QUIC

Sometimes UDP is used simply because it allows new or experimental protocols to run entirely as user-space applications; no kernel updates are required, as would be the case with TCP changes. Google has created a protocol named **QUIC** (Quick UDP Internet Connections, chromium.org/quic) in this category, rather specifically to support the HTTP protocol. QUIC can in fact be viewed as a transport protocol specifically tailored to HTTPS: HTTP plus TLS encryption (*22.10.2 TLS*).

QUIC also takes advantage of UDP's freedom from head-of-line blocking. For example, one of QUIC's goals includes supporting multiplexed streams in a single connection (*eg* for the multiple components of a web page). A lost packet blocks its own stream until it is retransmitted, but the other streams can continue

without waiting. An early version of QUIC supported error-correcting codes (*5.4.2 Error-Correcting Codes*); this is another feature that would be difficult to add to TCP.

In many cases QUIC eliminates the initial RTT needed for setting up a TCP connection, allowing data delivery with the very first packet. This usually this requires a recent previous connection, however, as otherwise accepting data in the first packet opens the recipient up to certain spoofing attacks. Also, QUIC usually eliminates the second (and maybe third) RTT needed for negotiating TLS encryption (*22.10.2 TLS*).

QUIC provides support for advanced congestion control, currently (2014) including a UDP analog of TCP CUBIC (*15.15 TCP CUBIC*). QUIC does this at the application layer but new congestion-control mechanisms within TCP often require client operating-system changes even when the mechanism lives primarily at the server end. (QUIC may require kernel support to make use of ECN congestion feedback, *14.8.2 Explicit Congestion Notification (ECN)*, as this requires setting bits in the IP header.) QUIC represents a promising approach to using UDP's flexibility to support innovative or experimental transport-layer features.

One downside of QUIC is its nonstandard programming interface, but note that Google can (and does) achieve widespread web utilization of QUIC simply by distributing the client side in its Chrome browser. Another downside, more insidious, is that QUIC breaks the "social contract" that everyone should use TCP so that everyone is on the same footing regarding congestion. It turns out, though, that TCP users are *not* in fact all on the same footing, as there are now multiple TCP variants (*15 Newer TCP Implementations*). Furthermore, QUIC is *supposed* to compete fairly with TCP. Still, QUIC does open an interesting can of worms.

Because many of the specific features of QUIC were chosen in response to perceived difficulties with TCP, we will explore the protocol's details after introducing TCP, in *12.22.4 QUIC Revisited*.

11.1.2 DCCP

The Datagram Congestion Control Protocol, or **DCCP**, is another transport protocol build atop UDP, preserving UDP's fundamental tolerance to packet loss. It is outlined in **RFC 4340**. DCCP adds a number of TCP-like features to UDP; for our purposes the most significant are connection setup and teardown (see *12.22.3 DCCP*) and TCP-like congestion management (see *14.6.3 DCCP Congestion Control*).

DCCP data packets, while numbered, are delivered to the application in order of arrival rather than in order of sequence number. DCCP also adds acknowledgments to UDP, but in a specialized form primarily for congestion control. There is no assumption that unacknowledged data packets will ever be retransmitted; that decision is entirely up to the application. Acknowledgments can acknowledge single packets or, through the DCCP acknowledgment-vector format, all packets received in a range of recent sequence numbers (SACK TCP, *13.6 Selective Acknowledgments (SACK)*, also supports this).

DCCP does support reliable delivery of control packets, used for connection setup, teardown and option negotiation. Option negotiation can occur at any point during a connection.

DCCP packets include not only the usual application-specific UDP port numbers, but also a 32-bit **service code**. This allows finer-grained packet handling as it unambiguously identifies the processing requested by an incoming packet. The use of service codes also resolves problems created when applications are forced to use nonstandard port numbers due to conflicts.

DCCP is specifically intended to run in in the operating-system kernel, rather than in user space. This is because the ECN congestion-feedback mechanism (*14.8.2 Explicit Congestion Notification (ECN)*) requires setting flag bits in the IP header, and most kernels do not allow user-space applications to do this.

11.1.3 UDP Simplex-Talk

One of the early standard examples for socket programming is simplex-talk. The client side reads lines of text from the user's terminal and sends them over the network to the server; the server then displays them on its terminal. The server does not acknowledge anything sent to it, or in fact send any response to the client at all. "Simplex" here refers to the one-way nature of the flow; "duplex talk" is the basis for Instant Messaging, or IM.

Even at this simple level we have some details to attend to regarding the data protocol: we assume here that the lines are sent *with* a trailing end-of-line marker. In a world where different OS's use different end-of-line marks, including them in the transmitted data can be problematic. However, when we get to the TCP version, if arriving packets are queued for any reason then the embedded end-of-line character will be the only thing to separate the arriving data into lines.

As with almost every Internet protocol, the server side must select a port number, which with the server's IP address will form the **socket address** to which clients connect. Clients must discover that port number or have it written into their application code. Clients too will *have* a port number, but it is largely invisible.

On the server side, simplex-talk must do the following:

- ask for a designated port number
- create a **socket**, the sending/receiving endpoint
- **bind** the socket to the socket address, if this is not done at the point of socket creation
- receive packets sent to the socket
- for each packet received, print its sender and its content

The client side has a similar list:

- **look up** the server's IP address, using DNS
- create an "anonymous" socket; we don't care what the client's port number is
- read a line from the terminal, and send it to the socket address ⟨server_IP,port⟩

11.1.3.1 The Server

We will start with the server side, presented here in Java. The Java socket implementation is based mostly on the BSD socket library, *1.16 Berkeley Unix*. We will use port 5432; this can easily be changed if, for example, on startup an error message like "cannot create socket with port 5432" appears. The port we use here, 5432, has also been adopted by PostgreSQL for TCP connections. (The client, of course, would also need to be changed.)

Java DatagramPacket type

Java DatagramPacket objects contain the packet data and the ⟨IP_address,port⟩ source or destination. Packets themselves combine both data and address, of course, but nonetheless combining these in a single programming-language object is not an especially common design choice. The original BSD socket library implemented data and address as separate parameters, and many other languages have followed that precedent. A case can be made that the Java approach violates the single-responsibility principle, because data and address are so often handled separately.

The socket-creation and port-binding operations are combined into the single operation new DatagramSocket(destport). Once created, this socket will receive packets from any host that addresses a packet to it; there is no need for preliminary connection. In the original BSD socket library, a socket is created with socket() and bound to an address with the separate operation bind().

The server application needs no parameters; it just starts. (That said, we could make the port number a parameter, to allow easy change.) The server accepts both IPv4 and IPv6 connections; we return to this below.

Though it plays no role in the protocol, we will also have the server time out every 15 seconds and display a message, just to show how this is done. Implementations of real UDP protocols essentially *always* must arrange when attempting to receive a packet to time out after a certain interval with no response.

The file below is at udp_stalks.java.

```
/* simplex-talk server, UDP version */

import java.net.*;
import java.io.*;

public class stalks {

    static public int destport = 5432;
    static public int bufsize = 512;
    static public final int timeout = 15000; // time in milliseconds

    static public void main(String args[]) {
        DatagramSocket s;               // UDP uses DatagramSockets

        try {
            s = new DatagramSocket(destport);
        }
        catch (SocketException se) {
            System.err.println("cannot create socket with port " + destport);
            return;
        }
        try {
            s.setSoTimeout(timeout);       // set timeout in milliseconds
        } catch (SocketException se) {
            System.err.println("socket exception: timeout not set!");
        }

        // create DatagramPacket object for receiving data:
        DatagramPacket msg = new DatagramPacket(new byte[bufsize], bufsize);
```

```
        while(true) { // read loop
            try {
                msg.setLength(bufsize);  // max received packet size
                s.receive(msg);          // the actual receive operation
                System.err.println("message from <" +
                   msg.getAddress().getHostAddress() + "," + msg.getPort() + ">");
            } catch (SocketTimeoutException ste) {    // receive() timed out
                System.err.println("Response timed out!");
                continue;
            } catch (IOException ioe) {                // should never happen!
                System.err.println("Bad receive");
                break;
            }

            String str = new String(msg.getData(), 0, msg.getLength());
            System.out.print(str);        // newline must be part of str
        }
        s.close();
    } // end of main
}
```

11.1.3.2 UDP and IP addresses

The server line `s = new DatagramSocket(destport)` creates a `DatagramSocket` object **bound** to the given port. If a host has multiple IP addresses (that is, is multihomed), packets sent to that port to any of those IP addresses will be delivered to the socket, including `localhost` (and in fact all IPv4 addresses between 127.0.0.1 and 127.255.255.255) and the subnet broadcast address (*eg* 192.168.1.255). If a client attempts to connect to the subnet broadcast address, multiple servers may receive the packet (in this we are perhaps fortunate that the stalk server does not reply).

Alternatively, we could have used

```
s = new DatagramSocket(int port, InetAddress local_addr)
```

in which case only packets sent to the host and port through the host's specific IP address `local_addr` would be delivered. It does not matter here whether IP forwarding on the host has been enabled. In the original C socket library, this binding of a port to (usually) a server socket was done with the `bind()` call. To allow connections via any of the host's IP addresses, the special IP address `INADDR_ANY` is passed to `bind()`.

When a host has multiple IP addresses, the BSD socket library and its descendents do not appear to provide a way to find out to which these an arriving UDP packet was actually sent (although it is supposed to, according to **RFC 1122**, §4.1.3.5). Normally, however, this is not a major difficulty. If a host has only one interface on an actual network (*ie* not counting loopback), and only one IP address for that interface, then any remote clients must send to that interface and address. Replies (if any, which there are not with stalk) will also come from that address.

Multiple interfaces do not necessarily create an ambiguity either; the easiest such case to experiment with involves use of the loopback and Ethernet interfaces (though one would need to use an application that, unlike stalk, sends replies). If these interfaces have respective IPv4 addresses 127.0.0.1 and 192.168.1.1,

and the client is run on the same machine, then connections to the server application sent to 127.0.0.1 will be answered from 127.0.0.1, and connections sent to 192.168.1.1 will be answered from 192.168.1.1. The IP layer sees these as different subnets, and fills in the IP source-address field according to the appropriate subnet. The same applies if multiple Ethernet interfaces are involved, or if a single Ethernet interface is assigned IP addresses for two different subnets, *eg* 192.168.**1**.1 and 192.168.**2**.1.

Life is slightly more complicated if a single interface is assigned multiple IP addresses on the *same* subnet, *eg* 192.168.1.**1** and 192.168.1.**2**. Regardless of which address a client sends its request to, the server's reply will generally always come from one designated address for that subnet, *eg* 192.168.1.1. Thus, it is possible that a *legitimate UDP reply will come from a different IP address than that to which the initial request was sent*.

If this behavior is not desired, one approach is to create multiple server sockets, and to bind each of the host's network IP addresses to a different server socket.

11.1.3.3 The Client

Next is the Java client version udp_stalkc.java. The client – any client – *must* provide the name of the host to which it wishes to send; as with the port number this can be hard-coded into the application but is more commonly specified by the user. The version here uses host `localhost` as a default but accepts any other hostname as a command-line argument. The call to `InetAddress.getByName(desthost)` invokes the DNS system, which looks up name desthost and, if successful, returns an IP address. (`InetAddress.getByName()` also accepts addresses in numeric form, *eg* "127.0.0.1", in which case DNS is not necessary.) When we create the socket we do not designate a port in the call to `new DatagramSocket()`; this means any port will do for the client. When we create the `DatagramPacket` object, the first parameter is a zero-length array as the actual data array will be provided within the loop.

A certain degree of messiness is introduced by the need to create a `BufferedReader` object to handle terminal input.

```
// simplex-talk CLIENT in java, UDP version

import java.net.*;
import java.io.*;

public class stalkc {

    static public BufferedReader bin;
    static public int destport = 5432;
    static public int bufsize = 512;

    static public void main(String args[]) {
        String desthost = "localhost";
        if (args.length >= 1) desthost = args[0];

        bin = new BufferedReader(new InputStreamReader(System.in));

        InetAddress dest;
        System.err.print("Looking up address of " + desthost + "...");
        try {
            dest = InetAddress.getByName(desthost);        // DNS query
```

```
        }
        catch (UnknownHostException uhe) {
            System.err.println("unknown host: " + desthost);
            return;
        }
        System.err.println(" got it!");

        DatagramSocket s;
        try {
            s = new DatagramSocket();
        }
        catch(IOException ioe) {
            System.err.println("socket could not be created");
            return;
        }

        System.err.println("Our own port is " + s.getLocalPort());

        DatagramPacket msg = new DatagramPacket(new byte[0], 0, dest, destport);

        while (true) {
            String buf;
            int slen;
            try {
                buf = bin.readLine();
            }
            catch (IOException ioe) {
                System.err.println("readLine() failed");
                return;
            }

            if (buf == null) break;      // user typed EOF character

            buf = buf + "\n";            // append newline character
            slen = buf.length();
            byte[] bbuf = buf.getBytes();

            msg.setData(bbuf);
            msg.setLength(slen);

            try {
                s.send(msg);
            }
            catch (IOException ioe) {
                System.err.println("send() failed");
                return;
            }
        } // while
        s.close();
    }
}
```

The default value of `desthost` here is `localhost`; this is convenient when running the client and the

server on the same machine, in separate terminal windows.

All packets are sent to the ⟨dest,destport⟩ address specified in the initialization of `msg`. Alternatively, we could have called `s.connect(dest,destport)`. This causes nothing to be sent over the network, as UDP is connectionless, but locally marks the socket `s` allowing it to send only to ⟨dest,destport⟩. In Java we still have to embed the destination address in every `DatagramPacket` we `send()`, so this offers no benefit, but in other languages this can simplify subsequent sending operations.

Like the server, the client works with both IPv4 and IPv6. The `InetAddress` object `dest` in the server code above can hold either IPv4 or IPv6 addresses; `InetAddress` is the base class with child classes `Inet4Address` and `Inet6Address`. If the client and server can communicate at all via IPv6 and if the value of `desthost` supplied to the client is an IPv6-only name, then `dest` will be an `Inet6Address` object and IPv6 will be used.

For example, if the client is invoked from the command line with `java stalkc ip6-localhost`, and the name `ip6-localhost` resolves to the IPv6 loopback address `::1`, the client will send its packets to an stalk server on the same host using IPv6 (and the loopback interface).

If greater IPv4-versus-IPv6 control is desired, one can replace the `getByName()` call with the following, where `dests` now has type `InetAddress[]`:

```
dests = InetAddress.getAllByName(desthost);
```

This returns an array of all addresses associated with the given name. One can then find the IPv6 addresses by searching this array for addresses `addr` for which `addr instanceof Inet6Address`.

For non-Java languages, IP-address objects often have an `AddressFamily` attribute that can be used to determine whether an address is IPv4 or IPv6. See also *8.11 Using IPv6 and IPv4 Together*.

Finally, here is a simple python version of the client, udp_stalkc.py.

```
#!/usr/bin/python3

from socket import *
from sys import argv

portnum = 5432

def talk():
        rhost = "localhost"
        if len(argv) > 1:
            rhost = argv[1]
        print("Looking up address of " + rhost + "...", end="")
        try:
            dest = gethostbyname(rhost)
        except (GAIerror, herror) as mesg:     # GAIerror: error in gethostbyname()
            errno,errstr=mesg.args
            print("\n   ", errstr);
            return;
        print("got it: " + dest)
        addr=(dest, portnum)                   # a socket address
        s = socket(AF_INET, SOCK_DGRAM)
        s.settimeout(1.5)              # we don't actually need to set timeout here
        while True:
```

```
            try:
                buf = input("> ")
            except:
                break
            s.sendto(bytes(buf + "\n", 'ascii'), addr)

talk()
```

Why not C?

While C is arguably the most popular language for network programming, it does not support IP addresses and other network objects as first-class types, and so we omit it here. But see *22.2.2 An Actual Stack-Overflow Example* and *22.10.3 A TLS Programming Example* for TCP-based C versions of stalk-like programs. (The problem is not entirely C's fault; a network address might be an IPv4 address or an IPv6 address (or even a "named pipe" address); these objects are of different sizes and so addresses must be handled by reference, which is awkward in C.)

To experiment with these on a single host, start the server in one window and one or more clients in other windows. One can then try the following:

- have two clients simultaneously running, and sending alternating messages to the same server
- invoke the client with the external IP address of the server in dotted-decimal, *eg* 10.0.0.3 (note that `localhost` is 127.0.0.1)
- run the java and python clients simultaneously, sending to the same server
- run the server on a different host (*eg* a virtual host or a neighboring machine)
- invoke the client with a nonexistent hostname

One can also use `netcat`, below, as a client, though `netcat` as a server will not work for the multiple-client experiments.

Note that, depending on the DNS server, the last one may not actually fail. When asked for the DNS name of a nonexistent host such as zxqzx.org, many ISPs will return the IP address of a host running a web server hosting an error/search/advertising page (usually their own). This makes some modicum of sense when attention is restricted to web searches, but is annoying if it is not, as it means non-web applications have no easy way to identify nonexistent hosts.

Simplex-talk will work if the server is on the public side of a NAT firewall. No server-side packets need to be delivered to the client! But if the other direction works, something is very wrong with the firewall.

11.1.4 netcat

The versatile `netcat` utility (also sometimes spelled `nc`) utility enables sending and receiving of individual UDP (and TCP) packets; we can use it to substitute for the stalk client, or, with a limitation, the server. (The `netcat` utility, unlike stalk, supports bidirectional communication.)

The `netcat` utility is available for Windows, linux and Macintosh systems, in both binary and source forms, from a variety of places and in something of a variety of versions. The classic version is available

from sourceforge.net/projects/nc110; a newer implementation is ncat. The Wikipedia page has additional information.

As with stalk, `netcat` sends the final end-of-line marker along with its data. The `-u` flag is used to request UDP. To send to port 5432 on `localhost` using UDP, like an stalk client, the command is

```
netcat -u localhost 5432
```

One can then type multiple lines that should all be received by a running stalk server. If desired, the source port can be specified with the `-p` option; *eg* `netcat -u -p 40001 localhost 5432`.

To act as an stalk *server*, we need the `-l` option to ask `netcat` to listen instead of sending:

```
netcat -l -u 5432
```

One can then send lines using `stalkc` or `netcat` in client mode. However, once `netcat` in server mode receives its first UDP packet, it will not accept later UDP packets from different sources (some versions of `netcat` have a `-k` option to allow this for TCP, but not for UDP). (This situation arises because `netcat` makes use of the `connect()` call on the server side as well as the client, after which the server can only send to and receive from the socket address to which it has connected. This simplifies bidirectional communication. Often, UDP `connect()` is called only by the client, if at all. See the paragraph about `connect()` following the Java `stalkc` code in *11.1.3.3 The Client*.)

11.1.5 Binary Data

In the stalk example above, the client sent strings to the server. However, what if we are implementing a protocol that requires us to send *binary* data? Or designing such a protocol? The client and server will now have to agree on how the data is to be **encoded**.

As an example, suppose the client is to send to the server a list of 32-bit integers, organized as follows. The length of the list is to occupy the first two bytes; the remainder of the packet contains the consecutive integers themselves, four bytes each, as in the diagram:

4	2003	3011	4003	10009

packet data layout

The client needs to create the byte array organized as above, and the server needs to extract the values. (The inclusion of the list length as a `short int` is not really necessary, as the receiver will be able to infer the list length from the packet size, but we want to be able to illustrate the encoding of both `int` and `short int` values.)

The protocol also needs to define how the integers themselves are laid out. There are two common ways to represent a 32-bit integer as a sequence of four bytes. Consider the integer 0x01020304 = $1\times256^3 + 2\times256^2 + 3\times256 + 4$. This can be encoded as the byte sequence [1,2,3,4], known as **big-endian** encoding, or as [4,3,2,1], known as **little-endian** encoding; the former was used by early IBM mainframes and the latter is used by most Intel processors. (We are assuming here that both architectures represent signed integers using twos-complement; this is now universal but was not always.)

To send 32-bit integers over the network, it is certainly possible to tag the data as big-endian or little-endian, or for the endpoints to negotiate the encoding. However, by far the most common approach on the Internet – at least below the application layer – is to follow the convention of **RFC 1700** and use big-endian encoding exclusively; big-endian encoding has since come to be known as "network byte order".

How one converts from "host byte order" to "network byte order" is language-dependent. It must always be done, even on big-endian architectures, as code may be recompiled on a different architecture later.

In Java the byte-order conversion is generally combined with the process of conversion from `int` to `byte[]`. The client will use a `DataOutputStream` class to support the writing of the binary values to an output stream, through methods such as `writeInt()` and `writeShort()`, together with a `ByteArrayOutputStream` class to support the conversion of the output stream to type `byte[]`. The code below assumes the list of integers is initially in an `ArrayList<Integer>` named `theNums`.

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);
try {
    dos.writeShort(theNums.size());
    for (int n : theNums) {
        dos.writeInt(n);
    }
} catch (IOException ioe) { /* exception handling */ }

byte[] bbuf = baos.toByteArray();
msg.setData(bbuf);                          // msg is the DatagramPacket object to be sent
```

The server then needs to to the reverse; again, `msg` is the arriving DatagramPacket. The code below simply calculates the sum of the 32-bit integers in `msg`:

```
ByteArrayInputStream bais = new ByteArrayInputStream(msg.getData(), 0, msg.getLength());
DataInputStream dis = new DataInputStream(bais);
int sum = 0;
try {
    int count = dis.readShort();
    for (int i=0; i<count; i++) {
        sum += dis.readInt();
    }
} catch (IOException ioe) {  /* more exception handling */  }
```

A version of simplex-talk for lists of integers can be found in client saddc.java and server sadds.java. The client reads from the command line a list of character-encoded integers (separated by whitespace), constructs the binary encoding as above, and sends them to the server; the server prints their sum. Port 5434 is used; this can be changed if necessary.

In the C language, we can simply allocate a `char[]` of the appropriate size and write the network-byte-order values directly into it. Conversion to network byte order and back is done with the following library calls:

- `htonl()`: host-to-network conversion for long (32-bit) integers
- `ntohl()`: network-to-host conversion for long integers
- `htons()`: host-to-network conversion for short (16-bit) integers
- `ntohs()`: network-to-host conversion for short integers

A certain amount of casting between `int *` and `char *` is also necessary.

In general, the designer of a protocol needs to select an unambiguous format for all binary data; protocol-defining RFCs always include such format details. This can be a particular issue for floating-point data, for which two formats can have the same endianness but still differ, *eg* in normalization or the size of the exponent field. Formats for structured data, such as arrays, must also be spelled out; in the example above the list size was indicated by a length field but other options are possible.

The example above illustrates fixed-field-width encoding. Another possible option, using variable-length encoding, is ASN.1 using the Basic Encoding Rules (*21.6 ASN.1 Syntax and SNMP*); fixed-field encoding sometimes becomes cumbersome as data becomes more hierarchical.

At the application layer, the use of non-binary encodings is common, though binary encodings continue to remain common as well. Two popular formats using human-readable unicode strings for data encoding are ASN.1 with its XML Encoding Rules and JSON. While the latter format originated with JavaScript, it is now widely supported by many other languages.

11.2 Trivial File Transport Protocol, TFTP

We now introduce a real protocol based on UDP: the Trivial File Transport Protocol, or TFTP. While TFTP supports file transfers in both directions, we will restrict attention to the more common case where the client requests a file from the server. TFTP does not support a mechanism for authentication; any requestable files are available to anyone. In this TFTP does not differ from basic web browsing; as with web servers, a TFTP file server must ensure that requests are disallowed if the file – for example `../../../etc/passwd` – is not within a permitted directory.

Because TFTP is UDP-based, and clients can be implemented very compactly, it is well-suited to the downloading of startup files to very compact systems, including diskless systems. Because it uses stop-and-wait, often uses a fixed timeout interval, and offers limited security, TFTP is typically confined to internal use within a LAN.

Although TFTP is a very simple protocol, for correct operation it must address several fundamental transport issues; these are discussed in detail in the following section. TFTP is presented here partly as a way to introduce these transport issues; we will later return to these same issues in the context of TCP (*12.11 Anomalous TCP scenarios*).

TFTP, documented first in **RFC 783** and updated in **RFC 1350**, has five packet types:

- Read ReQuest, RRQ, containing the filename and a text/binary indication
- Write ReQuest, WRQ
- Data, containing a 16-bit block number and up to 512 bytes of data
- ACK, containing a 16-bit block number
- Error, for certain designated errors. All errors other than “Unknown Transfer ID” are cause for sender termination.

Data block numbering begins at 1; we will denote the packet with the Nth block of data as Data[N]. Acknowledgments contain the block number of the block being acknowledged; thus, ACK[N] acknowledges Data[N]. All blocks of data contain 512 bytes except the final block, which is identified *as* the final block

by virtue of containing less than 512 bytes of data. If the file size was divisible by 512, the final block will contain 0 bytes of data. TFTP block numbers are 16 bits in length, and are not allowed to wrap around.

Because TFTP uses UDP (as opposed to TCP) it must take care of packetization itself, and thus must choose a block size small enough to avoid fragmentation (*7.4 Fragmentation*). While negotiation of the block size would have been possible, as is done by TCP's *12.13 Path MTU Discovery*, it would have added considerable complexity.

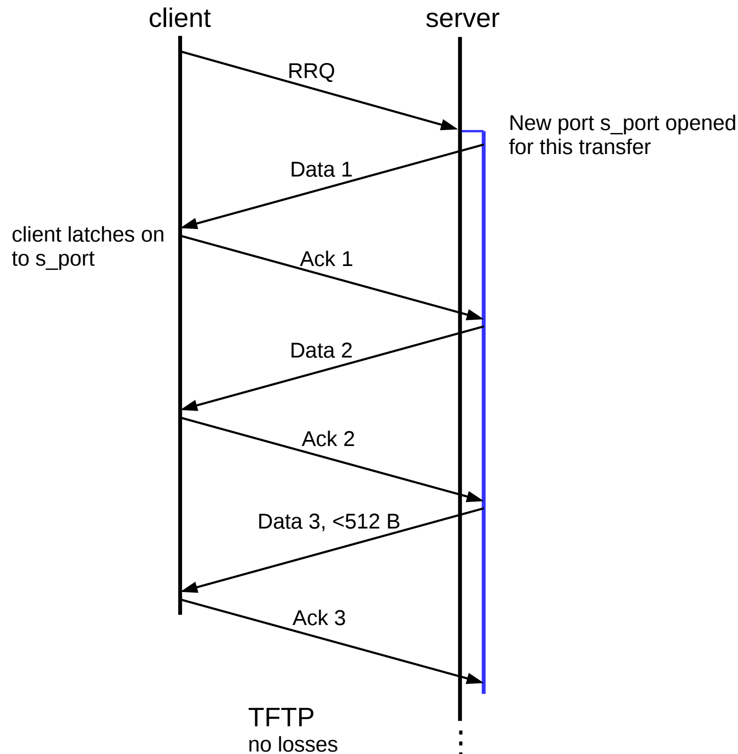
The TFTP server listens on UDP port 69 for arriving RRQ packets (and WRQ, though we will not consider those here). For each RRQ requesting a valid file, TFTP server implementations almost always create a separate process (or thread) to handle the transfer. That child process will then obtain an entirely new UDP port, which will be used for all further interaction with the client, at least for this particular transfer.

As we shall see below, this port change has significant functional implications in preventing old-duplicate packets, though for now we can justify it as making the implementer's life much easier. With the port change, the server child process responsible for the transfer has to interact with only one client; all arriving packets must have come from the client for which the child process was created (while it is possible for stray packets to arrive from other endpoints, the child process can ignore them). Without the port change, on the other hand, handling multiple concurrent transfers would be decidedly complicated: the server would have to sort out, for each arriving packet, which transfer it belonged to. Each transfer would have its own state information including block number, open file, and the time of the last successful packet. The port-change rule does have the drawback of preventing the use of TFTP through NAT firewalls.

In the absence of packet loss or other errors, TFTP file requests typically proceed as follows:

1. The client sends a RRQ to server port 69.
2. The server creates a child process, which obtains a new port, s_port, from the operating system.
3. The server child process sends Data[1] from s_port.
4. The client receives Data[1], and thus learns the value of s_port. The client will verify that each future Data[N] arrives from this same port.
5. The client sends ACK[1] (and all future ACKs) to the server's s_port.
6. The server child process sends Data[2], *etc*, each time waiting for the client ACK[N] before sending Data[N+1].
7. The transfer process stops when the server sends its final block, of size less than 512 bytes, and the client sends the corresponding ACK.

We will refer to the client's learning of s_port in step 3 as **latching on** to that port. Here is a diagram; the server child process (with new port s_port) is represented by the blue line at right.



We turn next to the complications introduced by taking packet losses and reordering into account.

11.3 Fundamental Transport Issues

The possibility of lost or delayed packets introduces several fundamental issues that any transport strategy must handle correctly for proper operation; we will revisit these in the context of TCP in *12.11 Anomalous TCP scenarios*. The issues we will consider include

- old duplicate packets
- lost final ACK
- duplicated connection request
- reboots

In this section we will discuss these issues both in general and in particular how TFTP takes them into account.

11.3.1 Old Duplicate Packets

Perhaps the trickiest issue is **old duplicate packets**: packets from the past arriving quite late, but which are mistakenly accepted as current.

For a TFTP example, suppose the client chooses port 2000 and requests file "foo", and the server then chooses port 4000 for its child process. During this transfer, Data[2] gets duplicated (perhaps through

timeout and retransmission) and one of the copies is greatly delayed. The other copy arrives on time, though, and the transfer concludes.

Now, more-or-less immediately after, the client initiates a second request, this time for file “bar”. Fatefully, the client again chooses port 2000 and the server child process again chooses port 4000.

At the point in the second transfer when the client is waiting for Data[2] from file “bar”, we will suppose the old-duplicate Data[2] from file “foo” finally shows up. There is nothing in the packet to indicate anything is amiss: the block number is correct, the destination port of 4000 ensures delivery to the current server child process, and the source port of 2000 makes the packet appear to be coming from the current client. The wrong Data[2] is therefore accepted as legitimate, and the file transfer is corrupted.

An old packet from a *previous* instance of the connection, as described above, is called an **external** old duplicate. An essential feature of the external case is that the connection is closed and then reopened a short time later, using the same port numbers at each end. As a connection is often *defined* by its endpoint port numbers (more precisely, its socket addresses), we refer to “reopening” the connection even if the second instance is completely unrelated. Two separate instances of a connection between the same socket addresses are sometimes known as **incarnations** of the connection, particularly in the context of TCP.

Old duplicates can also be **internal**, from an earlier point in the *same* connection instance. For example, if TFTP allowed its 16-bit block numbers to wrap around, then a very old Data[3] might be accepted in lieu of Data[3+2^{16}]. Internal old duplicates are usually prevented – or rendered improbable – by numbering the data, either by block or by byte, and using sufficiently many bits that wrap-around is unlikely. TFTP prevents *internal* old duplicates simply by not allowing its 16-bit block numbers to wrap around; this is effective, but limits the maximum file to 512B $\times$ (2^{16}–1), or about 32 MB. If we were not concerned with old duplicates, TFTP’s stop-and-wait could make do with 1-bit sequence numbers (*6.5 Exercises*, exercise 8.5).

Random Ports?

RFC 1350 states “The TID’s [port numbers] chosen for a connection should be randomly chosen, so that the probability that the same number is chosen twice in immediate succession is very low.” A literal interpretation is that an implementation should choose a random 16-bit number and ask for that as its TID. But the author knows of no implementation that actually does this; all seem to create sockets (*eg* with Java’s `DatagramSocket()`) and accept the port number assigned to the new socket by the operating system. That port number will not be “random” in the statistical sense, but *will* be very likely different from any recently used port. Should this be interpreted as noncompliance with the RFC? The author has no idea.

TFTP’s defense against *external* old duplicates is based on requiring that both endpoints try to choose a different port for each separate transfer (**RFC 1350** states that each side should choose its port number “randomly”). As long as *either* endpoint succeeds in choosing a new port, external old duplicates cannot interfere; see exercise 7.0. If ports are chosen at random, the probability that both sides will chose the same pair of ports for the subsequent connection is around 1/2^{32}; if ports are assigned by the operating system, there is an implicit assumption that the OS will not reissue the same port twice in rapid succession. If a noncompliant implementation on one side reuses its port numbers, TFTP transfers are protected as long as the other side chooses a new port, though the random probability of failure rises to 1/2^{16}. Note that this issue represents a second, more fundamental reason for having the server choose a new port for each transfer, unrelated to making the implementer’s life easier.

After enough time, port numbers will eventually be recycled, but we will assume old duplicates have a much smaller lifetime.

Both the external and internal old-duplicate scenarios assume that the old duplicate was sent earlier, but was somehow delayed in transit for an extended period of time, while later packets were delivered normally. Exactly how this might occur remains unclear; perhaps the least far-fetched scenario is the following:

- A first copy of the old duplicate was sent
- A routing error occurs; the packet is stuck in a routing loop
- An alternative path between the original hosts is restored, and the packet is retransmitted successfully
- Some time later, the packet stuck in the routing loop is released, and reaches its final destination

Another scenario involves a link in the path that supplies link-layer acknowledgment: the packet was sent once across the link, the link-layer ACK was lost, and so the packet was sent again. Some mechanism is still needed to delay one of the copies.

Most solutions to the old-duplicate problem assume *some* cap on just how late an old duplicate can be. In practical terms, TCP officially once took this time limit to be 60 seconds, but implementations now usually take it to be 30 seconds. Other protocols often implicitly adopt the TCP limit. Once upon a time, IP routers were expected to decrement a packet's TTL field by 1 for each second the router held the packet in its queue; in such a world, IP packets cannot be more than 255 seconds old.

It is also possible to prevent external old duplicates by including a **connection count** parameter in the transport or application header. For each consecutive connection, the connection count is incremented by (at least) 1. A separate connection-count value must be maintained by each side; if a connection-count value is ever lost, a suitable backup mechanism based on delay might be used. As an example, see *12.12 TCP Faster Opening*.

11.3.2 Lost Final ACK

In most protocols, most packets will be acknowledged. The final packet (almost always an ACK), however, cannot itself be acknowledged, as then it would not be the final packet. *Somebody* has to go last. This leaves some uncertainty on the part of the sender: did the last packet make it through, or not?

In the TFTP setting, suppose the server sends the final packet, Data[3]. The client receives it and sends ACK[3], and then exits as the transfer is done; however, the ACK[3] is lost.

The server will eventually time out and retransmit Data[3] again. However, the client is no longer there to receive the packet! The server will continue to timeout and retransmit the final Data packet until it gives up; it will never receive confirmation that the transfer succeeded.

More generally, if A sends a message to B and B replies with an acknowledgment that is delivered to A, then A and B both are both certain the message has been delivered successfully. B is *not* sure, however, that A knows this.

An alternative formulation of the lost-final-ACK problem is the **two-generals problem**. Two generals wish to agree on a time to attack, by exchanging messages. However, the generals *must* attack together, or not at all. Because some messages may be lost, neither side can ever be completely sure of the agreement. If the generals are Alice and Bob (*22.5.1 Alice and Bob*), the messages might look like this:

- Alice sends: Attack at noon
- Bob replies: Agreed (*ie* ACK)

After Bob receives Alice's message, both sides know that a noon attack has been proposed. After Bob's reply reaches Alice, both sides know that the message has been delivered. If Alice's message was an *order* to Bob, this would be sufficient.

But if Alice and Bob must cooperate, this is not quite enough: at the end of the exchange above, Bob does not know that Alice has received his reply; Bob might thus hesitate, fearing Alice might not know he's on board. Alice, aware of this possibility, might hesitate herself.

Alice might attempt to resolve this by acknowledging Bob's ACK:

- Alice replies: Ok, we're agreed on noon

But at this point Alice does not know if Bob has received this message. If Bob does not, Bob might still hesitate. Not knowing, Alice too might hesitate. There is no end. See *[AEH75]*.

Mathematically, there is no perfect solution to the two-generals problem; the generals can never be certain they are in complete agreement to attack. Suppose, to the contrary, that a sequence of messages *did* bring certainty of agreement to both Alice and Bob. Let M_1, ..., M_n be the shortest possible such sequence; without loss of generality we may assume Alice sent M_n. Now consider what happens if this final message is lost. From Alice's perspective, there is no change at all, so Alice must still be certain Bob has agreed. However, the now-shorter sequence M_1, ..., M_{n-1} cannot also bring certainty to Bob, as this sequence has length less than n, the supposed minimum here. So Bob is *not* certain, and so Alice's certainty is misplaced.

In engineering terms, however, the probability of a misunderstanding can often be made vanishingly small. Typically, if Alice does not receive Bob's reply promptly, she will resend her message at regular timeout intervals, until she does receive an ACK. If Bob can count on this behavior, he can be reasonably sure that one of his ACKs must have made it back after enough time has elapsed.

For example, if Bob knows Alice will try a total of six times if she does not receive a response, and Bob only receives Alice's first two message instances, the fact that Alice appears to have stopped repeating her transmissions is reasonable evidence that she has received Bob's response. Alternatively, if each message/ACK pair has a 10% probability of failure, and Bob knows that Alice will retry her message up to six times over the course of a day, then by the end of the day Bob can conclude that the probability that all six of his ACKs failed is at most $(0.1)^6$, or one in a million. It is not necessary in this case that Bob actually keep count of Alice's retry attempts.

For a TCP example, see *12 TCP Transport*, exercise 12.0.

TFTP addresses the lost-final-ACK problem by recommending (though not requiring) that the receiver enter into a **DALLY** state when it has sent the final ACK. In this state, the receiver responds only to duplicates of the final DATA packet; its response is to retransmit the final ACK. While one lost final ACK is possible, multiple such losses are unlikely; sooner or later the sender should receive the final ACK and will then exit.

The dally state will expire after an interval. This interval should be at least twice the sender's timeout interval, allowing for the sender to make three tries with the final data packet in all. Note that the receiver has no direct way to determine the sender's timeout value. Note also that dallying only provides increased assurance, not certainty: it is *possible* that all final ACKs were lost.

The TCP analogue of dallying is the TIMEWAIT state (*12.9 TIMEWAIT*), though TIMEWAIT also has another role related to prevention of old duplicates.

11.3.3 Duplicated Connection Request

We would also like to be able to distinguish between duplicated (*eg* retransmitted) connection requests and close but separate connection requests, especially when the second of two separate connection requests represents the cancellation of the first. Here is an outline in TFTP terms of the scenario we are trying to avoid:

- The client sends RRQ("foo")
- The client changes its mind, or aborts, or reboots, or whatever
- The client sends RRQ("bar")
- The server responds with Data[1] from the first RRQ, that is, with file "foo", while the client is expecting file "bar"

In correct TFTP operation, it is up to the client to send the second RRQ("bar") from a new port. As long as the client does that, changing its mind is not a problem. The server might end up sending Data[1] for file "foo" off into the void – that is, to the first client port – until it times out, as TFTP doesn't have a cancellation message exactly. But the request for file "bar" should succeed normally. One minor issue is that, when a TFTP application terminates, it may not have preserved anywhere a record of the port it used last, and so may be unable to guarantee that a new port is different from those used previously. But both strategies of *11.3.1 Old Duplicate Packets* – choosing a port number at random, and having the operating system assign one – are quite effective here.

TFTP does run into a somewhat unexpected issue, however, when the client sends a duplicate RRQ; typically this happens when the first RRQ times out. It is certainly possible to implement a TFTP server so as to recognize that the second RRQ is a duplicate, perhaps by noting that it is from the same client socket address and contains the same filename. In practice, however, this is incompatible with the simplified implementation approach of *11.2 Trivial File Transport Protocol, TFTP* in which the server starts a new child process for each RRQ received.

What most TFTP server implementations do in this case is to start *two* sender processes, one for each RRQ received, from two ports s_port1 and s_port2. Both will send Data[1] to the receiver. The receiver is expected to "latch on" to the port of the first Data[1] packet it receives, recording its source port. The second Data[1] will now appear to be from an incorrect port. The TFTP specification requires that a receiver reply to any packets from an unknown port by sending an ERROR packet with the code "Unknown Transfer ID" (where "Transfer ID" means "port number"); this causes the sender process that sent the later-arriving Data[1] to shut down. The sender process that sent the winning Data[1] will continue normally. Were it not for this duplicate-RRQ scenario, packets from an unknown port could probably be simply ignored.

It is theoretically possible for a malicious actor on the LAN to take advantage of this TFTP "latching on" behavior to hijack anticipated RRQs. If the actor is aware that host C is about to request a file via TFTP, it might send repeated copies of bad Data[1] to likely ports on C. When C *does* request a file, it may receive the malicious file instead of what it asked for. Because the malicious application must guess the client's port number, though, this scenario appears to be of limited importance. However, many diskless devices do load a boot file on startup via TFTP, and may do so from a predictable port number.

11.3.4 Reboots

Any ongoing communications protocol has to take into account the possibility that one side may reboot in between messages from the other side. The primary issue is detection of the reboot, so the other side can close the now-broken connection.

If the sending side of a TFTP connection reboots, packet exchange simply stops, assuming a typical receiver that does not retransmit on timeouts. If the receiving side reboots, the sender will continue to send data packets, but will receive no further acknowledgments. In most cases, the newly rebooted client will simply ignore them.

The second issue with reboots is that the rebooting system typically loses all memory of what ports it has used recently, making it difficult to ensure that it doesn't reuse recently active ports. This leads to some risk of old duplicates.

Here is a scenario, based on the one at the start of the previous section, in which a client reboot leads to receipt of the wrong file. Suppose the client sends RRQ("foo"), but then reboots before sending ACK[1]. After reboot, the client then sends RRQ("bar"), from the same port; after the reboot the client will be unable to guarantee not reopening a recently used port. The server, having received the RRQ("foo"), belatedly proceeds to send Data[1] for "foo". The client latches on to this, and accepts file "foo" while believing it is receiving file "bar".

In practical terms, this scenario seems to be of limited importance, though "diskless" devices often do use TFTP to request their boot image file when restarting, and so might be potential candidates.

11.4 Other TFTP notes

We now take a brief look at other aspects of TFTP unrelated to the fundamental transport issues above. We include a brief outline of an implementation.

11.4.1 TFTP and the Sorcerer

TFTP uses a very straightforward implementation of stop-and-wait (*6.1 Building Reliable Transport: Stop-and-Wait*). Acknowledgment packets contain the block number of the data packet being acknowledged; that is, ACK[N] acknowledges Data[N].

In the original **RFC 783** specification, TFTP was vulnerable to the Sorcerer's Apprentice bug (*6.1.2 Sorcerer's Apprentice Bug*). Correcting this problem was the justification for updating the protocol in **RFC 1350**, eleven years later. The omnibus hosts-requirements document **RFC 1123** (referenced by **RFC 1350**) describes the necessary change this way:

> Implementations MUST contain the fix for this problem: the sender (*ie*, the side originating the DATA packets) must never resend the current DATA packet on receipt of a duplicate ACK.

11.4.2 TFTP States

The TFTP specification is relatively informal; more recent protocols are often described using finite-state terminology. In each allowable state, such a specification spells out the appropriate response to all packets.

We can apply this approach to TFTP as well.

Above we defined a DALLY state, for the receiver only, with a specific response to arriving Data[N] packets. There are two other important conceptual states for TFTP receivers, which we might call UNLATCHED and ESTABLISHED.

When the receiver-client first sends RRQ, it does not know the port number from which the sender will send packets. We will call this state UNLATCHED, as the receiver has not "latched on" to the correct port. In this state, the receiver waits until it receives a packet from the sender that *looks like* a Data[1] packet; that is, it is from the sender's IP address, it has a plausible length, it is a DATA packet, and its block number is 1. When this packet is received, the receiver records s_port, and enters the ESTABLISHED state.

Once in the ESTABLISHED state, the receiver verifies for all packets that the source port number is s_port. If a packet arrives from some other port, the receiver sends back to its source an ERROR packet with "Unknown Transfer ID", but continues with the original transfer.

Here is an outline, in java, of what part of the TFTP receiver source code might look like; the code here handles the ESTABLISHED state. Somewhat atypically, the code here times out and retransmits ACK packets if no new data is received in the interval `TIMEOUT`; generally timeouts are implemented only at the TFTP sender side. Error processing is minimal, though error responses *are* sent in response to packets from the wrong port as described in the previous section. For most of the other error conditions checked for, there is no defined TFTP response.

The variables `state`, `sendtime`, `TIMEOUT`, `thePacket`, `theAddress`, `thePort`, `blocknum` and `expected_block` would need to have been previously declared and initialized; `sendtime` represents the time the most recent ACK response was sent. Several helper functions, such as `getTFTPOpcode()` and `write_the_data()`, would have to be defined. The remote port `thePort` would be initialized at the time of entry to the ESTABLISHED state; this is the port from which a packet must have been sent if it is to be considered valid. The loop here transitions to the DALLY state when a packet marking the end of the data has been received.

```
// TFTP code for ESTABLISHED state

while (state == ESTABLISHED) {
    // check elapsed time
    if (System.currentTimeMillis() > sendtime + TIMEOUT) {
        retransmit_most_recent_ACK()
        sendtime = System.currentTimeMillis()
    // receive the next packet
    try {
        s.receive(thePacket);
    }
    catch (SocketTimeoutException stoe) { continue; }   // try again
    catch (IOException ioe) { System.exit(1); }         // other errors

    if (thePacket.getAddress() != theAddress) continue;
    if (thePacket.getPort() != thePort) {
        send_error_packet(...);                 // Unknown Transfer ID; see text
        continue;
    }
    if (thePacket.getLength() < TFTP_HDR_SIZE) continue;   // TFTP_HDR_SIZE = 4
    opcode = thePacket.getData().getTFTPOpcode()
    blocknum  = thePacket.getData().getTFTPBlock()
```

```
    if (opcode != DATA) continue;
    if (blocknum != expected_block) continue;
    write_the_data(...);
    expected_block ++;
    send_ACK(...);              // and save it too for possible retransmission
    sendtime = System.currentTimeMillis();
    datasize = thePacket.getLength() - TFTP_HDR_SIZE;
    if (datasize < MAX_DATA_SIZE) state = DALLY;  // MAX_DATA_SIZE = 512
}
```

Note that the check for elapsed time is quite separate from the check for the `SocketTimeoutException`. It is possible for the receiver to receive a steady stream of "wrong" packets, so that it never encounters a `SocketTimeoutException`, and yet no "good" packet arrives and so the receiver must still arrange (as above) for a timeout and retransmission.

11.4.3 TFTP Throughput

On a single physical Ethernet, the TFTP sender and receiver would alternate using the channel, with very little "turnaround" time; the effective throughput would be close to optimal.

As soon as the store-and-forward delays of switches and routers are introduced, though, stop-and-wait becomes a performance bottleneck. Suppose that the path from sender A to receiver B passes through two switches: A—S1—S2—B, and that on all three links only the bandwidth delay is significant. Because ACK packets are so much smaller than DATA packets, we can effectively ignore the ACK travel time from B to A.

With these assumptions, the throughput is about a third of the underlying bandwidth. This is because only one of the three links can be active at any given time; the other two must be idle. We could improve throughput threefold by allowing A to send three packets at a time:

- packet 1 from A to S1
- packet 2 from A to S1 while packet 1 goes from S1 to S2
- packet 3 from A to S1 while packet 2 goes from S1 to S2 and packet 1 goes from S2 to B

This amounts to sliding windows with a winsize of three. TFTP does not support this; in the next chapter we study TCP, which does.

11.5 Remote Procedure Call (RPC)

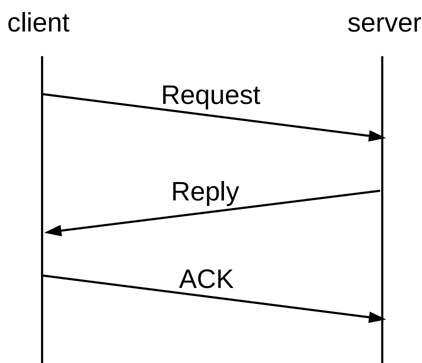
A very different communications model, usually but not always implemented over UDP, is that of **Remote Procedure Call**, or RPC. The name comes from the idea that a procedure call is being made over the network; host A packages up a *request*, with parameters, and sends it to host B, which returns a *reply*. The term **request/reply protocol** is also used for this. The side making the request is known as the *client*, and the other side the *server*.

One common example is that of DNS: a host sends a DNS lookup request to its DNS server, and receives a reply. Other examples include password verification, system information retrieval, database queries and file

I/O (below). RPC is also quite successful as the mechanism for interprocess communication within CPU clusters, perhaps its most time-sensitive application.

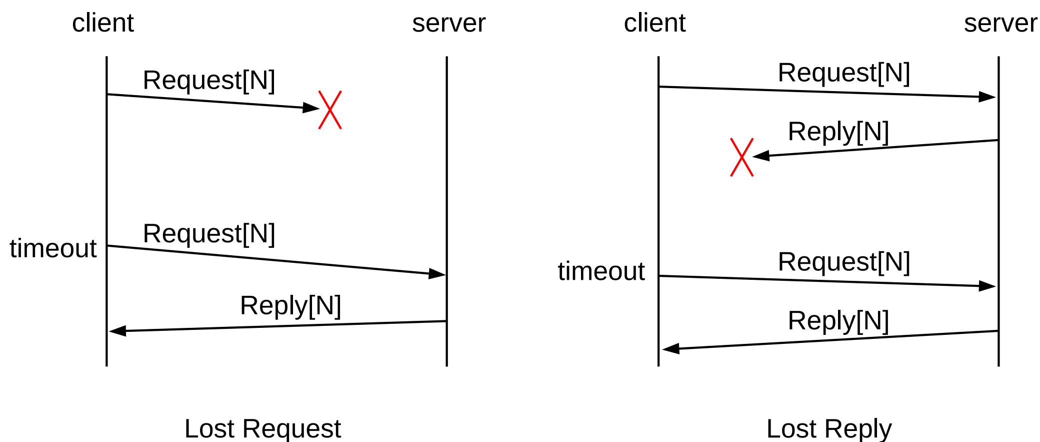
While TCP can be used for processes like these, this adds the overhead of creating and tearing down a connection; in many cases, the RPC exchange consists of nothing further beyond the request and reply and so the TCP overhead would be nontrivial. RPC over UDP is particularly well suited for transactions where both endpoints are quite likely on the same LAN, or are otherwise situated so that losses due to congestion are negligible.

The drawback to UDP is that the RPC layer must then supply its own acknowledgment protocol. This is not terribly difficult; usually the reply serves to acknowledge the request, and all that is needed is another ACK after that. If the protocol is run over a LAN, it is reasonable to use a static timeout period, perhaps somewhere in the range of 0.5 to 1.0 seconds.



Nonetheless, there are some niceties that early RPC implementations sometimes ignored, leading to a complicated history; see *11.5.2 Sun RPC* below.

It is essential that requests and replies be numbered (or otherwise identified), so that the client can determine which reply matches which request. This also means that the reply can serve to acknowledge the request; if reply[N] is not received; the requester retransmits request[N]. This can happen either if request[N] never arrived, or if it was reply[N] that got lost:



When the server creates reply[N] and sends it to the client, it must also keep a cached copy of the reply, until such time as ACK[N] is received.

After sending reply[N], the server may receive ACK[N], indicating all is well, or may receive request[N] again, indicating that reply[N] was lost, or may experience a timeout, indicating that either reply[N] or ACK[N] was lost. In the latter two cases, the server should retransmit reply[N] and wait again for ACK[N].

Finally, let us suppose that the server host delivers to its request-processing application the first copy of each request[N] to arrive, and that **neither side crashes** (or otherwise loses state in the middle of any one request/reply/ACK sequence). Let us also assume that no **packet reordering** occurs, and every request[N], reply[N] or ACK[N], retransmitted often enough, eventually makes it to its destination. We then have **exactly-once semantics**: while requests may be transmitted multiple times, they are processed (or "executed") once and only once.

11.5.1 Network File System

In terms of total packet volume, the application making the greatest use of early RPC was Sun's **Network File System**, or NFS; this allowed for a filesystem on the server to be made available to clients. When the client opened a file, the server would send back a *file handle* that typically included the file's identifying "inode" number. For `read()` operations, the request would contain the block number for the data to be read, and the corresponding reply would contain the data itself; blocks were generally 8 KB in size. For `write()` operations, the request would contain the block of data to be written together with the block number; the reply would contain an acknowledgment that it was received.

Usually an 8 KB block of data would be sent as a single UDP/IPv4 packet, using IPv4 fragmentation by the sender for transmission over Ethernet.

11.5.2 Sun RPC

The original simple model above is quite serviceable. However, in the RPC implementation developed by Sun Microsystems and documented in RFC 1831 (and now officially known as Open Network Computing, or ONC, RPC), the final acknowledgment was omitted. As there are relatively few packet losses on a LAN, this was not quite as serious as it might sound, but it did have a major consequence: the server could now not afford to cache replies, as it would never receive an indication that it was ok to delete them. Therefore, the request was re-executed upon receipt of a second request[N], as in the right-hand "lost reply" diagram above.

This was often described as **at-least-once** semantics: if a client sent a request, and eventually received a reply, the client could be sure that the request was executed at least once, but if a reply got lost then the request might be transmitted more than once. Applications, therefore, had to be aware that this was a possibility.

It turned out that for many requests, duplicate execution was not a problem. A request that has the same result (and same side effects on the server) whether executed once or executed twice is known as **idempotent**. While a request to read or write the *next* block of a file is not idempotent, a request to read or write block 37 (or any other specific block) *is* idempotent. Most data queries are also idempotent; a second query simply returns the same data as the first. Even file `open()` operations are idempotent, or at least can be implemented as such: if a file is opened the second time, the file handle is simply returned a second time.

Alas, there do exist fundamentally non-idempotent operations. File locking is one, or any form of *exclusive* file open. Creating a directory is another, because the operation must fail if the directory already exists. Even

opening a file is not idempotent if the server is expected to keep track of how many `open()` operations have been called, in order to determine if a file is still in use.

So why did Sun RPC take this route? One major advantage of at-least-once semantics is that it allowed the server to be **stateless**. The server would not need to maintain any RPC state, because without the final ACK there is no server RPC state to be maintained; for idempotent operations the server would generally not have to maintain any application state either. The practical consequence of this was that a server could crash and, because there was no state to be lost, could pick up right where it left off upon restarting.

Statelessness Inaction

Back when the Loyola CS department used Sun NFS extensively, server crashes would bring people calmly out of their offices to find out what had happened; client-workstation processes doing I/O would have locked up. Everyone would mill about in the hall until the server was rebooted, at which point they would return to their work and were almost always able to *pick up where they left off*. If the server had not been stateless, users would have been quite a bit less happy.
It is, of course, also possible to build recovery mechanisms into stateful protocols.

And for all that at-least-once semantics might today sound like an egregious shortcut, note that the exactly-once protocol outlined in the final paragraph of *11.5 Remote Procedure Call (RPC)* includes the decidedly unrealistic requirement that neither side crashes. A few approaches to the reboot problem are reviewed in *11.5.4 RPC Refinements*.

The lack of file-locking and other non-idempotent I/O operations, along with the rise of cheap client-workstation storage (and, for that matter, more-reliable servers), eventually led to the decline of NFS over RPC, though it has not disappeared. NFS can, if desired, also be run (statefully!) over TCP.

11.5.3 Serial Execution

In some RPC systems, even those with explicit ACKs, requests are executed serially by the server. Serial execution is automatic if request[N+1] serves as an implicit ACK[N]. This is a problem for file I/O operations, as physical disk drives are generally most efficient when the I/O operations can be reordered to suit the geometry of the disk. Disk drives commonly use the **elevator algorithm** to process requests: the read head moves from low-numbered tracks outwards to high-numbered tracks, pausing at each track for which there is an I/O request. Waiting for the Nth read to complete before asking the disk to start the N+1th one is slow.

The best solution here, from the server application's perspective, is to allow multiple outstanding requests and out-of-order replies. This complicates the RPC protocol, however.

11.5.4 RPC Refinements

One basic network-level improvement to RPC concerns the avoidance of IP-level fragmentation. While fragmentation is not a major performance problem on a single LAN, it may have difficulties over longer distances. One possible refinement is an RPC-level large-message protocol, that fragments at the RPC layer and which supports a mechanism for retransmission, if necessary, only of those fragments that are actually lost.

Another optimization might address the possibility that the client or the server might crash and reboot. To detect client restarts we can add to the client side a “boot counter”, incremented on each reboot and then rewritten to persistent storage. This value is then included in each request, and echoed back in each reply and ACK. This allows the server to distinguish between requests sent before and after a client reboot; such requests are conceptually unrelated and the mechanism here ensures they receive different identifiers. See *21.15.3 SNMPv3 Engines* for a related example.

On the server side, allowing for crashes and reboots is even more complicated. If the goal is simply to make the client aware that the server may have rebooted during a request/reply sequence, we might include the server’s boot counter in each reply[N]; if the client sees a change, there may be a problem with the current request. We might also include the client’s current estimate of the server’s boot counter in each request, and have the server deny requests for which there is a mismatch.

In exceptional cases, we can liken requests to database transactions and include on the server side a database-style crash-recovery journal. The goal is to allow the server, upon restarting, to identify requests that were in progress at the time of the crash, and either to roll them back or to complete them. This is not trivial, and can only be done for restricted classes of requests (*eg* reads and writes).

11.6 Epilog

UDP does not get as much attention as TCP, but between avoidance of connection-setup overhead, avoidance of head-of-line blocking and high LAN performance, it holds its own.

We also use UDP here to illustrate fundamental transport issues, both abstractly and for the specific protocol TFTP. We will revisit these fundamental issues extensively in the next chapter in the context of TCP; these issues played a major role in TCP’s design.

11.7 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercise 7.5 is distinct, for example, from exercises 7.0 and 8.0. Exercises marked with a ◊ have solutions or hints at* *24.10 Solutions for UDP*.

1.0. Perform the UDP simplex-talk experiments discussed at the end of *11.1.3 UDP Simplex-Talk*. Can multiple clients have simultaneous sessions with the same server?

2.0. Suppose that both sides of a TFTP transfer implement retransmit-on-timeout and neither side implements retransmit-on-duplicate. What would happen in each of the following cases if the first Data[3] packet is lost?

(a)◊. Sender timeout = receiver timeout = 2 seconds.

(b). Sender timeout = 1 second; receiver timeout = 3 seconds.

(c). Sender timeout = 3 seconds; receiver timeout = 1 second.

Assume the actual transfer time is negligible in comparison to the timeout intervals, and that the retransmitted Data[3] is received successfully.

3.0. In the previous exercise, how do things change if the first ACK[3] is the packet that is lost?

4.0. For each state below, spell out plausible responses for a TFTP receiver upon receipt of a Data[N] packet. Your answers may depend on N and the packet size. Indicate the events that cause a transition from one state to the next. The TFTP states were proposed in *11.4.2 TFTP States*.

(a). UNLATCHED
(b). ESTABLISHED
(c). DALLYING

Example: upon receipt of an ERROR packet, TFTP would in all three states exit.

5.0. In the TFTP-receiver code in *11.4.2 TFTP States*, explain why we must check `thePacket.getLength()` *before* extracting the opcode and block number.

6.0. Assume both the TFTP sender and the TFTP receiver implement retransmit-on-timeout but *not* retransmit-on-duplicate. Outline a specific TFTP scenario in which the TFTP receiver of *11.4.2 TFTP States* sets a socket timeout interval but never encounters a "hard" timeout – that is, a `SocketTimeoutException` – and yet must timeout and retransmit. Hint: arrange so the sender regularly times out and retransmits some packet, at an interval less than the receiver's `SocketTimeoutException` time, but it is not the packet the receiver is waiting for.

7.0. At the end of *11.3.1 Old Duplicate Packets*, we claimed that if either side in the TFTP protocol changed ports, the old-duplicate problem would not occur.

(a). If the client changes its port number on a subsequent connection, but the server does not, what prevents an old-duplicate data packet sent by the server from being accepted by the new client?
(b). If the server changes its port number on a subsequent connection, but the client does not, what prevents an old-duplicate DATA[N] packet, with N>1, sent by the server from being accepted by the new client?

7.1 In part (b) of the previous exercise, it was claimed that an old-duplicate DATA[N] could not be accepted as valid by the new client provided N>1. Give an example in which an old-duplicate DATA[1] *is* accepted as valid.

7.5. Suppose a TFTP server implementation resends DATA[N] on receipt of a duplicate ACK[N-1], contrary to *11.4.1 TFTP and the Sorcerer*. It receives a file request from a *partially implemented* TFTP client, that sends ACK[1] to the correct new port but then never increments the ACK number; the client's response to DATA[N] is always ACK[1]. What happens? (Based on a true story.)

8.0. In the simple RPC protocol at the beginning of *11.5 Remote Procedure Call (RPC)*, suppose that the server sends reply[N] and experiences a timeout, receiving nothing back from the client. In the text we suggested that most likely this meant ACK[N] was lost. Give another loss scenario, involving the loss of two packets. Assume the client and the server have the same timeout interval.

9.0. Suppose a Sun RPC `read()` request ends up executing twice. Unfortunately, in between successive `read()` operations the block of data is updated by another process, so different data is returned. Is this a failure of idempotence? Why or why not?

10.0. Outline an RPC protocol in which multiple requests can be outstanding, and replies can be sent in any order. Assume that requests are numbered, and that ACK[N] acknowledges reply[N]. Should ACKs be cumulative? If not, what should happen if an ACK is lost?

11.0. Consider the request[N]/reply[N]/ACK[N] protocol of *11.5 Remote Procedure Call (RPC)*, under the assumption that requests are numbered sequentially, but packets may potentially be delivered out of order. Thus, request[5] may arrive again after ACK[5] has been sent. and the first request[5] may even arrive after ACK[6] has been sent.

(a). If requests are handled serially, as in *11.5.3 Serial Execution*, what information does the server side need to maintain in order to avoid duplicate execution of a request?

(b). What information does the server side need to maintain if requests are handled non-serially, that is, there can be multiple outstanding requests at any one time? Assume that if request[6] arrives before request[5], the server responds with reply[6] even though there is now a temporary gap in sequence numbers.

12.0. Suppose an RPC client maintains a boot counter as in *11.5.4 RPC Refinements*. Draw diagrams for cases (a) and (b), and indicate how the boot counter is used to resolve the situation.

(a). The client sends request[N], but reboots before reply[N] is received.

(b). The client sends request[N], and then immediately reboots and sends an unrelated request that just happens also to be numbered N.

(c). What would happen in the scenario in part (b) if the reply[N] packet did *not* echo back the boot-counter value from the request[N] packet?

13.0. In this exercise we explore UDP connection state using `netcat` (*11.1.4 netcat*). Let A and B be two hosts (not necessarily distinct!).

(a). Verify that you can exchange messages between A and B after starting the following; `-u` is for UDP and `-l` is to create the server side (to "listen").

In a terminal on B: `netcat -u -l 5432`

In a terminal on A: `netcat -u B 5432`

(b). Now kill the `netcat` on A and restart it. A different local port is likely chosen by the second `netcat`; verify that communication fails.

(c). Now repeat the process, but this time in addition specify the *source* port on A with the `-p` option:

In a terminal on B: `netcat -u -l 5432`

In a terminal on A: `netcat -u -p 2345 B 5432`

Verify that killing and restarting the client on A allows communication to continue.

14.0. In this exercise we explore sending UDP packets through NAT routers (*7.7 Network Address Translation*), using `netcat` (*11.1.4 netcat*). Let A be an internal host, NR the *public* IP address of the NAT router, and C an outside host. We will initiate all connections by having A send to C at port 5432, which must not be firewalled (changing to a different port is straightforward).

(a). Verify that you can send from A to C:

In a terminal on C: `netcat -u -l 5432`

In a terminal on A: `netcat -u C 5432`

If this does not work, try changing port numbers or C's firewall settings.

(b). Try typing text into the terminal on C; `netcat` supports bidirectional communication. Does the output appear on A?

(c). Through experimentation, estimate the allowable delay between the A-to-C packets and the C-to-A response. 1 minute? 5 minutes? 10 minutes?

(d). Try to transmit the reply from C using an entirely separate pair of `netcat` sessions. For this to have any chance of working, A's source port must be known; we will set it here to 40001.

On C: as above

On A: `netcat -u -p 40001 C 5432`

As soon as data has been transmitted successfully from A to C, try the reverse path. Both A-to-C netcat processes, above, must first be terminated, to free the ports. Then:

On A: `netcat -u -l 40001`

On C: `netcat -u -p 5432 NR 40001`

The standard transport protocols riding above the IP layer are **TCP** and **UDP**. As we saw in *11 UDP Transport*, UDP provides simple datagram delivery to remote sockets, that is, to ⟨host,port⟩ pairs. TCP provides a much richer functionality for sending data to (connected) sockets.

TCP is quite different in several dimensions from UDP. TCP is **stream-oriented**, meaning that the application can write data in very small or very large amounts and the TCP layer will take care of appropriate packetization (and also that TCP transmits a stream of bytes, not messages or records; *cf 12.22.2 SCTP*). TCP is **connection-oriented**, meaning that a connection must be established before the beginning of any data transfer. TCP is **reliable**, in that TCP uses sequence numbers to ensure the correct order of delivery and a timeout/retransmission mechanism to make sure no data is lost short of massive network failure. Finally, TCP automatically uses the **sliding windows** algorithm to achieve throughput relatively close to the maximum available.

These features mean that TCP is very well suited for the transfer of large files. The two endpoints open a connection, the file data is written by one end into the connection and read by the other end, and the features above ensure that the file will be received correctly. TCP also works quite well for interactive applications where each side is sending and receiving streams of small packets. Examples of this include ssh or telnet, where packets are exchanged on each keystroke, and database connections that may carry many queries per second. TCP even works *reasonably* well for **request/reply** protocols, where one side sends a single message, the other side responds, and the connection is closed. The drawback here, however, is the overhead of setting up a new connection for each request; a better application-protocol design might be to allow multiple request/reply pairs over a single TCP connection.

Note that the connection-orientation and reliability of TCP represent abstract features built on top of the IP layer, which supports neither of them.

The connection-oriented nature of TCP warrants further explanation. With UDP, if a server opens a socket (the OS object, with corresponding socket address), then any client on the Internet can send to that socket, via its socket address. Any UDP application, therefore, must be prepared to check the source address of each packet that arrives. With TCP, all data arriving at a *connected* socket must come from the other endpoint of the connection. When a server S initially opens a socket s, that socket is "unconnected"; it is said to be in the LISTEN state. While it still has a socket address consisting of its host and port, a LISTENing socket will never receive data directly. If a client C somewhere on the Internet wishes to send data to s, it must first establish a connection, which will be defined by the **socketpair** consisting of the socket addresses (that is, the ⟨IP_addr,port⟩ pairs) at both C and S. As part of this connection process, a new *connected* child socket s_C will be created; it is s_C that will receive any data sent from C. Usually, the server will also create a new thread or process to handle communication with s_C. Typically the server will have multiple connected children of the original socket s, and, for each one, a process attached to it.

If C1 and C2 both connect to s, two connected sockets at S will be created, s_1 and s_2, and likely two separate processes. When a packet arrives at S addressed to the socket address of s, the *source* socket address will also be examined to determine whether the data is part of the C1–S or the C2–S connection, and thus whether a read on s_1 or on s_2, respectively, will see the data.

If S is acting as an ssh server, the LISTENing socket listens on port 22, and the connected child sockets correspond to the separate user login connections; the process on each child socket represents the login

process of that user, and may run for hours or days.

In Chapter 1 we likened TCP sockets to telephone connections, with the server like one high-volume phone number 800-BUY-NOWW. The unconnected socket corresponds to the number everyone dials; the connected sockets correspond to the actual calls. (This analogy breaks down, however, if one looks closely at the way such multi-operator phone lines are actually configured: each typically *does* have its own number.)

12.1 The End-to-End Principle

The End-to-End Principle is spelled out in *[SRC84]*; it states in effect that transport issues are the responsibility of the endpoints in question and thus should not be delegated to the core network. This idea has been very influential in TCP design.

Two issues falling under this category are data corruption and congestion. For the first, even though essentially all links on the Internet have link-layer checksums to protect against data corruption, TCP still adds its own checksum (in part because of a history of data errors introduced *within* routers). For the latter, TCP is today essentially the *only* layer that addresses congestion management.

Saltzer, Reed and Clark categorized functions that were subject to the End-to-End principle this way:

> The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)

This does not mean that the backbone Internet should not concern itself with congestion; it means that backbone congestion-management mechanisms should not completely replace end-to-end congestion management.

12.2 TCP Header

Below is a diagram of the TCP header. As with UDP, source and destination ports are 16 bits. The 4-bit Data Offset field specifies the number of 32-bit words in the header; if no options are present its value is 5.

As with UDP, the checksum covers the TCP header, the TCP data and an IP "pseudo header" that includes the source and destination IP addresses. The checksum must be updated by a NAT router that modifies any header values. (Although the IP and TCP layers are theoretically separate, and **RFC 793** in some places appears to suggest that TCP can be run over a non-IP internetwork layer, **RFC 793** also explicitly defines 4-byte addresses for the pseudo header. **RFC 2460** officially redefined the pseudo header to allow IPv6 addresses.)

<table>
<tr><td>0</td><td></td><td>8</td><td></td><td></td><td></td><td></td><td></td><td></td><td>16</td><td>24</td><td>32</td></tr>
<tr><td colspan="10">Source Port</td><td colspan="2">Destination Port</td></tr>
<tr><td colspan="12">Sequence Number</td></tr>
<tr><td colspan="12">Acknowledgment Number</td></tr>
<tr><td>Data Offset</td><td>Reserved</td><td>CWR</td><td>ECE</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td colspan="2">Window Size</td></tr>
<tr><td colspan="10">Checksum</td><td colspan="2">Urgent Pointer</td></tr>
<tr><td colspan="11">Options</td><td>Padding</td></tr>
</table>

The **sequence** and **acknowledgment** numbers are for numbering the data, at the byte level. This allows TCP to send 1024-byte blocks of data, incrementing the sequence number by 1024 between successive packets, or to send 1-byte telnet packets, incrementing the sequence number by 1 each time. There is no distinction between DATA and ACK packets; all packets carrying data from A to B also carry the most current acknowledgment of data sent from B to A. Many TCP applications are largely unidirectional, in which case the sender would include essentially the same acknowledgment number in each packet while the receiver would include essentially the same sequence number.

It is traditional to refer to the data portion of TCP packets as **segments**.

TCP History

The clear-cut division between the IP and TCP headers did not spring forth fully formed. See *[CK74]* for a discussion of a proto-TCP in which the sequence number (but not the acknowledgment number) appeared in the equivalent of the IP header (perhaps so it could be used for fragment reassembly).

The value of the sequence number, in *relative* terms, is the position of the first byte of the packet in the data stream, or the position of what would be the first byte in the case that no data was sent. The value of the acknowledgment number, again in relative terms, represents the byte position for the next byte expected. Thus, if a packet contains 1024 bytes of data and the first byte is number 1, then that would be the sequence number. The data bytes would be positions 1-1024, and the ACK returned would have acknowledgment number 1025.

The sequence and acknowledgment numbers, as sent, represent these relative values *plus* an **Initial Sequence Number**, or ISN, that is fixed for the lifetime of the connection. Each direction of a connection has its own ISN; see below.

TCP acknowledgments are **cumulative**: when an endpoint sends a packet with an acknowledgment number of N, it is acknowledging receipt of all data bytes numbered less than N. Standard TCP provides no mechanism for acknowledging receipt of packets 1, 2, 3 and 5; the highest cumulative acknowledgment that could be sent in that situation would be to acknowledge packet 3.

The TCP header defines some important flag bits; the brief definitions here are expanded upon in the sequel:

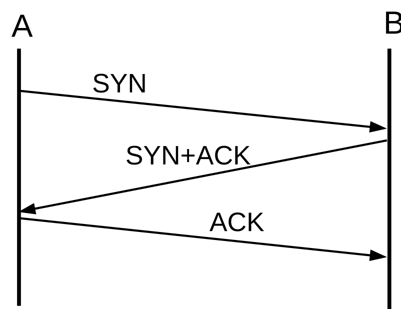
- **SYN**: for SYNchronize; marks packets that are part of the new-connection handshake

- **ACK**: indicates that the header Acknowledgment field is valid; that is, all but the first packet
- **FIN**: for FINish; marks packets involved in the connection closing
- **PSH**: for PuSH; marks "non-full" packets that should be delivered promptly at the far end
- **RST**: for ReSeT; indicates various error conditions
- **URG**: for URGent; part of a now-seldom-used mechanism for high-priority data
- **CWR** and **ECE**: part of the Explicit Congestion Notification mechanism, *14.8.2 Explicit Congestion Notification (ECN)*

12.3 TCP Connection Establishment

TCP connections are established via an exchange known as the **three-way handshake**. If A is the client and B is the LISTENing server, then the handshake proceeds as follows:

- A sends B a packet with the SYN bit set (a SYN packet)
- B responds with a SYN packet of its own; the ACK bit is now also set
- A responds to B's SYN with its own ACK



TCP three-way handshake

Normally, the three-way handshake is triggered by an application's request to connect; data can be sent only after the handshake completes. This means a one-RTT delay before any data can be sent. The original TCP standard **RFC 793** does allow data to be sent with the first SYN packet, as part of the handshake, but such data cannot be released to the remote-endpoint application until the handshake completes. Most traditional TCP programming interfaces offer no support for this early-data option.

There are recurrent calls for TCP to support data transmission within the handshake itself, so as to achieve request/reply turnaround comparable to that with RPC (*11.5 Remote Procedure Call (RPC)*). We return to this in *12.12 TCP Faster Opening*.

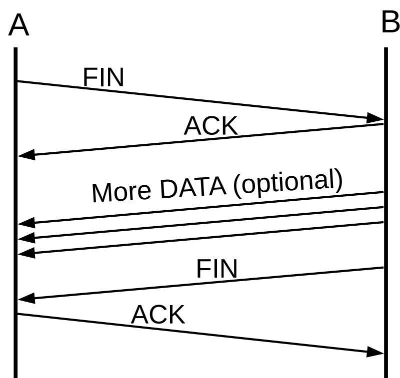
The three-way handshake is vulnerable to an attack known as **SYN flooding**. The attacker sends a large number of SYN packets to a server B. For each arriving SYN, B must allocate resources to keep track of what appears to be a legitimate connection request; with enough requests, B's resources may face exhaustion. SYN flooding is easiest if the SYN packets are simply spoofed, with forged, untraceable source-IP addresses; see spoofing at *7.1 The IPv4 Header*, and *12.10.1 ISNs and spoofing* below. SYN-flood attacks can also take the form of a large number of real connection attempts from a large number of real clients – often compromised and pressed into service by some earlier attack – but this requires considerably more resources

on the part of the attacker. See *12.22.2 SCTP* for an alternative handshake protocol (unfortunately not available to TCP) intended to mitigate SYN-flood attacks, at least from spoofed SYNs.

To **close** the connection, a superficially similar exchange involving FIN packets may occur:

- A sends B a packet with the FIN bit set (a FIN packet), announcing that it has finished sending data
- B sends A an ACK of the FIN
- B may continue to send additional data to A
- When B is also ready to cease sending, it sends its own FIN to A
- A sends B an ACK of the FIN; this is the final packet in the exchange

Here's the ladder diagram for this:



A typical TCP close

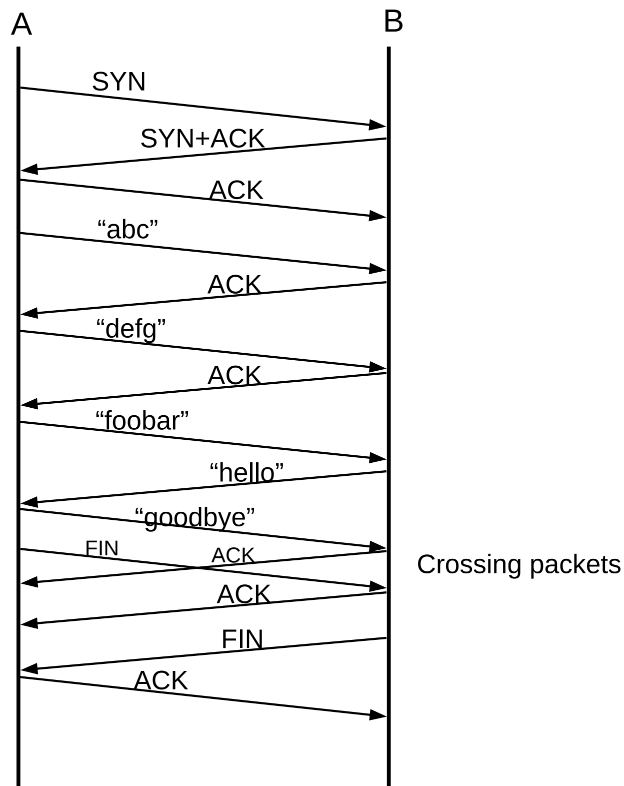
The FIN handshake is really more like two separate two-way FIN/ACK handshakes. We will return to TCP connection closing in *12.7.1 Closing a connection*.

Now let us look at the full exchange of packets in a representative connection, in which A sends strings "abc", "defg", and "foobar". B replies with "hello", and which point A sends "goodbye" and closes the connection. In the following table, *relative* sequence numbers are used, which is to say that sequence numbers begin with 0 on each side. The SEQ numbers in **bold** on the A side correspond to the ACK numbers in **bold** on the B side; they both count data flowing from A to B.

	A sends	B sends
1	SYN, **seq=0**	
2		SYN+ACK, seq=0, **ack=1** (expecting)
3	ACK, **seq=1**, ack=1 (ACK of SYN)	
4	"abc", **seq=1**, ack=1	
5		ACK, seq=1, **ack=4**
6	"defg", **seq=4**, ack=1	
7		seq=1, **ack=8**
8	"foobar", **seq=8**, ack=1	
9		seq=1, **ack=14**, "hello"
10	**seq=14**, ack=6, "goodbye"	
11,12	**seq=21**, ack=6, FIN	seq=6, **ack=21** ;; ACK of "goodbye", crossing packets
13		seq=6, **ack=22** ;; ACK of FIN
14		seq=6, **ack=22**, FIN
15	**seq=22**, ack=7 ;; ACK of FIN	

(We will see below that this table is slightly idealized, in that real sequence numbers do *not* start at 0.)

Here is the ladder diagram corresponding to this connection:



In terms of the sequence and acknowledgment numbers, SYNs count as 1 byte, as do FINs. Thus, the SYN counts as sequence number 0, and the first byte of data (the "a" of "abc") counts as sequence number 1. Similarly, the ack=21 sent by the B side is the acknowledgment of "goodbye", while the ack=22 is the acknowledgment of A's subsequent FIN.

Whenever B sends ACN=n, A follows by sending more data with SEQ=n.

TCP does *not* in fact transport relative sequence numbers, that is, sequence numbers as transmitted do not begin at 0. Instead, each side chooses its **Initial Sequence Number**, or **ISN**, and sends that in its initial SYN. The third ACK of the three-way handshake is an acknowledgment that the server side's SYN response was received correctly. All further sequence numbers sent are the ISN chosen by that side plus the relative sequence number (that is, the sequence number as if numbering did begin at 0). If A chose ISN_A=1000, we would add 1000 to all the bold entries above: A would send SYN(seq=1000), B would reply with ISN_B and ack=1001, and the last two lines would involve ack=1022 and seq=1022 respectively. Similarly, if B chose ISN_B=7000, then we would add 7000 to all the **seq** values in the "B sends" column and all the **ack** values in the "A sends" column. The table above up to the point B sends "goodbye", with actual sequence numbers instead of relative sequence numbers, is below:

	A, ISN=**1000**	B, ISN=7000
1	SYN, **seq=1000**	
2		SYN+ACK, seq=7000, **ack=1001**
3	ACK, **seq=1001**, ack=7001	
4	"abc", **seq=1001**, ack=7001	
5		ACK, seq=7001, **ack=1004**
6	"defg", **seq=1004**, ack=7001	
7		seq=7001, **ack=1008**
8	"foobar", **seq=1008**, ack=7001	
9		seq=7001, **ack=1014**, "hello"
10	**seq=1014**, ack=7006, "goodbye"	

If B had not been LISTENing at the port to which A sent its SYN, its response would have been **RST** ("reset"), meaning in this context "connection refused". Similarly, if A sent data to B before the SYN packet, the response would have been RST.

Finally, a RST can be sent by either side at any time to abort the connection. Sometimes routers along the path send "spoofed" RSTs to tear down TCP connections they are configured to regard as undesired; see *7.7.2 Middleboxes* and **RFC 3360**. Worse, sometimes external attackers are able to tear down a TCP connection with a spoofed RST; this requires brute-force guessing the endpoint port numbers and the current SYN value (**RFC 793** does not require the RST packet's ACK value to match). In the days of 4 KB window sizes, guessing a valid SYN was a one-in-a-million chance, but window sizes have steadily increased (*14.9 The High-Bandwidth TCP Problem*); a 4 MB window size makes SYN guessing quite feasible. See also **RFC 4953**, the RST-validation fix proposed in **RFC 5961** §3.2, and exercise 6.5.

If A sends a series of small packets to B, then B has the option of assembling them into a full-sized I/O buffer before releasing them to the receiving application. However, if A sets the **PSH** bit on each packet, then B should release each packet immediately to the receiving application. In Berkeley Unix and most (if not all) BSD-derived socket-library implementations, there is in fact no way to set the PSH bit; it is set automatically for each write. (But even this is not *guaranteed* as the sender may leave the bit off or consolidate several PuSHed writes into one packet; this makes using the PSH bit as a record separator difficult. In the program written to generate the WireShark packet trace, below, most of the time the strings "abc", "defg", *etc* were PuSHed separately but occasionally they were consolidated into one packet.)

As for the **URG** bit, imagine a telnet (or ssh) connection, in which A has sent a large amount of data to B, which is momentarily stalled processing it. The application at A wishes to abort that processing by sending the interrupt character CNTL-C. Under normal conditions, the application at B would have to finish

processing all the pending data before getting to the CNTL-C; however, the use of the URG bit can enable immediate asynchronous delivery of the CNTL-C. The bit is set, and the TCP header's Urgent Pointer field points to the CNTL-C in the current packet, far ahead in the normal data stream. The receiving application then skips ahead in its processing of the arriving data stream until it reaches the urgent data. For this to work, the receiving application process must have signed up to receive an asynchronous signal when urgent data arrives.

The urgent data does appear as part of the ordinary TCP data stream, and it is up to the protocol to determine the start of the data that is to be considered urgent, and what to do with the unread, buffered data sent ahead of the urgent data. For the CNTL-C example in the telnet protocol (**RFC 854**), the urgent data might consist of the telnet "Interrupt Process" byte, preceded by the "Interpret as Command" escape byte, and the earlier data is simply discarded.

Officially, the Urgent Pointer value, when the **URG** bit is set, contains the offset from the start of the current packet data to the *end* of the urgent data; it is meant to tell the receiver "you should read up to this point as soon as you can". The original intent was for the urgent pointer to mark the last byte of the urgent data, but §3.1 of **RFC 793** got this wrong and declared that it pointed to the first byte *following* the urgent data. This was corrected in **RFC 1122**, but most implementations to this day abide by the "incorrect" interpretation. **RFC 6093** discusses this and proposes, first, that the near-universal "incorrect" interpretation be accepted as standard, and, second, that developers avoid the use of the TCP urgent-data feature.

12.4 TCP and WireShark

Below is a screenshot of the WireShark program displaying a tcpdump capture intended to represent the TCP exchange above. Both hosts involved in the packet exchange were linux systems. Side A uses socket address ⟨10.0.0.3,45815⟩ and side B (the server) uses ⟨10.0.0.1,54321⟩.

WireShark is displaying *relative* TCP sequence numbers. The first three packets correspond to the three-way handshake, and packet 4 is the first data packet. Every data packet has the flags [PSH, ACK] displayed. The data in the packet can be inferred from the WireShark Len field, as each of the data strings sent has a different length.

The packets are numbered the same as in the table above up through packet 8, containing the string "foobar". At that point the table shows B replying by a combined ACK plus the string "hello"; in fact, TCP sent the ACK alone and then the string "hello"; these are WireShark packets 9 and 10 (note packet 10 has Len=5). Wireshark packet 11 is then a standalone ACK from A to B, acknowledging the "hello". WireShark packet 12 (the packet highlighted) then corresponds to table packet 10, and contains "goodbye" (Len=7); this string can be seen at the right side of the bottom pane.

The table view shows A's FIN (packet 11) crossing with B's ACK of "goodbye" (packet 12). In the WireShark view, A's FIN is packet 13, and is sent about 0.01 seconds after "goodbye"; B then ACKs them both with packet 14. That is, the table-view packet 12 does not exist in the WireShark view.

Packets 11, 13, 14 and 15 in the table and 13, 14, 15 and 16 in the WireShark screen dump correspond to the connection closing. The program that generated the exchange at B's side had to include a "sleep" delay of 40 ms between detecting the closed connection (that is, reading A's FIN) and closing its own connection (and sending its own FIN); otherwise the ACK of A's FIN traveled in the same packet with B's FIN.

The ISN for A in this example was 551144795 and B's ISN was 1366676578. The actual pcap packet-capture file is at demo_tcp_connection.pcap.

12.5 TCP Offloading

In the Wireshark example above, the hardware involved used **TCP checksum offloading**, or TCO, to have the network-interface card do the actual checksum calculations; this permits a modest amount of parallelism. As a result, the checksums for outbound packets are wrong in the capture file. WireShark has an option to disable the reporting of this.

It is also possible, with many newer network-interface cards, to offload the TCP segmentation process to the LAN hardware; this is most useful when the application is writing data continuously and is known as **TCP segmentation offloading**, or TSO. The use of TSO requires TCO, but not vice-versa.

TSO can be divided into large send offloading, LSO, for outbound traffic, and large receive offloading, LRO, for inbound. For outbound offloading, the host system transfers to the network card a large buffer of data (perhaps 64 KB), together with information about the headers. The network card then divides the buffer into 1500-byte packets, with proper TCP/IP headers, and sends them off.

For inbound offloading, the network card accumulates multiple inbound packets that are part of the same TCP connection, and consolidates them in proper sequence to one much larger packet. This means that the network card, upon receiving one packet, must wait to see if there will be more. This wait is very short, however, at most a few milliseconds. Specifically, all consolidated incoming packets must have the same TCP Timestamp value (*12.11 Anomalous TCP scenarios*).

TSO is of particular importance at very high bandwidths. At 10 Gbps, a system can send or receive close to a million packets per second, and offloading some of the packet processing to the network card can be essential to maintaining high throughput. TSO allows a host system to behave as if it were reading or writing very large packets, and yet the actual packet size on the wire remains at the standard 1500 bytes.

On linux systems, the status of TCO and TSO can be checked using the command `ethtool --show-offload` *interface*. TSO can be disabled with `ethtool --offload` *interface* `tso off`.

12.6 TCP simplex-talk

Here is a Java version of the simplex-talk server for TCP. As with the UDP version, we start by setting up the socket, here a `ServerSocket` called `ss`. This socket remains in the LISTEN state throughout. The main `while` loop then begins with the call `ss.accept()` at the start; this call blocks until an incoming connection is established, at which point it returns the connected child socket *s*. The `accept()` call models the TCP protocol behavior of waiting for three-way handshakes initiated by remote hosts and, for each, setting up a new connection.

Connections will be accepted from *all* IP addresses of the server host, *eg* the "normal" IP address, the loopback address 127.0.0.1 and, if the server is multihomed, any additional IP addresses. Unlike the UDP case (*11.1.3.2 UDP and IP addresses*), **RFC 1122** requires (§4.2.3.7) that server response packets always be sent from the same server IP address that the client first used to contact the server. (See exercise 13.0 for an example of non-compliance.)

A server application can process these connected children either serially or in parallel. The stalk version here can handle both situations, either one connection at a time (`THREADING = false`), or by creating a new thread for each connection (`THREADING = true`). Either way, the connected child socket is turned over to `line_talker()`, either as a synchronous procedure call or as a new thread. Data is then read from

the socket's associated `InputStream` using the ordinary `read()` call, versus the `receive()` used to read UDP packets. The main loop within `line_talker()` does not terminate until the client closes the connection (or there is an error).

In the serial, non-threading mode, if a second client connection is made while the first is still active, then data can be sent on the second connection but it sits in limbo until the first connection closes, at which point control returns to the `ss.accept()` call, the second connection is processed, and the second connection's data suddenly appears.

In the threading mode, the main loop spends almost all its time waiting in `ss.accept()`; when this returns a child connection we immediately spawn a new thread to handle it, allowing the parent process to go back to `ss.accept()`. This allows the program to accept multiple concurrent client connections, like the UDP version.

The code here serves as a very basic example of the creation of Java threads. The inner class `Talker` has a `run()` method, needed to implement the `runnable` interface. To start a new thread, we create a new `Talker` instance; the `start()` call then begins `Talker.run()`, which runs for as long as the client keeps the connection open. The file here is tcp_stalks.java.

```
/* THREADED simplex-talk TCP server */
/* can handle multiple CONCURRENT client connections */
/* newline is to be included at client side */

import java.net.*;
import java.io.*;

public class tstalks {

    static public int destport = 5431;
    static public int bufsize = 512;
    static public boolean THREADING = true;

    static public void main(String args[]) {
        ServerSocket ss;
        Socket s;
        try {
            ss = new ServerSocket(destport);
        } catch (IOException ioe) {
            System.err.println("can't create server socket");
            return;
        }
        System.err.println("server starting on port " + ss.getLocalPort());

        while(true) { // accept loop
            try {
                s = ss.accept();
            } catch (IOException ioe) {
                System.err.println("Can't accept");
                break;
            }

            if (THREADING) {
                Talker talk = new Talker(s);
```

```
                (new Thread(talk)).start();
            } else {
                line_talker(s);
            }
        } // accept loop
    } // end of main

    public static void line_talker(Socket s) {
       int port = s.getPort();
        InputStream istr;
        try { istr = s.getInputStream(); }
        catch (IOException ioe) {
            System.err.println("cannot get input stream");        // most likely cause: s
            return;
        }
        System.err.println("New connection from <" +
            s.getInetAddress().getHostAddress() + "," + s.getPort() + ">");
        byte[] buf = new byte[bufsize];
        int len;

        while (true) {         // while not done reading the socket
            try {
                len = istr.read(buf, 0, bufsize);
            }
            catch (SocketTimeoutException ste) {
                System.out.println("socket timeout");
                continue;
            }
            catch (IOException ioe) {
                System.err.println("bad read");
                break;        // probably a socket ABORT; treat as a close
            }
            if (len == -1) break;              // other end closed gracefully
            String str = new String(buf, 0, len);
            System.out.print("" + port + ": " + str);  // str should contain newline
        } //while reading from s


        try {istr.close();}
        catch (IOException ioe) {System.err.println("bad stream close");return;}
        try {s.close();}
        catch (IOException ioe) {System.err.println("bad socket close");return;}
        System.err.println("socket to port " + port + " closed");
    } // line_talker

    static class Talker implements Runnable {
        private Socket _s;

        public Talker (Socket s) {
            _s = s;
        }

        public void run() {
```

```
            line_talker(_s);
        } // run
    }  // class Talker
}
```

12.6.1 The TCP Client

Here is the corresponding client tcp_stalkc.java. As with the UDP version, the default host to connect to is `localhost`. We first call `InetAddress.getByName()` to perform the DNS lookup. Part of the construction of the Socket object is the connection to the desired dest and destport. Within the main `while` loop, we use an ordinary `write()` call to write strings to the socket's associated `OutputStream`.

```
// TCP simplex-talk CLIENT in java

import java.net.*;
import java.io.*;

public class stalkc {

    static public BufferedReader bin;
    static public int destport = 5431;

    static public void main(String args[]) {
        String desthost = "localhost";
        if (args.length >= 1) desthost = args[0];
        bin = new BufferedReader(new InputStreamReader(System.in));

        InetAddress dest;
        System.err.print("Looking up address of " + desthost + "...");
        try {
            dest = InetAddress.getByName(desthost);
        }
        catch (UnknownHostException uhe) {
            System.err.println("unknown host: " + desthost);
            return;
        }
        System.err.println(" got it!");

        System.err.println("connecting to port " + destport);
        Socket s;
        try {
            s = new Socket(dest, destport);
        }
        catch(IOException ioe) {
            System.err.println("cannot connect to <" + desthost + "," + destport + ">");
            return;
        }

        OutputStream sout;
        try {
            sout = s.getOutputStream();
        }
```

```
        catch (IOException ioe) {
            System.err.println("I/O failure!");
            return;
        }

        //==============================================================

        while (true) {
            String buf;
            try {
                buf = bin.readLine();
            }
            catch (IOException ioe) {
                System.err.println("readLine() failed");
                return;
            }
            if (buf == null) break;     // user typed EOF character

            buf = buf + "\n";           // protocol requires sender includes \n
            byte[] bbuf = buf.getBytes();

            try {
                sout.write(bbuf);
            }
            catch (IOException ioe) {
                System.err.println("write() failed");
                return;
            }
        } // while
    }
}
```

A Python3 version of the stalk client is available at tcp_stalkc.py.

Here are some things to try with THREADING=false in the server:

- start up two clients while the server is running. Type some message lines into both. Then exit the first client.
- start up the client before the server.
- start up the server, and then the client. Kill the server and then type some message lines to the client. What happens to the client? (It may take a couple message lines.)
- start the server, then the client. Kill the server and restart it. Now what happens to the client?

With THREADING=true, try connecting multiple clients simultaneously to the server. How does this behave differently from the first example above?

See also exercise 14.0.

12.6.2 netcat again

As with UDP (*11.1.4 netcat*), we can use the netcat utility to act as either end of the TCP simplex-talk connection. As the client we can use

```
netcat localhost 5431
```

while as the server we can use

```
netcat -l -k 5431
```

Here (but not with UDP) the -k option causes the server to accept multiple connections in sequence. The connections are handled one at a time, as is the case in the stalk server above with THREADING=false.

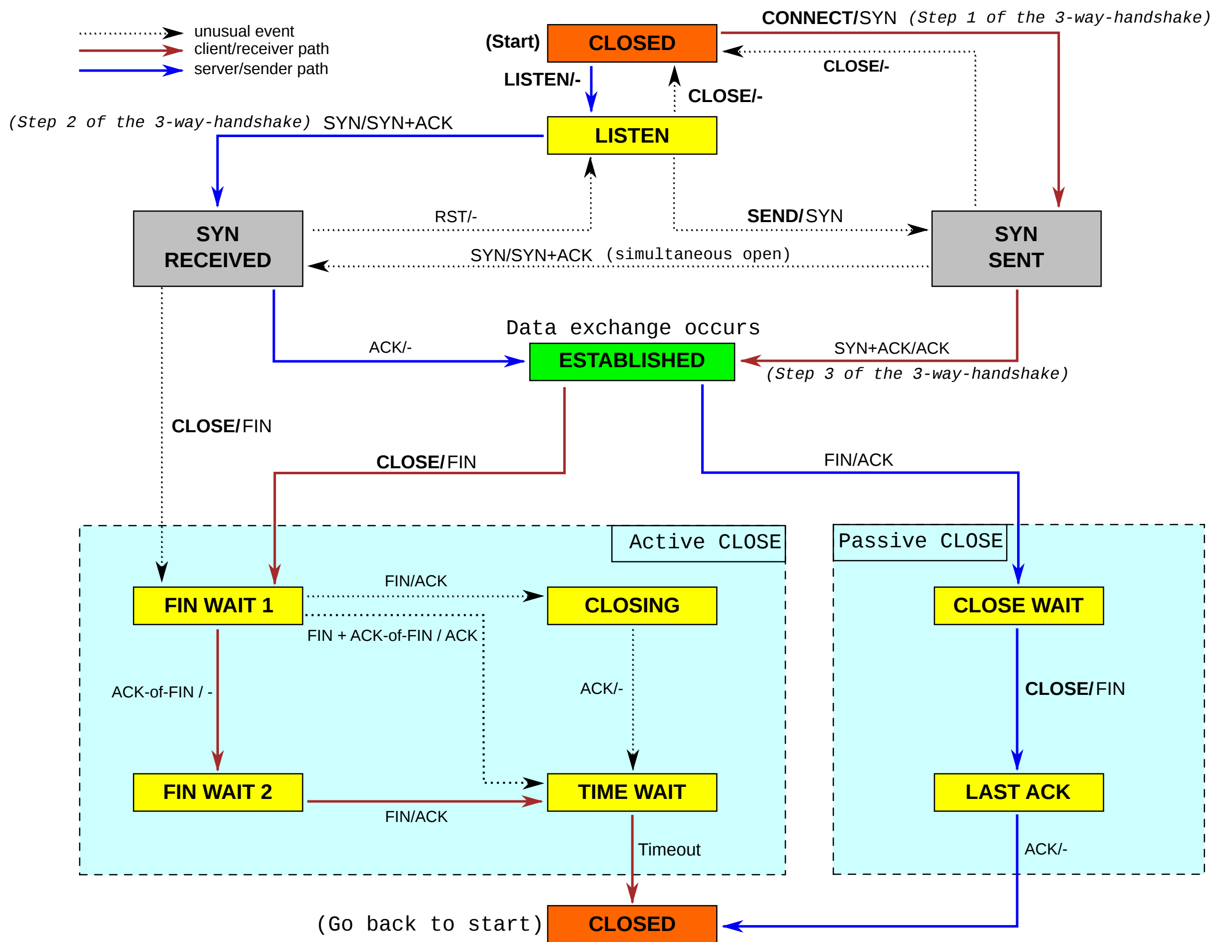
We can also use netcat to download web pages, using the HTTP protocol. The command below sends an HTTP GET request (version 1.1; **RFC 2616** and updates) to retrieve part of the website for this book; it has been broken over two lines for convenience.

```
echo -e 'GET /index.html HTTP/1.1\r\nHOST: intronetworks.cs.luc.edu\r\n'|
netcat intronetworks.cs.luc.edu 80
```

The \r\n represents the officially mandatory carriage-return/newline line-ending sequence, though \n will often work. The index.html identifies the file being requested; as index.html is the default it is often omitted, though the preceding / is still required. The webserver may support other websites as well via virtual hosting (*7.8.1 nslookup*); the HOST: specification identifies to the server the specific site we are looking for. Version 2 of HTTP is described in **RFC 7540**; its primary format is binary. (For production command-line retrieval of web pages, cURL and wget are standard choices.)

12.7 TCP state diagram

A formal definition of TCP involves the **state diagram**, with conditions for transferring from one state to another, and responses to all packets from each state. The state diagram originally appeared in **RFC 793**; the following diagram came from http://commons.wikimedia.org/wiki/File:Tcp_state_diagram_fixed.svg. The blue arrows indicate the sequence of state transitions typically followed by the server; the brown arrows represent the client. Arrows are labeled with **event / action**; that is, we move from LISTEN to SYN_RECD upon receipt of a SYN packet; the action is to respond with SYN+ACK.

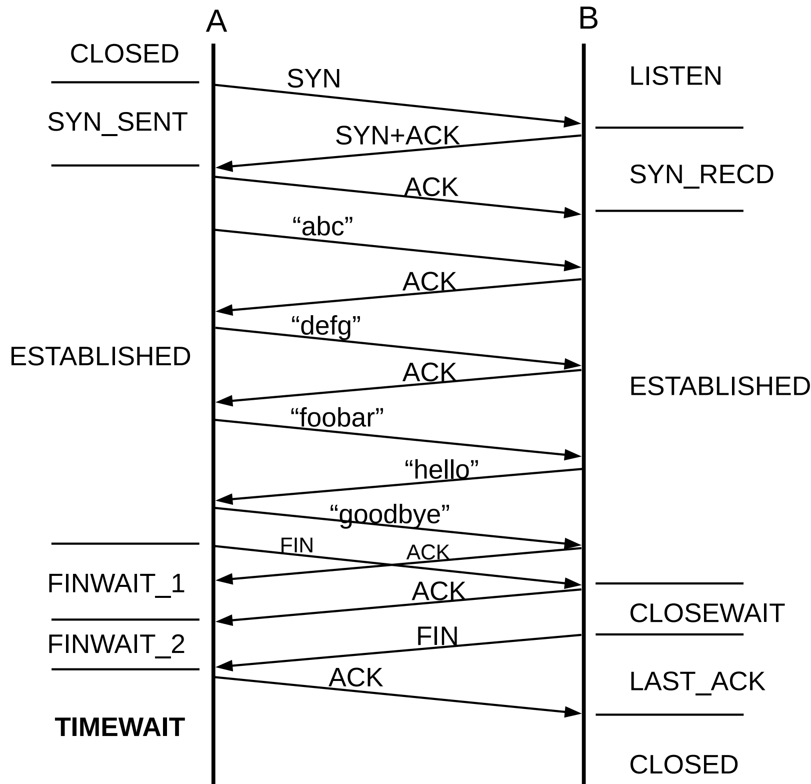


In general, this state-machine approach to protocol specification has proven very effective, and is used for most protocols. It makes it very clear to the implementer how the system should respond to each packet arrival. It is also a useful model for the implementation itself.

It is visually impractical to list every possible transition within the state diagram, full details are usually left to the accompanying text. For example, although this does not appear in the state diagram above, the per-state response rules of TCP require that in the ESTABLISHED state, if the receiver sends an ACK outside the current sliding window, then the correct response is to reply with ones own current ACK. This includes the case where the receiver *acknowledges data not yet sent*.

The ESTABLISHED state and the states below it are sometimes called the **synchronized** states, as in these states both sides have confirmed one another's ISN values.

Here is the ladder diagram for the 14-packet connection described above, this time labeled with TCP states.



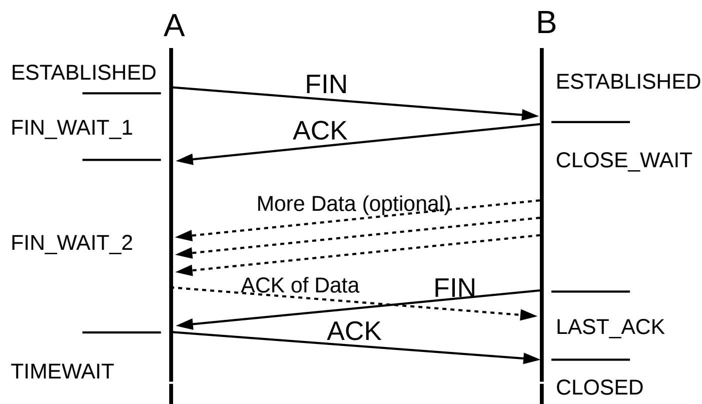
Although it essentially never occurs in practice, it is possible for each side to send the other a SYN, requesting a connection, **simultaneously** (that is, the SYNs cross on the wire). The telephony analogue occurs when each party dials the other simultaneously. On traditional land-lines, each party then gets a busy signal. On cell phones, your mileage may vary. With TCP, a single connection is created. With OSI TP4, two connections are created. The OSI approach is not possible in TCP, as a connection is determined only by the socketpair involved; if there is only one socketpair then there can be only one connection.

It is possible to view connection states under either linux or Windows with `netstat -a`. Most states are ephemeral, exceptions being LISTEN, ESTABLISHED, TIMEWAIT, and CLOSE_WAIT. One sometimes sees large numbers of connections in CLOSE_WAIT, meaning that the remote endpoint has closed the connection and sent its FIN, but the process at your end has not executed `close()` on its socket. Often this represents a programming error; alternatively, the process at the local end is still working on something. Given a local port number p in state CLOSE_WAIT on a linux system, the (privileged) command `lsof -i :`*p* will identify the process using port p.

The reader who is implementing TCP is encouraged to consult **RFC 793** and updates. For the rest of us, below are a few general observations about closing connections.

12.7.1 Closing a connection

The “normal” TCP close sequence is as follows:



Normal close

A's FIN is, in effect, a promise to B not to *send* any more. However, A must still be prepared to receive data from B, hence the optional data shown in the diagram. A good example of this occurs when A is sending a stream of data to B to be sorted; A sends FIN to indicate that it is done sending, and only then does B sort the data and begin sending it back to A. This can be generated with the command, on A, `cat thefile | ssh B sort`. That said, the presence of the optional B-to-A data above following A's FIN is relatively less common.

In the diagram above, A sends a FIN to B and receives an ACK, and then, later, B sends a FIN to A and receives an ACK. This essentially amounts to two separate two-way closure handshakes.

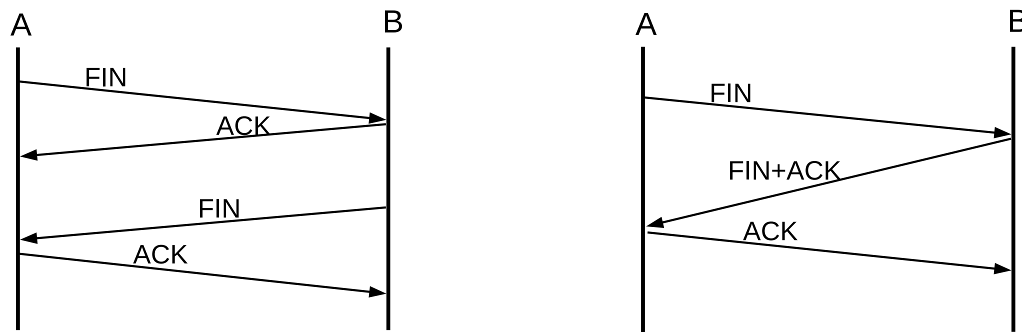
Either side may elect to close the connection, just as either party to a telephone call may elect to hang up. The first side to send a FIN – A in the diagram above – takes the **Active CLOSE** path; the other side takes the **Passive CLOSE** path. In the diagram, active-closer A moves from state ESTABLISHED to FIN_WAIT_1 to FIN_WAIT_2 (upon receipt of B's ACK of A's FIN), and then to TIMEWAIT and finally to CLOSED. Passive-closer B moves from ESTABLISHED to CLOSE_WAIT to LAST_ACK to CLOSED.

A **simultaneous close** – having both sides send each other FINs before receiving the other side's FIN – is a little more likely than a simultaneous open, earlier above, though still not very. Each side would send its FIN and move to state FIN_WAIT_1. Then, upon receiving each other's FIN packets, each side would send its final ACK and move to CLOSING. See exercises 4.0 and 4.5.

A TCP endpoint is **half-closed** if it has sent its FIN (thus promising not to send any more data) and is waiting for the other side's FIN; this corresponds to A in the diagram above in states FIN_WAIT_1 and FIN_WAIT_2. With the BSD socket library, an application can half-close its connection with the appropriate call to `shutdown()`.

Unrelatedly, A TCP endpoint is **half-open** if it is in the ESTABLISHED state, but during a lull in the exchange of packets the other side has rebooted; this has nothing to do with the close protocol. As soon as the ESTABLISHED side sends a packet, the rebooted side will respond with RST and the connection will be fully closed.

In the absence of the optional data from B to A after A sends its FIN, the closing sequence reduces to the left-hand diagram below:



Two TCP close scenarios with no B-to-A data

If B is ready to close immediately, it is possible for B's ACK and FIN to be combined, as in the right-hand diagram above, in which case the resultant diagram superficially resembles the connection-opening three-way handshake. In this case, A moves directly from FIN_WAIT_1 to TIMEWAIT, following the state-diagram link labeled "FIN + ACK-of-FIN". In theory this is rare, as the ACK of A's FIN is generated by the kernel but B's FIN cannot be sent until B's process is scheduled to run on the CPU. If the TCP layer adopts a policy of *immediately* sending ACKs upon receipt of any packet, this will never happen, as the ACK will be sent well before B's process can be scheduled to do anything. However, if B *delays* its ACKs slightly (and if it has no more data to send), then it is possible – and in fact not uncommon – for B's ACK and FIN to be sent together. Delayed ACKs, are, as we shall see below, a common strategy (*12.15 TCP Delayed ACKs*). To create the scenario of *12.4 TCP and WireShark*, it was necessary to introduce an artificial delay to prevent the simultaneous transmission of B's ACK and FIN.

12.7.2 Calling `close()`

Most network programming interfaces provide a `close()` method for ending a connection, based on the close operation for files. However, it usually closes bidirectionally and so models the TCP closure protocol rather imperfectly.

As we have seen in the previous section, the TCP close sequence is is followed more naturally if the active-closing endpoint calls `shutdown()` – promising not to send more, but allowing for continued receiving – before the final `close()`. Here is what *should* happen at the application layer if endpoint A of a TCP connection wishes to initiate the closing of its connection with endpoint B:

- A's application calls `shutdown()`, thereby promising not to send any more data. A's FIN is sent to B. A's application is expected to continue reading, however.
- The connection is now half-closed. On receipt of A's FIN, B's TCP layer knows this. If B's application attempts to read more data, it will receive an end-of-file indication (this is typically a `read()` or `recv()` operation that returns immediately with 0 bytes received).
- B's application is now done reading data, but it may or may not have more data to send. When B's application is done sending, it calls `close()`, at which point B's FIN is sent to A. Because the connection is already half-closed, B's `close()` is really a second half-close, ending further transmission by B.
- A's application keeps reading until it too receives an end-of-file indication, corresponding to B's FIN.

- The connection is now fully closed. No data has been lost.

It is sometimes the case that it is evident to A from the application protocol that B will not send more data. In such cases, A might simply call `close()` instead of `shutdown()`. This is risky, however, unless the protocol is crystal clear: if A calls `close()` and B later does send a little more data after all, or if B has already sent some data but A has not actually read it, A's TCP layer may send RST to B to indicate that not all B's data was received properly. **RFC 1122** puts it this way:

> If such a host issues a CLOSE call while received data is still pending in TCP, or if new data is received after CLOSE is called, its TCP SHOULD send a RST to show that data was lost.

If A's RST arrives at B before all of A's sent data has been processed by B's application, it is entirely possible that data sent by A will be lost, that is, will never be read by B.

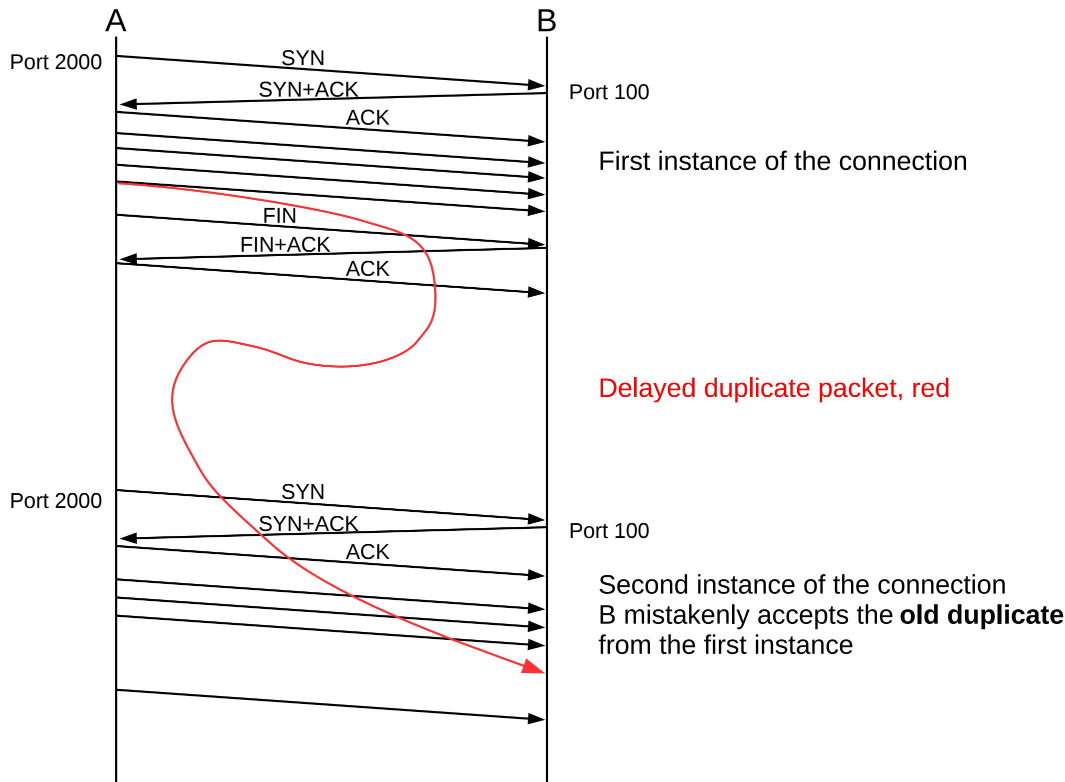
In the BSD socket library, A can set the `SO_LINGER` option, which causes A's `close()` to block until A's data has been delivered to B (or until the `SO_LINGER` timeout has expired). However, `SO_LINGER` has no bearing on the issue above; post-close data from B to A will still cause A to send a RST.

In the simplex-talk program at *12.6 TCP simplex-talk*, the client does not call `shutdown()` (it implicitly calls `close()` when it exits). When the client is done, the server calls `s.close()`. However, the fact that there is no data at all sent from the server to the client prevents the problem discussed above.

See exercises 14.0 and 15.0.

12.8 TCP Old Duplicates

Conceptually, perhaps the most serious threat facing the integrity of TCP data is external old duplicates (*11.3 Fundamental Transport Issues*), that is, very late packets from a previous instance of the connection. Suppose a TCP connection is opened between A and B. One packet from A to B is duplicated and unduly delayed, with sequence number N. The connection is closed, and then another instance is reopened, that is, a connection is created using the same ports. At some point in the second connection, when an arriving packet with seq=N would be acceptable at B, the old duplicate shows up. Later, of course, B is likely to receive a seq=N packet from the new instance of the connection, but that packet will be seen by B as a duplicate (even though the data does not match), and (we will assume) be ignored.



For TCP, it is the actual sequence numbers, rather than the relative sequence numbers, that would have to match up. The diagram above ignores that.

As with TFTP, coming up with a possible scenario accounting for the generation of such a late packet is not easy. Nonetheless, many of the design details of TCP represent attempts to minimize this risk.

Solutions to the old-duplicates problem generally involve setting an upper bound on the lifetime of any packet, the MSL, as we shall see in the next section. T/TCP (*12.12 TCP Faster Opening*) introduced a connection-count field for this.

TCP is also vulnerable to sequence-number wraparound: arrival of an old duplicates from the *same* instance of the connection. However, if we take the MSL to be 60 seconds, sequence-number wrap requires sending 2^{32} bytes in 60 seconds, which requires a data-transfer rate in excess of 500 Mbps. TCP offers a fix for this (Protection Against Wrapped Segments, or PAWS), but it was introduced relatively late; we return to this in *12.11 Anomalous TCP scenarios*.

12.9 TIMEWAIT

The TIMEWAIT state is entered by whichever side initiates the connection close; in the event of a simultaneous close, both sides enter TIMEWAIT. It is to last for a time 2×MSL, where MSL = Maximum Segment Lifetime is an agreed-upon value for the maximum lifetime on the Internet of an IP packet. Traditionally MSL was taken to be 60 seconds, but more modern implementations often assume 30 seconds (for a TIMEWAIT period of 60 seconds).

One function of TIMEWAIT is to solve the external-old-duplicates problem. TIMEWAIT requires that

between closing and reopening a connection, a long enough interval must pass that any packets from the first instance will disappear. After the expiration of the TIMEWAIT interval, an old duplicate cannot arrive.

A second function of TIMEWAIT is to address the lost-final-ACK problem (*11.3 Fundamental Transport Issues*). If host A sends its final ACK to host B and this is lost, then B will eventually retransmit *its* final packet, which will be its FIN. As long as A remains in state TIMEWAIT, it can appropriately reply to a retransmitted FIN from B with a duplicate final ACK. As with TFTP, it is possible (though unlikely) for the final ACK to be lost as well as all the retransmitted final FINs sent during the TIMEWAIT period; should this happen, one side thinks the connection closed normally while the other side thinks it did not. See exercise 12.0.

TIMEWAIT only blocks reconnections for which both sides reuse the same port they used before. If A connects to B and closes the connection, A is free to connect again to B using a different port at A's end.

Conceptually, a host may have many old connections to the same port simultaneously in TIMEWAIT; the host must thus maintain for each of its ports a list of all the remote ⟨IP_address,port⟩ sockets currently in TIMEWAIT for that port. If a host is connecting as a client, this list likely will amount to a list of recently used ports; no port is likely to have been used twice within the TIMEWAIT interval. If a host is a server, however, accepting connections on a standardized port, and happens to be the side that initiates the active close and thus later goes into TIMEWAIT, then its TIMEWAIT list for that port can grow quite long.

Generally, busy servers prefer to be free from these bookkeeping requirements of TIMEWAIT, so many protocols are designed so that it is the client that initiates the active close. In the original HTTP protocol, version 1.0, the server sent back the data stream requested by the http GET message (*12.6.2 netcat again*), and indicated the end of this stream by closing the connection. In HTTP 1.1 this was fixed so that the client initiated the close; this required a new mechanism by which the server could indicate "I am done sending this file". HTTP 1.1 also used this new mechanism to allow the server to send back multiple files over one connection.

In an environment in which many short-lived connections are made from host A to the same port on server B, port exhaustion – having all ports tied up in TIMEWAIT – is a theoretical possibility. If A makes 1000 connections per second, then after 60 seconds it has gone through 60,000 available ports, and there are essentially none left. While this rate is high, early Berkeley-Unix TCP implementations often made only about 4,000 ports available to clients; with a 120-second TIMEWAIT interval, port exhaustion would occur with only 33 connections per second.

If you use ssh to connect to a server and then issue the `netstat -a` command on your own host (or, more conveniently, `netstat -a |grep -i tcp`), you should see your connection in ESTABLISHED state. If you close your connection and check again, your connection should be in TIMEWAIT.

12.10 The Three-Way Handshake Revisited

As stated earlier in *12.3 TCP Connection Establishment*, both sides choose an ISN; actual sequence numbers are the sum of the sender's ISN and the relative sequence number. There are two original reasons for this mechanism, and one later one (*12.10.1 ISNs and spoofing*). The original TCP specification, as clarified in **RFC 1122**, called for the ISN to be determined by a special **clock**, incremented by 1 every 4 microseconds.

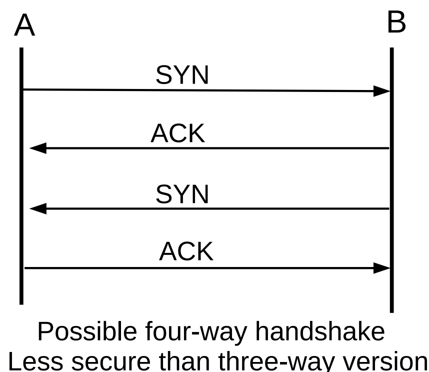
The most basic reason for using ISNs is to detect duplicate SYNs. Suppose A initiates a connection to B by sending a SYN packet. B replies with SYN+ACK, but this is lost. A then times out and retransmits its

SYN. B now receives A's second SYN while in state SYN_RECEIVED. Does this represent an entirely new request (perhaps A has suddenly restarted), or is it a duplicate? If A uses the clock-driven ISN strategy, B can tell (*almost* certainly) whether A's second SYN is new or a duplicate: only in the latter case will the ISN values in the two SYNs match.

While there is no danger to data integrity if A sends a SYN, restarts, and sends the SYN again as part of a reopening the same connection, the arrival of a second SYN with a new ISN means that the original connection cannot proceed, because that ISN is now wrong. The receiver of the duplicate SYN should drop any connection state it has recorded so far, and restart processing the second SYN from scratch.

The clock-driven ISN also originally added a second layer of protection against external old duplicates. Suppose that A opens a connection to B, and chooses a clock-based ISN N_1. A then transfers M bytes of data, closed the connection, and reopens it with ISN N_2. If $N_1 + M < N_2$, then the old-duplicates problem *cannot occur*: all of the absolute sequence numbers used in the first instance of the connection are less than or equal to $N_1 + M$, and all of the absolute sequence numbers used in the second instance will be greater than N_2. In fact, early Berkeley-Unix implementations of the socket library often allowed a second connection meeting this ISN requirement to be reopened *before* TIMEWAIT would have expired; this potentially addressed the problem of port exhaustion. Of course, if the first instance of the connection transferred data faster than the ISN clock rate, that is at more than 250,000 bytes/sec, then $N_1 + M$ would be greater than N_2, and TIMEWAIT would have to be enforced. But in the era in which TCP was first developed, sustained transfers exceeding 250,000 bytes/sec were not common.

The three-way handshake was extensively analyzed by Dalal and Sunshine in [DS78]. The authors noted that with a two-way handshake, the second side receives no confirmation that its ISN was correctly received. The authors also observed that a four-way handshake – in which the ACK of ISN_A is sent separately from ISN_B, as in the diagram below – could fail if one side restarted.



Possible four-way handshake
Less secure than three-way version

For this failure to occur, assume that after sending the SYN in line 1, with ISN_{A1}, A restarts. The ACK in line 2 is either ignored or not received. B now sends its SYN in line 3, but A interprets this as a new connection request; it will respond after line 4 by sending a fifth, SYN packet containing a different ISN_{A2}. For B the connection is now ESTABLISHED, and if B acknowledges this fifth packet but fails to update its record of A's ISN, the connection will fail as A and B would have different notions of ISN_A.

12.10.1 ISNs and spoofing

The clock-based ISN proved to have a significant weakness: it often allowed an attacker to guess the ISN a remote host might use. It did not help any that an early version of Berkeley Unix, instead of incrementing the

ISN 250,000 times a second, incremented it once a second, by 250,000 (plus something for each connection). By guessing the ISN a remote host would choose, an attacker might be able to mimic a local, trusted host, and thus gain privileged access.

Specifically, suppose host A trusts its neighbor B, and executes with privileged status commands sent by B; this situation was typical in the era of the `rhost` command. A authenticates these commands because the connection comes from B's IP address. The bad guy, M, wants to send packets to A so as to *pretend* to be B, and thus get a privileged command invoked. The connection only needs to be *started*; if the ruse is discovered after the command is executed, it is too late. M can easily send a SYN packet to A with B's IP address in the source-IP field; M can probably temporarily disable B too, so that A's SYN-ACK response, which is sent to B, goes unnoticed. What is harder is for M to figure out how to guess how to ACK ISN_A. But if A generates ISNs with a slowly incrementing clock, M can guess the pattern of the clock with previous connection attempts, and can thus guess ISN_A with a considerable degree of accuracy. So M sends SYN to A with B as source, A sends SYN-ACK to B containing ISN_A, and M *guesses* this value and sends ACK(ISN_A+1) to A, again with B listed in the IP header as source, followed by a single-packet command.

This TCP-layer IP-spoofing technique was first described by Robert T Morris in *[RTM85]*; Morris went on to launch the Internet Worm of 1988 using unrelated attacks. The IP-spoofing technique was used in the 1994 Christmas Day attack against UCSD, launched from Loyola's own apollo.it.luc.edu; the attack was associated with Kevin Mitnick though apparently not actually carried out by him. Mitnick was arrested a few months later.

RFC 1948, in May 1996, introduced a technique for introducing a degree of randomization in ISN selection, while still ensuring that the same ISN would not be used twice in a row for the same connection. The ISN is to be the sum of the 4-µs clock, C(t), and a secure hash of the connection information as follows:

ISN = C(t) + hash(local_addr, local_port, remote_addr, remote_port, key)

The `key` value is a random value chosen by the host on startup. While M, above, can poll A for its current ISN, and can probably guess the hash function and the first four parameters above, without knowing the key it cannot determine (or easily guess) the ISN value A would have sent to B. Legitimate connections between A and B, on the other hand, see the ISN increasing at the 4-µs rate.

RFC 5925 addresses spoofing and related attacks by introducing an optional TCP authentication mechanism: the TCP header includes an option containing a secure hash (*22.6 Secure Hashes*) of the rest of the TCP header and a shared secret key. The need for key management limits when this mechanism can be used; the classic use case is BGP connections between routers (*10.6 Border Gateway Protocol, BGP*).

Another approach to the prevention of spoofing attacks is to ask sites and ISPs to refuse to forward outwards any IP packets with a source address not from within that site or ISP. If an attacker's ISP implements this, the attacker will be unable to launch spoofing attacks against the outside world. A concrete proposal can be found in **RFC 2827**. Unfortunately, it has been (as of 2015) almost entirely ignored.

See also the discussion of SYN flooding at *12.3 TCP Connection Establishment*, although that attack does not involve ISN manipulation.

12.11 Anomalous TCP scenarios

TCP, like any transport protocol, must address the transport issues in *11.3 Fundamental Transport Issues*.

As we saw above, TCP addresses the Duplicate Connection Request (Duplicate SYN) issue by noting whether the ISN has changed. This is handled at the kernel level by TCP, versus TFTP's application-level (and rather desultory) approach to handing Duplicate RRQs.

TCP addresses Loss of Final ACK through TIMEWAIT: as long as the TIMEWAIT period has not expired, if the final ACK is lost and the other side resends its final FIN, TCP will still be able to reissue that final ACK. TIMEWAIT in this sense serves a similar function to TFTP's DALLY state.

External Old Duplicates, arriving as part of a previous instance of the connection, are prevented by TIMEWAIT, and may also be prevented by the use of a clock-driven ISN.

Internal Old Duplicates, from the *same* instance of the connection, that is, sequence number wraparound, is only an issue for bandwidths exceeding 500 Mbps: only at bandwidths above that can 4 GB be sent in one 60-second MSL. TCP implementations now address this with PAWS: Protection Against Wrapped Segments (**RFC 1323**). PAWS adds a 32-bit "timestamp option" to the TCP header. The granularity of the timestamp clock is left unspecified; one tick must be small enough that sequence numbers cannot wrap in that interval (*eg* less than 3 seconds for 10,000 Mbps), and large enough that the timestamps cannot wrap in time MSL. On linux systems the timestamp clock granularity is typically 1 to 10 ms; measurements on the author's systems have been 4 ms. With timestamps, an old duplicate due to sequence-number wraparound can now easily be detected.

The PAWS mechanism also requires ACK packets to echo back the sender's timestamp, in addition to including their own. This allows senders to accurately measure round-trip times.

Reboots are a potential problem as the host presumably has no record of what aborted connections need to remain in TIMEWAIT. TCP addresses this on paper by requiring hosts to implement Quiet Time on Startup: no new connections are to be accepted for 1*MSL. No known implementations actually do this; instead, they assume that the restarting process itself will take at least one MSL. This is no longer as certain as it once was, but serious consequences have not ensued.

12.12 TCP Faster Opening

If a client wants to connect to a server, send a request and receive an immediate reply, TCP mandates one full RTT for the three-way handshake before data can be delivered. This makes TCP one RTT slower than UDP-based request-reply protocols. There have been periodic calls to allow TCP clients to include data with the first SYN packet and have it be delivered immediately upon arrival – this is known as **accelerated open**.

If there will be a series of requests and replies, the simplest fix is to **pipeline** all the requests and replies over one persistent connection; the one-RTT delay then applies only to the first request. If the pipeline connection is idle for a long-enough interval, it may be closed, and then reopened later if necessary.

An early accelerated-open proposal was **T/TCP**, or TCP for Transactions, specified in **RFC 1644**. T/TCP introduced a **connection count** TCP option, called CC; each participant would include a 32-bit CC value in its SYN; each participant's own CC values were to be monotonically increasing. Accelerated open was allowed if the server side had the client's previous CC in a cache, and the new CC value was strictly greater than this cached value. This ensured that the new SYN was not a duplicate of an older SYN.

Unfortunately, this also bypasses the modest authentication of the client's IP address provided by the full three-way handshake, worsening the spoofing problem of *12.10.1 ISNs and spoofing*. If malicious host M wants to pretend to be B when sending a privileged request to A, all M has to do is send a single SYN+Data

packet with an extremely large value for CC. Generally, the accelerated open succeeded as long as the CC value presented was larger that the value A had cached for B; it did not have to be larger by exactly 1.

The recent **TCP Fast Open** proposal, described in **RFC 7413**, involves a secure "cookie" sent by the client as a TCP option; if a SYN+Data packet has a valid cookie, then the client has proven its identity and the data may be released immediately to the receiving application. Cookies are cryptographically secure, and are requested ahead of time from the server.

Because cookies have an expiration date and must be requested ahead of time, TCP Fast Open is not fundamentally faster from the connection-pipeline option, except that holding a TCP connection open uses more resources than simply storing a cookie. The likely application for TCP Fast Open is in accessing web servers. Web clients and servers already keep a persistent connection open for a while, but often "a while" here amounts only to several seconds; TCP Fast Open cookies could remain active for much longer.

One serious practical problem with TCP Fast Open is that some middleboxes (*7.7.2 Middleboxes*) remove TCP options they do not understand, or even block the connection attempt entirely.

12.13 Path MTU Discovery

TCP connections are more efficient if they can keep large packets flowing between the endpoints. Once upon a time, TCP endpoints included just 512 bytes of data in each packet that was not destined for local delivery, to avoid fragmentation. TCP endpoints now typically engage in **Path MTU Discovery** which almost always allows them to send larger packets; backbone ISPs are now usually able to carry 1500-byte packets. The **Path MTU** is the largest packet size that can be sent along a path without fragmentation.

The IPv4 strategy is to send an initial data packet with the IPv4 `DONT_FRAG` bit set. If the ICMP message `Frag_Required/DONT_FRAG_Set` comes back, or if the packet times out, the sender tries a smaller size. If the sender receives a TCP ACK for the packet, on the other hand, indicating that it made it through to the other end, it might try a larger size. Usually, the size range of 512-1500 bytes is covered by less than a dozen discrete values; the point is not to find the exact Path MTU but to determine a reasonable approximation rapidly.

IPv6 has no DONT_FRAG bit. Path MTU Discovery over IPv6 involves the periodic sending of larger packets; if the ICMPv6 message `Packet Too Big` is received, a smaller packet size must be used. **RFC 1981** has details.

12.14 TCP Sliding Windows

TCP implements sliding windows, in order to improve throughput. Window sizes are measured in terms of bytes rather than packets; this leaves TCP free to packetize the data in whatever segment size it elects. In the initial three-way handshake, each side specifies the maximum window size it is willing to accept, in the **Window Size** field of the TCP header. This 16-bit field can only go to 64 KB, and a 1 Gbps × 100 ms bandwidth×delay product is 12 MB; as a result, there is a TCP **Window Scale** option that can also be negotiated in the opening handshake. The scale option specifies a power of 2 that is to be multiplied by the actual Window Size value. In the WireShark example above, the client specified a Window Size field of 5888 (= 4 × 1472) in the third packet, but with a Window Scale value of 2^6 = 64 in the first packet, for an

effective window size of 64 × 5888 = 256 segments of 1472 bytes. The server side specified a window size of 5792 and a scaling factor of 2^5 = 32.

TCP may either transmit a bulk stream of data, using sliding windows fully, or it may send slowly generated interactive data; in the latter case, TCP may never have even one full segment outstanding.

In the following chapter we will see that a sender frequently reduces the actual TCP window size, in order to avoid congestion; the window size included in the TCP header is known as the **Advertised** Window Size. On startup, TCP does not send a full window all at once; it uses a mechanism called "slow start".

12.15 TCP Delayed ACKs

TCP receivers are allowed briefly to delay their ACK responses to new data. This offers perhaps the most benefit for interactive applications that exchange small packets, such as ssh and telnet. If A sends a data packet to B and expects an immediate response, delaying B's ACK allows the receiving *application* on B time to wake up and generate that application-level response, which can then be sent together with B's ACK. Without delayed ACKs, the kernel layer on B may send its ACK before the receiving application on B has even been scheduled to run. If response packets are small, that doubles the total traffic. The maximum ACK delay is 500 ms, according to **RFC 1122** and **RFC 2581**.

For bulk traffic, delayed ACKs simply mean that the ACK traffic volume is reduced. Because ACKs are cumulative, one ACK from the receiver can in principle acknowledge multiple data packets from the sender. Unfortunately, acknowledging too many data packets with one ACK can interfere with the self-clocking aspect of sliding windows; the arrival of that ACK will then trigger a burst of additional data packets, which would otherwise have been transmitted at regular intervals. Because of this, the RFCs above specify that an ACK be sent, at a minimum, for every other data packet. For a discussion of how the sender should respond to delayed ACKs, see *13.2.1 Per-ACK Responses*.

Bandwidth Conservation

Delayed ACKs and the Nagle algorithm both originated in a bygone era, when bandwidth was in much shorter supply than it is today. In **RFC 896**, John Nagle writes (in 1984) "In general, we have not been able to afford the luxury of excess long-haul bandwidth that the ARPANET possesses, and our long-haul links are heavily loaded during peak periods. Transit times of several seconds are thus common in our network." Today, it is unlikely that extra small packets would cause significant problems.

The TCP ACK-delay time can usually be adjusted globally as a system parameter. Linux offers a `TCP_QUICKACK` option, as a flag to `setsockopt()`, to disable delayed ACKs on a per-connection basis, but only until the next TCP system call. It must be invoked immediately after every receive operation to disable delayed ACKs entirely. This option is also not very portable.

The TSO option of *12.5 TCP Offloading*, used at the receiver, can also reduce the number of ACKs sent. If every two arriving data packets are consolidated via TSO into a single packet, then the receiver will appear to the sender to be acknowledging every other data packet. The ACK delay introduced by TSO is, however, usually quite small.

12.16 Nagle Algorithm

Like delayed ACKs, the Nagle algorithm (**RFC 896**) also attempts to improve the behavior of interactive small-packet applications. It specifies that a TCP endpoint generating small data segments should queue them until either it accumulates a full segment's worth or receives an ACK for the previous batch of small segments. If the full-segment threshold is not reached, this means that only one (consolidated) segment will be sent per RTT.

As an example, suppose A wishes to send to B packets containing consecutive letters, starting with "a". The application on A generates these every 100 ms, but the RTT is 501 ms. At T=0, A transmits "a". The application on A continues to generate "b", "c", "d", "e" and "f" at times 100 ms through 500 ms, but A does not send them immediately. At T=501 ms, ACK("a") arrives; at this point A transmits its backlogged "bcdef". The ACK for this arrives at T=1002, by which point A has queued "ghijk". The end result is that A sends a fifth as many packets as it would without the Nagle algorithm. If these letters are generated by a user typing them with telnet, and the ACKs also include the echoed responses, then if the user pauses the echoed responses will very soon catch up.

The Nagle algorithm does not always interact well with delayed ACKs, or with user expectations; see exercises 10.0 and 10.5. It can usually be disabled on a per-connection basis, in the BSD socket library by calling `setsockopt()` with the `TCP_NODELAY` flag.

12.17 TCP Flow Control

It is possible for a TCP sender to send data faster than the receiver can process it. When this happens, a TCP receiver may reduce the advertised Window Size value of an open connection, thus informing the sender to switch to a smaller window size. This provides support for **flow control**.

The window-size reduction appears in the ACKs sent back by the receiver. A given ACK is not supposed to reduce the window size by so much that the upper end of the window gets smaller. A window might shrink from the byte range [20,000..28,000] to [22,000..28,000] but never to [20,000..26,000].

If a TCP receiver uses this technique to shrink the advertised window size to 0, this means that the sender may not send data. The receiver has thus informed the sender that, yes, the data was received, but that, no, more may not yet be sent. This corresponds to the ACK_{WAIT} suggested in *6.1.3 Flow Control*. Eventually, when the receiver is ready to receive data, it will send an ACK increasing the advertised window size again.

If the TCP sender has its window size reduced to 0, and the ACK from the receiver increasing the window is lost, then the connection would be deadlocked. TCP has a special feature specifically to avoid this: if the window size is reduced to zero, the sender sends dataless packets to the receiver, at regular intervals. Each of these "polling" packets elicits the receiver's current ACK; the end result is that the sender will receive the eventual window-enlargement announcement reliably. These "polling" packets are regulated by the so-called **persist** timer.

12.18 Silly Window Syndrome

The silly-window syndrome is a term for a scenario in which TCP transfers only small amounts of data at a time. Because TCP/IP packets have a minimum fixed header size of 40 bytes, sending small packets uses

the network inefficiently. The silly-window syndrome can occur when either by the receiving application consuming data slowly or when the sending application generating data slowly.

As an example involving a slow-consuming receiver, suppose a TCP connection has a window size of 1000 bytes, but the receiving application consumes data only 10 bytes at a time, at intervals about equal to the RTT. The following can then happen:

- The sender sends bytes 1-1000. The receiving application consumes 10 bytes, numbered 1-10. The receiving TCP buffers the remaining 990 bytes and sends an ACK reducing the window size to 10, per *12.17 TCP Flow Control*.
- Upon receipt of the ACK, the sender sends 10 bytes numbered 1001-1010, the most it is permitted. In the meantime, the receiving application has consumed bytes 11-20. The window size therefore remains at 10 in the next ACK.
- the sender sends bytes 1011-1020 while the application consumes bytes 21-30. The window size remains at 10.

The sender may end up sending 10 bytes at a time indefinitely. This is of no benefit to either side; the sender might as well send larger packets less often. The standard fix, set forth in **RFC 1122**, is for the receiver to use its ACKs to keep the window at 0 until it has consumed one full packet's worth (or half the window, for small window sizes). At that point the sender is invited – by an appropriate window-size advertisement in the next ACK – to send another full packet of data.

The silly-window syndrome can also occur if the sender is *generating* data slowly, say 10 bytes at a time. The Nagle algorithm, above, can be used to prevent this, though for interactive applications sending small amounts of data in separate but closely spaced packets may actually be useful.

12.19 TCP Timeout and Retransmission

When TCP sends a packet containing user data (this excludes ACK-only packets), it sets a timeout. If that timeout expires before the packet data is acknowledged, it is retransmitted. Acknowledgments are sent for every arriving data packet (unless Delayed ACKs are implemented, *12.15 TCP Delayed ACKs*); this amounts to receiver-side retransmit-on-duplicate of *6.1.1 Packet Loss*. Because ACKs are cumulative, and so a later ACK can replace an earlier one, lost ACKs are seldom a problem.

For TCP to work well for both intra-server-room and trans-global connections, with RTTs ranging from well under 1 ms to close to 1 second, the length of the timeout interval must *adapt*. TCP manages this by maintaining a running estimate of the RTT, EstRTT. In the original version, TCP then set TimeOut = 2×EstRTT (in the literature, the TCP TimeOut value is often known as RTO, for Retransmission TimeOut). EstRTT itself was a running average of periodically measured SampleRTT values, according to

$$\text{EstRTT} = \alpha\times\text{EstRTT} + (1\text{-}\alpha)\times\text{SampleRTT}$$

for a fixed α, $0<\alpha<1$. Typical values of α might be α=1/2 or α=7/8. For α close to 1 this is "conservative" in that EstRTT is slow to change. For α closer to 0, EstRTT is more volatile.

There is a potential RTT measurement ambiguity: if a packet is sent twice, the ACK received could be in response to the first transmission or the second. The Karn/Partridge algorithm resolves this: on packet loss (and retransmission), the sender

- Doubles Timeout

- Stops recording SampleRTT
- Uses the doubled Timeout as EstRTT when things resume

Setting TimeOut = 2×EstRTT proved too short during congestion periods and too long other times. Jacobson and Karels (*[JK88]*) introduced a way of calculating the TimeOut value based on the statistical variability of EstRTT. After each SampleRTT value was collected, the sender would also update EstDeviation according to

> SampleDev = | SampleRTT – EstRTT |
>
> EstDeviation = β×EstDeviation + (1-β)×SampleDev

for a fixed β, $0<\beta<1$. Timeout was then set to EstRTT + 4×EstDeviation. EstDeviation is an estimate of the so-called *mean deviation*; 4 mean deviations corresponds (for normally distributed data) to about 5 *standard* deviations. If the SampleRTT values were normally distributed (which they are not), this would mean that the chance that a non-lost packet would arrive outside the TimeOut period is vanishingly small.

Keeping track of when packets time out is usually handled by putting a record for each packet sent into a **timer list**. Each record contains the packet's timeout time, and the list is kept sorted by these times. Periodically, *eg* every 100 ms, the list is inspected and all packets with expired timeout are then retransmitted. When an ACK arrives, the corresponding packet timeout record is removed from the list. Note that this approach means that a packet's timeout processing may be slightly late.

12.20 KeepAlive

There is no reason that a TCP connection should not be idle for a long period of time; ssh/telnet connections, for example, might go unused for days. However, there is the turned-off-at-night problem: a workstation might telnet into a server, and then be shut off (not shut down gracefully) at the end of the day. The connection would now be half-open, but the server would not generate any traffic and so might never detect this; the connection itself would continue to tie up resources.

KeepAlive in action

One evening long ago, when dialed up (yes, that long ago) into the Internet, my phone line disconnected while I was typing an email message in an ssh window. I dutifully reconnected, expecting to find my message in the file "dead.letter", which is what would have happened had I been disconnected while using the even-older tty dialup. Alas, nothing was there. I reconstructed my email as best I could and logged off.
The next morning, there was my lost email in a file "dead.letter", dated two hours after the initial crash! What had happened, apparently, was that the original ssh connection on the server side just hung there, half-open. Then, after two hours, KeepAlive kicked in, and aborted the connection. At that point ssh sent my mail program the HangUp signal, and the mail program wrote out what it had in "dead.letter".

To avoid this, TCP supports an optional **KeepAlive** mechanism: each side "polls" the other with a dataless packet. The original **RFC 1122** KeepAlive timeout was 2 hours, but this could be reduced to 15 minutes. If a connection failed the KeepAlive test, it would be closed.

Supposedly, some TCP implementations are not exactly **RFC 1122**-compliant: either KeepAlives are enabled by default, or the KeepAlive interval is much smaller than called for in the specification.

12.21 TCP timers

To summarize, TCP maintains the following four kinds of timers. All of them can be maintained by a single timer list, above.

- **TimeOut**: a per-segment timer; TimeOut values vary widely
- 2×MSL **TIMEWAIT**: a per-connection timer
- **Persist**: the timer used to poll the receiving end when winsize = 0
- **KeepAlive**, above

12.22 Variants and Alternatives

One alternative to TCP is UDP with programmer-implemented timout and retransmission; many RPC implementations (*11.5 Remote Procedure Call (RPC)*) do exactly this, with reasonable results. Within a LAN a static timeout of around half a second usually works quite well (unless the LAN has some tunneled links), and implementation of a simple timeout-retransmission mechanism is quite straightforward. Implementing adaptive timeouts as in *12.19 TCP Timeout and Retransmission* can, however, be a bit trickier. QUIC (*11.1.1 QUIC*) is an example of this strategy.

We here consider four other protocols. The first, MPTCP, is based on TCP itself. The second, SCTP, is a message-oriented alternative to TCP that is an entirely separate protocol. The last two, DCCP and QUIC, are attempts to create a TCP-like transport layer on top of UDP.

12.22.1 MPTCP

Multipath TCP, or MPTCP, allows connections to use multiple network interfaces on a host, either sequentially or simultaneously. MPTCP architectural principles are outlined in **RFC 6182**; implementation details are in **RFC 6824**.

To carry the actual traffic, MPTCP arranges for the creation of multiple standard-TCP **subflows** between the sending and receiving hosts; these subflows typically connect between different pairs of IP addresses on the respective hosts.

For example, a connection to a server can start using the client's wired Ethernet interface, and continue via Wi-Fi after the user has unplugged. If the client then moves out of Wi-Fi range, the connection might continue via a mobile network. Alternatively, MPTCP allows the parallel use of multiple Ethernet interfaces on both client and server for higher throughput.

MPTCP officially forbids the creation of multiple TCP connections between a single pair of interfaces in order to simulate Highspeed TCP (*15.5 Highspeed TCP*); **RFC 6356** spells out an MWTCP congestion-control algorithm to enforce this.

Suppose host A, with two interfaces with IP addresses A_1 and A_2, wishes to connect to host B with IP addresses B_1 and B_2. Connection establishment proceeds via the ordinary TCP three-way handshake, between one of A's IP addresses, say A_1, and one of B's, B_1. The SYN packets must each carry the `MP_CAPABLE` TCP option, to signal one another that MPTCP is supported. As part of the `MP_CAPABLE` option, A and

B also exchange pseudorandom 64-bit connection keys, sent unencrypted; these will be used to sign later messages as in *22.6.1 Secure Hashes and Authentication*. This first connection is the initial subflow.

Once the MPTCP initial subflow has been established, additional subflow connections can be made. Usually these will be initiated from the client side, here A, though the B side can also do this. At this point, however, A does not know of B's address B_2, so the only possible second subflow will be from A_2 to B_1. New subflows will carry the `MP_JOIN` option with their initial SYN packets, along with digital signatures signed by the original connection keys verifying that the new subflow is indeed part of this MPTCP connection.

At this point A and B can send data to one another using both connections simultaneously. To keep track of data, each side maintains a 64-bit data sequence number, DSN, for the data it sends; each side also maintains a mapping between the DSN and the subflow sequence numbers. For example, A might send 1000-byte blocks of data alternating between the A_1 and A_2 connections; the blocks might have DSN values 10000, 11000, 12000, 13000, ... The A_1 subflow would then carry blocks 10000, 12000, *etc*, numbering these consecutively (perhaps 20000, 21000, ...) with its own sequence numbers. The sides exchange DSN mapping information with a `DSS` TCP option. This mechanism means that all data transmitted over the MWTCP connection can be delivered in the proper order, and that if one subflow fails, its data can be retransmitted on another subflow.

B can inform A of its second IP address, B_2, using the `ADD_ADDR` option. Of course, it is possible that B_2 is not directly reachable by A; for example, it might be behind a NAT router. But if B_2 *is* reachable, A can now open two more subflows A_1—B_2 and A_2—B_2.

All the above works equally well if either or both of A's addresses is behind a NAT router, simply because the NAT router is able to properly forward the subflow TCP connections. Addresses sent from one host to another, such as B's transmission of its address B_2, may be rendered invalid by NAT, but in this case A's attempt to open a connection to B_2 simply fails.

Generally, hosts can be configured to use multiple subflows in parallel, or to use one interface only as a backup, when the primary interface is unplugged or out of range. APIs have been proposed that allow an control over MPTCP behavior on a per-connection basis.

12.22.2 SCTP

The Stream Control Transmission Protocol, SCTP, is an entirely separate protocol from TCP, running directly above IP. It is, in effect, a message-oriented alternative to TCP: an application writes a sequence of messages and SCTP delivers each one as a unit, fragmenting and reassembling it as necessary. Like TCP, SCTP is connection-oriented and reliable. SCTP uses a form of sliding windows, and, like TCP, adjusts the window size to manage congestion.

An SCTP connection can support multiple **message streams**; the exact number is negotiated at startup. A retransmission delay in one stream never blocks delivery in other streams. Within each stream, SCTP messages are sequentially numbered, and are normally delivered in order of message number. A receiver can request, however, to receive messages immediately upon successful delivery, that is, potentially out of order. Either way, the data within each message is guaranteed to be delivered in order and without loss.

Internally, message data is divided into SCTP **chunks** for inclusion in packets. One SCTP packet can contain data chunks from different messages and different streams; packets can also contain control chunks.

Messages themselves can be quite large; there is no set limit. Very large messages may need to be received in multiple system calls (*eg* calls to `recvmsg()`).

SCTP supports an MPTCP-like feature by which each endpoint can use multiple network interfaces.

SCTP connections are set up using a four-way handshake, versus TCP's three-way handshake. The extra packet provides some protection against so-called SYN flooding (*12.3 TCP Connection Establishment*). The central idea is that if client A initiates a connection request with server B, then B allocates no resources to the connection until after B has received a response to its own message to A. This means that, at a minimum, A is a real host with a real IP address.

The full four-way handshake between client A and server B is, in brief, as follows:

- A sends B an INIT chunk (corresponding to SYN), along with a pseudorandom Tag_A.
- B sends A an `INIT ACK`, with Tag_B and a **state cookie**. The state cookie contains all the information B needs to allocate resources to the connection, and is digitally signed (*22.6.1 Secure Hashes and Authentication*) with a key known only to B. Crucially, B does **not** at this point allocate any resources to the incipient connection.
- A returns the state cookie to B in a `COOKIE ECHO` packet.
- B enters the ESTABLISHED state and sends a `COOKIE ACK` to A. Upon receipt, A enters the ESTABLISHED state.

When B receives the `COOKIE ECHO`, it verifies the signature. At this point B knows that it sent the cookie to A and received a response, so A must exist. Only then does B allocate memory resources to the connection. Spoofed INITs in the first step cost B essentially nothing.

The Tag_A and Tag_B in the first two packets are called **verification tags**. From this point on, B will include Tag_A in every packet it sends to A, and vice-versa. Although these tags are sent unencrypted, they nonetheless make it much harder for an attacker to inject data into the connection.

Data can be included in the third and fourth packets above; *ie* A can begin sending data after one RTT.

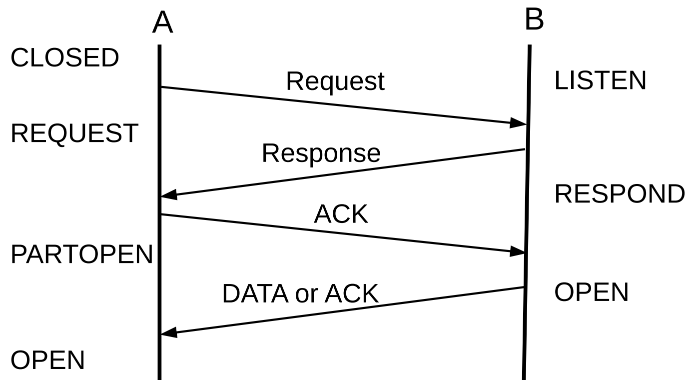
Unfortunately for potential SCTP applications, few if any NAT routers recognize SCTP; this limits the use of SCTP to Internet paths along which NAT is not used. In principle SCTP could simplify delivery of web pages, transmitting one page component per message, but lack of NAT support makes this infeasible. SCTP is also blocked by some middleboxes (*7.7.2 Middleboxes*) on the grounds that it is an unknown protocol, and therefore suspect. While this is not quite as common as the NAT problem, it is common enough to prevent by itself the widespread adoption of SCTP in the general Internet. SCTP *is* widely used for telecommunications signaling, both within and between providers, where NAT and recalcitrant middleboxes can be banished.

12.22.3 DCCP

As we saw in *11.1.2 DCCP*, DCCP is a UDP-based transport protocol that supports, among other things, connection establishment. While it is used much less often than TCP, it provides an alternative example of how transport can be done.

DCCP defines a set of distinct packet types, rather than TCP's independent packet flags; this disallows unforeseen combinations such as TCP SYN+RST. Connection establishment involves Request and Respond; data transmission involves Data, ACK and DataACK, and teardown involves CloseReq, Close and Reset. While one cannot have, for example, a Respond+ACK, Respond packets do carry an acknowledgment field.

Like TCP, DCCP uses a three-way handshake to open a connection; here is a diagram:



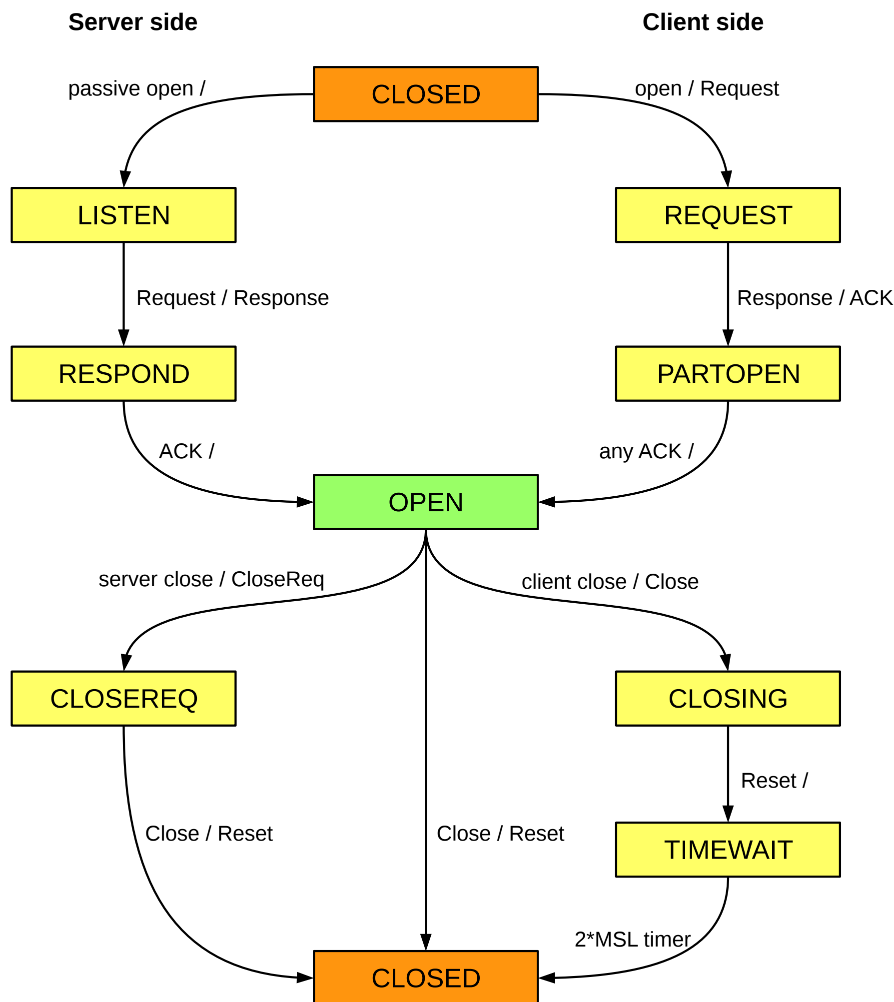
DCCP “three”-way handshake

The fourth packet, DATA or ACK, is not considered part of the handshake itself.

The OPEN state corresponds to TCP’s ESTABLISHED state. Like TCP, each side chooses an ISN (not shown in the diagram). Because packet delivery is not reliable, and because ACKs are not cumulative, the client remains in PARTOPEN state until it has confirmed that the server has received its ACK of the server’s Response. While in state PARTOPEN, the client can send ACK and DataACK but not ACK-less Data packets.

Packets are numbered sequentially. The numbering includes all packets, not just Data packets, and is by packet rather than by byte.

The DCCP state diagram is shown below. It is simpler than the TCP state diagram because DCCP does not support simultaneous opens.



DCCP State Diagram

To close a connection, one side sends Close and the other responds with Reset. Reset is used for normal close as well as for exceptional conditions. Because whoever sends the Close is then stuck with TIMEWAIT, the server side may send CloseReq to ask the client to send Close.

There are also two special packet formats, Sync and SyncAck, for resynchronizing sequence numbers after a burst of lost packets.

The other major TCP-like feature supported by DCCP is congestion control; see *14.6.3 DCCP Congestion Control*.

12.22.4 QUIC Revisited

Like DCCP, QUIC is also a UDP-based transport protocol, aimed rather squarely at HTTP plus TLS (*22.10.2 TLS*). The fundamental goal of QUIC is to provide TLS encryption protection with as little overhead as possible, in a manner that competes fairly with TCP in the presence of congestion. Opening a QUIC connection, encryption included, takes a single RTT. QUIC can also be seen, however, as a complete rewrite

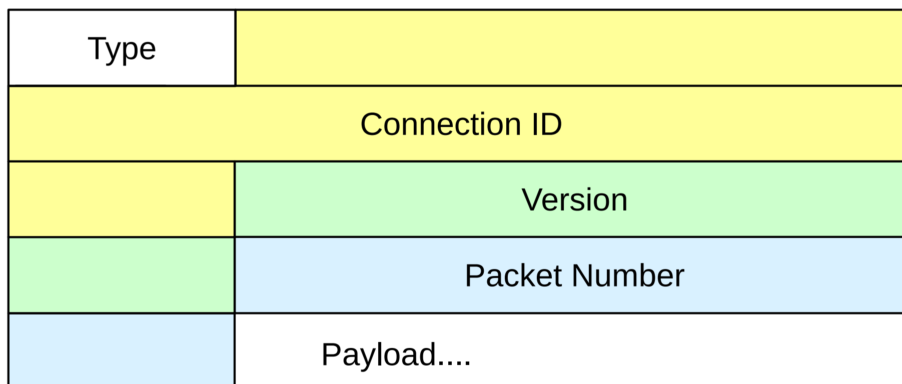
of TCP from the ground up; a reading of specific features sheds quite a bit of light on how the corresponding TCP features have fared over the past thirty-odd years. QUIC is currently (2018) documented in a set of Internet Drafts:

- Transport basics: draft-ietf-quic-transport
- Loss and congestion management: draft-ietf-quic-recovery
- TLS encryption over QUIC: draft-ietf-quic-tls
- HTTP over QUIC: draft-ietf-quic-http

The design of QUIC was influenced by the fate of SCTP above; the latter, as a new protocol above IP, was sometimes blocked by overly security-conscious middleboxes (*7.7.2 Middleboxes*).

12.22.4.1 Headers

We will start with the QUIC header. While there are some alternative forms, the basic header is diagrammed below, with a 1-byte `Type` field, an 8-byte `Connection ID`, and 4-byte `Version` and `Packet Number` fields.



Typical QUIC Long Header

Perhaps the most striking thing about this header is that 4-byte alignment – used consistently in the IPv4, IPv6, UDP and TCP headers – has been completely abandoned. On most contemporary processors, the performance advantages of alignment are negligible; see the last paragraph at *7.1 The IPv4 Header*.

IP packets are identified as such by the Ethernet type field, and TCP and UDP packets are identified as such by the IPv4-header Protocol field. But QUIC packets are *not* identified as such by any flag in the preceding IP or UDP headers; there is in fact no place in those headers for a QUIC marker to go. QUIC appears to an observer as just another form of UDP traffic. This acts as a form of middlebox defense; QUIC packets cannot be identified as such in isolation. WireShark, sidebar below, identifies QUIC packets by looking at the whole history of the connection, and even then must make some (educated) guesses. Middleboxes could do that too, but it would take work.

The initial `Connection ID` consists of 64 random bits chosen by the client. The server, upon accepting the connection, may change the `Connection ID`; at that point the `Connection ID` is fixed for the lifetime of the connection. The `Connection ID` may be omitted for packets whose connection can

be determined from the associated IP address and port values; this is signaled by the `Type` field. The `Connection ID` can also be used to migrate a connection to a different IP address and port, as might happen if a mobile device moves out of range of Wi-Fi and the mobile-data plan continues the communication. This may also happen if a connection passes through a NAT router. The NAT forwarding entry may time out (see the comment on UDP and inactivity at *7.7 Network Address Translation*), and the connection may be assigned a different outbound UDP port if it later resumes. QUIC uses the `Connection ID` to recognize that the reassigned connection is still the same one as before.

The `Version` field gets dropped as soon as the version is negotiated. As part of the version negotiation, a packet might have multiple version fields. Such packets put a *random* value into the low-order seven bits of the `Type` field, as a prevention against middleboxes' blocking unknown types. This way, aggressive middlebox behavior should be discovered early, before it becomes widespread.

QUIC-watching

QUIC packets can be observed in WireShark by using the filter string "quic". To generate QUIC traffic, use a Chromium-based browser and go to a Google-operated site, say, google.com. Often the only non-encrypted fields are the `Type` field and the packet number.

The packet number can be reduced to one or two bytes once the connection is established; this is signaled by the `Type` field. Internally, QUIC uses packet numbers in the range 0 to 2^{62}; these internal numbers are not allowed to wrap around. The low-order 32 bits (or 16 bits or 8 bits) of the internal number are what is transmitted in the packet header. A packet receiver infers the high-order bits from the most recent acknowledgment.

The initial packet number is to be chosen randomly in the range 0 to 2^{32}–1025.

Use of 16-bit or 8-bit transmitted packet numbers is restricted to cases where there can be no ambiguity. At a minimum, this means that the number of outstanding packets (the QUIC winsize) cannot exceed 2^7–1 for 8-bit packet numbering or 2^{15}–1 for 16-bit packet numbering. These maximum winsizes represent the ideal case where there is no packet reordering; smaller values are likely to be used in practice. (See *6.5 Exercises*, exercise 9.0.)

12.22.4.2 Frames and streams

Data in a QUIC packet is partitioned into one or more **frames**. Each frame's data is prefixed by a simple frame header indicating its length and type. Some frames contain management information; frames containing higher-layer data are called STREAM frames. Each frame must be fully contained in one packet.

The application's data can be divided into multiple **streams**, depending on the application requirements. This is particularly useful with HTTP, as a client may request a large number of different resources (html, images, javascript, *etc*) simultaneously. Stream data is contained in STREAM frames. Streams are numbered, with Stream 0 reserved for the TLS cryptographic handshake. The HTTP/2 protocol has introduced its own notion of streams; these map neatly onto QUIC streams.

The two low-order bits of each stream number indicate whether the stream was initiated by the client or by the server, and whether it is bi- or uni-directional. This design decision means that either side can create a stream and send data on it immediately, *without negotiation*; this is important for reducing unnecessary RTTs.

Each individual stream is guaranteed in-order delivery, but there are no ordering guarantees between different streams. Within a packet, the data for a particular stream is contained in a frame for that stream.

One packet can contain stream frames for multiple streams. However, if a packet is lost, streams that have frames contained in that packet are blocked until retransmission. Other streams can continue without interruption. This creates an incentive for keeping separate streams in separate packets.

Stream frames contain the byte offset of the frame's block of stream data (starting from 0), to enable in-order stream reassembly. TCP, as we have seen, uses this byte-numbering approach exclusively, though starting with the Initial Sequence Number rather than zero. QUIC's stream-level numbering by byte is unrelated to its top-level numbering by packet.

In addition to stream frames, there are a large number of management frames. Here are a few of them:

- `RST_STREAM`: like TCP RST, but for one stream only.
- `MAX_DATA`: this corresponds to the TCP advertised window size. As with TCP, it can be reduced to zero to pause the flow of data and thereby implement flow control. There is also a similar `MAX_STREAM_DATA`, applying per stream.
- `PING` and `PONG`: to verify that the other endpoint is still responding. These serve as the equivalent of TCP KEEPALIVEs, among other things.
- `CONNECTION_CLOSE` and `APPLICATION_CLOSE`: these initiate termination of the connection; they differ only in that a `CONNECTION_CLOSE` might be accompanied by a QUIC-layer error or explanation message while an `APPLICATION_CLOSE` might be accompanied by, say, an HTTP error/explanation messge.
- `PAD`: to pad out the packet to a larger size.
- `ACK`: for acknowledgments, below.

12.22.4.3 Acknowledgments

QUIC assigns a new, sequential packet number (the `Packet ID`) to every packet, including retransmissions. TCP, by comparison, assigns sequence numbers to each byte. (QUIC stream frames do number data by byte, as noted above.)

Lost QUIC packets are retransmitted, but with a new packet number. This makes it impossible for a receiver to send cumulative acknowledgments, as lost packets will never be acknowledged. The receiver handles this as below. At the sender side, the sender maintains a list of packets it has sent that are both unacknowledged and also not known to be lost. These represent the packets **in flight**. When a packet is retransmitted, its old packet number is removed from this list, as lost, and the new packet number replaces it.

To the extent possible given this retransmission-renumbering policy, QUIC follows the spirit of sliding windows. It maintains a state variable `bytes_in_flight`, corresponding to TCP's winsize, listing the total size of all the packets in flight. As with TCP, new acknowledgments allow new transmissions.

Acknowledgments themselves are sent in special acknowledgment frames. These begin with the number of the highest packet received. This is followed by a list of pairs, as long as will fit into the frame, consisting of the length of the next block of contiguous packets received followed by the length of the intervening gap of packets *not* received. The TCP Selective ACK (*13.6 Selective Acknowledgments (SACK)*) option is similar, but is limited to three blocks of received packets. It is quite possible that some of the gaps in a QUIC ACK

frame refer to lost packets that were long since retransmitted with new packet numbers, but this does not matter.

The sender is allowed to skip packet numbers occasionally, to prevent the receiver from trying to increase throughput by acknowledging packets not yet received. Unlike with TCP, acknowledging an unsent packet is considered to be a fatal error, and the connection is terminated.

As with TCP, there is a delayed-ACK timer, but, while TCP's is typically 250 ms, QUIC's is 25 ms. QUIC also includes in each ACK frame the receiver's best estimate of the elapsed time between arrival of the most recent packet and the sending of the ACK it triggered; this allows the sender to better estimate the RTT. The primary advantage of the design decision not to reuse packet IDs is that there is never any ambiguity as to a retransmitted packet's RTT, as there is in TCP (*12.19 TCP Timeout and Retransmission*). Note, however, that because QUIC runs in a user process and not the kernel, it may not be able to respond immediately to an arriving packet, and so the time-delay estimate may be slightly short.

ACK frames are not themselves acknowledged. This means that, in a one-way data flow, the receiver may have no idea if its ACKs are getting through. The receiver may send a PING frame to the sender, which will respond not only with a matching PONG frame but also an ACK frame acknowledging the receiver's recent acknowledgment packets.

QUIC adjusts its `bytes_in_flight` value to manage congestion, much as TCP manages its winsize (or more properly its `cwnd`, *13 TCP Reno and Congestion Management*) for the same purpose. Specifically, QUIC attempts to mimic the congestion response of TCP Cubic, *15.15 TCP CUBIC*, and so should in theory compete fairly with TCP Cubic connections.

12.22.4.4 Connection handshake and TLS encryption

The opening of a QUIC connection makes use of the TLS handshake, *22.10.2 TLS*, specifically TLS v1.3, *22.10.2.4.2 TLS version 1.3*. A client wishing to connect sends a QUIC `Initial` packet, containing the TLS `ClientHello` message. The server responds (with a `ServerHello`) in a QUIC `Handshake` packet. (There is also a `Retry` packet, for special situations.) The TLS negotiation is contained in QUIC's Stream 0. While the TLS and QUIC handshake rules are rather precise, there is as yet no formal state-diagram description of connection opening.

The `Initial` packet also contains a set of QUIC **transport parameters** declared unilaterally by the client; the server makes a similar declaration in its response. These parameters include, among other things, the maximum packet size, the connection's idle timeout, and initial value for `MAX_DATA`, above.

An important feature of TLS v1.3 is that, if the client has connected to the server previously and still has the key negotiated in that earlier session, it can use that old key to send an encrypted application-layer request (in a STREAM frame) immediately following the `Initial` packet. This is called **0-RTT** protection (or encryption). The advantage of this is that the client may receive an answer from the server within a single RTT, versus four RTTs for traditional TCP (one for the TCP three-way handshake, two for TLS negotiation, and one for the application request/reply). As discussed at *22.10.2.4.2 TLS version 1.3*, requests submitted with 0-RTT protection must be idempotent, to prevent replay attacks.

Once the server's first `Handshake` packet makes it back to the client, the client is in possession of the key negotiated by the new session, and will encrypt everything using that going forward. This is known as the **1-RTT** key, and all further data is said to be 1-RTT protected. The negotiated key is initially calculated by

the TLS layer, which then exports it to QUIC. The QUIC layer then encrypts the entire data portion of its packets, using the format of **RFC 5116**.

The QUIC header is not encrypted, but is still covered by an authentication checksum, making it impossible for middleboxes to rewrite anything. Such rewriting has been observed for TCP, and has sometimes complicated TCP evolution.

The type field of a QUIC packet contains a special code to mark 0-RTT data, ensuring that the receiver will know what level of protection is in effect.

When a QUIC server receives the `ClientHello` and sends off its `ServerHello`, it has not yet received any evidence that the client "owns" the IP address it claims to have; that is, that the client is not spoofing its IP address (*12.10.1 ISNs and spoofing*). Because of the idempotency restriction on responses to 0-RTT data, the server cannot give away privileges if spoofed in this way by a client. The server may, however, be an unwitting participant in a **traffic-amplification** attack, if the real client can trigger the sending by the server to a spoofed client of a larger response than the real client sends directly. The solution here is to require that the QUIC `Initial` packet, containing the `ClientHello`, be at least 1200 bytes. The server's `Handshake` response is likely to be smaller, and so represents no amplification of traffic.

To close the connection, one side sends a `CONNECTION_CLOSE` or `APPLICATION_CLOSE`. It may continue to send these in response to packets from the other side. When the other side receives the `CLOSE` packet, it should send its own, and then enter the so-called **draining** state. When the initiator of the close receives the other side's echoed `CLOSE`, it too will enter the draining state. Once in this state, an endpoint may not send any packets. The draining state corresponds to TCP's TIMEWAIT (*12.9 TIMEWAIT*), for the purpose of any lost final ACKs; it should last three RTT's. There is no need of a TIMEWAIT analog to prevent old duplicates, as a second QUIC connection will select a new `Connection ID`.

QUIC connection closing has no analog of TCP's feature in which one side sends FIN and the other continues to send data indefinitely, *12.7.1 Closing a connection*. This use of FIN, however, is allowed in bidirectional streams; the per-stream (and per-direction) FIN bit lives in the stream header. Alternatively, one side can send its request and close its stream, and the other side can then answer on a different stream.

12.23 Epilog

At this point we have covered the basic mechanics of TCP, but have one important topic remaining: how TCP manages its window size so as to limit congestion, while maintaining fairness. This turns out to be complex, and will be the focus of the next three chapters.

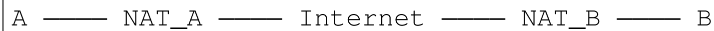
12.24 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercise 4.5 is distinct, for example, from exercises 4.0 and 5.0.

1.0. Experiment with the TCP version of simplex-talk. How does the server respond differently with threading enabled and without, if two simultaneous attempts to connect are made, from two different client instances?

2.0. Trace the states visited if nodes A and B attempt to create a TCP connection by *simultaneously* sending each other SYN packets, that then cross in the network. Draw the ladder diagram, and label the states on each side. Hint: there should be two pairs of crossing packets. A SYN+ACK counts as an ACK.

2.5. Suppose nodes A and B are each behind their own NAT firewall (*7.7 Network Address Translation*).



A and B attempt to connect to one another simultaneously, using TCP. A sends to the public IPv4 address of NAT_B, and vice-versa for B. Assume that neither NAT_A nor NAT_B changes the port numbers in outgoing packets, at least for the packets involved in this connection attempt. Show that the connection succeeds.

3.0. When two nodes A and B simultaneously attempt to connect to one another using the OSI TP4 protocol, two bidirectional network connections are created (rather than one, as with TCP). If TCP had instead chosen the TP4 semantics here, what would have to be added to the TCP header? Hint: if a packet from ⟨A,port1⟩ arrives at ⟨B,port2⟩, how would we tell to which of the two possible connections it belongs?

4.0. Simultaneous connection initiations are rare, but simultaneous connection termination is relatively common. How do two TCP nodes negotiate the simultaneous sending of FIN packets to one another? Draw the ladder diagram, and label the states on each side. Which node goes into TIMEWAIT state? Hint: there should be two pairs of crossing packets.

4.5. The state diagram at *12.7 TCP state diagram* shows a dashed path from FIN_WAIT_1 to TIMEWAIT on receipt of FIN+ACK. All FIN packets contain a valid ACK field, but that is not what is meant here. Under what circumstances is this direct arc from FIN_WAIT_1 to TIMEWAIT taken? Explain why this arc can never be used during simultaneous close. Hint: consider the ladder diagram of a "normal" close.

5.0. (a) Suppose you see multiple connections on your workstation in state FIN_WAIT_1. What is likely going on? Whose fault is it? | (b). What might be going on if you see connections languishing in state FIN_WAIT_2?

6.0. Suppose that, after downloading a file, the client host is unplugged from the network, so it can send no further packets. The server's connection is still in the ESTABLISHED state. In each case below, use the TCP state diagram to list all states that are reachable by the server.

(a). Before being unplugged, the client was in state ESTABLISHED; *ie* it had *not* sent the first FIN.

(b). Before being unplugged the client had sent its FIN, and moved to FIN_WAIT_1.

(Eventually, the server connection should transition to CLOSED due to repeated timeouts, but this is not shown in the state diagram.)

6.5. In *12.3 TCP Connection Establishment* we noted that RST packets had to have a valid SYN value, but that "**RFC 793** does not require the RST packet's ACK value to match". There is an exception for RST packets arriving at state SYN-SENT: "the RST is acceptable if the ACK field acknowledges the SYN". Explain the reasoning behind this exception.

7.0. Suppose A and B create a TCP connection with ISN_A=20,000 and ISN_B=5,000. A sends three 1000-byte packets (Data1, Data2 and Data3 below), and B ACKs each. Then B sends a 1000-byte packet DataB

to A and terminates the connection with a FIN. In the table below, fill in the SEQ and ACK fields for each packet shown.

A sends	B sends
SYN, ISN_A=20,000	
	SYN, ISN_B=5,000, ACK=______
ACK, SEQ=______, ACK=______	
Data1, SEQ=______, ACK=______	
	ACK, SEQ=______, ACK=______
Data2, SEQ=______, ACK=______	
	ACK, SEQ=______, ACK=______
Data3, SEQ=______, ACK=______	
	ACK, SEQ=______, ACK=______
	DataB, SEQ=______, ACK=______
ACK, SEQ=_____, ACK=______	
	FIN, SEQ=______, ACK=______

8.0. Suppose you are downloading a large file, and there is a progress bar showing how much of the file has been downloaded. For definiteness, assume the progress bar moves 1 mm per MB, the throughput averages 0.5 MB per second (so the progress bar advances at a rate of 0.5 mm/sec), and the winsize is 5 MB.

A packet is now lost, and is retransmitted after a timeout. What will happen to the progress bar? If someone measured the progress bar at two times 1 second apart, just before and just after the lost packet arrived, what value would they calculate for the throughput?

9.0. Suppose you are creating software for a streaming-video site. You want to limit the video read-ahead – the gap between how much has been downloaded and how much the viewer has actually watched – to approximately 1 MB; the server should pause in sending when necessary to enforce this. On the other hand, you do want the receiver to be able to read ahead by up to this much. You should assume that the TCP connection throughput will be higher than the actual video-data-consumption rate.

(a). Suppose the TCP window size happens to be exactly 1 MB. If the receiver simply reads each video frame from the TCP connection, displays it, and then pauses briefly before reading the next frame in accordance with the frame rate, explain how the flow-control mechanism of *12.17 TCP Flow Control* will achieve the desired effect.

(b). Applications, however, cannot control their TCP window size. What support would you have to add to the video-transfer *application* to allow it to read ahead by 1 MB but not to exceed this? Hint: both client and server sides of the application will have to implement something to enable this feature.

10.0. A user moves the computer mouse and sees the mouse-cursor's position updated on the screen. Suppose the mouse-position updates are being transmitted over a TCP connection with a relatively long RTT. The user attempts to move the cursor to a specific point. How will the user perceive the mouse's motion

(a). with the Nagle algorithm

(b). without the Nagle algorithm

10.5. Host A sends two single-byte packets, one containing "x" and the other containing "y", to host B. A implements the Nagle algorithm and B implements delayed ACKs, with a 500 ms maximum delay. The RTT is negligible. How long does the transmission take? Draw a ladder diagram.

11.0. Suppose you have fallen in with a group that wants to add to TCP a feature so that, if A and B1 are connected, then B1 can **hand off** its connection to a different host B2; the end result is that A and B2 are connected and A has received an uninterrupted stream of data. Either A or B1 can initiate the handoff.

(a). Suppose B1 is the host to send the final FIN (or HANDOFF) packet to A. How would you handle appropriate analogues of the TIMEWAIT state for host B1? Does the fact that A is continuing the connection, just not with B1, matter?

(b). Now suppose A is the party to send the final FIN/HANDOFF, to B1. What changes to TIMEWAIT would have to be made at A's end? Note that A may potentially hand off the connection again and again, *eg* to B3, B4 and then B5.

12.0. Suppose A connects to B via TCP, and sends the message "Attack at noon", followed by FIN. Upon receiving this, B is sure it has received the entire message.

(a). What can A be sure of upon receiving B's own FIN+ACK?

(b). What can B be sure of upon receiving A's final ACK?

(c). What is A not absolutely sure of after sending its final ACK?

13.0. Host A connects to the Internet via Wi-Fi, receiving IPv4 address 10.0.0.2, and then opens a TCP connection *conn1* to remote host B. After *conn1* is established, A's Ethernet cable is plugged in. A's Ethernet interface receives IP address 10.0.0.3, and A automatically selects this new Ethernet connection as its default route. **Assume** that A now starts using 10.0.0.3 as the source address of packets it sends as part of *conn1* (contrary to **RFC 1122**).

Assume also that A's TCP implementation is such that when a packet arrives from $\langle B_{IP}, B_{port} \rangle$ to $\langle A_{IP}, A_{port} \rangle$ and this socketpair is to be matched to an existing TCP connection, the field A_{IP} is allowed to be any of A's IP addresses (that is, either 10.0.0.2 or 10.0.0.3); it does not have to match the IP address with which the connection was originally negotiated.

(a). Explain why *conn1* will now fail, as soon as any packet is sent from A. Hint: the packet will be sent from 10.0.0.3. What will B send in response? In light of the second assumption, how will A react to B's response packet?

(The author regularly sees connections appear to fail this way. Perhaps some justification for this behavior is that, at the time of establishment of *conn1*, A was not yet multihomed.)

(b). Now suppose all four fields of the socketpair (⟨B_{IP}, B_{port}⟩, ⟨A_{IP}, A_{port}⟩) are used to match an incoming packet to its corresponding TCP connection. The connection *conn1* still fails, though not as immediately. Explain what happens.

See also *7.9.5 ARP and multihomed hosts*, *7 IP version 4* exercise 11.0, and *9 Routing-Update Algorithms* exercise 13.0.

14.0. Modify the simplex-talk server of *12.6 TCP simplex-talk* so that `line_talker()` breaks out of the `while` loop as soon as it has printed the first string received (or simply remove the while loop). Once out of the `while` loop, the existing code calls `s.close()`.

(a). Start up the modified server, and connect to it with a client. Send a single message line, and use `netstat` to examine the TCP states of the client and server. What are these states?

(b). Send two message lines to the server. What are the TCP states of the client and server?

(c). Send three message lines to the server. Is there an error message at the client?

(d). Send two message lines to the server, while monitoring packets with WireShark. The WireShark filter expression `tcp.port == 5431` may be useful for eliminating irrelevant traffic. What FIN packets do you see? Do you see a RST packet?

15.0. Outline a scenario in which TCP endpoint A sends data to B and then calls `close()` on its socket, and after the connection terminates B has not received all the data, even though the network has not failed. In the style of *12.6.1 The TCP Client*, A's code might look like this:

```
s = new Socket(dest, destport);
sout = s.getOutputStream();
sout.write(large_buffer)
s.close()
```

Hint: see *12.7.2 Calling close()*.

This chapter addresses how TCP manages congestion, both for the connection's own benefit (to improve its throughput) and for the benefit of other connections as well (which may result in our connection *reducing* its own throughput). Early work on congestion culminated in 1990 with the flavor of TCP known as **TCP Reno**. The congestion-management mechanisms of TCP Reno remain the dominant approach on the Internet today, though alternative TCPs are an active area of research and we will consider a few of them in *15 Newer TCP Implementations*.

The central TCP mechanism here is for a connection to adjust its window size. A smaller winsize means fewer packets are out in the Internet at any one time, and less traffic means less congestion. A larger winsize means better throughput, up to a point. All TCPs reduce winsize when congestion is apparent, and increase it when it is not. The trick is in figuring out when and by how much to make these winsize changes. Many of the improvements to TCP have come from mining more and more information from the stream of returning ACKs.

The Anternet

The Harvester Ant *Pogonomyrmex barbatus* uses a mechanism related to TCP Reno to "decide" how many ants should be out foraging at any one time *[PDG12]*. The rate of ants leaving the nest to forage is closely tied to the rate of returning foragers; if foragers return quickly (meaning more food is available), the total number of foragers will increase (like the increasing winsize below). The ant algorithm is probabilistic, however, while most TCP algorithms are deterministic.

Recall Chiu and Jain's definition from *1.7 Congestion* that the "knee" of congestion occurs when the queue first starts to grow, and the "cliff" of congestion occurs when packets start being dropped. Congestion can be managed at either point, though dropped packets can be a significant waste of resources. Some newer TCP strategies attempt to take action at the congestion knee (starting with *15.6 TCP Vegas*), but TCP Reno is a cliff-based strategy: packets must be lost before the sender reduces the window size.

In *20 Quality of Service* we will consider some router-centric alternatives to TCP for Internet congestion management. However, for the most part these have not been widely adopted, and TCP is all that stands in the way of Internet congestive collapse.

The first question one might ask about TCP congestion management is just how did it get this job? A TCP sender is expected to monitor its transmission rate so as to *cooperate* with other senders to reduce overall congestion among the routers. While part of the goal of every TCP node is good, stable performance for its own connections, this emphasis on end-user cooperation introduces the prospect of "cheating": a host might be tempted to maximize the throughput of its own connections at the expense of others. Putting TCP nodes in charge of congestion among the core routers is to some degree like putting the foxes in charge of the henhouse. More accurately, such an arrangement has the potential to lead to the Tragedy of the Commons. Multiple TCP senders share a common resource – the Internet backbone – and while the backbone is most efficient if every sender cooperates, each individual sender can improve its own situation by sending faster than allowed. Indeed, one of the arguments used by virtual-circuit routing adherents is that it provides support for the implementation of a wide range of congestion-management options under control of a central authority.

Nonetheless, TCP has been quite successful at distributed congestion management. In part this has been because system vendors do have an incentive to take the big-picture view, and in the past it has been quite difficult for individual users to replace their TCP stacks with rogue versions. Another factor contributing to TCP's success here is that most bad TCP behavior requires cooperation at the *server* end, and most server managers have an incentive to behave cooperatively. Servers generally want to distribute bandwidth fairly among their multiple clients, and – theoretically at least – a server's ISP could penalize misbehavior. So far, at least, the TCP approach has worked remarkably well.

13.1 Basics of TCP Congestion Management

TCP's congestion management is **window-based**; that is, TCP adjusts its window size to adapt to congestion. The window size can be thought of as the number of packets out there in the network; more precisely, it represents the number of packets and ACKs either in transit or enqueued. An alternative approach often used for real-time systems is **rate-based** congestion management, which runs into an unfortunate difficulty if the sending rate momentarily happens to exceed the available rate.

In the very earliest days of TCP, the window size for a TCP connection came from the AdvertisedWindow value suggested by the receiver, essentially representing how many packet buffers it could allocate. This value is often quite large, to accommodate large bandwidth×delay products, and so is often reduced out of concern for congestion. When winsize is adjusted downwards for this reason, it is generally referred to as the **Congestion Window**, or `cwnd` (a variable name first appearing in Berkeley Unix). Strictly speaking, winsize = min(`cwnd`, AdvertisedWindow). In newer TCP implementations, the variable `cwnd` may actually be used to mean the sender's estimate of the number of packets in flight; see the sidebar at *13.4 TCP Reno and Fast Recovery*.

If TCP is sending over an idle network, the per-packet RTT will be RTT_{noLoad}, the travel time with no queuing delays. As we saw in *6.3.2 RTT Calculations*, ($RTT–RTT_{noLoad}$) is the time each packet spends in the queue. The path bandwidth is winsize/RTT, and so the number of packets in queues is winsize × ($RTT–RTT_{noLoad}$) / RTT. Usually all the queued packets are at the router at the head of the bottleneck link. Note that the sender can calculate this number (assuming we can estimate RTT_{noLoad}; the most common approach is to assume that the smallest RTT measured corresponds to RTT_{noLoad}).

TCP's self-clocking (*ie* that new transmissions are paced by returning ACKs) guarantees that, again assuming an otherwise idle network, the queue will build only at the bottleneck router. Self-clocking means that the rate of packet transmissions is equal to the available bandwidth of the bottleneck link. There are some spikes when a burst of packets is sent (*eg* when the sender increases its window size), but in the steady state self-clocking means that packets accumulate only at the bottleneck.

We will return to the case of the *non*-otherwise-idle network in the next chapter, in *14.2 Bottleneck Links with Competition*.

The "optimum" window size for a TCP connection would be bandwidth × RTT_{noLoad}. With this window size, the sender has exactly filled the transit capacity along the path to its destination, and has used none of the queue capacity.

Actually, TCP Reno does not do this.

Instead, TCP Reno does the following:

- guesses at a reasonable initial window size, using a form of polling

- slowly increases the window size if no losses occur, on the theory that maximum available throughput may not yet have been reached
- rapidly decreases the window size otherwise, on the theory that if losses occur then drastic action is needed

In practice, this usually leaves TCP's window size well above the theoretical "optimum".

One interpretation of TCP's approach is that there is a time-varying "ceiling" on the number of packets the network can accept. Each sender tries to stay near but just below this level. Occasionally a sender will overshoot and a packet will be dropped somewhere, but this just teaches the sender a little more about where the network ceiling is. More formally, this ceiling represents the largest `cwnd` that does not lead to packet loss, *ie* the `cwnd` that at that particular moment completely fills but does not overflow the bottleneck queue. We have reached the ceiling when the queue is full.

In Chiu and Jain's terminology, the far side of the ceiling is the "cliff", at which point packets are lost. TCP tries to stay above the "knee", which is the point when the queue first begins to be persistently utilized, thus keeping the queue at least partially occupied; whenever it sends too much and falls off the "cliff", it retreats.

The ceiling concept is often useful, but not necessarily as precise as it might sound. If we have reached the ceiling by *gradually* expanding the sliding-windows window size, then winsize will be as large as possible. But if the sender suddenly releases a burst of packets, the queue may fill and we will have reached a "temporary ceiling" without fully utilizing the transit capacity. Another source of ceiling ambiguity is that the bottleneck link may be shared with other connections, in which case the ceiling represents our connection's particular share, which may fluctuate greatly with time. Finally, at the point when the ceiling is reached, the queue is *full* and so there are a considerable number of packets waiting in the queue; it is not possible for a sender to pull back instantaneously.

It is time to acknowledge the existence of different versions of TCP, each incorporating different congestion-management algorithms. The two we will start with are **TCP Tahoe** (1988) and **TCP Reno** (1990); the names Tahoe and Reno were originally the codenames of the Berkeley Unix distributions that included these respective TCP implementations. The ideas behind TCP Tahoe came from a 1988 paper by Jacobson and Karels *[JK88]*; TCP Reno then refined this a couple years later. TCP Reno is still in widespread use over twenty years later, and is still the undisputed TCP reference implementation, although some modest improvements (NewReno, SACK) have crept in.

A common theme to the development of improved implementations of TCP is for one end of the connection (usually the sender) to extract greater and greater amounts of information from the packet flow. For example, TCP Tahoe introduced the idea that duplicate ACKs likely mean a lost packet; TCP Reno introduced the idea that returning duplicate ACKs are associated with packets that have successfully been transmitted but follow a loss. TCP Vegas (*15.6 TCP Vegas*) introduced the fine-grained measurement of RTT, to detect when $\text{RTT} > \text{RTT}_{\text{noLoad}}$.

It is often helpful to think of a TCP sender as having breaks between successive windowfuls; that is, the sender sends `cwnd` packets, is briefly idle, and then sends another `cwnd` packets. The successive windowfuls of packets are often called **flights**. The existence of any separation between flights is, however, not guaranteed.

13.1.1 The Somewhat-Steady State

We will begin with the state in which TCP has established a reasonable guess for cwnd, comfortably below the Advertised Window Size, and which largely appears to be working. TCP then engages in some fine-tuning. This TCP "steady state" – steady here in the sense of regular oscillation – is usually referred to as the **congestion avoidance** phase, though all phases of the process are ultimately directed towards avoidance of congestion. The central strategy is that when a packet is lost, cwnd should decrease rapidly, but otherwise should increase "slowly". This leads to slow oscillation of cwnd, which over time allows the average cwnd to adapt to long-term changes in the network capacity.

As TCP finishes each windowful of packets, it notes whether a loss occurred. The cwnd-adjustment rule introduced by TCP Tahoe and *[JK88]* is the following:

- if there were no losses in the previous windowful, cwnd = cwnd +1
- if packets were lost, cwnd = cwnd/2

We are informally measuring cwnd in units of full packets; strictly speaking, cwnd is measured in bytes and is incremented by the maximum TCP segment size.

This strategy here is known as **Additive Increase, Multiplicative Decrease**, or AIMD; cwnd = cwnd+1 is the additive increase and cwnd = cwnd/2 is the multiplicative decrease. Typically, setting cwnd=cwnd/2 is a medium-term goal; in fact, TCP Tahoe briefly sets cwnd=1 in the immediate aftermath of an actual timeout. With no losses, TCP will send successive windowfuls of, say, 20, 21, 22, 23, 24, ... This amounts to conservative "probing" of the network and, in particular, of the queue at the bottleneck router. TCP tries larger cwnd values because the absence of loss means the current cwnd is below the "network ceiling"; that is, the queue at the bottleneck router is not yet overfull.

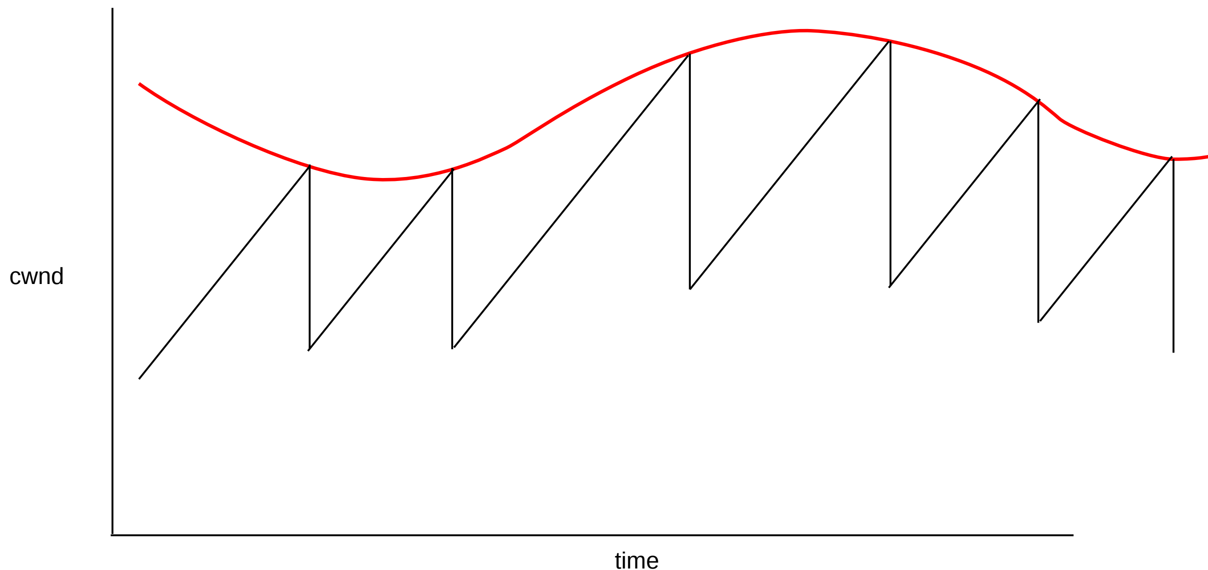
If a loss occurs (including multiple losses in a single windowful), TCP's response is to cut the window size in half. (As we will see, TCP Tahoe actually handles this in a somewhat roundabout way.) Informally, the idea is that the sender needs to respond aggressively to congestion. More precisely, lost packets mean the queue of the bottleneck router has filled, and the sender needs to dial back to a level that will allow the queue to clear. If we assume that the transit capacity is roughly equal to the queue capacity (say each is equal to N), then we overflow the queue and drop packets when cwnd = 2N, and so cwnd = cwnd/2 leaves us with cwnd = N, which just fills the transit capacity and leaves the queue empty. (When the sender sets cwnd=N, the actual number of packets in transit takes at least one RTT to fall from 2N to N.)

Of course, assuming any relationship between transit capacity and queue capacity is highly speculative. On a 5,000 km fiber-optic link with a bandwidth of 10 Gbps, the round-trip transit capacity would be about 60 MB, or 60,000 1KB packets. Most routers probably do not have queues that large. Queue capacities in excess of the transit capacity are common, however. On the other hand, a competing model of a long-haul high-bandwidth TCP path is that the queue size should be a small fraction of the bandwidth×delay product. We return to this in *13.7 TCP and Bottleneck Link Utilization* and *13.7.1 Bufferbloat*.

Note that if TCP experiences a packet loss, and there is an actual timeout (as opposed to a packet loss detected by Fast Retransmit, *13.3 TCP Tahoe and Fast Retransmit*), then the sliding-window pipe has drained. No packets are in flight. No self-clocking can govern new transmissions. Sliding windows therefore needs to restart from scratch.

The congestion-avoidance algorithm leads to the classic "TCP sawtooth" graph, where the peaks are at the points where the slowly rising cwnd crossed above the "network ceiling". We emphasize that the

TCP sawtooth is specific to TCP Reno and related TCP implementations that share Reno's additive-increase/multiplicative-decrease mechanism.



TCP Sawtooth, red curve represents the network capacity

During periods of no loss, TCP's `cwnd` increases linearly; when a loss occurs, TCP sets `cwnd` = `cwnd`/2. This diagram is an idealization as when a loss occurs it takes the sender some time to discover it, perhaps as much as the TimeOut interval.

The fluctuation shown here in the red ceiling curve is somewhat arbitrary. If there are only one or two other competing senders, the ceiling variation may be quite dramatic, but with many concurrent senders the variations may be smoothed out.

For some TCP sawtooth graphs created through actual simulation, see *16.2.1 Graph of cwnd v time* and *16.4.1 Some TCP Reno cwnd graphs*.

13.1.1.1 A first look at fairness

The transit capacity of the path is more-or-less unvarying, as is the physical capacity of the queue at the bottleneck router. However, these capacities are also shared with other connections, which may come and go with time. This is why the ceiling does vary in real terms. If two other connections share a path with total capacity 60 packets, the "fairest" allocation might be for each connection to get about 20 packets as its share. If one of those other connections terminates, the two remaining ones might each rise to 30 packets. And if instead a fourth connection joins the mix, then after equilibrium is reached each connection might hope for a fair share of 15 packets.

Will this kind of "fair" allocation actually happen? Or might we end up with one connection getting 90% of the bandwidth while two others each get 5%?

Chiu and Jain *[CJ89]* showed that the additive-increase/multiplicative-decrease algorithm does indeed converge to roughly equal bandwidth sharing when two connections have a common bottleneck link, provided also that

- both connections have the same RTT
- during any given RTT, either both connections experience a packet loss, or neither connection does

To see this, let cwnd1 and cwnd2 be the connections' congestion-window sizes, and consider the quantity cwnd1 – cwnd2. For any RTT in which there is no loss, cwnd1 and cwnd2 both increment by 1, and so cwnd1 – cwnd2 stays the same. If there is a loss, then both are cut in half and so cwnd1 – cwnd2 is also cut in half. Thus, over time, the original value of cwnd1 – cwnd2 is repeatedly cut in half (during each RTT in which losses occur) until it dwindles to inconsequentiality, at which point cwnd1 ≃ cwnd2.

Graphical and tabular versions of this same argument are in the next chapter, in *14.3 TCP Fairness with Synchronized Losses*.

The second bulleted hypothesis above we may call the **synchronized-loss hypothesis**. While it is very reasonable to suppose that the two connections will experience the same number of losses as a long-term *average*, it is a much stronger statement to suppose that all loss events are shared by both connections. This behavior may not occur in real life and has been the subject of some debate; see *[GV02]*. We return to this point in *16.3 Two TCP Senders Competing*. Fortunately, equal-RTT fairness still holds if each connection is *equally likely* to experience a packet loss: both connections will have the same loss rate, and so, as we shall see in *14.5 TCP Reno loss rate versus cwnd*, will have the same cwnd. However, convergence to fairness may take rather much longer. In *14.3 TCP Fairness with Synchronized Losses* we also look at some alternative hypotheses for the unequal-RTT case.

13.2 Slow Start

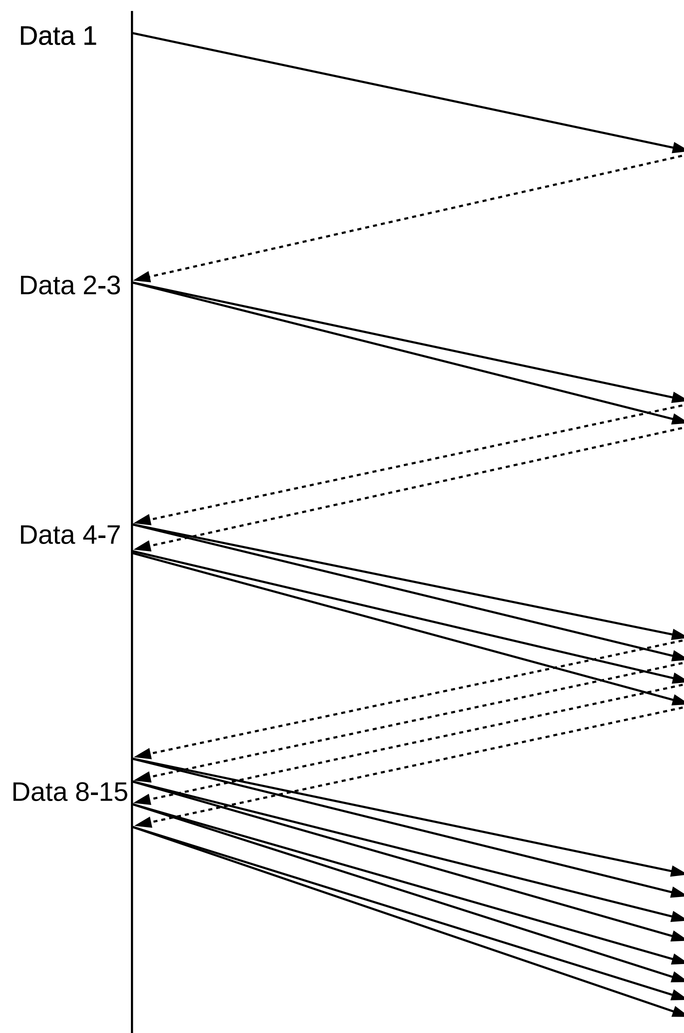
How do we make that initial guess as to the network capacity? What value of cwnd should we begin with? And even if we have a good target for cwnd, how do we avoid flooding the network sending an initial burst of packets?

The answer is known as **slow start**. If you are trying to guess a number in a fixed range, you are likely to use binary search. Not knowing the range for the "network ceiling", a good strategy is to guess cwnd=1 (or cwnd=2) at first and keep doubling until you have gone too far. Then revert to the previous guess, which is known to have worked. At this point you are guaranteed to be within 50% of the true capacity.

The actual slow-start mechanism is to increment cwnd by 1 for each ACK received. This seems linear, but that is misleading: after we send a windowful of packets (cwnd many), we have received cwnd ACKs and so have incremented cwnd-many times, and so have set cwnd to (cwnd+cwnd) = 2×cwnd. In other words, cwnd=cwnd×2 after each *windowful* is the same as cwnd+=1 after each *packet*.

Assuming packets travel together in windowfuls, all this means cwnd *doubles* each RTT during slow start; this is possibly the only place in the computer science literature where exponential growth is described as "slow". It is indeed slower, however, than the alternative of sending an entire windowful at once.

Here is a diagram of slow start in action. This diagram makes the implicit assumption that the no-load RTT is large enough to hold well more than the 8 packets of the maximum window size shown.



Slow Start with discrete packet flights

For a different case, with a much smaller RTT, see *13.2.3 Slow-Start Multiple Drop Example*.

Eventually the bottleneck queue gets full, and drops a packet. Let us suppose this is after N RTTs, so `cwnd`=2^N. Then during the previous RTT, `cwnd`=2^{N-1} worked successfully, so we go back to that previous value by setting `cwnd` = `cwnd`/2.

13.2.1 Per-ACK Responses

During slow start, incrementing `cwnd` by one per ACK received is equivalent to doubling `cwnd` after each windowful. We can find a similar equivalence for the congestion-avoidance phase, above.

During congestion avoidance, `cwnd` is incremented by 1 after each windowful. To formulate this as a **per-ACK** increase, we spread this increment of 1 over the entire windowful, which of course has size `cwnd`. This amounts to the following upon each ACK received:

cwnd = cwnd + 1/cwnd

This is a slight approximation, because cwnd keeps changing, but it works well in practice. Because TCP actually measures cwnd in bytes, floating-point arithmetic is normally not required; see exercise 13.0. An exact equivalent to the per-windowful incrementing strategy is cwnd = cwnd + 1/$cwnd_0$, where $cwnd_0$ is the value of cwnd at the start of that particular windowful. Another, simpler, approach is to use cwnd += 1/cwnd, and to keep the fractional part recorded, but to use floor(cwnd) (the integer part of cwnd) when actually sending packets.

Most actual implementations keep track of cwnd in bytes, in which case using integer arithmetic is sufficient until cwnd becomes quite large.

If **delayed ACKs** are implemented (*12.15 TCP Delayed ACKs*), then in bulk transfers one arriving ACK actually acknowledges two packets. **RFC 3465** permits a TCP receiver to increment cwnd by 2/cwnd in that situation, which is the response consistent with incrementing cwnd by 1 upon receipt of enough ACKs to acknowledge an entire windowful.

13.2.2 Threshold Slow Start

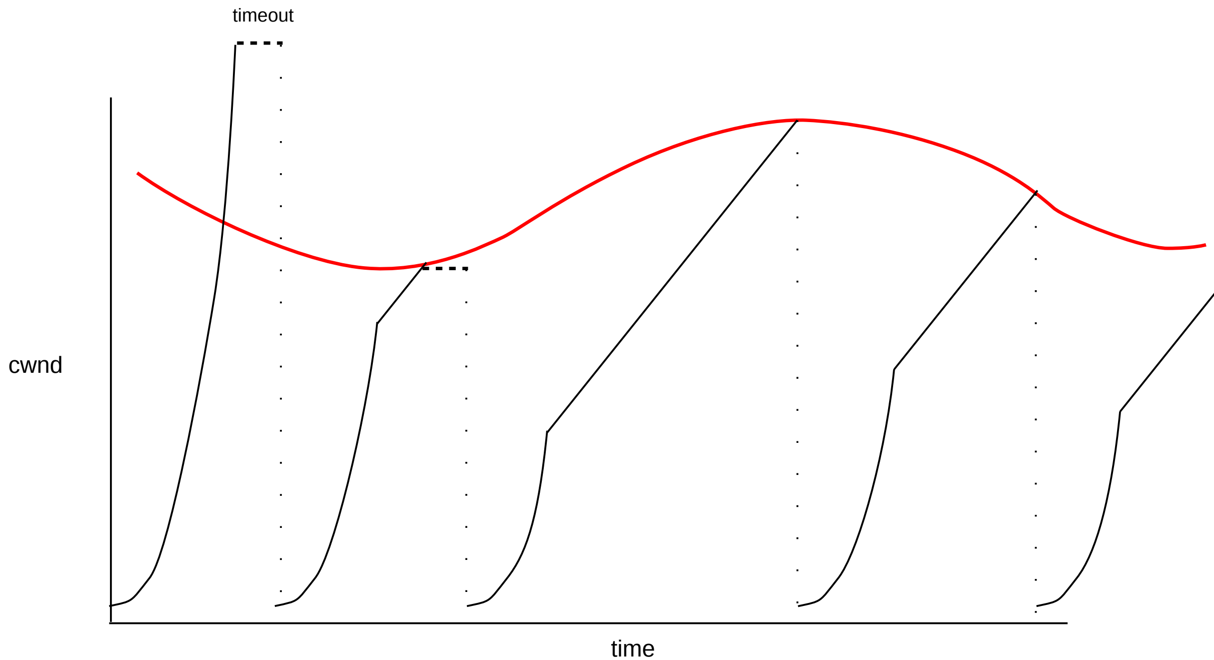
Sometimes TCP uses slow start even when it knows the working network capacity. After a packet loss and timeout, TCP knows that a new cwnd of $cwnd_{old}$/2 should work. If cwnd had been 100, TCP halves it to 50. The problem, however, is that after timeout there are no returning ACKs to self-clock the continuing transmission, and we do not want to dump 50 packets on the network all at once. So in restarting the flow TCP uses what might be called **threshold slow start**: it uses slow-start, but stops when cwnd reaches the target. Specifically, on packet loss we set the variable ssthresh to cwnd/2; this is our new target for cwnd. We set cwnd itself to 1, and switch to the slow-start mode (cwnd += 1 for each ACK). However, as soon as cwnd reaches ssthresh, we switch to the congestion-avoidance mode (cwnd += 1/cwnd for each ACK). Note that the transition from threshold slow start to congestion avoidance is completely natural, and easy to implement.

TCP will use threshold slow-start whenever it is restarting from a pipe drain; that is, every time slow-start is needed after its very first use. (If a connection has simply been *idle*, non-threshold slow start is typically used when traffic starts up again.)

Threshold slow-start can be seen as an attempt at combining rapid window expansion with self-clocking.

By comparison, we might refer to the initial, non-threshold slow start as **unbounded slow start**. Note that unbounded slow start serves a fundamentally different purpose – initial probing to determine the network ceiling to within 50% – than threshold slow start.

Here is the TCP sawtooth diagram above, modified to show timeouts and slow start. The first two packet losses are displayed as “coarse timeouts”; the rest are displayed as if Fast Retransmit, below, were used.



TCP Tahoe Sawtooth, red curve represents the network capacity
Slow Start is used after each packet loss until ssthresh is reached

RFC 2581 allows slow start to begin with `cwnd`=2.

13.2.3 Slow-Start Multiple Drop Example

Slow start has the potential to cause multiple dropped packets at the bottleneck link; packet losses continue for quite some time because the TCP sender is slow to discover them. The network topology is as follows, where the A–R link is infinitely fast and the R–B link has a bandwidth in the R⟶B direction of 1 packet/ms.

Assume that R has a queue capacity of 100, not including the packet it is currently forwarding to B, and that ACKs travel instantly from B back to A. In this and later examples we will continue to use the Data[N]/ACK[N] terminology of *6.2 Sliding Windows*, beginning with N=1; TCP numbering is not done quite this way but the distinction is inconsequential.

When A uses slow-start here, the successive windowfuls will almost immediately begin to overlap. A will send one packet at T=0; it will be delivered at T=1. The ACK will travel instantly to A, at which point A will send two packets. From this point on, ACKs will arrive regularly at A at a rate of one per second. Here is a brief chart:

Time	A receives	A sends	R sends	R's queue
0		Data[1]	Data[1]	
1	ACK[1]	Data[2],Data[3]	Data[2]	Data[3]
2	ACK[2]	4,5	3	4,5
3	ACK[3]	6,7	4	5..7
4	ACK[4]	8,9	5	6..9
5	ACK[5]	10,11	6	7..11
..				
N	ACK[N]	2N,2N+1	N+1	N+2 .. 2N+1

At T=N, R's queue contains N packets. At T=100, R's queue is full. Data[200], sent at T=100, will be delivered and acknowledged at T=200, giving it an RTT of 100. At T=101, R receives Data[202] and Data[203] and drops the latter one. Unfortunately, A's timeout interval must of course be greater than the RTT, and so A will not detect the loss until, at an absolute minimum, T=200. At that point, A has sent packets up through Data[401], and the 100 packets Data[203], Data[205], ..., Data[401] have all been lost. In other words, at the point when A *first* receives the news of one lost packet, in fact at least 100 packets have already been lost.

Fortunately, unbounded slow start generally occurs only once per connection.

13.2.4 Summary of TCP so far

So far we have the following features:

- Unbounded slow start at the beginning
- Congestion avoidance with AIMD once some semblance of a steady state is reached
- Threshold slow start after each loss
- Each threshold slow start transitioning naturally to congestion avoidance

Here is a table expressing the slow-start and congestion-avoidance phases in terms of manipulating `cwnd`.

phase	`cwnd` change, loss	`cwnd` change, no loss	
	per window	per window	per ACK
slow start	`cwnd`/2	`cwnd` *= 2	`cwnd` += 1
cong avoid	`cwnd`/2	`cwnd` +=1	`cwnd` += 1/`cwnd`

Viewing cwnd

Linux users can view the current values of `cwnd` and `ssthresh`, as well as a host of other TCP statistics, using the command `ss --tcp --info`.

The problem TCP often faces, in both slow-start and congestion-avoidance phases, is that when a packet is lost the sender will not detect this until much later (at least until the bottleneck router's current queue has been sent); by then, it may be too late to avoid further losses.

13.3 TCP Tahoe and Fast Retransmit

TCP Tahoe has one more important feature. Recall that TCP ACKs are cumulative; if packets 1 and 2 have been received and now Data[4] arrives, but not yet Data[3], all the receiver can (and must!) do is to send back another ACK[2]. Thus, from the sender's perspective, if we send packets 1,2,3,4,5,6 and get back ACK[1], ACK[2], ACK[2], ACK[2], ACK[2], we can infer two things:

- Data[3] got lost, which is why we are stuck on ACK[2]
- Data 4,5 and 6 probably *did* make it through, and triggered the three duplicate ACK[2]s (the three ACK[2]s following the first ACK[2]).

The **Fast Retransmit** strategy is to resend Data[N] when we have received three dupACKs for Data[N-1]; that is, four ACK[N-1]'s in all. Because this represents a packet loss, we also set `ssthresh` = `cwnd`/2, set `cwnd`=1, and begin the threshold-slow-start phase. The effect of this is typically to reduce the delay associated with the lost packet from that of a full timeout, typically $2\times$RTT, to just a little over a single RTT. The lost packet is now discovered *before* the TCP pipeline has drained. However, at the end of the next RTT, when the ACK of the retransmitted packet will return, the TCP pipeline *will* have drained, hence the need for slow start.

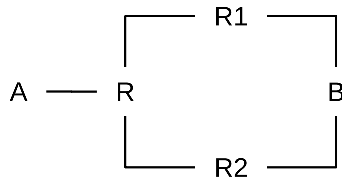
TCP Tahoe included all the features discussed so far: the `cwnd`+=1 and `cwnd`=`cwnd`/2 responses, slow start and Fast Retransmit.

Fast Retransmit waits for the *third* dupACK to allow for the possibility of moderate packet reordering. Suppose packets 1 through 6 are sent, but they arrive in the order 1,3,4,2,6,5, perhaps due to a router along the way with an architecture that is strongly parallelized. Then the ACKs that would be sent back would be as follows:

Received	Response
Data[1]	ACK[1]
Data[3]	ACK[1]
Data[4]	ACK[1]
Data[2]	ACK[4]
Data[6]	ACK[4]
Data[5]	ACK[6]

Waiting for the third dupACK is in most cases a successful compromise between responsiveness to lost packets and reasonable evidence that the data packet in question is actually lost.

However, a router that does more substantial delivery reordering would wreck havoc on connections using Fast Retransmit. In particular, consider the router R in the diagram below; when sending packets to B it might in principle wish to alternate on a packet-by-packet basis between the path via R1 and the path via R2. This would be a mistake; if the R1 and R2 paths had different propagation delays then this strategy would introduce major packet reordering. R should send all the packets belonging to any one TCP connection via a single path.



In the real world, routers generally go to considerable lengths to accommodate Fast Retransmit; in particular, use of multiple paths for a single TCP connection is almost universally frowned upon. Some actual data on packet reordering can be found in *[VP97]*; the author suggests that a switch to retransmission on the second dupACK would be risky.

13.4 TCP Reno and Fast Recovery

Fast Retransmit requires a sender to set `cwnd`=1 because the pipe has drained and there are no arriving ACKs to pace transmission. Fast Recovery is a technique that often allows the sender to avoid draining the pipe, and to move from `cwnd` to `cwnd`/2 in the space of a single RTT. TCP Reno is TCP Tahoe with the addition of Fast Recovery.

The idea is to use the arriving dupACKs to pace retransmission. We make the assumption that each arriving dupACK indicates that *some* data packet following the lost packet has been delivered successfully; it turns out not to matter which one. On discovery of the lost packet through Fast Retransmit, we set `cwnd`=`cwnd`/2; the next step is to figure out how many dupACKs we have to wait for before we can resume transmissions of new data.

Initially, at least, we assume that only one data packet is lost, though in the following section we will see that multiple losses can be handled via a slight modification of the Fast Recovery strategy.

During the recovery process, we cannot use `cwnd` directly, as the sliding window cannot budge until the lost packet is retransmitted. Instead we will use the concept of Estimated FlightSize, or **EFS**, which is the sender's best guess at the number of outstanding packets. Under normal circumstances, EFS is the same as `cwnd`. The crucial Fast Recovery observation is that EFS should be decremented by 1 for each arriving dupACK.

Linux `cwnd` *is* Estimated FlightSize

This chapter defines `cwnd` to be the sender winsize strictly construed. As such, packet N+`cwnd` cannot be sent until packet N is ACKed. However, the linux kernel actually uses `cwnd` as a synonym for Estimated FlightSize, which simplifies the Fast-Recovery code. This usage applies, in fact, to all TCP varieties, though it often makes little difference. See tcp_cwnd_test().

We first outline the general case, and then look at a specific example. Let `cwnd` = N, and suppose packet 1 is lost (packet numbers here may be taken as relative). Until packet 1 is retransmitted, the sender can only send up through packet N (Data[N] can be sent only after ACK[0] has arrived at the sender). The receiver will send N–1 dupACK[0]s representing packets 2 through N.

At the point of the third dupACK, when the loss of Data[1] is discovered, the sender calculates as follows: EFS had been `cwnd` = N. Three dupACKs have arrived, representing three later packets no longer in flight,

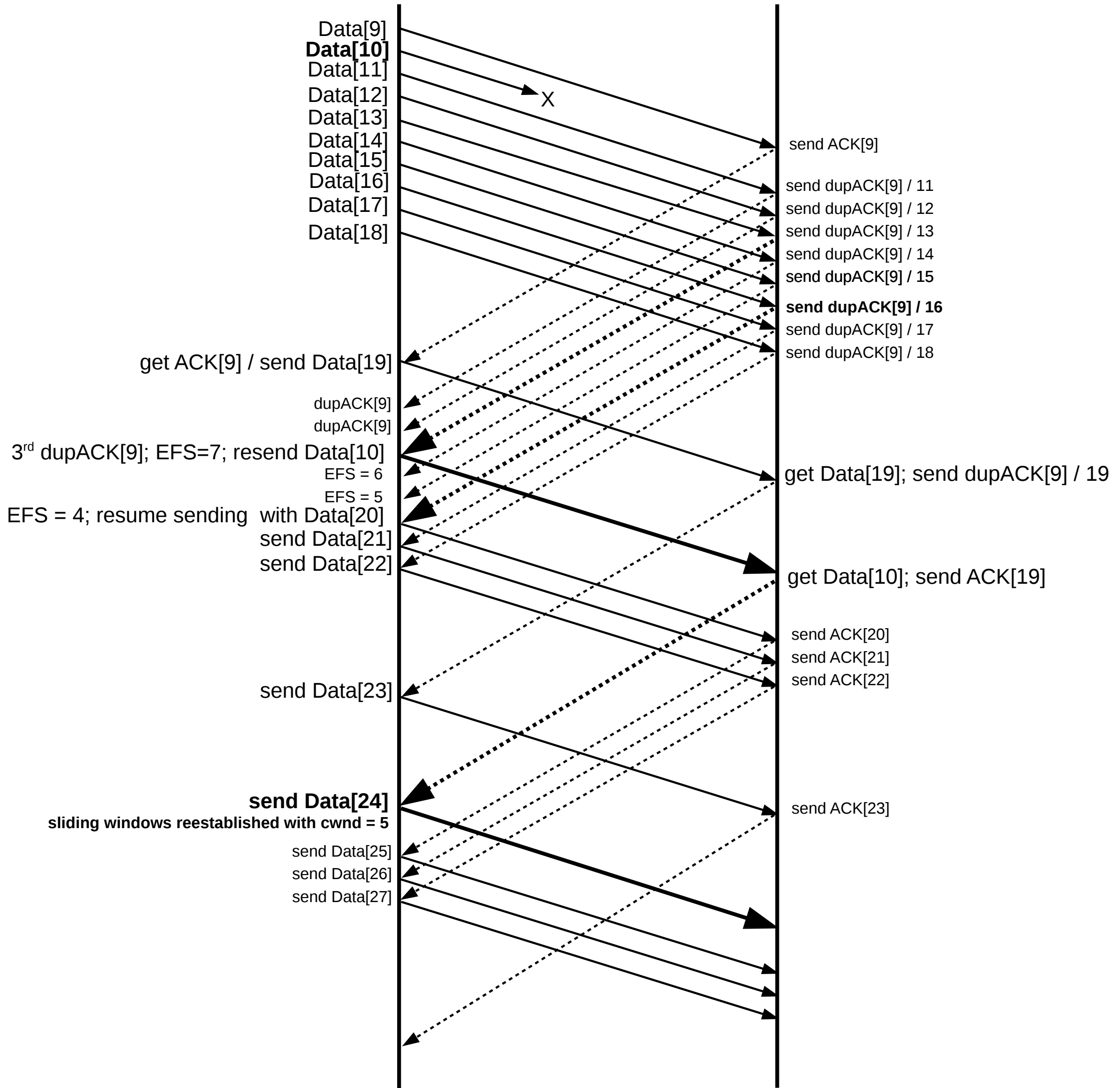
so EFS is now N–3. At this point the sender realizes a packet has been lost, which makes EFS = N–4 briefly, but that packet is then immediately retransmitted, which brings EFS back to N–3.

The sender expects at this point to receive N–4 more dupACKs, followed by one new ACK for the retransmission of the packet that was lost. This last ACK will be for the entire original windowful.

The new target for `cwnd` is N/2 (for simplicity, we will assume N is even). So, we wait for N/2 – 3 more dupACKs to arrive, at which point EFS is N–3–(N/2–3) = N/2. *After* this point the sender will resume sending new packets; it will send one new packet for each of the N/2–1 subsequently arriving dupACKs (recall that there are N–1 dupACKs in all). These new transmissions will be Data[N+1] through Data[N+(N/2–1)].

After the last of the dupACKs will come the ACK corresponding to the retransmission of the lost packet; it will be ACK[N], acknowledging all of the original windowful. At this point, there are N/2 – 1 unacknowledged packets Data[N+1] through Data[N+(N/2)–1]. The sender now sends Data[N+N/2] and is thereby able to resume sliding windows with `cwnd` = N/2: the sender has received ACK[N] and has exactly one full windowful outstanding for the new value N/2 of `cwnd`. That is, we are right where we are supposed to be.

Here is a diagram illustrating Fast Recovery for `cwnd`=10. Data[10] is lost.



Data[9] elicits the initial ACK[9], and the nine packets Data[11] through Data[19] each elicit a dupACK[9]. We denote the dupACK[9] elicited by Data[N] by dupACK[9]/N; these are shown along the upper right. Unless SACK TCP (below) is used, the sender will have no way to determine N or to tell these dupACKs apart. When dupACK[9]/13 (the third dupACK) arrives at the sender, the sender uses Fast Recovery to infer that Data[10] was lost and retransmits it. At this point EFS = 7: the sender has sent the original batch of 10 data packets, plus Data[19], and received one ACK and three dupACKs, for a total of 10+1-1-3 = 7. The sender has also inferred that Data[10] is lost (EFS –= 1) but then retransmitted it (EFS += 1). Six more dupACK[9]'s are on the way.

EFS is decremented for each subsequent dupACK arrival; after we get two more dupACK[9]'s, EFS is 5. The next dupACK[9] (dupACK[9]/16) reduces EFS to 4 and so allows us transmit Data[20] (which promptly bumps EFS back up to 5). We have

receive	send
dupACK[9]/16	Data[20]
dupACK[9]/17	Data[21]
dupACK[9]/18	Data[22]
dupACK[9]/19	Data[23]

We emphasize again that the TCP sender does not see the numbers 16 through 19 in the receive column above; it determines when to begin transmission by counting dupACK[9] arrivals.

Working Backwards

Figuring out when a fast-recovery sender should resume transmissions of new data is error-prone. Perhaps the simplest approach is to work backwards from the retransmitted lost packet: it should trigger at the receiver an ACK for the entire original windowful. When Data[10] above was lost, the "stuck" window was Data[10]-Data[19]. The retransmitted Data[10] thus triggers ACK[19]; when ACK[19] arrives, `cwnd` should be 10/2 = 5 so Data[24] should be sent. That in turn means the four packets Data[20] through Data[23] must have been sent earlier, via Fast Recovery. There are 10–1 = 9 dupACKs, so to send on the last four we must start with the sixth. The diagram above indeed shows new Fast Recovery transmissions beginning with the sixth dupACK.

The next thing to arrive at the sender side is the ACK[19] elicited by the retransmitted Data[10]; at the point Data[10] arrives at the receiver, Data[11] through Data[19] have already arrived and so the cumulative-ACK response is ACK[19]. The sender responds to ACK[19] with Data[24], and the transition to `cwnd`=5 is now complete.

During sliding windows without losses, a sender will send `cwnd` packets per RTT. If a "coarse" timeout occurs, typically it is not discovered until after at least one complete RTT of link idleness; there are additional underutilized RTTs during the slow-start phase. It is worth examining the Fast Recovery sequence shown in the illustration from the perspective of underutilized bandwidth. The diagram shows three round-trip times, as seen from the sender side. During the first RTT, the ten packets Data[9]-Data[18] are sent. The second RTT begins with the sending of Data[19] and continues through sending Data[22], along with the retransmitted Data[10]. The third RTT begins with sending Data[23], and includes through Data[27]. In terms of recovery efficiency, the RTTs send 9, 5 and 5 packets respectively (we have counted Data[10] twice); this is remarkably close to the ideal of reducing `cwnd` to 5 instantaneously.

The reason we cannot use `cwnd` directly in the formulation of Fast Recovery is that, until the lost Data[10] is acknowledged, the window is frozen at Data[10]-Data[19]. The original **RFC 2001** description of Fast

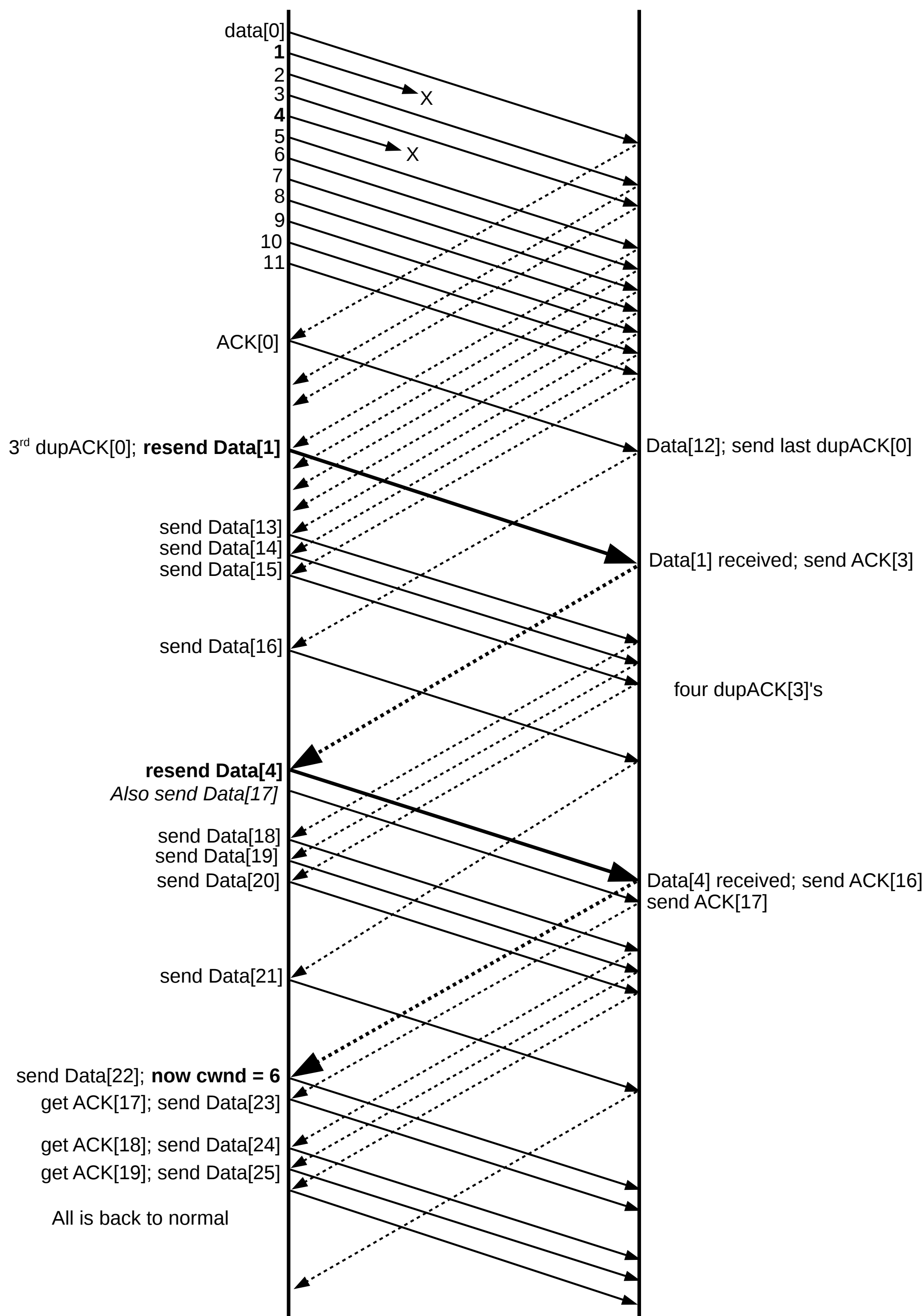
Recovery described retransmission in terms of cwnd **inflation** and **deflation**. Inflation would begin at the point the sender resumed transmitting new packets, at which point cwnd would be incremented for each dupACK; in the diagram above, cwnd would finish at 15. When the ACK[20] elicited by the lost packet finally arrived, the recovery phase would end and cwnd would immediately deflate to 5. For a diagram illustrating cwnd inflation and deflation, see *17.2.1 Running the Script*.

13.5 TCP NewReno

TCP NewReno, described in *[JH96]* and **RFC 2582** (currently **RFC 6582**), is a modest tweak to Fast Recovery which greatly improves handling of the case when two or more packets are lost in a windowful. It is considered to be a part of contemporary TCP Reno. If two data packets are lost and the first is retransmitted, the receiver will acknowledge data up to just before the second packet, and then continue sending dupACKs of this until the second lost packet is also retransmitted. These ACKs of data up to just before the second packet are sometimes called **partial ACKs**, because retransmission of the first lost packet did not result in an ACK of all the outstanding data. The NewReno mechanism uses these partial ACKs as evidence to retransmit later lost packets, and also to keep pacing the Fast Recovery process.

In the diagram below, packets 1 and 4 are lost in a window 0..11 of size 12. Initially the sender will get dupACK[0]'s; the first 11 ACKs (dashed lines from right to left) are ACK[0] and 10 dupACK[0]'s. When packet 1 is successfully retransmitted on receipt of the third dupACK[0], the receiver's response will be ACK[3] (the heavy dashed line). This is the first partial ACK (a full ACK would have been ACK[12]). On receipt of any partial ACK during the Fast Recovery process, TCP NewReno assumes that the immediately following data packet was lost and retransmits it immediately; the sender does not wait for three dupACKs because if the following data packet had not been lost, no instances of the partial ACK would ever have been generated, even if packet reordering had occurred.

The TCP NewReno sender response here is, in effect, to treat each partial ACK as a dupACK[0], except that the sender *also* retransmits the data packet that – based upon receipt of the partial ACK – it is able to infer is lost. NewReno continues pacing Fast Recovery by whatever ACKs arrive, whether they are the original dupACKs or later partial ACKs or dupACKs.



When the receiver's first ACK[3] arrives at the sender, NewReno infers that Data[4] was lost and resends it; this is the second heavy data line. No dupACK[3]'s need arrive; as mentioned above, the sender can infer from the single ACK[3] that Data[4] is lost. The sender also responds as if another dupACK[0] had arrived, and sends Data[17].

The arrival of ACK[3] signals a reduction in the EFS by 2: one for the inference that Data[4] was lost, and

one as if another dupACK[0] had arrived; the two transmissions in response (of Data[4] and Data[17]) bring EFS back to where it was. At the point when Data[16] is sent the actual (not estimated) flightsize is 5, not 6, because there is one less dupACK[0] due to the loss of Data[4]. However, once NewReno resends Data[4] and then sends Data[17], the actual flightsize is back up to 6.

There are four more dupACK[3]'s that arrive. NewReno keeps sending new data on receipt of each of these; these are Data[18] through Data[21].

The receiver's response to the retransmitted Data[4] is to send ACK[16]; this is the cumulative of all the data received up to that moment. At the point this ACK arrives back at the sender, it had just sent Data[21] in response to the fourth dupACK[3]; its response to ACK[16] is to send the next data packet, Data[22]. *The sender is now back to normal sliding windows*, with a `cwnd` of 6. Similarly, the Data[17] immediately following the retransmitted Data[4] elicits an ACK[17] (this is the first Data[N] to elicit an exactly matching ACK[N] since the losses began), and the corresponding response to the ACK[17] is to continue with Data[23].

As with the previous Fast Recovery example, we consider the number of packets sent per RTT; the diagram shows four RTTs as seen from the sender side.

RTT	First packet	Packets sent	count
first	Data[0]	Data[0]-Data[11]	12
second	Data[12]	Data[12]-Data[15], Data[1]	5
third	Data[16]	Data[16]-Data[20], Data[4]	6
fourth	Data[21]	Data[21]-Data[26]	6

Again, after the loss is detected we segue to the new `cwnd` of 6 with only a single missed packet (in the second RTT). NewReno is, however, only able to send one retransmitted packet per RTT.

Note that TCP Newreno, like TCPs Tahoe and Reno, is a **sender-side** innovation; the receiver does not have to do anything special. The next TCP flavor, SACK TCP, requires receiver-side modification.

13.6 Selective Acknowledgments (SACK)

A traditional TCP ACK is a cumulative acknowledgment of all data received up to that point. If Data[1002] is received but not Data[1001], then all the receiver can send is a duplicate ACK[1000]. This does indicate that *something* following Data[1001] made it through, but nothing more.

To provide greater specificity, TCP now provides a **Selective ACK** (SACK) option, implemented at the receiver. If this is available, the sender does not have to guess from dupACKs what has gotten through. The receiver can send an ACK that says:

- All packets up through 1000 have been received (the cumulative ACK)
- All packets up through 1050 have been received *except for* 1001, 1022, and 1035.

The second line is the SACK part. Almost all TCP implementations now support this.

Specifically, SACKs include the following information; the additional data beyond the cumulative ACK is included in a TCP Option field.

- The latest cumulative ACK
- The *three* most recent blocks of consecutive packets received

Thus, if we have lost 1001, 1022, 1035, and now 1051, and the highest received is 1060, the SACK might say:

- All packets up through 1000 have been received
- 1060-1052 have been received
- 1050-1036 have been received
- 1034-1023 have been received

From this the sender know 1001s and 1022 were not received, but nothing about the packets in between. However, if the sender has been paying close attention to the previous SACKs received, it likely already knows that all packets 1002 through 1021 have been received.

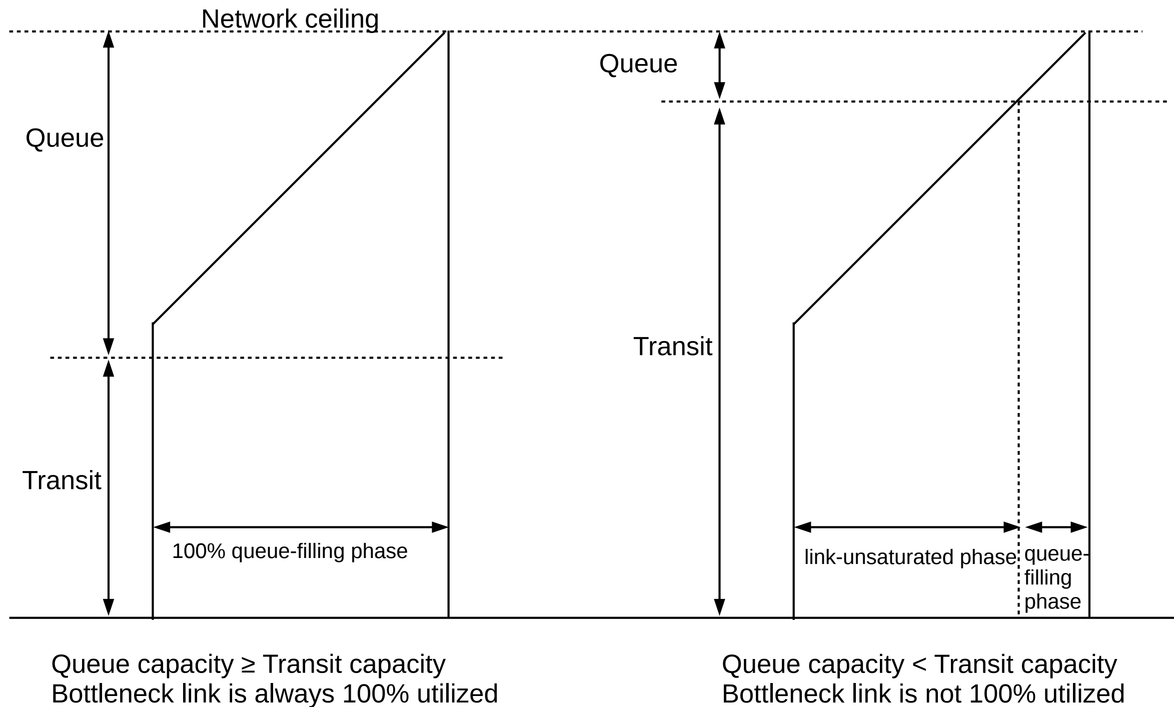
The term **SACK TCP** is typically used to mean that the receiving side supports selective ACKs, and the sending side is a straightforward modification of TCP Reno to take advantage of them.

In practice, selective ACKs provide at best a modest performance improvement in many situations; TCP NewReno does rather well, in moderate-loss environments. The paper *[FF96]* compares Tahoe, Reno, NewReno and SACK TCP, in situations involving from one to four packet losses in a single RTT. While Classic Reno performed poorly with two packet losses in a single RTT and extremely poorly with three losses, the three-loss NewReno and SACK TCP scenarios played out remarkably similarly. Only when connections experienced four losses in a single RTT did SACK TCP's performance start to pull slightly ahead of that of NewReno.

13.7 TCP and Bottleneck Link Utilization

Consider a TCP Reno sender with no competing traffic. As `cwnd` saws up and down, what happens to throughput? Do those halvings of `cwnd` result in at least a dip in throughput? The answer depends to some extent on the size of the queue ahead of the bottleneck link, relative to the transit capacity of the path. As was discussed in *6.3.2 RTT Calculations*, when `cwnd` is less than the transit capacity, the link is less than 100% utilized and the queue is empty. When `cwnd` is more than the transit capacity, the link is saturated (100% utilized) and the queue has about (`cwnd` – transit_capacity) packets in it.

The diagram below shows two TCP Reno teeth; in the first, the queue capacity exceeds the path transit capacity and in the second the queue capacity is a much smaller fraction of the total.



In the first diagram, the bottleneck link is always 100% utilized, even at the left edge of the teeth. In the second the interval between loss events (the left and right margins of the tooth) is divided into a **link-unsaturated** phase and a **queue-filling phase**. In the unsaturated phase, the bottleneck link utilization is less than 100% and the queue is empty; in the later phase, the link is saturated and the queue begins to fill.

Consider again the idealized network below, with an R–B bandwidth of 1 packet/ms.

We first consider the queue $\geqslant$ transit case. Assume that the total RTT_{noLoad} delay is 100 ms, mostly due to propagation delay; this makes the bandwidth×delay product 100 packets. The question for consideration is to what extent TCP Reno, once slow-start is over, sometimes leaves the R–B link idle.

The R–B link will be saturated at all times provided A always keeps 100 packets in transit, that is, we always have `cwnd` $\geqslant$ 100 (*6.3.2 RTT Calculations*). If $\texttt{cwnd}_{min}$ = 100, then $\texttt{cwnd}_{max}$ = 2×$\texttt{cwnd}_{min}$ = 200. For this to be the maximum, the queue capacity must be at least 99, so that the path can accommodate 199 packets without loss: 100 packets in transit plus 99 packets in the queue. In general, TCP Reno never leaves the bottleneck link idle as long as the queue capacity in front of that link is at least as large as the path round-trip transit capacity.

Now suppose instead that the queue size is 49, or about 50% of the transit capacity. Packet loss will occur when `cwnd` reaches 150, and so $\texttt{cwnd}_{min}$ = 75. Qualitatively this case is represented by the second diagram above, though the queue-to-network_ceiling proportion illustrated there is more like 1:8 than 1:3. There are now periods when the R–B link is idle. During RTT intervals when `cwnd`=75, throughput will be 75% of the maximum and the R–B link will be idle 25% of the time.

However, cwnd will be 75 just for the first RTT following the loss. After 25 RTTs, cwnd will be back up to 100, and the link will be saturated. So we have 25 RTTs with an average cwnd of 87.5 (=&75+100)/2), meaning the link is 87.5% saturated, followed by 50 RTTs where the link is 100% saturated. The long-term average here is 95.8% utilization of the bottleneck link. This is not bad at all, given that using 10% of the link bandwidth on packet headers is almost universally considered reasonable. Furthermore, at the point when cwnd drops after a loss to $cwnd_{min}$=75, the queue must have been full. It may take one or two RTTs for the queue to drain; during this time, link utilization will be even higher.

If most or all of the time the bottleneck link is saturated, as in the first diagram, it may help to consider the average queue size. Let the queue capacity be C_{queue} and the transit capacity be $C_{transit}$, with $C_{queue} > C_{transit}$. Then cwnd will vary from a maximum of $C_{queue}+C_{transit}$ to a minimum of what works out to be $(C_{queue}-C_{transit})/2 + C_{transit}$. We would expect an average queue size about halfway between these, less the $C_{transit}$ term: $3/4 \times C_{queue} - 1/4 \times C_{transit}$. If $C_{queue}=C_{transit}$, the expected average queue size should be about $C_{queue}/2$.

See exercises 12.0 and 12.5.

13.7.1 Bufferbloat

From the perspective of link utilization, the previous section suggests that router queues be larger rather than smaller. A queue capacity at least as large as transit capacity seems like an excellent choice. To configure a router this way, we first make an educated guess at the average RTT, and then multiply this by the output bandwidth to get the desired queue capacity. For an average RTT of 50 ms, a bandwidth of 1 Gbps leads to a queue capacity of about 6 MB, or 4000 packets of 1500 bytes each. If the numbers rise to 100 ms and 10 Gbps, queue capacity needs to be 125 MB. Still, RAM is cheap.

Unfortunately, while large queues are helpful when the traffic consists exclusively of bulk TCP transfers, they introduce proportionately large queuing delays. A bottleneck router with a queue size matching a flow's bandwidth×delay product will *double* the RTT for that flow, at points when the queue is full. And yet, because RAM is cheap, many residential routers have a queue capacity several times the average bandwidth × delay product, meaning that queuing delay potentially becomes much larger than propagation delay. The term **bufferbloat** is sometimes used to describe excessive queuing delay brought on by too much queue capacity. Large queues can also lead to delay *variability*, or jitter.

All these delay-related issues do not play well with interactive traffic or real-time traffic (**RFC 7567**). Often, they even have a serious impact on ordinary web-page loading as well. As a result, the second approach, with queue capacity less than transit capacity, has also been proposed; see, for example, *[WM05]* and *[EGMR05]*. In this case a tooth of a TCP Reno connection is divided into a large link-unsaturated phase and a small queue-filling phase.

The *need* for large buffers, if near-100% queue utilization is the goal, is to a large degree specific to the TCP Reno sawtooth. Some other TCP implementations (in particular TCP Vegas, *15.6 TCP Vegas*), do not overfill the queue. However, TCP Vegas does not compete well with TCP Reno, at least with traditional FIFO queuing (*14.1 A First Look At Queuing*) (but see *19.6.1 Fair Queuing and Bufferbloat*).

The worst case for TCP link utilization is if the queue size is close to zero. Using again a bandwidth×delay product 100 of packets, a zero-sized queue will mean that $cwnd_{max}$ will be 100 (or 101), and so $cwnd_{min}$ will be 50. Link utilization therefore ranges, over the lifetime of the tooth, from a low of 50/100 = 50% to a high of 100%; the average utilization is **75%**. While this is not ideal, and while some non-Reno TCP variants have attempted to improve this figure, 75% link utilization is not all that bad, and can be compared

with the 10% of the bandwidth consumed as packet headers (though that figure assumes 512 bytes of data per packet, which is low). (A literally zero-sized queue will not work at all well; one reason – though not the only one – is that TCP Reno sends a two-packet burst whenever `cwnd` is incremented.)

Traffic mix has a major influence on the appropriate queue size. For example, the analysis of the previous section assumed a single long-term TCP connection. The link-utilization situation improves with increasing numbers of TCP connections, at least if the losses are unsynchronized, because the halving of one connection's `cwnd` has a proportionately smaller impact on the total queue use. In *[AKM04]* it is shown that for a router with N TCP connections with unsynchronized losses, a queue size of $(\text{RTT}_{\text{average}} \times \text{bandwidth})/\sqrt{N}$ is sufficient to keep the link almost always saturated. Larger values of N here are typically associated with "core" (backbone) routers. The paper *[EGMR05]* proposes even smaller buffer capacities, on the order of the logarithm of the maximum window size. The argument makes two important assumptions, however: first, that we are willing to tolerate a link utilization somewhat less than 100% (though greater than 75%), and second, perhaps more importantly, that TCP is modified so as to spread out any packet bursts – even bursts of size two – over small intervals of time.

There are other problems created by too-small queues, even if we are willing to accept 75% link utilization. Internet traffic, not unlike city-bus traffic, tends to "bunch up"; queues serve as a way to keep these packet bunches from leading to unnecessary losses. For one example of unexpected traffic bunching, see *16.4.1.3 Transient queue peaks*. Increased traffic randomization helps reduce the need for very large queues, but may increase the bunching effect. Internet "core" routers see more highly randomized traffic than end-user or "edge" routers; queues in the latter are often the most difficult to configure.

We will return to the issue of link utilization in *16.2.6 Single-sender Throughput Experiments* and (for two senders) *16.3.10.2 Higher bandwidth and link utilization*, using the ns simulator to get experimental data. See also exercise 12.0.

Finally, the queue capacity does not necessarily have to remain static. We will return to this point in *14.8 Active Queue Management*. Furthermore, many queue-size problems ultimately spring from the fact that all traffic is being dumped into a single FIFO queue; we will look at alternative queuing strategies in *19 Queuing and Scheduling*. For a particular example related to bufferbloat, see *19.6.1 Fair Queuing and Bufferbloat*.

13.8 Single Packet Losses

Again assuming no competition on the bottleneck link, the TCP Reno additive-increase policy has a simple consequence: at the end of each tooth, only a single packet will be lost.

To see this, let A be the sender, R be the bottleneck router, and B be the receiver:

Let T be the bandwidth delay at R, so that packets leaving R are spaced at least time T apart. A will therefore transmit packets T time units apart, except for those times when `cwnd` has just been incremented and A sends a pair of packets back-to-back. Let us call the second packet of such a back-to-back pair the "extra" packet. To simplify the argument slightly, we will assume that the two packets of a pair arrive at R essentially simultaneously.

Only an extra packet can result in an increase in queue utilization; every other packet arrives after an interval T from the previous packet, giving R enough time to remove a packet from its queue.

A consequence of this is that cwnd will reach the sum of the transit capacity and the queue capacity without R dropping a packet. (This is not necessarily the case if a cwnd this large were sent as a single burst.)

Let C be this combined capacity, and assume cwnd has reached C. When A executes its next cwnd += 1 additive increase, it will as usual send a pair of back-to-back packets. The second of this pair – the extra – is doomed; it will be dropped when it reaches the bottleneck router.

At this point there are C = cwnd – 1 packets outstanding, all spaced at time intervals of T. Sliding windows will continue normally until the ACK of the packet just before the lost packet arrives back at A. After this point, A will receive only dupACKs. A has received C = cwnd–1 ACKs since the last increment to cwnd, but must receive C+1 = cwnd ACKs in order to increment cwnd again. This will not happen, as no more new ACKs will arrive until the lost packet is transmitted.

Following this, cwnd is reduced and the next sawtooth begins; the only packet that is lost is the "extra" packet of the previous flight.

See *16.2.3 Single Losses* for experimental confirmation, and exercise 14.0.

13.9 TCP Assumptions and Scalability

In the TCP design portrayed above, several embedded assumptions have been made. Perhaps the most important is that **every loss is treated as evidence of congestion**. As we shall see in the next chapter, this fails for high-bandwidth TCP (when rare random losses become significant); it also fails for TCP over wireless (either Wi-Fi or other), where lost packets are much more common than over Ethernet. See *14.9 The High-Bandwidth TCP Problem* and *14.10 The Lossy-Link TCP Problem*.

The TCP cwnd-increment strategy – to increment cwnd by 1 for each RTT – has some assumptions of scale. This mechanism works well for cross-continent RTT's on the order of 100 ms, and for cwnd in the low hundreds. But if cwnd = 2000, then it takes 100 RTTs – perhaps 20 seconds – for cwnd to grow 10%; linear increase becomes *proportionally* quite slow. Also, if the RTT is very long, the cwnd increase is slow. The absolute set-by-the-speed-of-light minimum RTT for geosynchronous-satellite Internet is 480 ms, and typical satellite-Internet RTTs are close to 1000 ms. Such long RTTs also lead to slow cwnd growth; furthermore, as we shall see below, such long RTTs mean that these TCP connections compete poorly with other connections. See *14.11 The Satellite-Link TCP Problem*.

Another implicit assumption is that if we have a lot of data to transfer, we will send all of it in one single connection rather than divide it among multiple connections. The web http protocol violates this routinely, though. With multiple short connections, cwnd may never properly converge to the steady state for any of them; TCP Reno does not support carrying over what has been learned about cwnd from one connection to the next. A related issue occurs when a connection alternates between relatively idle periods and full-on data transfer; most TCPs set cwnd=1 and return to slow start when sending resumes after an idle period.

Finally, TCP's Fast Retransmit assumes that routers do not significantly reorder packets.

13.10 TCP Parameters

In TCP Reno's Additive Increase, Multiplicative Decrease strategy, the increase increment is 1.0 and the decrease factor is 1/2. It is natural to ask if these values have some especial significance, or what are the consequences if they are changed.

Neither of these values plays much of a role in determining the average value of cwnd, at least in the short term; this is largely dictated by the path capacity, including the queue size of the bottleneck router. It seems clear that the exact value of the increase increment has no bearing on congestion; the per-RTT increase is too small to have a major effect here. The decrease factor of 1/2 *may* play a role in responding promptly to incipient congestion, in that it reduces cwnd sharply at the first sign of lost packets. However, as we shall see in *15.6 TCP Vegas*, TCP Vegas in its "normal" mode manages quite successfully with an Additive Decrease strategy, decrementing cwnd by 1 at the point it detects approaching congestion (to be sure, it detects this well before packet loss), and, by some measures, responds better to congestion than TCP Reno. In other words, not only is the exact value of the AIMD decrease factor not critical for congestion management, but multiplicative decrease itself is not mandatory.

There are two informal justifications in *[JK88]* for a decrease factor of 1/2. The first is in slow start: if at the Nth RTT it is found that cwnd = 2^N is too big, the sender falls back to cwnd/2 = 2^{N-1}, which is known to have worked without losses the previous RTT. However, a change here in the decrease policy might best be addressed with a concomitant change to slow start; alternatively, the reduction factor of 1/2 might be left still to apply to "unbounded" slow start, while a new factor of β might apply to threshold slow start.

The second justification for the reduction factor of 1/2 applies directly to the congestion avoidance phase; written in 1988, it is quite remarkable to the modern reader:

> If the connection is steady-state running and a packet is dropped, it's probably because a new connection started up and took some of your bandwidth.... [I]t's probable that there are now exactly two conversations sharing the bandwidth. I.e., you should reduce your window by half because the bandwidth available to you has been reduced by half. *[JK88]*, §D

Today, busy routers may have thousands of simultaneous connections. To be sure, Jacobson and Karels go on to state, "if there are more than two connections sharing the bandwidth, halving your window is conservative – and being conservative at high traffic intensities is probably wise". This advice remains apt today.

But while they do not play a large role in setting cwnd or in avoiding "congestive collapse", it turns out that these increase-increment and decrease-factor values of 1 and 1/2 respectively play a *great* role in **fairness**: making sure competing connections get the bandwidth allocation they "should" get. We will return to this in *14.3 TCP Fairness with Synchronized Losses*, and also *14.7 AIMD Revisited*.

13.11 Epilog

TCP Reno's core congestion algorithm is based on algorithms in Jacobson and Karel's 1988 paper *[JK88]*, now twenty-five years old, although NewReno and SACK have been almost universally added to the standard "Reno" implementation.

There are also broad changes in TCP usage patterns. Twenty years ago, the vast majority of all TCP traffic represented downloads from "major" servers. Today, over half of all Internet TCP traffic is peer-to-peer

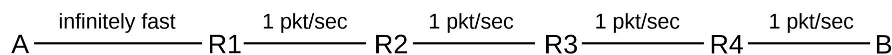
rather than server-to-client. The rise in online video streaming creates new demands for excellent TCP real-time performance.

In the next chapter we will examine the dynamic behavior of TCP Reno, focusing in particular on fairness between competing connections, and on other problems faced by TCP Reno senders. Then, in *15 Newer TCP Implementations*, we will survey some attempts to address these problems.

13.12 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercise 12.5 is distinct, for example, from exercises 12.0 and 13.0. Exercises marked with a ♢ have solutions or hints at 24.11 Solutions for TCP Reno.

1.0. Consider the following network, with each link other than the first having a bandwidth delay of 1 packet/second. Assume ACKs travel instantly from B to R (and thus to A). Assume there are no propagation delays, so the RTT_{noLoad} is 4; the bandwidth×RTT product is then 4 packets. If A uses sliding windows with a window size of 6, the queue at R1 will eventually have size 2.



Suppose A uses **threshold** slow start (*13.2.2 Threshold Slow Start*) with `ssthresh` = 6, and with `cwnd` initially 1. Complete the table below until two rows after `cwnd` = 6; for these final two rows, `cwnd` has reached `ssthresh` and so A will send only one new packet for each ACK received. How big will the queue at R1 grow?

T	A sends	R1 queues	R1 sends	B receives/ACKs	cwnd
0	1		1		1
1					
2					
3					
4	2,3	3	2	1	2
5			3		2
6					
7					
8	4,5	5	4	2	3

Note that if, instead of using slow start, A simply sends the initial windowful of 6 packets all at once, then the queue at R1 will initially hold 6-1 = 5 packets.

2.0. Consider the following network from *13.2.3 Slow-Start Multiple Drop Example*, with links labeled with bandwidths in packets/ms. Assume ACKs travel instantly from B to R (and thus to A).

A begins sending to B using unbounded slow start, beginning with Data[1] at T=0. Write out a table of packet transmissions and deliveries assuming R's queue size is 5 (not counting the packet currently being forwarded). Stop with the arrival at A of the first dupACK triggered by the arrival at B of the packet that followed the first packet that was dropped by R. No retransmissions will occur by then.

3.0. Consider the network from exercise 2.0 above. A again begins sending to B using unbounded slow start, but this time R's queue size is 2, not counting the packet currently being forwarded. Make a table showing all packet transmissions by A, all packet drops by R, and other columns as are useful. Assume no **re**transmission mechanism is used at all (no timeouts, no fast retransmit), and that A sends new data only when it receives new ACKs (dupACKs, in other words, do not trigger new data transmissions). With these assumptions, new data transmissions will eventually cease; continue the table until all transmitted data packets are received by B.

4.0. Suppose a connection starts with `cwnd`=1 and increments `cwnd` by 1 each RTT with no loss, and sets `cwnd` to `cwnd`/2, rounding down, on each RTT with at least one loss. Lost packets are **not retransmitted**, and propagation delays dominate so each windowful is sent more or less together. Packets 5, 13, 14, 23 and 30 are lost. What is the window size each RTT, up until the first 40 packets are sent? What packets are sent each RTT? Hint: in the first RTT, Data[1] is sent. There is no loss, so in the second RTT `cwnd` = 2 and Data[2] and Data[3] are sent.

5.0. Suppose TCP Reno is used to transfer a large file over a path with bandwidth high enough that, during slow start, `cwnd` can be treated as doubling each RTT as in *13.2 Slow Start*. Assume the receiver places no limits on window size.

(a). How many RTTs will it take for the window size to first reach ~8,000 packets, assuming unbounded slow start is used and there are no packet losses?

(b). Approximately how many packets will have been sent and acknowledged by that point?

(c). Now assume the bandwidth is 100 packets/ms and the RTT is 80 ms, making the bandwidth×delay product 8,000 packets. What fraction of the total bandwidth will have been used by the connection up to the point where the window size reaches 8000? Hint: the total bandwidth is 8,000 packets per RTT.

6.0. (a) Repeat the diagram in *13.4 TCP Reno and Fast Recovery*, done there with `cwnd`=10, for a window size of 8. Assume as before that the lost packet is Data[10]. There will be seven dupACK[9]'s, which it may be convenient to tag as dupACK[9]/11 through dupACK[9]/17. Be sure to indicate clearly when sending resumes.

(b). Suppose you try to do this with a window size of 6. Is this window size big enough for Fast Recovery still to work? If so, at what dupACK[9]/N does new data transmission begin? If not, what goes wrong?

7.0. Suppose the window size is 100, and Data[1001] is lost. There will be 99 dupACK[1000]'s sent, which we may denote as dupACK[1000]/1002 through dupACK[1000]/1100. TCP Reno is used.

(a). At which dupACK[1000]/N does the sender start sending new data?

(b). When the retransmitted data[1001] arrives at the receiver, what ACK is sent in response?

(c). When the acknowledgment in (b) arrives back at the sender, what data packet is sent?

Hint: express EFS in terms of dupACK[1000]/N, for N⩾1004. The third dupACK is dupACK[1000]/1004; what is EFS at that point after retransmission of Data[1001]?

8.0. Suppose the window size is 40, and Data[1001] is lost. Packet 1000 will be ACKed normally. Packets 1001-1040 will be sent, and 1002-1040 will each trigger a duplicate ACK[1000].

(a). What actual data packets trigger the first three dupACKs? (The first ACK[1000] is triggered by Data[1000]; don't count this one as a duplicate.)
(b). After the third dupACK[1000] has been received and the lost data[1001] has been retransmitted, how many packets/ACKs should the sender estimate as in flight?

When the retransmitted Data[1001] arrives at the receiver, ACK[1040] will be sent back.

(c). What is the first Data[N] sent for which the response is ACK[N], for N>1000?
(d). What is the first N for which Data[N+20] is sent in response to ACK[N] (this represents the point when the connection is back to normal sliding windows, with a window size of 20)?

9.0. Suppose slow-start is modified so that, on each arriving ACK, three new packets are sent rather than two; `cwnd` will now triple after each RTT.

(a). For each arriving ACK, by how much must `cwnd` now be incremented?
(b). Suppose a path has mostly propagation delay. Progressively larger windowfuls are sent, until a `cwnd` is reached where a packet loss occurs. What window size can the sender be reasonably sure *does* work, based on earlier experience?

10.0. Suppose in the example of *13.5 TCP NewReno*, Data[4] had *not* been lost.

(a). When Data[1] is received, what ACK would be sent in response?
(b). At what point in the diagram is the sender able to resume ordinary sliding windows with `cwnd` = 6?

11.0. Suppose in the example of *13.5 TCP NewReno*, Data[1] and Data[2] had been lost, but not Data[4].

(a). The third dupACK[0] is sent in response to what Data[N]?
(b). When the retransmitted Data[1] reaches the receiver, ACK[1] is the response. When this ACK[1] reaches the sender, which Data packets are sent in response?

12.0. Suppose two TCP connections have the same RTT and share a bottleneck link, on which there is no

other traffic. The size of the bottleneck queue is negligible when compared to the bandwidth × RTT_{noLoad} product. Loss events occur at regular intervals, and are completely synchronized. Show that the two connections together will use 75% of the total bottleneck-link capacity, as in *13.7 TCP and Bottleneck Link Utilization* (there done for a single connection).

See also Exercise 18.0 of the next chapter.

12.5. In *13.7 TCP and Bottleneck Link Utilization* we showed that, if the bottleneck router queue capacity was 50% of a TCP Reno connection's transit capacity, and there was no other traffic, then the bottleneck-link utilization would be 95.8%.

(a). Suppose the queue capacity is 1/3 of the transit capacity. Show the bottleneck link utilization is 11/12, or 91.7%. Draw a diagram of the tooth, and find the relative lengths of the link-unsaturated and queue-filling phases. You may round off $\texttt{cwnd}_{max}$ to 4/3 the transit capacity (the value of `cwnd` just before the packet loss; the exact value of $\texttt{cwnd}_{max}$ is higher by 1).
(b).◇ Derive a formula for the link utilization in terms of the ratio **f**<1 of queue capacity to transit capacity. Make the same simplifying assumption as in part (a).

13.0. In *13.2.1 Per-ACK Responses* we stated that the per-ACK response of a TCP sender was to increment `cwnd` as follows:

> `cwnd` = `cwnd` + 1/`cwnd`

(a). What is the corresponding formulation if the window size is in fact measured in bytes rather than packets? Let `SMSS` denote the sender's maximum segment size, and let `bwnd` = `SMSS`×`cwnd` denote the congestion window as measured in bytes. Hint: solve this last equation for `cwnd` and plug the result in above.
(b). What is the appropriate formulation of `cwnd` = `cwnd` + 1/`cwnd` if delayed ACKs are used (*12.15 TCP Delayed ACKs*) and we still want `cwnd` to be incremented by 1 for each windowful? Assume we are back to measuring `cwnd` in packets.

14.0. In *13.8 Single Packet Losses* we simplified the argument slightly by assuming that when A sent a pair of packets, they arrived at R "essentially simultaneously".

Give a scenario in which it is not the "extra" packet (the second of the pair) that is lost, but the packet that follows it. Hint: see *16.3.4.1 Single-sender phase effects*.

In this chapter we introduce, first and foremost, the possibility that there are other TCP connections out there competing with us for throughput. In *6.3 Linear Bottlenecks* (and in *13.7 TCP and Bottleneck Link Utilization*) we looked at the performance of TCP through an *uncontested* bottleneck; now we allow for competition.

We also look more carefully at the long-term behavior of TCP Reno (and Reno-like) connections, as the value of `cwnd` increases and decreases according to the TCP sawtooth. In particular we analyze the average `cwnd`; recall that the average `cwnd` divided by the RTT is the connection's average throughput (we momentarily ignore here the fact that RTT is not constant, but the error this introduces is usually small).

A few of the ideas presented here apply as well to non-Reno connections as well. Some non-Reno TCP alternatives are presented in the following chapter; the section on TCP Friendliness below addresses how to extend TCP Reno's competitive behavior even to UDP.

We also consider some router-based mechanisms such as RED and ECN that take advantage of TCP Reno's behavior to provide better overall performance.

The chapter closes with a summary of the central real-world performance problems faced by TCP today; this then serves to introduce the proposed TCP fixes in the following chapter.

14.1 A First Look At Queuing

In what order do we transmit the packets in a router's outbound-interface queue? The conventional answer is in the order of arrival; technically, this is **FIFO** (First-In, First-Out) queuing. What happens to a packet that arrives at a router whose queue for the desired outbound interface is full? The conventional answer is that it is dropped; technically, this is known as **tail-drop**.

While **FIFO tail-drop** remains very important, there are alternatives. In an admittedly entirely different context (the IPv6 equivalent of ARP), **RFC 4681** states, "When a queue overflows, the new arrival SHOULD replace the oldest entry." This might be called "head drop"; it is not used for *router* queues.

Queuing Theory

While there are lots of queues in this chapter, we avoid the language of traditional queuing theory. That's because queues created through sliding windows are highly deterministic in terms of both arrivals and departures (so-called "D/D/1" queues). From a queuing-theory perspective, interesting queues tend to have random (*eg* Poisson) arrival or service times, or both; *eg* "M/M/1" queues. The advantage, for us, of sliding-windows queues is that they are a good deal more tractable than the general case. When analyzing independent *messages*, however, M/M/1 queuing is much more important; see for example *[LK78]*.

An alternative drop-policy mechanism that *has* been considered for router queues is **random drop**. Under this policy, if a packet arrives but the destination queue is full, with N packets waiting, then one of the N+1 packets in all – the N waiting plus the new arrival – is chosen at random for dropping. The most recent

arrival has thus a very good chance of gaining an initial place in the queue, but also a reasonable chance of being dropped later on.

While random drop is seldom if ever put to production use its original form, it does resolve a peculiar synchronization problem related to TCP's natural periodicity that can lead to starvation for one connection. This situation – known as **phase effects** – will be revisited in *16.3.4 Phase Effects*. Mathematically, random-drop queuing is sometimes more tractable than tail-drop because a packet's loss probability has little dependence on arrival-time race conditions with other packets.

14.1.1 Priority Queuing

A quite different alternative to FIFO is **priority queuing**. We will consider this in more detail in *19.3 Priority Queuing*, but the basic idea is straightforward: whenever the router is ready to send the next packet, it looks first to see if it has any higher-priority packets to send; lower-priority packets are sent only when there is no waiting higher-priority traffic. This can, of course, lead to complete starvation for the lower-priority traffic, but often there are bandwidth constraints on the higher-priority traffic (*eg* that it amounts to less than 10% of the total available bandwidth) such that starvation does not occur.

In an environment of mixed real-time and bulk traffic, it is natural to use priority queuing to give the real-time traffic priority service, by assignment of such traffic to the higher-priority queue. This works quite well as long as, say, the real-time traffic is less than some fixed fraction of the total; we will return to this in *20 Quality of Service*.

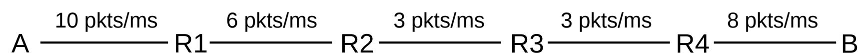
14.2 Bottleneck Links with Competition

So far we have been ignoring the fact that there are other TCP connections out there. A single connection in isolation needs not to overrun its bottleneck router and drop packets, at least not too often. However, once there are other connections present, then each individual TCP connection also needs to consider how to maximize its share of the aggregate bandwidth.

Consider a simple network path, with bandwidths shown in packets/ms. The minimum bandwidth, or **path bandwidth**, is 3 packets/ms.

14.2.1 Example 1: linear bottleneck

Below is the example we considered in *6.3 Linear Bottlenecks*; bandwidths are shown in packets/ms.

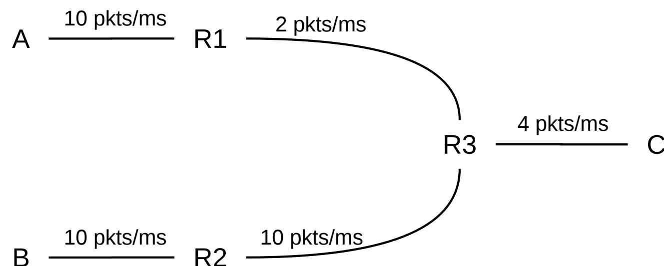


The bottleneck link for A→B traffic is at R2, and the queue will form at R2's outbound interface.

We claimed earlier that if the sender uses sliding windows with a fixed window size, then the network will converge to a steady state in relatively short order. This is also true if multiple senders are involved; however, a mathematical proof of convergence may be more difficult.

14.2.2 Example 2: router competition

The bottleneck-link concept is a useful one for understanding congestion due to a single connection. However, if there are multiple senders in **competition** for a link, the situation is more complicated. Consider the following diagram, in which links are labeled with bandwidths in packets/ms:



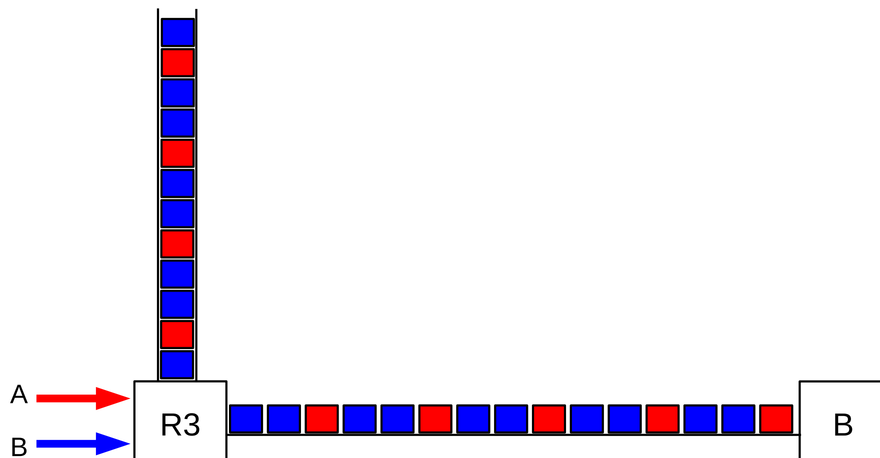
For a moment, assume R3 uses *priority* queuing, with the B⟶C path given priority over A⟶C. If B's flow to C is fixed at 3 packets/ms, then A's share of the R3–C link will be 1 packet/ms, and A's bottleneck will be at R3. However, if B's total flow rate drops to 1 packet/ms, then the R3–C link will have 3 packets/ms available, and the bottleneck for the A–C path will become the 2 packet/ms R1–R3 link.

Now let us switch to the more-realistic *FIFO* queuing at R3. If B's flow is 3 packets/ms and A's is 1 packet/ms, then the R3–C link will be saturated, but just barely: if each connection sticks to these rates, no queue will develop at R3. However, it is no longer accurate to describe the 1 packet/ms as A's *share*: if A wishes to send more, it will begin to compete with B. At first, the queue at R3 will grow; eventually, it is quite possible that B's total flow rate might drop because *B is losing to A in the competition for R3's queue*. This latter effect is very real.

In general, if two connections share a bottleneck link, they are competing for the bandwidth of that link. That bandwidth share, however, is *precisely dictated by the queue share as of a short while before*. R3's fixed rate of 4 packets/ms means one packet every 250 µs. If R3 has a queue of 100 packets, and in that queue there are 37 packets from A and 63 packets from B, then over the next 25 ms (= 100 × 250 µs) R3's traffic to C will consist of those 37 packets from A and the 63 from B. Thus the competition between A and B for R3–C bandwidth is *first fought as a competition for R3's queue space*. This is important enough to state as as rule:

> **Queue-Competition Rule**: in the steady state, if a connection utilizes fraction $\alpha \leqslant 1$ of a FIFO router's queue, then that connection has a share of α of the router's total outbound bandwidth.

Below is a picture of R3's queue and outbound link; the queue contains four packets from A and eight from B. The link, too, contains packets in this same ratio; presumably packets from B are consistently arriving twice as fast as packets from A.



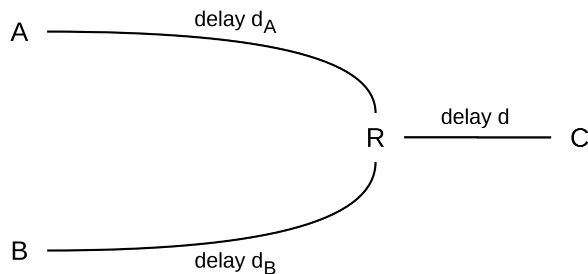
In the steady state here, A and B will use four and eight packets, respectively, of R3's queue capacity. As acknowledgments return, each sender will replenish the queue accordingly. However, it is not in A's long-term interest to settle for a queue utilization at R3 of four packets; A may want to take steps that will lead in this setting to a gradual increase of its queue share.

Although we started the discussion above with fixed packet-sending **rates** for A and B, in general this leads to instability. If A and B's combined rates add up to more than 4 packets/ms, R3's queue will grow without bound. It is much better to have A and B use **sliding windows**, and give them each fixed window sizes; in this case, as we shall see, a stable equilibrium is soon reached. Any combination of window sizes is legal regardless of the available bandwidth; the queue utilization (and, if necessary, the loss rate) will vary as necessary to adapt to the actual bandwidth.

If there are several competing flows, then a given connection may have multiple bottlenecks, in the sense that there are several routers on the path experiencing queue buildups. In the steady state, however, we can still identify the link (or first link) with minimum bandwidth; we can call this link the bottleneck. Note that the bottleneck link in this sense can change with the sender's winsize and with competing traffic.

14.2.3 Example 3: competition and queue utilization

In the next diagram, the bottleneck R–C link has a normalized bandwidth of 1 packet per ms (or, more abstractly, one packet per unit time). The bandwidths of the A–R and B–R links do not matter, except they are greater than 1 packet per ms. Each link is labeled with the **propagation delay**, measured in the same time unit as the bandwidth; the delay thus represents the number of packets the link can be transporting at the same time, if sent at the bottleneck rate.



The network layout here, with the shared R–C link as the bottleneck, is sometimes known as the **singlebell** topology. A perhaps-more-common alternative is the **dumbbell** topology of *14.3 TCP Fairness with Synchronized Losses*, though the two are equivalent for our purposes.

Suppose A and B each send to C using sliding windows, each with **fixed** values of winsize w_A and w_B. Suppose further that these winsize values are large enough to saturate the R–C link. *How big will the queue be at R?* And how will the bandwidth divide between the A⟶C and B⟶C flows?

For the two-competing-connections example above, assume we have reached the steady state. Let α denote the fraction of the bandwidth that the A⟶C connection receives, and let β = 1-α denote the fraction that the B⟶C connection gets; because of our normalization choice for the R–C bandwidth, α and β also represent respective throughputs. From the Queue-Competition Rule above, these bandwidth proportions must agree with the queue proportions; if Q denotes the combined queue utilization of both connections, then that queue will have about αQ packets from the A⟶C flow and about βQ packets from the B⟶C flow.

We worked out the queue usage precisely in *6.3.2 RTT Calculations* for a *single* flow; we derived there the following:

$$\text{queue_usage} = \text{winsize} - \text{throughput} \times \text{RTT}_{\text{noLoad}}$$

where we have here used "throughput" instead of "bandwidth" to emphasize that this is the dynamic share rather than the physical transmission capacity.

This equation remains true for each separate flow in the present case, where the $\text{RTT}_{\text{noLoad}}$ for the A⟶C connection is $2(d_A+d)$ (the factor of 2 is to account for the round-trip) and the $\text{RTT}_{\text{noLoad}}$ for the B⟶C connection is $2(d_B+d)$. We thus have

$$\alpha Q = w_A - 2\alpha(d_A+d)$$

$$\beta Q = w_B - 2\beta(d_B+d)$$

or, alternatively,

$$\alpha[Q + 2d + 2d_A] = w_A$$

$$\beta[Q + 2d + 2d_B] = w_B$$

If we add the first pair of equations above, we can obtain the combined queue utilization:

$$Q = w_A + w_B - 2d - 2(\alpha d_A+\beta d_B)$$

The last term here, $2(\alpha d_A+\beta d_B)$, represents the number of A's packets in flight on the A–R link plus the number of B's packets in flight on the B–R link.

We can solve these equations exactly for α, β and Q in terms of the known quantities, but the algebraic solution is not particularly illuminating. Instead, we examine a few more-tractable special cases.

14.2.3.1 The equal-delays case

We consider first the special case of **equal delays**: $d_A = d_B = d'$. In this case the term $(\alpha d_A+\beta d_B)$ simplifies to d', and thus we have $Q = w_A + w_B - 2d - 2d'$. Furthermore, if we divide corresponding sides of the second pair of equations above, we get $\alpha/\beta = w_A/w_B$; that is, the bandwidth (and thus the queue utilization) divides in exact accordance to the window-size proportions.

If, however, d_A is larger than d_B, then a greater fraction of the A⟶C packets will be in transit, and so fewer will be in the queue at R, and so α will be somewhat smaller and β somewhat larger.

14.2.3.2 The equal-windows case

If we assume equal winsize values instead, $w_A = w_B = w$, then we get

$$\alpha/\beta = [Q + 2d + 2d_B] / [Q + 2d + 2d_A]$$

The bandwidth ratio here is biased against the larger of d_A or d_B. That is, if $d_A > d_B$, then more of A's packets will be in transit, and thus fewer will be in R's queue, and so A will have a smaller fraction of the the bandwidth. This bias is, however, not quite proportional: if we assume d_A is double d_B and $d_B = d = Q/2$, then $\alpha/\beta = 3/4$, and A gets 3/7 of the bandwidth to B's 4/7.

Still assuming $w_A = w_B = w$, let us decrease w to the point where the link is just saturated, but Q=0. At this point $\alpha/\beta = [d+d_B]/[d +d_A]$; that is, bandwidth divides according to the respective RTT_{noLoad} values. As w rises, additional queue capacity is used and α/β will move closer to 1.

14.2.3.3 The fixed-w_B case

Finally, let us consider what happens if w_B is **fixed** at a large-enough value to create a queue at R from the B–C traffic alone, while w_A then increases from zero to a point much larger than w_B. Denote the number of B's packets in R's queue by Q_B; with $w_A = 0$ we have β=1 and $Q = Q_B = w_B - 2(d_B+d)$ = throughput × $(RTT - RTT_{noLoad})$.

As w_A begins to increase from zero, the competition will decrease B's throughput. We have $\alpha = w_A/[Q+2d+2d_A]$; **small** changes in w_A will not lead to much change in Q, and even less in $Q+2d+2d_A$, and so α will initially be approximately proportional to w_A.

For B's part, increased competition from A (increased w_A) will always decrease B's share of the bottleneck R–C link; this link is saturated and every packet of A's in transit there must take away one slot on that link for a packet of B's. This in turn means that B's bandwidth β must decrease as w_A rises. As B's bandwidth decreases, $Q_B = \beta Q = w_B - 2\beta(d_B+d)$ must increase; another way to put this is as the transit capacity falls, the queue utilization rises. For $Q_B = \beta Q$ to increase while β decreases, Q must be increasing faster than β is decreasing.

Finally, we can conclude that as w_A gets large and $\beta\to 0$, the limiting value for B's queue utilization Q_B at R will be the entire windowful w_B, up from its starting value (when w_A=0) of $w_B - 2(d_B+d)$. If d_B+d had been

small relative to w_B, then Q_B's increase will be modest, and it may be appropriate to consider Q_B relatively constant.

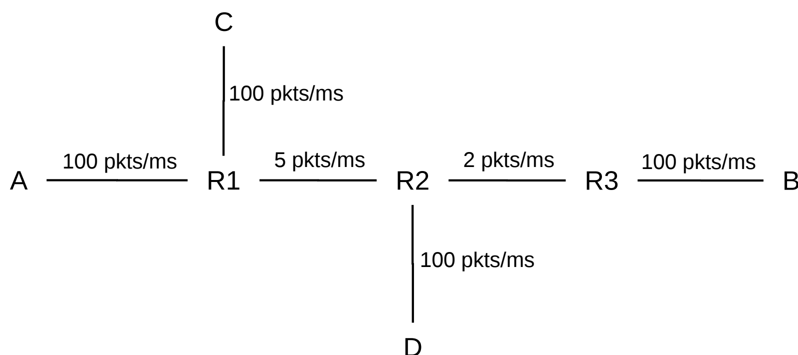
We remark again that the formulas here are based on the assumption that the bottleneck bandwidth is one packet per unit time; see exercise 0.5 for the necessary adjustments for conventional bandwidth measurements.

14.2.3.4 The iterative solution

Given d, d_A, d_B, w_A and w_B, one way to solve for α, β and Q is to proceed **iteratively**. Suppose an initial $\langle\alpha,\beta\rangle$ is given, as the respective fractions of packets in the queue at R. Over the next period of time, α and β must (by the Queue Rule) become the bandwidth ratios. If the A–C connection has bandwidth α (recall that the R–C connection has bandwidth 1.0, in packets per unit time, so a bandwidth fraction of α means an actual bandwidth of α), then the number of packets in bidirectional transit will be $2\alpha(d_A+d)$, and so the number of A–C packets in R's queue will be $Q_A = w_A - 2\alpha(d_A+d)$; similarly for Q_B. At that point we will have $\alpha_{new} = Q_A/(Q_A+Q_B)$. Starting with an appropriate guess for α and iterating $\alpha \rightarrow \alpha_{new}$ a few times, if the sequence converges then it will converge to the steady-state solution. Convergence is not guaranteed, however, and is dependent on the initial guess for α. One guess that often leads to convergence is $w_A/(w_A+w_B)$.

14.2.4 Example 4: cross traffic and RTT variation

In the following diagram, let us consider what happens to the A–B traffic when the C⟶D link ramps up. Bandwidths shown are expressed as packets/ms and all queues are FIFO. (Because the bandwidth is not equal to 1.0, we cannot apply the formulas of the previous section directly.) We will assume that propagation delays are small enough that only an inconsequential number of packets from C to D can be simultaneously in transit at the bottleneck rate of 5 packets/ms. All senders will use sliding windows.



Let us suppose the A–B link is idle, and the C⟶D connection begins sending with a window size chosen so as to create a queue of 30 of C's packets at R1 (if propagation delays are such that two packets can be in transit each direction, we would achieve this with winsize=34).

Now imagine A begins sending. If A sends a single packet, is not shut out even though the R1–R2 link is 100% busy. A's packet will simply have to wait at R1 behind the 30 packets from C; the waiting time in the

queue will be 30 packets ÷ (5 packets/ms) = 6 ms. If we change the winsize of the C⟶D connection, the delay for A's packets will be directly proportional to the number of C's packets in R1's queue.

To most intents and purposes, the C⟶D flow here has increased the RTT of the A⟶B flow by 6 ms. As long as A's contribution to R1's queue is small relative to C's, the delay at R1 for A's packets looks more like propagation delay than bandwidth delay, because if A sends two back-to-back packets they will likely be enqueued consecutively at R1 and thus be subject to a single 6 ms queuing delay. By varying the C⟶D window size, we can, within limits, increase or decrease the RTT for the A⟶B flow.

Let us return to the fixed C⟶D window size – denoted w_C – chosen to yield a queue of 30 of C's packets at R1. As A increases its own window size from, say, 1 to 5, the C⟶D throughput will decrease slightly, but C's contribution to R1's queue will remain dominant.

As in the argument at the end of *14.2.3.3 The fixed-wB case*, small propagation delays mean that w_C will not be much larger than 30. As w_A climbs from zero to infinity, C's contribution to R1's queue rises from 30 to at most w_C, and so the 6ms delay for A⟶B packets remains relatively constant even as A's winsize rises to the point that A's contribution to R1's queue far outweighed C's. (As we will argue in the next paragraphs, this can actually happen only if the R2–R3 bandwidth is increased). Each packet from A arriving at R1 will, on average, face 30 or so of C's packets ahead of it, along with anywhere from many fewer to many more of A's packets.

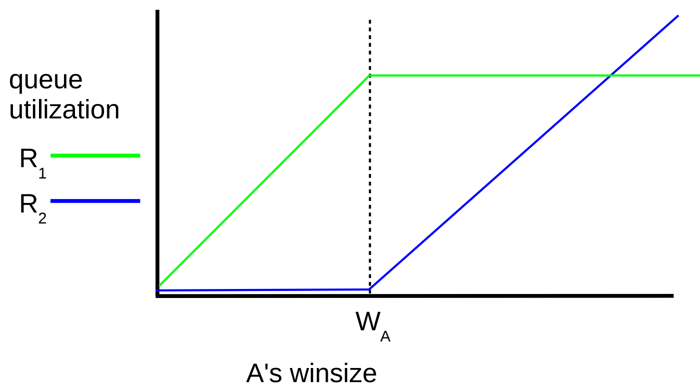
If A's window size is 1, its one packet at a time will wait 6 ms in the queue at R1. If A's window size is greater than 1 but remains small, so that A contributes only a small proportion of R1's queue, then A's packets will wait only at R1. Initially, as A's winsize increases, the queue at R1 grows but all other queues remain empty.

However, if A's winsize grows large enough that its packets consume 40% of R1's queue in the steady state, then this situation changes. At the point when A has 40% of R1's queue, by the Queue Competition Rule it will also have a 40% share of the R1–R2 link's bandwidth, that is, 40% × 5 = 2 packets/ms. Because the R2–R3 link has a bandwidth of 2 packets/ms, *the A–B throughput can never grow beyond this*. If the C–D contribution to R1's queue is held constant at 30 packets, then this point is reached when A's contribution to R1's queue is 20 packets.

Because A's proportional contribution to R1's queue cannot increase further, any additional increase to A's winsize must result in those packets now being enqueued at R2.

We have now reached a situation where A's packets are queuing up at both R1 and at R2, contrary to the single-sender principle that packets can queue at only one router. Note, however, that for any fixed value of A's winsize, a small-enough increase in A's winsize will result in either that increase going entirely to R1's queue or entirely to R2's queue. Specifically, if w_A represents A's winsize at the point when A has 40% of R1's queue (a little above 20 packets if propagation delays are small), then for winsize < w_A any queue growth will be at R1 while for winsize > w_A any queue growth will be at R2. In a sense the bottleneck link "switches" from R1–R2 to R2–R3 at the point winsize = w_A.

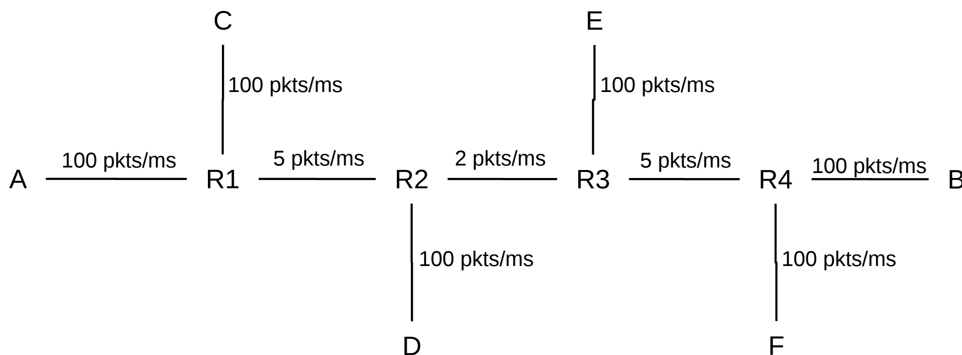
In the graph below, A's contribution to R1's queue is plotted in green and A's contribution to R2's queue is in blue. It may be instructive to compare this graph with the third graph in *6.3.3 Graphs at the Congestion Knee*, which illustrates a single connection with a single bottleneck.



In Exercise 5.0 we consider some minor changes needed if propagation delay is *not* inconsequential.

14.2.5 Example 5: dynamic bottlenecks

The next example has two links offering potential competition to the A⟶B flow: C⟶D and E⟶F. Either of these could send traffic so as to throttle (or at least compete with) the A⟶B traffic. Either of these could choose a window size so as to build up a persistent queue at R1 or R3; a persistent queue of 20 packets would mean that A⟶B traffic would wait 4 ms in the queue.



Despite situations like this, we will usually use the term "bottleneck link" as if it were a precisely defined concept. In Examples 2, 3 and 4 above, a better term might be "competitive link"; for Example 5 we perhaps should say "competitive link**s**.

14.2.6 Packet Pairs

One approach for a sender to attempt to measure the physical bandwidth of the bottleneck link is the **packet-pairs** technique: the sender repeatedly sends a pair of packets P1 and P2 to the receiver, one right after the other. The receiver records the time difference between the arrivals.

Sooner or later, we would expect that P1 and P2 would arrive consecutively at the bottleneck router R, and be put into the queue next to each other. They would then be sent one right after the other on the bottleneck

link; if T is the time difference in arrival at the far end of the link, the physical bandwidth is size(P1)/T. At least some of the time, the packets will remain spaced by time T for the rest of their journey.

The theory is that the receiver can measure the different arrival-time differences for the different packet pairs, and look for the *minimum* time difference. Often, this will be the time difference introduced by the bandwidth delay on the bottleneck link, as in the previous paragraph, and so the ultimate receiver will be able to infer that the bottleneck physical bandwidth is size(P1)/T.

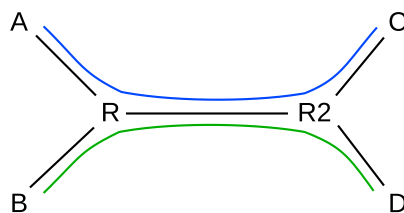
Two things can mar this analysis. First, packets may be reordered; P2 might arrive before P1. Second, P1 and P2 can arrive together at the bottleneck router and be sent consecutively, but then, later in the network, the two packets can arrive at a second router R2 with a (transient) queue large enough that P2 arrives while P1 is in R2's queue. If P1 and P2 are consecutive in R2's queue, then the ultimate arrival-time difference is likely to reflect R2's (higher) outbound bandwidth rather than R's.

Additional analysis of the problems with the packet-pair technique can be found in [VP97], along with a proposal for an improved technique known as *packet bunch mode*.

14.3 TCP Fairness with Synchronized Losses

This brings us to the question of just what *is* a "fair" division of bandwidth. A starting place is to assume that "fair" means "equal", though, as we shall see below, the question does not end there.

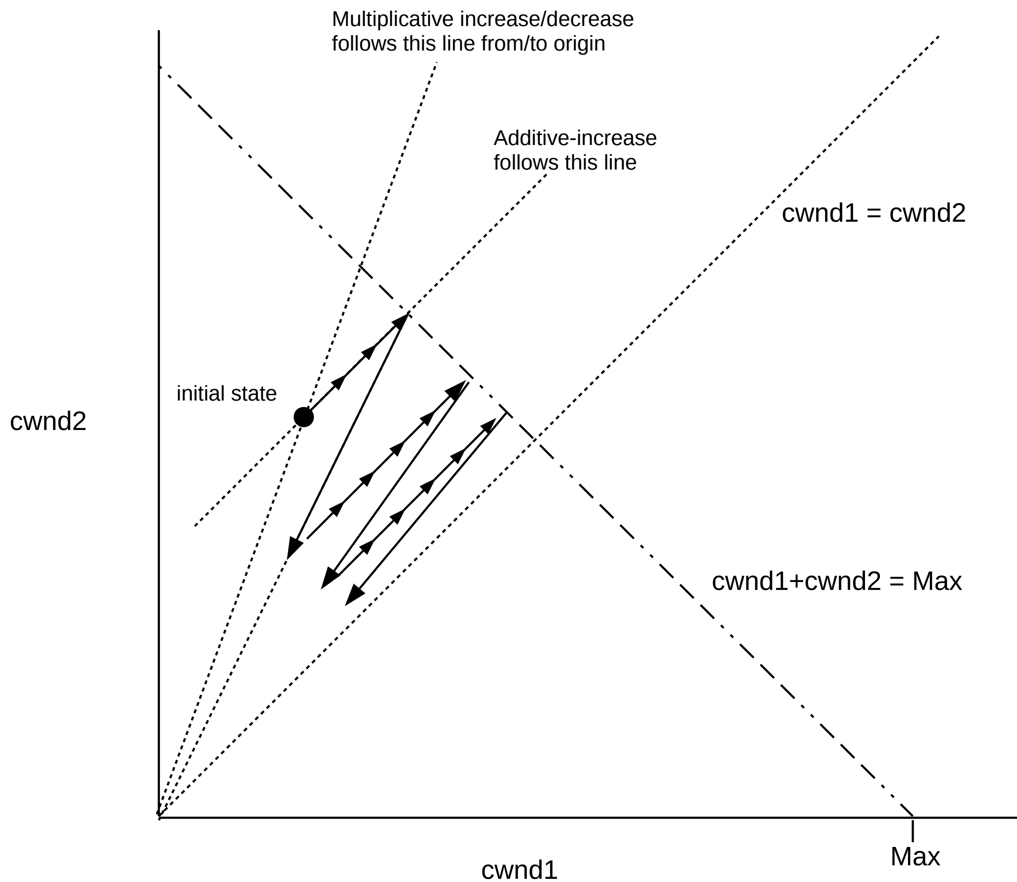
For the moment, consider again two competing TCP connections: Connection 1 (in blue) from A to C and Connection 2 (in green) from B to D, through the same bottleneck router R, *and with the same RTT*. The router R will use tail-drop queuing.



The layout illustrated here, with the shared link somewhere in the middle of each path, is sometimes known as the **dumbbell** topology.

For the time being, we will also continue to assume the **synchronized-loss hypothesis**: that in any one RTT either *both* connections experience a loss or *neither* does. (This assumption is suspect; we explore it further in *14.3.3 TCP RTT bias* and in *16.3 Two TCP Senders Competing*). This was the model reviewed previously in *13.1.1.1 A first look at fairness*; we argued there that in any RTT without a loss, the expression ($\mathtt{cwnd}_1$ - $\mathtt{cwnd}_2$) remained the same (both `cwnd`s incremented by 1), while in any RTT *with* a loss the expression ($\mathtt{cwnd}_1$ - $\mathtt{cwnd}_2$) decreased by a factor of 2 (both `cwnd`s decreased by factors of 2).

Here is a graphical version of the same argument, as originally introduced in *[CJ89]*. We plot $\mathtt{cwnd}_1$ on the x-axis and $\mathtt{cwnd}_2$ on the y-axis. An additive increase of both (in equal amounts) moves the point (x,y) = ($\mathtt{cwnd}_1$,$\mathtt{cwnd}_2$) along the line parallel to the 45° line y=x; equal multiplicative decreases of both moves the point (x,y) along a line straight back towards the origin. If the maximum network capacity is Max, then a loss occurs whenever x+y exceeds Max, that is, the point (x,y) crosses the line x+y=Max.



Beginning at the initial state, additive increase moves the state at a 45° angle up to the line x+y=Max, in small increments denoted by the small arrowheads. At this point a loss would occur, and the state jumps back halfway *towards the origin*. The state then moves at 45° incrementally back to the line x+y=Max, and continues to zigzag slowly towards the equal-shares line y=x.

Any attempt to increase cwnd faster than linear will mean that the increase phase is not parallel to the line y=x, but in fact veers away from it. This will slow down the process of convergence to equal shares.

Finally, here is a **timeline** version of the argument. We will assume that the A–C path capacity, the B–D path capacity and R's queue size all add up to 24 packets, and that in any RTT in which $cwnd_1 + cwnd_2 > 24$, both connections experience a packet loss. We also assume that, initially, the first connection has cwnd=20, and the second has cwnd=1.

T	A–C	B–D	
0	20	1	
1	21	2	
2	22	3	total cwnd is 25; packet loss
3	11	1	
4	12	2	
5	13	3	
6	14	4	

Continued on next page

Table 1 – continued from previous page

T	A–C	B–D	
7	15	5	
8	16	6	
9	17	7	
10	18	8	second packet loss
11	9	4	
12	10	5	
13	11	6	
14	12	7	
15	13	8	
16	14	9	
17	15	10	third packet loss
18	7	5	
19	8	6	
20	9	7	
21	10	8	
22	11	9	
23	12	10	
24	13	11	
25	14	12	fourth loss
26	7	6	`cwnd`s are quite close
...			
32	13	12	loss
33	6	6	`cwnd`s are equal

So far, fairness seems to be winning.

14.3.1 Example 2: Faster additive increase

Here is the same kind of timeline – again with the synchronized-loss hypothesis – but with the additive-increase increment changed from 1 to 2 for the B–D connection (but not for A–C); both connections start with `cwnd`=1. Again, we assume a loss occurs when $\texttt{cwnd}_1 + \texttt{cwnd}_2 > 24$

T	A–C	B–D	
0	1	1	
1	2	3	
2	3	5	
3	4	7	
4	5	9	
5	6	11	
6	7	13	
7	8	15	
8	9	17	first packet loss
9	4	8	
10	5	10	
11	6	12	
12	7	14	
13	8	16	
14	9	18	second loss
15	4	9	essentially where we were at T=9

The effect here is that the second connection's average cwnd, and thus its throughput, is double that of the first connection. Thus, changes to the additive-increase increment lead to very significant changes in fairness. In general, an additive-increase value of α increases throughput, relative to TCP Reno, by a factor of α.

14.3.2 Example 3: Longer RTT

For the next example, we will return to standard TCP Reno, with an increase increment of 1. But here we assume that the RTT of the A–C connection is **double** that of the B–D connection, perhaps because of additional delay in the A–R link. The longer RTT means that the first connection sends packet flights only when T is even. Here is the timeline, where we allow the first connection a hefty head-start. As before, we assume a loss occurs when $\mathtt{cwnd}_1 + \mathtt{cwnd}_2 > 24$.

T	A–C	B–D	
0	20	1	
1		2	
2	21	3	
3		4	
4	22	5	first loss
5		2	
6	11	3	
7		4	
8	12	5	
9		6	
10	13	7	
11		8	
12	14	9	
13		10	

Continued on next page

Table 2 – continued from previous page

T	A–C	B–D	
14	15	11	second loss
15		5	
16	7	6	
17		7	
18	8	8	B–D has caught up
20	9	10	**from here on only even values for T shown**
22	10	12	
24	11	14	third loss
26	5	8	B–D is now ahead
28	6	10	
30	7	12	
32	8	14	
34	9	16	fourth loss
35		8	
36	4	9	
38	5	11	
40	6	13	
42	7	15	
44	8	17	fifth loss
45		8	
46	4	9	exactly where we were at T=36

The interval 36⩽T<46 represents the steady state here; the first connection's average `cwnd` is 6 while the second connection's average is (8+9+...+16+17)/10 = 12.5. Worse, the first connection sends a windowful only half as often. In the interval 36⩽T<46 the first connection sends 4+5+6+7+8 = 30 packets; the second connection sends 125. The cost of the first connection's longer RTT is *quadratic*; in general, as we argue more formally below, if the first connection has RTT = $\lambda > 1$ relative to the second's, then its bandwidth will be reduced by a factor of $1/\lambda^2$.

Is this fair?

Early thinking was that there was something to fix here; see *[F91]* and *[FJ92]*, §3.3 where the Constant-Rate window-increase algorithm is discussed. A more recent attempt to address this problem is **TCP Hybla**, *[CF04]*; discussed later in *15.12 TCP Hybla*.

Alternatively, we may simply *define* TCP Reno's bandwidth allocation as "fair", at least in some contexts. This approach is particularly common when the issue at hand is making sure other TCP implementations – and non-TCP flows – compete for bandwidth in roughly the same way that TCP Reno does. While TCP Reno's strategy is now understood to be "greedy" in some respects, "fixing" it in the Internet at large is generally recognized as a very difficult option.

14.3.3 TCP RTT bias

Let us consider more carefully the way TCP allocates bandwidth between two connections sharing a bottleneck link with relative RTTs of 1 and λ>1. We claimed above that the slower connection's bandwidth will

be reduced by a factor of $1/\lambda^2$; we will now show this under some assumptions. First, uncontroversially, we will assume FIFO droptail queuing at the bottleneck router, and also that the network ceiling (and hence `cwnd` at the point of loss) is "sufficiently" large. We will also assume, for simplicity, that the network ceiling C is constant.

We need one more assumption: that most loss events are experienced by both connections. This is the **synchronized losses** hypothesis, and is the most debatable; we will explore it further in the next section. But first, here is the general argument with this assumption.

Let connection 1 be the faster connection, and assume a steady state has been reached. Both connections experience loss when $cwnd_1+cwnd_2 \geqslant C$, because of the synchronized-loss hypothesis. Let c_1 and c_2 denote the respective window sizes at the point just before the loss. Both `cwnd` values are then halved. Let N be the number of RTTs *for connection 1* before the network ceiling is reached again. During this time c_1 increases by N; c_2 increases by approximately N/λ if N is reasonably large. Each of these increases represents half the corresponding `cwnd`; we thus have $c_1/2 = N$ and $c_2/2 = N/\lambda$. Taking ratios of respective sides, we get $c_1/c_2 = N/(N/\lambda) = \lambda$, and from that we can solve to get $c_1 = C\lambda/(1+\lambda)$ and $c_2 = C/(1+\lambda)$.

To get the relative bandwidths, we have to count packets sent during the interval between losses. Both connections have `cwnd` averaging about 3/4 of the maximum value; that is, the average `cwnd`s are 3/4 c_1 and 3/4 c_2 respectively. Connection 1 has N RTTs and so sends about 3/4 $c_1 \times N$ packets. Connection 2, with its slower RTT, has only about N/λ RTTs (again we use the assumption that N is reasonably large), and so sends about 3/4 $c_2 \times N/\lambda$ packets. The ratio of these is $c_1/(c_2/\lambda) = \lambda^2$. Connection 1 sends fraction $\lambda^2/(1+\lambda^2)$ of the packets; connection 2 sends fraction $1/(1+\lambda^2)$.

14.3.4 Synchronized-Loss Hypothesis

The synchronized-loss hypothesis is based on the idea that, if the queue is full, late-arriving packets from *each* connection will find it so, and be dropped. Once the queue becomes full, in other words, it stays full for long enough for each connection to experience a packet loss.

That said, it is certainly possible to come up with hypothetical situations where losses are not synchronized. Recall that a TCP Reno connection's `cwnd` is incremented by only 1 each RTT; losses generally occur when this single extra packet generated by the increment to `cwnd` arrives to find a full queue. Generally speaking, packets are leaving the queue about as fast as they are arriving; actual overfull-queue instants may be rare. It is certainly conceivable that, at least some of the time, one connection would overflow the queue by one packet, and halve its `cwnd`, in a short enough time interval that the other connection misses the queue-full moment entirely. Alternatively, if queue overflows lead to effectively random selection of lost packets (as would certainly be true for random-drop queuing, and might be true for tail-drop if there were sufficient randomness in packet arrival times), then there is a finite probability that all the lost packets at a given loss event come from the same connection.

The synchronized-loss hypothesis is still valid if either or both connection experiences *more* than one packet loss, within a single RTT; the hypothesis fails only when one connection experiences no losses.

We will return to possible failure of the synchronized-loss hypothesis in *14.5.2 Unsynchronized TCP Losses*. In *16.3 Two TCP Senders Competing* we will consider some TCP Reno simulations in which actual measurement does not entirely agree with the synchronized-loss model. Two problems will emerge. The first is that when two connections compete in isolation, a form of synchronization known as **phase effects** (*16.3.4 Phase Effects*) can introduce a persistent perhaps-unexpected bias. The second is that the longer-RTT connection often does manage to miss out on the full-queue moment entirely, as discussed above

in the second paragraph of this section. This results in a larger cwnd than the synchronized-loss hypothesis would predict.

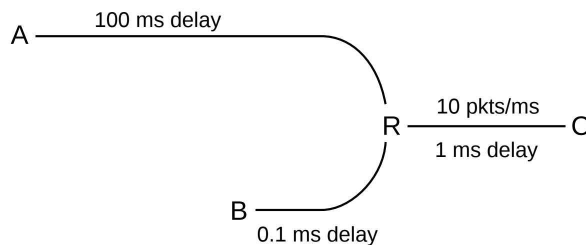
14.3.5 Loss Synchronization

The synchronized-loss hypothesis assumes *all* losses are synchronized. There is another side to this phenomenon that is an issue even if only some reasonable fraction of loss events are synchronized: synchronized losses may represent a collective inefficiency in the use of bandwidth. In the immediate aftermath of a synchronized loss, it is very likely that the bottleneck link will go underutilized, as (at least) two connections using it have just cut their sending rate in half. Better utilization would be achieved if the loss events could be staggered, so that at the point when connection 1 experiences a loss, connection 2 is only halfway to its next loss. For an example, see exercise 18.0.

This loss synchronization is a very real effect on the Internet, even if losses are not necessarily *all* synchronized. A major contributing factor to synchronization is the relatively slow response of all parties involved to packet loss. In the diagram above at *14.3 TCP Fairness with Synchronized Losses*, if A increments its cwnd leading to an overflow at R, the A–R link is likely still full of packets, and R's queue remains full, and so there is a reasonable likelihood that sender B will also experience a loss, even if its cwnd was not particularly high, simply because its packets arrived at the wrong instant. Congestion, unfortunately, takes time to clear.

14.3.6 Extreme RTT Ratios

What happens to TCP fairness if one TCP connection has a 100-fold-larger RTT than another? The short answer is that the shorter connection *may* get 10,000 times the throughput. The longer answer is that this isn't quite as easy to set up as one might imagine. For the arguments above, it is necessary for the two connections to have a common bottleneck link:



In the diagram above, the A–C connection wants its cwnd to be about 200 ms $\times$ 10 packets/ms = 2,000 packets; it is competing for the R–C link with the B–D connection which is happy with a cwnd of 22. If R's queue capacity is also about 20, then with most of the bandwidth the B–C connection will experience a loss about every 20 RTTs, which is to say every 22 ms. If the A–C link shares even a modest fraction of those losses, it is indeed in trouble.

However, the A–C cwnd cannot fall below 1.0; to test the 10,000-fold hypothesis taking this constraint into account we would have to scale up the numbers on the B–C link so the transit capacity there was at least 10,000. This would mean a 400 Gbps R–C bandwidth, or else an unrealistically large A–R delay.

As a second issue, realistically the A–C link is much more likely to have its bottleneck somewhere in the middle of its long path. In a typical real scenario along the lines of that diagrammed above, B, C and R are all local to a site, and bandwidth of long-haul paths is almost always less than the local LAN bandwidth within a site. If the A–R path has a 1 packet/ms bottleneck somewhere, then it may be less likely to be as dramatically affected by B–C traffic.

A few actual simulations using the methods of *16.3 Two TCP Senders Competing* resulted in an average `cwnd` for the A–C connection of between 1 and 2, versus a B–C `cwnd` of 20-25, regardless of whether the two links shared a bottleneck or if the A–C link had its bottleneck somewhere along the A–R path. This *may* suggest that the A–C connection was indeed saved by the 1.0 `cwnd` minimum.

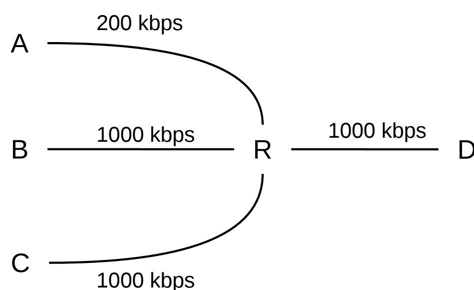
14.4 Notions of Fairness

There are several definitions for fair allocation of bandwidth among flows sharing a bottleneck link. One is **equal-shares fairness**; another is what we might call **TCP-Reno fairness**: to divide the bandwidth the way TCP Reno would. There are additional approaches to deciding what constitutes a fair allocation of bandwidth.

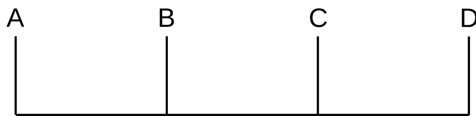
14.4.1 Max-Min Fairness

A natural generalization of equal-shares fairness to the case where some flows may be capped is **max-min fairness**, in which no flow bandwidth can be increased without decreasing some *smaller* flow rate. Alternatively, we maximize the bandwidth of the smallest-capacity flow, and then, with that flow fixed, maximize the flow with the next-smallest bandwidth, *etc*. A more intuitive explanation is that we distribute bandwidth in tiny increments equally among the flows, until the bandwidth is exhausted (meaning we have divided it equally), or one flow reaches its externally imposed bandwidth cap. At this point we continue incrementing among the remaining flows; any time we encounter a flow's external cap we are done with it.

As an example, consider the following, where we have connections A–D, B–D and C–D, and where the A–R link has a bandwidth of 200 Kbps and all other links are 1000 Kbps. Starting from zero, we increment the allocations of each of the three connections until we get to 200 Kbps per connection, at which point the A–D connection has maxed out the capacity of the A–R link. We then continue allocating the remaining 400 Kbps equally between B–D and C–D, so they each end up with 400 Kbps.



As another example, known as the **parking-lot topology**, suppose we have the following network:



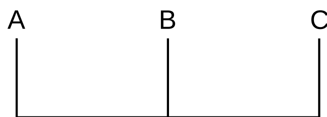
There are four connections: one from A to D covering all three links, and three single-link connections A–B, B–C and C–D. Each link has the same bandwidth. If bandwidth allocations are incrementally distributed among the four connections, then the first point at which any link bandwidth is maxed out occurs when all four connections each have 50% of the link bandwidth; max-min fairness here means that each connection has an equal share.

14.4.2 Proportional Fairness

A bandwidth allocation of rates $\langle r_1,r_2,...,r_N\rangle$ for N connections satisfies **proportional fairness** if it is a legal allocation of bandwidth, and for any other allocation $\langle s_1,s_2,...,s_N\rangle$, the aggregate proportional change satisfies

$$(r_1–s_1)/s_1 + (r_2–s_2)/s_2 + ... + (r_N–s_N)/s_N < 0$$

Alternatively, proportional fairness means that the sum $\log(r_1)+\log(r_2)+...+\log(r_N)$ is minimized. If the connections share only the bottleneck link, proportional fairness is achieved with equal shares. However, consider the following two-stage parking-lot network:



Suppose the A–B and B–C links have bandwidth 1 unit, and we have three connections A–B, B–C and A–C. Then a proportionally fair solution is to give the A–C link a bandwidth of 1/3 and each of the A–B and B–C links a bandwidth of 2/3 (so each link has a total bandwidth of 1). For any change Δb in the bandwidth for the A–C link, the A–B and B–C links each change by $-\Delta b$. Equilibrium is achieved at the point where a 1% reduction in the A–C link results in two 0.5% increases, that is, the bandwidths are divided in proportion 1:2. Mathematically, if x is the throughput of the A–C connection, we are minimizing $\log(x) + 2\log(1-x)$.

Proportional fairness partially addresses the problem of TCP Reno's bias against long-RTT connections; specifically, TCP's bias here is still not proportionally fair, but TCP's response is closer to proportional fairness than it is to max-min fairness.

14.5 TCP Reno loss rate versus `cwnd`

It turns out that we can express a connection's average `cwnd` in terms of the **packet loss rate**, p, *eg* $p = 10^{-4}$ = one packet lost in 10,000. The relationship comes by assuming that all packet losses are because the network ceiling was reached. We will also assume that, when the network ceiling is reached, only one packet is lost, although we can dispense with this by counting a "cluster" of related losses (within, say, one RTT) as a single *loss event*.

Let C represent the network ceiling – so that when cwnd reaches C a packet loss occurs. While C is constant only for a very stable network, C usually does not vary by much; we will assume here that it is constant. Then cwnd varies between C/2 and C, with packet drops occurring whenever cwnd = C is reached. Let N = C/2. Then between two consecutive packet loss events, that is, over one "tooth" of the TCP connection, a total of N+(N+1)+ ... +2N packets are sent in N+1 flights; this sum can be expressed algebraically as 3/2 N(N+1) $\simeq$ 1.5 N^2. The loss rate is thus one packet out of every 1.5 N^2, and the loss rate p is 1/(1.5 N^2).

The average cwnd in this scenario is 3/2 N (that is, the average of N=$cwnd_{min}$ and 2N=$cwnd_{max}$). If we let M = 3/2 N represent the average cwnd, $cwnd_{mean}$, we can express the above loss rate in terms of M: the number of packets between losses is 2/3 M^2, and so p=3/2 M^{-2}.

Now let us solve this for M=$cwnd_{mean}$ in terms of p; we get M^2 = 3/2 p^{-1} and thus

$$M = cwnd_{mean} = 1.225\ p^{-1/2}$$

where 1.225 is the square root of 3/2. Seen in this form, a given network loss rate sets the window size; this loss rate is ultimately be tied to the network capacity. If we are interested in the maximum cwnd instead of the mean, we multiply the above by 4/3.

From the above, the *bandwidth* available to a connection is now as follows (though RTT may not be constant):

$$bandwidth = cwnd/RTT = 1.225/(RTT \times \sqrt{p})$$

In *[PFTK98]* the authors consider a TCP Reno model that takes into account the measured frequency of coarse timeouts (in addition to fast-recovery responses leading to cwnd halving), and develop a related formula with additional terms.

As the bottleneck queue capacity increases, both cwnd and the number of packets between losses (1/p) increase, connected as above. Once the queue is large enough that the bottleneck link is 100% utilized, however, the bandwidth no longer increases.

Another way to view this formula is to recall that 1/p is the number of packets per tooth; that is, 1/p is the tooth "area". Squaring both sides, the formula says that the TCP Reno tooth area is proportional to the square of the average tooth height (that is, to $cwnd_{mean}$) as the network capacity increases (that is, as $cwnd_{mean}$ increases).

14.5.1 Irregular teeth

In the preceding, we assumed that all teeth were the same size. What if they are not? In *[OKM96]*, this problem was considered under the assumption that every packet faces the same (small) loss probability (and so the intervals between packet losses are exponentially distributed). In this model, it turns out that the above formula still holds except the constant changes from 1.225 to 1.309833.

To understand how irregular teeth lead to a bigger constant, imagine sending a large number K of packets which encounter n losses. If the losses are regularly spaced, then the TCP graph will have n equally sized teeth, each with K/n packets. But if the n losses are randomly distributed, some teeth will be larger and some will be smaller. The *average* tooth height will be the same as in the regularly-spaced case (see exercise 13.0). However, the number of packets in any one tooth is generally related to the *square* of the height of that tooth, and so larger teeth will count disproportionately more. Thus, the random distribution will have a higher total number of packets delivered and thus a higher mean cwnd.

See also exercise 17.0, for a simple simulation that generates a numeric estimate for the constant 1.309833.

Note that losses at uniformly distributed random intervals may not be an ideal model for TCP either; in the presence of congestion, loss events are far from statistical independence. In particular, immediately following one loss another loss is unlikely to occur until the queue has time to fill up.

14.5.2 Unsynchronized TCP Losses

In *14.3.3 TCP RTT bias* we considered a model in which all loss events are fully synchronized; that is, whenever the queue becomes full, *both* TCP Reno connections always experience packet loss. In that model, if $RTT_2/RTT_1 = \lambda$ then $cwnd_1/cwnd_2 = \lambda$ and $bandwidth_1/bandwidth_2 = \lambda^2$, where $cwnd_1$ and $cwnd_2$ are the respective average values for `cwnd`.

What happens if loss events for two connections do not have such a neat one-to-one correspondence? We will derive the ratio of loss events (or, more precisely, TCP loss *responses*) for connection 1 versus connection 2 in terms of the bandwidth and RTT ratios, without using the synchronized-loss hypothesis.

Note that we are comparing the total number of loss events (or loss responses) here – the total number of TCP Reno teeth – over a large time interval, and not the relative *per-packet* loss probabilities. One connection might have numerically more losses than a second connection but, by dint of a smaller RTT, send more packets between its losses than the other connection and thus have *fewer* losses per packet.

Let $losscount_1$ and $losscount_2$ be the number of loss responses for each connection over a long time interval T. For i=1 and i=2, the ith connection's per-packet loss probability is $p_i = losscount_i/(bandwidth_i \times T) = (losscount_i \times RTT_i)/(cwnd_i \times T)$. But by the result of *14.5 TCP Reno loss rate versus cwnd*, we also have $cwnd_i = k/\sqrt{p_i}$, or $p_i = k^2/cwnd_i^2$. Equating, we get

$$p_i = k^2/cwnd_i^2 = (losscount_i \times RTT_i) / (cwnd_i \times T)$$

and so

$$losscount_i = k^2T / (cwnd_i \times RTT_i)$$

Dividing and canceling, we get

$$losscount_1/losscount_2 = (cwnd_2/cwnd_1) \times (RTT_2/RTT_1)$$

We will make use of this in *16.4.2.2 Relative loss rates*.

We can go just a little further with this: let γ denote the losscount ratio above:

$$\gamma = (cwnd_2/cwnd_1) \times (RTT_2/RTT_1)$$

Therefore, as $RTT_2/RTT_1 = \lambda$, we must have $cwnd_2/cwnd_1 = \gamma/\lambda$ and thus

$$bandwidth_1/bandwidth_2 = (cwnd_1/cwnd_2) \times (RTT_2/RTT_1) = \lambda^2/\gamma.$$

Note that if $\gamma=\lambda$, that is, if the longer-RTT connection has fewer loss events in exact inverse proportion to the RTT, then $bandwidth_1/bandwidth_2 = \lambda = RTT_2/RTT_1$, and also $cwnd_1/cwnd_2 = 1$.

14.6 TCP Friendliness

Suppose we are sending packets using a non-TCP real-time protocol. How are we to manage congestion? In particular, how are we to manage congestion in a way that treats other connections – particularly TCP Reno connections – fairly?

For example, suppose we are sending interactive audio data in a congested environment. Because of the real-time nature of the data, we cannot wait for lost-packet recovery, and so must use UDP rather than TCP. We might further suppose that we can modify the encoding so as to reduce the sending rate as necessary – that is, that we are using *adaptive* encoding – but that we would prefer in the absence of congestion to keep the sending rate at the high end. We might also want a relatively uniform *rate* of sending; the TCP sawtooth leads to periodic variations in throughput that we may wish to avoid.

Our application may not be windows-based, but we can still monitor the number of packets it has in flight on the network at any one time; if the packets are small, we can count bytes instead. We can use this count instead of the TCP `cwnd`.

We will say that a given communications strategy is **TCP Friendly** if the number of packets on the network at any one time is approximately equal to the TCP Reno $\texttt{cwnd}_{mean}$ for the prevailing packet loss rate p. Note that – assuming losses are independent events, which is definitely not quite right but which is often Close Enough – in a long-enough time interval, all connections sharing a common bottleneck can be expected to experience approximately the same packet loss rate.

The point of TCP Friendliness is to regulate the number of the non-Reno connection's outstanding packets in the presence of competition with TCP Reno, so as to achieve a degree of fairness. In the absence of competition, the number of any connection's outstanding packets will be bounded by the transit capacity plus capacity of the bottleneck queue. Some non-Reno protocols (*eg* TCP Vegas, *15.6 TCP Vegas*, or constant-**rate** traffic, *14.6.2 RTP*) may in the absence of competition have a loss rate of zero, simply because they never overflow the queue.

Another way to approach TCP Friendliness is to start by *defining* "Reno Fairness" to be the bandwidth allocations that TCP Reno assigns in the face of competition. TCP Friendliness then simply means that the given non-Reno connection will get its Reno-Fair share – not more, not less.

We will return to TCP Friendliness in the context of general AIMD in *14.7 AIMD Revisited*.

14.6.1 TFRC

TFRC, or TCP-Friendly Rate Control, **RFC 3448**, uses the loss rate experienced, p, and the formulas above to calculate a sending rate. It then allows sending at that rate; that is, TFRC is rate-based rather than window-based. As the loss rate increases, the sending rate is adjusted downwards, and so on. However, adjustments are done more smoothly than with TCP, giving the application a more gradually changing transmission rate.

From **RFC 5348**:

> TFRC is designed to be reasonably fair when competing for bandwidth with TCP flows, where we call a flow "reasonably fair" if its sending rate is generally within a **factor of two** of the sending rate of a TCP flow under the same conditions. [emphasis added; a factor of two might not be considered "close enough" in some cases.]

The penalty of having smoother throughput than TCP while competing fairly for bandwidth is that TFRC responds more slowly than TCP to changes in available bandwidth.

TFRC senders include in each packet a sequence number, a timestamp, and an estimated RTT.

The TFRC receiver is charged with sending back feedback packets, which serve as (partial) acknowledgments, and also include a receiver-calculated value for the loss rate over the previous RTT. The response packets also include information on the current actual RTT, which the sender can use to update its estimated RTT. The TFRC receiver might send back only one such packet per RTT.

The actual response protocol has several parts, but if the loss rate increases, then the primary feedback mechanism is to *calculate* a new (lower) sending rate, using some variant of the `cwnd` = $k/\sqrt{p}$ formula, and then shift to that new rate. The rate would be cut in half only if the loss rate p quadrupled.

Newer versions of TFRC have a various features for responding more promptly to an unusually sudden problem, but in normal use the calculated sending rate is used most of the time.

14.6.2 RTP

The **Real-Time Protocol**, or RTP, is sometimes (though not always) coupled with TFRC. RTP is a UDP-based protocol for streaming time-sensitive data.

Some RTP features include:

- The sender establishes a *rate* (rather than a window size) for sending packets
- The receiver returns periodic summaries of loss rates
- ACKs are relatively infrequent
- RTP is suitable for *multicast* use; a very limited ACK rate is important when every packet sent might have hundreds of recipients
- The sender adjusts its `cwnd`-equivalent up or down based on the loss rate and the TCP-friendly `cwnd`=$k/\sqrt{p}$ rule
- Usually some sort of "stability" rule is incorporated to avoid sudden changes in rate

As a common RTP example, a typical VoIP connection using a DS0 (64 Kbps) rate might send one packet every 20 ms, containing 160 bytes of voice data, plus headers.

For a combination of RTP and TFRC to be useful, the underlying application must be **rate-adaptive**, so that the application can still function when the available rate is reduced. This is often not the case for simple VoIP encodings; see *20.11.4 RTP and VoIP*.

We will return to RTP in *20.11 Real-time Transport Protocol (RTP)*.

The UDP-based QUIC transport protocol (*11.1.1 QUIC*) uses a congestion-control mechanism compatible with Cubic TCP (*15.15 TCP CUBIC*), which isn't quite the same as TCP Reno. But QUIC could just as easily have used TFRC to achieve TCP-Reno-friendliness.

14.6.3 DCCP Congestion Control

We saw DCCP earlier in *11.1.2 DCCP* and *12.22.3 DCCP*. DCCP also includes a set of congestion-management "profiles"; a connection can choose the profile that best fits its needs. The two standard ones are the TCP-Reno-like profile (**RFC 4341**) and the TFRC profile (**RFC 4342**).

In the Reno-like profile, every packet is acknowledged (though, as with TCP, ACKs may be sent on the arrival of every other Data packet). Although DCCP ACKs are not cumulative, use of the TCP-SACK-like ACK-vector format ensures that acknowledgments are received reliably except in extreme-loss situations.

The sender maintains `cwnd` much as a TCP Reno sender would. It is incremented by one for each RTT with no loss, and halved in the event of packet loss. Because sliding windows is not used, `cwnd` does not represent a window size. Instead, the sender maintains an Estimated FlightSize (*13.4 TCP Reno and Fast Recovery*), which is the sender's best guess at the number of outstanding packets. In **RFC 4341** this is referred to as the **pipe value**. The sender is then allowed to send additional packets as long as `pipe` < `cwnd`.

The Reno-like profile also includes a slow start mechanism.

In the TFRC profile, an ACK is sent at least once per RTT. Because ACKs are sent less frequently, it may occasionally be necessary for the sender to send an ACK of ACK.

As with TFRC generally, a DCCP sender using the TFRC profile has its rate limited, rather than its window size.

DCCP provides a convenient programming framework for use of TFRC, complete with (at least in the linux world), a traditional socket interface. The developer does not have to deal with the TFRC rate calculations directly.

14.7 AIMD Revisited

TCP Tahoe chose an increase increment of 1 on no losses, and a decrease factor of 1/2 otherwise.

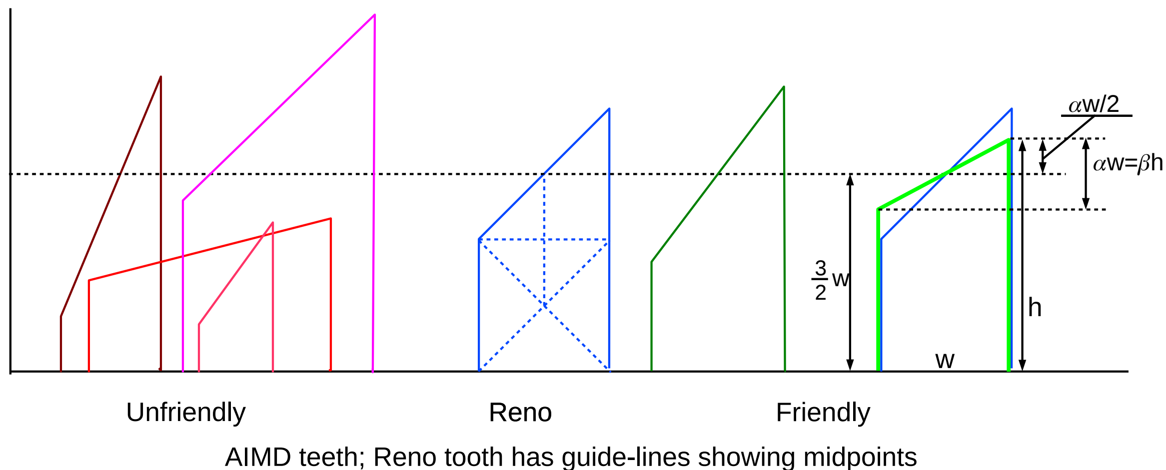
Another approach to TCP Friendliness is to retain TCP's additive-increase, multiplicative-decrease strategy, but to change the numbers. Suppose we denote by AIMD(α,β) the strategy of incrementing the window size by α after a window of no losses, and multiplying the window size by (1-β)<1 on loss (so β=0.1 means the window is reduced *by* 10%). TCP Reno is thus AIMD(1,0.5).

Any AIMD(α,β) protocol also follows a sawtooth, where the slanted top to the tooth has slope α. All combinations of α>0 and 0<β<1 are possible. The dimensions of one tooth of the sawtooth are somewhat constrained by α and β. Let h be the maximum height of the tooth and let w be the width (as measured in RTTs). Then, if the losses occur at regular intervals, the height of the tooth at the left (low) edge is (1-β)h and the total vertical difference is βh. This vertical difference must also be αw, and so we get αw = βh, or h/w = α/β; these values are labeled on the rightmost teeth in the diagram below. These equations mean that the proportions of the tooth (h to w) are determined by α and β. Finally, the mean height of the tooth is (1-β/2)h.

We are primarily interested in AIMD(α,β) cases which are TCP Friendly (*14.6 TCP Friendliness*). TCP friendliness means that an AIMD(α,β) connection with the same loss rate as TCP Reno will have the same mean `cwnd`. Each tooth of the sawtooth represents one loss. The number of packets sent per tooth is, using h and w as in the previous paragraph, (1-β/2)hw.

Geometrically, the number of packets sent per tooth is the *area* of the tooth, so two connections with the same per-packet loss rate will have teeth with the same area. TCP Friendliness means that two connections will have the same mean `cwnd` and thus the same average tooth height. If the teeth of two connections have the same area and the same average height, they must have the same width (in RTTs), and thus that the rates of loss per unit *time* must be equal, not just the rates of loss per number of packets.

The diagram below shows a TCP Reno tooth (blue) together with some unfriendly AIMD(α,β) teeth on the left (red) and two friendly teeth on the right (green), the second friendly tooth is superimposed on the Reno tooth.



AIMD teeth; Reno tooth has guide-lines showing midpoints

The additional dashed lines within the central Reno tooth demonstrate the Reno $1\times1\times2$ proportions, and show that the horizontal dashed line, representing $\texttt{cwnd}_{\text{mean}}$, is at height 3/2 w, where w is, as before, the width.

In the rightmost green tooth, superimposed on the Reno tooth, we can see that h = (3/2)$\times$w + (α/2)$\times$w. We already know h = (α/β)$\times$w; setting these expressions equal, canceling the w and multiplying by 2 we get (3+α) = 2α/β, or β = 2α/(3+α). Solving for β we get

$$\alpha = 3\beta/(2-\beta)$$

or $\alpha \simeq 1.5\beta$ for small β. As the reduction factor 1-β gets closer to 1, the protocol can remain TCP-friendly by appropriately reducing α; eg AIMD(1/5, 1/8).

Having a small β means that a connection does not have sudden bandwidth drops when losses occur; this can be important for applications that rely on a regular rate of data transfer (such as voice). Such applications are sometimes said to be slowly responsive, in contrast to TCP's `cwnd` = `cwnd`/2 rapid response.

14.7.1 AIMD and Convergence to Fairness

While TCP-friendly AIMD(α,β) protocols will converge to fairness when competing with TCP Reno (with equal RTTs), a consequence of decreasing β is that fairness may take longer to arrive; here is an example. We will assume, as above in *14.3.3 TCP RTT bias*, that loss events for the two competing connections are synchronized. Recall that for two same-RTT TCP Reno connections (that is, AIMD(α,β) where β=1/2), if the initial difference in the connections' respective `cwnds` is D, then D is reduced by half on each loss event.

Now suppose we have two AIMD(α,β) connections with some other value of β, and again with a difference D in their cwnd values. The two connections will each increase cwnd by α each RTT, and so when losses are not occurring D will remain constant. At loss events, D will be reduced by a factor of 1-β. If β=1/4, corresponding to α=3/7, then at each loss event D will be reduced only to 3/4 D, and the "half-life" of D will be almost twice as large. The two connections will still converge to fairness as D→0, but it will take twice as long.

14.8 Active Queue Management

Active Queue Management (AQM) means that routers take some active steps to manage their queues. The primary goal of AQM is to reduce excessive queuing delays; *cf* *13.7.1 Bufferbloat*. A secondary goal is to improve the performance of TCP connections – which constitute the vast majority of Internet traffic – through the router. By signaling to TCP connections that they should reduce cwnd, overall queuing delays are also reduced.

Generally routers manage their queues either by **marking** packets or by **dropping** them. All routers drop packets, but this falls into the category of active management when packets are dropped before the queue has run completely out of space. Queue management can be done at the congestion "knee", when queues just start to build (and when marking is more appropriate), or as the queue starts to become full and approaches the "cliff".

Broadly speaking, the priority queuing and random drop mechanisms (*14.1 A First Look At Queuing*, above) might be considered forms of AQM, at least if the goal was to manage the overall queue size. So might fair queuing and hierarchical queuing (*19 Queuing and Scheduling*). The mechanism most commonly associated with the AQM category, though, is RED, below, and its successors. For a discussion of the potential benefits of fair queuing to queue management, see *19.6.1 Fair Queuing and Bufferbloat*.

14.8.1 DECbit

In the congestion-avoidance technique proposed in *[RJ90]*, routers encountering early signs of congestion **marked** the packets they forwarded; senders used these markings to adjust their window size. The system became known as DECbit in reference to the authors' employer and was implemented in DECnet (closely related to the OSI protocol suite), though apparently there was never a TCP/IP implementation. The idea behind DECbit eventually made it into TCP/IP in the form of ECN, below, but while ECN – like TCP's other congestion responses – applies control near the congestion cliff, DECbit proposed introducing control when congestion was still minimal, just above the congestion knee.

The DECbit mechanism allowed routers to set a designated "congestion bit". This would be set in the data packet being forwarded, but the status of this bit would be echoed back in the corresponding ACK (otherwise the sender would never hear about the congestion).

DECbit *routers* defined "congestion" as an average queue size greater than 1.0; that is, congestion meant that the connection was just past the "knee". Routers would set the congestion bit whenever this average-queue condition was met.

The target for DECbit *senders* would then be to have 50% of packets marked as "congested". If fewer than 50% of packets were marked, cwnd would be incremented by 1; if more than 50% were marked, then cwnd would be decreased by a factor of 0.875. Note this is very different from the TCP approach in that DECbit

begins marking packets at the congestion "knee" while TCP Reno responds only to packet losses which occur at the "cliff".

A consequence of this knee-based mechanism is that DECbit shoots for very limited queue utilization, unlike TCP Reno. At a congested router, a DECbit connection would attempt to keep about 1.0 packets in the router's queue, while a TCP Reno connection might fill the remainder of the queue. Thus, DECbit would in principle compete poorly with any connection where the sender ignored the marked packets and simply tried to keep `cwnd` as large as possible. As we will see in *15.6 TCP Vegas*, TCP Vegas also strives for limited queue utilization; in *16.5 TCP Reno versus TCP Vegas* we investigate through simulation how fairly TCP Vegas competes with TCP Reno.

14.8.2 Explicit Congestion Notification (ECN)

ECN is the TCP/IP equivalent of DECbit, though the actual mechanics are quite different. The current version is specified in **RFC 3168**, modifying an earlier version in **RFC 2481**. The IP header contains a two-bit ECN field, consisting of the ECN-Capable Transport (ECT) bit and the Congestion Experienced (CE) bit; the ECN field is shown in *7.1 The IPv4 Header*. The ECT bit is set by a sender to indicate to routers that it is able to use the ECN mechanism. (These are actually the older **RFC 2481** names for the bits, but they will serve our purposes here.) The TCP header contains an additional two bits: the ECN-Echo bit (ECE) and the Congestion Window Reduced (CWR) bit; these are shown in the fourth row in *12.2 TCP Header*.

ECN and Middleboxes

In the early days of ECN, some non-ECN-aware firewalls responded to packets with the CWR or ECE bits set by dropping them and returning a spoofed RST packet to the sender. See **RFC 3360** for details and a discussion of this nonstandard use of RST. This is a good example of how "middleboxes" are sometimes obstacles to protocol evolution; see *7.7.2 Middleboxes*.

Routers set the CE bit in the IP header when they might otherwise drop the packet (or possibly when the queue is at least half full, or in lieu of a RED drop, below). As in DECbit, receivers echo the CE status back to the sender in the ECE bit of the next ACK; the reason for using the ECE bit is that this bit belongs to the TCP header and thus the TCP layer can be assured of control of it.

TCP senders treat ACKs with the ECE bit set the same as if a loss occurred: `cwnd` is cut in half. Because there is no actual loss, the arriving ACKs can still pace continued sliding-windows sending. The Fast Recovery mechanism is not needed.

When the TCP sender has responded to an ECE bit (by halving `cwnd`), it sets the CWR bit. Once the receiver has received a packet with the CE bit set in the IP layer, it sets the ECE bit in all subsequent ACKs until it receives a data packet with the CWR bit set. This provides for reliable communication of the congestion information, and helps the sender respond just once to multiple packet losses within a single windowful.

Note that the initial packet marking is done at the IP layer, but the generation of the marked ACK and the sender response to marked packets is at the TCP layer (the same is true of DECbit though the layers have different names).

Only a packet that would otherwise have been dropped has its CE bit set; the router does *not* mark all waiting packets once its queue reaches a certain threshold. Any marked packet must, as usual, wait in the

queue for its turn to be forwarded. The sender finds out about the congestion after one full RTT, versus one full RTT plus four packet transmission times for Fast Retransmit. A much earlier, "legacy" strategy was to require routers, upon dropping a packet, to immediately send back to the sender an `ICMP Source Quench` packet. This is a faster way (the fastest possible way) to notify a sender of a loss. It was never widely implemented, however, and was officially deprecated by **RFC 6633**.

Because ECN congestion is treated the same way as packet drops, ECN competes fairly with TCP Reno.

RFC 3540 is a proposal (as of 2016 not yet official) to slightly amend the mechanism described above to support detection of receivers who attempt to conceal evidence of congestion. A receiver would do this by not setting the ECE bit in the ACK when a data packet arrives marked as having experienced congestion. Such an unscrupulous (or incorrectly implemented) receiver may then gain a greater share of the bandwidth, because its sender maintains a larger `cwnd` than it should. The amendment also detects erasure of the ECE bit (or other ECN bits) by middleboxes.

The new strategy, known as the **ECN nonce**, treats the ECN bits ECT and CE as a single unit. The value 00 is used by non-ECN-aware senders, and the value 11 is used by routers as the congestion marker. ECN-aware senders mark data packets by *randomly* choosing 10 (known as ECT(0)) or 01 (known as ECT(1)). This choice encodes the *nonce bit*, with ECT(0) representing a nonce bit of 0 and ECT(1) representing 1; the nonce bit can also be viewed as the value of the second ECN bit.

The receiver is now expected, in addition to setting the ECE bit, to also return the one-bit running sum of the nonce bits in a new TCP-header bit called the nonce-sum (NS) bit, which immediately precedes the CRW bit. This sum is over all data packets received since the previous packet loss or congestion-experienced packet. The point of this is that if the receiver attempts to conceal congestion by leaving the ECE bit zero, the receiver cannot properly set the NS bit, because it does not know which of ECT(0) or ECT(1) was used. For each packet marked as experiencing congestion, the receiver has a 50% chance of guessing correctly, but over time successful guessing becomes increasingly unlikely. If ECN-noncompliance is detected, the sender must now stop using ECN, and may choose a smaller `cwnd` as a precaution.

Although we have described ECN as a mechanism implemented by routers, it can just as easily be implemented by switches, and is available in many commercial switches.

14.8.3 RED

"Traditional" routers drop packets only when the queue is full; senders have no overt indication before then that the cliff is looming. ECN improves this by informing TCP connections of the impending cliff so they can reduce `cwnd` without actually losing packets. The idea behind **Random Early Detection** (RED) routers, introduced in *[FJ93]*, is that the router is allowed to drop an occasional packet much earlier, say when the queue is less than half full. These early packet drops provide a signal to senders that they should slow down; we will call them **signaling losses**. While packets are indeed lost, they are dropped in such a manner that usually only one packet per windowful (per connection) will be lost. Classic TCP Reno, in particular, behaves poorly with multiple losses per window and RED is able to avoid such multiple losses.

RED is, strictly speaking, a *queuing discipline* in the sense of *19.4 Queuing Disciplines*; FIFO is another. It is often more helpful, however, to think of RED as a technique that an otherwise-FIFO router can use to improve the performance of TCP traffic through it.

Designing an early-drop algorithm is not trivial. A predecessor of RED known as Early Random Drop (ERD) gateways simply introduced a small uniform drop probability p, *eg* p=0.01, once the queue had

reached a certain threshold. This addresses the TCP Reno issue reasonably well, except that dropping with a uniform probability p leads to a surprisingly high rate of multiple drops in a cluster, or of long stretches with no drops. More uniformity was needed, but drops at regular intervals are too uniform.

The actual RED algorithm does two things. First, the base drop probability – p_{base} – rises steadily from a minimum queue threshold q_{min} to a maximum queue threshold q_{max} (these might be 40% and 80% respectively of the absolute queue capacity); at the maximum threshold, the drop probability is still quite small. The base probability p_{base} increases linearly in this range according to the following formula, where p_{max} is the maximum RED-drop probability; the value for p_{max} proposed in *[FJ93]* was 0.02.

$$p_{base} = p_{max} \times (avg_queuesize - q_{min})/(q_{max} - q_{min})$$

Second, as time passes after a RED drop, the actual drop probability p_{actual} begins to rise, according to the next formula:

$$p_{actual} = p_{base} / (1 - count \times p_{base})$$

Here, count is the number of packets sent since the last RED drop. With count=0 we have $p_{actual} = p_{base}$, but p_{actual} rises from then on with a RED drop guaranteed within the next $1/p_{base}$ packets. This provides a mechanism by which RED drops are uniformly enough spaced that it is unlikely two will occur in the same window of the same connection, and yet random enough that it is unlikely that the RED drops will remain synchronized with a single connection, thus targeting it unfairly.

A significant drawback to RED is that the choice of the various parameters is decidedly *ad hoc*. It is not clear how to set them so that TCP connections with both small and large bandwidth×delay products are handled appropriately, or even how to set them for a given output bandwidth. The probability p_{base} should, for example, be roughly 1/winsize, but winsize for TCP connections can vary by several orders of magnitude. **RFC 2309**, from 1998, recommended RED, but its successor **RFC 7567** from 2015 has backed away from this, recommending instead that an appropriate AQM strategy be implemented, but that the details should be left to the discretion of the router manager.

In *20.8 RED with In and Out* we will look at an application of RED to quality-of-service guarantees.

14.8.4 ADT

The paper *[SKS06]* proposes the Adaptive Drop-Tail algorithm, in which the maximum queue capacity is adjusted at intervals (of perhaps 5 minutes) in order to maintain a specific desired link-utilization target (perhaps 95%). At the end of each interval, the available queue capacity is increased or decreased (perhaps by 5%) depending on whether the link utilization was under or over the target. ADT does not selectively drop packets otherwise; if there is space for an arriving packet within the current queue capacity, it is accepted.

ADT does a good job adjusting the overall queue capacity to meet circumstances that change slowly; an example might be the size of the user pool. However, ADT does not respond to short-term fluctuations. In particular, it does not attempt to respond to fluctuations occurring within a single TCP Reno tooth. ADT also does not maintain additional queue space for transient packet bursts.

14.8.5 CoDel

The CoDel algorithm (pronounced "coddle") attempts, like RED, to use signaling losses to encourage connections to reduce their queue utilization. This allows CoDel to maintain a large total queue capacity,

available to absorb bursts, while at the same time maintaining on average a much smaller level of queue utilization. To achieve this, CoDel is able to distinguish between transient queue spikes and "standing-queue" utilization, persisting over multiple RTTs; the canonical example of the latter is TCP Reno's queue buildup towards the right-hand edge of each sawtooth. Unlike RED, CoDel has essentially no tunable parameters, and adapts to a wide range of bandwidths and traffic types. See *[NJ12]* and the Internet Draft draft-ietf-aqm-codel-06.

CoDel measures the minimum value of queue utilization over a designated short time period known as the **Interval**. The Interval is intended to be a little larger than most connection RTT_{noLoad} values; it is typically 100 ms. Through this minimum-utilization statistic, CoDel will easily be able to detect a TCP Reno connection's queue-building phase, which except for short-RTT connections will last for many Intervals.

CoDel measures this queue utilization in terms of the time the packet spends in the queue (its "sojourn time") rather than the size of the queue in bytes. While these two measures are proportional at any one router, the use of time rather than space means that the CoDel algorithm is independent of the outbound bandwidth, and so does not need to be configured for that bandwidth.

CoDel's target for the minimum queue utilization is typically 5% of the interval, or 5 ms, although 10% is also reasonable. If the minimum utilization is smaller, no action is taken. If the minimum utilization becomes larger, then CoDel enters its "dropping mode", drops a packet, and begins scheduling additional packet drops. This lasts until the minimum utilization is again below the target, and CoDel returns to its "normal mode".

Once dropping mode begins, the second drop is scheduled for one Interval after the first drop, though the second drop may not occur if CoDel is able to return to normal mode. While CoDel remains in dropping mode, additional packet drops are scheduled after times of Interval/$\sqrt{2}$, Interval/$\sqrt{3}$, *etc*; that is, the dropping rate accelerates in proportion to $\sqrt{n}$ until the minimum time packets spend in the queue is small enough again that CoDel is able to return to normal mode.

If the traffic consists of a single TCP Reno connection, CoDel will drop one of its packets as soon as the queue utilization hits 5%. That will cause `cwnd` to halve, most likely making even a second packet drop unnecessary. If the connection's RTT was approximately equal to the Interval, then its link utilization will be the same as if the queue capacity was fixed at 5% of the transit capacity, or 79% (*13.12 Exercises*, 12.5). However, if there are a modest number of unsynchronized TCP connections, the link-utilization rate climbs to above 90% (*[NJ12]*, figs 5 and 8).

If the traffic consists of several TCP Reno connections, a few drops should be all that are necessary to force most of the connections to halve their `cwnd`s, and thus greatly reduce their collective queue utilization. However, even if the traffic consists of a single *fixed-rate* UDP connection, with too high a rate for the bottleneck, CoDel still works. In this case it drops as many packets as it needs to in order to drive down the queue utilization, and this cycle repeats as necessary. An additional feature of CoDel is that if the dropping mode is re-entered quickly, the dropping rate picks up where it left off.

14.9 The High-Bandwidth TCP Problem

The TCP Reno algorithm has a serious consequence for high-bandwidth connections: the `cwnd` needed implies a very small – unrealistically small – packet-loss rate p. "Noise" losses (losses not due to congestion) are not frequent but no longer negligible; these keep the window significantly smaller than it should be. The

following table, from **RFC 3649**, is based on an RTT of 0.1 seconds and a packet size of 1500 bytes, for various throughputs. The `cwnd` values represent the bandwidth×RTT products.

TCP Throughput (Mbps)	RTTs between losses	`cwnd`	Packet Loss Rate P
1	5.5	8.3	0.02
10	55	83	0.0002
100	555	833	2×10^{-6}
1000	5555	8333	2×10^{-8}
10,000	55555	83333	2×10^{-10}

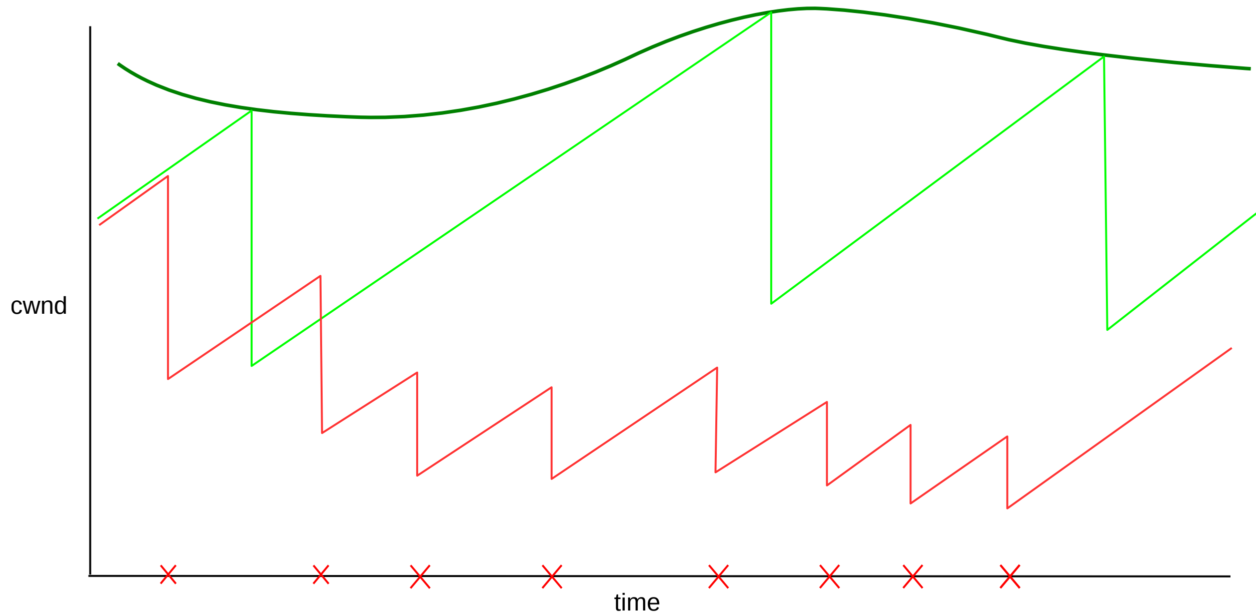
Note the very small value of the loss probability needed to support 10 Gbps; this works out to a bit error rate of less than 2×10^{-14}. For fiber optic data links, alas, a physical bit error rate of 10^{-13} is often considered acceptable; there is thus no way to support the window size of the final row above. (The use of error-correcting codes on OTN links, *4.2.3 Optical Transport Network*, can reduce the bit error rate to less than 10^{-15}.) Another source of "noise" losses are queue overflows within Ethernet switches; switches tend to have much shorter queues than routers. At 10 Gbps, a switch is forwarding one packet every microsecond; at that rate a burst does not have to last long to overrun the switch's queue.

Here is a similar table, expressing `cwnd` in terms of the packet loss rate:

Packet Loss Rate P	`cwnd`	RTTs between losses
10^{-2}	12	8
10^{-3}	38	25
10^{-4}	120	80
10^{-5}	379	252
10^{-6}	1,200	800
10^{-7}	3,795	2,530
10^{-8}	12,000	8,000
10^{-9}	37,948	25,298
10^{-10}	120,000	80,000

The above two tables indicate that large window sizes require extremely small drop rates. This is the **high-bandwidth-TCP problem**: how do we maintain a large window when a path has a large bandwidth×delay product? The primary issue is that non-congestive (noise) packet losses bring the window size down, potentially far below where it could be. A secondary issue is that, even if such random drops are not significant, the increase of `cwnd` to a reasonable level can be quite slow. If the network ceiling were about 2,000 packets, then the normal sawtooth return to the ceiling after a loss would take 1,000 RTTs. This is slow, but the sender would still average 75% throughput, as we saw in *13.7 TCP and Bottleneck Link Utilization*. Perhaps more seriously, if the network ceiling were to double to 4,000 packets due to decreases in competing traffic, it would take the sender an additional 2,000 RTTs to reach the point where the link was saturated.

In the following diagram, the network ceiling and the ideal TCP sawtooth are shown in green. The ideal TCP sawtooth should range between 50% and 100% of the ceiling; in the diagram, "noise" or non-congestive losses occur at the red x's, bringing down the throughput to a much lower average level.



TCP without (green) and with (red) random losses
In this diagram, red random losses occur 3-4 times as often as green congestion losses

14.10 The Lossy-Link TCP Problem

Closely related to the high-bandwidth problem is the lossy-link problem, where one link on the path has a relatively high **non-congestive-loss** rate; the classic example of such a link is Wi-Fi. If TCP is used on a path with a 1.0% loss rate, then *14.5 TCP Reno loss rate versus cwnd* indicates that the sender can expect an average `cwnd` of only about 12, no matter how high the bandwidth×delay product is.

The only difference between the lossy-link problem and the high-bandwidth problem is one of scale; the lossy-link problem involves unusually large values of p while the high-bandwidth problem involves circumstances where p is quite low *but not low enough*. For a given non-congestive loss rate p, if the bandwidth×delay product is much in excess of $1.22/\sqrt{p}$ then the sender will be unable to maintain a `cwnd` close to the network ceiling.

14.11 The Satellite-Link TCP Problem

A third TCP problem, only partially related to the previous two, is that encountered by TCP users with very long RTTs. The most dramatic example of this involves satellite Internet links (*3.9.2 Satellite Internet*). Communication each way involves routing the signal through a satellite in geosynchronous orbit; a round trip involves four up-or-down trips of ~36,000 km each and thus has a propagation delay of about 500ms. If we take the per-user bandwidth to be 1 Mbps (satellite ISPs usually provide quite limited bandwidth, though peak bandwidths can be higher), then the bandwidth×delay product is about 40 packets. This is not especially high, even when typical queuing delays of another ~500ms are included, but the fact that it takes many seconds to reach even a moderate `cwnd` is an annoyance for many applications. Most ISPs provide an "acceleration" mechanism when they can identify a TCP connection as a file download; this usually involves transferring the file over the satellite portion of the path using a proprietary protocol. However, this is not

much use to those using TCP connections that involve multiple bidirectional exchanges; *eg* those using VPN connections.

14.12 Epilog

TCP Reno's core congestion algorithm is based on algorithms in Jacobson and Karel's 1988 paper *[JK88]*, now twenty-five years old. There are concerns both that TCP Reno uses too much bandwidth (the greediness issue) and that it does not use enough (the high-bandwidth-TCP problem).

In the next chapter we consider alternative versions of TCP that attempt to solve some of the above problems associated with TCP Reno.

14.13 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercise 2.5 is distinct, for example, from exercises 2.0 and 3.0. Exercises marked with a ♢ have solutions or hints at 24.12 Solutions for Dynamics of TCP Reno.

0.5. In the section *14.2.3 Example 3: competition and queue utilization*, we derived the formula

$$Q = w_A + w_B - 2d - 2(\alpha d_A + \beta d_B)$$

under the assumption that the bottleneck bandwidth was 1 packet per unit time. Give the formula when the bottleneck bandwidth is r packets per unit time. Hint: the formula above will apply if we measure time in units of 1/r; only the delays d, d_A and d_B need to be re-scaled to refer to "normal" time. A delay d measured in "normal" time corresponds to a delay $d' = r \times d$ measured in 1/r units.

1.0. Consider the following network, where the bandwidths marked are all in packets/ms. C is sending to D using sliding windows and A and B are idle.

Suppose the propagation delay on the 100 packet/ms links is 1 ms, and the propagation delay on the R1–R2 link is 2 ms. The RTT_{noLoad} for the C–D path is thus about 8 ms, for a bandwidth×delay product of 40 packets. If C uses winsize = 50, then the queue at R1 will have size 10.

Now suppose A starts sending to B using sliding windows, also with winsize = 50. What will be the size of the queue at R1?

Hint: by symmetry, the queue will be equally divided between A's packets and C's, and A and C will each see a throughput of 2.5 packets/ms. RTT_{noLoad}, however, does not change. The number of packets in transit for each connection will be 2.5 packets/ms × RTT_{noLoad}.

2.0. In the previous exercise, give the average number of **data** packets in transit on each individual link:

(a).◇ for the original case in which C is the only sender, with winsize = 50 (the only active links here are C–R1, R1–R2 and R2–D).
(b). for the new case in which B is also sending, also with winsize = 50. In this case all links are active.

Each link will also have an equal number of **ACK** packets in transit in the reverse direction. Hint: since winsize ⩾ bandwidth×delay, packets are sent at the bottleneck rate.

2.5.◇ Consider the following network, with links labeled with one-way propagation delays in milliseconds (so, ignoring bandwidth delay, A's RTT_{noLoad} is 40 ms and B's is 20 ms). The bottleneck link is R–D, with a bandwidth of 6 packets/ms.

Initially B sends to D using a winsize of 120, the bandwidth×round-trip-delay product for the B–D path. A then begins sending as well, increasing its winsize until its share of the bandwidth is 2 packets/ms.

What is A's winsize at this point? How many packets do A and B each have in the queue at R?

It is perhaps easiest to solve this by repeated use of the observation that the number of packets in transit on a connection is always equal to RTT_{noLoad} times the actual bandwidth received by that connection. The algebraic methods of *14.2.3 Example 3: competition and queue utilization* can also be used, but bandwidth there was normalized to 1; all propagation delays given here would therefore need to be multiplied by 6.

3.0. Consider the C–D path from the diagram of *14.2.4 Example 4: cross traffic and RTT variation*:

Link numbers are bandwidths in packets/ms. Assume C is the only sender.

(a).◇ Give propagation delays for the links C–R1 and R2–D so that there will be an average of 5 packets in transit on the C–R1 and R2–D links, in each direction, if C uses a winsize sufficient to saturate the bottleneck R1–R2 link.
(b). Give propagation delays for all three links so that, when C uses a winsize equal to the round-trip transit capacity, there are 5 packets each way on the C–R1 link, 10 on the R1–R2 link, and 20 on the R2–D link.

4.0. Suppose we have the network layout below of *14.2.4 Example 4: cross traffic and RTT variation*, except that the R1–R2 bandwidth is 6 packets/ms and the R2–R3 bandwidth is 3. Suppose also that A and C

have settled upon window sizes so that each contributes 30 packets to R1's queue – and thus each has 50% of the bandwidth. R2 will then be sending 3 packets/ms to R3 and so will have no queue.

Now A's winsize is incremented by 10, initially, at least, leading to A contributing more than 50% of R1's queue. When the steady state is reached, how will these extra 10 packets be distributed between R1 and R2? Hint: As A's winsize increases, A's overall throughput cannot rise due to the bandwidth restriction of the R2–R3 link.

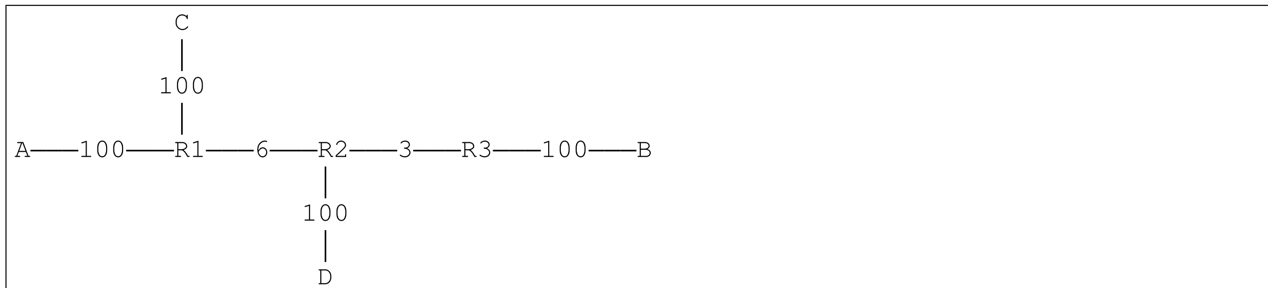


Diagram for exercises 4 and 5.0

5.0. Suppose we have the network layout above, similar to *14.2.4 Example 4: cross traffic and RTT variation*, for which we are given that the round-trip C–D RTT_{noLoad} is 5 ms. The round-trip A–B RTT_{noLoad} may be different.

The R1–R2 bandwidth is 6 packets/ms, so with A idle the C–D throughput is 6 packets/ms.

(a). Suppose that A and C have window sizes such that, with *both* transmitting, each has 30 packets in the queue at R1. What is C's winsize? Hint: C's throughput is now 3 packets/ms.

(b). Now suppose C's winsize, with A idle, is 60. In this case the C–D transit capacity would be 5 ms × 6 packets/ms = 30 packets, and so C would have 60–30 = 30 packets in R1's queue. A then begins sending, with a winsize chosen so that A and C's contributions to R1's queue are equal; C's winsize remains at 60. What will be C's (and thus A's) queue usage at R1? Hint: find the transit capacity for a throughput of 3 packets/ms.

(c). Suppose the A–B RTT_{noLoad} is 10 ms. If C's winsize is 60, find the winsize for A that makes A and C's contributions to R1's queue equal.

6.0. One way to address the reduced bandwidth TCP Reno gives to long-RTT connections is for all connections to use an increase increment of RTT^2 instead of 1; that is, everyone uses AIMD(RTT^2,1/2) instead of AIMD(1,1/2) (or AIMD($k \times RTT^2$,1/2), where k is an arbitrary scaling factor that applies to everyone).

(a). Construct a table in the style of of *14.3.2 Example 3: Longer RTT* above, showing the result of two connections using this strategy, where one connection has RTT = 1 and the other has RTT = 2. Start the connections with `cwnd`=RTT^2, and assume a loss occurs when $cwnd_1 + cwnd_2 > 24$.

(b). Explain why this strategy might not be desirable if one connection is over a direct LAN with an RTT of 1 ms, while the second connection has a very long path and an RTT of 1.0 sec.

7.0. For each value α or β below, find the other value so that AIMD(α,β) is TCP-friendly.

(a). β = 1/5
(b). β = 2/9
(c). α = 1/5

Then pick the pair that has the smallest α, and draw a sawtooth diagram that is approximately proportional: α should be the slope of the linear increase, and β should be the decrease fraction at the end of each tooth.

8.0. Suppose two TCP flows compete. The flows have the same RTT. The first flow uses AIMD(α_1,β_1) and the second uses AIMD(α_2,β_2); neither flow is necessarily TCP-Reno-friendly. The two connections, however, compete fairly with one another; that is, they have the same average packet-loss rates. Show that

$$\alpha_1/\beta_1 = (2-\beta_2)/(2-\beta_1) \times \alpha_2/\beta_2.$$

Assume regular losses, and use the methods of *14.7 AIMD Revisited*. Hint: first, apply the argument there to show that the two flows' teeth must have the same width w and same average height. The average height is no longer 3w/2, but can still be expressed in terms of w, α and β. Use a diagram to show that, for any tooth, average_height = h×(1–β/2), with h the right-edge height of the tooth. Then equate the two average heights of the h_1/β_1 and h_2/β_2 teeth. Finally, use the $\alpha_i w = \beta_i h_i$ relationships to eliminate h_1 and h_2.

8.5. Using the result of the previous exercise, show that AIMD(α_1,β) is equivalent to (in the sense of competing fairly with) AIMD(α_2,0.5), with $\alpha_2 = \alpha_1 \times (2-\beta)/3\beta$.

9.0. Suppose two 1KB packets are sent as part of a packet-pair probe, and the minimum time measured between arrivals is 5 ms. What is the estimated bottleneck bandwidth?

10.0. Consider the following three causes of a 1-second network delay between A and B. In all cases, assume ACKs travel instantly from B back to A.

> (i) An intermediate router with a 1-second-per-packet bandwidth delay; all other bandwidth delays negligible
>
> (ii) An intermediate link with a 1-second propagation delay; all bandwidth delays negligible
>
> (iii) An intermediate router with a 100-ms-per-packet bandwidth delay, and a steadily replenished queue of 10 packets, from another source (as in the diagram in *14.2.4 Example 4: cross traffic and RTT variation*).

What would happen in each of these cases if the packet-pair technique (*14.2.6 Packet Pairs*) were used to measure the bandwidth? What would be the values of the measured bandwidths?

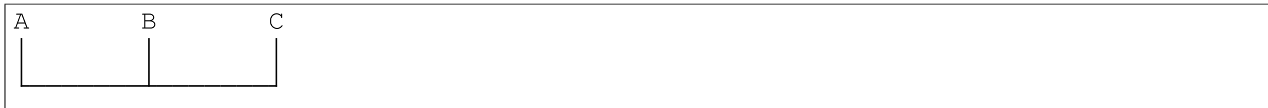
11.0. Consider again the three-link parking-lot network from *14.4.1 Max-Min Fairness*:

(a). Suppose we have *two* end-to-end connections, in addition to one single-link connection for each link.

Find the max-min-fair allocation.

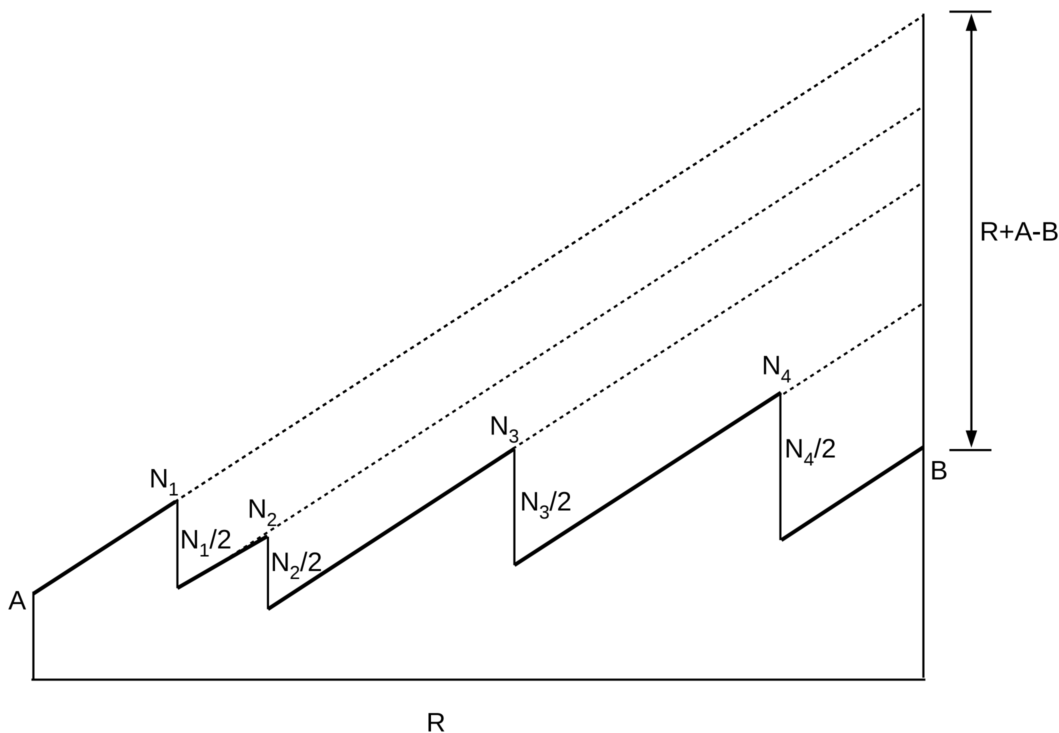
(b). Suppose we have a single end-to-end connection, and one B–C and C–D connection, but two A–B connections. Find the max-min-fair allocation.

12.0. Consider the two-link parking-lot network:



Suppose there are two A–C connections, one A–B connection and one A–C connection. Find the allocation that is proportionally fair.

13.0. Suppose we use TCP Reno to send K packets over R RTT intervals. The transmission experiences n not-necessarily-uniform loss events; the TCP `cwnd` graph thus has n sawtooth peaks of heights N_1 through N_n. At the start of the graph, `cwnd` = A, and at the end of the graph, `cwnd` = B. Show that the sum N_1 + ... + N_n is 2(R+A-B), and in particular the average tooth height is independent of the distribution of the loss events.



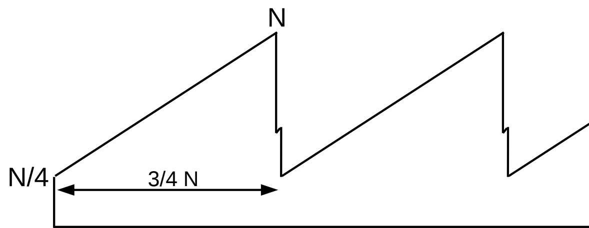
13.5. Suppose the bandwidth×delay product for a network path is 1000 packets. The only traffic on the path is from a single TCP Reno connection. For each of the following cases, find the average `cwnd` and the approximate number of packets between losses (the reciprocal of the loss rate). Your answers, collectively, should reflect the formula in *14.5 TCP Reno loss rate versus cwnd*.

(a).◇ The bottleneck queue capacity is close to zero.

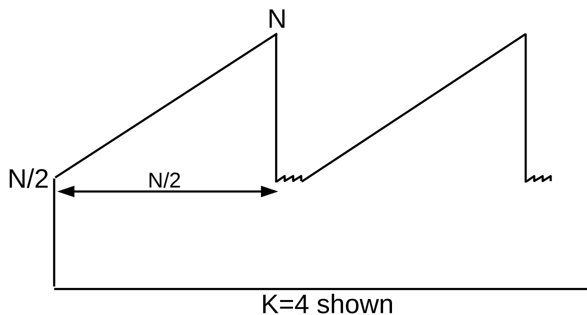
(b). The bottleneck queue capacity is 1000 packets.

(c). The bottleneck queue capacity is 3000 packets.

14.0. Suppose TCP Reno has regularly spaced sawtooth peaks of the same height, but the packet losses come in pairs, with just enough separation that both losses in a pair are counted separately. N is large enough that the spacing between the two losses is negligible. The net effect is that each large-scale tooth ranges from height N/4 to N. As in *14.5 TCP Reno loss rate versus cwnd*, $\texttt{cwnd}_{\text{mean}} = K/\sqrt{p}$ for some constant K. Find the constant. Hint: from the given information one can determine both $\texttt{cwnd}_{\text{mean}}$ and the number of packets sent in one tooth. The loss rate is p = 2/(number of packets sent in one tooth).

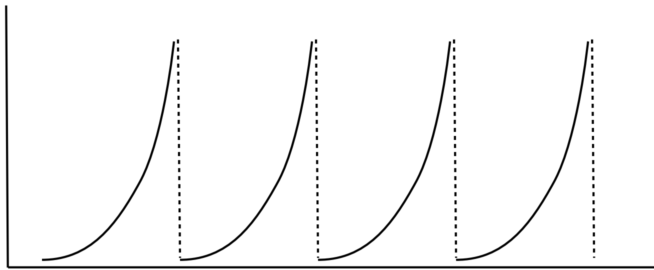


15.0. As in the previous exercise, suppose a TCP transmission has large-scale teeth of height N. Between each pair of consecutive large teeth, however, there are K-1 additional losses resulting in K-1 additional tiny teeth; N is large enough that these tiny teeth can be ignored. A non-Reno variant of TCP is used, so that between these tiny teeth `cwnd` is assumed not to be cut in half; during the course of these tiny teeth `cwnd` does not change much at all. The large-scale tooth has width N/2 and height ranging from N/2 to N, and there are K losses per large-scale tooth. Find the ratio $\texttt{cwnd}/\sqrt{p}$, in terms of K. When K=1 your answer should reduce to that derived in *14.5 TCP Reno loss rate versus cwnd*.



16.0. Suppose a TCP Reno tooth starts with `cwnd` = c, and contains N packets. Let w be the width of the tooth, in RTTs as usual. Show that $w = (c^2 + 2N)^{1/2} - c$. Hint: the maximum height of the tooth will be c+w, and so the average height will be c + w/2. Find an equation relating c, w and N, and solve for w using the quadratic formula.

16.5. Suppose we have a non-Reno implementation of TCP, in which the formula relating the `cwnd` c to the time t, as measured in RTTs since the most recent loss event, is $c = t^2$ (versus TCP Reno's $c = c_0 + t$). The sawtooth then looks like the following:



The number of packets sent in a tooth of width T RTTs, which is the reciprocal of the loss rate p, is now approximately $k \times T^3$ (this may be accepted on faith, or by integrating t^2 from 0 to T). The average `cwnd` is therefore $k \times T^3/T = k \times T^2$.

Derive a formula expressing the average `cwnd` in terms of the loss rate p. Hint: the exponent for p should be –2/3, versus –1/2 in the formula in *14.5 TCP Reno loss rate versus cwnd*.

16.6. Using the TCP assumptions of exercise 16.5 above, `cwnd` is incremented by about 2t per each RTT. Show that the `cwnd` increment rule can be expressed as

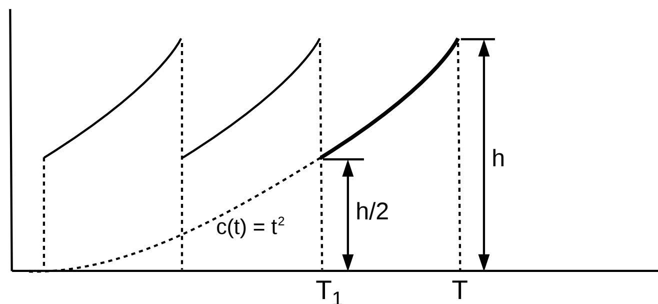
$$\texttt{cwnd} \mathrel{+}= \alpha \texttt{cwnd}^{1/2}$$

and find the value of α.

16.7. Using the same TCP assumptions as in exercise 16.5 above, show that `cwnd` is still proportional to $p^{-2/3}$, where p is the loss rate, assuming the following:

- the top boundary of each tooth follows the curve `cwnd` = $c(t) = t^2$, as before.
- each tooth has a right boundary at t=T and a left boundary at $t=T_1$, where $c(T_1) = 0.5 \times c(T)$.

(In the previous exercise we assumed, in effect, that $T_1 = 0$ and that `cwnd` dropped to 0 after each loss event; here we assume multiplicative decrease is in effect with β=1/2.) The number of packets sent in one tooth is now $k \times (T^3 - T_1{}^3)$, and the mean `cwnd` is this divided by $T–T_1$.



Note that as the teeth here become higher, they become proportionately narrower. Hint: show $T_1 = (0.5)^{0.5} \times T$, and then eliminate T_1 from the above equations.

17.0. Suppose in a TCP Reno run each packet is equally likely to be lost; the number of packets N in each tooth will therefore be distributed exponentially. That is, N = -k log(X), where X is a uniformly distributed random number in the range 0<X<1 (k, which does not really matter here, is the mean interval between

losses). Write a simple program that simulates such a TCP Reno run. At the end of the simulation, output an estimate of the constant C in the formula $cwnd_{mean} = C/\sqrt{p}$. You should get a value of about 1.31, as in the formula in *14.5.1 Irregular teeth*.

Hint: There is no need to simulate packet transmissions; we simply create a series of teeth of random size, and maintain running totals of the number of packets sent, the number of RTT intervals needed to send them, and the number of loss events (that is, teeth). After each loss event (each tooth), we update:

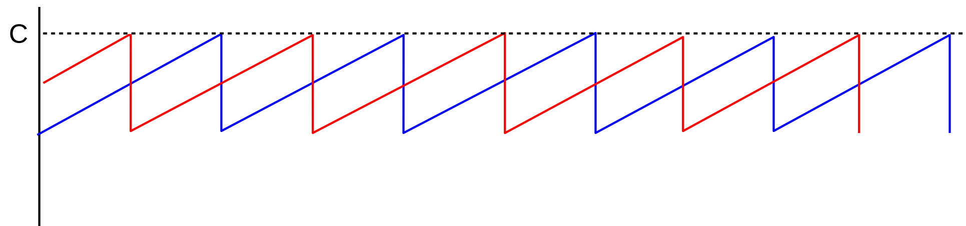
- total_packets += *packets sent in this tooth*
- RTT_intervals += *RTT intervals in this tooth*
- loss_events += 1 (one tooth = one loss event)

If a loss event marking the end of one tooth occurs at a specific value of `cwnd`, the next tooth begins at height c = `cwnd`/2. If N is the random value for the number of packets in this tooth, then by the previous exercise the tooth width in RTTs is $w = (c^2 + 2N)^{1/2} - c$; the next peak (that is, loss event) therefore occurs when `cwnd` = c+w. Update the totals as above and go on to the next tooth. It should be possible to run this simulation for 1 million teeth in modest time.

18.0. Suppose two TCP connections have the same RTT and share a bottleneck link, for which there is no other competition. The size of the bottleneck queue is negligible when compared to the bandwidth × RTT_{noLoad} product. Loss events occur at regular intervals.

In Exercise 12.0 of the previous chapter, you were to show that if losses are synchronized then the two connections together will use 75% of the total bottleneck-link capacity

Now assume the two TCP connections have no losses in common, and, in fact, *alternate* losses at regular intervals as in the following diagram.



Both connections have a maximum `cwnd` of C. When Connection 1 experiences a loss, Connection 2 will have `cwnd` = 75% of C, and vice-versa.

(a). What is the combined transit capacity of the paths, in terms of C? (Because the queue size is negligible, the transit capacity is approximately the sum of the `cwnd`s at the point of loss.)

(b). Find the bottleneck-link utilization. Hint: Again because the queue size is negligible, this is approximately the ratio of the average total cwnd to the transit capacity of part (a). It should be at least 85%.

Since the rise of TCP Reno, several TCP alternatives to Reno have been developed; each attempts to address some perceived shortcoming of Reno. While many of them are very specific attempts to address the high-bandwidth problem we considered in *14.9 The High-Bandwidth TCP Problem*, some focus primarily or entirely on other TCP Reno foibles. One such issue is TCP Reno's "greediness" in terms of queue utilization; another is the lossy-link problem (*14.10 The Lossy-Link TCP Problem*) experienced by, say, Wi-Fi users.

Generally speaking, a TCP implementation can respond to congestion at the cliff – that is, it can respond to packet losses – or can respond to congestion at the knee – that is, it can detect the increase in RTT associated with the filling of the queue. These strategies are sometimes referred to as **loss-based** and **delay-based**, respectively; the latter term because of the rise in RTT. TCP implementers can tweak both the loss response – the multiplicative decrease of TCP Reno – and also the way TCP increases its `cwnd` in the absence of loss. There is a rich variety of options available.

The concept of monitoring the RTT to avoid congestion at the knee was first introduced in TCP Vegas (*15.6 TCP Vegas*). One striking feature of TCP Vegas is that, in the absence of competition, the queue may never fill, and thus there may not be any congestive losses. The TCP sawtooth, in other words, is not inevitable.

When losses do occur, most of the mechanisms reviewed here continue to use the TCP NewReno recovery strategy. As most of the implementations here are relatively recent, the senders can generally expect that the receiving end will support SACK TCP, which allows more rapid recovery from multiple losses.

15.1 Choosing a TCP on linux

On linux systems, the TCP congestion-control mechanism can be set by writing an appropriate string to `/proc/sys/net/ipv4/tcp_congestion_control` (or, equivalently, by passing the string as a parameter to the `sysctl net.ipv4.tcp_congestion_control` command). The standard options on the author's system as of 2013 are listed below (as of 2016, several are now only available if loaded explicitly, *eg* with `modprobe`). The list comes from `/proc/sys/net/ipv4/tcp_available_congestion_control`.

- highspeed
- htcp
- hybla
- illinois
- vegas
- veno
- westwood
- bic
- cubic

We review several of these below; see *15.4 A Roadmap* for an overview. TCP Cubic is currently (2013) the default linux congestion-control implementation; TCP Bic was a precursor.

The TCP congestion-control mechanism can also be set on a per-connection basis. Non-root users can select any mechanism listed in `/proc/sys/net/ipv4/tcp_allowed_congestion_control`; entries in tcp_available_congestion_control can be copied to this by the root user.

Many TCP flavors are not available by default, but can be loaded via `modprobe`. The modules containing the TCP implementations are generally in /lib/modules/$(uname -r)/kernel/net/ipv4. Executing `ls tcp_*` in this directory yields (on the author's system in 2017) the following:

- tcp_bic.ko
- tcp_cdg.ko
- tcp_dctcp.kp
- tcp_highspeed.ko
- tcp_htcp.ko
- tcp_hybla.ko
- tcp_illinois.ko
- tcp_scalable.ko
- tcp_vegas.ko
- tcp_veno.ko
- tcp_westwood.ko
- tcp_yeah.ko

To load, *eg*, TCP Vegas, use `modprobe tcp_vegas` (without the ".ko"). This will last until the next reboot (or until the module is manually unloaded). At this point /proc/sys/net/ipv4/tcp_available_congestion_control will contain "vegas" (not tcp_vegas).

In the C language, we can select the linux congestion control mechanism, after socket creation but before connection, by including the `setsockopt()` call below; see *22.2.2 An Actual Stack-Overflow Example* and *22.10.3 A TLS Programming Example* for complete C examples (though without this call).

```
#include <netinet/in.h>
#include <netinet/tcp.h>
...
char * cong_algorithm = "vegas";
int slen = strlen( cong_algorithm ) + 1;
int rc = setsockopt( sock, IPPROTO_TCP, TCP_CONGESTION, cong_algorithm, slen);
if (rc < 0) { /* error */ }
```

Checking the return code is essential to determine if the algorithm request succeeded.

In Python3 (and Python2) we can do this as well; the file below is also available at tcp_stalkc_cong.py. See also *18.6.1.1 sender.py*.

```
#!/usr/bin/python3
# stalk client allowing specification of the TCP congestion algorithm

from socket import *
from sys import argv

default_host = "localhost"
portnum = 5431

cong_algorithm = 'vegas'

def talk():
    global cong_algorithm
    rhost = default_host
    if len(argv) > 1:
        rhost = argv[1]
    if len(argv) > 2:
        cong_algorithm = argv[2]
    print("Looking up address of " + rhost + "...", end="")
    try:
        dest = gethostbyname(rhost)
    except gaierror as mesg:    # host not found
        errno,errstr=mesg.args
        print("\n   ", errstr);
        return;
    print("got it: " + dest)
    addr=(dest, portnum)
    s = socket()

    TCP_CONGESTION = 13   # defined in /usr/include/netinet/tcp.h
    cong = bytes(cong_algorithm, 'ascii')
    try:
       s.setsockopt(IPPROTO_TCP, TCP_CONGESTION, cong)
    except OSError as mesg:
       errno, errstr = mesg.args
       print ('congestion mechanism {} not available: {}'.format(cong_algorithm, errstr))
       return

    res=s.connect_ex(addr)      # make the actual connection
    if res!=0:
        print("connect to port ", portnum, " failed")
        exit()

    while True:
        try:
            buf = input("> ")
        except:
            break;
        buf = buf + "\n"
        s.send(bytes(buf, 'ascii'))

talk()
```

As of version 3.5, Python did not define the constant `TCP_CONGESTION`; the value 13 above was found in the C include file mentioned in the comment. Fortunately, Python simply passes the parameters of `s.setsockopt()` to the underlying C call, and everything works. Supposedly `TCP_CONGESTION` is pre-defined in Python 3.6.

15.2 High-Bandwidth Desiderata

One goal of all TCP implementations that attempt to fix the high-bandwidth problem is to be **un**fair to TCP Reno: the whole point is to allow `cwnd` to increase more aggressively than is permitted by Reno. Beyond that, let us review what else a TCP version should do.

First is the **backwards-compatibility** constraint: any new TCP should exhibit reasonable **fairness** with TCP Reno at lower bandwidth×delay products. In particular, it should not ever have a significantly lower `cwnd` than a competing TCP Reno would get. But also it should not take bandwidth unfairly from a TCP Reno connection: the above comment about unfairness to Reno notwithstanding, the new TCP, when competing with TCP Reno, should leave the Reno connection with about the same bandwidth it would have if it were competing with another Reno connection. This is possible because at higher bandwidth×delay products TCP Reno does not efficiently use the available bandwidth; the new TCP should to the extent possible restrict itself to consuming this previously unavailable bandwidth rather than eating significantly into the bandwidth of a competing TCP Reno connection.

There is also the **self-fairness** issue: multiple connections using the new TCP should receive similar bandwidth allocations, at least with similar RTTs. For dissimilar RTTs, the bandwidth proportions should ideally be no worse than they would be under TCP Reno.

Ideally, we also want relatively **rapid convergence** to fairness; fairness is something of a hollow promise if only connections transferring more than a gigabyte will benefit from it. For TCP Reno, two connections halve the difference in their respective `cwnds` at each shared loss event; as we saw in *14.7.1 AIMD and Convergence to Fairness*, slower convergence is possible.

It is harder to hope for fairness between competing new implementations. However, at the very least, if new implementations tcp1 and tcp2 are competing, then neither should get less than TCP Reno would get.

Some new TCPs make use of careful RTT measurements, and, as we shall see below, such measurements are subject to a considerable degree of noise. Any new TCP implementation should be reasonably **robust** in the face of inaccuracies in RTT measurement; a modest or transient measurement error should not make the protocol behave badly, in either the direction of low `cwnd` or of high.

Finally, a new TCP should ideally try to avoid clusters of **multiple losses** at each loss event. Such multiple losses, for example, are a problem for TCP NewReno without SACK: as we have seen, it takes one RTT to retransmit each lost packet. Even with SACK, multiple losses complicate recovery. Yet if a new TCP increments `cwnd` by an amount N>1 after each RTT, then there is potential for the network ceiling to be exceeded by N within one RTT, making a cluster of N losses reasonably likely to occur. These losses are likely distributed among all connections, not just the new-TCP one.

All TCPs addressing the high-bandwidth issue will need a `cwnd`-increment N that is fairly large, at least some of the time, apparently conflicting with this no-multiple-losses ideal. One trick is to reduce N when packet loss appears to be imminent. TCP Illinois and TCP Cubic do have mechanisms in place to reduce multiple losses.

15.3 RTTs

The exact performance of some of the faster TCPs we consider – for that matter, the exact performance of TCP Reno – is influenced by the RTT. This may affect individual TCP performance and also competition between different TCPs. For reference, here are a few typical RTTs from Chicago to various other places:

- US West Coast: 50-100 ms
- Europe: 100-150 ms
- Southeast Asia: 100-200 ms

15.4 A Roadmap

We start with Highspeed TCP, an early and relatively simple attempt to address the high-bandwidth-TCP problem.

After that is the group TCP Vegas, FAST TCP, TCP Westwood, TCP Illinois and Compound TCP. These all involve so-called **delay-based** congestion control, in which the sender carefully monitors the RTT for the minute increases that signal queuing. TCP Vegas, which dates from 1995, is the earliest TCP here and in fact predates widespread recognition of the high-bandwidth-TCP problem. Its goal – then and now – was to prove that one could build a TCP that, in the absence of competition, could transfer arbitrarily long streams of data with no losses and with 100% bottleneck-link utilization.

The next group, consisting of TCP Veno, TCP Hybla and DCTCP, represent special-purpose TCPs. While TCP Veno may be a reasonable high-bandwidth TCP candidate, its primary goal is to improve TCP performance over lossy links such as Wi-Fi. TCP Hybla is targeted at satellite-Internet users with very long RTTs while DCTCP is for internal connections within a data center (which, among other things, have very short RTTs).

The last triad represents newer, non-delay-based attempts to solve the high-bandwidth-TCP problem: H-TCP, TCP Cubic and TCP BBR. TCP Cubic has become the default TCP on linux.

15.5 Highspeed TCP

An early proposed fix for the high-bandwidth-TCP problem is HighSpeed TCP, documented in **RFC 3649** (Floyd, 2003). Highspeed TCP is sometimes called HS-TCP, but we use the longer name here to avoid confusion with the entirely unrelated H-TCP, below.

Highspeed TCP adjusts the additive-increase and multiplicative-decrease parameters α and β so that, for larger values of `cwnd`, the rate of `cwnd` increase between losses is much faster, and the `cwnd` decrease at loss events is much smaller. This allows efficient use of all the available bandwidth for large bandwidth×delay products. Correspondingly, when `cwnd` is in the range where TCP Reno works well, Highspeed TCP's throughput is only modestly larger than TCP Reno's, so the two compete relatively fairly.

The threshold for Highspeed TCP diverging from TCP Reno is a loss rate less than 10^{-3}, which for TCP Reno occurs when `cwnd` = 38. Beyond that point, Highspeed TCP gradually increases α and decreases β. The overall effect is to outperform TCP Reno by a factor N = N(`cwnd`) according to the table below. This N

can also be interpreted as the "unfairness" of Highspeed TCP with respect to TCP Reno; fairness is arguably "close to" 1.0 until cwnd⩾1000, at which point TCP Reno is likely not using the full bandwidth available due to the high-bandwidth TCP problem.

cwnd	N(cwnd)
1	1.0
10	1.0
100	1.4
1,000	3.6
10,000	9.2
100,000	23.0

An algebraic expression for N(cwnd), for N⩾38, is

$$N(\text{cwnd}) = 0.23\times\text{cwnd}^{0.4}$$

At cwnd=38 this is about 1.0; for smaller cwnd we stick with N=1.

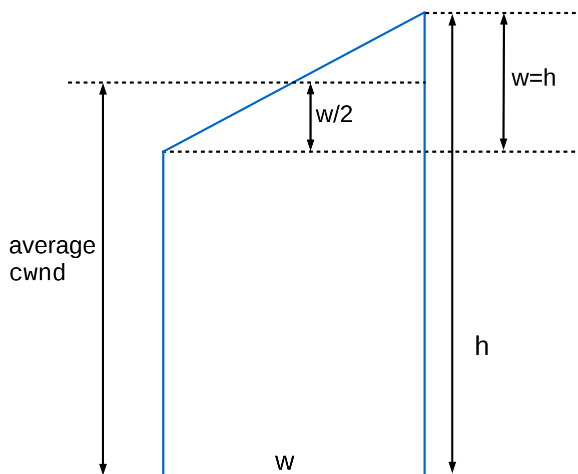
To specify the details of Highspeed TCP, we start by considering a 10 Gbps link, which was the fastest generally available at the time Highspeed TCP was developed. If the RTT is 100 ms, then the bandwidth×delay product works out to 83,000 packets. The central strategy of Highspeed TCP is to *choose* the desired loss rate for an average cwnd of 83,000 to be 1 packet in 10^7; this number was empirically determined. This is quite a bit larger than the corresponding TCP Reno loss rate of 1 packet in 5×10^9 (*14.9 The High-Bandwidth TCP Problem*); in this context, a larger congestion loss rate is better. The loss rate is the reciprocal of the tooth area; it turns out (below) that we have a great deal of latitude in choosing the tooth area by adjusting the α and β window-growth parameters. After determining α and β for cwnd = 83,000, Highspeed TCP then uses interpolation to cover cwnd values in between 38 and 83,000. (The Highspeed TCP rules do extend to larger cwnds too, but there is not necessarily an expectation that they will work well there.)

We start with the TCP Reno relationship $\text{cwnd} = 1.225\times p^{-0.5}$, from *14.5 TCP Reno loss rate versus cwnd* (**RFC 3649** uses a numerator of 1.20 in this formula.) We fit the relationship $\text{cwnd} = k\times p^{-\alpha}$ to the above two pairs of (cwnd,p) values, $(38,10^{-3})$ and $(83000,10^{-7})$. This turns out to yield

$$\text{cwnd} = 0.12 \times p^{-0.835}$$

From this we can derive the TCP Reno multiplier N(cwnd) above, by using the TCP Reno relationship $\text{cwnd} = 1.2\times N\times p^{-0.5}$ for N synchronized connections, eliminating p and then solving for N.

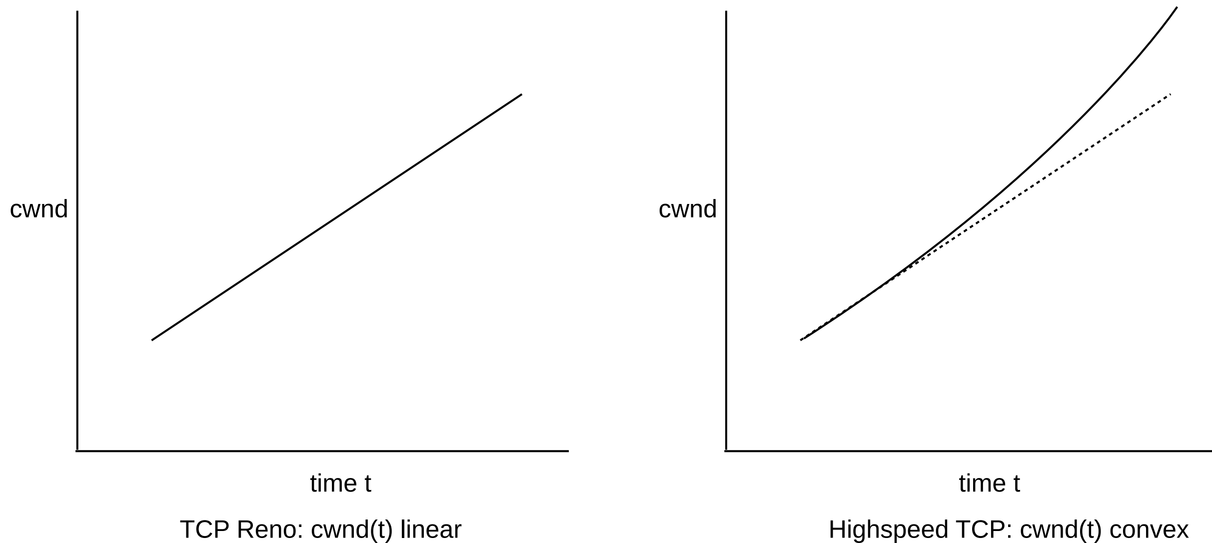
The next step is to define the additive-increase and multiplicative-decrease values $\alpha = \alpha(\text{cwnd})$ and $\beta = \beta(\text{cwnd})$, thus allowing us to build an actual implementation. While α and β are allowed to vary with cwnd, we will assume they do so only slowly, so that for any given steady-state connection the α values are relatively constant (the β value is that at the maximum cwnd). This gives us a standard AIMD tooth:



When cwnd is 83,000 we want the loss rate to be 10^{-7}, meaning that the area of the tooth, w×cwnd = w×h×(1-β/2), should be 10^7. From this we get w = 10^7/83,000 = 120.5 RTTs. We also have, very generally, αw = βh, and combining this with cwnd = h×(1-β/2), we get α = βh/w = cwnd×(2β/(1-β/2))/w $\simeq$ 1378×β/(1-β/2). **RFC 3649** suggests β=0.1 at this cwnd, making α = 73. The value of β for values of cwnd between 38 and 83,000 is determined by logarithmic interpolation between 0.5 and 0.1; the corresponding value of α(cwnd) is then set by the formula.

The 1-in-10^7 loss rate – corresponding to a bit error rate of about one in 1.2×10^{11} – is large enough that it is at least two orders of magnitude higher than the rate of noise-induced non-congestive packet losses. On the other hand, it is small enough that the Highspeed TCP derived from it competes reasonably fairly with TCP Reno, at least with bandwidth×delay products small enough that TCP Reno alone performs reasonably well.

It may be helpful to view Highspeed TCP in terms of the cwnd graph between losses. For ordinary TCP, the graph increases linearly. For Highspeed TCP, the graph is slightly **convex** (lying above its tangent). This means that there is a modest increase in the *rate* of cwnd increase, as time goes on (up to the point of packet loss).



TCP Reno: cwnd(t) linear Highspeed TCP: cwnd(t) convex

This might be an appropriate time to point out that in TCP Reno, the `cwnd`-versus-**time** graph between losses is actually slightly **concave** (lying below its tangent). We do get a strictly linear graph if we plot `cwnd` as a function of the count of elapsed RTTs, but the RTTs are themselves slowly increasing as a function of time once the queue starts filling up. At that point, the `cwnd`-versus-time graph bends slightly down. If the bottleneck queue capacity matches the total path transit capacity, the RTTs for a full queue are about double the RTTs for an empty queue.

In general, when Highspeed-TCP competes with a new TCP Reno flow it is N times as aggressive, and grabs N times the bandwidth, where N = N(`cwnd`) is as above. For `cwnd` = 83,000, the formula above yields N = 21. This may be surprising, as for this value of `cwnd` Highspeed TCP is AIMD(73,0.1), which is equivalent to AIMD(459,0.5) (either via the formula above or by *14.13 Exercises*, exercise 8.0). We might naively suppose that AIMD(459,0.5) would out-compete TCP Reno – AIMD(1,0.5) – by a factor of 459, by the reasoning of *14.3.1 Example 2: Faster additive increase*. But this is true only if losses are synchronized, which, for such lopsided differences in α, is manifestly not the case. Because Highspeed TCP uses the lion's share of the queue, it encounters the lion's share of loss events, and TCP Reno is able to do much better than the α values alone would suggest.

Finally, with a little math we can compare Highspeed TCP with an AIMD-type flavor of TCP with an additive-increase rule (per RTT) of the form

$$\texttt{cwnd} \mathrel{+}= \alpha \times \texttt{cwnd}^k$$

For TCP Reno, k=0, and in the example of exercises 16.5 and 16.6 of *14.13 Exercises* we have k=1/2. For compatibility with Highspeed TCP, it turns out what we need is k=0.8. We will return to this in *15.10 Compound TCP*, which intentionally mimics the behavior of Highspeed TCP when queue utilization is low.

15.6 TCP Vegas

TCP Vegas, introduced in *[BP95]*, is the only new TCP version we consider here that dates from the previous century. The goal was not directly to address the high-bandwidth problem, but rather to improve TCP throughput generally; indeed, in 1995 the high-bandwidth problem had not yet surfaced as a practical con-

cern. The ambitious goal of TCP Vegas is essentially to eliminate congestive losses, and to try to keep the bottleneck link 100% utilized at all times. Recall from *13.7 TCP and Bottleneck Link Utilization* that, with a large queue, the average bottleneck-link utilization for TCP Reno can be as low as 75%.

TCP Vegas achieves this improvement by, like DECbit, recognizing TCP congestion at the *knee*, that is, at the point where the bottleneck link has become saturated and further `cwnd` increases simply result in RTT increases. A TCP Vegas sender alone or in competition only with other TCP Vegas connections will seldom if ever approach the "cliff" where packet losses occur.

To accomplish this, no special router cooperation – or even receiver cooperation – is necessary. Instead, the sender uses careful monitoring of the RTT to keep track of the number of "extra packets" (*ie* packets sitting in queues) it has injected into the network. In the absence of competition, the RTT will remain constant, equal to RTT_{noLoad}, until `cwnd` has increased to the point when the bottleneck link has become saturated and the queue begins to fill (*6.3.2 RTT Calculations*). By monitoring the bandwidth as well, a TCP sender can even determine the actual number of packets in the bottleneck queue, as bandwidth $\times$ (RTT – RTT_{noLoad}). TCP Vegas uses this information to attempt to maintain at all times a small but positive number of packets in the bottleneck queue.

This TCP Vegas strategy is now often referred to as **delay-based congestion control**, as opposed to TCP Reno's **loss-based** congestion control. TCP Reno's periodic losses followed by the halving of `cwnd` is what leads to the "TCP sawtooth"; TCP Vegas, however, has no sawtooth.

A TCP sender can readily measure its throughput. The simplest measurement is `cwnd`/RTT as in *6.3.2 RTT Calculations*; this amounts to averaging throughput over an entire RTT. Let us denote this bandwidth estimate by BWE; for the time being we will accept BWE as accurate, though see *15.8.1 ACK Compression and Westwood+* below. TCP Vegas estimates RTT_{noLoad} by the minimum RTT (RTT_{min}) encountered during the connection. The "ideal" `cwnd` that just saturates the bottleneck link is $BWE \times RTT_{noLoad}$. Note that BWE will be much more volatile than RTT_{min}; the latter will typically reach its final value early in the connection, while BWE will fluctuate up and down with congestion (which will also act on RTT, but by increasing it).

As in *6.3.2 RTT Calculations*, any TCP sender can estimate queue utilization as

$$\text{queue_use} = \texttt{cwnd} - BWE \times RTT_{noLoad} = \texttt{cwnd} \times (1 - RTT_{noLoad}/RTT_{actual})$$

TCP Vegas then adjusts `cwnd` regularly to maintain the following:

$$\alpha \leqslant \text{queue_use} \leqslant \beta$$

which is the same as

$$BWE \times RTT_{noLoad} + \alpha \leqslant \texttt{cwnd} \leqslant BWE \times RTT_{noLoad} + \beta$$

Typically α = 2-3 packets and β = 4-6 packets. We increment `cwnd` by 1 if `cwnd` falls below the lower limit (*eg* if BWE has increased). Similarly, we decrement `cwnd` by 1 if BWE drops and `cwnd` exceeds $BWE \times RTT_{noLoad} + \beta$. These adjustments are conceptually done once per RTT. Typically a TCP Vegas sender would also set `cwnd = cwnd`/2 if a packet were actually lost, though this does not necessarily happen nearly as often as with TCP Reno.

TCP Vegas achieves its goal quite well. If one monitors the number of packets in queues, through real measurement or in simulation, the number does indeed stay between α and β. In the absence of competition from TCP Reno, a single TCP Vegas connection will *never* experience congestive packet loss. This is a remarkable achievement.

The use of returning ACKs to determine BWE is subject to errors due to "ACK compression", *15.8.1 ACK Compression and Westwood+*. This is generally not a major problem with TCP Vegas, however.

15.6.1 TCP Vegas versus TCP Reno

Despite its striking ability to avoid congestive losses in the absence of competition, TCP Vegas encounters a potentially serious fairness problem when competing with TCP Reno, at least for the case when queue capacity exceeds or is close to the transit capacity (*13.7 TCP and Bottleneck Link Utilization*). TCP Vegas will try to minimize its queue use, while TCP Reno happily fills the queue. And whoever has more packets in the queue has a proportionally greater share of bandwidth.

To make this precise, suppose we have two TCP connections sharing a bottleneck router R, the first using TCP Vegas and the second using TCP Reno. Suppose further that both connections have a path transit capacity of 10 packets, and R's queue capacity is 40 packets. If α=3 and β=5, TCP Vegas might keep an average of four packets in the queue. Unfortunately, TCP Reno then gobbles up most of the rest of the queue space, as follows. There are 40-4 = 36 spaces left in the queue after TCP Vegas takes its quota, and 10 in the TCP Reno connection's path, for a total of 46. This represents the TCP Reno connection's network ceiling, and is the point at which TCP Reno halves `cwnd`; therefore `cwnd` will vary from 23 to 46 with an average of about 34. Of these 34 packets, if 10 are in transit then 24 are in R's queue. If on average R has 24 packets from the Reno connection and 4 from the Vegas connection, then the bandwidth available to these connections will also be in this same 6:1 proportion. The TCP Vegas connection will get 1/7 the bandwidth, because it occupies 1/7 the queue, and the TCP Reno connection will take the other 6/7.

To put it another way, TCP Vegas is potentially too "civil" to compete with TCP Reno.

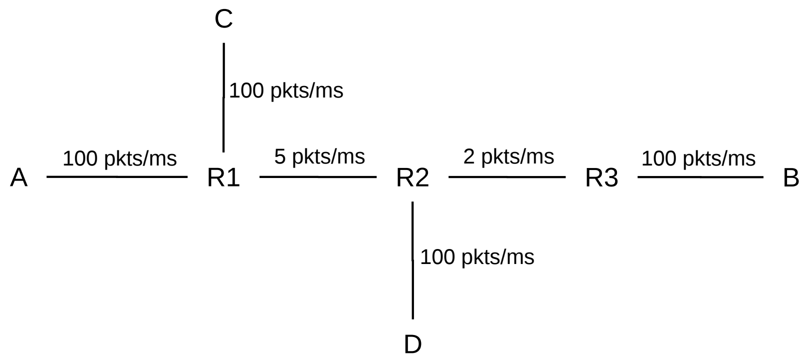
Even worse, Reno's aggressive queue filling will eventually force the TCP Vegas `cwnd` to decrease; see Exercise 4.0 below.

This Vegas-Reno fairness problem is most significant when the queue size is an appreciable fraction of the path transit capacity. During periods when the queue is empty, TCPs Vegas and Reno increase `cwnd` at the same rate, so when the queue size is small compared to the path capacity, TCP Vegas and TCP Reno are much closer to being fair.

In *16.5 TCP Reno versus TCP Vegas* we compare TCP Vegas with TCP Reno in simulation. With a transit capacity of 220 packets and a queue capacity of 10 packets, TCPs Vegas and Reno receive almost exactly the same bandwidth.

TCP Reno's advantage here assumes a router with a single FIFO queue. That advantage can disappear if a different queuing discipline is in effect. For example, if the bottleneck router used fair queuing (to be introduced in *19.5 Fair Queuing*) on a per-connection basis, then the TCP Reno connection's queue greediness would not be of any benefit, and both connections would get similar shares of bandwidth with the TCP Vegas connection experiencing lower delay. See *19.6.1 Fair Queuing and Bufferbloat*.

Let us next consider how TCP Vegas behaves when there is an increase in RTT due to the kind of cross traffic shown in *14.2.4 Example 4: cross traffic and RTT variation* and again in the diagram below. Let A–B be the TCP Vegas connection and assume that its queue-size target is 4 packets (*eg* α=3, β=5). We will also assume that the RTT_{noLoad} for the A–B path is about 5ms and the RTT for the C–D path is also low. As before, the link labels represent bandwidths in packets/ms, meaning that the round-trip A–B transit capacity is 10 packets.



Initially, in the absence of C–D traffic, the A–B connection will send at a rate of 2 packets/ms (the R2–R3 bottleneck), and maintain a queue of four packets at R2. Because the RTT transit capacity is 10 packets, this will be achieved with a window size of 10+4 = 14.

Now let the C–D traffic start up, with a winsize so as to keep about four times as many packets in R1's queue as A, once the new steady-state is reached. If all four of the A–B connection's "queue" packets end up now at R1 rather than R2, then C would need to contribute at least 16 packets. These 16 packets will add a delay of about 16/5 $\simeq$ 3ms; the A–B path will see a more-or-less-fixed 3ms increase in RTT. A will also see a decrease in bandwidth due to competition; with C consuming 80% of R1's queue, A's share wll fall to 20% and thus its bandwidth will fall to 20% of the R1–R2 link bandwidth, that is, 1 packet/ms. Denote this new value by BWE_{new}. TCP Vegas will attempt to decrease `cwnd` so that

$$\texttt{cwnd} \simeq \text{BWE}_{\text{new}} \times \text{RTT}_{\text{noLoad}} + 4$$

A's estimate of $\text{RTT}_{\text{noLoad}}$, as RTT_{min}, will not change; the RTT has gotten larger, not smaller. So we have $\text{BWE}_{\text{new}} \times \text{RTT}_{\text{noLoad}} \simeq$ 1 packet/ms $\times$ 5 ms = 5 packets; adding the 4 reserved for the queue, the new value of `cwnd` is now about 9, down from 14.

On the one hand, this new value of `cwnd` does represent 5 packets now in transit, plus 4 in R1's queue; this is indeed the correct response. But note that this division into transit and queue packets is an average. The actual physical A–B round-trip transit capacity remains about 10 packets, meaning that if the new packets were all appropriately spaced then *none* of them might be in any queue.

15.7 FAST TCP

FAST TCP is closely related to TCP Vegas; the idea is to keep the fixed-queue-utilization feature of TCP Vegas to the extent possible, but to provide overall improved performance, in particular in the face of competition with TCP Reno. Details can be found in *[JWL04]* and *[WJLH06]*. FAST TCP is patented; see patent 7,974,195.

As with TCP Vegas, the sender estimates $\text{RTT}_{\text{noLoad}}$ as RTT_{min}. At regular short **fixed** intervals (*eg* 20ms) `cwnd` is updated via the following weighted average:

$$\texttt{cwnd}_{\text{new}} = (1\text{-}\gamma)\times\texttt{cwnd} + \gamma\times((\text{RTT}_{\text{noLoad}}/\text{RTT})\times\texttt{cwnd} + \alpha)$$

where γ is a constant between 0 and 1 determining how "volatile" the `cwnd` update is ($\gamma\simeq 1$ is the most volatile) and α is a fixed constant, which, as we will verify shortly, represents the number of packets the

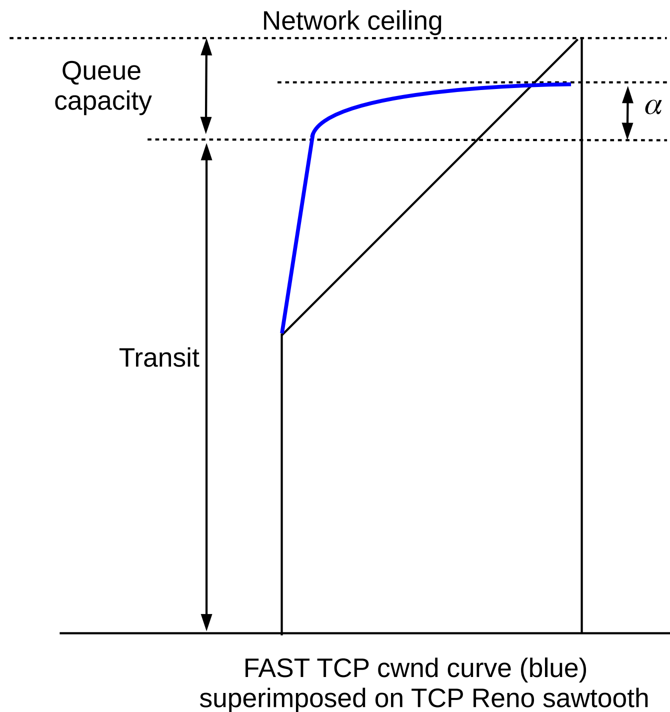
sender tries to keep in the bottleneck queue, as in TCP Vegas. Note that the `cwnd` update frequency is *not* tied to the RTT.

If RTT is constant for multiple consecutive update intervals, and is larger than RTT_{noLoad}, the above will converge to a constant `cwnd`, in which case we can solve for it. Convergence implies $cwnd_{new}$ = `cwnd` = ((RTT_{noLoad}/RTT)×`cwnd` + α), and from there we get `cwnd`×(RTT–RTT_{noLoad})/RTT = α. As we saw in *6.3.2 RTT Calculations*, `cwnd`/RTT is the throughput, and so α = throughput × (RTT–RTT_{noLoad}) is then the number of packets in the queue. In other words, FAST TCP, when it reaches a steady state, leaves α packets in the queue. As long as this is the case, the queue will not overflow (assuming α is less than the queue capacity).

Whenever the queue is not full, though, we have RTT = RTT_{noLoad}, in which case FAST TCP's `cwnd`-update strategy reduces to $cwnd_{new}$ = `cwnd` + $\gamma\times\alpha$. For γ=0.5 and α=10, this increments `cwnd` by 5. Furthermore, FAST TCP performs this increment at a specific rate independent of the RTT, *eg* every 20ms; for long-haul links this is much less than the RTT. FAST TCP will, in other words, increase `cwnd` very aggressively until the point when queuing delays occur and RTT begins to increase.

When FAST TCP is competing with TCP Reno, it does not directly address the queue-utilization competition problem experienced by TCP Vegas. FAST TCP will try to limit its queue utilization to α; TCP Reno, however, will continue to increase its `cwnd` until the queue is full. Once the queue begins to fill, TCP Reno will pull ahead of FAST TCP just as it did with TCP Vegas. However, FAST TCP does not *reduce* its `cwnd` in the face of TCP Reno competition as quickly as TCP Vegas.

Additionally, FAST TCP can often offset this Reno-competition problem in other ways as well. First, the value of α can be increased from the value of around 4 packets originally proposed for TCP Vegas; in *[TWHL05]* the value α=30 is suggested. Second, for high bandwidth×delay products, the queue-filling phase of a TCP Reno sawtooth (see *13.7 TCP and Bottleneck Link Utilization*) becomes relatively smaller. In the earlier link-unsaturated phase of each sawtooth, TCP Reno increases `cwnd` by 1 each RTT. As noted above, however, FAST TCP is allowed to increase `cwnd` much more rapidly in this earlier phase, and so FAST TCP can get substantially ahead of TCP Reno. It may fall back somewhat during the queue-filling phase, but overall the FAST and Reno flows may compete reasonably fairly.



FAST TCP cwnd curve (blue)
superimposed on TCP Reno sawtooth

The diagram above illustrates a FAST TCP graph of `cwnd` versus time, in blue; it is superimposed over one sawtooth of TCP Reno with the same network ceiling. Note that `cwnd` rises rapidly when it is below the path transit capacity, and then levels off sharply.

15.8 TCP Westwood

TCP Westwood represents an attempt to use the RTT-monitoring strategies of TCP Vegas to address the high-bandwidth problem; recall that the issue there is to distinguish between congestive and non-congestive losses. TCP Westwood can also be viewed as a refinement of TCP Reno's `cwnd`=`cwnd`/2 strategy, which is a greater drop than necessary if the queue capacity at the bottleneck router is less than the transit capacity. It remains a form of loss-based congestion control.

As in TCP Vegas, the sender keeps a continuous estimate of bandwidth, BWE, and estimates $\text{RTT}_{\text{noLoad}}$ by RTT_{min}. The minimum window size to keep the bottleneck link busy is, again as in TCP Vegas, BWE $\times$ $\text{RTT}_{\text{noLoad}}$. In TCP Vegas, BWE was calculated as `cwnd`/RTT; we will continue to use this for the time being but in fact TCP Westwood has used a wide variety of other algorithms, some of which are discussed in the following subsection, to infer the available average bandwidth from the returning ACKs.

The core TCP Westwood innovation is to, on loss, reduce `cwnd` as follows:

$$\texttt{cwnd} = \max(\texttt{cwnd}/2, \text{BWE}\times\text{RTT}_{\text{noLoad}}) \text{ if } \texttt{cwnd} > \text{BWE}\times\text{RTT}_{\text{noLoad}}$$

$$\text{no change, if } \texttt{cwnd} \leqslant \text{BWE}\times\text{RTT}_{\text{noLoad}}$$

The product $\text{BWE}\times\text{RTT}_{\text{noLoad}}$ represents what the sender believes is its current share of the "transit capacity" of the path. This product represents how many packets can be in transit (rather than in queues) at the current bandwidth BWE. The $\text{RTT}_{\text{noLoad}}$ estimate as RTT_{min} is relatively constant, but BWE may be

markedly reduced in the presence of competing traffic. A TCP Westwood sender never drops cwnd below what it believes to be the current transit capacity for the path.

Consider again the classic TCP Reno sawtooth behavior:

- cwnd alternates between cwnd_{min} and $\text{cwnd}_{max} = 2\times\text{cwnd}_{min}$.
- $\text{cwnd}_{max} \simeq$ transit_capacity + queue_capacity (or at least the sender's share of these)

As we saw in *13.7 TCP and Bottleneck Link Utilization*, if transit_capacity < cwnd_{min}, then Reno does a reasonably good job keeping the bottleneck link saturated. However, if transit_capacity > cwnd_{min}, then when Reno drops to cwnd_{min}, the bottleneck link is not saturated until cwnd climbs to transit_capacity. For high-speed networks, this latter case is the more likely one.

Westwood, on the other hand, would in that situation reduce cwnd only to transit_capacity, a smaller reduction. Thus TCP Westwood potentially better handles a wide range of router queue capacities. For bottleneck routers where the queue capacity is small compared to the transit capacity, TCP Westwood would in theory have a higher, finer-pitched sawtooth than TCP Reno: the teeth would oscillate between the network ceiling (= queue+transit) and the transit_capacity, versus Reno's oscillation between the network ceiling and half the ceiling.

In the event of a non-congestive (noise-related) packet loss, if it happens that cwnd is less than transit_capacity then TCP Westwood does not reduce the window size at all. That is, non-congestive losses with cwnd < transit_capacity have no effect. When cwnd > transit_capacity, losses reduce cwnd only to transit_capacity, and thus the link stays saturated.

In the large-cwnd, high-bandwidth case, non-congestive packet losses can easily lower the TCP Reno cwnd to well below what is necessary to keep the bottleneck link saturated. In TCP Westwood, on the other hand, the average cwnd may be lower than it would be without the non-congestive losses, but it will be high enough to keep the bottleneck link saturated.

TCP Westwood uses $\text{BWE}\times\text{RTT}_{noLoad}$ as a *floor* for reducing cwnd. TCP Vegas shoots to have the actual cwnd be just a few packets *above* this.

TCP Westwood is not any more aggressive than TCP Reno at increasing cwnd in no-loss situations. So while it handles the non-congestive-loss part of the high-bandwidth TCP problem very well, it does not particularly improve the ability of a sender to take advantage of a sudden large rise in the network ceiling.

TCP Westwood is also potentially very effective at addressing the lossy-link problem, as most non-congestive losses would result in no change to cwnd.

15.8.1 ACK Compression and Westwood+

So far, we have been assuming that ACKs never encounter queuing delays. They in fact will not, *if* they are traveling in the reverse direction from all *data* packets. But while this scenario covers any single-sender model and also systems of two or more competing senders, real networks have more complicated traffic patterns, and returning ACKs from an A⟶B data flow can indeed experience queuing delays if there is third-party traffic along some link in the B⟶A path.

Delay in the delivery of ACKs, leading to clustering of their arrival, is known as **ACK compression**; see *[ZSC91]* and *[JM92]* for examples. ACK compression causes two problems. First, arriving clusters of ACKs trigger corresponding bursts of data transmissions in sliding-windows senders; the end result is an uneven

data-transmission rate. Normally the bottleneck-router queue can absorb an occasional burst; however, if the queue is nearly full such bursts can lead to premature or otherwise unexpected losses.

The second problem with late-arriving ACKs is that they can lead to inaccurate or fluctuating measurements of bandwidth, upon which both TCP Vegas and TCP Westwood depend. For example, if bandwidth is estimated as `cwnd`/RTT, late-arriving ACKs can lead to inaccurate calculation of RTT. The original TCP Westwood strategy was to estimate bandwidth from the spacing between consecutive ACKs, much as is done with the packet-pairs technique (*14.2.6 Packet Pairs*) but smoothed with a suitable running average. This strategy turned out to be particularly vulnerable to ACK-compression errors.

For TCP Vegas, ACK compression means that occasionally the sender's `cwnd` may fail to be decremented by 1; this does not appear to be a significant impact, perhaps because `cwnd` is changed by at most ± 1 each RTT. For Westwood, however, if ACK compression happens to be occurring at the instant of a packet loss, then a resultant transient overestimation of BWE may mean that the new post-loss `cwnd` is too large; at a point when `cwnd` was supposed to fall to the transit capacity, it may fail to do so. This means that the sender has essentially taken a congestion loss to be non-congestive, and ignored it. The influence of this ignored loss will persist – through the much-too-high value of `cwnd` – until the following loss event.

To fix these problems, TCP Westwood has been amended to **Westwood+**, by increasing the time interval over which bandwidth measurements are made and by inclusion of an averaging mechanism in the calculation of BWE. Too much smoothing, however, will lead to an inaccurate BWE just as surely as too little.

Suitable smoothing mechanisms are given in *[FGMPC02]* and *[GM03]*; the latter paper in particular examines several smoothing algorithms in terms of their resistance to *aliasing effects*: the tendency for intermittent measurement of a periodic signal (the returning ACKs) to lead to much greater inaccuracy than might initially be expected. One smoothing filter suggested by *[GM03]* is to measure BWE only over entire RTTs, and then to keep a cumulative running average as follows, where BWM_k is the measured bandwidth over the kth RTT:

$$BWE_k = \alpha \times BWE_{k-1} + (1-\alpha) \times BWM_k$$

A suggested value of α is 0.9.

15.9 TCP Illinois

The general idea behind TCP Illinois, described in *[LBS06]*, is to use the usual AIMD(α,β) strategy but to have $\alpha = \alpha(RTT)$ be a decreasing function of the current RTT, rather than a constant. When the queue is empty and RTT is equal to RTT_{noLoad}, then α will be large, and `cwnd` will increase rapidly. Once RTT starts to increase, however, α will decrease rapidly, and the `cwnd` growth will level off. This leads to the same kind of concave `cwnd` graph as we saw above in FAST TCP; a consequence of this is that for most of the time between consecutive loss events `cwnd` is large enough to keep the bottleneck link close to saturated, and so to keep throughput high.

The actual $\alpha()$ function is not of RTT, but rather of `delay`, defined to be RTT – RTT_{noLoad}. As with TCP Vegas, RTT_{noLoad} is estimated by RTT_{min}. As a connection progresses, the sender maintains continually updated values not only for RTT_{min} but also for RTT_{max}. The sender then sets $delay_{max}$ to be RTT_{max} – RTT_{min}.

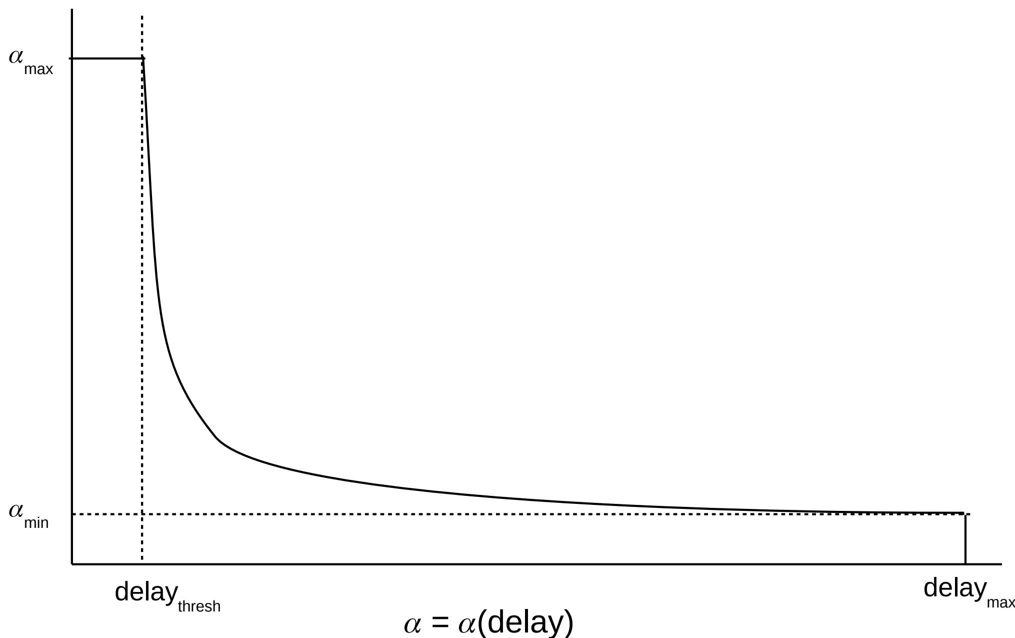
We are now ready to define α(delay). We first specify the highest value of α, α_{max}, and the lowest, α_{min}. In *[LBS06]* these are 10.0 and 0.1 respectively; in the linux 3.5 kernel they are 10.0 and 0.3. We also define

$\text{delay}_{\text{thresh}}$ to be $0.01 \times \text{delay}_{\text{max}}$ (the 0.01 is another tunable parameter). We then define $\alpha(\text{delay})$ as follows

$\alpha(\text{delay}) = \alpha_{\text{max}}$ if $\text{delay} \leqslant \text{delay}_{\text{thresh}}$

$\alpha(\text{delay}) = k_1/(\text{delay}+k_2)$ if $\text{delay}_{\text{thresh}} \leqslant \text{delay} \leqslant \text{delay}_{\text{max}}$

where k_1 and k_2 are chosen so that, for the lower formula, $\alpha(\text{delay}_{\text{thresh}}) = \alpha_{\text{max}}$ and $\alpha(\text{delay}_{\text{max}}) = \alpha_{\text{min}}$. In case there is a sudden spike in delay, $\text{delay}_{\text{max}}$ is updated before the above is evaluated, so we always have $\text{delay} \leqslant \text{delay}_{\text{max}}$. Here is a graph:



Whenever RTT = $\text{RTT}_{\text{noLoad}}$, delay=0 and so $\alpha(\text{delay}) = \alpha_{\text{max}}$. However, as soon as queuing delay just barely starts to begin, we will have delay > $\text{delay}_{\text{thresh}}$ and so $\alpha(\text{delay})$ begins to fall – rather precipitously – to α_{min}. The value of $\alpha(\text{delay})$ is always positive, though, so `cwnd` will continue to increase (unlike TCP Vegas) until a congestive loss eventually occurs. However, at that point the change in `cwnd` is very small, which minimizes the probability that multiple packets will be lost.

Note that, as with FAST TCP, the increase in delay is used to trigger the reduction in α.

TCP Illinois also supports having β be a decreasing function of delay, so that β(small_delay) might be 0.2 while β(larger_delay) might match TCP Reno's 0.5. However, the authors of *[LBS06]* explain that "the adaptation of β as a function of average queuing delay is only relevant in networks where there are non-congestion-related losses, such as wireless networks or extremely high speed networks".

15.10 Compound TCP

Compound TCP, or **CTCP**, is Microsoft's entry into the advanced-TCP field, although it is now available for linux as well; see *[TSZS06]*. The idea behind Compound TCP is to add a delay-based component to TCP Reno. To this end, CTCP supplements TCP Reno's `cwnd` with a delay-based contribution to the window size known as `dwnd`; the total window size is then

winsize = `cwnd` + `dwnd`

(As usual, winsize is also not allowed to exceed the receiver's advertised window size.) The per-RTT increment of `cwnd` is now 1/winsize (though note that `dwnd` has a separate per-RTT increment).

As in TCP Vegas, CTCP maintains RTT_{min} as a stand-in for RTT_{noLoad}, and also maintains a bandwidth estimate BWE = winsize/RTT_{actual}. These allow estimation of the current number of packets in the queue, denoted `diff` in *[TSZS06]*, as `diff` = `cwnd` × (1 – $RTT_{noLoad}/RTT_{actual}$).

When `diff` is less than γ packets, where the parameter γ is configurable but γ=30 is a good starting point, CTCP increases winsize (per RTT) according to the rule

winsize += $\alpha\times\text{winsize}^k$

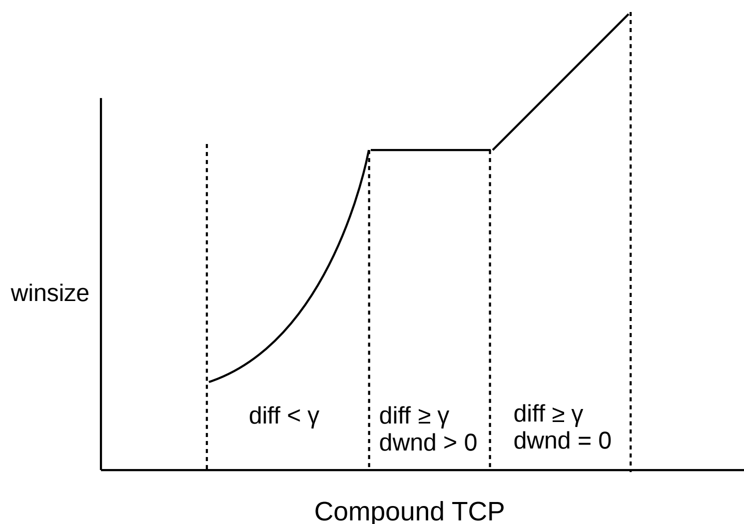
where the exponent k is chosen to be 0.8. (In *[TSZS06]* this increase is achieved by having `cwnd` be incremented by 1, and `dwnd` by $\alpha\times\text{winsize}^k - 1$.) This amounts to a fairly aggressive increase; for TCP Reno we have k=0. The choice of k=0.8 is intended to make CTCP competitive with Highspeed TCP; we will return to the justification of this below. We will also choose α=1/8, which we will take as given. The value γ=30 here plays very roughly a similar role as Fast TCP's α, also 30, in that both represent a threshold for queue utilization.

When CTCP encounters a loss, we set

winsize = winsize$\times$(1–β)

While β is potentially configurable, typically we will have the usual β=1/2.

Finally we have the case where `diff` > γ; that is, the queue has grown "significantly". If `dwnd` is also positive, it is decremented. The variable `cwnd` continues to increase, but `cwnd` and `dwnd` will cancel each other out over the short term, leading to a roughly constant value for winsize. When `dwnd` drops to 0, however, this cancellation ends, and TCP Reno's `cwnd` += 1 per RTT takes over; `dwnd` has no more effect until after the next packet loss. Considering all these cases, a rough graph of the growth of CTCP's winsize is the following:



Compound TCP

We next derive k=0.8 as the value that leads to fair competition with Highspeed TCP. To do this we need a modest bit of calculus; the derivation can be skipped if preferred. We start with a hypothetical TCP adjusting `cwnd` according to the rule `cwnd` += α × $\texttt{cwnd}^{0.8}$, per RTT, and show this TCP does indeed compete fairly with Highspeed TCP. If we measure time in RTTs, and denote `cwnd` by c = c(t), and extend

c(t) to a continuous function of t, this increment rule becomes $dc/dt = \alpha \times c^{0.8}$. Taking reciprocals, we get $dt/dc = (1/\alpha) \times c^{-0.8}$. We can now integrate both sides, which yields $t = k_1 \times c^{0.2}$ (ignoring the constant of integration), or $c = k_2 \times t^5$. Integrating again, we get the number of packets in one tooth (the area) to be proportional to T^6, where T is the time at the right edge of the tooth. (We are inappropriately ignoring the left edge of the tooth, but by the argument of exercise 16.7 in *14.13 Exercises* this turns out not to matter.) This area is the reciprocal of the loss rate p. Solving for T, we get T proportional to $(1/p)^{1/6}$. As the average `cwnd` is proportional to T^5 (the area divided by T), by substitution we can conclude that `cwnd` is proportional to $p^{-5/6} = p^{-0.833}$ (versus the original exponent in *15.5 Highspeed TCP* of –0.835).

Calculating $winsize^{0.8}$ is hard to do rapidly, so in practice the exponent 0.75 is used. With that value the exponentiation can be done with two applications of a fast square-root algorithm.

CTCP turns out to compete reasonably fairly one-on-one with Highspeed TCP, by virtue of the choice of k=0.8. However, when competing with a set of TCP Reno connections, CTCP leaves the Reno connections with nearly the same bandwidth they would have had if they were competing with one more TCP Reno connection instead. That is, CTCP resists "stealing" bandwidth. CTCP does, however, make effective use of the bandwidth that TCP Reno leaves unclaimed due to the high-bandwidth TCP problem.

15.11 TCP Veno

TCP Veno (*[FL03]*) is a synthesis of TCP Vegas and TCP Reno, which attempts to use the RTT-monitoring ideas of TCP Vegas while at the same time remaining about as "aggressive" as TCP Reno in using queue capacity. TCP Veno has generally been presented as an option to address TCP's lossy-link problem, rather than the high-bandwidth problem *per se*.

A TCP Veno sender estimates the number N of packets likely in the bottleneck queue as N_{queue} = `cwnd` - $BWE \times RTT_{noLoad}$, like TCP Vegas. TCP Veno then modifies the TCP Reno congestion-avoidance rule as follows, where the parameter β, representing the queue-utilization value at which TCP Veno slows down, might be around 5.

> if $N_{queue} < \beta$, `cwnd` = `cwnd` + 1 each RTT
> if $N_{queue} \geqslant \beta$, `cwnd` = `cwnd` + 0.5 each RTT

The above strategy makes `cwnd` growth less aggressive once link saturation is reached, but does continue to increase `cwnd` (half as fast as TCP Reno) until the queue is full and congestive losses occur.

When a packet loss does occur, TCP Veno uses its current value of N_{queue} to attempt to distinguish between non-congestive and congestive loss, as follows:

> if $N_{queue} < \beta$, the loss is probably *not* due to congestion; set `cwnd` = (4/5)×`cwnd`
> if $N_{queue} \geqslant \beta$, the loss probably *is* due to congestion; set `cwnd` = (1/2)×`cwnd` as usual

The idea here is that most router queues will have a total capacity much larger than β, so a loss with fewer than β likely does not represent a queue overflow. Note that, by comparison, TCP Westwood leaves `cwnd` unchanged if it thinks the loss is not due to congestion, and its threshold for making that determination is $N_{queue}=0$.

If TCP Veno encounters a series of non-congestive losses, the above rules make it behave like AIMD(1,0.8). Per the analysis in *14.7 AIMD Revisited*, this is equivalent to AIMD(2,0.5); this means TCP Veno will be about twice as aggressive as TCP Reno in recovering from non-congestive losses. This would provide

a definite improvement in lossy-link situations with modest bandwidth×delay products, but may not be enough to make a major dent in the high-bandwidth problem.

15.12 TCP Hybla

TCP Hybla (*[CF04]*) has one very specific focus: to address the TCP satellite problem (*3.9.2 Satellite Internet*) of very long RTTs. TCP Hybla selects a more-or-less arbitrary "reference" RTT, called RTT_0, and attempts to scale TCP Reno so as to behave like a TCP Reno connection with an RTT of RTT_0. In the paper *[CF04]* the authors suggest RTT_0 = 25ms.

Suppose a TCP Reno connection has, at a loss event at time t_0, reduced `cwnd` to $cwnd_{min}$. TCP Reno will then increment `cwnd` by 1 for each RTT, until the next loss event. This Reno behavior can be equivalently expressed in terms of the current time t as follows:

$cwnd = (t–t_0)/RTT + cwnd_{min}$

What TCP Hybla does is to use the above formula after replacing the actual RTT (or RTT_{noLoad}) with RTT_0. Equivalently, TCP Hybla defines the ratio of the two RTTs as $\varrho = RTT/RTT_0$, and then after each windowful (each time interval of length RTT) increments `cwnd` by ϱ^2 instead of by 1. In the event that RTT < RTT_0, ϱ is set to 1, so that short-RTT connections are not penalized.

Because `cwnd` now increases each RTT by ϱ^2, which can be relatively large, there is a good chance that when the network ceiling is reached there will be a burst of losses of size ~ϱ^2. Therefore, TCP Hybla strongly recommends that the receiving end support SACK TCP, so as to allow faster recovery from multiple packet losses. Another recommended feature is the use of TCP Timestamps; this is a standard TCP option that allows the sender to include its own timestamp in each data packet. The receiver is to echo back the timestamp in the corresponding ACK, thus allowing more accurate measurement by the receiver of the actual RTT.

Finally, to further avoid having these relatively large increments to `cwnd` result in multiple packet losses, TCP Hybla recommends some form of **pacing** to smooth out the actual transmission times. Rather than sending out four packets upon receipt of an ACK, for example, we might estimate the time T to the next transmission batch (*eg* when the next ACK arrives) and send the packets at intervals of T/4. At the time TCP Hybla was developed, pacing was poorly supported, but see *15.16 TCP BBR* below, where pacing is essential.

TCP Hybla applies a similar ϱ-fold scaling mechanism to threshold slow start, when a value for `ssthresh` is known. But the initial unbounded slow start is much more difficult to scale. Scaling at that point would mean repeatedly doubling `cwnd` and sending out flights of packets, *before* any ACKs have been received; this would likely soon lead to congestive collapse.

15.13 DCTCP

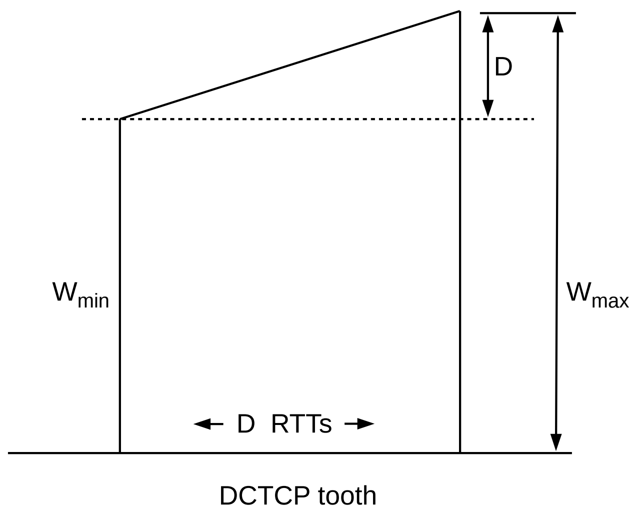
Unlike the other TCP flavors in this chapter, **Data Center TCP** (DCTCP) is intended for use only by connections starting and ending within the same data center. DCTCP is *not* meant to be used on the Internet at large, as it makes no pretense of competing fairly with TCP Reno. DCTCP was first described in *[AGMPS10]*, and is now also specified in **RFC 8257**.

A data center is a highly specialized networking environment. Round-trip times on the Internet at large might be 50-100 ms, but round-trip times in a data center are usually well under 1 ms. Communicating nodes in a data center are under common management, and so there is no "chicken and egg" problem regarding software installation: if a new TCP feature is desired, it can be made available everywhere. Finally, cost-saving is an issue: data centers have lots of switches and routers, and cheaper models generally have smaller queue capacities. Even with a 1-ms RTT, though, a 10 Gbps connection can have a bandwidth×delay product of 1.25 MB (800 packets); we would like to have queues be much smaller than this.

Recall that TCP Reno can be categorized as AIMD(1,0.5) (*14.7 AIMD Revisited*). The basic idea of DCTCP is to use AIMD(1,β) for values of β much smaller than 0.5. As the window-size reduction on packet loss is 1–β, this means that `cwnd` is relatively constant. If the transit capacity of a path is M, then the queue space needed to keep the minimum `cwnd` at M (and thus to keep the bottleneck link 100% utilized) is M×β/(1-β) $\simeq$ M×β if $\beta \simeq 0$.

This small β comes at the price of out-competing TCP Reno by a large margin. By Exercise 8.5 of *14.13 Exercises*, AIMD(1,β) is equivalent in terms of fairness to AIMD(α,0.5) for α = (2–β)/3β, and by the argument in *14.3.1 Example 2: Faster additive increase* an AIMD(α,0.5) connection out-competes TCP Reno by a factor of α. For β = 1/8 we have α = 5. For connections within a data center we can achieve fairness by implementing DCTCP everywhere, but introduction of DCTCP in the outside world DCTCP would be highly uncooperative.

The next step is to specify β. For the moment, we will make a simplifying assumption that there is only one connection, and no other traffic; in this case, the queue utilization increases by 1 for each RTT (once the queue becomes nonempty). We now determine β dynamically: we simply count the number D of RTTs before the queue is sufficiently full, and let β = 1/2D. (In *[AGMPS10]* and **RFC 8257**, 1/D is denoted by α, making $\beta = \alpha/2$, but to avoid confusion with the α in AIMD(α,β) we will write out the DCTCP α as `alpha` when we return to it below.)



DCTCP tooth

We can now relate D to `cwnd` and to the amplitude of `cwnd` variation. Let W_{max} denote the maximum `cwnd`, and W_{min} the minimum. Making the usual large-window simplifying assumptions, we have

> W_{min} + D = W_{max}, because `cwnd` increases by 1 each RTT
>
> $W_{min} = W_{max} \times (1–1/2D)$

Eliminating W_{min} and solving, we get $W_{max} = 2D^2$, or $D = \sqrt{(W_{max}/2)}$. Note that D is also the amplitude of the queue variation, assuming we keep the bottleneck link saturated, and so is the absolute minimum queue capacity needed. If the goal is keeping the queue small, this compares quite favorably to TCP Reno, in which $D = W_{min} = W_{max}/2$.

Now let K represent the maximum queue capacity; the next step is to relate K and D. We need to ensure that we can avoid having K be much larger than D. We have $W_{max} = TC + K$, where TC is the transit capacity of the link, that is, bandwidth×delay. We can express the minimum queue utilization as $Q_{min} = K - D = K - \sqrt{((TC+K)/2}$. If we choose K = TC, which is necessary with TCP Reno to avoid underutilized bandwidth, we certainly will have K much larger than D. However, to ensure $Q_{min} \geqslant 0$ we need $K = \sqrt{((TC+K)/2}$, or $K^2 = TC/2 + K/2$, which, because TC is relatively large (perhaps 800 packets), simply requires K just a bit larger than $\sqrt{(TC/2)}$. That is, K need not be much larger than D. At this point, we can afford to be more concerned with K's being too small, and thereby allowing intervals where the bottleneck link is idle.

Empirically, a workable value for the queue capacity K is around 65 for 10 Gbps Ethernet, which is moderately above $\sqrt{(TC/2)}$ but still very affordable. It is large enough that link utilization remains near 100%.

We now need to address the simplifying assumption that there was only one connection. First, there might be N connections, quite possibly synchronized. This means that the queue variation is N×D. It also means D will be somewhat smaller, though, as the total `cwnd` will be increasing N times faster.

A more serious issue is that there is also a *lot* of other traffic in a data center, so much so that queue utilization is dominated by a more-or-less random component. Instead of measuring when the queue utilization reaches a set level, we must measure when the *average* utilization reaches that level. DCTCP achieves this with a clever application of ECN (*14.8.2 Explicit Congestion Notification (ECN)*). The use of ECN to detect queue fullness, rather than packet drops, has the added advantage of avoiding packet losses and timeouts. Within a data center, DCTCP may very well rely on switch-based ECN rather than router-based.

In normal ECN, once the receiver has seen a packet with the CE bit, it is supposed to mark ECE (CE echo) in all returning ACKs until the sender acknowledges having responded to the congestion through the use of the CWR bit. DCTCP modifies this by having the receiver mark *only* ACKs to packets that arrive with CE set. This allows the sender to gauge the severity of congestion: if every other data packet has its CE bit set, then half the returning ACKs will be marked. (Delayed ACKs may complicate this, as the two data packets being acknowledged may have different CE marks, but mostly this is both infrequent and not serious, and in any event DCTCP recommends sending two separate ACKs with different ECE marks in such a case.)

Classically, having every other data packet marked should never happen: all data packets arriving at the router before the queue capacity K is reached should be unmarked, and all packets arriving after K is reached should be marked. But due to the random queue fluctuations described in the previous paragraph, this all-unmarked-then-all-marked pattern may be riddled with exceptions. What the DCTCP sender does is to measure the *average* marking rate, using an averaging interval at least as long as one "tooth". If there are D–1 unmarked RTTs and 1 marked RTT, then the average marking rate should be 1/D.

This is exactly what DCTCP looks for: once a significant number of marked ACKs arrives, indicating that congestion is experienced, the DCTCP sender looks at its running average of the marked fraction, and takes that to be 1/D. (More precisely, DCTCP denotes by `alpha` the marked fraction, and sets D = 1/`alpha`, and then β = 1/2D = `alpha`/2.) DCTCP then reduces its `cwnd` by 1–β as above. The actual algorithm does not involve the queue capacity K, as a TCP sender is unlikely to know K.

While it is not part of DCTCP proper, another common configuration choice for intra-data-center connections is to reduce the minimum TCP retransmission timeout (RTO). The RTO value is computed adaptively,

as in *12.19 TCP Timeout and Retransmission*, but is subject to a minimum. As late as 2011, **RFC 6298** recommended (but did not require) a minimum RTO of 1.0 seconds, which is three orders of magnitude too large for a data center.

There is no global linux minimum-RTO configuration setting, but this can be altered on a per-destination basis using the `ip route` command:

```
ip route change to 10.1.2.0/24 via 10.0.2.1 dev eth0 rto_min 20ms
```

The actual RTO values of current TCP connections can be viewed using the linux command `ss --info`. On recent versions of Windows, a global minimum RTO can be set, for the `custom` template, using `netsh interface tcp set supplemental template=custom minRto=20`

15.13.1 TCP Incast

There is one particular congestion issue, mostly but not entirely exclusive to data centers, that DCTCP does not handle directly: the **TCP incast** problem. Imagine one node sending out multiple simultaneous queries to "helper" nodes, and expecting more-or-less-simultaneous responses. One example might be a request for a large data block that has been distributed over multiple file-server systems; another might be a MapReduce request for calculation results. Either way, all the respondents may reply at about the same time, and all the responses may arrive together at the router and lead to queue overflow and packet loss. DCTCP (and any other TCP) cannot be of much help if each individual connection may be sending only one packet. This is one reason for having a slightly larger queue capacity than the DCTCP analysis alone might suggest.

The TCP incast problem is made much worse when (as is often the case) the helper-node requests must be executed serially; we saw this issue before with RPC in *11.5.3 Serial Execution*. Sometimes serialization requirements can be eliminated through careful design; sometimes they cannot.

15.14 H-TCP

H-TCP, or TCP-Hamilton, is described in *[LSL05]*. Like Highspeed-TCP it primarily allows for faster growth of `cwnd`; unlike Highspeed-TCP, the `cwnd` increment is determined not by the size of `cwnd` but by the elapsed time since the previous loss event. The threshold for accelerated `cwnd` growth is generally set to be 1.0 seconds after the most recent loss event. Using an RTT of 50 ms, that amounts to 20 RTTs, suggesting that when $\mathtt{cwnd}_{min}$ is less than 20 then H-TCP behaves very much like TCP Reno.

The specific H-TCP acceleration rule first defines a time threshold t_L. If t is the elapsed time in seconds since the previous loss event, then for $t \leqslant t_L$ the per-RTT window-increment α is 1. However, for $t > t_L$ we define

$$\alpha(t) = 1 + 10(t-t_L) + (t-t_L)^2/4$$

We then increment `cwnd` by $\alpha(t)$ after each RTT, or, equivalently, by $\alpha(t)$/`cwnd` after each received ACK.

At $t=t_L+1$ seconds (nominally 2 seconds), α is 12. The quadratic term dominates the linear term when $t-t_L > 40$. If RTT = 50 ms, that is 800 RTTs.

Even if `cwnd` is very large, growth is at the same rate as for TCP Reno until $t>t_L$; one consequence of this is that, at least in the first second after a loss event, H-TCP competes fairly with TCP Reno, in the sense that

both increase cwnd at the same absolute rate. H-TCP starts "from scratch" after each packet loss, and does not re-enter its "high-speed" mode, even if cwnd is large, until after time t_L.

A full H-TCP implementation also adjusts the multiplicative factor β as follows (the paper *[LSL05]* uses β to represent what we denote by 1–β). The RTT is monitored, as with TCP Vegas. However, the RTT increase is not used for per-packet or per-RTT adjustments; instead, these measurements are used after each loss event to update β so as to have

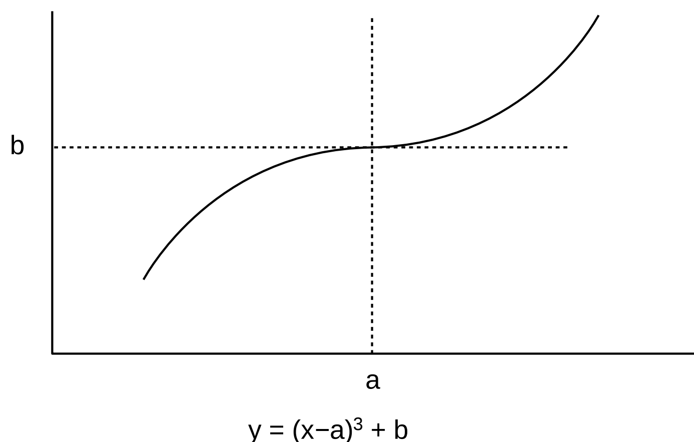
$$1-\beta = RTT_{min}/RTT_{max}$$

The value 1–β is capped at a maximum of 0.8, and at a minimum of 0.5. To see where the ratio above comes from, first note that RTT_{min} is the usual stand-in for RTT_{noLoad}, and RTT_{max} is, of course, the RTT when the bottleneck queue is full. Therefore, by the reasoning in *6.3.2 RTT Calculations*, equation 5, 1–β is the ratio transit_capacity / (transit_capacity + queue_capacity). At a congestion event involving a single uncontested flow we have cwnd = transit_capacity + queue_capacity, and so after reducing cwnd to $(1-\beta)\times$cwnd, we have $cwnd_{new}$ = transit_capacity, and hence (as in *13.7 TCP and Bottleneck Link Utilization*) the bottleneck link will remain 100% utilized after a loss. The cap on 1–β of 0.8 means that if the queue capacity is smaller than a quarter of the transit capacity then the bottleneck link *will* experience some idle moments.

When β is changed, H-TCP also adjusts α to $\alpha' = 2\beta\alpha(t)$ so as to improve fairness with other H-TCP connections with different current values of β.

15.15 TCP CUBIC

TCP Cubic attempts, like Highspeed TCP, to solve the problem of efficient TCP transport when bandwidth$\times$delay is large. TCP Cubic allows very fast window expansion; however, it also makes attempts to slow the growth of cwnd sharply as cwnd approaches the current network ceiling, and to treat other TCP connections fairly. Part of TCP Cubic's strategy to achieve this is for the window-growth function to slow down (become concave) as the previous network ceiling is approached, and then to increase rapidly again (become convex) if this ceiling is surpassed without losses. This concave-then-convex behavior mimics the graph of the cubic polynomial cwnd = t^3, hence the name (TCP Cubic also improves an earlier TCP version known as TCP BIC).



As mentioned above, TCP Cubic is currently (2013) the default linux congestion-control implementation.

TCP Cubic is documented in *[HRX08]*. TCP Cubic is not described in an RFC, but there is an Internet Draft http://tools.ietf.org/id/draft-rhee-tcpm-cubic-02.txt.

TCP Cubic has a number of interrelated features, in an attempt to address several TCP issues:

- Reduction in RTT bias
- TCP Friendliness when most appropriate
- Rapid recovery of `cwnd` following its decrease due to a loss event, maximizing throughput
- Optimization for an unchanged network ceiling (corresponding to $\texttt{cwnd}_{max}$)
- Rapid expansion of `cwnd` when a raised network ceiling is detected

The eponymous cubic polynomial $y=x^3$, appropriately shifted and scaled, is used to determine changes in `cwnd`. No special algebraic properties of this polynomial are used; the point is that the curve, while steadily increasing, is first concave and then convex; the authors of *[HRX08]* write "[t]he choice for a cubic function is incidental and out of convenience". This $y=x^3$ polynomial has an *inflection point* at x=0 where the tangent line is horizontal; this is the point where the graph changes from concave to convex.

We start with the basic outline of TCP Cubic and then consider some of the bells and whistles. We assume a loss has just occurred, and let W_{max} denote the value of `cwnd` at the point when the loss was discovered. TCP Cubic then sets `cwnd` to $0.8\times W_{max}$; that is, TCP Cubic uses β = 0.2. The corresponding α for TCP-Friendly AIMD(α,β) would be α=1/3, but TCP Cubic uses this α only in its TCP-Friendly adjustment, below.

We now define a cubic polynomial W(t), a shifted and scaled version of $w=t^3$. The parameter t represents the elapsed time since the most recent loss, in seconds. At time t>0 we set `cwnd` = W(t). The polynomial W(t), and thus the `cwnd` rate of increase, as in TCP Hybla, *is no longer tied to the connection's RTT*; this is done to reduce if not eliminate the RTT bias that is so deeply ingrained in TCP Reno.

We want the function W(t) to pass through the point representing the `cwnd` just after the loss, that is, $\langle t,W\rangle = \langle 0,0.8\times W_{max}\rangle$. We also want the inflection point to lie on the horizontal line $y=W_{max}$. To fully determine the curve, it is at this point sufficient to specify the value of t at this inflection point; that is, how far horizontally W(t) must be stretched. This horizontal distance from t=0 to the inflection point is represented by the constant K in the following equation; W(t) returns to its pre-loss value W_{max} at t=K. C is a second constant.

$$W(t) = C\times(t-K)^3 + W_{max}$$

It suffices algebraically to specify either C or K; the two constants are related by the equation obtained by plugging in t=0. K changes with each loss event, but it turns out that the value of C can be constant, not only for any one connection but for all TCP Cubic connections. TCP Cubic specifies for C the *ad hoc* value 0.4; we can then set t=0 and, with a bit of algebra, solve to obtain

$$K = (W_{max}/2)^{1/3} \text{ seconds}$$

If W_{max} = 250, for example, K=5; if RTT = 100 ms, this is 50 RTTs.

When each ACK arrives, TCP Cubic records the arrival time t, calculates W(t), and sets `cwnd` = W(t). At the next packet loss the parameters of W(t) are updated.

If the network ceiling does not change, the next packet loss will occur when `cwnd` again reaches the same W_{max}; that is, at time t=K after the previous loss. As t approaches K and the value of `cwnd` approaches W_{max}, the curve W(t) flattens out, so `cwnd` increases slowly.

This concavity of the cubic curve, increasing rapidly but flattening near W_{max}, achieves two things. First, throughput is boosted by keeping `cwnd` close to the available path transit capacity. In *13.7 TCP and Bottleneck Link Utilization* we argued that if the path transit capacity is large compared to the bottleneck queue capacity (and this is the case for which TCP Cubic was designed), then TCP Reno averages 75% utilization of the available bandwidth. The bandwidth utilization increases linearly from 50% just after a loss event to 100% just before the next loss. In TCP Cubic, the initial rapid rise in `cwnd` following a loss means that the average will be much closer to 100%. Another important advantage of the flattening is that when `cwnd` is finally incremented to the point of loss, it likely is just over the network ceiling; the connection has an excellent chance that only one or two packets are lost rather than a large burst. This facilitates the NewReno Fast Recovery algorithm, which TCP Cubic still uses if the receiver does not support SACK TCP.

Once t>K, W(t) becomes convex, and in fact begins to increase rapidly. In this region, `cwnd` > W_{max}, and so the sender knows that the network ceiling has increased since the previous loss. The TCP Cubic strategy here is to probe aggressively for additional capacity, increasing `cwnd` very rapidly until the new network ceiling is encountered. The cubic increase function is in fact quite aggressive when compared to any of the other TCP variants discussed here, and time will tell what strategy works best. As an example in which the TCP Cubic approach seems to pay off, let us suppose the current network ceiling is 2,000 packets, and then (because competing connections have ended) increases to 3,000. TCP Reno would take 1,000 RTTs for `cwnd` to reach the new ceiling, starting from 2,000; if one RTT is 50 ms that is 50 seconds. To find the time t-K that TCP Cubic will need to increase `cwnd` from 2,000 to 3,000, we solve $3000 = W(t) = C\times(t–K)^3 + 2000$, which works out to t-K $\simeq$ 13.57 seconds (recall 2000 = W(K) here).

The constant C=0.4 is determined empirically. The cubic inflection point occurs at $t = K = (W_{max}\times\beta/C)^{1/3}$. A larger C reduces the time K between the a loss event and the next inflection point, and thus the time between consecutive losses. If W_{max} = 2000, we get K=10 seconds when β=0.2 and C=0.4. If the RTT were 50 ms, 10 seconds would be 200 RTTs.

For TCP Reno, on the other hand, the interval between adjacent losses is $W_{max}/2$ RTTs. If we assume a specific value for the RTT, we can compare the Reno and Cubic time intervals between losses; for an RTT of 50 ms we get

W_{max}	Reno	Cubic
2000	50 sec	10 sec
250	6.2 sec	5 sec
54	1.35 sec	3 sec

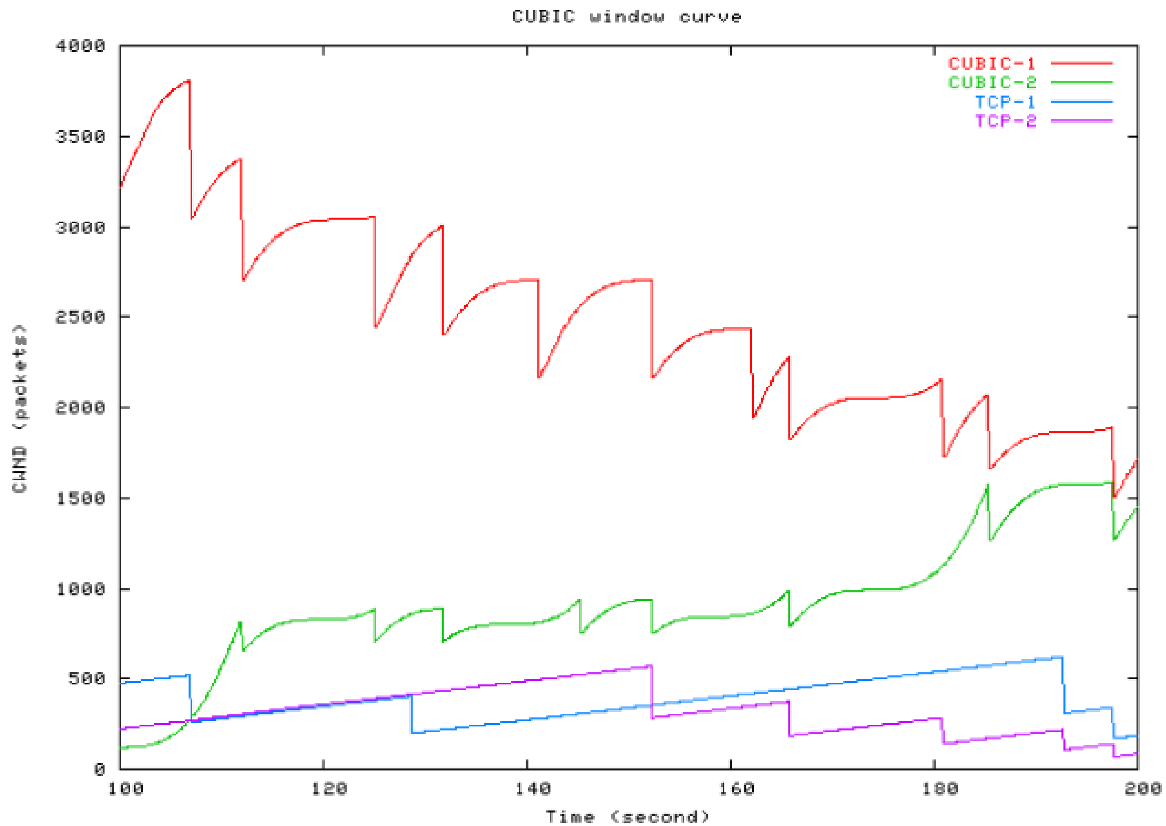
For smaller RTTs, the basic TCP Cubic strategy above runs the risk of being at a competitive disadvantage compared to TCP Reno. For this reason, TCP Cubic makes a **TCP-Friendly adjustment** in the window-size calculation: on each arriving ACK, `cwnd` is set to the maximum of W(t) and the window size that TCP Reno would compute. The TCP Reno calculation can be based on an actual count of incoming ACKs, or be based on the formula $(1-\beta)\times W_{max} + \alpha\times t/RTT$.

Note that this adjustment is only "half-friendly": it guarantees that TCP Cubic will not choose a window size smaller than TCP Reno's, but places no restraints on the choice of a larger window size.

A consequence of the TCP-Friendly adjustment is that, on networks with modest bandwidth×delay products, TCP Cubic behaves exactly like TCP Reno.

TCP Cubic also has a provision to detect if a given W_{max} is *lower* than the previous value, suggesting increasing congestion; in this situation, `cwnd` is lowered by an additional factor of $1–\beta/2$. This is known as **fast convergence**, and helps TCP Cubic adapt more quickly to reductions in available bandwidth.

The following graph is taken from *[RX05]*, and shows TCP Cubic connections competing with each other and with TCP Reno.



The diagram shows four connections, all with the same RTT. Two are TCP Cubic and two are TCP Reno. The red connection, cubic-1, was established and with a maximum `cwnd` of about 4000 packets when the other three connections started. Over the course of 200 seconds the two TCP Cubic connections reach a fair equilibrium; the two TCP Reno connections reach a reasonably fair equilibrium with one another, but it is much lower than that of the TCP Cubic connections.

On the other hand, here is a graph from *[LSM07]*, showing the result of competition between two flows using an earlier version of TCP Cubic over a low-speed connection. One connection has an RTT of 160ms and the other has an RTT a tenth that. The bottleneck bandwidth is 1 Mbit/sec, meaning that the bandwidth×delay product for the 160ms connection is 13-20 packets (depending on the packet size used).

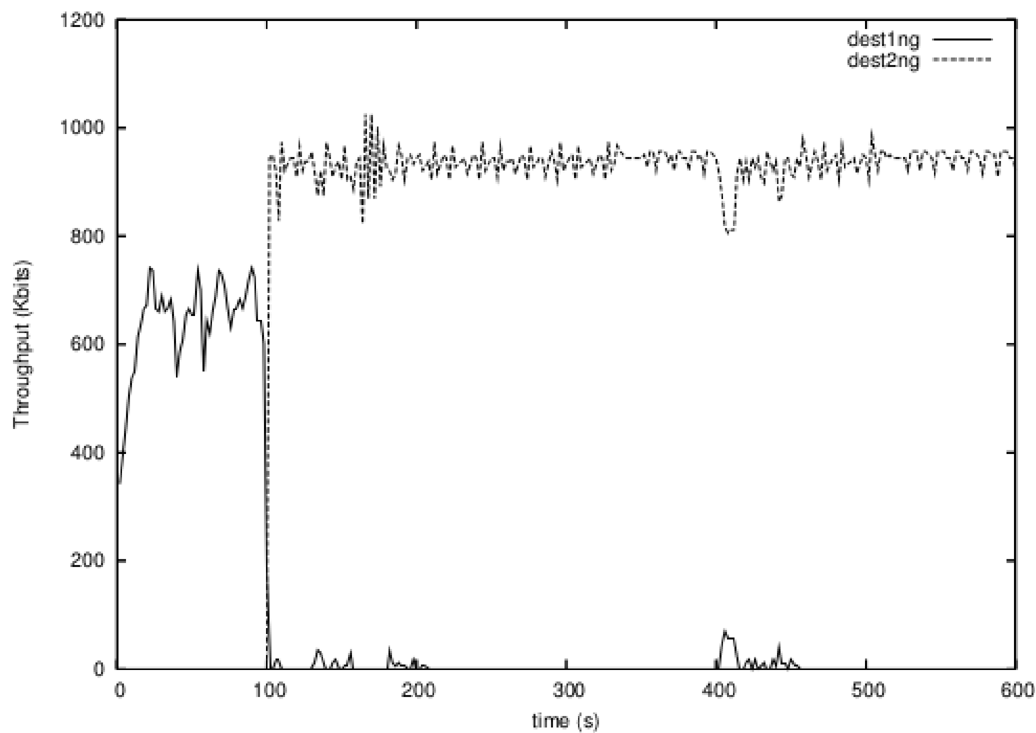


Fig. 14. Two Cubic TCP flows. 1Mbps link, flow 1 RTT 160ms, flow 2 RTT 16ms.

Note that the longer-RTT connection (the solid line) is almost completely starved, once the shorter-RTT connection starts up at T=100. This is admittedly an extreme case, and there have been more recent fixes to TCP Cubic, but it does serve as an example of the need for testing a wide variety of competition scenarios.

15.16 TCP BBR

TCP BBR returns to the central idea of TCP Vegas: to measure the available bandwidth and RTT_{min}, and to base the number of in-flight packets on the measured bandwidth×delay product. "BBR" here stands for **B**ottleneck **B**andwidth and **R**TT; it is described in *[CGYJ16]* and in an Internet Draft. There are some large differences from TCP Vegas, however; ultimately, these differences enable TCP BBR to compete reasonably fairly with TCP Reno. One important difference is that TCP BBR does not engage in the high-precision monitoring of RTT for increases above RTT_{noLoad}. As a result, TCP BBR does not fit the TCP Vegas delay-based congestion-control model; it is for that reason sometimes referred to as **congestion-based** congestion control.

TCP BBR is, in practice, **rate-based** rather than window-based; that is, at any one time, TCP BBR sends at a given calculated **rate**, instead of sending new data in direct response to each received ACK. Each arriving ACK does potentially update the current rate, much as each arriving ACK in TCP Reno slides the sender's window forwards; however, the connection between arriving ACKs and new data transmissions is decidedly indirect.

Rate-based sending requires some form of **pacing support** by the underlying LAN layer, so that packets

can be sent at equal time intervals. On a 10 Gbps link, this time interval can be as small as a microsecond; conventional timers don't work well at these time scales. Linux TCP BBR implementations generally use the pacing support built into the so-called Fair Queuing (FQ) queuing discipline (which is not actually a true Fair Queuing implementation in the sense of *19.5 Fair Queuing*).

Throughout the lifetime of a connection, TCP BBR maintains an estimate for RTT_{min}, which is nominally the stand-in for RTT_{noLoad} except that it may go up in the presence of competition; see below. TCP BBR also maintains a current bandwidth estimate, which we denote BWE. As with TCP Vegas, BWE is much more volatile than RTT_{min} as it better reflects the current degree of bandwidth competition. After each RTT, TCP BBR records the throughput during that RTT; BWE is then the maximum of the last ten per-RTT throughput measurements. That BWE is the *maximum* rate recorded over the past ten RTTs, rather than the *average*, will be important below.

The fundamental congestion indicators for TCP BBR are changes to its BWE and RTT_{min} estimates; packet losses are not used directly as evidence of congestion. As we shall see below, TCP BBR reduces its sending rate in response to decreases in BWE; this is TCP BBR's primary congestion response. When losses do occur, TCP BBR does enter a recovery mode, but it is much less conservative than TCP Reno's halving of `cwnd`. TCP BBR's initial response to a loss is to limit the number of packets in flight (FlightSize) to the number currently in flight, which allows it to continue to send new data at the rate of arriving ACKs. This is not necessarily a reduction in FlightSize, and, if it is, FlightSize may be allowed to grow, even if additional losses are discovered. Overall, this strategy is quite effective at handling non-congestive losses without losing throughput.

In its core state, known as **PROBE_BW**, TCP BBR continually updates BWE as above and then sets its base sending rate to BWE. It then sets its `cwnd` target (or, more properly, its FlightSize target, as losses may have occurred) to $2 \times BWE \times RTT_{min}$. This results in a bottleneck queue utilization equal to the transit capacity. If the actual available bandwidth does not change, then sending at rate BWE will send new packets at exactly the rate of returning ACKs and so FlightSize will not change. TCP BBR does allow for faster initial growth (see STARTUP mode, below) to reach the FlightSize target.

If the actual available bandwidth falls, BWE will not reflect that for ten RTTs. As a result, TCP BBR may for a while send faster than the rate of returning ACKs. If this happens, the bottleneck queue utilization will rise. Eventually, BWE will fall to match the rate of returning ACKs. Similarly, if the actual available bandwidth rises, queue utilization will fall. However, it will not fall to zero – and so cause sending to starve – in a single RTT unless the bandwidth doubles, and after that the increased bandwidth will be reflected in the updated BWE.

TCP BBR must, like every TCP flavor, regularly probe to see if additional bandwidth is available. TCP BBR does this by periodically (currently every eight RTTs, where RTT is measured as RTT_{min}) increasing its sending rate by an additional factor of 1.25; that is, it sets a variable `pacing_gain` to 1.25 and sends at the new rate `pacing_gain` × BWE. The increase lasts one RTT interval. *If* there was no competition, and if the bottleneck link was fully utilized, this `pacing_gain` increase results in no change to BWE. All that happens is that the queue builds up, and the 1.25-fold larger flight of packets results in an RTT that is also 1.25 times larger. In the next RTT interval, TCP BBR sets `pacing_gain` to 0.75, which causes the newly created additional queue to dissipate. After that it resumes its regular rate, that is, with `pacing_gain` = 1.0, for the next six RTT intervals.

Consider, however, what happens if TCP BBR is *competing*, perhaps with TCP Reno. Increasing the sending rate by a factor of 1.25 now results in greater queue (or bottleneck link) utilization, which results in an immediate increase in BWE for that RTT. At this point, recall that BWE is the maximum of the last ten per-RTT measurements; the end result is that BWE is set to this elevated value for the next ten RTTs. In

the following RTT, `pacing_gain` drops to 0.75 as before, but this time TCP BBR has measured a larger BWE, and this change to BWE persists.

Here is a concrete example of BWE increase. To simplify the analysis, we will assume TCP BBR's Flight-Size is BWE × RTT_{min}, dropping the factor of 2. Suppose a TCP BBR connection and a TCP Reno connection share a bottleneck link with a bandwidth of 2 packets/ms. The RTT_{min} (= RTT_{noLoad}) of each connection is 80 ms, making the transit capacity 160 packets. Finally, suppose that each connection has 80 packets in flight, exactly filling the transit capacity but with no queue utilization (so RTT_{min} = RTT_{actual}). Over the course of the eight-RTT `pacing_gain` cycle, the Reno connection's `cwnd` rises by 8, to 88 packets. This means the total queue utilization is now 8 packets, divided on average between BBR and Reno in the proportion 80 to 88.

Now the BBR cycle with `pacing_gain`=1.25 arrives; for the next RTT, the BBR connection has 80×1.25 = 100 packets in flight. The total number of packets in flight is now 188. The RTT climbs to 188/2 = 94 ms, and the next BBR BWE measurement is 100 packets in 94 ms, or 1.064 packets/ms (the precise value may depend on exactly when the measurement is recorded). For the following RTT, `pacing_gain` drops to 0.75, but the higher BWE persists. For the rest of the `pacing_gain` cycle, TCP BBR calculates a base rate corresponding to 1.064 × 80 = 85 packets in flight per RTT, which is close to the TCP Reno `cwnd`. See also exercise 14.0.

TCP BBR also has another mechanism, arguably more important in the long run, for maintaining its fair share of the bandwidth. Periodically (every ~10 seconds), TCP BBR connections re-measure RTT_{min}, entering **PROBE_RTT** mode. In this state the number of packets in flight drops to four, and stays there for at least one RTT_{actual} as measured for these four packets (with a minimum of 200 ms). Afterwards the connection returns to PROBE_**BW** mode with a freshly estimated RTT_{min}. The value of BWE is picked up where it was left off, so that if RTT_{min} increases, then so does the sending rate BWE × RTT_{min}. A certain amount of potential throughput is "wasted" during these PROBE_RTT intervals, but as they amount to ~200 ms out of every 10 sec, or 2%, the impact is negligible.

If, during the PROBE_RTT mode, competing connections keep some packets in the bottleneck queue, then the queuing delay corresponding to those packets will be incorporated into the new RTT_{min} measurement; because of this, RTT_{min} may significantly exceed RTT_{noLoad} and thus cause TCP BBR to send at a more competitive rate. Suppose, for example, that in the BBR-vs-Reno scenario above, Reno has gobbled up a total of 240 spots in the bottleneck queue, thus increasing the RTT for both connections to (240+80)/2 = 160. During a PROBE_RTT cycle, TCP BBR will drop its link utilization essentially to zero, but TCP Reno will still have 240 packets in transit, so TCP BBR will measure RTT_{min} as 240/2 = 120 ms. After the PROBE_RTT phase is over, TCP BBR will increase its sending rate by 50% over what it had been when RTT_{min} was 80.

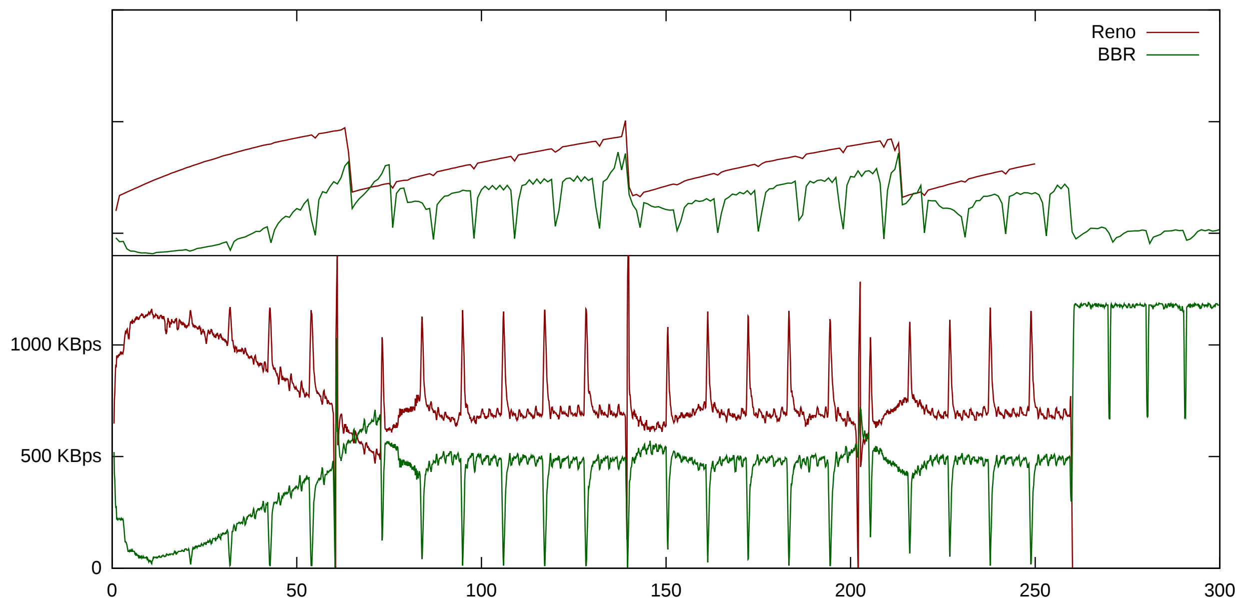
Note that, in any one RTT, we can either measure bottleneck bandwidth *or* RTT, but not both. If the number of packets in flight is larger than the transit capacity then the packet return rate reflects the bottleneck bandwidth. Conversely, we can measure RTT_{min} only if the number of packets in flight is smaller than the transit capacity.

When a connection is first opened, a TCP BBR connection is in **STARTUP** mode, which is similar to TCP Reno's slow start. In this mode, `pacing_gain` is 2.89 (2/log(2)) consistently, which leads to exponential growth of the number of packets in flight. STARTUP mode ends when an additional RTT yields no improvement in BWE. At this point TCP BBR has overfilled the queue substantially (just as a TCP Reno connection does in slow start), and so the connection enters **DRAIN** mode to reduce the queue. This is accomplished by setting `pacing_gain` = 1/2.89. The connection transitions from DRAIN to PROBE_RTT when the number of packets in flight drops to 2 × BWE × RTT_{min}.

Below is a diagram of TCP BBR competing with TCP Reno in a setting where the bottleneck queue capacity is eight times the bandwidth×delay product, which is 40 ms. It was produced using the Mininet network emulator, *18.7 TCP Competition: Reno vs BBR*. The large queue capacity was contrived specifically to be beneficial to TCP Reno, in that in a similar setting with a queue capacity approximately equal to the bandwidth×delay product TCP BBR often ends up quite a bit ahead of TCP Reno. Such large queues are, however, a not-uncommon real-world situation on high-capacity backbone links (*13.7.1 Bufferbloat*). Acting alone, Reno's `cwnd` would range between 4.5 and 9 times the bandwidth×delay product, which works out to keeping the queue over 70% full on average.

The lower part of the diagram shows each connection's share of the 10 Mbps (1.25 KBps) bottleneck bandwidth. The upper part shows the number of packets "in flight" (for TCP Reno, outside of Fast Recovery, that is of course `cwnd`). The Reno sawtooth pattern is clearly visible.

A dominant feature of the graph is the spikes every 10 seconds (down for BBR, correspondingly up for Reno) caused by TCP BBR's periodic PROBE_RTT mode.



For the first ten seconds, TCP Reno does indeed run away with all the bandwidth. But after the first PROBE_RTT event TCP BBR begins to catch up, and the two tie at around T=60 seconds. After that Reno mostly stays a little ahead of TCP BBR, typically with about 58% of the bandwidth versus BBR's 42%, but the point here is that, even in circumstances favorable to Reno, BBR does not collapse.

It is evident from the graph, particularly during the first 60 seconds, that the PROBE_RTT intervals do not lead to sudden jumps in throughput. Almost all of the change in throughput occurs during the PROBE_BW intervals. That said, it is the PROBE_RTT interval at T=10 that triggers the ensuing turnaround in throughput.

In addition to the sharp PROBE_RTT spikes every 10 seconds, we also see smaller spikes at a rate of about 6 every 10 seconds. These represent the pacing-gain cycling within BBR's PROBE_BW phase. If eight RTT_{min} times amount to 10/6 seconds, then RTT_{min} must be about 200 ms. When the queue is completely full, RTT_{actual} is 9 × 40 ms = 360 ms, but during TCP BBR's PROBE_RTT cycles RTT_{actual} does indeed drop considerably, which accounts for the 200 ms value. This value is then used as RTT_{min} for the next ten seconds.

Experimental results in *[CGYJ16]* indicate that TCP BBR has been much more successful than TCP Cubic in addressing the high-bandwidth TCP problem on parts of Google's network. This is presumably because TCP BBR does not necessarily reduce throughput at all when faced with occasional non-congestive losses.

15.17 Epilog

TCP Reno's core congestion algorithm is based on algorithms in Jacobson and Karel's 1988 paper *[JK88]*, now (2017) approaching thirty years old. There are concerns both that TCP Reno uses too much bandwidth (the greediness issue) and that it does not use enough (the high-bandwidth-TCP problem).

There are also broad changes in TCP usage patterns. Twenty years ago, the vast majority of all TCP traffic represented downloads from "major" servers. Today, over half of all Internet TCP traffic is peer-to-peer rather than server-to-client. The rise in online video streaming creates new demands for excellent TCP real-time performance.

So which TCP version to use? That depends on circumstances; some of the TCPs above are primarily intended for relatively specific environments; for example, TCP Hybla for satellite links and TCP Veno for mobile devices (including wireless laptops). If the sending and receiving hosts are under common management, and especially if intervening traffic patterns are relatively stable, one can run a few simple throughput-comparison experiments to find which TCP version works best.

But there are two problems with this experimental approach. First, intervening traffic patterns are often *not* stable; a TCP version that worked well in one traffic environment might perform poorly in another. TCP Vegas, after all, does well in a Vegas-only environment; problems arise only when there is competing TCP Reno traffic, or the equivalent. Second, and perhaps more seriously, the best-performing TCP version might achieve its throughput at the expense of other users' TCP traffic. As a simple example, consider the effect of simply increasing the TCP Reno additive-increase value, perhaps from AIMD(1,0.5) to AIMD(**10**,0.5). As we saw in *14.3.1 Example 2: Faster additive increase*, this gives the faster-incrementing TCP an unfair (in fact tenfold) advantage. If the goal is to find a TCP version that *all* users will be happy with, this will not be effective.

Then there is the question of what TCP to use on a server that is serving up large volumes of data, to a range of disparate hosts and with a wide variety of competing-traffic scenarios. Here, experimentation is even more difficult. Many trials will be needed to determine reliably which TCP version works best in the most cases, even ignoring the impact on competing traffic. These issues suggest a need for continued research into how to update and improve TCP, and Internet congestion-management generally.

Finally, while most new TCPs are designed to hold their own in a Reno world, there is some question that perhaps we would all be better off with a radical rather than incremental change. Might TCP Vegas be a better choice, if only the queue-grabbing greediness of TCP Reno could be restrained? Questions like these are today entirely hypothetical, but it is not impossible to envision an Internet backbone that implemented non-FIFO queuing mechanisms (*19 Queuing and Scheduling*) that fundamentally changed the rules of the game.

15.18 Exercises

1.0. How would TCP Vegas respond if it estimated RTT_{noLoad} = 100ms, with a bandwidth of 1 packet/ms, and then due to a routing change the RTT_{noLoad} increased to 200ms without changing the bandwidth? What `cwnd` would be chosen? Assume no competition from other senders.

2.0. Suppose a TCP Vegas connection from A to B passes through a bottleneck router R. The RTT_{noLoad} is 50 ms and the bottleneck bandwidth is 1 packet/ms.

(a). If the connection keeps 4 packets in the queue (*eg* α=3, β=5), what will RTT_{actual} be? What value of `cwnd` will the connection choose? What will be the value of BWE?

(b). Now suppose a competing (non-Vegas) connection keeps 6 packets in the queue to the Vegas connection's 4, eventually meaning that the other connection will have 60% of the bandwidth. What will be the Vegas connection's steady-state values for RTT_{actual}, `cwnd` and BWE?

3.0. Suppose a TCP Vegas connection has R as its bottleneck router. The transit capacity is M, and the queue utilization is currently Q>0 (meaning that the transit path is 100% utilized, although not necessarily by the TCP Vegas packets). The current TCP Vegas `cwnd` is $cwnd_V$. Show that the number of packets TCP Vegas calculates are in the queue, **queue_use**, is

queue_use = $cwnd_V \times Q/(Q+M)$

4.0. Suppose that at time T=0 a TCP Vegas connection and a TCP Reno connection share the same path, and each has 100 packets in the bottleneck queue, exactly filling the transit capacity of 200. TCP Vegas uses α=1, β=2. By the previous exercise, in any RTT with $cwnd_V$ TCP Vegas packets and $cwnd_R$ TCP Reno packets in flight and $cwnd_V+cwnd_R>200$, N_{queue} is $cwnd_V/(cwnd_V+cwnd_R)$ multiplied by the total queue utilization $cwnd_V+cwnd_R-200$.

Continue the following table, where T is measured in RTTs, up through the next two RTTs where $cwnd_V$ is *not* decremented; that is, find the next two rows where the TCP Vegas queue share is less than 2. (After each of these RTTs, $cwnd_V$ is not decremented.) This can be done either with a spreadsheet or by simple algebra. Note that the TCP Reno $cwnd_R$ will always increment.

T	$cwnd_V$	$cwnd_R$	TCP Vegas queue share
0	100	100	0
1	101	101	1
2	102	102	2
3	101	103	(101/204)x4 = 1.980 < β
4	101	104	Vegas has (101/205)×5 = 2.463 packets in queue
5	100	105	Vegas has (100/205)×5 = 2.435 packets in queue
6	99	106	(99/205)×5 = 2.439

This exercise attempts to explain the *linear decrease* in the TCP Vegas graph in the diagram in *16.5 TCP Reno versus TCP Vegas*. Competition with TCP Reno means not only that $cwnd_V$ stops increasing, but in fact it decreases by 1 most RTTs.

5.0. Suppose that, as in the previous exercise, a FAST TCP connection and a TCP Reno connection share the same path, and at T=0 each has 100 packets in the bottleneck queue, exactly filling the transit capacity

of 200. The FAST TCP parameter γ is 0.5. The FAST TCP and TCP Reno connections have respective cwnds of cwnd_F and cwnd_R. You may use the fact that, as long as the queue is nonempty, $\text{RTT}/\text{RTT}_\text{noLoad}$ = (cwnd_F+cwnd_R)/200.

Find the value of cwnd_F at T=40, where T is counted in units of 20 ms until T = 40, using α=4, α=10 and α=30. Assume RTT $\simeq$ 20 ms as well. Use of a spreadsheet is recommended. The table here uses α=10.

T	cwnd_F	cwnd_R
0	100	100
1	105	101
2	108.47	102
3	110.77	103
4	112.20	104

6.0. Suppose A sends to B as in the layout below. The packet size is 1 KB and the bandwidth of the bottleneck R–B link is 1 packet / 10 ms; returning ACKs are thus normally spaced 10 ms apart. The RTT_noLoad for the A–B path is 200 ms.

However, large amounts of traffic are also being sent from C to A; the bottleneck link for that path is R–A with bandwidth 1 KB / 5 ms. The queue at R for the R–A link has a capacity of 40 KB. ACKs are 50 bytes.

(a). What is the maximum possible arrival time difference on the A–B path for ACK[0] and ACK[20], if there are no queuing delays at R in the A→B direction? ACK[0] should be forwarded immediately by R; ACK[20] should have to wait for 40 KB at R

(b). What is the minimum possible arrival time difference for the same ACK[0] and ACK[20]?

7.0. Suppose a TCP Veno and a TCP Reno connection compete along the same path; there is no other traffic. Both start at the same time with cwnds of 50; the total transit capacity is 160. Both share the next loss event. The bottleneck router's queue capacity is 60 packets; sometimes the queue fills and at other times it is empty. TCP Veno's parameter β is zero, meaning that it shifts to a slower cwnd increment as soon as the queue just begins filling.

(a). In how many RTTs will the queue begin filling?

(b). At the point the queue is completely filled, how much larger will the Reno cwnd be than the Veno cwnd?

8.0. Suppose two connections use TCP Hybla. They do not compete. The first connection has an RTT of 100 ms, and the second has an RTT of 1000 ms. Both start with cwnd_min = 0 (literally meaning that nothing is sent the first RTT).

(a). How many packets are sent by each connection in four RTTs (involving three cwnd increases)?

(b). How many packets are sent by each connection in four seconds? Recall 1+2+3+...+N = N(N+1)/2.

9.0. Suppose that at time T=0 a TCP Illinois connection and a TCP Reno connection share the same path, and each has 100 packets in the bottleneck queue, exactly filling the transit capacity of 200. The respective `cwnd`s are $\texttt{cwnd}_I$ and $\texttt{cwnd}_R$. The bottleneck queue capacity is 100.

Find the value of $\texttt{cwnd}_I$ at T=50, where T is the number of elapsed RTTs. At this point $\texttt{cwnd}_R$ is, of course, 150.

T	$\texttt{cwnd}_I$	$\texttt{cwnd}_R$
0	100	100
1	101	101
2	?	102

You may assume that the delay, RTT – RTT_{noLoad}, is proportional to queue_utilization = $\texttt{cwnd}_I$+$\texttt{cwnd}_R$–200α. Using this expression to represent delay, $delay_{max}$ = 100 and so $delay_{thresh}$ = 1. When calculating α(delay), assume α_{max} = 10 and α_{min} = 0.1.

10.0. Assume that a TCP connection has an RTT of 50 ms, and the time between loss events is 10 seconds.

(a). For a TCP Reno connection, what is the bandwidth×delay product?

(b). For an H-TCP connection, what is the bandwidth×delay product?

11.0. For each of the values of W_{max} below, find the *change* in TCP Cubic's `cwnd` over one 100 ms RTT at each of the following points:

i. Immediately after the previous loss event, when t = 0.

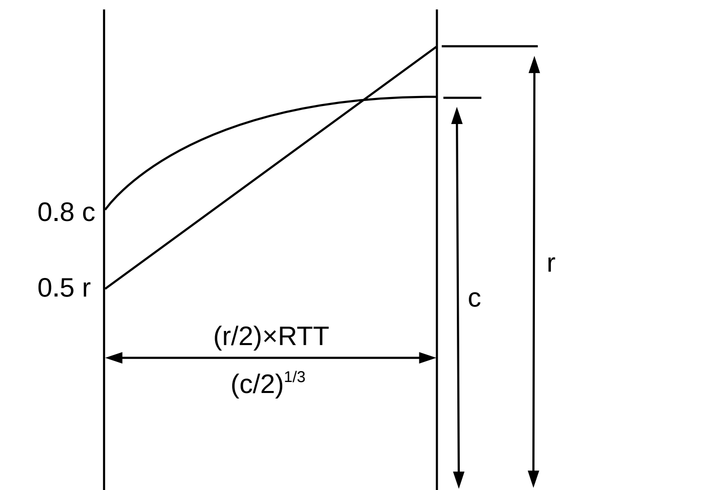
ii. At the midpoint of the tooth, when t=K/2

iii. At the point when `cwnd` has returned to W_{max}, at t=K

(a). W_{max} = 250 (making K=5)

(b). W_{max} = 2000 (making K=10)

12.0. Suppose a TCP Reno connection is competing with a TCP Cubic connection. There is no other traffic. All losses are synchronized. In this setting, once the steady state is reached, the `cwnd` graphs for one tooth will look like this:



One tooth, TCP Cubic v TCP Reno

Let c be the maximum `cwnd` of the TCP Cubic connection ($c=W_{max}$) and let r be the maximum of the TCP Reno connection. Let M be the network ceiling, so a loss occurs when c+r reaches M. The width of the tooth for TCP Reno is $(r/2)\times$RTT, where RTT is measured in seconds; the width of the TCP Cubic tooth is $(c/2)^{1/3}$. For the examples here, ignore the TCP-Friendly feature of TCP Cubic.

(a). If M = 200 and RTT = 50 ms = 0.05 sec, show that at the steady state r $\simeq$ 130.4 and c = M–r $\simeq$ 69.6.

(b). Find equilibrium r and c (to the nearest integer) for M=1000 and RTT = 50 ms. Hint: use of a spreadsheet or scripting language makes trial-and-error quite practical.

(c). Find equilibrium r and c for M = 1000 and RTT = 100 ms.

13.0. Suppose a TCP Westwood connection has the path A——R1——R2——B. The R1–R2 link is the bottleneck, with bandwidth 1 packet/ms, and RTT_{noLoad} is 200 ms. At T=0, with `cwnd` = 300 so the queue at R1 has 100 A–B packets, the R1–R2 throughput for A's packets falls to 1 packet / 2 ms, perhaps due to competition. At that same time, and perhaps also due to competition, a single A–B packet is lost at R1.

(a). Suppose A responds to the loss using the original BWE of 1 packet/ms. What transit capacity will A calculate, and how will A update its `cwnd`?

(b). Now suppose A uses the new throughput of 1 packet / 2 ms as its BWE. What transit capacity will A calculate, and how will A update its `cwnd`?

(c). Suppose A calculates BWE as `cwnd`/RTT. What value of BWE does A obtain by measuring the RTT of the packet just before the one that was lost?

14.0. In *15.16 TCP BBR* we estimated the impact on TCP BBR's BWE value during the interval when `pacing_gain`=1.25. Suppose now that the BBR and Reno connections each have 800 packets in transit, instead of 80. Assume the bottleneck bandwidth rises tenfold to 20 packets/ms, so RTT_{noLoad} is still 80 ms. During the 8-RTT pacing-gain cycle, Reno increases its `cwnd` to 808. *If* BWE is measured at the optimum point after BBR's `pacing_gain`=1.25 rate increase, what is the new value of BWE?

> Between the idea
> And the reality
> Between the motion
> And the act
> Falls the Shadow
>
> – TS Eliot, *The Hollow Men*

> Try to leave out the part that readers tend to skip.
>
> – Elmore Leonard, *10 Rules for Writing*

In previous chapters, especially *14 Dynamics of TCP Reno*, we have at times made simplifying assumptions about TCP Reno traffic. In the present chapter we will look at actual TCP behavior, through simulation, enabling us to explore the accuracy of some of these assumptions. The primary goal is to provide comparison between idealized TCP behavior and the often-messier real behavior; a secondary goal is perhaps to shed light on some of the issues involved in simulation. For example, in the discussion in *16.3 Two TCP Senders Competing* of competition between TCP Reno connections with different RTTs, we address several technical issues that affect the relevance of simulation results.

Parts of this chapter may serve as a primer on using the ns-2 simulator, though a primer focused on the goal of illuminating some of the basic operation and theory of TCP through experimentation. However, some of the outcomes described may be of interest even to those not planning on designing their own simulations.

Simulation is frequently used in the literature when comparing different TCP flavors for effective throughput (for example, the graphs excerpted in *15.15 TCP CUBIC* were created through simulation). An alternative to simulation, network *emulation*, involves running actual system networking software in a multi-node environment created through containerization or virtualization; we present this in *18 Mininet*. An important advantage of simulation over emulation, however, is that as emulations get large and complex they also get bogged down, and it can be hard to distinguish results from artifacts.

We begin this chapter by looking at a single connection and analyzing how well the TCP sawtooth utilizes the bottleneck link. We then turn to competition between two TCP senders. The primary competition example here is between TCP Reno connections with different RTTs. This example allows us to explore the synchronized-loss hypothesis (*14.3.4 Synchronized-Loss Hypothesis*) and to introduce phase effects, transient queue overflows, and other unanticipated TCP behavior. We also introduce some elements of designing simulation experiments. The second example compares TCP Reno and TCP Vegas. We close with a straightforward example of a wireless simulation.

16.1 The ns-2 simulator

The tool used for much research-level network simulations is **ns**, for **n**etwork **s**imulator and originally developed at the Information Sciences Institute. The ns simulator grew out of the REAL simulator developed by Srinivasan Keshav *[SK88]*; later development work was done by the Network Research Group at the Lawrence Berkeley National Laboratory.

We will describe in this chapter the **ns-2** simulator, hosted at www.isi.edu/nsnam/ns. There is now also an ns-3 simulator, available at nsnam.org. Because ns-3 is not backwards-compatible with ns-2 and the programming interface has changed considerably, we take the position that ns-3 is an entirely different package, though one likely someday to supercede ns-2 entirely. While there is a short introduction to ns-3 in this book (*17 The ns-3 Network Simulator*), its use is arguably quite a bit more complicated for beginners, and the particular simulation examples presented below are well-suited to ns-2. While ns-3 supports more complex and realistic modeling, and is the tool of choice for serious research, this added complexity comes at a price in terms of configuration and programming. The standard ns-2 tracefile format is also quite easy to work with using informal scripting.

Research-level use of ns-2 often involves building new modules in C++, and compiling them in to the system. For our purposes here, the stock ns-2 distribution is sufficient. The simplest way to install ns-2 is probably with the "allinone" distribution, which does still require compiling but comes with a very simple `install` script. (As of 2014, the ns-allinone-2.35 version appeared to compile only with gcc/g++ no more recent than version 4.6.)

The native environment for ns-2 (and ns-3) is linux. Perhaps the simplest approach for Windows users is to install a linux virtual machine, and then install ns-2 under that. It is also possible to compile ns-2 under the Cygwin system; an older version of ns-2 may still be available as a Cygwin binary.

To create an ns-2 simulation, we need to do the following (in addition to a modest amount of standard housekeeping).

- define the network **topology**, including all nodes, links and router queuing rules
- create some TCP (or UDP) connections, called **Agents**, and attach them to nodes
- create some **Applications** – usually FTP for bulk transfer or telnet for intermittent random packet generation – and attach them to the agents
- start the simulation

Once started, the simulation runs for the designated amount of time, driven by the packets generated by the Application objects. As the simulated applications generate packets for transmission, the ns-2 system calculates when these packets arrive and depart from each node, and generates simulated acknowledgment packets as appropriate. Unless delays are explicitly introduced, node responses – such as forwarding a packet or sending an ACK – are instantaneous. That is, if a node begins sending a simulated packet from node N1 to N2 at time T=1.000 over a link with bandwidth 60 ms per packet and with propagation delay 200 ms, then at time T=1.260 N2 will have received the packet. N2 will then respond at that same instant, if a response is indicated, *eg* by enqueuing the packet or by forwarding it if the queue is empty.

Ns-2 does not necessarily require assigning IP addresses to every node, though this is possible in more elaborate simulations.

Advanced use of ns-2 (and ns-3) often involves the introduction of **randomization**; for example, we will in *16.3 Two TCP Senders Competing* introduce both random sliding-windows delays and traffic generators that release packets at random times. While it is possible to seed the random-number generator so that different runs of the same experiment yield different outcomes, we will not do this here, so the random-number generator will always produce the same sequence. A consequence is that the same ns-2 script should yield exactly the same result each time it is run.

16.1.1 Using ns-2

The scripting interface for ns-2 uses the language **Tcl**, pronounced "tickle"; more precisely it is object-Tcl, or OTcl. For simple use, learning the general Tcl syntax is not necessary; one can proceed quite successfully by modifying standard examples.

The network simulation is defined in a Tcl file, perhaps `sim.tcl`; to run the simulation one then runs the command

```
ns sim.tcl
```

The result of running the ns-2 simulation will be to create some files, and perhaps print some output. The most common files created are the ns-2 trace file – perhaps `sim.tr` – which contains a record for every packet arrival, departure and queue event, and a file `sim.nam` for the **network animator**, nam, that allows visual display of the packets in motion (the ns-3 version of nam is known as NetAnim). The sliding-windows video in *6.2 Sliding Windows* was created with nam.

Within Tcl, variables can be assigned using the `set` command. Expressions in `[ ... ]` are evaluated. Numeric expressions must also use the `expr` command:

```
set foo [expr $foo + 1]
```

As in unix-style shell scripting, the value of a variable X is $X; the name X (without the $) is used when setting the value (in Perl and PHP, on the other hand, many variable names begin with $, which is included both when evaluating and setting the variable). Comments are on lines beginning with the '#' character. Comments can *not* be appended to a line that contains a statement (although it is possible first to start a new logical line with `;`).

Objects in the simulation are generally created using built-in constructors; the constructor in the line below is the part in the square brackets (recall that the brackets must enclose an expression to be evaluated):

```
set tcp0 [new Agent/TCP/Reno]
```

Object attributes can then be assigned values; for example, the following sets the data portion of the packets in TCP connection `tcp0` to 960 bytes:

```
$tcp0 set packetSize_ 960
```

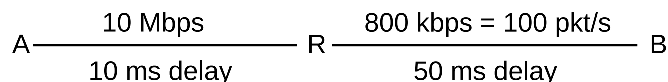
Object attributes are retrieved using `set` without a value; the following assigns variable `ack0` the current value of the `ack_` field of `tcp0`:

```
set ack0 [$tcp0 set ack_]
```

The **goodput** of a TCP connection is, properly, the number of application bytes received. This differs from the **throughput** – the total bytes sent – in two ways: the latter includes both packet headers and retransmitted packets. The `ack0` value above includes no retransmissions; we will occasionally refer to it as "goodput" in this sense.

16.2 A Single TCP Sender

For our first script we demonstrate a single sender sending through a router. Here is the topology we will build, with the delays and bandwidths:



The smaller bandwidth on the R–B link makes it the bottleneck. The default TCP packet size in ns-2 is 1000 bytes, so the bottleneck bandwidth is nominally 100 packets/sec or 0.1 packets/ms. The bandwidth×RTT_{noLoad} product is 0.1 packets/ms × 120 ms = 12 packets. Actually, the default size of 1000

bytes refers to the data segment, and there are an additional 40 bytes of TCP and IP header. We therefore set `packetSize_` to 960 so the actual transmitted size is 1000 bytes; this makes the bottleneck bandwidth exactly 100 packets/sec.

We want the router R to have a queue capacity of 6 packets, plus the one currently being transmitted; we set `queue-limit` = 7 for this. We create a TCP connection between A and B, create an ftp sender on top that, and run the simulation for 20 seconds. The nodes A, B and R are named; the links are not.

The ns-2 default maximum window size is 20; we increase that to 100 with `$tcp0 set window_ 100`; otherwise we will see an artificial cap on the `cwnd` growth (in the next section we will increase this to 65000).

The script itself is in a file basic1.tcl, with the 1 here signifying a single sender.

```
# basic1.tcl simulation: A---R---B

#Create a simulator object
set ns [new Simulator]

#Open the nam file basic1.nam and the variable-trace file basic1.tr
set namfile [open basic1.nam w]
$ns namtrace-all $namfile
set tracefile [open basic1.tr w]
$ns trace-all $tracefile

#Define a 'finish' procedure
proc finish {} {
        global ns namfile tracefile
        $ns flush-trace
        close $namfile
        close $tracefile
        exit 0
}

#Create the network nodes
set A [$ns node]
set R [$ns node]
set B [$ns node]

#Create a duplex link between the nodes

$ns duplex-link $A $R 10Mb 10ms DropTail
$ns duplex-link $R $B 800Kb 50ms DropTail

# The queue size at $R is to be 7, including the packet being sent
$ns queue-limit $R $B 7

# some hints for nam
# color packets of flow 0 red
$ns color 0 Red
$ns duplex-link-op $A $R orient right
$ns duplex-link-op $R $B orient right
$ns duplex-link-op $R $B queuePos 0.5
```

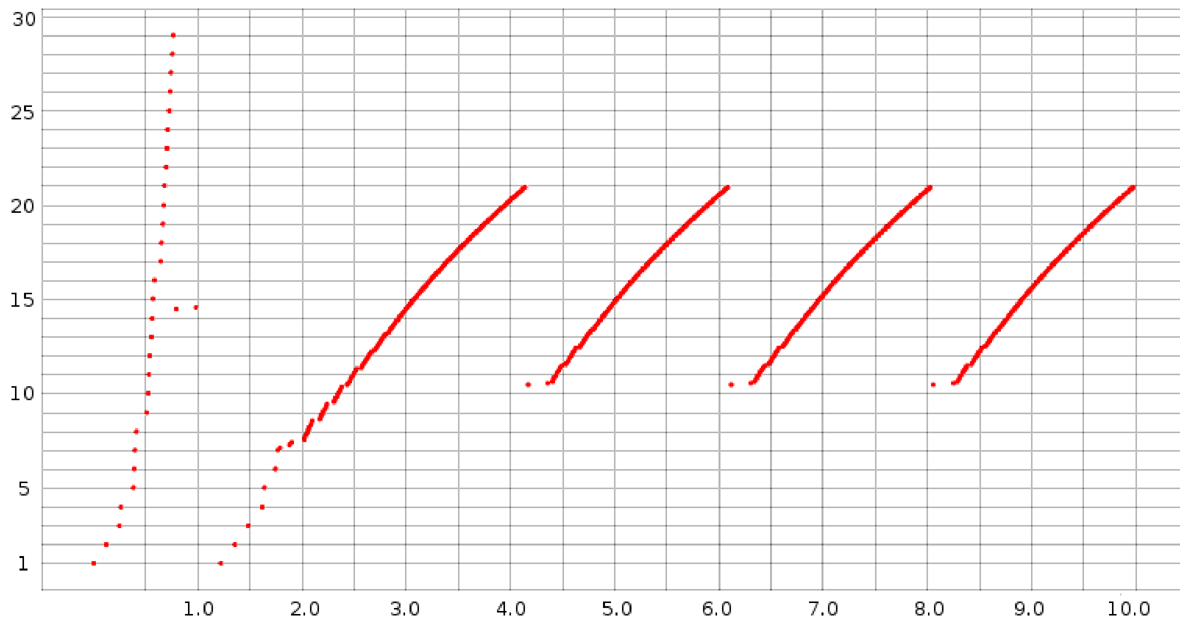
```
# Create a TCP sending agent and attach it to A
set tcp0 [new Agent/TCP/Reno]
# We make our one-and-only flow be flow 0
$tcp0 set class_ 0
$tcp0 set window_ 100
$tcp0 set packetSize_ 960
$ns attach-agent $A $tcp0

# Let's trace some variables
$tcp0 attach $tracefile
$tcp0 tracevar cwnd_
$tcp0 tracevar ssthresh_
$tcp0 tracevar ack_
$tcp0 tracevar maxseq_

#Create a TCP receive agent (a traffic sink) and attach it to B
set end0 [new Agent/TCPSink]
$ns attach-agent $B $end0

#Connect the traffic source with the traffic sink
$ns connect $tcp0 $end0

#Schedule the connection data flow; start sending data at T=0, stop at T=10.0
set myftp [new Application/FTP]
$myftp attach-agent $tcp0
$ns at 0.0 "$myftp start"
$ns at 10.0 "finish"

#Run the simulation
$ns run
```

After running this script, there is no command-line output (because we did not ask for any); however, the files basic1.tr and basic1.nam are created. Perhaps the simplest thing to do at this point is to view the animation with nam, using the command `nam basic1.nam`.

In the animation we can see slow start at the beginning, as first one, then two, then four and then eight packets are sent. A little past T=0.7, we can see a string of packet losses. This is visible in the animation as a tumbling series of red squares from the top of R's queue. After that, the TCP sawtooth takes over; we alternate between the `cwnd` linear-increase phase (congestion avoidance), packet loss, and threshold slow start. During the linear-increase phase the bottleneck link is at first incompletely utilized; once the bottleneck link is saturated the router queue begins to build.

16.2.1 Graph of `cwnd` v time

Here is a graph of `cwnd` versus time, prepared (see below) from data in the **trace file** basic1.tr:



Slow start is at the left edge. Unbounded slow start runs until about T=0.75, when a timeout occurs; bounded slow start runs from about T=1.2 to T=1.8. After that, all losses have been handled with fast recovery (we can tell this because `cwnd` does not drop below half its previous peak). The first three teeth following slow start have heights (`cwnd` peak values) of 20.931, 20.934 and 20.934 respectively; when the simulation is extended to 1000 seconds all subsequent peaks have exactly the same height, `cwnd` = 20.935. The spacing between the peaks is also constant, 1.946 seconds.

Because `cwnd` is incremented by ns-2 after each arriving ACK as described in *13.2.1 Per-ACK Responses*, during the linear-increase phase there are a great many data points jammed together; the bunching effect is made stronger by the choice here of a large-enough dot size to make the slow-start points clearly visible. This gives the appearance of continuous line segments. Upon close examination, these line segments are slightly concave, as discussed in *15.5 Highspeed TCP*, due to the increase in RTT as the queue fills. Individual flights of packets can just be made out at the lower-left end of each tooth, especially the first.

16.2.2 The Trace File

To examine the simulation (or, for that matter, the animation) more quantitatively, we turn to a more detailed analysis of the trace file, which contains records for all **packet events** plus (because it was requested) **variable-trace** information for `cwnd_`, `ssthresh_`, `ack_` and `maxseq_`; these were the variables for which we requested traces in the basic1.tcl file above.

The bulk of the trace-file lines are **event** records; three sample records are below. (These are in the default event-record format for point-to-point links; ns-2 has other event-record formats for wireless. See `use-newtrace` in *16.6 Wireless Simulation* below.)

```
r 0.58616 0 1 tcp 1000 ------- 0 0.0 2.0 28 43
+ 0.58616 1 2 tcp 1000 ------- 0 0.0 2.0 28 43
d 0.58616 1 2 tcp 1000 ------- 0 0.0 2.0 28 43
```

The twelve event-record fields are as follows:

1. **r** for received, **d** for dropped, **+** for enqueued, **-** for dequeued. Every arriving packet is enqueued, even if it is immediately dequeued. The third packet above was the first dropped packet in the entire simulation.
2. the time, in seconds.
3. the number of the sending node, in the order of node definition and starting at 0. If the first field was "+", "-" or "d", this is the number of the node doing the enqueuing, dequeuing or dropping. Events beginning with "-" represent this node sending the packet.
4. the number of the destination node. If the first field was "r", this record represents the packet's arrival at this node.
5. the protocol.
6. the packet size, 960 bytes of data (as we requested) plus 20 of TCP header and 20 of IP header.
7. some TCP flags, here represented as "`-------`" because none of the flags are set. Flags include E and N for ECN and A for reduction in the advertised winsize.
8. the flow ID. Here we have only one: flow 0. This value can be set via the `fid_` variable in the Tcl source file; an example appears in the two-sender version below. The same flow ID is used for both directions of a TCP connection.
9. the source node (0.0), in form (node . connectionID). ConnectionID numbers here are simply an abstraction for connection endpoints; while they superficially resemble port numbers, the node in question need not even simulate IP, and each connection has a unique connectionID at each end. ConnectionID numbers start at 0.
10. the destination node (2.0), again with connectionID.
11. the packet sequence number as a TCP packet, starting from 0.
12. a packet identifier uniquely identifying this packet throughout the simulation; when a packet is forwarded on a new link it keeps its old sequence number but gets a new packet identifier.

The three trace lines above represent the arrival of packet 28 at R, the enqueuing of packet 28, and then the dropping of the packet. All these happen at the same instant.

Mixed in with the event records are **variable-trace** records, indicating a particular variable has been changed. Here are two examples from t=0.3833:

```
0.38333  0  0  2  0  ack_ 3
0.38333  0  0  2  0  cwnd_ 5.000
```

The format of these lines is

1. time
2. source node of the flow
3. source port (as above, an abstract connection endpoint, not a simulated TCP port)
4. destination node of the flow
5. destination port
6. name of the traced variable

7. value of the traced variable

The two variable-trace records above are from the instant when the variable `cwnd_` was set to 5. It was initially 1 at the start of the simulation, and was incremented upon arrival of each of ack0, ack1, ack2 and ack3. The first line shows the ack counter reaching 3 (that is, the arrival of ack3); the second line shows the resultant change in `cwnd_`.

The graph above of cwnd v time was made by selecting out these `cwnd_` lines and plotting the first field (time) and the last. (Recall that during the linear-increase phase `cwnd` is incremented by 1.0/cwnd with each arriving new ACK.)

The last ack in the tracefile is

> 9.98029 0 0 2 0 ack_ 808

Since ACKs started with number 0, this means we sent 809 packets successfully. The theoretical bandwidth was 100 packets/sec × 10 sec = 1000 packets, so this is about an 81% goodput. Use of the `ack_` value this way tells us how much data was actually delivered. An alternative statistic is the final value of `maxseq_` which represents the number of distinct packets sent; the last `maxseq_` line is

> 9.99029 0 0 2 0 maxseq_ 829

As can be seen from the `cwnd`-v-time graph above, slow start ends around T=2.0. If we measure goodput from then until the end, we do a little better than 81%. The first data packet sent after T=2.0 is at 2.043184; it is data packet 72. 737 packets are sent from packet 72 until packet 808 at the end; 737 packets in 8.0 seconds is a goodput of 92%.

It is not necessary to use the tracefile to get the final values of TCP variables such as `ack_` and `maxseq_`; they can be printed from within the Tcl script's `finish()` procedure. The following example illustrates this, where `ack_` and `maxseq_` come from the connection `tcp0`. The `global` line lists global variables that are to be made available within the body of the procedure; `tcp0` must be among these.

```
proc finish {} {
        global ns nf f tcp0
        $ns flush-trace
        close $namfile
        close $tracefile
        set lastACK [$tcp0 set ack_]
        set lastSEQ [$tcp0 set maxseq_]
        puts stdout "final ack: $lastACK, final seq num: $lastSEQ"
        exit 0
}
```

For TCP sending agents, useful member variables to set include:

- `class_`: the identifying number of a flow
- `window_`: the maximum window size; the default is much too small.
- `packetSize_`: we set this to 960 above so the total packet size would be 1000.

Useful member variables either to trace or to print at the simulation's end include:

- `maxseq_`: the number of the last packet sent, starting at 1 for data packets
- `ack_`: the number of the last ACK received

- cwnd_: the current value of the congestion window
- nrexmitpack_: the number of retransmitted packets

To get a count of the data actually received, we need to look at the TCPsink object, $end0 above. There is no packet counter here, but we can retrieve the value bytes_ representing the total number of bytes received. This will include 40 bytes from the threeway handshake which can either be ignored or subtracted:

```
set ACKed [expr round ( [$end0 set bytes_] / 1000.0)]
```

This is a slightly better estimate of goodput. In very long simulations, however, this (or any other) byte count will wrap around long before any of the packet counters wrap around.

In the example above every packet event was traced, a consequence of the line

```
$ns trace-all $trace
```

We could instead have asked only to trace particular links. For example, the following line would request tracing for the bottleneck (R⟶B) link:

```
$ns trace-queue $R $B $trace
```

This is often useful when the overall volume of tracing is large, and we are interested in the bottleneck link only. In long simulations, full tracing can increase the runtime 10-fold; limiting tracing only to what is actually needed can be quite important.

16.2.3 Single Losses

By examining the basic1.tr file above for packet-drop records, we can confirm that only a single drop occurs at the end of each tooth, as was argued in *13.8 Single Packet Losses*. After slow start finishes at around T=2, the next few drops are at T=3.963408, T=5.909568 and T=7.855728. The first of these drops is of Data[254], as is shown by the following record:

```
d 3.963408 1 2 tcp 1000 ------- 0 0.0 2.0 254 531
```

Like most "real" implementations, the ns-2 implementation of TCP increments cwnd (cwnd_ in the tracefile) by 1/cwnd on each new ACK (*13.2.1 Per-ACK Responses*). An additional packet is sent by A whenever cwnd is increased this way past another whole number; that is, whenever floor(cwnd) increases. At T=3.95181, cwnd_ was incremented to 20.001, triggering the double transmission of Data[253] and the doomed Data[254]. At this point the RTT is around 190 ms.

The loss of Data[254] is discovered by Fast Retransmit when the third dupACK[253] arrives. The first ACK[253] arrives at A at T=4.141808, and the dupACKs arrive every 10 ms, clocked by the 10 ms/packet transmission rate of R. Thus, A detects the loss at T=4.171808; at this time we see cwnd_ reduced by half to 10.465; the tracefile times for variables are only to 5 decimal places, so this is recorded as

```
4.17181  0  0  2  0  cwnd_ 10.465
```

That represents an elapsed time from when Data[254] was dropped of 207.7 ms, more than one RTT. As described in *13.8 Single Packet Losses*, however, A stopped incrementing cwnd_ when the first ACK[253] arrived at T=4.141808. The value of cwnd_ at that point is only 20.931, not quite large enough to trigger transmission of another back-to-back pair and thus eventually a second packet drop.

16.2.4 Reading the Tracefile in Python

Deeper analysis of ns-2 data typically involves running some sort of script on the tracefiles; we will mostly use the Python (python3) language for this, although the awk language is also traditional. The following is the programmer interface to a simple module (library) nstrace.py:

- **nsopen**(filename): opens the tracefile
- **isEvent**(): returns `true` if the current line is a normal ns-2 event record
- **isVar**(): returns `true` if the current line is an ns-2 variable-trace record
- **isEOF**(): returns `true` if there are no more tracefile lines
- **getEvent**(): returns a twelve-element tuple of the ns-2 **event**-trace values, each cast to the correct type. The ninth and tenth values, which are node.port pairs in the tracefile, are returned as (node port) sub-tuples.
- **getVar**(): returns a seven-element tuple of ns-2 **var**iable-trace values
- **skipline**(): skips the current line (useful if we are interested only in event records, or only in variable-trace records, and want to ignore the other type of record)

We will first make use of this in *16.2.6.1 Link utilization measurement*; see also *16.4 TCP Loss Events and Synchronized Losses*. The nstrace.py file above includes regular-expression checks to verify that each tracefile line has the correct format, but, as these are slow, they are *disabled* by default. Enabling these checks is potentially useful, however, if some wireless trace records are also included.

16.2.5 The nam Animation

Let us now re-examine the nam animation, in light of what can be found in the trace file.

At T=0.120864, the first 1000-byte data packet is sent (at T=0 a 40-byte SYN packet is sent); the actual packet identification number is 1 so we will refer to it as Data[1]. At this point `cwnd_` = 2, so Data[2] is enqueued at this same time, and sent at T=0.121664 (the delay exactly matches the A–R link's bandwidth of 8000 bits in 0.0008 sec). The first loss occurs at T=0.58616, of Data[28]; at T=0.59616 Data[30] is lost. (Data[29] was not lost because R was able to send a packet and thus make room).

From T=.707392 to T=.777392 we begin a string of losses: packets 42, 44, 46, 48, 50, 52, 54 and 56.

At T=0.76579 the first ACK[27] makes it back to A. The first dupACK[27] arrives at T=0.77576; another arrives at T=0.78576 (10 ms later, exactly the bottleneck per-packet time!) and the third dupACK arrives at T=0.79576. At this point, Data[28] is retransmitted and cwnd is halved from 29 to 14.5.

At T=0.985792, the sender receives ACK[29]. DupACK[29]'s are received at T=1.077024, T=1.087024, T=1.097024 and T=1.107024. Alas, this is TCP Reno, in Fast Recovery mode, and it is not implementing Fast Retransmit while Fast Recovery is going on (TCP NewReno in effect fixes this). Therefore, the connection experiences a **hard timeout** at T=1.22579; the last previous event was at T=1.107024. At this point ssthresh is set to 7 and cwnd drops to 1. Slow start is used up to ssthresh = 7, at which point the sender switches to the linear-increase phase.

16.2.6 Single-sender Throughput Experiments

According to the theoretical analysis in *13.7 TCP and Bottleneck Link Utilization*, a queue size of close to zero should yield about a 75% bottleneck utilization, a queue size such that the mean cwnd equals the transit capacity should yield about 87.5%, and a queue size equal to the transit capacity should yield close to 100%. We now test this.

We first increase the per-link propagation times in the basic1.tcl simulation above to 50 and 100 ms:

```
$ns duplex-link $A $R  10Mb  50ms DropTail
$ns duplex-link $R $B 800Kb 100ms DropTail
```

The bottleneck link here is 800 Kb, or 100 Kbps, or 10 ms/packet, so these propagation-delay changes mean a round-trip transit capacity of 30 packets (31 if we include the bandwidth delay at R). In the table below, we run the simulation while varying the queue-limit parameter from 3 to 30. The simulations run for 1000 seconds, to minimize the effect of slow start. Tracing is disabled to reduce runtimes. The "received" column gives the number of distinct packets received by B; if the link utilization were 100% then in 1,000 seconds B would receive 100,000 packets.

queue_limit	received	utilization %, R→B
3	79767	**79.8**
4	80903	**80.9**
5	83313	**83.3**
8	87169	**87.2**
10	89320	**89.3**
12	91382	**91.4**
16	94570	**94.6**
20	97261	**97.3**
22	98028	**98.0**
26	99041	**99.0**
30	99567	**99.6**

In ns-2, every arriving packet is first enqueued, even if it is immediately dequeued, and so queue-limit cannot actually be zero. A queue-limit of 1 or 2 gives very poor results, probably because of problems with slow start. The run here with queue-limit = 3 is not too far out of line with the 75% predicted by theory for a queue-limit close to zero. When queue-limit is 10, then cwnd will range from 20 to 40, and the link-unsaturated and queue-filling phases should be of equal length. This leads to a theoretical link utilization of about (75%+100%)/2 = 87.5%; our measurement here of 89.3% is in good agreement. As queue-limit continues to increase, the link utilization rapidly approaches 100%, again as expected.

16.2.6.1 Link utilization measurement

In the experiment above we estimated the utilization of the R→B link by the number of distinct packets arriving at B. But packet duplicate transmissions sometimes occur as well (see *16.2.6.4 Packets that are delivered twice*); these are part of the R→B link utilization but are hard to estimate (nominally, most packets retransmitted by A are *dropped* by R, but not all).

If desired, we can get an exact value of the R→B link utilization through analysis of the ns-2 trace file. In this file R is node 1 and B is node 2 and our flow is flow 0; we look for all lines of the form

> - *time* **1 2** tcp 1000 ------- **0** *0.0 2.0 x y*

that is, with field1 = '-', field3 = 1, field4 = 2, field6 = 1000 and field8 = 0 (if we do not check field6=1000 then we count one extra packet, a simulated SYN). We then simply count these lines; here is a simple script to do this in Python using the nstrace module above:

```
#!/usr/bin/python3
import nstrace
import sys

def link_count(filename):
   SEND_NODE = 1
   DEST_NODE = 2
   FLOW = 0
   count = 0

   nstrace.nsopen(filename)
   while not nstrace.isEOF():
       if nstrace.isEvent():
           (event, time, sendnode, dest, dummy, size, dummy, flow, dummy, dummy, dummy, dur
           if (event == "-" and sendnode == SEND_NODE and dest == DEST_NODE and size >= 100
               count += 1
       else:
           nstrace.skipline()
   print ("packet count:", count);

link_count(sys.argv[1])
```

For completeness, here is the same program implemented in the Awk scripting language.

```
BEGIN  {count=0; SEND_NODE=1; DEST_NODE=2; FLOW=0}
$1 == "-" { if ($3 == SEND_NODE && $4 == DEST_NODE && $6 >= 1000 && $8 == FLOW) {
               count++;
            }
          }
END    {print count;}
```

16.2.6.2 ns-2 command-line parameters

Experiments where we vary one parameter, *eg* `queue-limit`, are facilitated by passing in the parameter value on the command line. For example, in the basic1.tcl file we can include the following:

```
set queuesize $argv
...
$ns queue-limit $R $B $queuesize
```

Then, from the command line, we can run this as follows:

```
ns basic1.tcl 5
```

If we want to run this simulation with parameters ranging from 0 to 10, a simple shell script is

```
queue=0
while [ $queue -le 10 ]
do
        ns basic1.tcl $queue
        queue=$(expr $queue + 1)
done
```

If we want to pass multiple parameters on the command line, we use `lindex` to separate out arguments from the $argv string; the first argument is at position 0 (in bash and awk scripts, by comparison, the first argument is $1). For two optional parameters, the first representing `queuesize` and the second representing `endtime`, we would use

```
if { $argc >= 1 } {
   set queuesize [expr [lindex $argv 0]]
}
if { $argc >= 2 } {
   set endtime [expr [lindex $argv 1]]
}
```

16.2.6.3 Queue utilization

In our previous experiment we examined link utilization when `queue-limit` was smaller than the bandwidth×delay product. Now suppose `queue-limit` is greater than the bandwidth×delay product, so the bottleneck link is essentially never idle. We calculated in *13.7 TCP and Bottleneck Link Utilization* what we might expect as an average queue utilization. If the transit capacity is 30 packets and the queue capacity is 40 then $cwnd_{max}$ would be 70 and $cwnd_{min}$ would be 35; the queue utilization would vary from 70-30 = 40 down to 35-30 = 5, averaging around (40+5)/2 = 22.5.

Let us run this as an ns-2 experiment. As before, we set the A–R and R–B propagation delays to 50 ms and 100 ms respectively, making the RTT_{noLoad} 300 ms, for about 30 packets in transit. We also set the `queue-limit` value to 40. The simulation runs for 1000 seconds, enough, as it turns out, for about 50 TCP sawteeth.

At the end of the run, the following Python script maintains a running time-weighted average of the queue size. Because the queue capacity exceeds the total transit capacity, the queue is seldom empty.

```
#!/usr/bin/python3
import nstrace
import sys

def queuesize(filename):
   QUEUE_NODE = 1
   nstrace.nsopen(filename)
   sum = 0.0
   size= 0
   prevtime=0
   while not nstrace.isEOF():
       if nstrace.isEvent():           # counting regular trace lines
           (event, time, sendnode, dnode, proto, dummy, dummy, flow, dummy, dummy, seqno, p
           if (sendnode != QUEUE_NODE): continue
           if (event == "r"): continue
```

```
            sum += size * (time -prevtime)
            prevtime = time
            if (event=='d'): size -= 1
            elif (event=="-"): size -= 1
            elif (event=="+"): size += 1
        else:
            nstrace.skipline()

    print("avg queue=", sum/time)

queuesize(sys.argv[1])
```

The answer we get for the average queue size is about 23.76, which is in good agreement with our theoretical value of 22.5.

16.2.6.4 Packets that are delivered twice

Every dropped TCP packet is ultimately *transmitted* twice, but classical TCP theory suggests that relatively few packets are actually delivered twice. This is pretty much true once the TCP sawtooth phase is reached, but can fail rather badly during slow start.

The following Python script will count packets *delivered* two or more times. It uses a dictionary, COUNTS, which is indexed by sequence numbers.

```
#!/usr/bin/python3
import nstrace
import sys

def dup_counter(filename):
    SEND_NODE = 1
    DEST_NODE = 2
    FLOW = 0
    count = 0
    COUNTS = {}
    nstrace.nsopen(filename)
    while not nstrace.isEOF():
        if nstrace.isEvent():
            (event, time, sendnode, dest, dummy, size, dummy, flow, dummy, dummy, seqno, dur
            if (event == "r" and dest == DEST_NODE and size >= 1000 and flow == FLOW):
                if (seqno in COUNTS):
                    COUNTS[seqno] += 1
                else:
                    COUNTS[seqno] = 1
        else:
            nstrace.skipline()
    for seqno in sorted(COUNTS.keys()):
        if (COUNTS[seqno] > 1): print(seqno, COUNTS[seqno])

dup_counter(sys.argv[1])
```

When run on the basic1.tr file above, it finds 13 packets delivered twice, with TCP sequence numbers 43, 45, 47, 49, 51, 53, 55, 57, 58, 59, 60, 61 and 62. These are sent the second time between T=1.437824 and

T=1.952752; the first transmissions are at times between T=0.83536 and T=1.046592. If we look at our `cwnd`-v-time graph above, we see that these first transmissions occurred during the gap between the end of the unbounded slow-start phase and the beginning of threshold-slow-start leading up to the TCP sawtooth. Slow start, in other words, is messy.

16.2.6.5 Loss rate versus `cwnd`: part 1

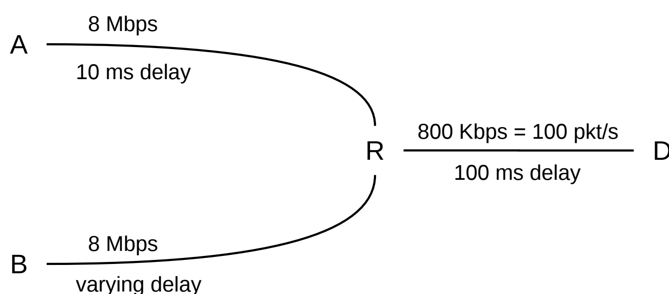
If we run the basic1.tcl simulation above until time 1000, there are 94915 packets acknowledged and 512 loss events. This yields a loss rate of p = 512/94915 = 0.00539, and so by the formula of *14.5 TCP Reno loss rate versus cwnd* we should expect the average `cwnd` to be about $1.225/\sqrt{p} \simeq 16.7$. The true average `cwnd` is the number of packets sent divided by the elapsed time in RTTs, but as RTTs are not constant here (they get significantly longer as the queue fills), we turn to an approximation. From *16.2.1 Graph of cwnd v time* we saw that the peak `cwnd` was 20.935; the mean `cwnd` should thus be about 3/4 of this, or 15.7. While not perfect, agreement here is quite reasonable.

See also *16.4.3 Loss rate versus cwnd: part 2*.

16.3 Two TCP Senders Competing

Now let us create a simulation in which two TCP Reno senders compete for the bottleneck link, and see how fair an allocation each gets. According to the analysis in *14.3 TCP Fairness with Synchronized Losses*, this is really a test of the synchronized-loss hypothesis, and so we will also examine the ns-2 trace files for losses and loss responses. We will start with "classic" TCP Reno, but eventually also consider SACK TCP. Note that, in terms of packet losses in the immediate vicinity of any one queue-filling event, we can expect TCP Reno and SACK TCP to behave identically; they differ only in how they *respond* to losses.

The initial topology will be as follows (though we will very soon raise the bandwidths tenfold, though not the propagation delays):



Broadly speaking, the simulations here will demonstrate that the longer-delay B–D connection receives less bandwidth than the A–D connection, but not quite so much less as was predicted in *14.3 TCP Fairness with Synchronized Losses*. The synchronized-loss hypothesis increasingly fails as the B–R delay increases, in that the B–D connection begins to escape some of the packet-loss events experienced by the A–D connection.

We admit at the outset that we will not, however, obtain a *quantitative* answer to the question of bandwidth allocation. In fact, as we shall see, we run into some difficulties even formulating the proper question. In the course of developing the simulation, we encounter several potential problems:

1. The two senders can become synchronized in an unfortunate manner
2. When we resolve the previous issue by introducing randomness, the bandwidth division is sensitive to the method selected
3. As R's queue fills, the RTT may increase significantly, thus undermining RTT-based measurements (*16.3.9 The RTT Problem*)
4. Transient queue spikes may introduce unexpected losses
5. Coarse timeouts may introduce additional unexpected losses

The experiments and analyses below divide into two broad categories. In the first category, we make use only of the final goodput measurements for the two connections. We consider the first two points of the list above in *16.3.4 Phase Effects*, and the third in *16.3.9 The RTT Problem* and *16.3.10 Raising the Bandwidth*. The first category concludes with some simple loss modeling in *16.3.10.1 Possible models*.

In the second category, beginning at *16.4 TCP Loss Events and Synchronized Losses*, we make use of the ns-2 tracefiles to extract information about packet losses and the extent to which they are synchronized. Examples related to points four and five of the list above are presented in *16.4.1 Some TCP Reno cwnd graphs*. The second category concludes with *16.4.2 SACK TCP and Avoiding Loss Anomalies*, in which we demonstrate that SACK TCP is, in terms of loss and recovery, much better behaved than TCP Reno.

16.3.1 The Tcl Script

Below is a simplified version of the ns-2 script for our simulation; the full version is at basic2.tcl. The most important variable is the additional one-way delay on the B–R link, here called `delayB`. Other defined variables are `queuesize` (for R's queue_limit), `bottleneckBW` (for the R–D bandwidth), `endtime` (the length of the simulation), and `overhead` (for introducing some small degree of randomization, below). As with basic1.tcl, we set the packet size to 1000 bytes total (960 bytes TCP portion), and increase the advertised window size to 65000 (so it is never the limiting factor).

We have made the `delayB` value be a command-line parameter to the Tcl file, so we can easily experiment with changing it (in the full version linked to above, `overhead`, `bottleneckBW`, `endtime` and `queuesize` are also parameters). The one-way propagation delay for the A–D path is 10 ms + 100 ms = 110 ms, making the RTT 220 ms plus the bandwidth delays. At the bandwidths above, the bandwidth delay for data packets adds an additional 11 ms; ACKs contribute an almost-negligible 0.44 ms. We return to the script variable `RTTNL`, intended to approximate RTT_{noLoad}, below.

With `endtime`=300, the theoretical maximum number of data packets that can be delivered is 30,000. If `bottleneckBW` = 0.8 Mbps (100 packets/sec) then the R–D link can hold ten R⟶D data packets in transit, plus another ten D⟶R ACKs.

In the `finish()` procedure we have added code to print out the number of packets received by D for each connection; we could also extract this from the trace file.

To gain better control over printing, we have used the `format` command, which works something like C's `sprintf`. It returns a string containing spliced-in numeric values replacing the corresponding `%d` or `%f` tokens in the control string; this returned string can then be printed with `puts`.

The full version linked to above also contains some nam directives, support for command-line arguments, and arranges to name any tracefiles with the same base filename as the Tcl file.

```
# NS basic2.tcl example of two TCPs competing on the same link.

# Create a simulator object
set ns [new Simulator]

#Open the trace file
set trace [open basic2.tr w]
$ns trace-all $trace

############## some globals (modify as desired) ##############

# queuesize on bottleneck link
set queuesize 20
# default run time, in seconds
set endtime 300
# "overhead" of D>0 introduces a uniformly randomized delay d, 0⩽d⩽D; 0 turns it off.
set overhead 0
# delay on the A--R link, in ms
set basedelay 10
# ADDITIONAL delay on the B--R link, in ms
set delayB 0
# bandwidth on the bottleneck link, either 0.8 or 8.0 Mbit
set bottleneckBW 0.8
# estimated no-load RTT for the first flow, in ms
set RTTNL 220

############## arrange for output ##############

set outstr [format "parameters: delayB=%f overhead=%f bottleneckBW=%f" $delayB $overhead $b
puts stdout $outstr

# Define a 'finish' procedure that prints out progress for each connection
proc finish {} {
        global ns tcp0 tcp1 end0 end1 queuesize trace delayB overhead RTTNL
        set ack0 [$tcp0 set ack_]
        set ack1 [$tcp1 set ack_]
        # counts of packets *received*
        set recv0 [expr round ( [$end0 set bytes_] / 1000.0)]
        set recv1 [expr round ( [$end1 set bytes_] / 1000.0)]
        # see numbers below in topology-creation section
        set rttratio [expr (2.0*$delayB+$RTTNL)/$RTTNL]
        # actual ratio of throughputs fast/slow; the 1.0 forces floating point
        set actualratio [expr 1.0*$recv0/$recv1]
        # theoretical ratio fast/slow with squaring; see text for discussion of ratio1 and
        set rttratio2 [expr $rttratio*$rttratio]
        set ratio1 [expr $actualratio/$rttratio]
        set ratio2 [expr $actualratio/$rttratio2]
        set outstr [format "%f %f %d %d %f %f %f %f %f" $delayB $overhead $recv0 $recv1 $rt
        puts stdout $outstr
        $ns flush-trace
        close $trace
        exit 0
}
```

```
############### create network topology ##############

# A
#   \
#     \
#      R---D (Destination)
#     /
#   /
# B

#Create four nodes
set A [$ns node]
set B [$ns node]
set R [$ns node]
set D [$ns node]

set fastbw [expr $bottleneckBW * 10]
#Create links between the nodes; propdelay on B--R link is 10+$delayB ms
$ns duplex-link $A $R ${fastbw}Mb ${basedelay}ms DropTail
$ns duplex-link $B $R ${fastbw}Mb [expr $basedelay + $delayB]ms DropTail
# this last link is the bottleneck; 1000 bytes at 0.80Mbps => 10 ms/packet
# A--D one-way delay is thus 110 ms prop + 11 ms bandwidth
# the values 0.8Mb, 100ms are from Floyd & Jacobson

$ns duplex-link $R $D ${bottleneckBW}Mb 100ms DropTail

$ns queue-limit $R $D $queuesize

############## create and connect TCP agents, and start ##############

Agent/TCP set window_ 65000
Agent/TCP set packetSize_ 960
Agent/TCP set overhead_ $overhead

#Create a TCP agent and attach it to node A, the delayed path
set tcp0 [new Agent/TCP/Reno]
$tcp0 set class_ 0
# set the flowid here, used as field 8 in the trace
$tcp0 set fid_ 0
$tcp0 attach $trace
$tcp0 tracevar cwnd_
$tcp0 tracevar ack_
$ns attach-agent $A $tcp0

set tcp1 [new Agent/TCP/Reno]
$tcp1 set class_ 1
$tcp1 set fid_ 1
$tcp1 attach $trace
$tcp1 tracevar cwnd_
$tcp1 tracevar ack_
$ns attach-agent $B $tcp1

set end0 [new Agent/TCPSink]
```

```
$ns attach-agent $D $end0

set end1 [new Agent/TCPSink]
$ns attach-agent $D $end1

#Connect the traffic source with the traffic sink
$ns connect $tcp0 $end0
$ns connect $tcp1 $end1

#Schedule the connection data flow
set ftp0 [new Application/FTP]
$ftp0 attach-agent $tcp0

set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp1

$ns at 0.0 "$ftp0 start"
$ns at 0.0 "$ftp1 start"
$ns at $endtime "finish"

#Run the simulation
$ns run
```

16.3.2 Equal Delays

We first try this out by running the simulation with equal delays on both A–R and R–B. The following values are printed out (arranged here vertically to allow annotation)

value	variable	meaning
0.000000	`delayB`	Additional B–R propagation delay, compared to A–R delay
0.000000	`overhead`	overhead; a value of 0 means this is effectively disabled
14863	`recv0`	Count of cumulative A–D packets received at D (that is, goodput)
14771	`recv1`	Count of cumulative B–D packets received at D (again, goodput)
1.000000	`rttratio`	RTT_ratio: B–D/A–D (long/short)
1.000000	`rttratio2`	The square of the previous value
1.006228	`actualratio`	Actual ratio of A–D/B–D goodput, that is, 14863/14771 (note change in order versus RTT_ratio)
1.006228	`ratio1`	actual_ratio/RTT_ratio
1.006228	`ratio2`	actual_ratio/RTT_ratio^2

The one-way A–D propagation delay is 110 ms; the bandwidth delays as noted above amount to 11.44 ms, 10 ms of which is on the R–D link. This makes the A–D RTT_{noLoad} about 230 ms. The B–D delay is, for the time being, the same, as `delayB` = 0. We set `RTTNL` = 220, and calculate the RTT ratio (within Tcl, in the finish() procedure) as (2×`delayB` + `RTTNL`)/`RTTNL`. We really should use `RTTNL`=230 instead of 220 here, but 220 will be closer when we later change `bottleneckBW` to 8.0 Mbit/sec rather than 0.8, below. Either way, the difference is modest.

Note that the above RTT calculations are for when the queue at R is empty; when the queue contains 20 packets this adds another 200 ms to the A⟶D and B⟶D times (the reverse direction is unchanged). This may make a rather large difference to the RTT ratio, and we will address it below, but does not matter yet

because the propagation delays so far are identical.

In the model of *14.3.3 TCP RTT bias* we explored a model in which we expect that **ratio2**, above, would be about 1.0. The final paragraph of *14.5.2 Unsynchronized TCP Losses* hints at a possible model (the $\gamma=\lambda$ case) in which **ratio1** would be about 1.0. We will soon be in a position to test these theories experimentally. Note that the order of B and A in the goodput and RTT ratios is reversed, to reflect the expected inverse relationship.

In the 300-second run here, 14863+14771 = 29634 packets are sent. This means that the bottleneck link is 98.8% utilized.

In *16.4.1 Some TCP Reno cwnd graphs* we will introduce a script (teeth.py) to count and analyze the teeth of the TCP sawtooth. Applying this now, we find there are 67 loss events total, and thus 67 teeth, and in *every* loss event each flow loses exactly one packet. This is remarkably exact conformance to the synchronized-loss hypothesis of *14.3.3 TCP RTT bias*. So far.

16.3.3 Unequal Delays

We now begin increasing the additional B–R delay (`delayB`). Some preliminary data are in the table below, and point to a significant problem: goodput ratios in the last column here are not varying smoothly. The value of 0 ms in the first row means that the B–R delay is equal to the A–R delay; the value of 110 ms in the last row means that the B–D RTT_{noLoad} is double the A–D RTT_{noLoad}. The column labeled (RTT ratio)^2 is the expected goodput ratio, according to the model of *14.3 TCP Fairness with Synchronized Losses*; the actual A–D/B–D goodput ratio is in the final column.

delayB	RTT ratio	A–D goodput	B–D goodput	(RTT ratio)^2	goodput ratio
0	1.000	14863	14771	1.000	1.006
5	1.045	4229	24545	1.093	**0.172**
23	1.209	22142	6879	1.462	**3.219**
24	1.218	17683	9842	1.484	1.797
25	1.227	14958	13754	1.506	1.088
26	1.236	24034	5137	1.529	**4.679**
35	1.318	16932	11395	1.738	1.486
36	1.327	25790	3603	1.762	**7.158**
40	1.364	20005	8580	1.860	2.332
41	1.373	24977	4215	1.884	**5.926**
45	1.409	18437	10211	1.986	1.806
60	1.545	18891	9891	2.388	1.910
61	1.555	25834	3135	2.417	**8.241**
85	1.773	20463	8206	3.143	2.494
110	2.000	22624	5941	4.000	3.808

For a few rows, such as the first and the last, agreement between the last two columns is quite good. However, there are some decidedly anomalous cases in between (particularly the numbers in **bold**). As `delayB` changes from 35 to 36, the goodput ratio jumps from 1.486 to 7.158. Similar dramatic changes in goodput appear as `delayB` ranges through the sets {23, 24, 25, 26}, {40, 41, 45}, and {60, 61}. These values were, admittedly, specially chosen by trial and error to illustrate relatively discontinuous behavior of the goodput ratio, but, still, what is going on?

16.3.4 Phase Effects

This erratic behavior in the goodput ratio in the table above turns out to be due to what are called **phase effects** in *[FJ92]*; transmissions of the two TCP connections become precisely synchronized in some way that involves a persistent negative bias against one of them. What is happening is that a "race condition" occurs for the last remaining queue vacancy, and one connection consistently loses this race the majority of the time.

16.3.4.1 Single-sender phase effects

We begin by taking a more detailed look at the bottleneck queue when no competition is involved. Consider a single sender A using a **fixed** window size to send to destination B through bottleneck router R (so the topology is A–R–B), and suppose the window size is large enough that R's queue is not empty. For the sake of definiteness, assume R's bandwidth delay is 10 ms/packet; R will send packets every 10 ms, and an ACK will arrive back at A every 10 ms, and A will transmit every 10 ms.

Now imagine that we have an output meter that reports the percentage that has been transmitted of the packet R is currently sending; it will go from 0% to 100% in 10 ms, and then back to 0% for the next packet.

Our first observation is that at each instant when a packet from A fully arrives at R, R is always at exactly the same point in forwarding some earlier packet on towards B; the output meter always reads the same percentage. This percentage is called the **phase** of the connection, sometimes denoted ϕ.

We can determine ϕ as follows. Consider the total elapsed time for the following:

- R finishes transmitting a packet and it arrives at B
- its ACK makes it back to A
- the data packet triggered by that ACK fully arrives at R

In the absence of other congestion, this **R-to-R time** includes no variable queuing delays, and so is constant. The output-meter percentage above is determined by this elapsed time. If for example the R-to-R time is 83 ms, and Data[N] leaves R at T=0, then Data[N+winsize] (sent by A upon arrival of ACK[N]) will arrive at R when R has completed sending packets up through Data[N+8] (80 ms) and is 3 ms into transmitting Data[N+9]. We can get this last as a percentage by dividing 83 by R's 10-ms bandwidth delay and taking the fractional part (in this case, 30%). Because the elapsed R-to-R time here is simply RTT_{noLoad} minus the bandwidth delay at R, we can also compute the phase as

$$\phi = \text{fractional_part}(RTT_{noLoad} \div \text{(R's bandwidth delay)}).$$

If we ratchet up the winsize until the queue becomes full when each new packet arrives, then the phase percentage represents the fraction of the time the queue has a vacancy. In the scenario above, if we start the clock at T=0 when R has finished transmitting a packet, then the queue has a vacancy until T=3 when a new packet from A arrives. The queue is then full until T=10, when R starts transmitting the next packet in its queue.

Finally, even in the presence of competition through R, the phase of a single connection remains constant provided there are no queuing delays along the bidirectional A–B path except at R itself, and there only in the forward direction towards B. Other traffic sent through R can only add delay in integral multiples of R's bandwidth delay, and so cannot affect the A–B phase.

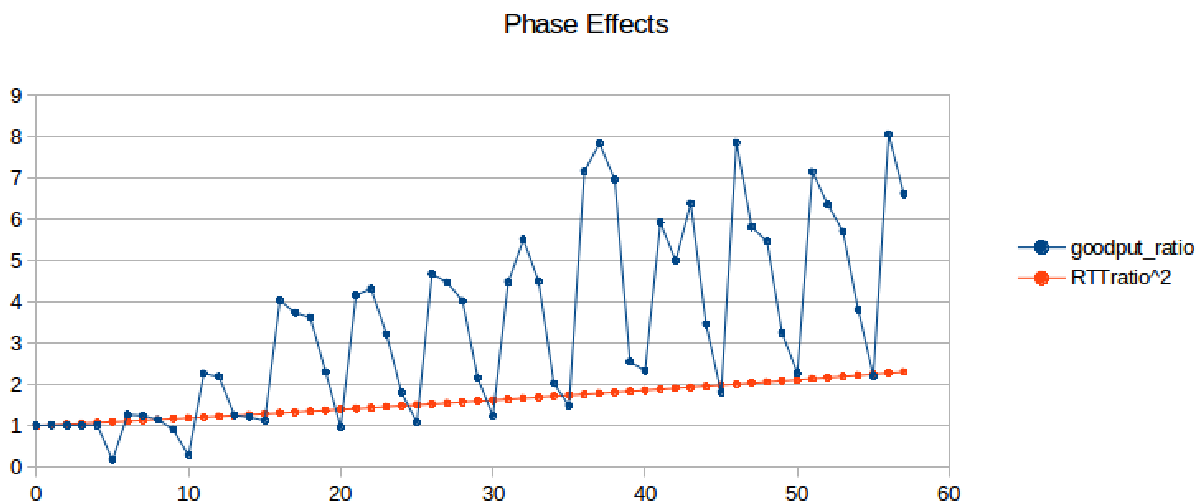
16.3.4.2 Two-sender phase effects

In the present simulation, we can by the remark in the previous paragraph calculate the phase for each sender; let these be ϕ_A and ϕ_B. The significance of phase to competition is that whenever A and B send packets that happen to arrive at R in the same 10-ms interval while R is forwarding some other packet, if the queue has only one vacancy then the connection with the smaller phase will always win it.

As a concrete example, suppose that the respective RTT_{noLoad}'s of A and B are 221 and 263 ms. Then A's phase is 0.1 (fractional_part(221÷10)) and B's is 0.3. The important thing is not that A's packets take less time, but that in the event of a near-tie A's packet must arrive first at R. Imagine that R forwards a B packet and then, four packets (40 ms) later, forwards an A packet. The ACKs elicited by these packets will cause new packets to be sent by A and B; A's packet will arrive first at R followed 2 ms later by B's packet. Of course, R is not likely to send an A packet four packets after *every* B packet, but when it does so, the arrival order is predetermined (in A's favor) rather than random.

Now consider what happens when a packet is dropped. If there is a single vacancy at R, and packets from A and B arrive in a near tie as above, then it will always be B's packet that is dropped. The occasional packet-pair sent by A or B as part of the expansion of `cwnd` will be the ultimate cause of loss events, but the phase effect has introduced a persistent degree of bias in A's favor.

We can visualize phase effects with ns-2 by letting `delayB` range over, say, 0 to 50 in small increments, and plotting the corresponding values of ratio2 (above). Classically we expect ratio2 to be close to 1.00. In the graph below, the blue curve represents the goodput ratio; it shows a marked (though not perfect) periodicity with period 5 ms.



The orange curve represents $(RTT_ratio)^2$; according to *14.3 TCP Fairness with Synchronized Losses* we would expect the blue and orange curves to be about the same. When the blue curve is high, the slower B–D connection is proportionately at an unexpected disadvantage. Seldom do phase effects work in favor of the B–D connection, because A's phase here is quite small (0.144, based on A's exact RTT_{noLoad} of 231.44 ms). (If we change the A–R propagation delay (`basedelay`) to 12 ms, making A's phase 0.544, the blue curve oscillates somewhat more evenly both above and below the orange curve, but still with approximately the same amplitude.)

Recall that a 5 ms change in `delayB` corresponds to a 10 ms change in the A–D connection's RTT, *equal to router R's transmission time*. What is happening here is that as the B–D connection's RTT increases through

a range of 10 ms, it cycles through from phase-effect neutrality to phase-effect deficit and back.

16.3.5 Minimizing Phase Effects

In the real world, the kind of precise transmission synchronization that leads to phase effects is seldom evident, though perhaps this has less to do with rarity and more with the fact that head-to-head TCP competitions are difficult to observe intimately. Usually, however, there seems to be sufficient other traffic present to disrupt the synchronization. How can we break this synchronization in simulations? One way or another, we must inject some degree of **randomization** into the bulk TCP flows.

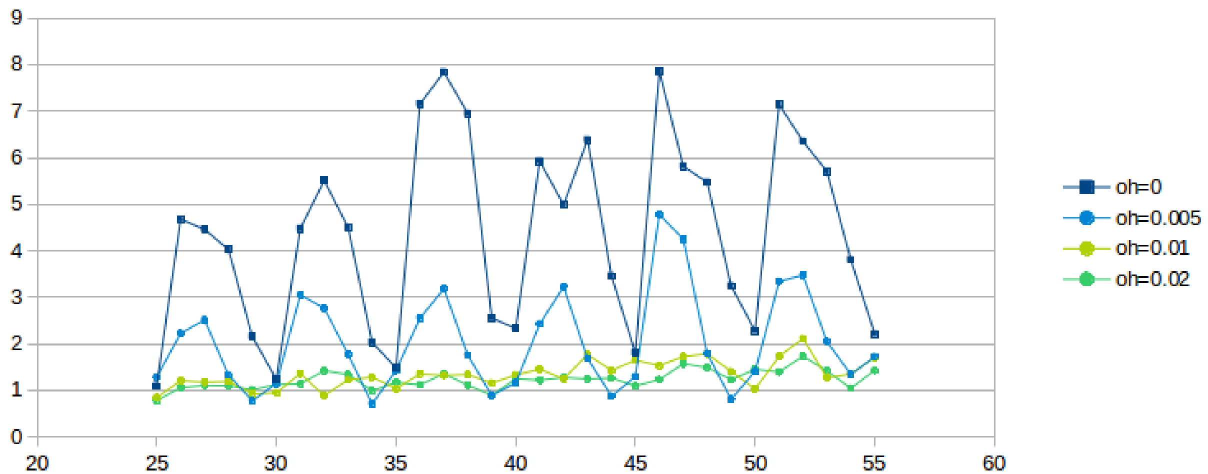
Techniques introduced in *[FJ92]* to break synchronization in ns-2 simulations were random "telnet" traffic – involving smaller packets sent according to a given random distribution – and the use of random-drop queues (not included in the standard ns-2 distribution). The second, of course, means we are no longer simulating FIFO queues.

A third way of addressing phase effects is to make use of the ns-2 `overhead` variable, which introduces some modest randomization in packet-departure times at the TCP sender. Because this technique is simpler, we will start with it. One difference between the use of `overhead` and telnet traffic is that the latter has the effect of introducing delays at all nodes of the network that carry the traffic, not just at the TCP sources.

16.3.6 Phase Effects and `overhead`

For our first attempt at introducing phase-effect-avoiding randomization in the competing TCP flows, we will start with ns-2's TCP `overhead` attribute. This is equal to 0 by default and is measured in units of seconds. If `overhead` > 0, then the TCP source introduces a uniformly distributed random delay of between 0 and `overhead` seconds whenever an ACK arrives and the source is allowed to send a new packet. Because the distribution is uniform, the average delay so introduced is thus `overhead`/2. To introduce an average delay of 10 ms, therefore, one sets `overhead = 0.02`. Packets are always sent in order; if packet 2 is assigned a small `overhead` delay and packet 1 a large `overhead` delay, then packet 2 waits until packet 1 has been sent. For this reason, it is a good idea to keep the average `overhead` delay no more than the average packet interval (here 10 ms).

The following graph shows four curves representing `overhead` values of 0, 0.005, 0.01 and 0.02 (that is, 5 ms, 10 ms and 20 ms). For each curve, ratio1 (not the actual goodput ratio and not ratio2) is plotted as a function of `delayB` as the latter ranges from 25 to 55 ms. The simulations run for 300 seconds, and `bottleneckBW` = 0.8. (We will return to the choice of ratio1 here in *16.3.9 The RTT Problem*; the corresponding ratio2 graph is however quite similar, at least in terms of oscillatory behavior.)



The dark-blue curve for `overhead` = 0 is wildly erratic due to phase effects; the light-blue curve for `overhead` = 0.005 has about half the oscillation. Even the light-green `overhead` = 0.01 curve exhibits some wiggling; it is not until `overhead` = 0.02 for the darker green curve that the graph really settles down. We conclude that the latter two values for `overhead` are quite effective at mitigating phase effects.

One crude way to quantify the degree of graph oscillation is by calculating the mean deviation; the respective deviation values for the curves above are are 1.286, 0.638, 0.136 and 0.090.

Recall that the time to send one packet on the bottleneck link is 0.01 seconds, and that the *average* delay introduced by `overhead` d is d/2; thus, when `overhead` is 0.02 each connection would, *if acting alone*, have an average sender delay equal to the bottleneck-link delay (though `overhead` delay is like propagation delay, and so a high `overhead` will not prevent queue buildup).

Compared to the 10-ms-per-packet R–D transmission time, average delays of 5 and 10 ms per flow (`overhead` of 0.01 and 0.02 respectively) may not seem disproportionate. They are, however, *quite* large when compared to the 1.0 ms bandwidth delay on the A–R and B–R legs. Generally, if the goal is to reduce phase effects then `overhead` should be comparable to the bottleneck-router transmission rate. Using `overhead` > 0 does increase the RTT, but in this case not considerably.

We conclude that using `overhead` to break the synchronization that leads to phase effects appears to have worked, at least in the sense that with the value of `overhead` = 0.02 the goodput ratio increases more-or-less monotonically with increasing `delayB`.

The problem with using `overhead` this way is that it **does not correspond to any physical network delay** or other phenomenon. Its use here represents a decidedly *ad hoc* strategy to introduce enough randomization that phase effects disappear. That said, it does seem to produce results similar to those obtained by injecting random "telnet" traffic, introduced in the following section.

16.3.7 Phase Effects and telnet traffic

We can also introduce anti-phase-effect randomization by making use of the ns-2 telnet application to generate low-to-moderate levels of random traffic. This requires an additional two Agent/TCP objects, representing A–D and B–D telnet connections, to carry the traffic; this telnet traffic will then introduce slight delays

in the corresponding bulk (ftp) traffic. The size of the telnet packets sent is determined by the TCP agents' usual `packetSize_` attribute.

For each telnet connection we create an Application/Telnet object and set its attribute `interval_`; in the script fragment below this is set to **tninterval**. This represents the average packet spacing in seconds; transmissions are then scheduled according to an exponential random distribution with `interval_` as its mean. We remark that "real" telnet terminal-login traffic seldom if ever follows an exponential random distribution, though this is not necessarily a concern as there are other common traffic patterns (*eg* web traffic or database traffic) that fit this model better.

Actual (simulated) transmissions, however, are also constrained by the telnet connection's sliding window. It is quite possible that the telnet application releases a new packet for transmission, but it cannot yet be sent because the telnet TCP connection's sliding window is momentarily frozen, waiting for the next ACK. If the telnet packets encounter congestion and the `interval_` is small enough then the sender may have a backlog of telnet packets in its outbound queue that are waiting for the sliding window to advance enough to permit their departure.

```
set tcp10 [new Agent/TCP]
$ns attach-agent $A $tcp10
set tcp11 [new Agent/TCP]
$ns attach-agent $B $tcp11

set end10 [new Agent/TCPSink]
set end11 [new Agent/TCPSink]
$ns attach-agent $D $end10
$ns attach-agent $D $end11

set telnet0 [new Application/Telnet]
set telnet1 [new Application/Telnet]

set tninterval 0.001    ;# see text for discussion

$telnet0 set interval_ $tninterval
$tcp10 set packetSize_ 210

$telnet1 set interval_ $tninterval
$tcp11 set packetSize_ 210

$telnet0 attach-agent $tcp10
$telnet1 attach-agent $tcp11
```

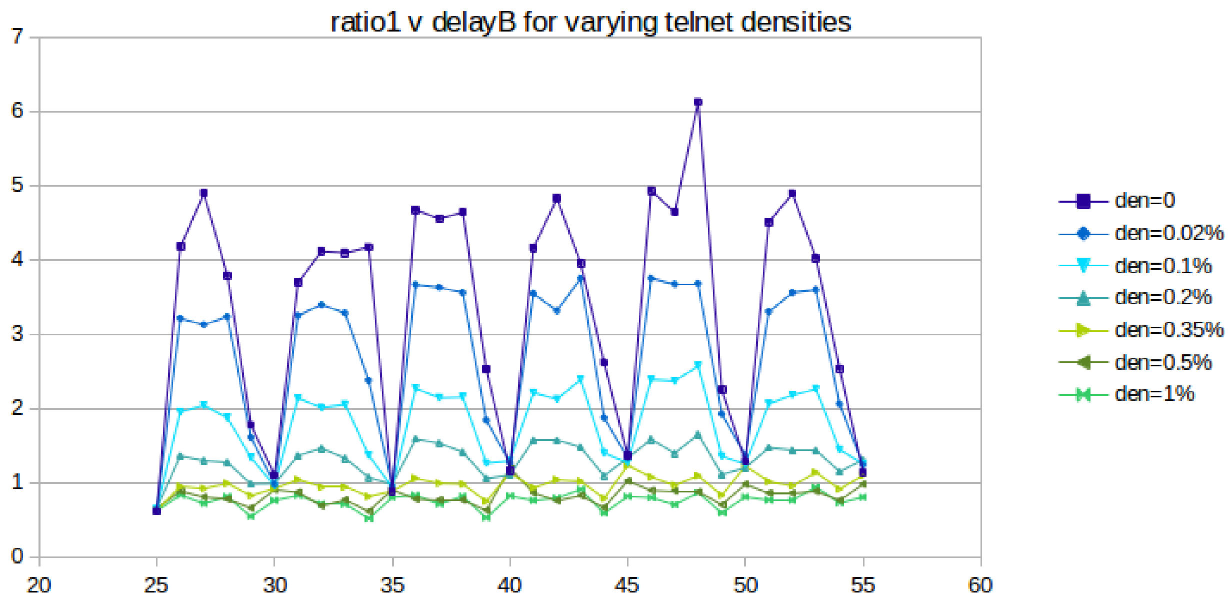
"Real" telnet packets, besides not necessarily being exponentially distributed, are most often quite small. In the simulations here we use an uncharacteristically large size of 210 bytes, leading to a total packet size of 250 bytes after the 40-byte simulated TCP/IP header is attached. We denote the latter number by `actualSize`. See exercise 9.

The bandwidth theoretically consumed by the telnet connection is simply `actualSize`/`$tninterval`; the actual bandwidth may be lower if the telnet packets are encountering congestion as noted above. It is convenient to define an attribute `tndensity` that denotes the fraction of the R–D link's bandwith that the Telnet application will be allowed to use, *eg* 2%. In this case we have

```
$tninterval = actualSize / ($tndensity * $bottleneckBW)
```

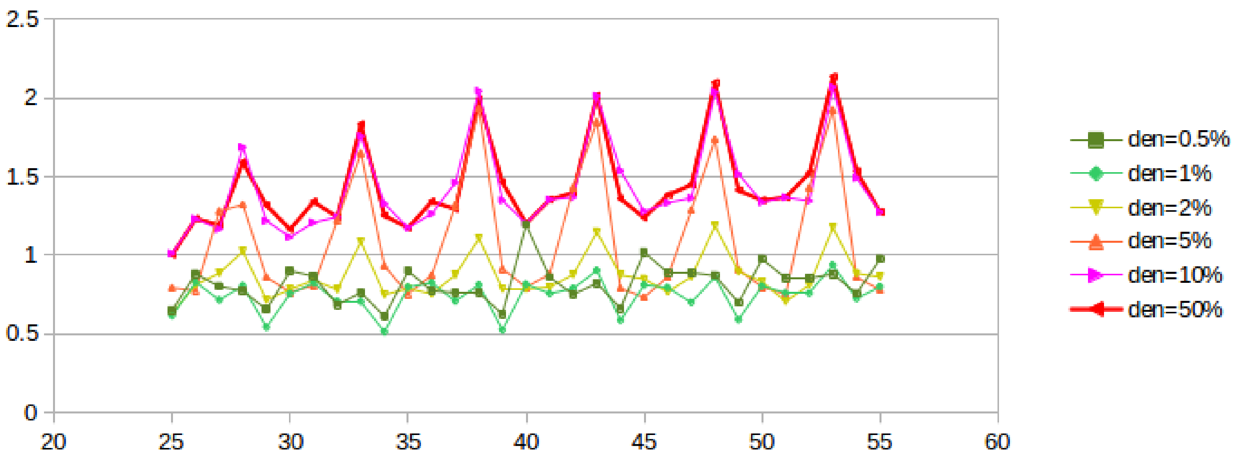
For example, if `actualSize` = 250 bytes, and `$bottleneckBW` corresponds to 1000 bytes every 10 ms, then the telnet connection could saturate the link if its packets were spaced 2.5 ms apart. However, if we want the telnet connection to use up to 5% of the bottleneck link, then `$tninterval` should be 2.5/0.05 = 50 ms (converted for ns-2 to 0.05 sec).

The first question we need to address is whether telnet traffic can sufficiently dampen phase-effect oscillations, and, if so, at what densities. The following graph is the telnet version of that above in *16.3.6 Phase Effects and overhead*; `bottleneckBW` is still 0.8 but the simulations now run for 3000 seconds. The telnet total packet size is 250 bytes. The given telnet density percentages apply to each of the A–D and B–D telnet connections; the total telnet density is thus double the value shown.



As we hoped, the oscillation does indeed substantially flatten out as the telnet density increases from 0 (dark blue) to 1% (bright green); the mean deviation falls from 1.36 to 0.084. So far the use of telnet is very promising.

Unfortunately, if we continue to increase the telnet density past 1%, the oscillation *increases* again, as can be seen in the next graph:



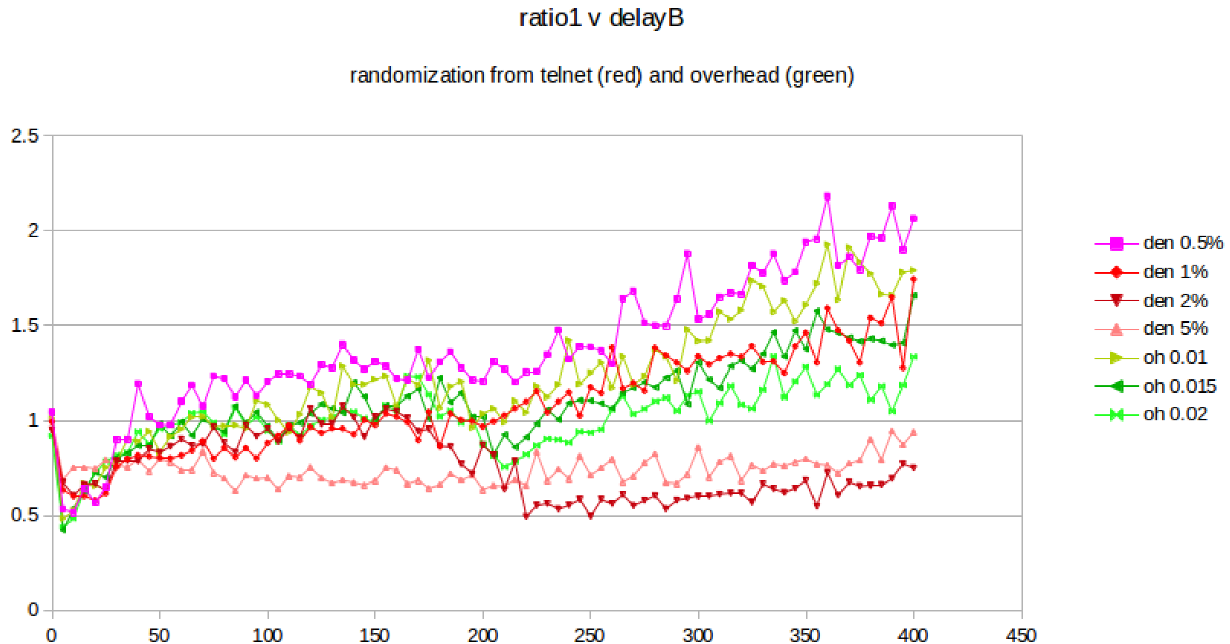
The first two curves, plotted in green, correspond to the "good" densities of the previous graph, with the vertical axis stretched by a factor of two. As densities increase, however, phase-effect oscillation returns, and the curves converge towards the heavier red curve at the top.

What appears to be happening is that, beyond a density of 1% or so, the limiting factor in telnet transmission becomes the telnet sliding window rather than the random traffic generation, as mentioned in the third paragraph of this section. Once the sliding window becomes the limiting factor on telnet packet transmission, the telnet connections behave much like – and become synchronized with – their corresponding bulk-traffic ftp connections. At that point their ability to moderate phase effects is greatly diminished, as actual packet departures no longer have anything to do with the exponential random distribution that generates the packets.

Despite this last issue, the fact that small levels of random traffic can lead to large reductions in phase effects can be taken as evidence that, in the real world, where other traffic sources are ubiquitous, phase effects will seldom be a problem.

16.3.8 `overhead` versus telnet

The next step is to compare the effect on the original bulk-traffic flows of `overhead` randomization and telnet randomization. The graphs below plot ratio1 as a function of `delayB` as the latter ranges from 0 to 400 in increments of 5. The `bottleneckBW` is 0.8 Mbps, the queue capacity at R is 20, and the run-time is 3000 seconds.



The four reddish-hued curves represent the result of using telnet with a packet size of 250, at densities ranging from 0.5% to 5%. These may be compared with the three green curves representing the use of `overhead`, with values 0.01, 0.015 and 0.02. While telnet with a density of 1% is in excellent agreement with the use of `overhead`, it is also apparent that smaller telnet densities give a larger ratio1 while larger densities give a smaller. This raises the awkward possibility that the exact mechanism by which we introduce randomization may have a material effect on the fairness ratios that we ultimately observe. There is no "right" answer here; different randomization sources or levels may simply lead to differing fairness results.

The advantage of using telnet for randomization is that it represents an actual network phenomenon, unlike `overhead`. The drawback to using telnet is that the effect on the bulk-traffic goodput ratio is, as the graph above shows, somewhat sensitive to the exact value chosen for the telnet density.

In the remainder of this chapter, we will continue to use the `overhead` model, for simplicity, though we do not claim this is a universally appropriate approach.

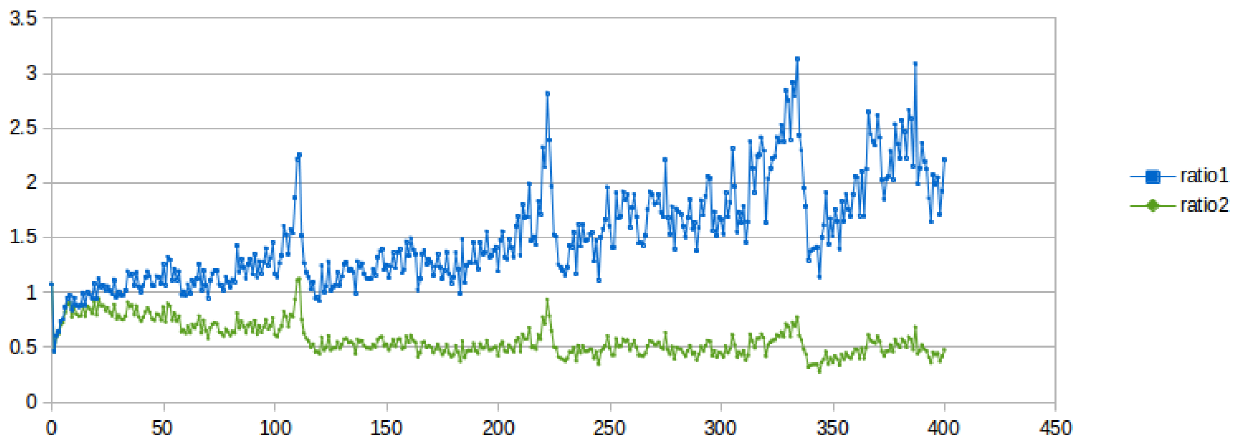
16.3.9 The RTT Problem

In all three of the preceding graphs (*16.3.6 Phase Effects and overhead*, *16.3.7 Phase Effects and telnet traffic* and *16.3.8 overhead versus telnet*), the green curves on the graphs appear to show that, once sufficient randomization has been introduced to disrupt phase effects, ratio1 converges to 1.0. This, however, is in fact an artifact, due to the second flaw in our simulated network: *RTT is not very constant*. While $\text{RTT}_{\text{noLoad}}$ for the A–D link is about 220 ms, queuing delays at R (with `queuesize` = 20) can almost double that by adding up to 20 × 10 ms = 200 ms. This means that the computed values for `RTTratio` are too large, and the computed values for `ratio1` are thus too small. While one approach to address this problem is to keep careful track of $\text{RTT}_{\text{actual}}$, a simpler strategy is to create a simulated network in which the queuing delay is small compared to the propagation delay and so the RTT is relatively constant. We turn to this next.

16.3.10 Raising the Bandwidth

In modern high-bandwidth networks, queuing delays tend to be small compared to the propagation delay; see *13.7 TCP and Bottleneck Link Utilization*. To simulate such a network here, we simply increase the bandwidth of all the links tenfold, while leaving the existing propagation delays the same. We achieve this by setting bottleneckBW = 8.0 instead of 0.8. This makes the A–D RTT_{noLoad} equal to about 221 ms; queuing delays can now amount to at most an additional 20 ms. The value of `overhead` also needs to be scaled down by a factor of 10, to 0.002 sec, to reflect an average delay in the same range as the bottleneck-link packet transmission time.

Here is a graph of results for `bottleneckBW` = 8.0, time = 3000, `overhead` = 0.002 and queuesize = 20. We still use 220 ms as the estimated RTT, though the actual RTT will range from 221 ms to 241 ms. The `delayB` parameter runs from 0 to 400 in steps of 1.0.



The first observation to make is that ratio1 is generally too large and ratio2 is generally too small, when compared to 1.0. In other words, neither is an especially good fit. This appears to be a fairly general phenomenon, in both simulation and the real world: TCP Reno throughput ratios tend to be somewhere between the corresponding RTT ratio and the square of the RTT ratio.

The synchronized-loss hypothesis led to the prediction (*14.3 TCP Fairness with Synchronized Losses*) that the goodput ratio would be close to RTT_ratio^2. As this conclusion appears to fail, the hypothesis too must fail, at least to a degree: it must be the case that not all losses are shared.

Throughout the graph we can observe a fair amount of "noise" variation. Most of this variation appears unrelated to the 5 ms period we would expect for phase effects (as in the graph at *16.3.4.2 Two-sender phase effects*). However, it is important to increment `delayB` in amounts much smaller than 5 ms in order to rule this out, hence the increment of 1.0 here.

There are strong peaks at `delayB` = 110, 220 and 330. These `delayB` values correspond to increasing the RTT by integral multiples 2, 3 and 4 respectively, and the peaks are presumably related to some kind of higher-level phase effect.

16.3.10.1 Possible models

If the synchronized-loss fairness model fails, with what do we replace it? Here are two *ad hoc* options. First, we can try to fit a curve of the form

$$\text{goodput_ratio} = K \times (\text{RTT_ratio})^{\alpha}$$

to the above data. If we do this, the value for the exponent α comes out to about 1.58, sort of a "compromise" between ratio2 (α=2) and ratio1 (α=1), although the value of the exponent here is somewhat sensitive to the details of the simulation.

An entirely different curve to fit to the data, based on the appearance in the graph that ratio2 $\simeq$ 0.5 past 120, is

$$\text{goodput_ratio} = 1/2 \times (\text{RTT_ratio})^2$$

We do not, however, possess for either of these formulas a model for the relative losses in the two primary TCP connections that is precise enough to offer an explanation of the formula (though see the final paragraph of *16.4.2.2 Relative loss rates*).

16.3.10.2 Higher bandwidth and link utilization

One consequence of raising the bottleneck bandwidth is that total link utilization drops, for `delayB` = 0, to 80% of the bandwidth of the bottleneck link, from 98%; this is in keeping with the analysis of *13.7 TCP and Bottleneck Link Utilization*. The transit capacity is 220 packets and another 20 can be in the queue at R; thus an ideal sawtooth would oscillate between 120 and 240 packets. We do have two senders here, but when `delayB` = 0 most losses are synchronized, meaning the two together behave like one sender with an additive-increase value of 2. As `cwnd` varies linearly from 120 to 240, it spends 5/6 of the time below the transit capacity of 220 packets – during which period the average `cwnd` is (120+220)/2 = 170 – and 1/6 of the time with the path 100% utilized; the weighted average estimating total goodput is thus (5/6)×170/220 + (1/6)×1 = 81%.

When `delayB` = 400, combined TCP Reno goodput falls to about 51% of the bandwidth of the bottleneck link. This low utilization, however, is indeed related to loss and timeouts; the corresponding combined goodput percentage for SACK TCP (which as we shall see in *16.4.2 SACK TCP and Avoiding Loss Anomalies* is much better behaved) is 68%.

16.4 TCP Loss Events and Synchronized Losses

If the synchronized-loss model is not entirely accurate, as we concluded from the graph above, what *does* happen with packet losses when the queue fills?

At this point we shift focus from analyzing goodput ratios to analyzing the underlying loss events, using the ns-2 tracefile. We will look at the synchronization of loss events between different connections and at how many individual packet losses may be involved in a single TCP loss response.

One of the conclusions we will reach is that TCP Reno's response to queue overflows in the face of competition is often quite messy, versus the single-loss behavior in the absence of competition as described above in *16.2.3 Single Losses*. If we are trying to come up with a packet-loss model to replace the synchronized-loss hypothesis, it turns out that we would do better to switch to SACK TCP, though use of SACK TCP will not

change the goodput ratios much at all. (The fact that Selective ACKs are nearly universal in the real world is another good reason to enable them here.)

Packets are dropped when the queue fills. It may be the case that only a single packet is dropped; it may be the case that multiple packets are dropped from each of multiple connections. We will refer to the set of packets dropped as a **drop cluster**. After a packet is dropped, TCP Reno discovers this just over one RTT after it was sent, through Fast Retransmit, and then responds by halving `cwnd`, through Fast Recovery.

TCP Reno retransmits only one lost packet per RTT; TCP NewReno does the same but SACK TCP may retransmit multiple lost packets together. If enough packets were lost from a TCP Reno/NewReno connection, not all of them may be retransmitted by the point the retransmission-timeout timer expires (typically 1-2 seconds), resulting in a coarse timeout. At that point, TCP abandons its Fast-Recovery process, even if it had been progressing steadily.

Eventually the TCP senders that have experienced packet loss reduce their `cwnd`, and thus the queue utilization drops. At that point we would classically not expect more losses until the senders involved have had time to grow their `cwnd`s through the additive-increase process to the point of again filling the queue. As we shall see in *16.4.1.3 Transient queue peaks*, however, this classical expectation is not entirely correct.

It is possible that the packet losses associated with one full-queue period are spread out over sufficient time (more than one RTT, at a minimum) that TCP Reno responds to them separately and halves `cwnd` more than once in rapid succession. We will refer to this as a **loss-response cluster**, or sometimes as a **tooth cluster**.

In the ns-2 simulator, counting individual lost packets and TCP loss responses is straightforward enough. For TCP Reno, there are only two kinds of loss responses: Fast Recovery, in which `cwnd` is halved, and coarse timeout, in which `cwnd` is set to 1.

Counting *clusters*, however, is more subjective; we need to decide when two drops or responses are close enough to one another that they should be counted as part of the same cluster. We use the notion of **granularity** here: two or more losses separated by less time than the granularity time interval are counted as a single event. We can also use granularity to decide when two loss responses in different connections are to be considered parts of the same event, or to tie loss responses to packet losses.

The appropriate length of the granularity interval is not as clear-cut as might be hoped. In some cases a couple RTTs may be sufficient, but note that the RTT_{noLoad} of the B–D connection above ranges from 0.22 sec to over 1.0 sec as `delayB` increases from 0 to 400 ms. In order to cover coarse timeouts, a granularity of from two to three seconds often seems to work well for packet drops.

If we are trying to count losses to estimate the loss rate as in the formula `cwnd` = $1.225/\sqrt{p}$ as in *14.5 TCP Reno loss rate versus cwnd*, then we should count every loss response separately; the argument in *14.5 TCP Reno loss rate versus cwnd* depended on counting *all* loss responses. The difference between one fast-recovery response and two in rapid succession is that in the latter case `cwnd` is halved twice, to about a quarter of its original value.

However, if we are interested in whether or not losses between two connections are synchronized, we need again to make use of granularity to make sure two "close" losses are counted as one. In this setting, a granularity of one to two seconds is often sufficient.

16.4.1 Some TCP Reno cwnd graphs

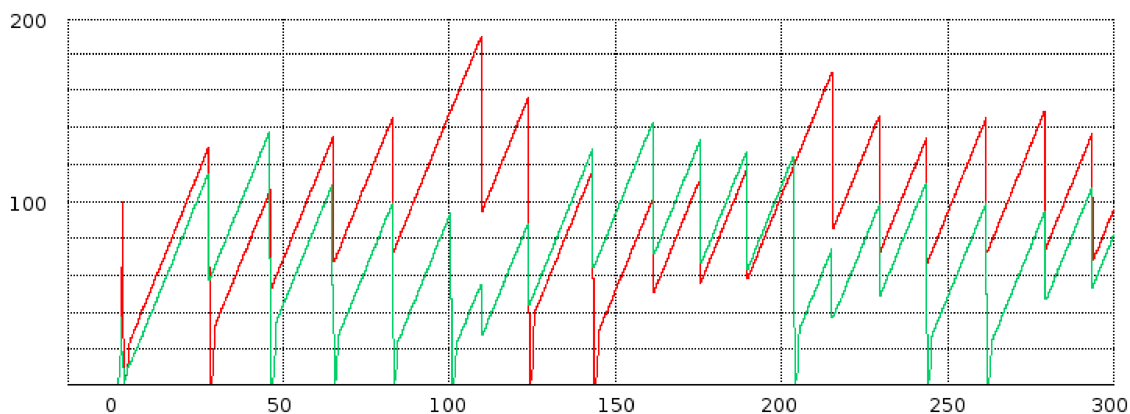
We next turn to some examples of actual TCP behavior, and present a few hopefully representative `cwnd` graphs. In each figure, the red line represents the **longer-path** (B–D) flow and the green line represents the **shorter** (A–D) flow. The graphs are of our usual simulation with `bottleneckBW` = 8.0 and `overhead` = 0.002, and run for 300 seconds.

> **ns-2 sensitivity**
>
> Some of the graphs here were prepared with an earlier version of the basic2.tcl simulation script above in which the nodes A and B were reversed, which means that packet-arrival ties at R may be resolved in a different order. That is enough sometimes to lead to noticeably different sets of packet losses.

While the immediate goal is to illuminate some of the above loss-clustering issues above, the graphs serve as well to illustrate the general behavior of TCP Reno competition and its variability. We will also use one of the graphs to explore the idea of **transient queue spikes**.

In each case we can count teeth visually or via a Python script; see *16.4.2.1 Counting teeth in Python*. In the latter case we must use an appropriate granularity interval (*eg* 2.0 seconds) if we want the count to agree even approximately with the visual count. Many times the tooth count is quite dependent on the exact value of the granularity.

16.4.1.1 delayB = 0

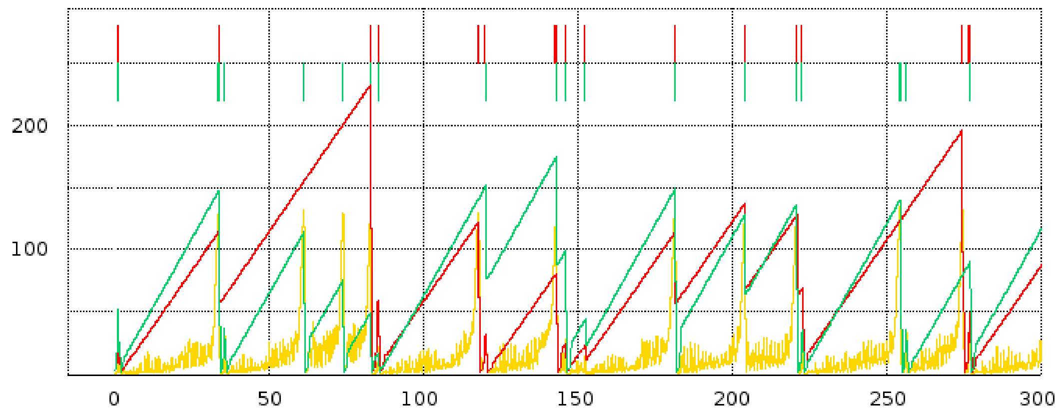


We start with the equal-RTTs graph, that is, `delayB` = 0. In this figure the teeth (loss events) are almost completely synchronized; the only unsynchronized loss events are the green flow's losses at T=100 and at about T=205. The two `cwnd` graphs, though, do not exactly move in lockstep. The red flow has three coarse timeouts (where `cwnd` drops to 0), at about T=30, T=125 and T=145; the green flow has seven coarse timeouts.

The red graph gets a little ahead of the green in the interval 50-100, despite synchronized losses there. Just before T=50 the green graph has a fast-recovery response followed by a coarse timeout; the next three green losses also involve coarse timeouts. Despite perfect loss synchronization in the range from T=40 to T=90,

the green graph ends up set back for a while because its three loss events all involve coarse timeouts while none of the red graph's do.

16.4.1.2 delayB = 25



In this delayB = 25 graph, respective packet losses for the red and green flows are marked along the top, and the cwnd graphs are superimposed over a graph of the averaged queue utilization in gold. The time scale for queue-utilization averaging is about one RTT here. The queue graph is scaled vertically (by a factor of 8) so the queue values (maximum 20) are numerically comparable to the cwnd values (the transit capacity is about 230).

There is one large red tooth from about T=40 to T=90 that corresponds to three green teeth; from about T=220 to T=275 there is one red tooth corresponding to two green teeth. Aside from these two points, representing three isolated green-only losses, the red and green teeth appear quite well synchronized.

We also see evidence of loss-response clusters. At around T=143 the large green tooth peaks; halfway down there is a little notch ending at T=145 that represents a fast-recovery loss event interpreted by TCP as distinct. There is actually a third event, as well, representing a coarse timeout shortly after T=145, and then a fourth fast-recovery event at about T=152 which we will examine shortly. At T≃80, the red tooth has a fast-recovery loss event followed very shortly by a coarse timeout; this happens for both flows at about T=220.

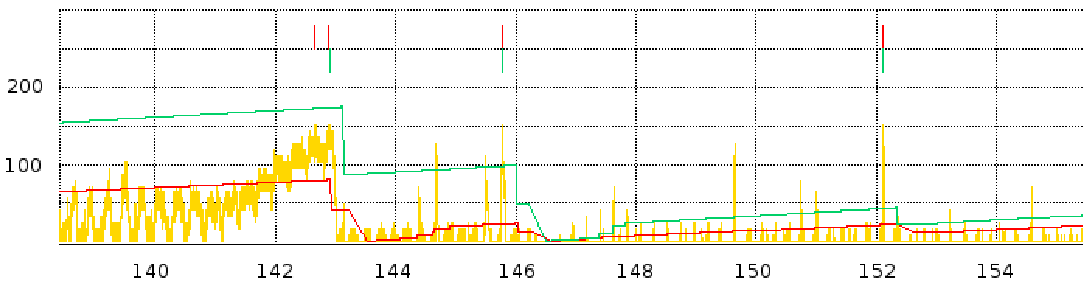
Overall, the red path has 11 teeth if we use a tooth-counting granularity of 3 seconds, and 8 teeth if the granularity is 5 seconds. We do not count anything that happens in the slow-start phase, generally before T=3.0, nor do we count the "tooth" at T=300 when the graph ends.

The slope of the green teeth is slightly greater than the slope of the red teeth, representing the longer RTT for the red connection.

As for the queue graph, there is perhaps more "noise" than expected, but generally the right edges of the teeth – the TCP loss responses – are very well aligned with peaks representing the queue filling up. Recall that the transit capacity for flow1 here is about 230 packets and the queue capacity is about 20; we therefore in general expect the sum of the two cwnds to range between 125 and 250, and that the queue should mostly remain empty until the sum reached 230. We can indeed confirm visually that in most cases the tooth peaks do indeed add up to about 250.

16.4.1.3 Transient queue peaks

The loss in the graph above at around T=152 is a little peculiar; this is fully 10 seconds after the primary tooth peak that preceded it. Let us zoom in on it a little more closely, this time *without* any averaging of the queue-utilization graph.



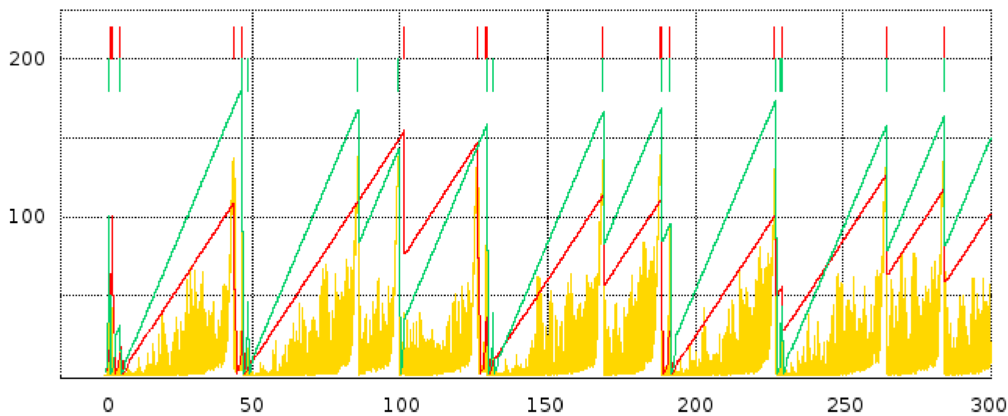
There are two very sharp, narrow peaks in the queue graph, just before T=146 and just after T=152, each causing packet drops for both flows. Neither peak, especially the latter one, appears to have much to do with the gradual queue-filling and loss event at T=143. Neither one of these **transient queue peaks** is associated with the sum of the `cwnds` approaching the maximum of about 250; at the T≃146 peak the sum of the `cwnds` is about 22+97 = 119 and at T≃152 the sum is about 21+42 = 63. ACKs for either sender cannot return from the destination D faster than the bottleneck-link rate of 1 packet/ms; this might suggest new packets from senders A and B cannot arrive at R faster than a combined rate of 1 packet/ms. *What is going on*?

What is happening is that each flow sends a flight of about 20 packets, spaced 1 ms apart, but by coincidence these two flights begin arriving at R at the same moment. The runup in the queue near T=152 occurs from T=152.100 to the first drop at T=152.121. During this 21 ms interval, a flight of 20 packets arrive from node A (flow 0), and a flight of 19 packets arrive from node B (flow 1). These 39 packets in 21 ms means the queue utilization at R must increase by 39-21 = 18, which is sufficient to overflow the queue as it was not quite empty beforehand. The ACK flights that triggered these data-packet flights were indeed spaced 1 ms apart, consistent with the bottleneck link, but the ACKs (and the data-packet flights that triggered those ACKs) passed through R at quite different times, because of the 25-ms difference in propagation delay on the A–R and B–R links.

Transient queue peaks like this complicate any theory of relative throughput based on gradually filling the queue. Fortunately, while transient peaks themselves are quite common (as can be seen from the zoomed graph above), only occasionally do they amount to enough to overflow the queue. And, in the examples created here, the majority of transient overflow peaks (including the one analyzed above) are within a few seconds of a TCP coarse timeout response and *may* have some relationship to that timeout.

The existence of transient queue peaks can be used to argue that queues should always have a reasonable "surge" capacity; or, more specifically, that bufferbloat should not be addressed simply by reducing the queue capacity to a very small level (*13.7.1 Bufferbloat*). However, transient peaks are relatively infrequent.

16.4.1.4 delayB = 61

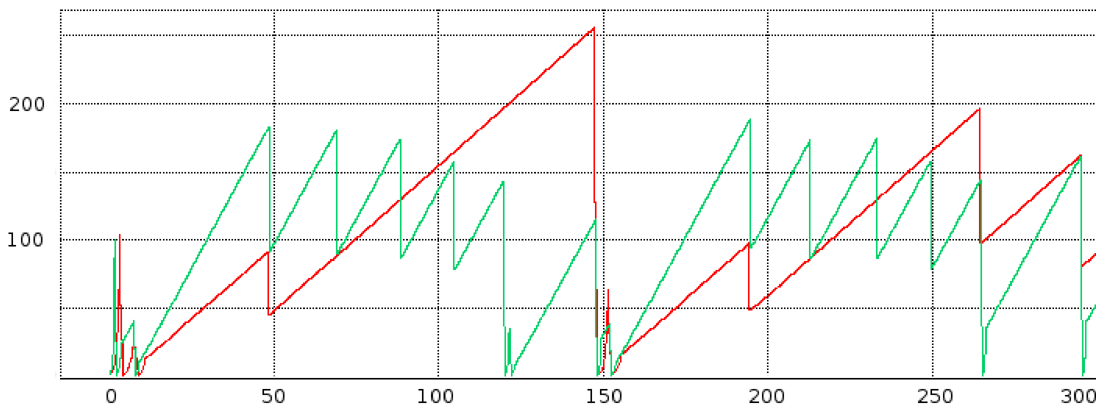


For this graph, note that the red and green loss events at T=130 are not quite aligned; the same happens at T=45 and T=100.

We also have several multiple-loss-response clusters, at T=40, T=130, T=190 and T=230.

The queue graph gives the appearance of much more solid yellow; this is due to a large number of transient queue spikes. Under greater magnification it becomes clear these spikes are still relatively sparse, however. There are transient queue overflows at T=46 and T=48, following a "normal" overflow at T=44, at T=101 following a normal overflow at T=99, at T=129 and T=131 following a normal overflow at T=126, and at T=191 following a normal overflow at T=188.

16.4.1.5 delayB = 120



Here we have (without the queue data) a good example of highly **un**synchronized teeth: the green graph has 12 teeth, after the start, while the red graph has five. But note that the red-graph loss events are a subset of the green loss events.

16.4.2 SACK TCP and Avoiding Loss Anomalies

Neither the coarse timeouts nor the "clustered" loss responses in the graphs above were anticipated in our original fairness model in *14.3 TCP Fairness with Synchronized Losses*. It is time to see if we can avoid these anomalies and thus obtain behavior closer to the classic sawtooth. It turns out that SACK TCP fills the bill quite nicely.

In the following subsection (*16.4.2.1 Counting teeth in Python*) is a script for counting loss responses ("teeth"). If we run it on the tracefiles from our TCP Reno simulations with `bottleneckBW` = 8.0 and time = 300, we find that, for each flow, about 30-35% of all loss responses are coarse timeouts. There is little if any dependence on the value for `delayB`.

If we change the two tcp connections in the simulation to `Agent/TCP/Newreno`, the coarse-timeout fraction falls by over half, to under 15%. This is presumably because TCP NewReno is better able to handle multiple packet drops.

However, when we change the TCP connections to use SACK TCP, the situation improves dramatically. We get essentially *no* coarse timeouts. Runs for 3000 seconds, typically involving 100-200 `cwnd`-halving adjustments per flow, almost never have more than one coarse timeout and the majority of the time have none.

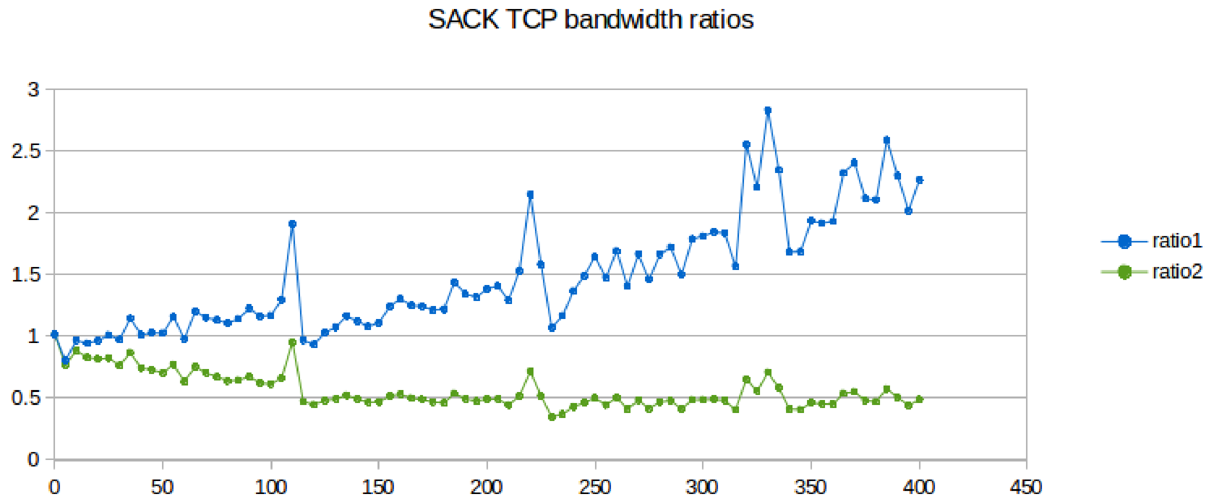
Clusters of multiple loss responses also vanish almost entirely. In these 3000-second runs, there are usually 1-2 cases where two loss responses occur within 1.0 seconds (over 4 RTTs for the A–D connection) of one another.

SACK TCP's smaller number of packet losses results in a marked improvement in goodput. Total link utilization ranges from 80.4% when `delayB` = 0 down to 68.3% when `delayB` = 400. For TCP Reno, the corresponding utilization range is from 77.5% down to 51.5%.

Note that switching to SACK TCP is unlikely to have a significant impact on the distribution of packet losses; SACK TCP differs from TCP Reno only in the way it *responds* to losses.

The SACK TCP sender is specified in ns-2 via `Agent/TCP/Sack1`. The receiver must also be SACK-aware; this is done by creating the receiver with `Agent/TCPSink/Sack1`.

When we switch to SACK TCP, the underlying fairness situation does not change much. Here is a graph similar to that above in *16.3.10 Raising the Bandwidth*. The run time is 3000 seconds, `bottleneckBW` is 8 Mbps, and `delayB` runs from 0 to 400 in increments of 5. (We did not use an increment of 1, as in the similar graph in *16.3.10 Raising the Bandwidth*, because we are by now confident that phase effects have been taken care of.)



Ratio1 is shown in blue and ratio2 in green. We again try to fit curves as in *16.3.10.1 Possible models* above. For the exponential model, goodput_ratio = K×(RTT_ratio)$^{\alpha}$, the value for the exponent α comes out this time to about 1.31, noticeably below TCP Reno's value of 1.57. There remains, however, considerable "noise" despite the less-frequent sampling interval, and it is not clear the difference is significant. The second model, goodput_ratio $\simeq$ 0.5 $\times$ (RTT_ratio)^2, still also appears to remain a good fit.

16.4.2.1 Counting teeth in Python

Using our earlier Python nstrace.py module, we can easily count the times `cwnd_` is reduced; these events correspond to loss responses. Anticipating more complex scenarios, we define a Python class `flowstats` to hold the per-flow information, though in this particular case we know there are exactly two flows.

We count *every* time `cwnd_` gets smaller; we also keep a count of coarse timeouts. As a practical matter, two closely spaced reductions in `cwnd_` are always due to a fast-recovery event followed about a second later by a coarse timeout. If we want a count of each separate TCP response, we count them both.

We do not count anything until after STARTPOINT. This is to avoid counting losses related to slow start.

```
#!/usr/bin/python3
import nstrace
import sys

# counts all points where cwnd_ drops.

STARTPOINT = 3.0        # wait for slow start to be done

class flowstats:                # python object to hold per-flow data
   def __init__(self):
       self.toothcount = 0
       self.prevcwnd = 0
       self.CTOcount = 0                # Coarse TimeOut count

def countpeaks(filename):
   global STARTPOINT
```

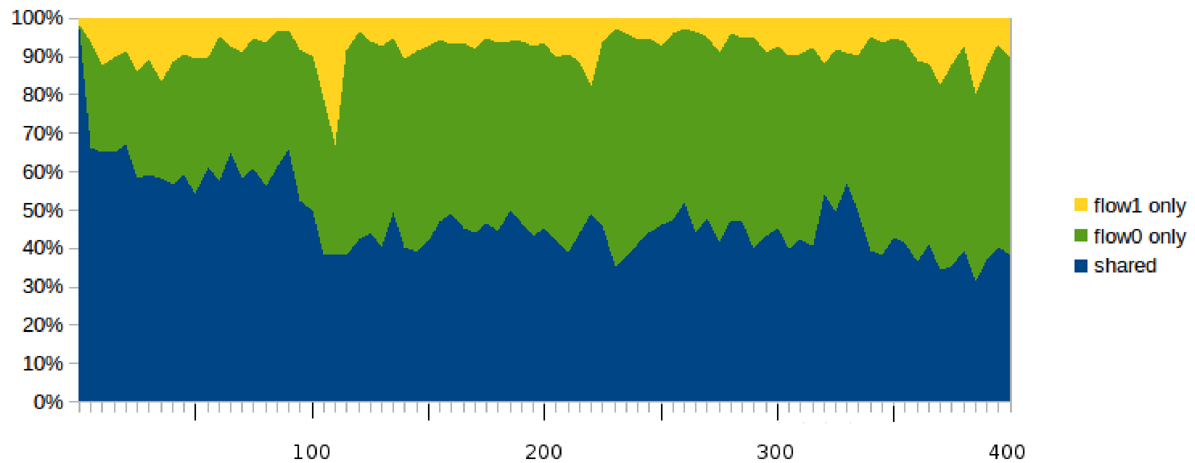
```
    nstrace.nsopen(filename)

    flow0 = flowstats()
    flow1 = flowstats()

    while not nstrace.isEOF():
        if nstrace.isVar():              # counting cwnd_ trace lines
            (time, snode, dummy, dummy, dummy, varname, cwnd) = nstrace.getVar()
            if (time < STARTPOINT):      continue
            if varname != "cwnd_":       continue
            if snode == 0: flow=flow0
            else: flow=flow1
            if cwnd < flow.prevcwnd:
                flow.toothcount += 1                   # count this as a tooth
                if cwnd == 1.0: flow.CTOcount += 1     # coarse timeout
            flow.prevcwnd=cwnd
        else:
            nstrace.skipline()

    print ("flow 0 teeth:", flow0.toothcount, "flow 1 teeth:", flow1.toothcount)
    print ("flow 0 coarse timeouts:", flow0.CTOcount, "flow 1 coarse timeouts:", flow1.CTOc

countpeaks(sys.argv[1])
```

A more elaborate version of this script, set up to count clusters of teeth and clusters of drops, is available in teeth.py. If a new cluster starts at time T, all teeth/drops by time T + granularity are part of the same cluster; the next tooth/drop after T + granularity starts the next new cluster.

16.4.2.2 Relative loss rates

At this point we accept that the A–D/B–D throughput ratio is generally smaller than the value predicted by the synchronized-loss hypothesis, and so the A–D flow must have additional losses to account for this. Our next experiment is to count the loss events in each flow, and to identify the loss events common to both flows.

The following graph demonstrates the rise in A–D losses as a percentage of the total, using SACK TCP. We use the tooth-counting script from the previous section, with a granularity of 1.0 sec. With SACK TCP it is rare for two drops in the same flow to occur close to one another; the granularity here is primarily to decide when two teeth in the two different flows are to be counted as shared.



The blue region at the bottom represents the percentage of all loss-response events (teeth) that are shared between both flows. The green region in the middle represents the percentage of all loss-response events that apply only to the faster A–D connection (flow 0); these rise to 30% very quickly and remain at about 50% as `delayB` ranges from just over 100 up to 400. The uppermost yellow region represents the loss events that affected only the slower B–D connection (flow 1); these are usually under 10%.

As a rough approximation, if we assume a 50/50 division between shared (blue) and flow-0-only (green) loss events, we have $\text{losscount}_0/\text{losscount}_1 = \gamma = 2$. Applying the formula of *14.5.2 Unsynchronized TCP Losses*, we get $\text{bandwidth}_0/\text{bandwidth}_1 = (\text{RTT_ratio})^2/2$, or ratio2 = 1/2. While it is not at all clear this will continue to hold as `delayB` continues to increase beyond 400, it does suggest a possible underlying explanation for the second formula of *16.3.10.1 Possible models*.

16.4.3 Loss rate versus `cwnd`: part 2

In *14.5 TCP Reno loss rate versus cwnd* we argued that the average value of `cwnd` was about $K/\sqrt{p}$, where the constant K was somewhere between 1.225 and 1.309. In *16.2.6.5 Loss rate versus cwnd: part 1* above we tested this for a single connection with very regular teeth; in this section we will test this hypothesis in the two-connections simulation. In order to avoid coarse timeouts, which were not included in the original model, we will use SACK TCP.

Over the lifetime of a connection, the average `cwnd` is the number of packets sent divided by the number of RTTs. This simple relationship is slightly complicated by the fact that the RTT varies with the fullness of the bottleneck queue, but, as before, this effect can be minimized if we choose models where $\text{RTT}_{\text{noLoad}}$ is large compared with the queuing delay. We will as before set `bottleneckBW` = 8.0; at this bandwidth queuing delays add less than 10% to $\text{RTT}_{\text{noLoad}}$. For the longer-path flow1, $\text{RTT}_{\text{noLoad}} \simeq 220 + 2\times$`delayB` ms. We will for both flows use the appropriate $\text{RTT}_{\text{noLoad}}$ as an approximation for RTT, and will use the number of packets acknowledged as an approximation for the number transmitted. These calculations give the true average `cwnd` values in the table below. The estimated average `cwnd` values are from the formula $K/\sqrt{p}$, where the loss ratio p is the total number of teeth, as counted earlier, divided by the number of that flow's packets.

With the relatively high value for `bottleneckBW` it takes a long simulation to get a reasonable number

of losses. The simulations used here ran 3000 seconds, long enough that each connection ended up with 100-200 losses.

Here is the data, for each flow, using K=1.225. The bold columns represent the extent by which the ratio of the estimated average `cwnd` to the true average `cwnd` differs from unity. These error values are reasonably close to zero, suggesting that the formula `cwnd` = $K/\sqrt{p}$ is reasonably accurate.

`delayB`	true avg cwnd0	est avg cwnd0	error0	true avg cwnd1	est avg cwnd1	error1
0	89.2	93.5	**4.8%**	87.7	92.2	**5.1%**
10	92.5	95.9	**3.6%**	95.6	99.2	**3.8%**
20	95.6	99.2	**3.7%**	98.9	101.9	**3.0%**
40	104.9	108.9	**3.8%**	103.3	105.6	**2.2%**
70	117.0	121.6	**3.9%**	101.5	102.8	**1.3%**
100	121.9	126.9	**4.1%**	104.1	104.1	**0%**
150	125.2	129.8	**3.7%**	112.7	109.7	**-2.6%**
200	133.0	137.5	**3.4%**	95.9	93.3	**-2.7%**
250	133.4	138.2	**3.6%**	81.0	78.6	**-3.0%**
300	134.6	138.8	**3.1%**	74.2	70.9	**-4.4%**
350	135.2	139.1	**2.9%**	69.8	68.4	**-2.0%**
400	137.2	140.6	**2.5%**	60.4	58.5	**-3.2%**

The table also clearly shows the rise in flow0's `cwnd`, cwnd0, and also the fall in cwnd1.

A related observation is that the *absolute* number of losses – for either flow – slowly declines as `delayB` increases, at least in the `delayB` $\leqslant$ 400 range we have been considering. For flow 1, with the longer RTT, this is because the number of packets transmitted drops precipitously (almost sevenfold) while `cwnd` – and therefore the loss-event probability p – stays relatively constant. For flow 0, the total number of packets sent rises as flow 1 drops to insignificance, but only by about 50%, and the average `cwnd0` (above) rises sufficiently fast (due to flow 1's decline) that

$$\text{total_losses} = \text{total_sent} \times p = \text{total_sent} \times K/\texttt{cwnd}^2$$

generally does not increase.

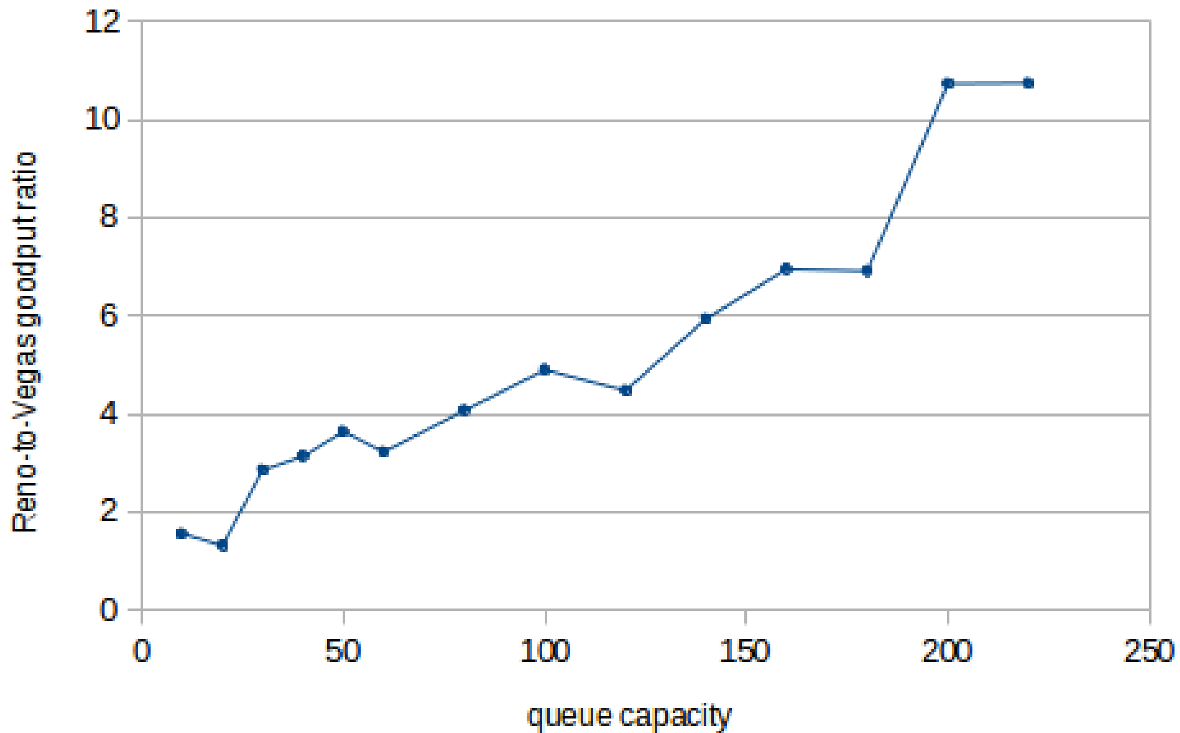
16.5 TCP Reno versus TCP Vegas

In *15.6 TCP Vegas* we described an alternative to TCP Reno known as TCP Vegas; in *15.6.1 TCP Vegas versus TCP Reno* we argued that when the capacity of a FIFO bottleneck queue was significant relative to the transit capacity, TCP Vegas would be at a disadvantage, but competition might be fairer when the queue capacity was small. Here we will test that theory.

In our standard model in *16.3 Two TCP Senders Competing* with `bottleneckBW` = 8.0 Mbps, A–R and

B–R delays of 10 ms and an R–D delay of 100 ms, the transit capacity for the A–D and B–D paths is 220 packets. We will simulate a competition between TCP Reno and TCP Vegas for queue sizes from 10 to 220; as earlier, we will use `overhead` = 0.002.

In our first attempt, with the default TCP Vegas parameters of α=1 and β=3, things start out rather evenly: when the queuesize is 10 the Reno/Vegas goodput ratio is 1.02. However, by the time the queue size is 220, the ratio has climbed almost to 22; TCP Vegas is swamped. Raising α and β helps a little for larger queues (perhaps at the expense of performance at small queue sizes); here is the graph for α=3 and β=6:



The vertical axis plots the Reno-to-Vegas goodput ratio; the horizontal axis represents the queue capacity. The performance of TCP Vegas falls off quite quickly as the queue size increases. One reason the larger α may help is that this slightly increases the range in which TCP Vegas behaves like TCP Reno. (For a rather different outcome using fair queuing rather than FIFO queuing, see *19.6.1 Fair Queuing and Bufferbloat*.)

To create an ns-2 TCP Vegas connection and set α and β one uses

```
set tcp1 [new Agent/TCP/Vegas]
$tcp1 set v_alpha_ 3
$tcp1 set v_beta_  6
```

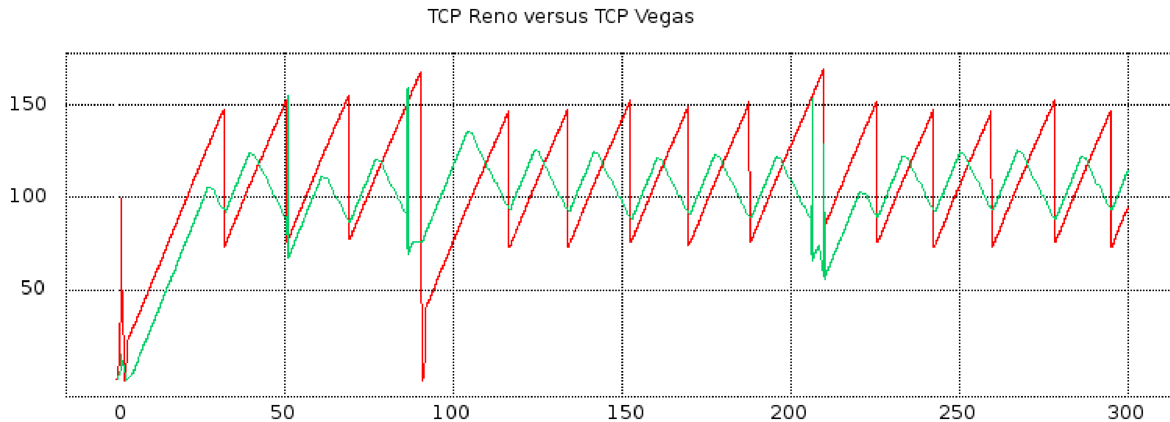
In prior simulations we have also made the following setting, in order to make the total TCP packetsize including headers be 1000:

```
Agent/TCP set packetSize_ 960
```

It turns out that for TCP Vegas objects in ns-2, the `packetSize_` *includes* the headers, as can be verified by looking at the tracefile, and so we need

```
Agent/TCP/Vegas set packetSize_ 1000
```

Here is a cwnd-v-time graph comparing TCP Reno and TCP Vegas; the queuesize is 20, bottleneckBW is 8 Mbps, overhead is 0.002, and α=3 and β=6. The first 300 seconds are shown. During this period the bandwidth ratio is about 1.1; it rises to close to 1.3 (all in TCP Reno's favor) when T=1000.



The red plot represents TCP Reno and the green represents TCP Vegas. The green plot shows some spikes that probably represent implementation artifacts.

Five to ten seconds before each sharp TCP Reno peak, TCP Vegas has its own softer peak. The RTT has begun to rise, and TCP Vegas recognizes this and begins decrementing cwnd by 1 each RTT. At the point of packet loss, TCP Vegas begins incrementing cwnd again. During the cwnd-decrement phase, TCP Vegas falls behind relative to TCP Reno, but it *may* catch up during the subsequent increment phase because TCP Vegas often avoids the cwnd = cwnd/2 multiplicative decrease and so often continues after a loss event with a larger cwnd than TCP Reno.

We conclude that, for smaller bottleneck-queue sizes, TCP Vegas does indeed hold its own. Unfortunately, in the scenario here the bottleneck-queue size has to be *quite* small for this to work; TCP Vegas suffers in competition with TCP Reno even for moderate queue sizes. That said, queue capacities out there in the real Internet tend to increase much more slowly than bandwidth, and there may be real-world situations were TCP Vegas performs quite well when compared to TCP Reno.

16.6 Wireless Simulation

When simulating wireless networks, there are no links; all the configuration work goes into setting up the nodes, the traffic and the wireless behavior itself. Wireless nodes have multiple wireless-specific attributes, such as the antenna type and radio-propagation model. Nodes are also now in charge of packet queuing; before this was the responsibility of the links. Finally, nodes have coordinates for position and, if **mobility** is introduced, velocities.

For wired links the user must set the bandwidth and delay. For wireless, these are both generally provided by the wireless model. Propagation delay is simply the distance divided by the speed of light. Bandwidth is usually built in to the particular wireless model chosen; for the Mac/802_11 model, it is available in attribute dataRate_ (which can be set). To find the current value, one can print [Mac/802_11 set dataRate_]; in ns-2 version 2.35 it is 1mb.

Ad hoc wireless networks must also be configured with a routing protocol, so that paths may be found from one node to another. We looked briefly at DSDV in *9.4.1 DSDV*; there are many others.

The maximum range of a node is determined by its power level; this can be set with `node-config` below (using the `txPower` attribute), but the default is often used. In the ns-2 source code, in file `wireless-phy.cc`, the variable `Pt_` – for transmitter power – is declared; the default value of 0.28183815 translates to a physical range of 250 meters using the appropriate radio-attenuation model.

We create a simulation here in which one node (`mover`) moves horizontally above a sequence of fixed-position nodes (stored in the Tcl array `rownodes`). The leftmost fixed-position node transmits continuously to the `mover` node; as the `mover` node progresses, packets must be routed through other fixed-position nodes. The fixed-position nodes here are 200 m apart, and the `mover` node is 150 m above their line; this means that the `mover` reaches the edge of the range of the ith rownode when it is directly above the i+1th rownode.

We use Ad hoc On-demand Distance Vector (AODV) as the routing protocol. When the `mover` moves out of range of one fixed-position node, AODV finds a new route (which will be via the next fixed-position node) quite quickly; we return to this below. DSDV (*9.4.1 DSDV*) is much slower, which leads to many packet losses until the new route is discovered. Of course, whether a given routing mechanism is fast enough depends very much on the speed of the `mover`; the simulation here does not perform nearly as well if the time is set to 10 seconds rather than 100 as the `mover` moves too fast even for AODV to keep up.

Because there are so many configuration parameters, to keep them together we adopt the common convention of making them all attributes of a single Tcl object, named `opt`.

We list the simulation file itself in pieces, with annotation; the complete file is at wireless.tcl. We begin with the options.

```
# ======================================================================
# Define options
# ======================================================================
set opt(chan)           Channel/WirelessChannel  ;# channel type
set opt(prop)           Propagation/TwoRayGround ;# radio-propagation model
set opt(netif)          Phy/WirelessPhy          ;# network interface type
set opt(mac)            Mac/802_11               ;# MAC type
set opt(ifq)            Queue/DropTail/PriQueue  ;# interface queue type
set opt(ll)             LL                       ;# link layer type
set opt(ant)            Antenna/OmniAntenna      ;# antenna model
set opt(ifqlen)         50                       ;# max packet in ifq

set opt(bottomrow)      5                        ;# number of bottom-row nodes
set opt(spacing)        200                      ;# spacing between bottom-row nodes
set opt(mheight)        150                      ;# height of moving node above bottom-row
set opt(brheight)       50                       ;# height of bottom-row nodes from bottom
set opt(x)              [expr ($opt(bottomrow)-1)*$opt(spacing)+1]    ;# x coordinate of to
set opt(y)              300                      ;# y coordinate of topology

set opt(adhocRouting)   AODV                     ;# routing protocol
set opt(finish)         100                      ;# time to stop simulation

# the next value is the speed in meters/sec to move across the field
set opt(speed)          [expr 1.0*$opt(x)/$opt(finish)]
```

The Channel/WirelessChannel class represents the physical terrestrial wireless medium; there is also a Channel/Sat class for satellite radio. The Propagation/TwoRayGround is a particular radio-propagation model. The TwoRayGround model takes into account ground reflection; for larger inter-node distances d, the received power level is proportional to $1/d^4$. Other models are the free-space model (in which received power at distance d is proportional to $1/d^2$) and the shadowing model, which takes into account other types of interference. Further details can be found in the Radio Propagation Models chapter of the ns-2 manual.

The Phy/WirelessPhy class specifies the standard wireless-node interface to the network; alternatives include Phy/WirelessPhyExt with additional options and a satellite-specific Phy/Sat. The Mac/802_11 class specifies IEEE 802.11 (that is, Wi-Fi) behavior; other options cover things like generic CSMA/CA, Aloha, and satellite. The Queue/DropTail/PriQueue class specifies the queuing behavior of each node; the opt(ifqlen) value determines the maximum queue length and so corresponds to the queue-limit value for wired links. The LL class, for Link Layer, defines things like the behavior of ARP on the network.

The Antenna/OmniAntenna class defines a standard omnidirectional antenna. There are many kinds of directional antennas in the real world – *eg* parabolic dishes and waveguide "cantennas" – and a few have been implemented as ns-2 add-ons.

The next values are specific to our particular layout. The opt(bottomrow) value determines the number of fixed-position nodes in the simulation. The spacing between adjacent bottom-row nodes is opt(spacing) meters. The moving node mover moves at height 150 meters above this fixed row. When mover is directly above a fixed node, it is thus at distance $\sqrt{(200^2 + 150^2)} = 250$ from the previous fixed node, at which point the previous node is out of range. The fixed row itself is 50 meters above the bottom of the topology. The opt(x) and opt(y) values are the dimensions of the simulation, in meters; the number of bottom-row nodes and their spacing determine opt(x).

As mentioned earlier, we use the AODV routing mechanism. When the mover node moves out of range of the bottom-row node that it is currently in contact with, AODV receives notice of the failed transmission from the Wi-Fi link layer (ultimately this news originates from the absence of the Wi-Fi link-layer ACK). This triggers an immediate search for a new route, which typically takes less than 50 ms to complete. The earlier DSDV (*9.4.1 DSDV*) mechanism does not use Wi-Fi link-layer feedback and so does not look for a new route until the next regularly scheduled round of distance-vector announcements, which might be several seconds away. Other routing mechanisms include TORA, PUMA, and OLSR.

The finishing time opt(finish) also represents the time the moving node takes to move across all the bottom-row nodes; the necessary speed is calculated in opt(speed). If the finishing time is reduced, the mover speed increases, and so the routing mechanism has less time to find updated routes.

The next section of Tcl code sets up general bookkeeping:

```
# create the simulator object
set ns [new Simulator]

# set up tracing
$ns use-newtrace
set tracefd  [open wireless.tr w]
set namtrace [open wireless.nam w]
$ns trace-all $tracefd
$ns namtrace-all-wireless $namtrace $opt(x) $opt(y)
```

```
# create  and define the topography object and layout
set topo [new Topography]

$topo load_flatgrid $opt(x) $opt(y)

# create an instance of General Operations Director, which keeps track of nodes and
# node-to-node reachability. The parameter is the total number of nodes in the simulation.

create-god [expr $opt(bottomrow) + 1]
```

The `use-newtrace` option enables a different tracing mechanism, in which each attribute except the first is prefixed by an identifying tag, so that parsing is no longer position-dependent. We look at an example below.

Note the special option `namtrace-all-wireless` for tracing for nam, and the dimension parameters `opt(x)` and `opt(y)`. The next step is to create a `Topography` object to hold the layout (still to be determined). Finally, we create a General Operations Director, which holds information about the layout not necessarily available to any node.

The next step is to call `node-config`, which passes many of the `opt()` parameters to ns and which influences future node creation:

```
# general node configuration
set chan1 [new $opt(chan)]

$ns node-config -adhocRouting $opt(adhocRouting) \
                 -llType $opt(ll) \
                 -macType $opt(mac) \
                 -ifqType $opt(ifq) \
                 -ifqLen $opt(ifqlen) \
                 -antType $opt(ant) \
                 -propType $opt(prop) \
                 -phyType $opt(netif) \
                 -channel $chan1 \
                 -topoInstance $topo \
                 -wiredRouting OFF \
                 -agentTrace ON \
                 -routerTrace ON \
                 -macTrace OFF
```

Finally we create our nodes. The bottom-row nodes are created within a Tcl `for`-loop, and are stored in a Tcl array `rownode()`. For each node we set its coordinates (`X_`, `Y_` and `Z_`); it is at this point that the rownode() nodes are given positions along the horizontal line y=50 and spaced `opt(spacing)` apart.

```
# create the bottom-row nodes as a node array $rownode(), and the moving node as $mover

for {set i 0} {$i < $opt(bottomrow)} {incr i} {
    set rownode($i) [$ns node]
    $rownode($i) set X_ [expr $i * $opt(spacing)]
    $rownode($i) set Y_ $opt(brheight)
    $rownode($i) set Z_ 0
}
```

```
set mover [$ns node]
$mover set X_ 0
$mover set Y_ [expr $opt(mheight) + $opt(brheight)]
$mover set Z_ 0
```

We now make the `mover` node move, using `setdest`. If the node reaches the destination supplied in `setdest`, it stops, but it is also possible to change its direction at later times using additional `setdest` calls, if a zig-zag path is desired. Various external utilities are available to create a file of Tcl commands to create a large number of nodes each with a designated motion; such a file can then be imported into the main Tcl file.

```
set moverdestX [expr $opt(x) - 1]

$ns at 0 "$mover setdest $moverdestX [$mover set Y_] $opt(speed)"
```

Next we create a UDP agent and a CBR (Constant Bit Rate) application, and set up a connection from `rownode(0)` to `mover`. CBR traffic does *not* use sliding windows.

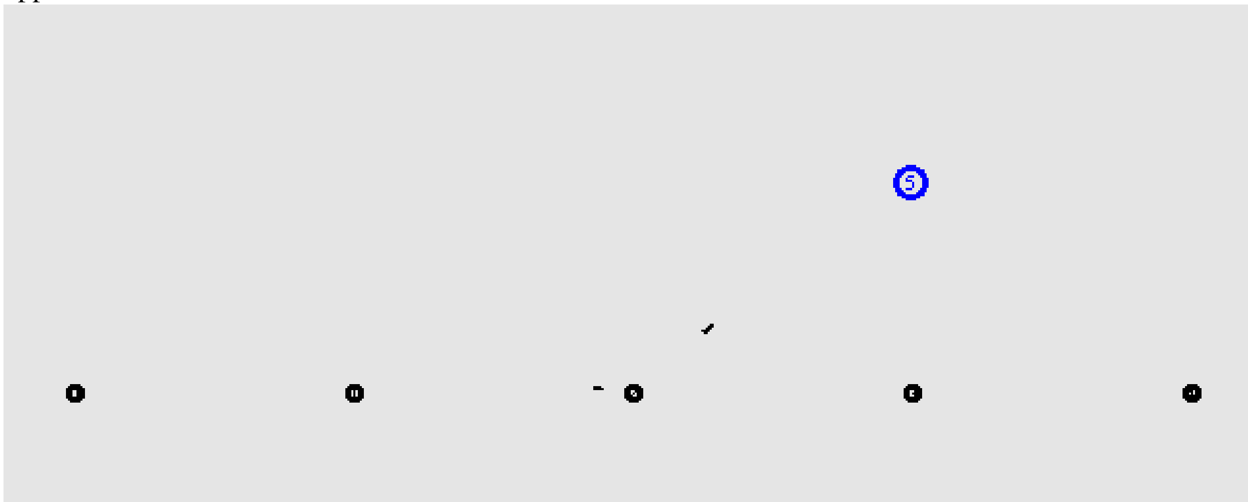
```
# setup UDP connection, using CBR traffic

set udp [new Agent/UDP]
set null [new Agent/Null]
$ns attach-agent $rownode(0) $udp
$ns attach-agent $mover $null
$ns connect $udp $null
set cbr1 [new Application/Traffic/CBR]
$cbr1 set packetSize_ 512
$cbr1 set rate_ 200Kb
$cbr1 attach-agent $udp
$ns at 0 "$cbr1 start"
$ns at $opt(finish) "$cbr1 stop"
```

The remainder of the Tcl file includes additional bookkeeping for nam, a `finish{}` procedure, and the startup of the simulation.

```
# tell nam the initial node position (taken from node attributes)
# and size (supplied as a parameter)

for {set i 0} {$i < $opt(bottomrow)} {incr i} {
    $ns initial_node_pos $rownode($i) 10
}

$ns initial_node_pos $mover 20

# set the color of the mover node in nam
$mover color blue
$ns at 0.0 "$mover color blue"

$ns at $opt(finish) "finish"

proc finish {} {
    global ns tracefd namtrace
    $ns flush-trace
```

```
    close $tracefd
    close $namtrace
    exit 0
}

# begin simulation

$ns run
```

The simulation can be viewed from the nam file, available at wireless.nam. In the simulation, the `mover` node moves across the topography, over the bottom-row nodes. The CBR traffic reaches `mover` from `rownode(0)` first directly, then via `rownode(1)`, then via `rownode(1)` and `rownode(2)`, *etc*. The motion of the `mover` node is best seen by speeding up the animation frame rate using the nam control for this, though doing this means that aliasing effects often make the CBR traffic appear to be moving in the opposite direction.



Above is one frame from the animation, with the `mover` node is almost (but not quite) directly over `rownode(3)`, and so is close to losing contact with `rownode(2)`. Two CBR packets can be seen *en route*; one has almost reached `rownode(2)` and one is about a third of the way from `rownode(2)` up to the blue `mover` node. The packets are not shown to scale; see exercise 17.

The tracefile is specific to wireless networking, and even without the use of `use-newtrace` has a rather different format from the link-based simulations earlier. The `newtrace` format begins with a letter for **s**end/**r**eceive/**d**rop/**f**orward; after that, each logged attribute is identified with a prefixed tag rather than by position. Full details can be found in the ns-2 manual. Here is an edited record of the first packet drop (the initial `d` indicates a drop-event record):

```
d -t 22.586212333 -Hs 0 -Hd 5 ... -Nl RTR -Nw CBK ... -Ii 1100 ... -Pn cbr -Pi 1100 ...
```

The `-t` tag indicates the time. The `-Hs` and `-Hd` tags indicate the source and destination, respectively. The `-Nl` tag indicates the "level" (RouTeR) at which the loss was logged, and the `-Nw` tag indicates the cause: `CBK`, for "CallBacK", means that the packet loss was detected at the link layer but the information was passed up to the routing layer. The `-Ii` tag is the packet's unique serial number, and the `P` tags supply information about the constant-bit-rate agent.

We can use the tracefile to find clusters of drops beginning at times 22.586, 47.575, 72.707 and 97.540,

corresponding roughly to the times when the route used to reach the `$mover` node shifts to passing through one more bottom-row node. Between t=72.707 and t=97.540 there are several other somewhat more mysterious clusters of drops; some of these clusters may be related to ordinary queue overflow but others may reflect decreasing reliability of the forwarding mechanism as the path grows longer.

16.7 Epilog

Simulations using ns (either ns-2 or ns-3) are a central part of networks research. Most scientific papers addressing comparisons between TCP flavors refer to ns simulations to at least some degree; ns is also widely used for non-TCP research (especially wireless).

But simulations are seldom a matter of a small number of runs. New protocols must be tested in a wide range of conditions, with varying bandwidths, delays and levels of background traffic. Head-to-head comparisons in isolation, such as our first runs in *16.3.3 Unequal Delays*, can be very misleading. Good simulation design, in other words, is not easy.

Our simulations here involved extremely simple networks. A great deal of effort has been expended by the ns community in the past decade to create simulations involving much larger sets of nodes; the ultimate goal is to create realistic simulations of the Internet itself. We refer the interested reader to *[FP01]*.

16.8 Exercises

1. In the graph in *16.2.1 Graph of cwnd v time*, examine the trace file to see what accounts for the dot at the start of each tooth (at times approximately 4.1, 6.1 and 8.0). Note that the solid parts of the graph are as solid as they are because fractional `cwnd` values are used; `cwnd` is incremented by 1/`cwnd` on receipt of each ACK.

2. A problem with the single-sender link-utilization experiment at *16.2.6 Single-sender Throughput Experiments* was that the smallest practical value for `queue-limit` was 3, which is 10% of the path transit capacity. Repeat the experiment but arrange for the path transit capacity to be at least 100 packets, making a `queue-limit` of 3 much smaller proportionally. Be sure the simulation runs long enough that it includes multiple teeth. What link utilization do you get? Also try `queue-limit` values of 4 and 5.

3. Create a single-sender simulation in which the path transit capacity is 90 packets, and the bottleneck `queue-limit` is 30. This should mean `cwnd` varies between 60 and 120. Be sure the simulation runs long enough that it includes many teeth.

   1. What link utilization do you observe?
   2. What queue utilization do you observe?

4. Use the basic2 model with equal propagation delays (`delayB` = 0), but delay the starting time for the first connection. Let this delay time be `startdelay0`; at the end of the basic2.tcl file, you will have

```
$ns at $startdelay0 "$ftp0 start"
```

Try this for `startdelay0` ranging from 0 to 40 ms, in increments of 1.0 ms. Graph the ratio1 or ratio2 values as a function of `startdelay0`. Do you get a graph like the one in *16.3.4.2 Two-sender phase effects*?

5. If the bottleneck link forwards at 10 ms/packet (`bottleneckBW` = 0.8), then in 300 seconds we can send 30,000 packets. What percentage of this, total, are sent by two competing senders as in *16.3 Two TCP Senders Competing*, for `delayB` = 0, 50, 100, 200 and 400.

6. Repeat the previous exercise for `bottleneckBW` = 8.0, that is, a bottleneck rate of 1 ms/packet.

7. Pick a case above where the total is less than 100%. Write a script that keeps track of $\texttt{cwnd}_0$ + $\texttt{cwnd}_1$, and measure how much time this quantity is less than the transit capacity. What is the average of $\texttt{cwnd}_0$ + $\texttt{cwnd}_1$ over those periods when it is less than the transit capacity?

8. In the model of *16.3 Two TCP Senders Competing*, it is often the case that for small `delayB`, *eg* `delayB` < 5, the longer-path B–D connection has *greater* throughput. Demonstrate this. Use an appropriate value of `overhead` (*eg* 0.02 or 0.002).

9. In generating the first graph in *16.3.7 Phase Effects and telnet traffic*, we used a `packetSize_` of 210 bytes, for an `actualSize` of 250 bytes. Generate a similar graph using in the simulations a much smaller value of `packetSize_`, *eg* 10 or 20 bytes. Note that for a given value of `tndensity`, as the `actualSize` shrinks so should the `tninterval`.

10. In *16.3.7 Phase Effects and telnet traffic* the telnet connections ran from A and B to D, so the telnet traffic competed with the bulk ftp traffic on the bottleneck link. Change the simulation so the telnet traffic runs only as far as R. The variable `tndensity` should now represent the fraction of the A–R and B–R bandwidth that is used for telnet. Try values for `tndensity` of from 5% to 50% (note these densities are quite high). Generate a graph like the first graph in *16.3.7 Phase Effects and telnet traffic*, illustrating whether this form of telnet traffic is effective at reducing phase effects.

11. Again using the telnet simulation of the previous exercise, in which the telnet traffic runs only as far as R, generate a graph comparing ratio1 for the bulk ftp traffic when randomization comes from:

- `overhead` values in the range discussed in the text (*eg* 0.01 or 0.02 for `bottleneckBW` = 0.8 Mbps)
- A–R and B–R telnet traffic with `tndensity` in the range 5% to 50%.

The goal should be a graph comparable to that of *16.3.8 overhead versus telnet*. Do the two randomization mechanisms – `overhead` and telnet – still yield comparable values for ratio1?

12. Repeat the same experiment as in exercise 8, but using telnet instead of `overhead`. Try it with A–D/B–D telnet traffic and `tndensity` around 1%, and also A–R/B–R traffic as in exercise 10 with a `tndensity` around 10%. The effect may be even more marked.

13. Analyze the packet drops for each flow for the Reno-versus-Vegas competition shown in the second graph (red v green) of *16.5 TCP Reno versus TCP Vegas*. In that simulation, `bottleneckBW` = 8.0 Mbps, `delayB` = 0, `queuesize` = 20, α=3 and β=6; the full simulation ran for 1000 seconds.

(a). How many drop clusters are for the Reno flow only?

(b). How many drop clusters are for the Vegas flow only?

(c). How many shared drop clusters are there?

Use a drop-cluster granularity of 2.0 seconds.

14. Generate a `cwnd`-versus-time graph for TCP Reno versus TCP Vegas, like the second graph in *16.5 TCP Reno versus TCP Vegas*, except using the default α=1. Does the TCP Vegas connection perform better or worse?

15. Compare two TCP Vegas connections as in *16.3 Two TCP Senders Competing*, for `delayB` varying from 0 to 400 (perhaps in steps of about 20). Use `bottleneckBW` = 8 Mbps, and run the simulations for at least 1000 seconds. Use the default α and β (usually α=1 and β=3). Is ratio1 $\simeq$ 1 or ratio2 $\simeq$ 1 a better fit? How does the overall fairness compare with what ratio1 $\simeq$ 1 or ratio2 $\simeq$ 1 would predict? Does either ratio appear roughly constant? If so, what is its value?

16. Repeat the previous exercise for a much larger value of α, say α=10. Set β=α+2, and be sure `queuesize` is larger than 2β (recall that, in ns, α is `v_alpha_` and β is `v_beta_`). If both connections manage to keep the same number of packets in the bottleneck queue at `R`, then both should get about the same goodput, by the queue-competition rule of *14.2.2 Example 2: router competition*. *Do they?* Is the fairness situation better or worse than with the default α and β?

17. In nam animations involving point-to-point links, packet lengths are displayed proportionally: if a link has a propagation delay of 10 ms and a bandwidth of 1 packet/ms, then each packet will be displayed with length 1/10 the link length. Is this true of wireless as well? Consider the animation (and single displayed frame) of *16.6 Wireless Simulation*. Assume the signal propagation speed is c $\simeq$ 300 m/µsec, that the nodes are 300 m apart, and that the bandwidth is 1 Mbps.

(a). How long is a single bit? (That is, how far does the signal travel in the time needed to send a single bit?)

(b). If a sender transmits continuously, how many bits will it send before its first bit reaches its destination?

(c). In the nam animation of *16.6 Wireless Simulation*, is it plausible that what is rendered for the CBR packets represents just the first bit of the packet? Is the scale about right for a single bit?

(d). What might be a more accurate animated representation of wireless packets?

In this chapter we take a somewhat cursory look at the ns-3 simulator, intended as a replacement for ns-2. The project is managed by the NS-3 Consortium, and all materials are available at nsnam.org.

Ns-3 represents a rather sharp break from ns-2. Gone is the Tcl programming interface; instead, ns-3 simulation programs are written in the C++ language, with extensive calls to the ns-3 library, although they are often still referred to as simulation "scripts". As the simulator core itself is also written in C++, this in some cases allows improved interaction between configuration and execution. However, configuration and execution are still in most cases quite separate: at the end of the simulation script comes a call `Simulator::Run()` – akin to ns-2's `$ns run` – at which point the user-written C++ has done its job and the library takes over.

To configure a simple simulation, an ns-2 Tcl script had to create nodes and links, create network-connection "agents" attached to nodes, and create traffic-generating applications attached to agents. Much the same applies to ns-3, but in addition each node must be configured with its network interfaces, and each network interface must be assigned an IP address.

17.1 Installing and Running ns-3

We here outline the steps for installing ns-3 under linux from the "allinone" `tar` file, assuming that all prerequisite packages (such as gcc) are already in place. Much more general installation instructions can be found at www.nsnam.org. In particular, serious users are likely to want to download the current Mercurial repository directly. Information is also available for Windows and Macintosh installation, although perhaps the simplest option for Windows users is to run ns-3 in a linux virtual machine.

The first step is to unzip the `tar` file; this should leave a directory named ns-allinone-3.nn, where nn reflects the version number (20 in the author's installation as of this 2014 writing). This directory is the root of the ns-3 system; it contains a `build.py` (python) script and the primary ns-3 directory ns-3.nn. All that is necessary is to run the build.py script:

```
./build.py
```

Considerable configuration and then compiler output should ensue, hopefully terminating with a list of "Modules built" and "Modules not built".

From this point on, most ns-3 work will take place in the subdirectory ns-3.nn, that is, in ns-allinone-3.nn/ns-3.nn. This *development* directory contains the source directory `src`, the script directory `scratch`, and the execution script waf.

The development directory also contains a directory `examples` containing a rich set of example scripts. The scripts in `examples/tutorial` are described in depth in the ns-3 tutorial in `doc/tutorial`.

17.1.1 Running a Script

Let us now run a script, for example, the file first.cc included in the `examples/tutorial` directory. We first copy this file into the directory "scratch", and then, in the parent development directory, enter the command

```
./waf --run first
```

The program is compiled and, if compilation is successful, is run.

In fact, *every* uncompiled program in the scratch directory is compiled, meaning that projects in progress that are not yet compilable must be kept elsewhere. One convenient strategy is to maintain multiple project directories, and link them symbolically to `scratch` as needed.

The ns-3 system includes support for command-line options; the following example illustrates the passing by command line of the value 3 for the variable `nCsma`:

```
./waf --run "second --nCsma=3"
```

17.1.2 Compilation Errors

By default, ns-3 enables the `-Werror` option to the compiler, meaning that all warnings are treated as errors. This is good practice for contributed or published scripts, but can be rather exasperating for beginners. To disable this, edit the file waf-tools/cflags.py (relative to the development directory). Change the line

```
self.warnings_flags = [['-Wall'], ['-Werror'], ['-Wextra']]
```

to

```
self.warnings_flags = [['-Wall'], ['-Wextra']]
```

Then, in the development directory, run

```
./waf configure
./waf build
```

17.2 A Single TCP Sender

We begin by translating the single-TCP-sender script of *16.2 A Single TCP Sender*. The full program is in basic1.cc; we now review most of it line-by-line; some standard things such as `#include` directives are omitted.

```
/*
  Network topology:

  A----R----B

  A--R: 10 Mbps / 10 ms delay
  R--B: 800Kbps / 50 ms delay
  queue at R: size 7
*/

using namespace ns3;

std::string fileNameRoot = "basic1";    // base name for trace files, etc

void CwndChange (Ptr<OutputStreamWrapper> stream, uint32_t oldCwnd, uint32_t newCwnd)
```

```
{
  *stream->GetStream () << Simulator::Now ().GetSeconds () << " " <<  newCwnd << std::endl;
}

static void
TraceCwnd ()    // Trace changes to the congestion window
{
  AsciiTraceHelper ascii;
  Ptr<OutputStreamWrapper> stream = ascii.CreateFileStream (fileNameRoot + ".cwnd");
  Config::ConnectWithoutContext ("/NodeList/0/$ns3::TcpL4Protocol/SocketList/0/CongestionW
}
```

The function `TraceCwnd()` arranges for tracing of `cwnd`; the function `CwndChange` is a *callback*, invoked by the ns-3 system whenever `cwnd` changes. Such callbacks are common in ns-3.

The parameter string beginning `/NodeList/0/...` is an example of the *configuration namespace*. Each ns-3 attribute can be accessed this way. See *17.2.2 The Ascii Tracefile* below.

```
int main (int argc, char *argv[])
{
  int tcpSegmentSize = 1000;
  Config::SetDefault ("ns3::TcpSocket::SegmentSize", UintegerValue (tcpSegmentSize));
  Config::SetDefault ("ns3::TcpSocket::DelAckCount", UintegerValue (0));
  Config::SetDefault ("ns3::TcpL4Protocol::SocketType", StringValue ("ns3::TcpReno"));
  Config::SetDefault ("ns3::RttEstimator::MinRTO", TimeValue (MilliSeconds (500)));
```

The use of `Config::SetDefault()` allows us to configure objects that will not exist until some later point, perhaps not until the ns-3 simulator is running. The first parameter is an **attribute string**, of the form `ns3::`*class*`::`*attribute*. A partial list of attributes is at https://www.nsnam.org/docs/release/3.19/doxygen/group___attribute_list.html. Attributes of a class can also be determined by a command such as the following:

```
./waf --run "basic1 --PrintAttributes=ns3::TcpSocket
```

The advantage of the `Config::SetDefault` mechanism is that often objects are created indirectly, perhaps by "helper" classes, and so direct setting of class properties can be problematic.

It is perfectly acceptable to issue some `Config::SetDefault` calls, then create some objects (perhaps implicitly), and then change the defaults (again with `Config::SetDefault`) for creation of additional objects.

We pick the TCP congesion-control algorithm by setting `ns3::TcpL4Protocol::SocketType`. Options are `TcpRfc793` (no congestion control), `TcpTahoe`, `TcpReno`, `TcpNewReno` and `TcpWestwood`. TCP Cubic and SACK TCP are not supported natively (though they are available if the Network Simulation Cradle is installed).

Setting the `DelAckCount` attribute to 0 disables delayed ACKs. Setting the `MinRTO` value to 500 ms avoids some unexpected hard timeouts. We will return to both of these below in *17.2.3 Unexpected Timeouts and Other Phenomena*.

Next comes our local variables and command-line-option processing. In ns-3 the latter is handled via the `CommandLine` object, which also recognized the `--PrintAttributes` option above. Using the `--PrintHelp` option gives a list of variables that can be set via command-line arguments.

```
unsigned int runtime = 20;   // seconds
int delayAR = 10;            // ms
int delayRB = 50;            // ms
double bottleneckBW= 0.8;    // Mbps
double fastBW = 10;          // Mbps
uint32_t queuesize = 7;
uint32_t maxBytes = 0;       // 0 means "unlimited"

CommandLine cmd;
// Here, we define command line options overriding some of the above.
cmd.AddValue ("runtime", "How long the applications should send data", runtime);
cmd.AddValue ("delayRB", "Delay on the R--B link, in ms", delayRB);
cmd.AddValue ("queuesize", "queue size at R", queuesize);
cmd.AddValue ("tcpSegmentSize", "TCP segment size", tcpSegmentSize);

cmd.Parse (argc, argv);

std::cout << "queuesize=" << queuesize << ", delayRB=" << delayRB << std::endl;
```

Next we create three nodes, illustrating the use of smart pointers and `CreateObject()`.

```
Ptr<Node> A = CreateObject<Node> ();
Ptr<Node> R = CreateObject<Node> ();
Ptr<Node> B = CreateObject<Node> ();
```

Class `Ptr` is a "smart pointer" that manages memory through reference counting. The template function `CreateObject` acts as the ns-3 preferred alternative to `operator new`. Parameters for objects created this way can be supplied via `Config::SetDefault`, or by some later method call applied to the `Ptr` object. For `Node` objects, for example, we might call `A -> AddDevice(...)`.

A convenient alternative to creating nodes individually is to create a *container* of nodes all at once:

```
NodeContainer allNodes;
allNodes.Create(3);
Ptr<Node> A = allNodes.Get(0);
...
```

After the nodes are in place we create our point-to-point links, using the `PointToPointHelper` class. We also create `NetDeviceContainer` objects; we don't use these here (we could simply call `AR.Install(A,R)`), but will need them below when assigning IPv4 addresses.

```
// use PointToPointChannel and PointToPointNetDevice
NetDeviceContainer devAR, devRB;
PointToPointHelper AR, RB;

// create point-to-point link from A to R
AR.SetDeviceAttribute ("DataRate", DataRateValue (DataRate (fastBW * 1000 * 1000)));
AR.SetChannelAttribute ("Delay", TimeValue (MilliSeconds (delayAR)));
devAR = AR.Install(A, R);

// create point-to-point link from R to B
RB.SetDeviceAttribute ("DataRate", DataRateValue (DataRate (bottleneckBW * 1000 * 1000)));
RB.SetChannelAttribute ("Delay", TimeValue (MilliSeconds (delayRB)));
```

```
RB.SetQueue("ns3::DropTailQueue", "MaxPackets", UintegerValue(queuesize));
devRB = RB.Install(R,B);
```

Next we hand out IPv4 addresses. The `Ipv4AddressHelper` class can help us with individual LANs (*eg* A–R and R–B), but it is up to us to make sure our two LANs are on different subnets. If we attempt to put A and B on the same subnet, routing will simply fail, just as it would if we were to do this with real network nodes.

```
InternetStackHelper internet;
internet.Install (A);
internet.Install (R);
internet.Install (B);

// Assign IP addresses

Ipv4AddressHelper ipv4;
ipv4.SetBase ("10.0.0.0", "255.255.255.0");
Ipv4InterfaceContainer ipv4Interfaces;
ipv4Interfaces.Add (ipv4.Assign (devAR));
ipv4.SetBase ("10.0.1.0", "255.255.255.0");
ipv4Interfaces.Add (ipv4.Assign(devRB));

Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
```

Next we print out the addresses assigned. This gives us a peek at the `GetObject` template and the ns-3 object-aggregation model. The original `Node` objects we created earlier were quite generic; they gained their `Ipv4` component in the code above. Now we retrieve that component with the `GetObject<Ipv4>()` calls below.

```
Ptr<Ipv4> A4 = A->GetObject<Ipv4>();  // gets node A's IPv4 subsystem
Ptr<Ipv4> B4 = B->GetObject<Ipv4>();
Ptr<Ipv4> R4 = R->GetObject<Ipv4>();
Ipv4Address Aaddr = A4->GetAddress(1,0).GetLocal();
Ipv4Address Baddr = B4->GetAddress(1,0).GetLocal();
Ipv4Address Raddr = R4->GetAddress(1,0).GetLocal();

std::cout << "A's address: " << Aaddr << std::endl;
std::cout << "B's address: " << Baddr << std::endl;
std::cout << "R's #1 address: " << Raddr << std::endl;
std::cout << "R's #2 address: " << R4->GetAddress(2,0).GetLocal() << std::endl;
```

In general, `A->GetObject<T>` returns the component of type `T` that has been "aggregated" to `Ptr<Object> A`; often this aggregation is invisible to the script programmer but an understanding of how it works is sometimes useful. The aggregation is handled by the ns-3 `Object` class, which contains an internal list `m_aggregates` of aggregated companion objects. At most one object of a given type can be aggregated to another, making `GetObject<T>` unambiguous. Given a `Ptr<Object> A`, we can obtain an iterator over the aggregated companions via `A->GetAggregateIterator()`, of type `Object::AggregateIterator`. From each `Ptr<const Object> B` returned by this iterator, we can call `B->GetInstanceTypeId().GetName()` to get the class name of B.

The `GetAddress()` calls take two parameters; the first specfies the interface (a value of 0 gives the loopback interface) and the second distinguishes between multiple addresses assigned to

the same interface (which is not happening here). The call `A4->GetAddress(1,0)` returns an `Ipv4InterfaceAddress` object containing, among other things, an IP address, a broadcast address and a netmask; `GetLocal()` returns the first of these.

Next we create the receiver on B, using a `PacketSinkHelper`. A receiver is, in essense, a read-only form of an application server.

```
// create a sink on B
uint16_t Bport = 80;
Address sinkAaddr(InetSocketAddress (Ipv4Address::GetAny (), Bport));
PacketSinkHelper sinkA ("ns3::TcpSocketFactory", sinkAaddr);
ApplicationContainer sinkAppA = sinkA.Install (B);
sinkAppA.Start (Seconds (0.01));
// the following means the receiver will run 1 min longer than the sender app.
sinkAppA.Stop (Seconds (runtime + 60.0));

Address sinkAddr(InetSocketAddress(Baddr, Bport));
```

Now comes the sending application, on A. We must configure and create a `BulkSendApplication`, attach it to A, and arrange for a connection to be created to B. The `BulkSendHelper` class simplifies this.

```
BulkSendHelper sourceAhelper ("ns3::TcpSocketFactory",  sinkAddr);
sourceAhelper.SetAttribute ("MaxBytes", UintegerValue (maxBytes));
sourceAhelper.SetAttribute ("SendSize", UintegerValue (tcpSegmentSize));
ApplicationContainer sourceAppsA = sourceAhelper.Install (A);
sourceAppsA.Start (Seconds (0.0));
sourceAppsA.Stop (Seconds (runtime));
```

If we did not want to use the helper class here, the easiest way to create the `BulkSendApplication` is with an `ObjectFactory`. We configure the factory with the type we want to create and the relevant configuration parameters, and then call `factory.Create()`. (We could have used the `Config::SetDefault()` mechanism and `CreateObject()` as well.)

```
ObjectFactory factory;
factory.SetTypeId ("ns3::BulkSendApplication");
factory.Set ("Protocol", StringValue ("ns3::TcpSocketFactory"));
factory.Set ("MaxBytes", UintegerValue (maxBytes));
factory.Set ("SendSize", UintegerValue (tcpSegmentSize));
factory.Set ("Remote", AddressValue (sinkAddr));
Ptr<Object> bulkSendAppObj = factory.Create();
Ptr<Application> bulkSendApp = bulkSendAppObj -> GetObject<Application>();
bulkSendApp->SetStartTime(Seconds(0.0));
bulkSendApp->SetStopTime(Seconds(runtime));
A->AddApplication(bulkSendApp);
```

The above gives us no direct access to the actual TCP connection. Yet another alternative is to start by creating the TCP socket and connecting it:

```
Ptr<Socket> tcpsock = Socket::CreateSocket (A, TcpSocketFactory::GetTypeId ());
tcpsock->Bind();
tcpsock->Connect(sinkAddr);
```

However, there is then no mechanism for creating a `BulkSendApplication` that uses a pre-existing socket. (For a workaround, see the tutorial example `fifth.cc`.)

Before beginning execution, we set up tracing; we will look at the tracefile format later. We use the `AR PointToPointHelper` class here, but both ascii and pcap tracing apply to the entire A–R–B network.

```
// Set up tracing
AsciiTraceHelper ascii;
std::string tfname = fileNameRoot + ".tr";
AR.EnableAsciiAll (ascii.CreateFileStream (tfname));
// Setup tracing for cwnd
Simulator::Schedule(Seconds(0.01),&TraceCwnd);       // this Time cannot be 0.0

// This tells ns-3 to generate pcap traces, including "-node#-dev#-" in filename
AR.EnablePcapAll (fileNameRoot);    // ".pcap" suffix is added automatically
```

This last creates four `.pcap` files, *eg*

basic1-0-0.pcap
basic1-1-0.pcap
basic1-1-1.pcap
basic1-2-0.pcap

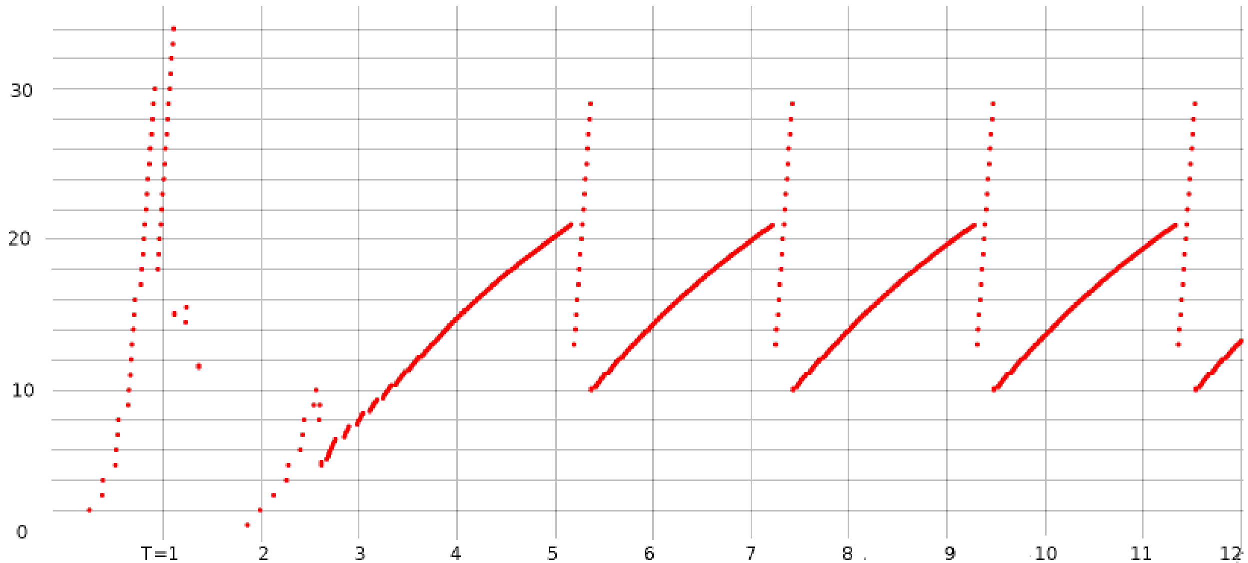
The first number refers to the node (A=0, R=1, B=2) and the second to the interface. A packet arriving at R but dropped there will appear in the second .pcap file but not the third. These files can be viewed with WireShark.

Finally we are ready to start the simulator! The `BulkSendApplication` will stop at time `runtime`, but traffic may be in progress. We allow it an additional 60 seconds to clear. We also, after the simulation has run, print out the number of bytes received by B.

```
  Simulator::Stop (Seconds (runtime+60));
  Simulator::Run ();

  Ptr<PacketSink> sink1 = DynamicCast<PacketSink> (sinkAppA.Get (0));
  std::cout << "Total Bytes Received from A: " << sink1->GetTotalRx () << std::endl;
  return 0;
}
```

17.2.1 Running the Script

When we run the script and plot the `cwnd` trace data (here for about 12 seconds), we get the following:



Compare this graph to that in *16.2.1 Graph of cwnd v time* produced by ns-2. The slow-start phase earlier ended at around 2.0 and now ends closer to 3.0. There are several modest differences, including the halving of `cwnd` just before T=1 and the peak around T=2.6; these were not apparent in the ns-2 graph.

After slow-start is over, the graphs are quite similar; `cwnd` ranges from 10 to 21. The period before was 1.946 seconds; here it is 2.0548; the difference is likely due to a more accurate implementation of the recovery algorithm.

One striking difference is the presence of the near-vertical line of dots just after each peak. What is happening here is that ns-3 implements the `cwnd` inflation/deflation algorithm outlined at the tail end of *13.4 TCP Reno and Fast Recovery*. When three dupACKs are received, `cwnd` is set to `cwnd`/2 + 3, and is then allowed to increase to 1.5×`cwnd`. See the end of *13.4 TCP Reno and Fast Recovery*.

17.2.2 The Ascii Tracefile

Below are four lines from the tracefile, starting with the record showing packet 271 (Seq=271001) being dropped by R.

```
d 4.9823  /NodeList/1/DeviceList/1/$ns3::PointToPointNetDevice/TxQueue/Drop ns3::PppHeader
r 4.98312 /NodeList/2/DeviceList/0/$ns3::PointToPointNetDevice/MacRx ns3::Ipv4Header (tos 0
+ 4.98312 /NodeList/2/DeviceList/0/$ns3::PointToPointNetDevice/TxQueue/Enqueue ns3::PppHead
- 4.98312 /NodeList/2/DeviceList/0/$ns3::PointToPointNetDevice/TxQueue/Dequeue ns3::PppHead
```

As with ns-2, the first letter indicates the action: **r** for received, **d** for dropped, **+** for enqueued, **-** for dequeued. For Wi-Fi tracefiles, **t** is for transmitted. The second field represents the time.

The third field represents the name of the event in the *configuration namespace*, sometimes called the *configuration path* name. The `NodeList` value represents the node (A=0, *etc*), the `DeviceList` represents the interface, and the final part of the name repeats the action: `Drop`, `MacRx`, `Enqueue`, `Dequeue`.

After that come a series of class names (*eg* `ns3::Ipv4Header`, `ns3::TcpHeader`), from the ns-3 attribute system, followed in each case by a parenthesized list of class-specific trace information.

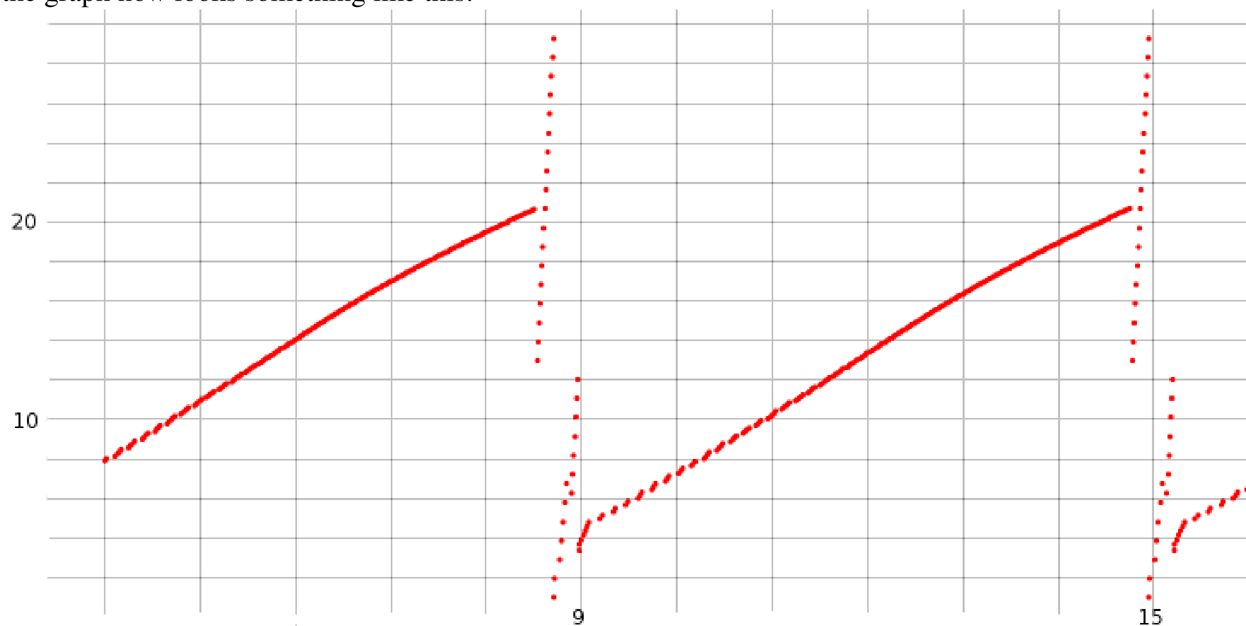
In the output above, the final three records all refer to node B (`/NodeList/2/`). Packet 258 has just arrived (Seq=258001), and ACK 259001 is then enqueued and sent.

17.2.3 Unexpected Timeouts and Other Phenomena

In the discussion of the script above at *17.2 A Single TCP Sender* we mentioned that we set `ns3::TcpSocket::DelAckCount` to 0, to disable delayed ACKs, and `ns3::RttEstimator::MinRTO` to 500 ms, to avoid unexpected timeouts.

If we comment out the line disabling delayed ACKs, little changes in our graph, except that the spacing between consecutive TCP teeth now almost doubles to 3.776. This is because with delayed ACKs the receiver sends only half as many ACKs, and the sender does not take this into account when incrementing `cwnd` (that is, the sender does not implement the suggestion of **RFC 3465** mentioned in *13.2.1 Per-ACK Responses*).

If we leave out the `MinRTO` adjustment, *and* set `tcpSegmentSize` to 960, we get a more serious problem: the graph now looks something like this:



We can enable ns-3's internal logging in the `TcpReno` class by entering the commands below, before running the script. (In some cases, as with `WifiHelper::EnableLogComponents()`, logging output can be enabled from within the script.) Once enabled, logging output is written to `stderr`.

```
NS_LOG=TcpReno=level_info
export NS_LOG
```

The log output shows the initial dupACK at 8.54:

```
8.54069 [node 0] Triple dupack. Reset cwnd to 12960, ssthresh to 10080
```

But then, despite Fast Recovery proceding normally, we get a hard timeout:

```
8.71463 [node 0] RTO. Reset cwnd to 960, ssthresh to 14400, restart from seqnum 510721
```

What is happening here is that the RTO interval was just a little too short, probably due to the use of the "awkward" segment size of 960.

After the timeout, there is another triple-dupACK!

```
8.90344 [node 0] Triple dupack. Reset cwnd to 6240, ssthresh to 3360
```

Shortly thereafter, at T=8.98, `cwnd` is reset to 3360, in accordance with the Fast Recovery rules.

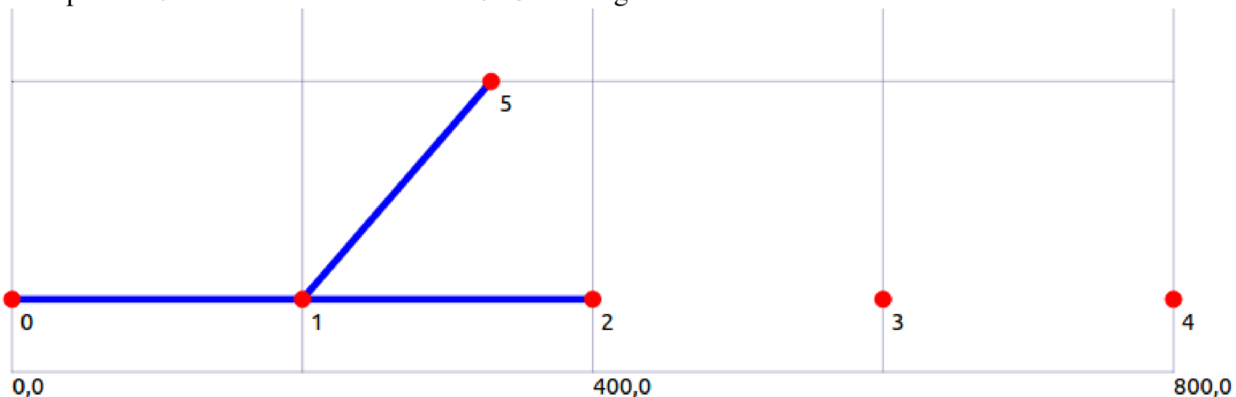
The overall effect is that `cwnd` is reset, not to 10, but to about 3.4 (in packets). This significantly slows down throughput.

In recovering from the hard timeout, the sequence number is reset to Seq=510721 (packet 532), as this was the last packet acknowledged. Unfortunately, several later packets had in fact made it through to B. By looking at the tracefile, we can see that at T=8.7818, B received Seq=538561, or packet 561. Thus, when A begins retransmitting packets 533, 534, *etc* after the timeout, B's response is to send the ACK the highest packet it has received, packet 561 (Ack=539521).

This scenario is not what the designers of Fast Recovery had in mind; it is likely triggered by a too-conservative timeout estimate. Still, exactly how to fix it is an interesting question; one approach might be to ignore, in Fast Recovery, triple dupACKs of packets now beyond what the sender is currently sending.

17.3 Wireless

We next present the wireless simulation of *16.6 Wireless Simulation*. The full script is at wireless.cc; the animation output for the `netanim` player is at wireless.xml. As before, we have one `mover` node moving horizontally 150 meters above a row of five fixed nodes spaced 200 meters apart. The limit of transmission is set to be 250 meters, meaning that a fixed node goes out of range of the `mover` node just as the latter passes over directly above the *next* fixed node. As before, we use Ad hoc On-demand Distance Vector (AODV) as the routing protocol. When the `mover` passes over fixed node N, it goes out of range of fixed node N-1, at which point AODV finds a new route to `mover` through fixed node N.



As in ns-2, wireless simulations tend to require considerably more configuration than point-to-point simulations. We now review the source code line-by-line. We start with two callback functions and the global variables they will need to access.

```
using namespace ns3;

Ptr<ConstantVelocityMobilityModel> cvmm;
double position_interval = 1.0;
std::string tracebase = "scratch/wireless";

// two callbacks
void printPosition()
{
  Vector thePos = cvmm->GetPosition();
  Simulator::Schedule(Seconds(position_interval), &printPosition);
  std::cout << "position: " << thePos << std::endl;
}

void stopMover()
{
  cvmm -> SetVelocity(Vector(0,0,0));
}
```

Next comes the data rate:

```
int main (int argc, char *argv[])
{
  std::string phyMode = "DsssRate1Mbps";
```

The `phyMode` string represents the Wi-Fi data rate (and modulation technique). DSSS rates are `DsssRate1Mbps`, `DsssRate2Mbps`, `DsssRate5_5Mbps` and `DsssRate11Mbps`. Also available are `ErpOfdmRate` constants to 54 Mbps and `OfdmRate` constants to 150 Mbps with a 40 MHz bandwidth (`GetOfdmRate150MbpsBW40MHz`). All these are defined in src/wifi/model/wifi-phy.cc.

Next are the variables that determine the layout and network behavior. The `factor` variable allows slowing down the speed of the `mover` node but correspondingly extending the runtime (though the new-route-discovery time is *not* scaled):

```
int bottomrow = 5;              // number of bottom-row nodes
int spacing = 200;              // between bottom-row nodes
int mheight = 150;              // height of mover above bottom row
int brheight = 50;              // height of bottom row

int X = (bottomrow-1)*spacing+1;              // X is the horizontal dimension of the field
int packetsize = 500;
double factor = 1.0;  // allows slowing down rate and extending runtime; same total # of pa
int endtime = (int)100*factor;
double speed = (X-1.0)/endtime;
double bitrate = 80*1000.0/factor;  // *average* transmission rate, in bits/sec
uint32_t interval = 1000*packetsize*8/bitrate*1000;    // in microsec
uint32_t packetcount = 1000000*endtime/ interval;
std::cout << "interval = " << interval <<", rate=" << bitrate << ", packetcount=" << packet
```

There are some niceties in calculating the packet transmission interval above; if we do it instead as 1000000*packetsize*8/bitrate then we sometimes run into 32-bit overflow problems or integer-division-roundoff problems.

Now we configure some Wi-Fi settings.

```
// disable fragmentation for frames below 2200 bytes
Config::SetDefault ("ns3::WifiRemoteStationManager::FragmentationThreshold", StringValue ("
// turn off RTS/CTS for frames below 2200 bytes
Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold", StringValue ("2200"))
// Set non-unicast data rate to be the same as that of unicast
Config::SetDefault ("ns3::WifiRemoteStationManager::NonUnicastMode", StringValue (phyMode))
```

Here we create the `mover` node with `CreateObject<Node>()`, but the fixed nodes are created via a `NodeContainer`, as is more typical with larger simulations

```
// Create nodes
NodeContainer fixedpos;
fixedpos.Create(bottomrow);
Ptr<Node> lowerleft = fixedpos.Get(0);
Ptr<Node> mover = CreateObject<Node>();
```

Now we put together a set of "helper" objects for more Wi-Fi configuration. We must configure both the PHY (physical) and MAC layers.

```
// The below set of helpers will help us to put together the desired Wi-Fi behavior
WifiHelper wifi;
wifi.SetStandard (WIFI_PHY_STANDARD_80211b);
wifi.SetRemoteStationManager ("ns3::AarfWifiManager"); // Use AARF rate control
```

The AARF rate changes can be viewed by enabling the appropriate logging with, at the shell level before `./waf`, `NS_LOG=AarfWifiManager=level_debug`. We are not otherwise interested in rate scaling (*3.7.2 Dynamic Rate Scaling*) here, though.

The PHY layer helper is `YansWifiPhyHelper`. The YANS project (Yet Another Network Simulator) was an influential precursor to ns-3; see *[LH06]*. Note the `AddPropagationLoss` configuration, where we set the Wi-Fi range to 250 meters. The MAC layer helper is `NqosWifiMacHelper`; the "nqos" means "no quality-of-service", *ie* no use of Wi-Fi PCF (*3.7.7 Wi-Fi Polling Mode*).

```
// The PHY layer here is "yans"
YansWifiPhyHelper wifiPhyHelper =  YansWifiPhyHelper::Default ();
// for .pcap tracing
// wifiPhyHelper.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);

YansWifiChannelHelper wifiChannelHelper;              // *not* ::Default() !
wifiChannelHelper.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel"); // pld:
// the following has an absolute cutoff at distance > 250
wifiChannelHelper.AddPropagationLoss ("ns3::RangePropagationLossModel", "MaxRange", DoubleV
Ptr<YansWifiChannel> pchan = wifiChannelHelper.Create ();
wifiPhyHelper.SetChannel (pchan);

// Add a non-QoS upper-MAC layer "AdhocWifiMac", and set rate control
NqosWifiMacHelper wifiMacHelper = NqosWifiMacHelper::Default ();
wifiMacHelper.SetType ("ns3::AdhocWifiMac");
NetDeviceContainer devices = wifi.Install (wifiPhyHelper, wifiMacHelper, fixedpos);
devices.Add (wifi.Install (wifiPhyHelper, wifiMacHelper, mover));
```

At this point the basic Wi-Fi configuration is done! The next step is to work on the positions and motion. First we establish the positions of the fixed nodes.

```
MobilityHelper sessile;              // for fixed nodes
Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
int Xpos = 0;
for (int i=0; i<bottomrow; i++) {
      positionAlloc->Add(Vector(Xpos, brheight, 0.0));
      Xpos += spacing;
}
sessile.SetPositionAllocator (positionAlloc);
sessile.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
sessile.Install (fixedpos);
```

Next we set up the mover node. ConstantVelocityMobilityModel is a subclass of MobilityModel. At the end we print out a couple things just for confirmation.

```
Vector pos (0, mheight+brheight, 0);
Vector vel (speed, 0, 0);
MobilityHelper mobile;
mobile.SetMobilityModel("ns3::ConstantVelocityMobilityModel");        // no Attributes
mobile.Install(mover);
cvmm = mover->GetObject<ConstantVelocityMobilityModel> ();
cvmm->SetPosition(pos);
cvmm->SetVelocity(vel);
std::cout << "position: " << cvmm->GetPosition() << " velocity: " << cvmm->GetVelocity() <<
std::cout << "mover mobility model: " << mobile.GetMobilityModelType() << std::endl;
```

Now we configure Ad hoc On-demand Distance Vector routing.

```
AodvHelper aodv;
OlsrHelper olsr;
Ipv4ListRoutingHelper listrouting;
//listrouting.Add(olsr, 10);                          // generates less traffic
listrouting.Add(aodv, 10);                            // fastest to find new routes
```

Uncommenting the olsr line (and commenting out the last line) is all that is necessary to change to OLSR routing. OLSR is slower to find new routes, but sends less traffic.

Now we set up the IP addresses. This is straightforward as all the nodes are on a single subnet.

```
InternetStackHelper internet;
internet.SetRoutingHelper(listrouting);
internet.Install (fixedpos);
internet.Install (mover);

Ipv4AddressHelper ipv4;
NS_LOG_INFO ("Assign IP Addresses.");
ipv4.SetBase ("10.1.1.0", "255.255.255.0");           // there is only one subnet
Ipv4InterfaceContainer i = ipv4.Assign (devices);
```

Now we create a receiving application UdpServer on node mover, and a sending application UdpClient on the lower-left node. These applications generate their own sequence numbers, which show up in the ns-3 tracefiles marked with ns3::SeqTsHeader. As in *17.2 A Single TCP Sender*, we use Config::SetDefault() and CreateObject<>() to construct the applications.

```
uint16_t port = 80;
// create a receiving application (UdpServer) on node mover
Address sinkaddr(InetSocketAddress (Ipv4Address::GetAny (), port));
Config::SetDefault("ns3::UdpServer::Port", UintegerValue(port));

Ptr<UdpServer> UdpRecvApp = CreateObject<UdpServer>();
UdpRecvApp->SetStartTime(Seconds(0.0));
UdpRecvApp->SetStopTime(Seconds(endtime+60));
mover->AddApplication(UdpRecvApp);

Ptr<Ipv4> m4 = mover->GetObject<Ipv4>();
Ipv4Address Maddr = m4->GetAddress(1,0).GetLocal();
std::cout << "IPv4 address of mover: " << Maddr << std::endl;
Address moverAddress (InetSocketAddress (Maddr, port));
```

Here is the `UdpClient` sending application:

```
Config::SetDefault("ns3::UdpClient::MaxPackets", UintegerValue(packetcount));
Config::SetDefault("ns3::UdpClient::PacketSize", UintegerValue(packetsize));
Config::SetDefault("ns3::UdpClient::Interval",   TimeValue (MicroSeconds (interval)));

Ptr<UdpClient> UdpSendApp = CreateObject<UdpClient>();
UdpSendApp -> SetRemote(Maddr, port);
UdpSendApp -> SetStartTime(Seconds(0.0));
UdpSendApp -> SetStopTime(Seconds(endtime));
lowerleft->AddApplication(UdpSendApp);
```

We now set up tracing. The first, commented-out line enables pcap-format tracing, which we do not need here. The `YansWifiPhyHelper` object supports tracing only of "receive" (**r**) and "transmit" (**t**) records; the `PointtoPointHelper` of *17.2 A Single TCP Sender* also traced enqueue and drop records.

```
//wifiPhyHelper.EnablePcap (tracebase, devices);

AsciiTraceHelper ascii;
wifiPhyHelper.EnableAsciiAll (ascii.CreateFileStream (tracebase + ".tr"));

// create animation file, to be run with 'netanim'
AnimationInterface anim (tracebase + ".xml");
anim.SetMobilityPollInterval(Seconds(0.1));
```

If we view the animation with `netanim`, the moving node's motion is clear. The `mover` node, however, sometimes appears to transmit back to both the fixed-row node below left and the fixed-row node below right. These transmissions represent the Wi-Fi link-layer ACKs; they appear to be sent to two fixed-row nodes because what `netanim` is actually displaying with its blue links is transmission every other node in range.

We can also "view" the motion in text format by uncommenting the first line below.

```
//Simulator::Schedule(Seconds(position_interval), &printPosition);

Simulator::Schedule(Seconds(endtime), &stopMover);
```

Finally it is time to run the simulator, and print some final output.

```
  Simulator::Stop(Seconds (endtime+60));
  Simulator::Run ();
  Simulator::Destroy ();

  int pktsRecd = UdpRecvApp->GetReceived();
  std::cout << "packets received: " << pktsRecd << std::endl;
  std::cout << "packets recorded as lost: " << (UdpRecvApp->GetLost()) << std::endl;
  std::cout << "packets actually lost: " << (packetcount - pktsRecd) << std::endl;

  return 0;
}
```

17.3.1 Tracefile Analysis

The tracefile provides no enqueue records, and Wi-Fi doesn't have fixed links; how can we verify that packets are being forwarded correctly? One thing we *can* do with the tracefile is to look at each value of the `UdpServer` application sequence number, and record

- when it was received by node `mover`
- when it was transmitted by any fixed-row node

If we do this, we get output like the following:

```
packet 0 received at 0.0248642, forwarded by 0 at 0.0201597
packet 1 received at 0.0547045, forwarded by 0 at 0.05
...
packet 499 received at 24.9506, forwarded by 0 at 24.95
packet 500 NOT recd, forwarded by 0 at 25, forwarded by 0 at 25.0019, forwarded by 0 at 25
packet 501 received at 25.0864, forwarded by 0 at 25.0767, forwarded by 1 at 25.0817
packet 502 received at 25.1098, forwarded by 0 at 25.1, forwarded by 1 at 25.1051
...
packet 1000 NOT recd, forwarded by 0 at 50, forwarded by 1 at 50.001, forwarded by 1 at 50
packet 1001 received at 50.082, forwarded by 0 at 50.0683, forwarded by 1 at 50.0722, forwa
packet 1002 received at 50.1107, forwarded by 0 at 50.1, forwarded by 1 at 50.101, forwarde
...
packet 1499 received at 74.9525, forwarded by 0 at 74.95, forwarded by 1 at 74.951, forwar
packet 1500 NOT recd, forwarded by 0 at 75, forwarded by 1 at 75.001, forwarded by 2 at 75
packet 1501 NOT recd, forwarded by 0 at 75.05
packet 1502 received at 75.1484, forwarded by 0 at 75.1287, forwarded by 1 at 75.1299, forw
packet 1503 received at 75.1621, forwarded by 0 at 75.15, forwarded by 1 at 75.151, forwar
...
```

That is, packets 0-499 were transmitted only by node 0. Packet 500 was never received by `mover`, but there were seven transmission attempts; these seven attempts follow the rules described in *3.7.1 Wi-Fi and Collisions*. Packets starting at 501 were transmitted by node 0 and then later by node 1. Similarly, packet 1000 was lost, and after that each packet arriving at `mover` was first transmitted by nodes 0, 1 and 2, in that order. In other words, packets are indeed being forwarded rightward along the line of fixed-row nodes until a node is reached that is in range of `mover`.

17.3.2 AODV Performance

If we change the line

```
listrouting.Add(aodv, 10);
```

to

```
listrouting.Add(dsdv, 10);
```

we find that the loss count goes from 4 packets out of 2000 to 398 out of 2000; for OLSR routing the loss count is 426. As we discussed in *16.6 Wireless Simulation*, the loss of one data packet triggers the AODV implementation to look for a new route. The DSDV and OLSR implementations, on the other hand, only look for new routes at regularly spaced intervals.

17.4 Exercises

In preparation.

Sometimes simulations are not possible or not practical, and network experiments must be run on actual machines. One can always use a set of interconnected virtual machines, but even pared-down virtual machines consume sufficient resources that it is hard to create a network of more than a handful of nodes. **Mininet** is a system that supports the creation of lightweight logical nodes that can be connected into networks. These nodes are sometimes called **containers**, or, more accurately, **network namespaces**. Virtual-machine technology is not used. These containers consume sufficiently few resources that networks of over a thousand nodes have been created, running on a single laptop. While Mininet was originally developed as a testbed for software-defined networking (*2.7 Software-Defined Networking*), it works just as well for demonstrations and experiments involving traditional networking.

A Mininet container is a process (or group of processes) that no longer has access to all the host system's "native" network interfaces, much as a process that has executed the `chroot()` system call no longer has access to the full filesystem. Mininet containers then are assigned virtual Ethernet interfaces (see the **ip-link** man page entries for **veth**), which are connected to other containers through virtual Ethernet links. The use of veth links ensures that the virtual links behave like Ethernet, though it may be necessary to disable TSO (*12.5 TCP Offloading*) to view Ethernet packets in WireShark as they would appear on the (virtual) wire. Any process started within a Mininet container inherits the container's view of network interfaces.

For efficiency, Mininet containers all share the same filesystem by default. This makes setup simple, but sometimes causes problems with applications that expect individualized configuration files in specified locations. Mininet containers *can* be configured with different filesystem views, though we will not do this here.

Mininet is a form of network **emulation**, as opposed to simulation. An important advantage of emulation is that all network software, at any layer, is simply run "as is". In a simulator environment, on the other hand, applications and protocol implementations need to be ported to run within the simulator before they can be used. A drawback of emulation is that as the network gets large and complex the emulation may slow down. In particular, it is not possible to emulate link speeds faster than the underlying hardware can support. (It is also not possible to emulate non-linux network software.)

The Mininet group maintains extensive documentation; three useful starting places are the Overview, the Introduction and the FAQ.

The goal of this chapter is to present a series of Mininet examples. Most examples are in the form of a self-contained Python2 file (Mininet does not at this time support Python3). Each Mininet Python2 file configures the network and then starts up the Mininet command-line interface (which is necessary to start commands on the various node containers). The use of self-contained Python files arguably makes the configurations easier to edit, and avoids the complex command-line arguments of many standard Mininet examples. The Python code uses what the Mininet documentation calls the mid-level API.

The Mininet distribution comes with its own set of examples, in the directory of that name. A few of particular interest are listed below; with the exception of linuxrouter.py, the examples presented here do not use any of these techniques.

- bind.py: demonstrates how to give each Mininet node its own private directory (otherwise all nodes share a common filesystem)

- controllers.py: demonstrates how to arrange for multiple SDN controllers, with different switches connecting to different controllers
- limit.py: demonstrates how to set CPU utilization limits (and link bandwidths)
- linuxrouter.py: creates a node that acts as a router. Any host node can act as a router, though, provided we enable forwarding with `sysctl net.ipv4.ip_forward=1`
- miniedit.py: a graphical editor for Mininet networks
- mobility.py: demonstrates how to move a host from one switch to another
- nat.py: demonstrates how to connect hosts to the Internet
- tree1024.py: creates a network with 1024 nodes

We will occasionally need supplemental programs as well, *eg* for sending, monitoring or receiving traffic. These are meant to be modified as necessary to meet circumstances; they contain few command-line option settings. Most of these supplemental programs are written, perhaps confusingly, in Python3. Python2 files are run with the `python` command, while Python3's command is `python3`. Alternatively, given that all these programs are running under linux, one can make all Python files executable and be sure that the first line is either `#!/usr/bin/python` or `#!/usr/bin/python3` as appropriate.

18.1 Installing Mininet

Mininet runs only under the linux operating system. Windows and Mac users can, however, easily run Mininet in a single linux virtual machine. Even linux users may wish to do this, as running Mininet has a nontrivial potential to affect normal operation (a virtual-switch process started by Mininet has, for example, interfered with the suspend feature on the author's laptop).

The Mininet group maintains a virtual machine with a current Mininet installation at their downloads site. The download file is actually a .zip file, which unzips to a modest .ovf file defining the specifications of the virtual machine and a much larger (~2 GB) .vmdk file representing the virtual disk image. (Some unzip versions have trouble with unzipping very large files; if that happens, search online for an alternative unzipper.)

There are several choices for virtual-machine software; two options that are well supported and free (as of 2017) for personal use are VirtualBox and VMware Workstation Player. The .ovf file should open in either (in VirtualBox with the "import appliance" option). However, it may be easier simply to create a new linux virtual machine and specify that it is to use an existing virtual disk; then select the downloaded .vmdk file as that disk.

Both the login name and the password for the virtual machine is "mininet". Once logged in, the sudo command can be used to obtain root privileges, which are needed to run Mininet. It is safest to do this on a command-by-command basis; *eg* `sudo python switchline.py`. It is also possible to keep a terminal window open that is permanently logged in as root, *eg* via `sudo bash`.

Another option is to set up a linux virtual machine from scratch (*eg* via the Ubuntu distribution) and then install Mininet on it, although the preinstalled version also comes with other useful software, such as the Pox controller for OpenFlow switches.

The preinstalled version does *not*, however, come with any graphical-interface desktop. One *can* install the full Ubuntu desktop with the command (as root) `apt-get install ubuntu-desktop`. This will, however, add more than 4 GB to the virtual disk. A lighter-weight option, recommended by the Mininet site, is to install the alternative desktop environment lxde; it is half the size of Ubuntu. Install it with

```
apt-get install xinit lxde
```

The standard graphical text editor included with lxde is leafpad, though of course others (*eg* gedit or emacs) can be installed as well.

After desktop installation, the command `startx` will be necessary after login to start the graphical environment (though one can automate this). A standard recommendation for new Debian-based linux systems, before installing anything else, is

```
apt-get update
apt-get upgrade
```

Most virtual-machine software offers a special package to improve compatibility with the host system. One of the most annoying incompatibilities is the tendency of the virtual machine to grab the mouse and not allow it to be dragged outside the virtual-machine window. (Usually a special keypress releases the mouse; on VirtualBox it is the **right-hand Control key** and on VMWare Player it is **Control-Alt**.) Installation of the compatibility package (in VirtualBox called Guest Additions) usually requires mounting a CD image, with the command

```
mount /dev/cdrom /media/cdrom
```

The Mininet installation itself can be upgraded as follows:

```
cd /home/mininet/mininet
git fetch
git checkout master   # Or a specific version like 2.2.1
git pull
make install
```

The simplest environment for beginners is to install a graphical desktop (*eg* lxde) and then work within it. This allows seamless opening of xterm and WireShark as necessary. Enabling copy/paste between the virtual system and the host is also convenient.

However, it is also possible to work entirely without the desktop, by using multiple ssh logins with X-windows forwarding enabled:

```
ssh -X -l username mininet
```

This does require an X-server on the host system, but these are available even for Windows (see, for example, Cygwin/X). At this point one can open a graphical program on the ssh command line, *eg* `wireshark &` or `gedit mininet-demo.py &`, and have the program window display properly (or close to properly).

Finally, it is possible to access the Mininet virtual machine solely via ssh terminal sessions, without X-windows, though one then cannot launch xterm or WireShark.

18.2 A Simple Mininet Example

Starting Mininet via the `mn` command (as root!), with no command-line arguments, creates a simple network of two hosts and one switch, h1–s1–h2, and starts up the Mininet command-line interface (CLI). By convention, Mininet host names begin with 'h' and switch names begin with 's'; numbering begins with 1.

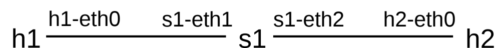
At this point one can issue various Mininet-CLI commands. The command `nodes`, for example, yields the following output:

```
available nodes are:
c0 h1 h2 s1
```

The node `c0` is the *controller* for the switch `s1`. The default controller action her makes `s1` behave like an Ethernet learning switch (*2.4.1 Ethernet Learning Algorithm*). The command `intfs` lists the interfaces for each of the nodes, and `links` lists the connections, but the most useful command is `net`, which shows the nodes, the interfaces and the connections:

```
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0
```

From the above, we can see that the network looks like this:



18.2.1 Running Commands on Nodes

The next step is to run commands on individual nodes. To do this, we use the Mininet CLI and prefix the command name with the node name:

```
h1 ifconfig
h1 ping h2
```

The first command here shows that h1 (or, more properly, h1-eth0) has IP address 10.0.0.1. Note that the name 'h2' in the second is recognized. The `ifconfig` command also shows the MAC address of h1-eth0, which may vary but might be something like 62:91:68:bf:97:a0. We will see in the following section how to get more human-readable MAC addresses.

There is a special Mininet command `pingall` that generates pings between each pair of hosts.

We can open a full shell window on node `h1` using the Mininet command below; this works for both host nodes and switch nodes.

```
xterm h1
```

Note that the xterm runs with root privileges. From within the xterm, the command `ping h2` now fails, as hostname `h2` is not recognized. We can switch to `ping 10.0.0.2`, or else add entries to /etc/hosts for the IP addresses of `h1` and `h2`:

```
10.0.0.1        h1
10.0.0.2        h2
```

As the Mininet system shares its filesystem with `h1` and `h2`, this means that the names `h1` and `h2` are now defined everywhere within Mininet (though be forewarned that when a different Mininet configuration assigns different addresses to `h1` or `h2`, chaos will ensue).

From within the xterm on `h1` we might try logging into `h2` via ssh: `ssh h2` (if h2 is defined in /etc/hosts as above). But the connection is refused: the ssh server is not running on node `h2`. We will return to this in the following example.

We can also start up WireShark, and have it listen on interface h1-eth0, and see the progress of our pings. (We can also usually start WireShark from the `mininet>` prompt using `h1 wireshark &`.)

Similarly, we can start an xterm on the switch and start WireShark there. However, there is another option, as switches by default share all their network systems with the Mininet host system. (In terms of the container model, switches do not by default get their own network namespace; they share the "root" namespace with the host.) We can see this by running the following from the Mininet command line

```
s1 ifconfig
```

and comparing the output with that of `ifconfig` run on the Mininet host, while Mininet is running but *outside* of the Mininet process itself. We see these interfaces:

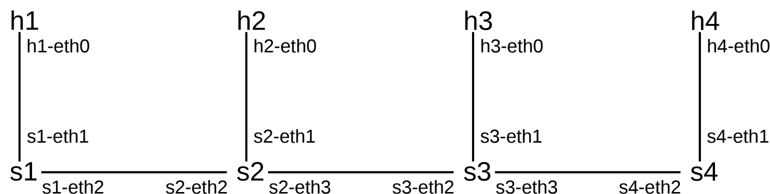
```
eth0
lo
s1
s1-eth1
s1-eth2
```

We see the same interfaces on the controller node `c0`, even though the `net` and `intfs` commands above showed no interfaces for `c0`.

Running WireShark on, say, `s1-eth1` is an excellent way to observe traffic on a nearly idle network; by default, the Mininet nodes are not connected to the outside world. As an example, suppose we start up xterm windows on h1 and h2, and run `netcat -l 5432` on h2 and then `netcat 10.0.0.2 5432` on h1. We can then watch the ARP exchange, the TCP three-way handshake, the content delivery and the connection teardown, with most likely no other traffic at all. Wireshark filtering is not needed.

18.3 Multiple Switches in a Line

The next example creates the topology below. All hosts are on the same subnet.



The Mininet-CLI command `links` can be used to determine which switch interface is connected to which neighboring switch interface.

The full Python2 program is switchline.py; to run it use

```
python switchline.py
```

This configures the network and starts the Mininet CLI. The default number of host/switch pairs is 4, but this can be changed with the `-N` command-line parameter, for example `python switchline.py -N 5`.

We next describe selected parts of switchline.py. The program starts by building the network topology object, `LineTopo`, extending the built-in Mininet class `Topo`, and then call `Topo.addHost()` to create the host nodes. (We here override __`init()`__, but overriding `build()` is actually more common.)

```
class LineTopo( Topo ):
   def __init__( self , **kwargs):
       "Create linear topology"
       super(LineTopo, self).__init__(**kwargs)
       h = []          # list of hosts; h[0] will be h1, etc
       s = []          # list of switches

       for key in kwargs:
          if key == 'N': N=kwargs[key]

       # add N hosts  h1..hN
       for i in range(1,N+1):
          h.append(self.addHost('h' + str(i)))
```

Method `Topo.addHost()` takes a string, such as "h2", and builds a host object of that name. We immediately append the new host object to the list `h[]`. Next we do the same to switches, using `Topo.addSwitch()`:

```
# add N switches s1..sN
for i in range(1,N+1):
   s.append(self.addSwitch('s' + str(i)))
```

Now we build the links, with `Topo.addLink`. Note that h[0]..h[N-1] represent h1..hN. First we build the host-switch links, and then the switch-switch links.

```
for i in range(N):                # Add links from hi to si
   self.addLink(h[i], s[i])

for i in range(N-1):              # link switches
   self.addLink(s[i],s[i+1])
```

Now we get to the main program. We use `argparse` to support the `-N` command-line argument.

```
def main(**kwargs):
    parser = argparse.ArgumentParser()
    parser.add_argument('-N', '--N', type=int)
    args = parser.parse_args()
    if args.N is None:
        N = 4
```

```
    else:
        N = args.N
```

Next we create a LineTopo object, defined above. We also set the log-level to 'info'; if we were having problems we would set it to 'debug'.

```
ltopo = LineTopo(N=N)
setLogLevel('info')
```

Finally we're ready to create the Mininet net object, and start it. We've specified the type of switch here, though at this point that does not really matter. It *does* matter that we're using the DefaultController, as otherwise the switches will not behave automatically as Ethernet learning switches. The autoSetMacs option sets the *host* MAC addresses to 00:00:00:00:00:01 through 00:00:00:00:00:04 (for N=4), which can be a great convenience when manually examining Ethernet addresses.

```
net = Mininet(topo = ltopo, switch = OVSKernelSwitch,
            controller = DefaultController,
            autoSetMacs = True
            )
net.start()
```

The next bit starts /usr/sbin/sshd on each node. This command automatically puts itself in the background; otherwise we would need to add an '&' to the string to run the command in the background.

```
for i in range(1, N+1):
   hi = net['h' + str(i)]
   hi.cmd('/usr/sbin/sshd')
```

Finally we start the Mininet CLI, and, when that exits, we stop the emulation.

```
CLI( net)
net.stop()
```

Using sshd requires a small bit of configuration, if ssh for the root user has not been set up already. We must first run ssh-keygen, which creates the directory /root/.ssh and then the public and private key files, id_rsa.pub and id_rsa respectively. There is no need, in this setting, to protect the keys with a password. The second step is to go to the .ssh directory and copy id_rsa.pub to the (new) file authorized_keys (if the latter file already exists, append id_rsa.pub to it). This will allow passwordless ssh connections between the different Mininet hosts.

Because we started sshd on each host, the command ssh 10.0.0.4 on h1 should successfully connect to h4. The first time a connection is made from h1 to h4 (as root), ssh will ask for confirmation, and then store h4's key in /root/.ssh/known_hosts. As this is the same file for all Mininet nodes, due to the common filesystem, a subsequent request to connect from h**2** to h4 will succeed immediately; h4 has already been authenticated for all nodes.

18.3.1 Running a webserver

Now let's run a web server on, say, host 10.0.0.4 of the switchline.py example above. Python includes a simple implementation that serves up the files in the directory in which it is started. After switchline.py is

running, start an xterm on host h4, and then change directory to `/usr/share/doc` (where there are some html files). Then run the following command (the 8000 is the server port number):

```
python -m SimpleHTTPServer 8000
```

If this is run in the background somewhere, output should be redirected to /dev/null or else the server will eventually hang.

The next step is to start a browser. If the lxde environment has been installed (*18.1 Installing Mininet*), then the chromium browser should be available. Start an xterm on host h1, and on h1 run the following (the `--no-sandbox` option is necessary to run chromium as root):

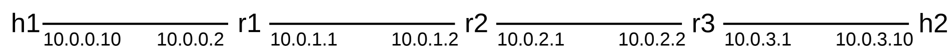
```
chromium-browser --no-sandbox
```

Assuming chromium opens successfully, enter the following URL: `10.0.0.4:8000`. If chromium does not start, try `wget 10.0.0.4:8000`, which stores what it receives as the file `index.html`. Either way, you should see a listing of the `/usr/share/doc` directory. It is possible to browse subdirectories, but only browser-recognized filetypes (*eg* `.html`) will open directly. A few directories with subdirectories named `html` are `iperf`, `iptables` and `xarchiver`; try navigating to these.

18.4 IP Routers in a Line

In the next example we build a Mininet example involving a router rather than a switch. A router here is simply a multi-interface Mininet host that has IP forwarding enabled in its linux kernel. Mininet support for multi-interface hosts is somewhat fragile; interfaces may need to be initialized in a specific order, and IP addresses often cannot be assigned at the point when the link is created. In the code presented below we assign IP addresses using calls to `Node.cmd()` used to invoke the linux command `ifconfig` (Mininet containers do not fully support the use of the alternative `ip addr` command).

Our first router topology has only two hosts, one at each end, and N routers in between; below is the diagram with N=3. All subnets are /24. The program to set this up is routerline.py, here invoked as `python routerline.py -N 3`. We will use N=3 in most of the examples below. A somewhat simpler version of the program, which sets up the topology specifically for N=3, is routerline3.py.



In both versions of the program, routing entries are created to route traffic from h1 to h2, *but not back again*. That is, every router has a route to 10.0.3.0/24, but only r1 knows how to reach 10.0.0.0/24 (to which r1 is directly connected). We can verify the "one-way" connectedness by running `WireShark` or `tcpdump` on h2 (perhaps first starting an `xterm` on h2), and then running `ping 10.0.3.10` on h1 (perhaps using the Mininet command `h1 ping h2`). `WireShark` or `tcpdump` should show the arriving ICMP ping packets from `h1`, and also the arriving ICMP Destination Network Unreachable packets from `r3` as `h2` tries to reply (see *7.11 Internet Control Message Protocol*).

It turns out that one-way routing is considered to be suspicious; one interpretation is that the packets involved have a source address that shouldn't be possible, perhaps spoofed. Linux provides the interface configuration option `rp_filter` – reverse-path filter – to block the forwarding of packets for which the router does not

have a route back to the packet's source. This must be disabled for the one-way example to work; see the notes on the code below.

Despite the lack of connectivity, we can reach h2 from h1 via a hop-by-hop sequence of ssh connections (the program enables `sshd` on each host and router):

```
h1: slogin 10.0.0.2
r1: slogin 10.0.1.2
r2: slogin 10.0.2.2
r3: slogin 10.0.3.10 (that is, h3)
```

To get the one-way routing to work from h1 to h2, we needed to tell r1 and r2 how to reach destination 10.0.3.0/24. This can be done with the following commands (which are executed automatically if we set `ENABLE_LEFT_TO_RIGHT_ROUTING = True` in the program):

```
r1: ip route add to 10.0.3.0/24 via 10.0.1.2
r2: ip route add to 10.0.3.0/24 via 10.0.2.2
```

To get full, bidirectional connectivity, we can create the following routes to 10.0.0.0/24:

```
r2: ip route add to 10.0.0.0/24 via 10.0.1.1
r3: ip route add to 10.0.0.0/24 via 10.0.2.1
```

When building the network topology, the single-interface hosts can have all their attributes set at once (the code below is from routerline3.py:

```
h1 = self.addHost( 'h1', ip='10.0.0.10/24', defaultRoute='via 10.0.0.2' )
h2 = self.addHost( 'h2', ip='10.0.3.10/24', defaultRoute='via 10.0.3.1' )
```

The routers are also created with `addHost()`, but with separate steps:

```
r1 = self.addHost( 'r1' )
r2 = self.addHost( 'r2' )
...

self.addLink( h1, r1, intfName1 = 'h1-eth0', intfName2 = 'r1-eth0')
self.addLink( r1, r2, inftname1 = 'r1-eth1', inftname2 = 'r2-eth0')
```

Later on the routers get their IPv4 addresses:

```
r1 = net['r1']
r1.cmd('ifconfig r1-eth0 10.0.0.2/24')
r1.cmd('ifconfig r1-eth1 10.0.1.1/24')
r1.cmd('sysctl net.ipv4.ip_forward=1')
rp_disable(r1)
```

The `sysctl` command here enables forwarding in r1. The `rp_disable(r1)` call disables Linux's default refusal to forward packets if the router does not have a route back to the packet's source; this is often what is wanted in the real world but *not* necessarily in routing demonstrations. It too is ultimately implemented via `sysctl` commands.

18.5 IP Routers With Simple Distance-Vector Implementation

The next step is to automate the discovery of the route from h1 to h2 (and back) by using a simple distance-vector routing-update protocol. We present a partial implementation of the Routing Information Protocol, RIP, as defined in **RFC 2453**.

The distance-vector algorithm is described in *9.1 Distance-Vector Routing-Update Algorithm*. In brief, the idea is to add a cost attribute to the forwarding table, so entries have the form $\langle$destination,next_hop,cost$\rangle$. Routers then send $\langle$destination,cost$\rangle$ lists to their neighbors; these lists are referred to the RIP specification as **update messages**. Routers receiving these messages then process them to figure out the lowest-cost route to each destination. The format of the update messages is diagrammed below:

Addr Family	route_tag
IP Address	
Netmask	
Next_hop Address	
metric	

The full RIP specification also includes request messages, but the implementation here omits these. The full specification also includes split horizon, poison reverse and triggered updates (*9.2.1.1 Split Horizon* and *9.2.1.2 Triggered Updates*); we omit these as well. Finally, while we include code for the third **next_hop increase** case of *9.1.1 Distance-Vector Update Rules*, we do not include any test for whether a link is down, so this case is never triggered.

The implementation is in the Python3 file rip.py. Most of the time, the program is waiting to read update messages from other routers. Every `UPDATE_INTERVAL` seconds the program sends out its own update messages. All communication is via UDP packets sent using IP multicast, to the official RIP multicast address 224.0.0.9. Port 520 is used for both sending and receiving.

Rather than creating separate threads for receiving and sending, we configure a short (1 second) `recv()` timeout, and then after each timeout we check whether it is time to send the next update. An update can be up to 1 second late with this approach, but this does not matter.

The program maintains a "shadow" copy `RTable` of the real system forwarding table, with an added cost column. The real table is updated whenever a route in the shadow table changes. In the program, `RTable` is a dictionary mapping `TableKey` values (consisting of the IP address and mask) to `TableValue` objects containing the interface name, the cost, and the next_hop.

To run the program, a "production" approach would be to use Mininet's `Node.cmd()` to start up rip.py on each router, *eg* via `r.cmd('python3 rip.py &')` (assuming the file rip.py is located in the same directory in which Mininet was started). For demonstrations, the program output can be observed if the program is started in an xterm on each router.

18.5.1 Multicast Programming

Sending IP multicast involves special considerations that do not arise with TCP or UDP connections. The first issue is that we are sending to a multicast group – 224.0.0.9 – but don't have any multicast routes (multicast trees, *20.5 Global IP Multicast*) configured. What we would *like* is to have, at each router, traffic to 224.0.0.9 forwarded to each of its neighboring routers.

However, we do not actually want to configure multicast routes; all we want is to reach the immediate neighbors. Setting up a multicast tree presumes we know something about the network topology, and, at the point where RIP comes into play, we do not. The multicast packets we send should in fact *not* be forwarded by the neighbors (we will enforce this below by setting TTL); the multicast model here is very local. Even if we did want to configure multicast routes, linux does not provide a standard utility for manual multicast-routing configuration; see the **ip-mroute.8** man page.

So what we do instead is to create a socket for each separate router interface, and configure the socket so that it forwards its traffic only out its associated interface. This introduces a complication: we need to get the list of all interfaces, and then, for each interface, get its associated IPv4 addresses with netmasks. (To simplify life a little, we will assume that each interface has only a single IPv4 address.) The function `getifaddrdict()` returns a dictionary with interface names (strings) as keys and pairs (ipaddr,netmask) as values. If `ifaddrs` is this dictionary, for example, then `ifaddrs['r1-eth0']` might be `('10.0.0.2','255.255.255.0')`. We could implement `getifaddrdict()` straightforwardly using the Python module netifaces, though for demonstration purposes we do it here via low-level system calls.

We get the list of interfaces using `myInterfaces = os.listdir('/sys/class/net/')`. For each interface, we then get its IP address and netmask (in `get_ip_info(intf)`) with the following:

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
SIOCGIFADDR    = 0x8915     # from /usr/include/linux/sockios.h
SIOCGIFNETMASK = 0x891b
intfpack = struct.pack('256s', bytes(intf, 'ascii'))
# ifreq, below, is like struct ifreq in /usr/include/linux/if.h
ifreq    = fcntl.ioctl(s.fileno(), SIOCGIFADDR, intfpack)
ipaddrn  = ifreq[20:24]     # 20 is the offset of the IP addr in ifreq
ipaddr   = socket.inet_ntoa(ipaddrn)
netmaskn = fcntl.ioctl(s.fileno(), SIOCGIFNETMASK, intfpack)[20:24]
netmask  = socket.inet_ntoa(netmaskn)
return (ipaddr, netmask)
```

We need to create the socket here (never connected) in order to call `ioctl()`. The `SIOCGIFADDR` and `SIOCGIFNETMASK` values come from the C language include file; the Python3 libraries do not make these constants available but the Python3 `fcntl.ioctl()` call does pass the values we provide directly to the underlying C `ioctl()` call. This call returns its result in a C `struct ifreq`; the ifreq above is a Python version of this. The binary-format IPv4 address (or netmask) is at offset 20.

18.5.1.1 `createMcastSockets()`

We are now in a position, for each interface, to create a UDP socket to be used to send and receive on that interface. Much of the information here comes from the linux **socket.7** and **ip.7** man pages. The function `createMcastSockets(ifaddrs)` takes the dictionary above mapping interface names to

(ipaddr,netmask) pairs and, for each interface `intf`, configures it as follows. The list of all the newly configured sockets is then returned.

The first step is to obtain the interface's address and mask, and then convert these to 32-bit integer format as `ipaddrn` and `netmaskn`. We then enter the subnet corresponding to the interface into the shadow routing table `RTable` with a cost of 1 (and with a next_hop of `None`), via

```
RTable[TableKey(subnetn, netmaskn)] = TableValue(intf, None, 1)
```

Next we create the socket and begin configuring it, first by setting its read timeout to a short value. We then set the TTL value used by outbound packets to 1. This goes in the IPv4 header Time To Live field (*7.1 The IPv4 Header*); this means that no downstream routers will ever forward the packet. This is exactly what we want; RIP uses multicast only to send to immediate neighbors.

```
sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, 1)
```

We also want to be able to bind the same socket source address, `224.0.0.9` and port 520, to all the sockets we are creating here (the actual `bind()` call is below):

```
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

The next call makes the socket *receive* only packets arriving on the specified interface:

```
sock.setsockopt(socket.SOL_SOCKET, socket.SO_BINDTODEVICE, bytes(intf, 'ascii'))
```

We add the following to prevent packets sent on the interface from being delivered back to the sender; otherwise multicast delivery may do just that:

```
sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_LOOP, False)
```

The next call makes the socket *send* on the specified interface. Multicast packets do have IPv4 destination addresses, and normally the kernel chooses the sending interface based on the IP forwarding table. This call overrides that, in effect telling the kernel how to route packets sent via this socket. (The kernel may also be able to figure out how to route the packet from the subsequent call joining the socket to the multicast group.)

```
sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_IF, socket.inet_aton(ipaddr))
```

Finally we can join the socket to the multicast group represented by `224.0.0.9`. We also need the interface's IP address, `ipaddr`.

```
addrpair = socket.inet_aton('224.0.0.9')+ socket.inet_aton(ipaddr)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, addrpair)
```

The last step is to bind the socket to the desired address and port, with `sock.bind(('224.0.0.9', 520))`. This specifies the source address of outbound packets; it would fail (given that we are using the same socket address for multiple interfaces) without the `SO_REUSEADDR` configuration above.

18.5.2 The RIP Main Loop

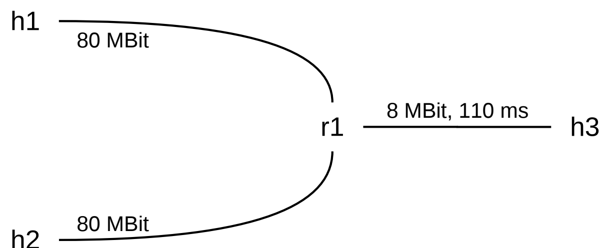
The rest of the implementation is relatively nontechnical. One nicety is the use of `select()` to wait for arriving packets on any of the sockets created by `createMcastSockets()` above; the alternatives might be to poll each socket in turn with a short read timeout or else to create a separate thread for each socket.

The `select()` call takes the list of sockets (and a timeout value) and returns a sublist consisting of those sockets that have data ready to read. Almost always, this will be just one of the sockets. We then read the data with `s.recvfrom()`, recording the source address `src` which will be used when we, next, call `update_tables()`. When a socket closes, it must be removed from the `select()` list, but the sockets here do not close; for more on this, see *18.6.1.2 dualreceive.py*.

The `update_tables()` function takes the incoming message (parsed into a list of `RipEntry` objects via `parse_msg()`) and the IP address from which it arrives, and runs the distance-vector algorithm of *9.1.1 Distance-Vector Update Rules*. `TK` is the `TableKey` object representing the new destination (as an (addr,netmask) pair). The **new destination** rule from *9.1.1 Distance-Vector Update Rules* is applied when `TK` is not present in the existing `RTable`. The **lower cost** rule is applied when `newcost < currentcost`, and the third **next_hop increase** rule is applied when `newcost > currentcost` but `currentnexthop == update_sender`.

18.6 TCP Competition: Reno vs Vegas

The next routing example uses the following topology in order to emulate competition between two TCP connections h1→h3 and h2→h3. We introduce Mininet features to set, on the links, an emulated bandwidth and delay, and to set on the router an emulated queue size. Our first application will be to arrange a competition between TCP Reno (*13 TCP Reno and Congestion Management*) and TCP Vegas (*15.6 TCP Vegas*). The Python2 file for running this Mininet configuration is competition.py.



To create links with bandwidth/delay support, we simply set `Link=TCLink` in the `Mininet()` call in `main()`. The TCLink class represents a Traffic Controlled Link. Next, in the topology section calls to `addLink()`, we add keyword parameters such as `bw=BottleneckBW` and `delay=DELAY`. To implement the bandwidth limit, Mininet then takes care of creating the virtual-Ethernet links with a **rate** constraint.

To implement the delay, Mininet uses a queuing hierarchy (*19.7 Hierarchical Queuing*). The hierarchy is managed by the **tc** (traffic control) command, part of the LARTC system. In the topology above, Mininet sets up h3's queue as an htb queue (*19.13.2 Linux htb*, *18.8 Linux Traffic Control (tc)*) with a **netem queue** below it (see the man page for **tc-netem.8**). The latter has a **delay** parameter set as requested, to 110 ms in our example here. Note that this means that the delay from h3 to r will be 110 ms, and the delay from r to h3 will be 0 ms.

The queue configuration is also handled via the **tc** command. Again Mininet configures r's r-eth3 interface to have an htb queue with a netem queue below it. Using the `tc qdisc show` command we can see that the "handle" of the netem queue is 10:; we can now set the maximum queue size to, for example, 25 with the following command on r:

```
tc qdisc change dev r-eth3 handle 10: netem limit 25
```

18.6.1 Running A TCP Competition

In order to arrange a TCP competition, we need the following tools:

- sender.py, to open the TCP connection and send bulk data, after requesting a specific TCP congestion-control mechanism (Reno or Vegas)
- dualreceive.py, to receive data from two connections and track the results
- randomtelnet.py, to send random additional data to break TCP phase effects.
- wintracker.py, to monitor the number of packets a connection has in flight (a good estimator of `cwnd`).

18.6.1.1 sender.py

The Python3 program sender.py is similar to tcp_stalkc.py, except that it allows specification of the TCP congestion algorithm. This is done with the following `setsockopt()` call:

```
s.setsockopt(socket.IPPROTO_TCP, TCP_CONGESTION, cong)
```

where `cong` is "reno" or "cubic" or some other available TCP flavor. The list is at /proc/sys/net/ipv4/tcp_allowed_congestion_control.

See also *15.1 Choosing a TCP on linux*.

18.6.1.2 dualreceive.py

The receiver for sender.py's data is dualreceive.py. It listens on two ports, by default 5430 and 5431, and, when both connections have been made, begins reading. The main loop starts with a call to `select()`, where `sset` is the list of all (both) connected sockets:

```
sl,_,_ = select(sset, [], [])
```

The value `sl` is a sublist of sset consisting of the sockets with data ready to read. It will normally be a list consisting of a single socket, though with so much data arriving it may sometimes contain both. We then call `s.recv()` for s in sl, and record in either `count1` or `count2` the running total of bytes received.

If a sender closes a socket, this results in a read of 0 bytes. At this point dualreceive.py must close the socket, at which point it *must* be removed from `sset` as it will otherwise always appear in the `sl` list.

We repeatedly set a timer (in `printstats()`) to print the values of `count1` and `count2` at 0.1 second intervals, reflecting the cumulative amounts of data received by the connections. (If the variable `PRINT_CUMULATIVE` is set to `False`, then the values printed are the amounts of data received in the last 0.1 seconds.) If the TCP competition is fair, `count1` and `count2` should stay approximately equal. When `printstats()` detects no change in `count1` and `count2`, it exits.

In Python, calling `exit()` only exits the current thread; the other threads keep running.

18.6.1.3 randomtelnet.py

In *16.3.4 Phase Effects* we show that, with completely deterministic travel times, two competing TCP connections can have throughputs differing by a factor of as much as 10 simply because of unfortunate synchronizations of transmission times. We *must* introduce at least some degree of packet-arrival-time randomization in order to obtain meaningful results.

In *16.3.6 Phase Effects and overhead* we used the ns2 `overhead` attribute for this. This is not available in real networks, however. The next-best thing is to introduce some random telnet-like traffic, as in *16.3.7 Phase Effects and telnet traffic*. This is the purpose of randomtelnet.py.

This program sends packets at random intervals; the lengths of the intervals are exponentially distributed, meaning that to find the length of the next interval we choose X randomly between 0 and 1 (with a *uniform* distribution), and then set the length of the wait interval to a constant times -log(X). The packet sizes are 210 bytes (a very atypical value for real telnet traffic). Crucially, the average rate of sending is held to a small fraction (by default 1%) of the available bottleneck bandwidth, which is supplied as a constant `BottleneckBW`. This means the randomtelnet traffic should not interfere significantly with the competing TCP connections (which, of course, have no additional interval whatsoever between packet transmissions, beyond what is dictated by sliding windows). The randomtelnet traffic appears to be quite effective at eliminating TCP phase effects.

Randomtelnet.py sends to port 5433 by default. We will usually use `netcat` (*12.6.2 netcat again*) as the receiver, as we are not interested in measuring throughput for this traffic.

18.6.1.4 Monitoring `cwnd` with wintracker.py

At the end of the competition, we can look at the dualreceive.py output and determine the overall throughput of each connection, as of the time when the first connection to send all its data has just finished. We can also plot throughput at intervals by plotting successive differences of the cumulative-throughput values.

However, this does not give us a view of each connection's `cwnd`, which is readily available when modeling competition in a simulator. Indeed, getting direct access to a connection's `cwnd` is nearly impossible, as it is a state variable in the sender's kernel.

However, we can do the next best thing: monitor the number of packets (or bytes) a connection has in flight; this is the difference between the highest byte sent and the highest byte acknowledged. The highest byte ACKed is one less than the value of the ACK field in the most recent ACK packet, and the highest byte sent is one less than the value of the SEQ field, plus the packet length, in the most recent DATA packet.

To get these ACK and SEQ numbers, however, requires eavesdropping on the network connections. We can do this using a packet-capture library such as libpcap. The Pcapy Python2 (not Python3) module is a wrapper for libpcap.

The program wintracker.py uses Pcapy to monitor packets on the interfaces r-eth1 and r-eth2 of router r. It would be slightly more accurate to monitor on h1-eth0 and h2-eth0, but that entails separate monitoring of two different nodes, and the difference is small as the h1–r and h2–r links have negligible delay and no queuing. Wintracker.py must be configured to monitor only the two TCP connections that are competing.

The way libpcap, and thus Pcapy, works is that we first create a **packet filter** to identify the packets we want to capture. The filter for both connections is

```
host 10.0.3.10 and tcp and portrange 5430-5431
```

The host is, of course, h3; packets are captured if either source host or destination host is h3. Similarly, packets are captured if either the source port or the destination port is either 5430 or 5431. The connection from h1 to h3 is to port 5430 on h3, and the connection from h2 to h3 is to port 5431 on h3.

For the h1–h3 connection, each time a packet arrives heading from h1 *to* h3 (in the code below we determine this because the destination port `dport` is 5430), we save in `seq1` the TCP header SEQ field plus the packet length. Each time a packet is seen heading from h3 to h1 (that is, with *source* port 5430), we record in `ack1` the TCP header ACK field. The packets themselves are captured as arrays of bytes, but we can determine the offset of the TCP header and read the four-byte SEQ/ACK values with appropriate helper functions:

```
_,p = cap1.next()                             # p is the captured packet
...
(_,iphdr,tcphdr,data) = parsepacket(p)           # find the headers
sport = int2(tcphdr, TCP_SRCPORT_OFFSET)         # extract port numbers
dport = int2(tcphdr, TCP_DSTPORT_OFFSET)
if dport == port1:                               # port1 == 5430
    seq1 = int4(tcphdr, TCP_SEQ_OFFSET) + len(data)
elif sport == port1:
    ack1 = int4(tcphdr, TCP_ACK_OFFSET)
```

Separate threads are used for each connection, as there is no variant of `select()` available to return the next captured packet of either connection.

Both the SEQ and ACK fields have had ISN_A added to them, but this will cancel out when we subtract. The SEQ and ACK values are subject to 32-bit wraparound, but subtraction again saves us here.

As with dualreceive.py, a timer fires every 100 ms and prints out the differences `seq1-ack1` and `seq2-ack2`. This isn't completely thread-safe, but it is close enough. There is some noise in the results; we can minimize that by taking the average of several differences in a row.

18.6.1.5 Synchronizing the start

The next issue is to get both senders to start at about the same time. We could use two ssh commands, but ssh commands can take several hundred milliseconds to complete. A faster method is to use netcat to trigger the start. On h1 and h2 we run shell scripts like the one below (separate values for $PORT and $CONG are needed for each of h1 and h2, which is simplest to implement with separate scripts, say h1.sh and h2.sh):

```
netcat -l 2345
python3 sender.py $BLOCKS 10.0.3.10 $PORT $CONG
```

We then start both at very close to the same time with the following on r (not on h3, due to the delay on the r–h3 link); these commands typically complete in under ten milliseconds.

```
echo hello | netcat h1 2345
echo hello | netcat h2 2345
```

The full sequence of steps is

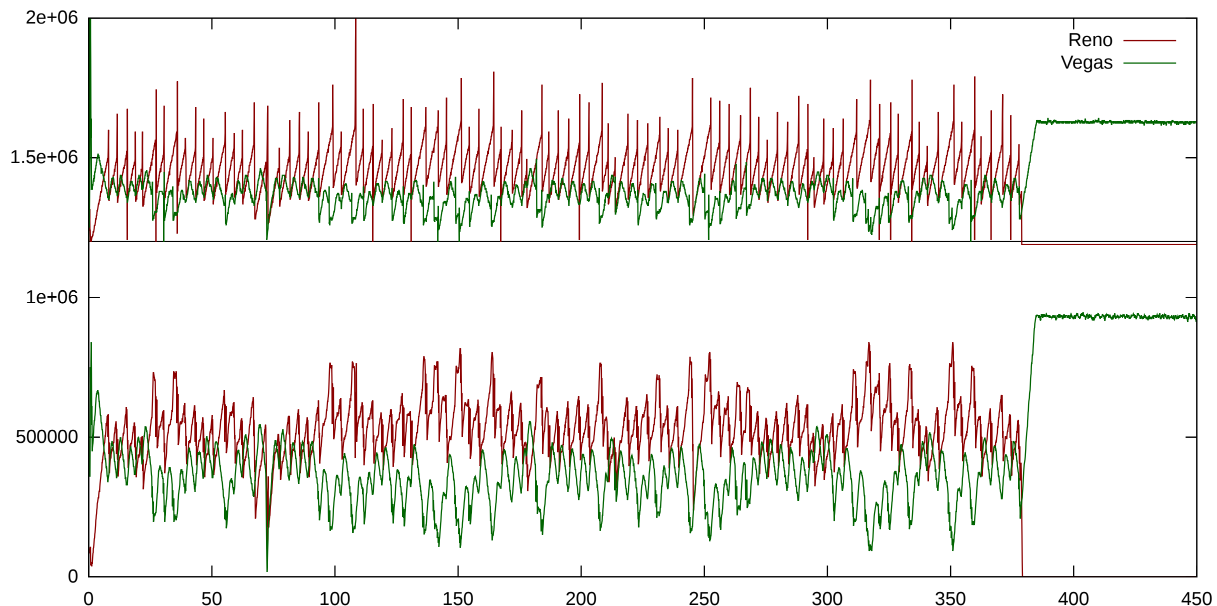
- On h3, start the `netcat -l ...` for the randomtelnet.py output (on two different ports)

- On h1 and h2, start the randomtelnet.py senders
- On h3, start dualreceive.py
- On h1 and h2, start the scripts (*eg* h1.sh and h2.sh) that wait for the signal and start sender.py
- On r, send the two start triggers via netcat

This is somewhat cumbersome; it helps to incorporate everything into a single shell script with ssh used to run subscripts on the appropriate host.

18.6.1.6 Reno vs Vegas results

In the Reno-Vegas graph at *16.5 TCP Reno versus TCP Vegas*, we set the Vegas parameters α and β to 3 and 6 respectively. The implementation of TCP Vegas on the Mininet virtual machine does not, however, support changing α and β, and the default values are more like 1 and 3. To give Vegas a fighting chance, we reduce the queue size at r to 10 in competition.py. Here is the graph, with the packets-in-flight monitoring above and the throughput below:



TCP Vegas is getting a smaller share of the bandwidth (overall about 40% to TCP Reno's 60%), but it is consistently holding its own. It turns out that TCP Vegas is greatly helped by the small queue size; if the queue size is doubled to 20, then Vegas gets a 17% share.

In the upper part of the graph, we can see the Reno sawteeth versus the Vegas triangular teeth (sloping down as well as sloping up); compare to the red-and-green graph at *16.5 TCP Reno versus TCP Vegas*. The tooth shapes are somewhat mirrored in the throughput graph as well, as throughput is proportional to queue utilization which is proportional to the number of packets in flight.

18.7 TCP Competition: Reno vs BBR

We can apply the same technique to compare TCP Reno to TCP BBR. This was done to create the graph at *15.16 TCP BBR*. The Mininet approach is usable as soon as a TCP BBR module for linux was released (in source form); to use a simulator, on the other hand, would entail waiting for TCP BBR to be ported to the simulator.

One nicety is that it is essential that the `fq` queuing discipline be enabled for the TCP BBR sender. If that is `h2`, for example, then the following Mininet code (perhaps in `competition.py`) removes any existing queuing discipline and adds `fq`:

```
h2.cmd('tc qdisc del dev h2-eth root')
h2.cmd('tc qdisc add dev h2-eth root fq')
```

The purpose of the `fq` queuing is to enable **pacing**; that is, the transmission of packets at regular, very small intervals.

18.8 Linux Traffic Control (tc)

The linux **tc** command, for traffic control, allows the attachment of any implemented queuing discipline (*19 Queuing and Scheduling*) to any network interface (usually of a router). A hierarchical example appears in *19.13.2 Linux htb*. The tc command is also used extensively by Mininet to control, for example, link queue capacities. An explicit example, of adding the `fq` queuing discipline, appears immediately above.

The two examples presented in this section involve "simple" token-bucket filtering, using **tbf**, and then "classful" token-bucket filtering, using **htb**. We will use the latter example to apply token-bucket filtering only to one class of connections; other connections receive no filtering.

The granularity of tc-tbf rate control is limited by the cpu-interrupt timer granularity; typically tbf is able schedules packets every 10 ms. If the transmission rate is 6 MB/s, or about four 1500-byte packets per millisecond, then tbf will schedule 40 packets for transmission every 10 ms. They will, however, most likely be sent as a burst at the start of the 10-ms interval. Some tc schedulers are able to achieve much finer *pacing* control; *eg* the 'fq' qdisc of *18.7 TCP Competition: Reno vs BBR* above.

The Mininet topology in both cases involves a single router between two hosts, h1—r—h2. We will here use the routerline.py example with the option `-N 1`; the router is then `r1` with interfaces `r1-eth0` connecting to `h1` and `r1-eth1` connecting to `h2`. The desired topology can also be built using competition.py and then ignoring the third host.

To send data we will use sender.py (*18.6.1.1 sender.py*), though with the default TCP congestion algorithm. To receive data we will use dualreceive.py, though initially with just one connection sending any significant data. We will set the constant `PRINT_CUMULATIVE` to `False`, so `dualreceive.py` prints at intervals the number of bytes received during the most recent interval; we will call this modified version `dualreceive_incr.py`. We will also redirect the `stderr` messages to `/dev/null`, and start this on `h2`:

```
python3 dualreceive_incr.py 2>/dev/null
```

We start the main sender on `h1` with the following, where `h2` has IPv4 address 10.0.1.10 and 1,000,000 is the number of blocks:

```
python3 sender.py 1000000 10.0.1.10 5430
```

The dualreceive program will not do any reading until both connections are enabled, so we also need to create a second connection from h1 in order to get started; this second connection sends only a single block of data:

```
python3 sender.py 1 10.0.1.10 5431
```

At this point dualreceive should generate output somewhat like the following (with timestamps in the first column rounded to the nearest millisecond). The byte-count numbers in the middle column are rather hardware-dependent

```
1.016   14079000   0
1.106   12702000   0
1.216   14724000   0
1.316   13666448   0
1.406   11877552   0
```

This means that, on average, h2 is receiving about 13 MB every 100ms, which is about 1.0 Gbps.

Now we run the command below on r1 to reduce the rate (tc requires the abbreviation mbit for megabit/sec; it treats mbps as MegaBytes per second). The token-bucket filter parameters are rate and burst. The purpose of the limit parameter – used by netem and several other qdiscs as well – is to specify the maximum queue size for the waiting packets. Its value here is not very significant, but too low a value can lead to packet loss and thus to momentarily plunging bandwidth. Too high a value, on the other hand, can lead to bufferbloat (*13.7.1 Bufferbloat*).

```
tc qdisc add dev r1-eth1 root tbf rate 40mbit burst 50kb limit 200kb
```

We get output something like this:

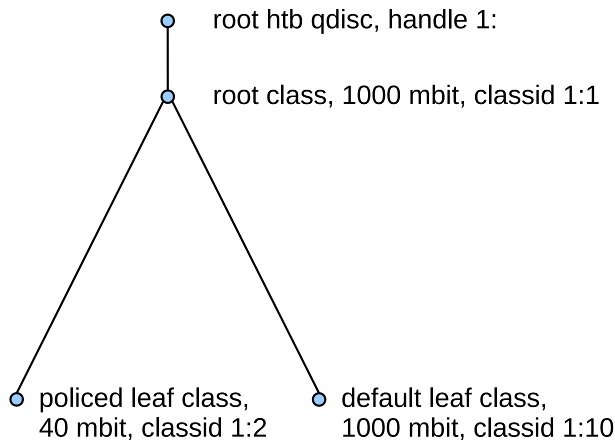
```
1.002   477840   0
1.102   477840   0
1.202   477840   0
1.302   482184   0
1.402   473496   0
```

477840 bytes per 100 ms is 38.2 Mbps. That is received application data; the extra 5% or so to 40 Mbps corresponds mostly to packet headers (66 bytes out of every 1514, though to see this with WireShark we need to disable TSO, *12.5 TCP Offloading*).

We can also change the rate dynamically:

```
tc qdisc change dev r1-eth1 root tbf rate 20mbit burst 100kb limit 200kb
```

The above use of tbf allows us to throttle (or police) *all* traffic through interface r1-eth1. Suppose we want to police selected traffic only? Then we can use *hierarchical* token bucket, or **htb**. We set up an htb *root* node, with no limits, and then create two child nodes, one for policed traffic and one for default traffic.



To create the htb hierarchy we will first create the root qdisc and associated root *class*. We need the raw interface rate, here taken to be 1000mbit. Class identifiers are of the form *major*:*minor*, where *major* is the integer root "handle" and *minor* is another integer.

```
tc qdisc add dev r1-eth1 root handle 1: htb default 10
tc class add dev r1-eth1 parent 1: classid 1:1 htb rate 1000mbit
```

We now create the two child classes (not qdiscs), one for the rate-limited traffic and one for default traffic. The rate-limited class has classid 1:2 here; the default class has classid 1:10.

```
tc class add dev r1-eth1 parent 1: classid 1:2 htb rate 40mbit
tc class add dev r1-eth1 parent 1: classid 1:10 htb rate 1000mbit
```

We still need a **classifier** (or **filter**) to assign selected traffic to class 1:2. Our goal is to police traffic to port 5430 (by default, dualreceive.py accepts traffic at ports 5430 and 5431).

There are several classifiers available; for example *u32* (man tc-u32) and *bpf* (man tc-bpf). The latter is based on the Berkeley Packet Filter virtual machine for packet recognition. However, what we use here – mainly because it seems to work most reliably – is the iptables *fwmark* mechanism, used earlier in *9.6 Routing on Other Attributes*. Iptables is intended for filtering – and sometimes modifying – packets; we can associate a `fwmark` value of 2 to packets bound for TCP port 5430 with the command below (the `fwmark` value does not become part of the packet; it exists only while the packet remains in the kernel).

```
iptables --append FORWARD --table mangle --protocol tcp --dport 5430 --jump MARK --set-mark
```

When this is run on `r1`, then packets forwarded by `r1` to TCP port 5430 receive the `fwmark` upon arrival.

The next step is to tell the `tc` subsystem that packets with a `fwmark` value of 2 are to be placed in class 1:2; this is the rate-limited class above. In the following command, `flowid` may be used as a synonym for `classid`.

```
tc filter add dev r1-eth1 parent 1:0 protocol ip handle 2 fw classid 1:2
```

We can view all these settings with

```
tc qdisc show dev r1-eth1
tc class show dev r1-eth1
```

```
tc filter show dev r1-eth1 parent 1:1
iptables --table mangle --list
```

We now verify that all this works. As with `tbf`, we start `dualreceive_incr.py` on `h2` and two senders on `h1`. This time, both senders send large amounts of data:

```
h2: python3 dualreceive_incr.py 2>/dev/null
h1: python3 sender.py 500000 10.0.1.10 5430
h1: python3 sender.py 500000 10.0.1.10 5431
```

If everything works, then shortly after the second sender starts we should see something like the output below (taken after both TCP connections have their `cwnd` stabilize). The middle column is the number of received data bytes to the policed port, 5430.

```
1.000   453224   10425600
1.100   457568   10230120
1.200   461912    9934728
1.300   476392   10655832
1.401   438744   10230120
```

With 66 bytes of TCP/IP headers in every 1514-byte packet, our requested 40 mbit data-rate cap should yield about 478,000 bytes every 0.1 sec. The slight reduction above appears to be related to TCP competition; the full 478,000-byte rate is achieved after the port-5431 connection terminates.

18.9 OpenFlow and the POX Controller

In this section we introduce the POX controller for OpenFlow (*2.7.1 OpenFlow Switches*) switches, allowing exploration of software-defined networking (*2.7 Software-Defined Networking*). In the `switchline.py` Ethernet-switch example from earlier, the `Mininet()` call included a parameter `controller=DefaultController`; this causes each switch to behave like an ordinary Ethernet learning switch. By using Pox to create customized controllers, we can investigate other options for switch operation. Pox is preinstalled on the Mininet virtual machine.

Pox is, like Mininet, written in Python2. It receives and sends OpenFlow messages, in response to **events**. Event-related messages, for our purposes here, can be grouped into the following categories:

- **PacketIn**: a switch is informing the controller about an arriving packet, usually because the switch does not know how to forward the packet or does not know how to forward the packet without flooding. Often, but not always, PacketIn events will result in the controller providing new forwarding instructions.
- **ConnectionUP**: a switch has connected to the controller. This will be the point at which the controller gives the switch its initial packet-handling instructions.
- **LinkEvent**: a switch is informing the controller of a link becoming available or becoming unavailable; this includes initial reports of link availability.
- **BarrierEvent**: a switch's response to an OpenFlow Barrier message, meaning the switch has completed its responses to all messages received before the Barrier and now may begin to respond to messages received after the Barrier.

The Pox program comes with several demonstration modules illustrating how controllers can be programmed; these are in the pox/misc and pox/forwarding directories. The starting point for Pox documentation is the Pox wiki (archived copy at poxwiki.pdf), which among other thing includes brief outlines of these programs. We now review a few of these programs; most were written by James McCauley and are licensed under the Apache license.

The Pox code data structures are very closely tied to the OpenFlow Switch Specification, versions of which can be found at the OpenNetworking.org technical library.

18.9.1 `hub.py`

As a first example of Pox, suppose we take a copy of the switchline.py file and make the following changes:

- change the controller specification, inside the `Mininet()` call, from `controller=DefaultController` to `controller=RemoteController`.
- add the following lines immediately following the `Mininet()` call:

```
c = RemoteController( 'c', ip='127.0.0.1', port=6633 )
net.addController(c)
```

This modified version is available as switchline_rc.py, "rc" for remote controller. If we now run this modified version, then pings fail because the RemoteController, c, does not yet exist; in the absence of a controller, the switches' default response is to do nothing.

We now start Pox, in the directory /home/mininet/pox, as follows; this loads the file pox/forwarding/hub.py

```
./pox.py forwarding.hub
```

Ping connectivity should be restored! The switch connects to the controller at IPv4 address 127.0.0.1 (more on this below) and TCP port 6633. At this point the controller is able to tell the switch what to do.

The `hub.py` example configures each switch as a simple hub, flooding each arriving packet out all other interfaces (though for the linear topology of `switchline_rc.py`, this doesn't matter much). The relevant code is here:

```
def _handle_ConnectionUp (event):
    msg = of.ofp_flow_mod()
    msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
    event.connection.send(msg)
```

This is the *handler* for ConnectionUp events; it is invoked when a switch first reports for duty. As each switch connects to the controller, the `hub.py` code instructs the switch to forward each arriving packet to the virtual port `OFPP_FLOOD`, which means to forward out all other ports.

The `event` parameter is of class `ConnectionUp`, a subclass of class `Event`. It is defined in `pox/openflow/__init__.py`. Most switch-event objects throughout Pox include a `connection` field, which the controller can use to send messages back to the switch, and a `dpid` field, representing the switch identification number. Generally the Mininet switch `s1` will have a `dpid` of 1, *etc*.

The code above creates an OpenFlow *modify-flow-table* message, `msg`; this is one of several types of controller-to-switch messages that are defined in the OpenFlow standard. The field `msg.actions` is a

list of actions to be taken; to this list we append the action of forwarding on the designated (virtual) port OFPP_FLOOD.

Normally we would also append to the list msg.match the matching rules for the packets to be forwarded, but here we want to forward all packets and so no matching is needed.

A different – though functionally equivalent – approach is taken in pox/misc/of_tutorial.py. Here, the response to the ConnectionUp event involves no communication with the switch (though the connection is stored in Tutorial.__init__()). Instead, as the switch reports each arriving packet to the controller, the controller responds by telling the switch to flood the packet out every port (this approach does result in sufficient unnecessary traffic that it would not be used in production code). The code (slightly consolidated) looks something like this:

```
def _handle_PacketIn (self, event):
    packet = event.parsed # This is the parsed packet data.
    packet_in = event.ofp # The actual ofp_packet_in message.
    self.act_like_hub(packet, packet_in)

def act_like_hub (self, packet, packet_in):
    msg = of.ofp_packet_out()
    msg.data = packet_in
    action = of.ofp_action_output(port = of.OFPP_ALL)
    msg.actions.append(action)
    self.connection.send(msg)
```

The event here is now an instance of class PacketIn. This time the switch sents a *packet out* message to the switch. The packet and packet_in objects are two different views of the packet; the first is parsed and so is generally easier to obtain information from, while the second represents the entire packet as it was received by the switch. It is the latter format that is sent back to the switch in the msg.data field. The virtual port OFPP_ALL is equivalent to OFPP_FLOOD.

For either hub implementation, if we start WireShark on h2 and then ping from h4 to h1, we will see the pings at h2. This demonstrates, for example, that s2 is behaving like a hub rather than a switch.

18.9.2 l2_pairs.py

The next Pox example, l2_pairs.py, implements a real Ethernet learning switch. This is the pairs-based switch implementation discussed in *2.7.2 Learning Switches in OpenFlow*. This module acts at the Ethernet address layer (layer 2, the *l2* part of the name), and flows are specified by (src,dst) *pairs* of addresses. The l2_pairs.py module is started with the Pox command ./pox.py forwarding.l2_pairs.

A straightforward implementation of an Ethernet learning switch runs into a problem: the switch needs to contact the controller whenever the packet *source* address has not been seen before, so the controller can send back to the switch the forwarding rule for how to reach that source address. But the primary lookup in the switch flow table must be by *destination* address. The approach used here uses a single OpenFlow table, versus the two-table mechanism of *18.9.3 l2_nx.py*. However, the learned flow table match entries will all include match rules for both the source and the destination address of the packet, so that a separate entry is necessary for each pair of communicating hosts. The number of flow entries thus scales as $O(N^2)$, which presents a scaling problem for very large switches but which we will ignore here.

When a switch sees a packet with an unmatched (dst,src) address pair, it forwards it to the controller, which has two cases to consider:

- If the controller does not know how to reach the destination address from the current switch, it tells the switch to flood the packet. However, the controller also *records*, for later reference, the packet source address and its arrival interface.
- If the controller knows that the destination address can be reached from this switch via switch port dst_port, it sends to the switch instructions to create a forwarding entry for (dst,src)→dst_port. At the same time, the controller also sends to the switch a reverse forwarding entry for (src,dst), forwarding via the port by which the packet arrived.

The controller maintains its partial map from addresses to switch ports in a dictionary `table`, which takes a (switch,destination) pair as its key and which returns switch port numbers as values. The switch is represented by the `event.connection` object used to reach the switch, and destination addresses are represented as Pox EthAddr objects.

The program handles only PacketIn events. The main steps of the PacketIn handler are as follows. First, when a packet arrives, we put its switch and source into `table`:

```
table[(event.connection,packet.src)] = event.port
```

The next step is to check to see if there is an entry in `table` for the destination, by looking up `table[(event.connection,packet.dst)]`. If there is *not* an entry, then the packet gets flooded by the same mechanism as in `of_tutorial.py` above: we create a packet-out message containing the to-be-flooded packet and send it back to the switch.

If, on the other hand, the controller finds that the destination address can be reached via switch port `dst_port`, it proceeds as follows. We first create the *reverse* entry; `event.port` is the port by which the packet just arrived:

```
msg = of.ofp_flow_mod()
msg.match.dl_dst = packet.src       # reversed dst and src
msg.match.dl_src = packet.dst       # reversed dst and src
msg.actions.append(of.ofp_action_output(port = event.port))
event.connection.send(msg)
```

This is like the forwarding rule created in `hub.py`, except that we here are forwarding via the specific port `event.port` rather than the virtual port `OFPP_FLOOD`, and, perhaps more importantly, we are adding two packet-matching rules to `msg.match`.

The next step is to create a similar matching rule for the src-to-dst flow, *and* to include the packet to be retransmitted. The modify-flow-table message thus does double duty as a packet-out message as well.

```
msg = of.ofp_flow_mod()
msg.data = event.ofp                # Forward the incoming packet
msg.match.dl_src = packet.src       # not reversed this time!
msg.match.dl_dst = packet.dst
msg.actions.append(of.ofp_action_output(port = dst_port))
event.connection.send(msg)
```

The `msg.match` object has quite a few potential matching fields; the following is taken from the Pox-Wiki:

Attribute	Meaning
in_port	Switch port number the packet arrived on
dl_src	Ethernet source address
dl_dst	Ethernet destination address
dl_type	Ethertype / length (e.g. 0x0800 = IPv4)
nw_tos	IPv4 TOS/DS bits
nw_proto	IPv4 protocol (e.g., 6 = TCP), or lower 8 bits of ARP opcode
nw_src	IPv4 source address
nw_dst	IP destination address
tp_src	TCP/UDP source port
tp_dst	TCP/UDP destination port

It is also possible to create a `msg.match` object that matches all fields of a given packet.

We can watch the forwarding entries created by `l2_pairs.py` with the linux program ovs-ofctl. Suppose we start `switchline_rc.py` and then the Pox module `l2_pairs.py`. Next, from within Mininet, we have h1 ping h4 and h2 ping h4. If we now run the command (on the Mininet virtual machine but from a linux prompt)

```
ovs-ofctl dump-flows s2
```

we get

> cookie=0x0, ...,**dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:04** actions=output:3
>
> cookie=0x0, ...,**dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:02** actions=output:1
>
> cookie=0x0, ...,**dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:04** actions=output:3
>
> cookie=0x0, ...,**dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:01** actions=output:2

Because we used the `autoSetMacs=True` option in the `Mininet()` call in `switchline_rc.py`, the Ethernet addresses assigned to hosts are easy to follow: h1 is 00:00:00:00:00:01, *etc*. The first and fourth lines above result from h1 pinging h4; we can see from the output port at the end of each line that s1 must be reachable from s2 via port 2 and s3 via port 3. Similarly, the middle two lines result from h2 pinging h4; h2 lies off s2's port 1. These port numbers correspond to the interface numbers shown in the diagram at *18.3 Multiple Switches in a Line*.

18.9.3 `l2_nx.py`

The `l2_nx.py` example accomplishes the same Ethernet-switch effect as `l2_pairs.py`, but using only O(N) space. It does, however, use two OpenFlow tables, one for destination addresses and one for source addresses. In the implementation here, source addresses are held in table 0, while destination addresses are held in table 1; this is the reverse of the multiple-table approach outlined in *2.7.2 Learning Switches in OpenFlow*. The *l2* again refers to network layer 2, and the *nx* refers to the so-called Nicira extensions to Pox, which enable the use of multiple flow tables.

Initially, table 0 is set up so that it tries a match on the source address. If there is no match, the packet is forwarded to the controller, *and* sent on to table 1. If there is a match, the packet is sent on to table 1 but not to the controller.

Table 1 then looks for a match on the destination address. If one is found then the packet is forwarded to the destination, and if there is no match then the packet is flooded.

Using two OpenFlow tables in Pox requires the loading of the so-called Nicira extensions (hence the "nx" in the module name here). These require a slightly more complex command line:

```
./pox.py openflow.nicira --convert-packet-in forwarding.l2_nx
```

Nicira will also require, *eg*, `nx.nx_flow_mod()` instead of `of.ofp_flow_mod()`.

The no-match actions for each table are set during the handling of the ConnectionUp events. An action becomes the default action when no `msg.match()` rules are included, and the priority is low; recall (*2.7.1 OpenFlow Switches*) that if a packet matches multiple flow-table entries then the entry with the highest priority wins. The priority is here set to 1; the Pox default priority – which will be used (implicitly) for later, more-specific flow-table entries – is 32768. The first step is to arrange for table 0 to forward to the controller and to table 1.

```
msg = nx.nx_flow_mod()
msg.table_id = 0              # not necessary as this is the default
msg.priority = 1              # low priority
msg.actions.append(of.ofp_action_output(port = of.OFPP_CONTROLLER))
msg.actions.append(nx.nx_action_resubmit.resubmit_table(table = 1))
event.connection.send(msg)
```

Next we tell table 1 to flood packets by default:

> msg = nx.nx_flow_mod() msg.table_id = 1 msg.priority = 1 msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD)) event.connection.send(msg)

Now we define the PacketIn handler. First comes the table 0 match on the packet source; if there is a match, then the source address has been seen by the controller, and so the packet is no longer forwarded to the controller (it is forwarded to table 1 *only*).

```
msg = nx.nx_flow_mod()
msg.table_id = 0
msg.match.of_eth_src = packet.src      # match the source
msg.actions.append(nx.nx_action_resubmit.resubmit_table(table = 1))
event.connection.send(msg)
```

Now comes table 1, where we match on the destination address. All we know at this point is that the packet with source address `packet.src` came from port `event.port`, and we forward any packets addressed *to* `packet.src` via that port:

> msg = nx.nx_flow_mod() msg.table_id = 1 msg.match.of_eth_dst = packet.src # this rule applies only for packets *to* packet.src msg.actions.append(of.ofp_action_output(port = event.port)) event.connection.send(msg)

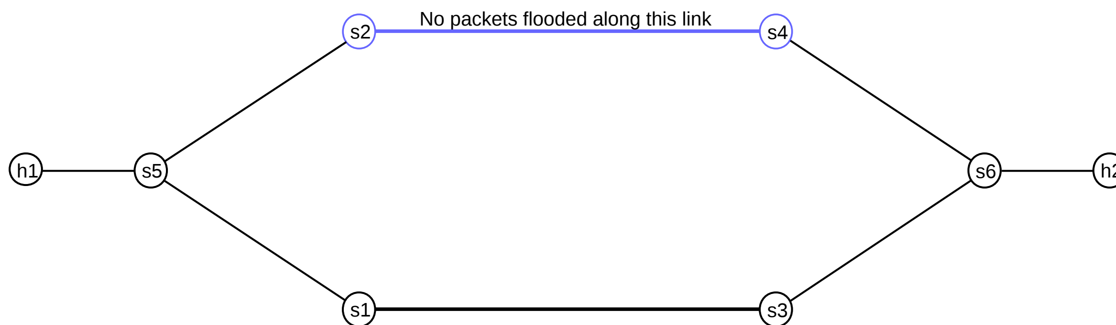
Note that there is no network state maintained at the controller; there is no analog here of the `table` dictionary of `l2_pairs.py`.

Suppose we have a simple network h1–s1–h2. When h1 sends to h2, the controller will add to s1's table 0 an entry indicating that h1 is a known source address. It will also add to s1's table 1 an entry indicating that h1 is reachable via the port on s1's left. Similarly, when h2 replies, s1 will have h2 added to its table 0, and then to its table 1.

18.9.4 `multitrunk.py`

The goal of the **multitrunk** example is to illustrate how different TCP connections between two hosts can be routed via different paths; in this case, via different "trunk lines". This example and the next are not part of the standard distributions of either Mininet or Pox. Unlike the other examples discussed here, these examples consist of Mininet code to set up a specific network topology and a corresponding Pox controller module that is written to work properly only with that topology. Most real networks evolve with time, making a tight link between topology and controller impractical (though this may sometimes work well in datacenters). The purpose here, however, is to illustrate specific OpenFlow possibilities in a (relatively) simple setting.

The multitrunk topology involves multiple "trunk lines" between host `h1` and `h2`, as in the following diagram; the trunk lines are the `s1–s3` and `s2–s4` links.



Multitrunk topology, N=1, K=2

The Mininet file is multitrunk12.py and the corresponding Pox module is multitrunkpox.py. The number of trunk lines is K=2 by default, but can be changed by setting the variable K. We will prevent looping of broadcast traffic by never flooding along the `s2–s4` link.

TCP traffic takes either the `s1–s3` trunk or the `s2–s4` trunk. We will refer to the two directions `h1→h2` and `h2→h1` of a TCP connection as **flows**, consistent with the usage in *8.1 The IPv6 Header*. Only `h1→h2` flows will have their routing vary; flows `h2→h1` will always take the `s1–s3` path. It does not matter if the original connection is opened from `h1` to `h2` or from `h2` to `h1`.

The first TCP flow from `h1` to `h2` goes via `s1–s3`. After that, subsequent connections alternate in round-robin fashion between `s1–s3` and `s2–s4`. To achieve this we must, of course, include TCP ports in the OpenFlow forwarding information.

All links will have a bandwidth set in Mininet. This involves using the `link=TCLink` option; TC here stands for Traffic Control. We do not otherwise make use of the bandwidth limits. TCLinks can also have a queue size set, as in *18.6 TCP Competition: Reno vs Vegas*.

For ARP and ICMP traffic, two OpenFlow tables are used as in *18.9.3 l2_nx.py*. The `PacketIn` messages for ARP and ICMP packets are how switches learn of the MAC addresses of hosts, and also how the controller learns which switch ports are directly connected to hosts. TCP traffic is handled differently, below.

During the initial handling of `ConnectionUp` messages, switches receive their default packet-handling instructions for ARP and ICMP packets, and a `SwitchNode` object is created in the controller for each

switch. These objects will eventually contain information about what neighbor switch or host is reached by each switch port, but at this point none of that information is yet available.

The next step is the handling of `LinkEvent` messages, which are initiated by the `discovery` module. This module must be included on the `./pox.py` command line in order for this example to work. The `discovery` module sends each switch, as it connects to the controller, a special discovery packet in the Link Layer Discovery Protocol (LLDP) format; this packet includes the originating switch's `dpid` value and the switch port by which the originating switch sent the packet. When an LLDP packet is received by the neighboring switch, that switch forwards it back to the controller, together with the dpid and port for the receiving switch. At this point the controller knows the switches and port numbers at each end of the link. The controller then reports this to our `multitrunkpox` module via a `LinkEvent` event.

As `LinkEvent` messages are processed, the `multitrunkpox` module learns, for each switch, which ports connect directly to neighboring switches. At the end of the `LinkEvent` phase, which generally takes several seconds, each switch's `SwitchNode` knows about all directly connected neighbor switches. Nothing is yet known about directly connected neighbor hosts though, as hosts have not yet sent any packets.

Once hosts `h1` and `h2` exchange a pair of packets, the associated `PacketIn` events tell `multitrunkpox` what switch ports are connected to hosts. Ethernet address learning also takes place. If we execute `h1 ping h2`, for example, then afterwards the information contained in the `SwitchNode` graph is complete.

Now suppose `h1` tries to open a TCP connection to `h2`, *eg* via ssh. The first packet is a TCP SYN packet. The switch `s5` will see this packet and forward it to the controller, where the `PacketIn` handler will process it. We create a flow for the packet,

```
flow = Flow(psrc, pdst, ipv4.srcip, ipv4.dstip, tcp.srcport, tcp.dstport)
```

and then see if a path has already been assigned to this flow in the dictionary `flow_to_path`. For the very first packet this will never be the case. If no path exists, we create one, first picking a trunk:

```
trunkswitch = picktrunk(flow)
path = findpath(flow, trunkswitch)
```

The first path will be the Python list [h1, s5, s1, s3, s6, h2], where the switches are represented by `SwitchNode` objects.

The supposedly final step is to call

```
result = create_path_entries(flow, path)
```

to create the forwarding rules for each switch. With the path as above, the `SwitchNode` objects know what port `s5` should use to reach `s1`, *etc*. Because the first TCP SYN packet *must* have been preceeded by an ARP exchange, and because the ARP exchange *will* result in `s6` learning what port to use to reach `h2`, this *should* work.

But in fact it does not, at least not always. The problem is that Pox creates separate internal threads for the ARP-packet handling and the TCP-packet handling, and the former thread may not yet have installed the location of `h2` into the appropriate `SwitchNode` object by the time the latter thread calls `create_path_entries()` and needs the location of `h2`. This race condition is unfortunate, but cannot be avoided. As a fallback, if creating a path fails, we flood the TCP packet along the `s1–s3` link (even if the chosen trunk is the `s2–s4` link) and wait for the next TCP packet to try again. Very soon, `s6` *will* know how to reach `h2`, and so `create_path_entries()` will succeed.

If we run everything, create two xterms on `h1`, and then create two ssh connections to `h2`, we can see the forwarding entries using `ovs-ofctl`. Let us run

```
ovs-ofctl dump-flows s5
```

Restricting attention only to those flow entries with foo=tcp, we get (with a little sorting)

> cookie=0x0, ..., tcp,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src=10.0.0.1,nw_dst=10.0.0.2,**tp_src=59404**,tp_dst=22 **actions=output:1**
>
> cookie=0x0, ..., tcp,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src=10.0.0.1,nw_dst=10.0.0.2,**tp_src=59526**,tp_dst=22 **actions=output:2**
>
> cookie=0x0, ..., tcp,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01,nw_src=10.0.0.2,nw_dst=10.0.0.1,tp_src=22,tp_dst=59404 actions=output:3
>
> cookie=0x0, ..., tcp,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01,nw_src=10.0.0.2,nw_dst=10.0.0.1,tp_src=22,tp_dst=59526 actions=output:3

The first two entries represent the `h1`→`h2` flows. The first connection has source TCP port 59404 and is routed via the `s1–s3` trunk; we can see that the output port from `s5` is port 1, which is indeed the port that s5 uses to reach s1 (the output of the Mininet `links` command includes `s5-eth1<->s1-eth2`). Similarly, the output port used at `s5` by the second connection, with source TCP port 59526, is 2, which is the port `s5` uses to reach `s2`. The switch `s5` reaches host `h1` via port 3, which can be seen in the last two entries above, which correspond to the reverse `h2`→`h1` flows.

The OpenFlow timeout here is infinite. This is not a good idea if the system is to be running indefinitely, with a steady stream of short-term TCP connections. It does, however, make it easier to view connections with `ovs-ofctl` before they disappear. A production implementation would need a finite timeout, and then would have to ensure that connections that were idle for longer than the timeout interval were properly re-established when they resumed sending.

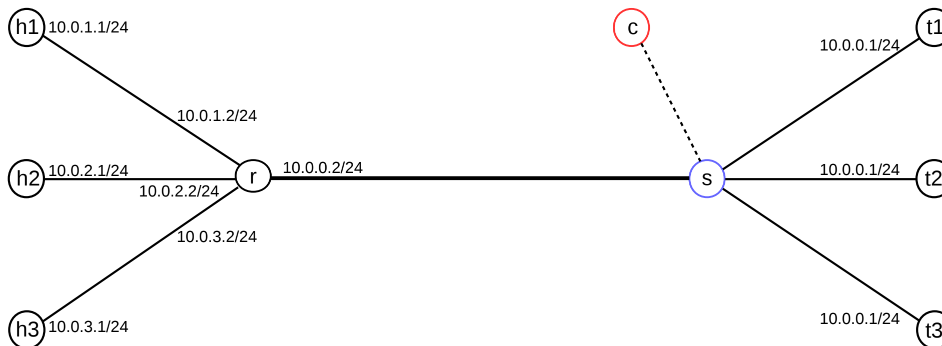
Although one prospective application of the multitrunk strategy is to have the `h1`→`h2` traffic share the trunk links in order to increase the effective `h1`→`h2` bandwidth, this is not the best way to achieve that. A better approach is to route the packets over the different trunks on a packet-by-packet basis rather than a connection-by-connection basis; that is, packet 1 is routed via `s1`, packet 2 via `s2`, packet 3 via `s1` again, and so on. One higher-layer protocol for doing this is equal-cost multipath routing, or ECMP.

OpenFlow has low-level support for this approach in the **select group** mechanism. A flow-table traffic-matching entry can forward traffic to a so-called *group* instead of out via a port. The action of a *select* group is then to select one of a set of output actions (often on a round-robin basis) and apply that action to the packet. In principle, we could implement this at `s5` to have successive packets sent to either `s1` or `s2` in round-robin fashion. In practice, Pox support for select groups appears to be insufficiently developed at the time of this writing (2017) to make this practical.

18.9.5 `loadbalance31.py`

The next example demonstrates a simple **load balancer**. The topology is somewhat the reverse of the previous example: there are now three hosts (N=3) at each end, and only one trunk line (K=1) (there are

also no left- and right-hand entry/exit switches). The right-hand hosts act as the "servers", and are renamed `t1`, `t2` and `t3`.



The servers all get the *same* IPv4 address, 10.0.0.1. This would normally lead to chaos, but the servers are not allowed to talk to one another, and the controller ensures that the servers are not even aware of one another. In particular, the controller makes sure that the servers never all simultaneously reply to an ARP "who-has 10.0.0.1" query from `r`.

The Mininet file is loadbalance31.py and the corresponding Pox module is loadbalancepox.py.

The node `r` is a router, not a switch, and so its four interfaces are assigned to separate subnets. Each host is on its own subnet, which it shares with `r`. The router `r` then connects to the only switch, `s`; the connection from `s` to the controller `c` is shown.

The idea is that each TCP connection from any of the hi to 10.0.0.1 is connected, via `s`, to one of the servers `ti`, but different connections will connect to different servers. In this implementation the server choice is round-robin, so the first three TCP connections will connect to `t1`, `t2` and `t3` respectively, and the fourth will connect again to `t1`.

The servers `t1` through `t3` are configured to all have the same IPv4 address 10.0.0.1; there is no address rewriting done to packets arriving from the left. However, as in the preceding example, when the first packet of each new TCP connection from left to right arrives at `s`, it is forwarded to `c` which then selects a specific `ti` and creates in `s` the appropriate forwarding rule for that connection. As in the previous example, each TCP connection involves two Flow objects, one in each direction, and separate OpenFlow forwarding entries are created for each flow.

There is no need for paths; the main work of routing the TCP connections looks like this:

```
server = pickserver(flow)
flow_to_server[flow] = server
addTCPrule(event.connection, flow, server+1)          # ti is at port i+1
addTCPrule(event.connection, flow.reverse(), 1)       # port 1 leads to r
```

The biggest technical problem is ARP: normally, `r` and the `ti` would contact one another via ARP to find the appropriate LAN addresses, but that will not end well with identical IPv4 addresses. So instead we create "static" ARP entries. We know (by checking) that the MAC address of `r-eth0` is 00:00:00:00:00:04, and so the Mininet file runs the following command on each of the `ti`:

```
arp -s 10.0.0.2 00:00:00:00:00:04
```

This creates a static ARP entry on each of the `ti`, which leaves them knowing the MAC address for their default router 10.0.0.2. As a result, none of them issues an ARP query to find `r`. The other direction is similar, except that `r` (which is not really in on the load-balancing plot) must think 10.0.0.1 has a single MAC address. Therefore, we give each of the `ti` the same MAC address (which would normally lead to even more chaos than giving them all the same IPv4 address); that address is 00:00:00:00:01:ff. We then install a permanent ARP entry on `r` with

```
arp -s 10.0.0.1 00:00:00:00:01:ff
```

Now, when `h1`, say, sends a TCP packet to 10.0.0.1, `r` forwards it to MAC address 00:00:00:00:01:ff, and then `s` forwards it to whichever of `t1`..`t3` it has been instructed by the controller `c` to forward it to. The packet arrives at `ti` with the correct IPv4 address (10.0.0.1) and correct MAC address (00:00:00:00:01:ff), and so is accepted. Replies are similar: `ti` sends to `r` at MAC address 00:00:00:00:00:04.

As part of the `ConnectionUp` processing, we set up rules so that ICMP packets from the left are always routed to `t1`. This way we have a single responder to ping requests. It is entirely possible that some important ICMP message – *eg* `Fragmentation required but DF flag set` – will be lost as a result.

If we run the programs and create xterm windows for h1, h2 and h3 and, from each, connect to 10.0.0.1 via ssh, we can tell that we've reached `t1`, `t2` or `t3` respectively by running `ifconfig`. The Ethernet interface on `t1` is named **t1**-eth0, and similarly for `t2` and `t3`. (Finding another way to distinguish the `ti` is not easy.) An even simpler way to see the connection rotation is to run `h1 ssh 10.0.0.1 ifconfig` at the `mininet>` prompt several times in succession, and note the successive interface names.

If we create three connections and then run `ovs-ofctl dump-flows s` and look at tcp entries with destination address 10.0.0.1, we get this:

> cookie=0x0, ...,
> tcp,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:01:ff,**nw_src=10.0.1.1**,nw_dst=10.0.0.1,tp_src=35110,tp_dst=22
> actions=**output:2**
>
> cookie=0x0, ...,
> tcp,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:01:ff,**nw_src=10.0.2.1**,nw_dst=10.0.0.1,tp_src=44014,tp_dst=22
> actions=**output:3**
>
> cookie=0x0, ...,
> tcp,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:01:ff,**nw_src=10.0.3.1**,nw_dst=10.0.0.1,tp_src=55598,tp_dst=22
> actions=**output:4**

The three different flows take output ports 2, 3 and 4 on `s`, corresponding to `t1`, `t2` and `t3`.

18.9.6 `l2_multi.py`

This final Pox controller example takes an arbitrary Mininet network, learns the topology, and then sets up OpenFlow rules so that all traffic is forwarded by the shortest path, as measured by hopcount. OpenFlow packet-forwarding rules are set up on demand, when traffic between two hosts is first seen.

This module is compatible with topologies with loops, provided the spanning_tree module is also loaded.

We start with the spanning_tree module. This uses the `openflow.discovery` module, as in *18.9.4 multitrunk.py*, to build a map of all the connections, and then runs the spanning-tree algorithm of *2.5 Spanning Tree Algorithm and Redundancy*. The result is a list of switch ports on which flooding should not occur; flooding is then disabled by setting the OpenFlow `NO_FLOOD` attribute on these ports. We can see the ports of a switch `s` that have been disabled via `NO_FLOOD` by using `ovs-ofctl show s`.

One nicety is that the spanning_tree module is never quite certain when the network is complete. Therefore, it recalculates the spanning tree after every LinkEvent.

We can see the spanning_tree module in action if we create a Mininet network of four switches in a loop, as in exercise 9.0 below, and then run the following:

```
./pox.py openflow.discovery openflow.spanning_tree forwarding.l2_pairs
```

If we run `ovs-ofctl show` for each switch, we get something like the following:

> s1: (s1-eth2): ... NO_FLOOD
> s2: (s2-eth2): ... NO_FLOOD

We can verify with the Mininet `links` command that s1-eth2 and s2-eth2 are connected interfaces. We can verify with `tcpdump -i s1-eth2` that no packets are endlessly circulating.

We can also verify, with `ovs-ofctl dump-flows`, that the `s1–s2` link is not used at all, not even for `s1–s2` traffic. This is not surprising; the `l2_pairs` learning strategy learns ultimately learns source addresses from flooded ARP packets, which are not sent along the `s1–s2` link. If `s1` hears nothing from `s2`, it will never learn to send anything to `s2`.

The `l2_multi` module, on the other hand, creates a full map of all network links (separate from the map created by the spanning_tree module), and then calculates the best route between each pair of hosts. To calculate the routes, `l2_multi` uses the **Floyd-Warshall** algorithm (outlined below), which is a form of the distance-vector algorithm optimized for when a full network map is available. (The shortest-path algorithm of *9.5.1 Shortest-Path-First Algorithm* might be a faster choice.) To avoid having to rebuild the forwarding map on each LinkEvent, `l2_multi` does not create any routes until it sees the first packet (not counting LLDP packets). By that point, usually the network is stable.

If we run the example above using the Mininet rectangle topology, we again find that the spanning tree has disabled flooding on the `s1–s2` link. However, if we have `h1` ping `h2`, we see that `h1→h2` traffic does take the `s1–s2` link. Here is part of the result of `ovs-ofctl dump-flows s1`:

> cookie=0x0, ..., priority=65535,icmp,in_port=1,...,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src=10.0.0.1,nw_dst=10.0.0.2,
> actions=**output:2**
>
> cookie=0x0, ..., priority=65535,icmp,in_port=1,...0,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src=10.0.0.1,nw_dst=10.0.0.2
> actions=**output:2**

Note that `l2_multi` creates separate flow-table rules not only for ARP and ICMP, but also for ping (icmp_type=8) and ping reply (icmp_type=0). Such fine-grained matching rules are a matter of preference.

Here is a brief outline of the Floyd-Warshall algorithm. We assume that the switches are numbered {1,...,N}. The outer loop has the form `for k<=N:`; at the start of stage k, we assume that we've found the best path between every i and j for which every intermediate switch on the path is less than k. For many (i,j) pairs, there may be no such path.

At stage k, we examine, with an inner loop, all pairs (i,j). We look to see if there is a path from i to k and a second path from k to j. If there is, we concatenate the i-to-k and k-to-j paths to create a new i-to-j path, which we will call P. If there was no previous i-to-j path, then we add P. If there was a previous i-to-j path Q that is *longer* than P, we replace Q with P. At the end of the k=N stage, all paths have been discovered.

18.10 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercise 2.5 is distinct, for example, from exercises 2.0 and 3.0. Exercises marked with a ♢ have solutions or hints at 24.13 Solutions for Mininet.

1.0. In the RIP implementation of *18.5 IP Routers With Simple Distance-Vector Implementation*, add Split Horizon (*9.2.1.1 Split Horizon*).

2.0. In the RIP implementation of *18.5 IP Routers With Simple Distance-Vector Implementation*, add support for link failures (the third rule of *9.1.1 Distance-Vector Update Rules*)

3.0. Explain why, in the example of *18.9.3 l2_nx.py*, table 0 and table 1 will always have the same entries.

4.0. Suppose we try to eliminate the source addresses from the l2_pairs implementation.

- by default, all switches report all packets to the controller, and the controller then tells the switch to flood the packet.
- if a packet from ha to hb arrives at switch S, and S reports the packet to the controller, and the controller knows how to reach hb from S, then the controller installs forwarding rules into S for destination hb. The controller then tells S to re-forward the packet. In the future, S will not report packets to hb to the controller.
- when S reports to the controller a packet from ha to hb, then the controller notes that ha is reachable via the port on S by which the packet arrived.

Why does this not work? Hint: consider the switchline example (*18.3 Multiple Switches in a Line*), with h1 sending to h4, h4 sending to h1, h3 sending to h1, and finally h1 sending to h3.

5.0. Suppose we make the following change to the above strategy:

- if a packet from ha to hb arrives at switch S, and S reports the packet to the controller, and the controller knows how to reach *both* ha and hb from S, then the controller installs forwarding rules into S for destinations ha and hb. The controller then tells S to re-forward the packet. In the future, S will not report packets to ha or hb to the controller.

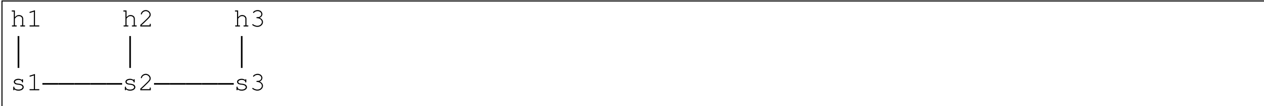
Show that this still does not work for the switchline example.

6.0. Suppose we try to implement an Ethernet switch as follows:

- the default switch action for an unmatched packet is to flood it and send it to the controller.
- if a packet from ha to hb arrives at switch S, and S reports the packet to the controller, and the controller knows how to reach both ha and hb from S, then the controller installs forwarding rules into S for destinations ha and hb. In the future, S will not report packets with these destinations to the controller.

- Unlike in exercise 4.0, the controller then tells S to *flood* the packet from ha to hb, even though it could be forwarded directly.

Traffic is sent in the network below:



(a)◇. Show that, if the traffic is as follows: h1 pings h2, **h3 pings h1**, then all three switches learn where h3 is.

(b). Show that, if the traffic is as follows: h1 pings h2, **h1 pings h3**, then none of the switches learn where h3 is.

Recall that each ping for a new destination starts with a **broadcast** ARP. Broadcast packets are always sent to the controller, as there is no destination match.

7.0. In *18.9.5 loadbalance31.py*, we could have configured the `ti` to have default router 10.0.0.3, say, and then created the appropriate static ARP entry for 10.0.0.3:

```
ip route add to default via 10.0.0.3 dev ti-eth0
arp -s 10.0.0.3 00:00:00:00:00:04
```

Everything still works, even though the `ti` think their router is at 10.0.0.3 and it is actually at 10.0.0.2. Explain why. (Hint: how is the router IPv4 address actually used by the `ti`?)

8.0. As discussed in the text, a race condition can arise in the example of *18.9.4 multitrunk.py*, where at the time the first TCP packet the controller still does not know where `h2` is, even though it should learn that after processing the first ARP packet.

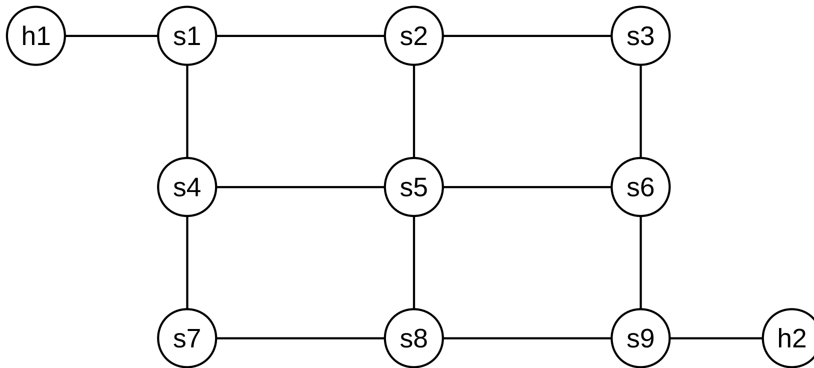
Explain why a similar race condition cannot occur in *18.9.5 loadbalance31.py*.

9.0. Create a Mininet network with four hosts and four switches as below:

The switches should use an external controller. Now let Pox be that controller, with

```
./pox.py openflow.discovery openflow.spanning_tree l2_pairs.py
```

10.0. Create the topology below with Mininet. Run the `l2_multi` Pox module as controller, with the `openflow.spanning_tree` option, and identify the spanning tree created. Also identify the path taken by icmp traffic from `h1` to `h2`.


h1
s1
s2
s3
s4
s5
s6
s7
s8
s9
h2

Is giving all control of congestion to the TCP layer really the only option? As the Internet has evolved, so have situations in which we may not want routers handling all traffic on a first-come, first-served basis. Traffic with delay bounds – so-called **real-time** traffic, often involving either voice or video – is likely to perform much better with preferential service, for example; we will turn to this in *20 Quality of Service*. But even without real-time traffic, we might be interested in guaranteeing that each of several customers gets an agreed-upon fraction of bandwidth, regardless of what the other customers are receiving. If we trust only to TCP Reno's bandwidth-allocation mechanisms, and if one customer has one connection and another has ten, then the bandwidth received may also be in the ratio of 1:10. This may make the first customer quite unhappy.

The fundamental mechanism for achieving these kinds of traffic-management goals in a shared network is through **queuing**; that is, in deciding how the routers prioritize traffic. In this chapter we will take a look at what router-based strategies are available in the toolbox; in the following chapter we will study how some of these ideas have been applied to develop distributed quality-of-service options.

Previously, in *14.1 A First Look At Queuing*, we looked at FIFO queuing – both tail-drop and random-drop variants – and (briefly) at priority queuing. These are examples of **queuing disciplines**, a catchall term for anything that supports a way to accept and release packets. The RED gateway strategy (*14.8.3 RED*) could qualify as a separate queuing discipline, too, although one closely tied to FIFO.

Queuing disciplines provide tools for meeting administratively imposed constraints on traffic. Two senders, for example, might be required to share an outbound link equally, or in the proportion 60%-40%, even if one participant would prefer to use 100% of the bandwidth. Alternatively, a sender might be required not to send in bursts of more than 10 packets at a time.

Closely allied to the idea of queuing is **scheduling**: deciding what packets get sent when. Scheduling may take the form of sending someone else's packets right now, or it may take the form of delaying packets that are arriving too fast.

While priority queuing is one practical alternative to FIFO queuing, we will also look at so-called **fair queuing**, in both flat and hierarchical forms. Fair queuing provides a straightforward strategy for dividing bandwidth among multiple senders according to preset percentages.

Also introduced here is the **token-bucket** mechanism, which can be used for traffic scheduling but also for traffic *description*.

Some of the material here – in particular that involving fair queuing and the Parekh-Gallager theorem – may give this chapter a more mathematical feel than earlier chapters. Mostly, however, this is confined to the proofs; the claims themselves are more straightforward.

19.1 Queuing and Real-Time Traffic

One application for advanced queuing mechanisms is to support **real-time** transport – that is, traffic with delay constraints on delivery.

In its original conception, the Internet was arguably intended for non-time-critical transport. If you wanted to place a digital phone call where every (or almost every) byte was guaranteed to arrive within 50 ms, your best bet might be to use the (separate) telephone network instead.

And, indeed, having an entirely separate network for real-time transport is definitely a workable solution. It is, however, expensive; there are many economies of scale to having just a single network. There is, therefore, a great deal of interest in figuring out how to get the Internet to support real-time traffic directly.

The central strategy for mixing real-time and bulk traffic is to use queuing disciplines to give the real-time traffic the service it requires. Priority queuing is the simplest mechanism, though the fair-queuing approach below offers perhaps greater flexibility.

We round out the chapter with the Parekh-Gallager theorem, which provides a precise delay bound for real-time traffic that shares a network with bulk traffic. All that is needed is that the real-time traffic satisfies a token-bucket specification and is assigned bandwidth guarantees through fair queuing; the volume of bulk traffic does not matter. This is exactly what is needed for real-time support.

While this chapter contains some rather elegant theory, it is not at all clear how much it is put into practice today, at least for real-time traffic at the ISP level. We will return to this issue in the following chapter, but for now we acknowledge that real-time traffic management using the queuing mechanisms described here has seen limited acceptance in the Internet marketplace.

19.2 Traffic Management

Even if none of your traffic has real-time constraints, you still may wish to allocate bandwidth according to administratively determined percentages. For example, you may wish to give each of three departments an equal share of download (or upload) capacity, or you may wish to guarantee them shares of 55%, 35% and 10%. If you want any unused capacity to be divided among the non-idle users, fair queuing is the tool of choice, though in some contexts it may require cooperation from your ISP. If the users are more like customers receiving only the bandwidth they pay for, you might want to enforce flat caps even if some bandwidth thus goes unused; token-bucket filtering would then be the way to go. If bandwidth allocations are not only by department (or customer) but also by workgroup (or customer-specific subcategory), then hierarchical queuing offers the necessary control.

In general, **network management** divides into managing the hardware and managing the traffic; the tools in this chapter address this latter component. These tools can be used internally by ISPs and at the customer/ISP interconnection, but traffic management often makes good economic sense even when entirely contained within a single organization.

19.3 Priority Queuing

To get started, let us fill in the details for **priority queuing**, which we looked at briefly in *14.1.1 Priority Queuing*. Here a given outbound interface can be thought of as having two (or more) physical queues representing different priority levels. Packets are placed into the appropriate subqueue based on some packet attribute, which might be an explicit priority tag, or which might be the packet's destination socket. Whenever the outbound link becomes free and the router is able to send the next packet, it always looks first to

the higher-priority queue; if it is nonempty then a packet is dequeued from there. Only if the higher-priority queue is empty is the lower-priority queue served.

Note that priority queuing is nonpreemptive: if a high-priority packet arrives while a low-priority packet is being sent, the latter is not interrupted. Only when the low-priority packet has finished transmission does the router again check its high-priority subqueue(s).

Priority queuing can lead to complete starvation of low-priority traffic, but only if the high-priority traffic consumes 100% of the outbound bandwidth. Often we are able to guarantee (for example, through admission control) that the high-priority traffic is limited to a designated fraction of the total outbound bandwidth.

19.4 Queuing Disciplines

As an abstract data type, a **queuing discipline** is simply a data structure that supports the following operations:

- `enqueue()`
- `dequeue()`
- `is_empty()`

Note that the `enqueue()` operation includes within it a way to handle dropping a packet in the event that the queue is full. For FIFO queuing, the `enqueue()` operation needs only to know the correct outbound interface; for priority queuing `enqueue()` also needs to be told – or be able to infer – the packet's priority classification.

We may also in some cases find it convenient to add a `peek()` operation to return the next packet that *would* be dequeued if we were actually to do that, or at least to return some important statistic (*eg* size or arrival time) about that packet.

As with FIFO and priority queuing, any queuing discipline is always tied to a specific *outbound* interface. In that sense, any queuing discipline has a single output.

On the input side, the situation may be more complex. The FIFO queuing discipline has a single input stream, though it may be fed by multiple physical input interfaces: the `enqueue()` operation puts all packets in the same physical queue. A queuing discipline may, however, have multiple input streams; we will call these **classes**, or **subqueues**, and will refer to the queuing discipline itself as **classful**. Priority queues, for example, have an input class for each priority level.

When we want to enqueue a packet for a classful queuing discipline, we must first invoke a **classifier** – possibly external to the queuing discipline itself – to determine the input class. (In the linux documentation, what we have called classifiers are often called *filters*.) For example, if we wish to use a priority queue to give priority to VoIP packets, the classifier's job is to determine which arriving packets are in fact VoIP packets (perhaps taking into account things like size or port number or source host), so as to be able to provide this information to the `enqueue()` operation. The classifier might also take into account pre-existing traffic **reservations**, so that packets that belong to flows with reservations get preferred service, or else packet **tags** that have been applied by some upstream router; we return to both of these in *20 Quality of Service*.

The number and configuration of classes is often fixed at the time of queuing-discipline creation; this is typically the case for priority queues. Abstractly, however, the classes can also be dynamic; an example of

this might be fair queuing (below), which often supports a configuration in which a separate input class is created on the fly for each separate TCP connection.

FIFO and priority queuing are both **work-conserving**, meaning that the associated outbound interface is not idle unless the queue is empty. A non-work-conserving queuing discipline might, for example, artificially delay some packets in order to enforce an administratively imposed bandwidth cap. Non-work-conserving queuing disciplines are often called traffic **shapers** or **policers**; see *19.9 Token Bucket Filters* below for an example.

19.5 Fair Queuing

An important alternative to FIFO and priority is **fair queuing**. Where FIFO and its variants have a single input class and put all the incoming traffic into a single physical queue, fair queuing maintains a separate logical FIFO subqueue for each input class; we will refer to these as the per-class subqueues. Division into classes can be fine-grained – *eg* a separate class for each TCP connection – or coarse-grained – *eg* a separate class for each arrival interface, or a separate class for each designated internal subnet.

Suppose for a moment that all packets are the same size; this makes fair queuing much easier to visualize. In this (special) case – sometimes called Nagle fair queuing, and proposed in **RFC 970** – the router simply services the per-class subqueues in round-robin fashion, sending one packet from each in turn. If a per-class subqueue is empty, it is simply skipped over. If all per-class subqueues are always nonempty this resembles time-division multiplexing (*4.2 Time-Division Multiplexing*). However, unlike time-division multiplexing if one of the per-class subqueues does become empty then it no longer consumes any outbound bandwidth. Recalling that all packets are the same size, the total bandwidth is then divided equally among the nonempty per-class subqueues; if there are K such queues, each will get 1/K of the output.

Fair queuing was extended to streams of variable-sized packets in *[DKS89]* and *[LZ89]*. Since then there has been considerable work in trying to figure out how to implement fair queuing efficiently and to support appropriate variants.

19.5.1 Weighted Fair Queuing

A straightforward extension of fair queuing is **weighted fair queuing** (WFQ), where instead of giving each class an equal share, we assign each class a different percentage. For example, we might assign bandwidth percentages of 10%, 30% and 60% to three different departments. If all three subqueues are active, each gets the listed percentage. If the 60% subqueue is idle, then the others get 25% and 75% respectively, preserving the 1:3 ratio of their allocations. If the 10% subqueue is idle, then the other two subqueues get 33.3% and 66.7%.

Weighted fair queuing is, conceptually, a straightforward generalization of fair queuing, although the actual implementation details are sometimes nontrivial as the round-robin implementation above naturally yields equal shares. If all packets are still the same size, and we have two per-class subqueues that are to receive allocations of 40% and 60% (that is, in the ratio 2:3), then we could implement WFQ by having one per-class subqueue send two packets and the other three. Or we might intermingle the two: class 1 sends its first packet, class 2 sends its first packet, class 1 sends its second, class 2 sends its second and its third. If the allocation is to be in the ratio 1:$\sqrt{2}$, the first sender might always send 1 packet while the second might send in a pattern – an irregular one – that averages $\sqrt{2}$: 1, 2, 1, 2, 1, 1, 2, ...

The fluid-based GPS model approach to fair queuing, *19.5.4 The GPS Model*, does provide an algorithm that has direct, natural support for weighted fair queuing.

19.5.2 Different Packet Sizes and Virtual Finishing Times

If not all the packets are the same size, fair queuing and weighted fair queuing are still possible but we have a little more work to do. This is an important practical case, as fair queuing is often used when one input class consists of small-packet real-time traffic.

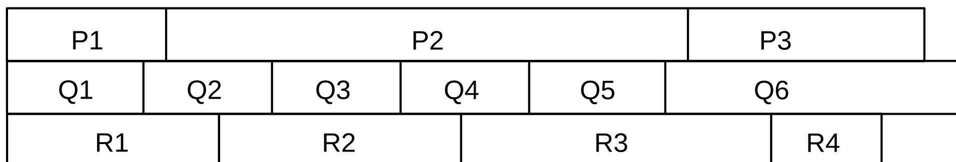
We present two mechanisms for handling different-sized packets; the two are ultimately equivalent. The first – *19.5.3 Bit-by-bit Round Robin* – is a straightforward extension of the round-robin idea, and the second – *19.5.4 The GPS Model* – uses a "fluid" model of simultaneous packet transmission. Both mechanisms share the idea of a "virtual clock" that runs at a rate inversely proportional to the number of active subqueues; as we shall see, the point of varying the clock rate in this way is so that the virtual-clock time at which a given packet would theoretically *finish* transmission does not depend on activity in any of the other subqueues.

Finally, we present the quantum algorithm – *19.5.5 The Quantum Algorithm* – which is a more-efficient approximation to either of the exact algorithms, but which – being an approximation – no longer satisfies a sometimes-important time constraint.

For a straightforward generalization of the round-robin idea to different packet sizes, we start with a simplification: let us assume that each per-class subqueue is always active, where a subqueue is **active** if it is nonempty whenever the router looks at it.

If each subqueue is always active for the **equal**-sized-packets case, then packets are transmitted in order of increasing (or at least nondecreasing) cumulative data sent by each subqueue. In other words, every subqueue gets to send its first packet, and only then do we go on to begin transmitting second packets, and so on.

Still assuming each subqueue is always active, we can handle **different**-sized packets by the same idea. For packet P, let C_P be the cumulative number of bytes that will have been sent by P's subqueue as of the *end* of P. Then we simply need to send packets in nondecreasing order of C_P.



Variable-packet-sized fair queuing with all subqueues active

Packet transmission order: Q1, P1, R1, Q2, Q3, R2, Q4, Q5, P2, R3, R4, P3, Q6

In the diagram above, transmission in nondecreasing order of C_P means transmission in left-to-right order of the vertical lines marking packet divisions, *eg* Q1, P1, R1, Q2, Q3, R2, ... This ensures that, in the long run, each subqueue gets an equal share of bandwidth.

A completely equivalent strategy, better suited for generalization to the case where not all subqueues are always active, is to send each packet in nondecreasing order of **virtual finishing times**, calculated for each packet with the assumption that only that packet's subqueue is active. The virtual finishing time F_P of packet

P is equal to C_P divided by the output bandwidth. We use finishing times rather than starting times because if one packet is very large, shorter packets in other subqueues that would finish sooner should be sent first.

19.5.2.1 A first virtual-finish example

As an example, suppose there are two subqueues, P and Q. Suppose further that a stream of 1001-byte packets P_1, P_2, P_3, ... arrives for P, and a stream of 400-byte packets Q_1, Q_2, Q_3, ... arrives for Q; each stream is steady enough that each subqueue is always active. Finally, assume the output bandwidth is 1 byte per unit time, and let T=0 be the starting point.

For the P subqueue, the virtual finishing times calculated as above would be P_1 at 1001, P_2 at 2002, P_3 at 3003, *etc*; for Q the finishing times would be Q_1 at 400, Q_2 at 800, Q_3 at 1200, *etc*. So the order of transmission of all the packets together, in increasing order of virtual finish, will be as follows:

Packet	virtual finish	actual finish
Q_1	400	400
Q_2	800	800
P_1	1001	1801
Q_3	1200	2201
Q_4	1600	2601
Q_5	2000	3001
P_2	2002	4002

For each packet we have calculated in the table above its virtual finishing time, and then its actual wallclock finishing time assuming packets are transmitted in nondecreasing order of virtual finishing time (as shown).

Because both subqueues are always active, and because the virtual finishing times assumed each subqueue received 100% of the output bandwidth, in the long run the actual finishing times will be about double the virtual times. This, however, is irrelevant; all that matters is the *relative* virtual finishing times.

19.5.2.2 A second virtual-finish example

For the next example, however, we allow a subqueue to be idle for a while and then become active. In this situation virtual finishing times do not work quite so well, at least when based directly on wallclock time. We return to our initial simplification that all packets are the same size, which we take to be 1 unit; this allows us to apply the round-robin mechanism to determine the transmission order and compare this to the virtual-finish order. Assume there are three queues P, Q and R, and P is empty until wallclock time 20. Q is constantly busy; its Kth packet Q_K, starting with K=1, has virtual finishing time F_K = K.

For the first case, assume R is completely idle. When P's first packet P_1 arrives at time 20, its virtual finishing time will be 21. At time 20 the head packet in Q will be Q_{21}; the two packets therefore have identical virtual finishing times. And, encouragingly, under round-robin queue service P_1 and Q_{21} will be sent in the same round.

For the second case, however, suppose R is *also* constantly busy. Up until time 20, Q and R have each sent 10 packets; their next packets are Q_{11} and R_{11}, each with a virtual finishing time of T=11. When P's first packet arrives at T=20, again with virtual finishing time 21, under round-robin service it should be sent in the same round as Q_{11} and R_{11}. Yet their virtual finishing times are off by a factor of about two; queue P's

stretch of inactivity has left it far behind. Virtual finishing times, as we have been calculating them so far, simply do not work.

The trick, as it turns out, is to measure elapsed time not in terms of packet-transmission times (*ie* wallclock time), but rather in terms of *rounds* of round-robin transmission. This amounts to *scaling* the clock used for measuring arrival times; counting in rounds rather than packets means that we run this clock at rate 1/N when N subqueues are active. If we do this in case 1, with N=1, then the finishing times are unchanged. However, in case 2, with N=2, packet P_1 arrives after 20 time units but only 10 rounds; the clock runs at half rate. Its calculated finishing time is thus 11, exactly matching the finishing times of the two long-queued packets Q_{11} and R_{11} with which P_1 shares a round-robin transmission round.

We formalize this in the next section, extending the idea to include both variable-sized packets and sometimes-idle subqueues. Note that only the clock that measures arrival times is scaled; we do not scale the calculated transmission times.

19.5.3 Bit-by-bit Round Robin

Imagine sending a single *bit* at a time from each active input subqueue, in round-robin fashion. While not directly implementable, this certainly meets the goal of giving each active subqueue equal service, even if packets are of different sizes. We will use bit-by-bit round robin, or **BBRR**, as a way of modeling packet-finishing times, and then, as in the previous example, send the packets the usual way – one full packet at a time – in order of increasing BBRR-calculated virtual finishing times.

It will sometimes happen that a larger packet is being transmitted at the point a new, shorter packet arrives for which a smaller finishing time is computed. The current transmission is not interrupted, though; the algorithm is **non-preemptive**.

The trick to making the BBRR approach workable is to find an "invariant" formulation of finishing time that does not change as later packets arrive, or as other subqueues become active or inactive. To this end, taking the lead from the example of the previous section, we define the "rounds counter" R(t), where t is the time measured in units of the transmission time for one bit. When there are any active (nonempty or currently transmitting) input subqueues, R(t) counts the number of round-robin 1-bit cycles that have occurred since the last time all the subqueues were empty. If there are K active input subqueues, then R(t) increments by 1 as t increments by K; that is, R(t) grows at rate 1/K.

An important attribute of R(t) is that, if a packet of size S bits starts transmission via BBRR at $R_0 = R(t_0)$, then it will finish when $R(t) = R_0+S$, *regardless of whether any other input subqueues become active or become empty*. For any packet actively being sent via BBRR, R(t) increments by 1 for each bit of that packet sent. If for a given round-robin cycle there are K subqueues active, then K bits will be sent in all, and R(t) will increment by 1.

To calculate the virtual BBRR finishing time of a packet P, we first record $R_P = R(t_P)$ at the moment of arrival. We now compute the BBRR-finishing R-value F_P as follows; we can think of this as a "time" measured via the rounds counter R(t). That is, R(t) represents a "virtual clock" that happens sometimes to run slow. Let S be the size of the packet P in bits. If P arrived on a previously empty input subqueue, then its BBRR transmission can begin immediately, and so its finishing R-value F_P is simply R_P+S. If the packet's subqueue was nonempty, we look up the (future) finishing R-value of the packet immediately ahead of P in its subqueue, say F_{prev}; the finishing R-value of P is then $F_P = F_{prev} + S$. This is sometimes described as:

Start = max(R(now), F_{prev})

F_P = Start + S (S = packet size, measured in bits)

As each new packet P arrives, we calculate its BBRR-finishing R-value F_P, and then send packets the conventional one-packet-at-a-time way in increasing order of F_P. As stated above, F_P will not change if other subqueues empty or become active, thus changing the rate of the rounds-counter R(t).

The router maintaining R(t) does not have to increment it on every bit; it suffices to update it whenever a packet arrives or a subqueue becomes empty. If the previous value of R(t) was R_{prev}, and from then to now exactly K subqueues were nonempty, and M bit-times have elapsed according to the wall clock, then the current value of R(t) is R_{prev} + M/K.

19.5.3.1 BBRR example

As an example, suppose the fair queuing router has three input subqueues P, Q and R, initially empty. The following packets arrive at the wall-clock times shown.

Packet	Queue	Size	Arrival time, t
P_1	P	1000	0
P_2	P	1000	0
Q_1	Q	600	800
Q_2	Q	400	800
Q_3	Q	400	800
R_1	R	200	1200
R_2	R	200	2100

At t=0, we have R(t)=0 and we assign finishing R-values $F(P_1)$=1000 to P_1 and $F(P_2)$ = $F(P_1)$+1000 = 2000 to P_2. Transmission of P_1 begins.

When the three Q packets arrive at t=800, we have R(t)=800 as well, as only one subqueue has been active. We assign finishing R-values for the newly arriving Q_1, Q_2 and Q_3 of $F(Q_1)$ = 800+600 = 1400, $F(Q_2)$ = 1400+400 = 1800, and $F(Q_3)$ = 1800+400 = 2200. At this point, BBRR begins serving two subqueues, so the R(t) rate is cut in half.

At t=1000, transmission of packet P_1 is completed; R(t) is 800 + 200/2 = 900. The smallest finishing R-value on the books is $F(Q_1)$, at 1400, so Q_1 is the second packet transmitted. Q_1's real finishing time will be t = 1000+600 = 1600.

At t=1200, R_1 arrives; transmission of Q_1 is still in progress. R(t) is 800 + 400/2 = 1000; we calculate $F(R_1)$ = 1000 + 200 = 1200. Note this is less than the finishing R-value for Q_1, which is currently transmitting, but Q_1 is not preempted. At this point (t=1200, R(t)=1000), the R(t) rate falls to 1/3.

At t=1600, Q_1 has finished transmission. We have R(t) = 1000 + 400/3 = 1133. The next smallest finishing R-value is $F(R_1)$ = 1200 so transmission of R_1 commences.

At t=1800, R_1 finishes. We have R(1800) = R(1200) + 600/3 = 1000 + 200 = 1200 (3 subqueues have been busy since t=1200). Queue R is now empty, so the R(t) rate rises from 1/3 to 1/2. The next smallest finishing R-value is $F(Q_2)$=1800, so transmission of Q_2 begins. It will finish at t=2200.

At t=2100, we have R(t) = R(1800) + 300/2 = 1200 + 150 = 1350. R_2 arrives and is assigned a finishing time of $F(R_2)$ = 1350 + 200 = 1550. Again, transmission of Q_2 is not preempted even though $F(R_2) < F(Q_2)$. The R(t) rate again falls to 1/3.

At t=2200, Q_2 finishes. R(t) = 1350 + 100/3 = 1383. The next smallest finishing R-value is F(R_2)=1550, so transmission of R_2 begins.

At t=2400, transmission of R_2 ends. R(t) is now 1350 + 300/3 = 1450. The next smallest finishing R-value is F(P_2) = 2000, so transmission of P_2 begins. The R(t) rate rises to 1/2, as queue R is again empty.

At t=3400, transmission of P_2 ends. R(t) is 1450 + 1000/2 = 1950. The only remaining unsent packet is Q_3, with F(Q_3)=2200. We send it.

At t=3800, transmission of Q_3 ends. R(t) is 1950 + 400/1 = 2350.

To summarize:

Packet	send-time, wall clock t	calculated finish R-value	R-value when sent	R-value at finish
P_1	0	1000	0	900
Q_1	1000	1400	900	1133
R_1	1600	1200*	1133	1200
Q_2	1800	1800	1200	1383
R_2	2200	1550*	1383	1450
P_2	2400	2000	1450	1950
Q_3	3400	2200	1950	2350

Packets arrive, begin transmission and finish in "real" time. However, the number of queues active in real time affects the rate of the rounds-counter R(t); this value is then attached to each packet as it arrives as its virtual finishing time, and determines the order of packet transmission.

The change in R-value from start to finish exactly matches the packet size when the packet is "virtually sent" via BBRR. When the packet is sent as an indivisible unit, as in the table above, the change in R-value is usually much smaller, as the R-clock runs slower whenever at least two subqueues are in use.

The calculated-finish R-values are not in fact increasing, as can be seen at the **starred (*) values**. This is because, for example, R_1 was not yet available when it was time to send Q_1.

Computationally, maintaining the R-value counter is inconsequential. The primary performance issue with BBRR simulation is the need to find the smallest R-value whenever a new packet is to be sent. If n is the number of packets waiting to be sent, then we can do this in time O(log(n)) by keeping the R-values sorted in an appropriate data structure.

The BBRR approach assumes equal weights for each subqueue; this does not generalize completely straightforwardly to weighted fair queuing as the number of subqueues cannot be fractional. If there are two queues, one which is to have weight 40% and the other 60%, we *could* use BBRR with five subqueues, two of which (2/5) are assigned to the 40%-subqueue and the other three (3/5) to the 60% subqueue. But this becomes increasingly awkward as the fractions become less simple; the GPS model, next, is a better option.

19.5.4 The GPS Model

An almost-equivalent model to BBRR is the **generalized processor sharing** model, or GPS; it was first developed as an application to CPU scheduling. In this approach we imagine the packets as liquid, and the outbound interface as a pipe that has a certain total capacity. The head packets from each subqueue are all squeezed into the pipe *simultaneously*, each at its designated fractional rate. The GPS model is essentially

an "infinitesimal" variant of BBRR. The GPS model has an advantage of generalizing straightforwardly to *weighted* fair queuing.

Other fluid models have also been used in the analysis of networks, *eg* for the study of TCP, though we do not consider these here. See *[MW00]* for one example.

For the GPS model, assume there are N input subqueues, and the ith subqueue, $0 \leqslant i < N$, is to receive fraction $\alpha_i > 0$, where $\alpha_0+\alpha_1+ ... + \alpha_{N-1}=1$. If at some point a set A of input subqueues is active , say A = {0,2,4}, then subqueue 0 will receive fraction $\alpha_0/(\alpha_0+\alpha_2+\alpha_4)$, and subqueues 2 and 4 similarly. The router forwards packets from each active subqueue simultaneously, each at its designated rate.

The GPS model (and the BBRR model) provides an ideal degree of **isolation** between input flows: each flow is insulated from any delay due to packets on competing flows. The ith flow receives bandwidth of at least α_i and packets wait only for other packets belonging to the same flow. When a packet arrives for an inactive subqueue, forwarding begins immediately, interleaved with any other work the router is doing. Traffic on other flows can reduce the real rate of a flow, but not its virtual rate.

While GPS is convenient as a model, it is even less implementable, literally, than BBRR. As with BBRR, though, we can use the GPS model to determine the order of one-packet-at-a-time transmission. As each real packet arrives, we calculate the time it *would* finish, if we were using GPS. Packets are then transmitted under WFQ one at a time, in order of increasing GPS finishing time.

In lieu of the BBRR rounds counter R(t), a virtual clock VC(t) is used that runs at an *increased* rate $1/\alpha \geqslant 1$ where α is the sum of the α_i for the active subqueues. That is, if subqueues 0, 2 and 4 are active, then the VC(t) clock runs at a rate of $1/(\alpha_0+\alpha_2+\alpha_4)$. If all the α_i are equal, each to 1/N, then VC(t) always runs N times faster than R(t), and so VC(t) = N×R(t); the VC clock runs at wallclock speed when all input subqueues are active and speeds up as subqueues become idle.

For any one active subqueue i, the GPS rate of transmission *relative to the virtual clock* (that is, in units of bits per virtual-second) is always equal to fraction α_i of the full output-interface rate. That is, if the output rate is 10 Mbps and an active flow has fraction $\alpha = 0.4$, then it will always transmit at 4 bits per virtual microsecond. When all the subqueues are active, and the VC clock runs at wallclock speed, the flow's actual rate will be 4 bits/µsec. When the subqueue is active alone, its speed measured by a real clock will be 10 bit/µsec but the virtual clock will run 2.5 times faster so 10 bits/µsec is 10 bits per 2.5 virtual microseconds, or 4 bits per virtual microsecond.

To make this claim more precise, let A be the set of active queues, and let α again be the sum of the α_j for j in A. Then VC(t) runs at rate $1/\alpha$ and active subqueue i's data is sent at rate α_i/α relative to wallclock time. Subqueue i's transmission rate relative to virtual time is thus $(\alpha_i/\alpha)/(1/\alpha) = \alpha_i$.

As other subqueues become inactive or become active, the VC(t) rate and the actual transmission rate move in lockstep. Therefore, as with BBRR, a packet P of size S on subqueue i that starts transmission at virtual time T will finish at $T + S/(r\times\alpha_i)$ by the VC clock, where r is the actual output rate of the router, regardless of what is happening in the other subqueues. In other words, *VC-calculated finishing times are invariant*.

To round out the calculation of finishing times, suppose packet P of size S arrives on an active GPS subqueue i. The p

$$F_P = \max(VC(now), F_{prev}) + S/(r\times\alpha_i)$$

In *19.8.1.1 WFQ with non-FIFO subqueues* below, we will consider WFQ routers that, as part of a hierarchy, are in effect only allowed to transmit intermittently. In such a case, the virtual clock should be

suspended whenever output is blocked. This is perhaps easiest to see for the BBRR scheduler: the rounds-counter RR(t) is to increment by 1 for each bit sent by each active subqueue. When no bits may be sent, the clock should not increase.

As an example of what happens if this is not done, suppose R has two subqueues A and B; the first is empty and the second has a long backlog. R normally processes one packet per second. At T=0/VC=0, R's output is suspended. Packets in the second subqueue b_1, b_2, b_3, ... have virtual finishing times 1, 2, 3, ... At T=10, R resumes transmission, and packet a_1 arrives on the A subqueue. If R's virtual clock had been suspended for the interval $0 \leqslant T \leqslant 10$, a_1 would be assigned finishing time T=1 and would have priority comparable to b_1. If R's virtual clock had continued to run, a_1 would be assigned finishing time T=11 and would not be sent until b_{11} reached the head of the B queue.

19.5.4.1 The WFQ scheduler

To schedule actual packet transmission under weighted fair queuing, we calculate upon arrival each packet's virtual-clock finishing time assuming it were to be sent using GPS. Whenever the sender is ready to start transmission of a new packet, it selects from the available packets the one with the smallest GPS-finishing-time value. By the argument above, a packet's GPS finishing time does not depend on any later arrivals or idle periods on other subqueues. As with BBRR, small but later-arriving packets might have smaller virtual finishing times, but a packet currently being transmitted will not be interrupted.

19.5.4.2 Finishing Order under GPS and WFQ

We now look at the order in which packets finish transmission under GPS versus WFQ. The goal is to provide in *19.5.4.7 Finishing-Order Bound* a tight bound on how long packets may have to wait under WFQ compared to GPS. We emphasize again:

- GPS finishing time: the theoretical finishing time based on parallel multi-packet transmissions under the GPS model
- WFQ finishing time: the real finishing time assuming packets are sent sequentially in increasing order of calculated GPS finishing time

One way to view this is as a quantification of the informal idea that WFQ provides a natural priority for smaller packets, at least smaller packets sent on previously idle subqueues. This is quite separate from the bandwidth guarantee that a given small-packet input class might receive; it means that small packets are likely to leapfrog larger packets waiting in other subqueues. The quantum algorithm, below, provides long-term WFQ bandwidth guarantees but does *not* provide the same delay assurances.

First, if all subqueues are always active (or if a fixed subset of subqueues is always active), then packets finish under WFQ in the same order as they do under GPS. This is because under WFQ packets are transmitted in the order of GPS finishing times according the virtual clock, and if all subqueues are always active the virtual clock runs at a rate identical to wallclock time (or, if a fixed subset of subqueues is always active, at a rate proportional to wallclock time).

If all subqueues are always active, we can assume that all packets were in their subqueues as of time T=0; the finishing order is the same as long as each packet arrived before its subqueue went inactive.

Finally, if all subqueues are always active then each packet finishes at least as early under WFQ as under GPS. To see this, let P_j be the jth packet to finish, under either GPS or WFQ. At the time when P_j finishes

under WFQ, the router R will have devoted 100% of its output bandwidth exclusively to P_1 through P_j. When P_j finishes under GPS, R will also have transmitted P_1 through P_j, and may have transmitted fractions of later packets as well. Therefore, the P_j finishing time under GPS cannot be earlier.

The finishing order and the relative GPS/WFQ finishing times may change, however, if – as will usually be the case – some subqueues are sometimes idle; that is, if packets sometimes "arrive late" for some subqueues.

19.5.4.3 GPS Example 1

As a first example we return to the scenario of *19.5.2.1 A first virtual-finish example*. The router's two subqueues are **always active**; each has an allocation of α=50%. Packets P_1, P_2, P_3, ..., all of size 1001, wait in the first queue; packets Q_1, Q_2, Q_3, ..., all of size 400, wait in the second queue. Output bandwidth is 1 byte per unit time, and T=0 is the starting point.

The router's virtual clock runs at wallclock speed, as both subqueues are always active. If F_i represents the virtual finishing time of R_i, then we now calculate F_i as $F_{i-1} + 400/\alpha = F_{i-1} + 800$. The virtual finishing times of P_1, P_2, *etc* are similarly at multiples of 2002.

Packet	virtual finish	actual finish time
Q_1	800	400
Q_2	1600	800
P_1	2002	1801
Q_3	2400	2201
Q_4	3200	2601
Q_5	4000	3001
P_2	4004	4002

In the table above, the "virtual finish" column is simply double that of the BBRR version, reflecting the fact that the virtual finishing times are now scaled by a factor of $1/\alpha = 2$. The actual finish times are identical to what we calculated before.

Note that, in every case, the actual WFQ finish time is always less than or equal to the virtual GPS finish time.

19.5.4.4 GPS Example 2

If the router has only a single active subqueue, with share α and packets P_1, P_2, P_3, ..., then the calculated virtual-clock packet finishing times will be equal to the time on the virtual clock at the point of actual finish, at least if this has been the case since the virtual clock last restarted at T=VC=0. Let r be the output rate of the router, let S_1, S_2, S_3 be the sizes of the packets and let F_1, F_2, F_3 be their virtual finishing times with F_0=0. Then

$$F_i = F_{i-1} + S_i/(r\alpha) = S_1/(r\alpha) + ... + S_i/(r\alpha)$$

The ith packet's actual finishing time A_i is $(S_1 + ... + S_i)/r$, which is $\alpha \times F_i$. But the virtual clock runs fast by a factor of $1/\alpha$, so the actual finishing time on the virtual clock is $A_i/\alpha = F_i$.

19.5.4.5 GPS Example 3

The next example illustrates a smaller but later-arriving packet, in this case Q_2, that finishes ahead of P_2 under GPS but not under WFQ. P_2 can be said to *leapfrog* Q_2 and R_1 under WFQ.

Suppose packets P_1, Q_1, P_2, Q_2 and R_1 arrive at a router at the following times T, and with the following lengths L. The output bandwidth is 1 length unit per time unit; that is, r=1. The total number of length units is 24. Each subqueue is allocated an equal share of the bandwidth; *eg* α=1/3.

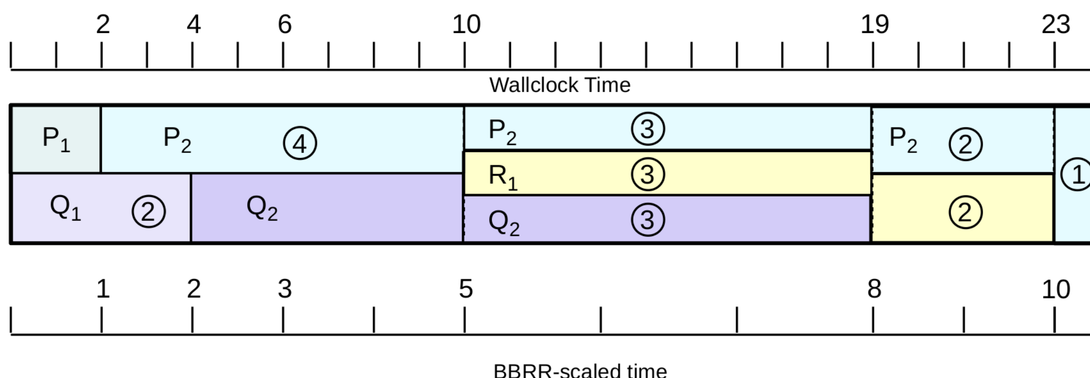
subqueue 1	subqueue 2	subqueue 3
P_1: T=0, L=1	Q_1: T=0, L=2	R_1: T=10, L=5
P_2: T=2, L=10	Q_2: T=4, L=6	

Under WFQ, we send P_1 and then Q_1; Q_1 is second because its finishing time is later. When Q_1 finishes the wallclock time is T=3. At this point, P_2 is the only packet available to send; it finishes at T=13.

Up until T=10, we have two packets in progress under GPS (because Q_1 finishes under GPS at T=4 and Q_2 arrives at T=4), and so the GPS clock runs at rate 3/2 of wallclock time and the BBRR clock runs at rate 1/2 of wallclock time. At T=4, when Q_2 arrives, the BBRR clock is at 2 and the VC clock is at 6 and we calculate the BBRR finishing time as 2+6=8 and the GPS finishing time as 6+6/(1/3) = 24. At T=10, the BBRR clock is at 5 and the GPS clock is 15. R_1 arrives then; we calculate its BBRR finishing time as 5+5=10 and its GPS finishing time as 15+5/α = 30.

Because Q_2 has the earlier virtual-clock finishing time, WFQ sends it next after P_2, followed by R_1.

Here is a diagram of transmission under GPS. The chart itself is scaled to wallclock times. The BBRR clock is on the scale below; the VC clock always runs three times faster.



The circled numbers represent the size of the portion of the packet sent in the intervals separated by the dotted vertical lines; for each packet, these add up to the packet's total size.

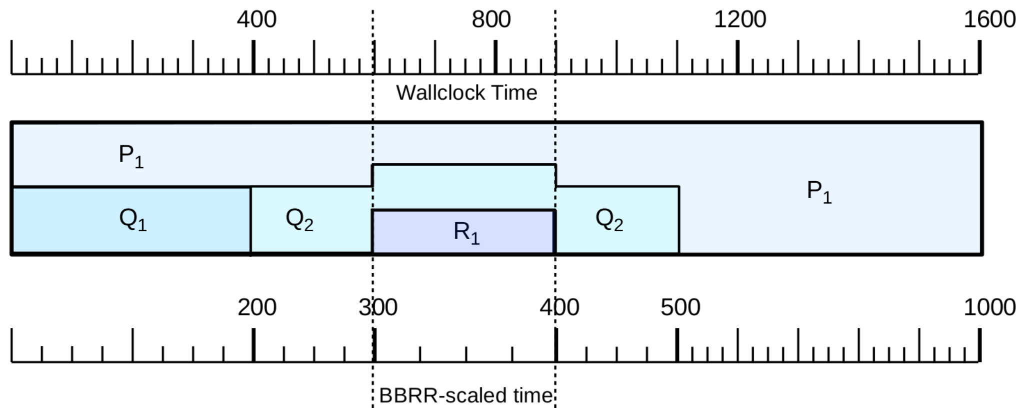
Note that, while the transmission order under WFQ is P_1, Q_1, P_2, Q_2, R_1, the finishing order under GPS is P_1, Q_1, Q_2, R_1, P_2. That is, P_2 managed to leapfrog Q_2 and R_1 under WFQ by the simple expedient of being the only packet available for transmission at T=3.

19.5.4.6 GPS Example 4

As a second example of leapfrogging, suppose we have the following arrivals; in this scenario, the smaller but later-arriving R_1 finishes ahead of P_1 and Q_2 under GPS, but not under WFQ.

subqueue 1	subqueue 2	subqueue3
P_1: T=0, L=1000	Q_1: T=0, L=200	R_1: T=600, L=100
	Q_2: T=0, L=300	

The following diagram shows how the packets shared the link under GPS over time. As can be seen, the GPS finishing order is Q_1, R_1, Q_2, P_1.



Under WFQ, the transmission order is Q_1, Q_2, P_1, R_1, because when Q_2 finishes at T=500, R_1 has not yet arrived.

19.5.4.7 Finishing-Order Bound

These examples bring us to the following delay-bound claim, due to Parekh and Gallager *[PG93]*; we will make use of it below in *19.13.3 Parekh-Gallager Theorem*. It is arguably the deepest part of the Parekh-Gallager theorem.

Claim: For any packet P, the wallclock finishing time of P at a router R under WFQ cannot be later than the wallclock finishing time of P at R under GPS by more than the time R needs to transmit the maximum-sized packet that can appear.

Expressed symbolically, if F_{WFQ} and F_{GPS} are the finishing times for P under WFQ and GPS, R's outbound transmission rate is r, and L_{max} is the maximum packet size that can appear at R, then

$$F_{WFQ} \leqslant F_{GPS} + L_{max}/r$$

This is the best possible bound; L_{max}/r is the time packet P must wait if it has arrived an instant too late and another packet of size L_{max} has started instead.

Note that, if a packet's subqueue is inactive, the packet *starts* transmitting immediately upon arrival under GPS; however, GPS may send the packet relatively slowly.

To prove this claim, let us number the packets P_1 through P_k in order of WFQ transmission, starting from the most recent point when at least one subqueue of the router became active. (Note that these packets may be spread over multiple input subqueues.) For each i, let F_i be the finishing time of P_i under WFQ, let G_i be the finishing time of P_i under GPS, and let L_i be the length of P_i; note that, for each i, $F_{i+1} = L_{i+1}/r + F_i$.

If P_k finishes after P_1 through P_{k-1} under GPS, then the argument above (*19.5.4.2 Finishing Order under GPS and WFQ*) for the all-subqueues-active case still applies to show P_k cannot finish earlier under GPS

than it does under WFQ; that is, we have $F_k \leqslant G_k$.

Otherwise, some packet P_i with i<k must finish after P_k under GPS; P_i has *leapfrogged* P_k under WFQ, presumably because P_k was late in arriving. Let P_m be the most recent (largest m<k) such leapfrogger packet, so that P_m finishes after P_k under GPS but P_{m+1} through P_{k-1} finish earlier (or are tied); this was illustrated above in *19.5.4.5 GPS Example 3* for k=5 and m=3.

We next claim that none of the packets P_{m+1} through P_k could have yet arrived at R at the time T_{start} when P_m began transmission under WFQ. If some P_i with i>m were present at time T_{start}, then the fact that it is transmitted after P_m under WFQ would imply that the calculated GPS finishing time came after that of P_m. But, as we argued earlier, calculated virtual-clock GPS finishing times are always the actual virtual-clock GPS finishing times, and we cannot have P_i finishing both ahead of P_m and behind it.

Recalling that F_m is the finishing time of P_m under WFQ, the time T_{start} above is simply $F_m - L_m/r$. Between T_{start} and G_k, all the packets P_{m+1} through P_k must arrive and then, under GPS, depart. The absolute minimum time to send these packets under GPS is the WFQ time for end-to-end transmission, which is $(L_{m+1} + .. + L_k)/r = F_k - F_m$. Therefore we have

$$G_k - T_{start} \geqslant F_k - F_m$$
$$G_k - (F_m - L_m/r) \geqslant F_k - F_m$$
$$F_k \leqslant G_k + L_m/r \leqslant G_k + L_{max}/r$$

The last line represents the desired conclusion.

19.5.5 The Quantum Algorithm

The BBRR approach has cost O(log n), where n is the number of packets waiting, as outlined earlier. One simple though approximate alternative that has O(1) performance, introduced in *[SV96]*, is to give each input subqueue a **quantum**. At each round, a subqueue will be able to send a quantum's worth of packets (plus, as we shall see, a carryover from the previous round). The quantum must be a number of bytes at least as large as the network MTU (so each time a subqueue has a chance to send, it will be able to send at least one packet). Then, we again service the subqueues in round-robin fashion, but each time we take as many packets from the subqueue as we can such that the total size does not exceed the quantum.

One cost of the quantum approach is that the elegant delay bound of *19.5.4.7 Finishing-Order Bound* no longer holds. In fact, a sender may be forced to wait for all other senders each to send a full quantum, even if the sender has only a small packet.

If we just allow subqueues to send up to a quantum's worth of bytes, the strategy will be biased against classes that, for example, only include packets with size just over half the quantum. Fairness can be improved significantly by keeping track of the difference between the quantum and what was actually sent; we will call this the **deficit**. If the quantum is 1000 bytes and we have two 600-byte packets, the first of the packets is sent and the deficit is recorded as 400; if we have two 400-byte packets then the deficit is 0 as sending the two 400-byte packets empties the subqueue.

The next time that subqueue is serviced, we add the previous deficit to the quantum, and send packets up to a total cumulative size of deficit+quantum. With this variation, if the quantum is 1000 bytes and a steady stream of 600-byte packets arrives on one subqueue, they are sent as follows

round	quantum + prev deficit	packets sent	new deficit
1	1000	1	400
2	1400	2,3	200
3	1200	4,5	0

In three rounds the subqueue has been allowed to send $5 \times 600 = 3000$ bytes, which is exactly 3 quantums. We will refer to this strategy, with provision for deficits as above, as the **quantum algorithm** for fair queuing. If a subqueue is ever empty, its deficit should immediately expire.

We can implement weighted fair queuing using the quantum algorithm by adjusting the quantum sizes in proportion to the weights; that is, if the weights are 40% and 60% then the respective quanta might be 1000 bytes and 1500 bytes. The quantum should be at least as large as the largest packet (*eg* the MTU), so that if one input class is to have 10% weight and the other 90%, then the second class will have a quantum of 9 times the MTU. Of course, if the smaller-weighted class happens to be a VoIP stream with smaller packets as well, this is less of an issue.

19.5.6 Stochastic Fair Queuing

Stochastic fair queuing [McK90] is a different kind of approximation to fair queuing. The ideal is to give each individual TCP connection through a router its own fair queuing class, so that no connection has an incentive to keep more than the minimum number of packets in the queue. However, managing so many separate (and dynamically changing) input classes is computationally intensive. Instead, a hash function is used to put each TCP connection (as identified by its socketpair) into one of several **buckets** (the current default bucket count in the linux implementation is 1024). The buckets are then used as the fair queuing input queues.

If two connections hash to the same bucket, each will get only half the bandwidth to which it is entitled. Therefore the hash function has some additional parameters that allow it to be *perturbed* at regular intervals; after perturbation, socketpairs that formerly hashed to the same bucket likely now will not.

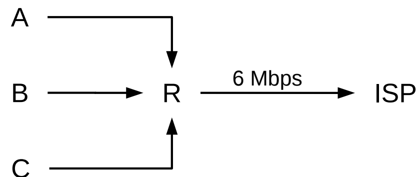
The linux implementation of stochastic fair queuing, known as **sfq**, uses the quantum algorithm, above, which may reduce the bandwidth share for real-time flows with small packets. The different subqueues ultimately share a common pool of buffer space. If a packet arrives when no more space is available, the last packet from the fullest subqueue is dropped. This rewards flows that hold their queue utilization to modest levels.

19.6 Applications of Fair Queuing

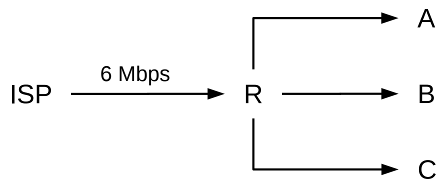
As we saw in the Queue-Competition Rule in *14.2.2 Example 2: router competition*, if a bottleneck router uses FIFO queuing then it pays a sender, in terms of the bandwidth it receives, to keep as many packets as possible in the queue. Fair queuing, however, can behave quite differently. If fair queuing is used with a separate class for each connection, this "greediness" benefit evaporates; a sender need only keep its per-class queue nonempty in order to be guaranteed its assigned share. Senders can limit their queue utilization to one or two packets without danger of being crowded out by competing traffic.

As another application of fair queuing, suppose we have three web servers, A, B and C, to which we want to give equal shares at sending along a 6 Mbps link; all three reach the link through router R. One way would be to have R restrict them each to 2 Mbps, but that is not work-conserving: if one server is idle then its share

goes unused. Fair queuing offers a better approach: if all three are busy, each gets 2 Mbps; if two are busy then each gets 3 Mbps, and if only one is busy it gets 6 Mbps. ISPs can also use this strategy internally whenever they have several flows competing for one outbound link.



Unfortunately, this strategy does not work quite as well with three *receivers*. Suppose now A, B and C are three pools of users (*eg* three departments), and we want to give them each equal shares of an *inbound* 6 Mbps link:



Inbound fair queuing could be used at the ISP (upstream) end of the connection, where the three flows would be identified by their destinations. But the ISP router is likely not ours to play with. Once the three flows arrive at R, it is too late to allocate shares; there is nothing to do but deliver the traffic to A, B and C respectively. We will return to this scenario below in *19.10 Applications of Token Bucket*.

Note that for these kinds of applications we need to be able to designate administratively how packets get classified into different input classes. The automatic classification of stochastic fair queuing, above, will not help here.

19.6.1 Fair Queuing and Bufferbloat

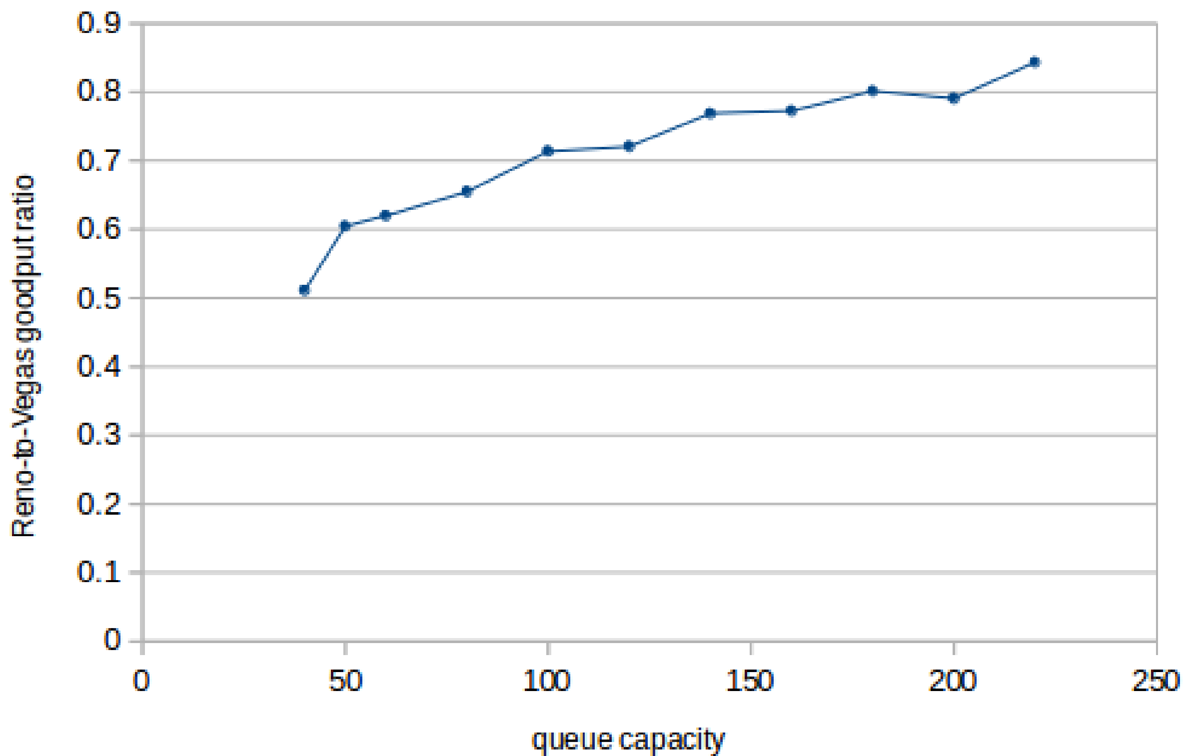
Fair Queuing provides an alternative approach, at least theoretically, to the bufferbloat problem (*13.7.1 Bufferbloat*). In *14.8 Active Queue Management* we discussed how FIFO routers can reduce excessive queue utilization. However, if a router abandoned FIFO and instead awarded each flow its own fair-queuing subqueue, there would no longer be an incentive for a TCP implementation to, like TCP Reno and TCP Cubic (*15.15 TCP CUBIC*), grab as much queue capacity as possible. The queue-competition rule of *14.2.2 Example 2: router competition* would simply no longer apply. N connections through the router would each receive 1/N of the bottleneck bandwidth, as long as each flow kept at least one packet in its subqueue at all times (WFQ might change the proportions, but not the principle).

If end users could count on fair queuing in routers, the best transit choice would become something like TCP Vegas (*15.6 TCP Vegas*), which keeps a minimum number of packets in the bottleneck queue. TCP Vegas is largely unused today because it competes poorly in FIFO queues with TCP Reno (*15.6.1 TCP Vegas versus TCP Reno*), but all that would change with fair queuing. Depending on the maximum subqueue capacity, TCP Reno users in this new routing regime might receive as little as 75% of their "fair share" through the bottleneck router (*13.7 TCP and Bottleneck Link Utilization*), but the link itself would never go idle as long

as there were active TCP Vegas connections. Real-time UDP flows (perhaps carrying VoIP traffic) would also see negligible queuing delay, even in the face of large file transfers going on in parallel.

As a practical matter, a fair-queuing subqueue for each separate TCP connection is expensive. A reasonable compromise, however, might be to use stochastic fair queuing, or SFQ (*19.5.6 Stochastic Fair Queuing*). A drop policy of dropping from the fullest subqueue further penalizes TCP Reno and TCP Cubic, and favors TCP Vegas.

In *16.5 TCP Reno versus TCP Vegas* there is a graph of the relative bandwidth utilization of TCP Reno versus TCP Vegas. In that graph, TCP Reno outperforms Vegas by at least fourfold for much of the range, and achieves tenfold greater throughput with the largest queue capacities. Here, by comparison, is a chart when the bottleneck queue is changed from FIFO to SFQ:



In this graph, TCP Vegas always gets more bandwidth than TCP Reno, though for large queue capacities Reno comes within 80%. The TCP Reno connection still faces bufferbloat problems, but the TCP Vegas connection is unaffected. (In the ns-2 simulator, SFQ's total queue capacity is set with, *eg*, `Queue/SFQ set maxqueue_ 100`.)

Of course, most routers do *not* enable fair queuing, stochastic or otherwise, at least not at the level of individual flows. So Reno and Cubic still win, and Vegas remains an unfruitful choice. Fair queuing costs money, as there is more packet processing involved. Compounding this is the chicken-and-egg problem: if one ISP did implement fair queuing, end users would stick to TCP Reno and TCP Cubic because they worked best everywhere else, and the benefits of using protocols like TCP Vegas would go unrealized.

19.7 Hierarchical Queuing

Any queuing discipline with multiple input classes can participate in a **hierarchy**: the root queuing discipline is at the top of the tree, and each of its input classes is fed by a separate lower-level queuing discipline.

Hierarchical Queuing Today

Hierarchical queuing is admittedly somewhat out of fashion. One of its original goals was to support complex sharing of a single link by multiple data streams with different bandwidth and delay requirements; in today's Internet providers often instead address this with the brute-force approach of excess bandwidth capacity. That said, hierarchical queuing offers an elegant and economical alternative, and despite today's trends it is not clear if providing excess capacity will always be an option in the future.

The usual understanding of hierarchical queuing is that the non-leaf queuing-discipline nodes are "virtual": they do not store data, but only make decisions as to which subqueue to serve. The only physical output interface is at the root, and all physical queues are attached to the leaf nodes. Each non-leaf queuing-discipline node is allowed to peek at what packet, if any, its subqueues *would* dequeue if asked, but any node will only actually dequeue a packet unless that packet will immediately be forwarded up the tree to the root. This might be referred to as a **leaf-storage** hierarchy.

An immediate corollary is that leaf nodes, which now hold all the physical packets, will do all the packet dropping.

While it is possible to chain real queue objects together so as to construct composite queuing structures in which packets are *not* stored only at leaf nodes, the resultant **internal-node-storage** hierarchy will often not function as might be expected. For example, a priority-queuing node that immediately forwards each arriving packet on to its parent has forfeited any control over the priority order of resultant packet transmissions. Similarly, if we try to avoid the complexity of hierarchical fair queuing, below, by connecting separate WFQ nodes into a tree so that the internal nodes immediately forward upwards, then either packets simply pile up at the root, or else the interior links become the bottleneck. For a case where the internal-node-storage approach does work, see the end of *19.12 Hierarchical Token Bucket*.

19.7.1 Generic Hierarchical Queuing

One simple way to enable lazy dequeuing is to require the following properties of the hierarchy's component queuing disciplines:

- each node supports a `peek()` operation to return the packet it would dequeue if asked, and
- any non-leaf node can determine what packet it would itself dequeue simply by calling `peek()` on each of its child nodes

With these in place, a dequeue operation at the root node will trigger a depth-first traversal of the entire tree through `peek()` calls; eventually one leaf node will dequeue its packet and pass it up towards the root. We will call this **generic** hierarchical queuing.

Note that if we have a `peek()` operation at the leaf nodes, and the second property above for the interior nodes, we can recursively define `peek()` at every node.

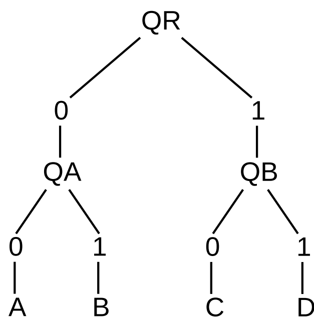
Many homogeneous queuing-discipline hierarchies support this generic mechanism. Priority queuing, as in the example below, works, because when a parent node needs to dequeue a packet it finds the highest-priority nonempty child subqueue, dequeues a packet from it, and sends it up the tree. In this case we do not even need `peek()`; all we need is `is_empty()`. Homogeneous FIFO queuing also works, assuming packets are each tagged with their arrival time, because for a node to determine which packet it would dequeue it needs only to `peek()` at its child nodes and find the earliest-arriving packet.

As an example of where generic hierarchical queuing does not work, imagine a hierarchy consisting of a queuing discipline node that sends the *smallest* packet first, fed by two FIFO child nodes. The parent would have to dequeue all packets from the child nodes to determine the smallest.

Fair queuing is not *quite* well-behaved with respect to generic hierarchical queuing. There is no problem if all packets are the same size; in this case the nodes can all implement simple round-robin selection in which they dequeue a packet from the next nonempty child node and send it immediately up the tree. Suppose, however, that to accommodate packets of different sizes all nodes use BBRR (or GPS). The problem is that without timely notification of when packets arrive at the leaf nodes, the interior nodes may not have enough information to determine the correct rates for their virtual clocks. An appropriate fair-queuing-specific modification to the generic hierarchical-queuing algorithm is below in *19.8.1 A Hierarchical Weighted Fair Queuing Algorithm*.

19.7.2 Hierarchical Examples

Our first example is of hierarchical priority queuing, though as we shall see shortly it turns out in this case that the hierarchy collapses. It may still serve to illustrate some principles, however. The basic queuing-discipline unit will be the two-input-class priority queue, with priorities 0 (high) and 1 (low). We put one of these at the root, named QR, and then put a pair of such queues (QA and QB in the diagram) at the next level down, each feeding into one of the input classes of the root queue.



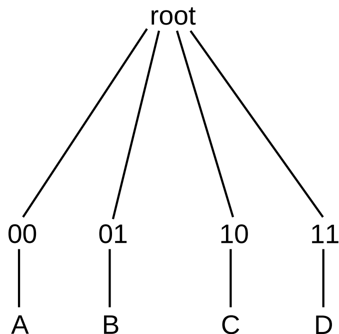
Generic Hierarchical Queuing

We now need a classifier, as with any multi-subqueue queuing discipline, to place input packets into one of the four leaf nodes labeled A, B, C and D above; these likely represent FIFO queues. Once everything is set up, the dequeuing rules are as follows:

- at the root node, we first see if packets are available in the left (high-priority) subqueue, QA. If so, we dequeue the packet from there; if not, we go on to the right subqueue QB.
- at either QA or QB, we first see if packets are available in the left input, labeled 0; if so, we dequeue from there. Otherwise we see if packets are available from the right input, labeled 1. If neither, then

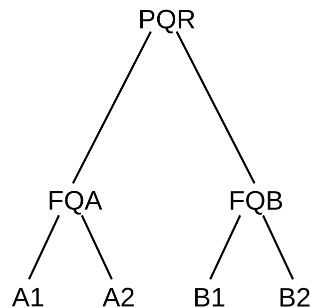
the subqueue is empty.

The catch here is that hierarchical priority queuing collapses to a single layer, with four priority levels 00, 01, 10, and 11 (from high to low):



Flattened Priority Hierarchy

For many other queuing disciplines, however, the hierarchy does not collapse. One example of this might be a root queuing discipline PQR using priority queuing, with two leaf nodes using fair queuing, FQA and FQB. (Fair queuing does not behave well as an interior node without modification, as mentioned above, but it can participate in generic hierarchical queuing just fine as a leaf node.)



Priority / Fair Queuing Hierarchy

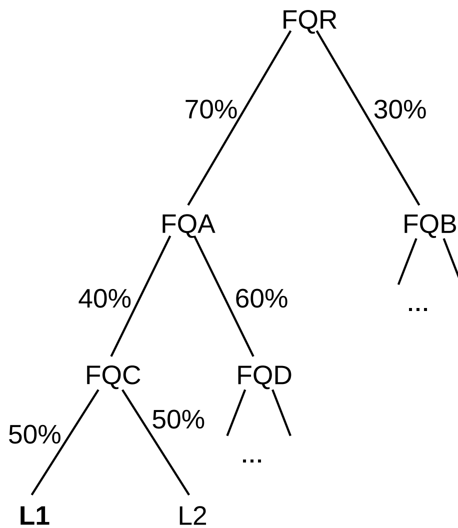
Senders A1 and A2 receive all the bandwidth, if either is active; when this is the case, B1 and B2 get nothing. If both A1 and A2 are active, they get equal shares. Only when neither is active do B1 and B2 receive anything, in which case they divide the bandwidth fairly between themselves. The root node will check on each `dequeue()` operation whether FQA is nonempty; if it is, it dequeues a packet from FQA and otherwise dequeues from FQB. Because the root does not dequeue a packet from FQA or FQB unless it is about to return it via its own `dequeue()` operation, those two child nodes do not have to decide which of their internal per-class queues to service until a packet is actually needed.

19.8 Hierarchical Weighted Fair Queuing

Hierarchical weighted fair queuing is an elegant mechanism for addressing naturally hierarchical bandwidth-allocation problems, even if it cannot be properly implemented by "generic" methods of creating queuing-discipline hierarchies. There are, fortunately, algorithms specific to hierarchical fair queuing.

The fluid model provides a convenient reference point for determining the goal of hierarchical weighted fair queuing. Each interior node divides its bandwidth according to the usual one-level WFQ mechanism: if N is an interior node, and if the ith child of N is guaranteed fraction α_i of N's bandwidth, and A is the set of N's currently active children, then WFQ on N will allocate to active child j the fraction β_j equal to α_j divided by the sum of the α_i for i in A.

Now we can determine the bandwidth allocated to each leaf node L. Suppose L is at level r (with the root at level 0), and let β_k be the fraction currently assigned to the node on the root-to-L path at level 0<k⩽r. Then L should receive service in proportion to $\beta_1 \times ... \times \beta_r$, the product of the fractions along the root-to-L path. For example, in the following hierarchical model, leaf node L1 should receive fraction 70%×40%×50% = 14% of the total bandwidth.

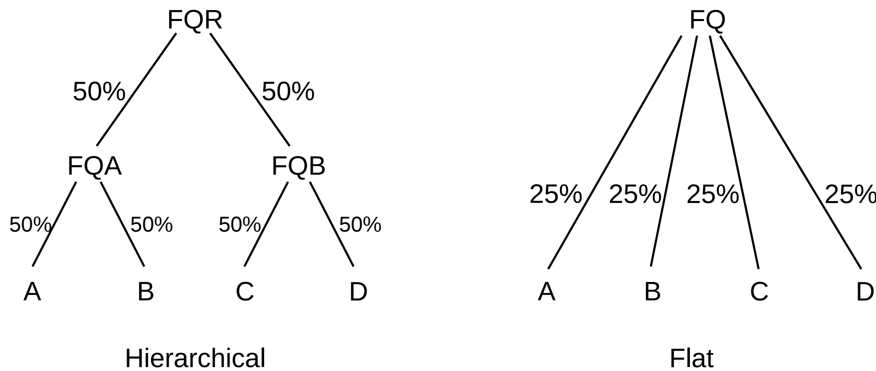


Hierarchical Weighted Fair Queuing
L1 is guaranteed 70%×40%×50% = 14%

If, however, node FQD becomes inactive, then FQA will assign 100% to FQC, in which case L1 will receive 70%×100%×50% = 35% of the bandwidth.

Alternatively, we can stick to real packets, but simplify by assuming all are the same size *and* that child allocations are all equal; in this situation we *can* implement hierarchical fair queuing via generic hierarchical queuing. Each parent node simply dequeues from its child nodes in round-robin fashion, skipping over any empty children.

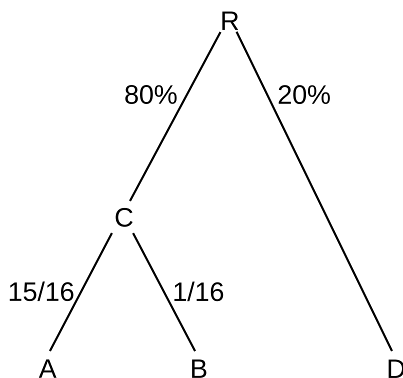
Hierarchical fair queuing – unlike priority queuing – does *not* collapse; the hierarchical queuing discipline is not equivalent to any one-level queuing discipline. Here is an example illustrating this. We form a tree of three fair queuing nodes, labeled FQR, FQA and FQB in the diagram. Each grants 50% of the bandwidth to each of its two input classes.



If all four input classes A,B,C,D are active in the hierarchical structure, then each gets 25% of the total bandwidth, just as for a flat four-input-class fair queuing structure. However, consider what happens when only A, B and C are active, and D is idle. In the flat model, A, B and C each get 33%. In the hierarchical model, however, as long as FQA and FQB are both active then each gets 50%, meaning that A and B split FQA's allocation and receive 25% each, while C gets 50%.

As an application, suppose two teams, Left and Right, are splitting a shared outbound interface; each team has contracted to receive at least 50% of the bandwidth. Each team then further subdivides its allocation among its own members, again using fair queuing. When A, B and C are the active senders, then Team Right – having active sender C – still expects to receive its contractual 50%. Team Right may in fact have decided to silence D specifically so C would receive the full allocation.

Hierarchical fair queuing can *not* be implemented by computing a finishing time for each packet at the point it arrives. By way of demonstration, consider the following example from [BZ97], with root node R and interior child node C.



If A is idle, but B and D are active, then B should receive four times the bandwidth of D. Assume for a moment that a large number of packets have arrived for each of B and D. Then B should receive the entire 80% of C's share. If transmission order can be determined by packet arrival, then the order will look something like

d_1, b_1, b_2, b_3, b_4,

d_2, b_5, b_6, b_7, b_8, ...

Now suppose A wakes up and becomes active. At that point, B goes from receiving 80% of the total bandwidth to 5% = 80% × 1/16, or from four times D's bandwidth to a quarter of it. The new b/d relative rate, *not showing A's packets*, must be

b_1, d_1, d_2, d_3, d_4,
b_2, d_5, d_6, d_7, d_8, ...

The relative frequencies of B and D packets have been reversed by the arrival of later A packets. The packet order for B and D is thus dependent on later arrivals on another queue entirely, and thus cannot be determined at the point the packets arrive.

After presenting the actual hierarchical-WFQ virtual-clock algorithm, we will return to this example in *19.8.1.2 A Hierarchical-WFQ Example*.

19.8.1 A Hierarchical Weighted Fair Queuing Algorithm

The GPS-based WFQ scheduling algorithm is *almost* suitable for use in the generic-hierarchical-queuing framework; two adjustments must be made. The first adjustment is that each non-leaf node must be notified whenever any of its formerly empty subqueues becomes active; the second adjustment is a modification to how – and more importantly when – a packet's virtual finishing time is calculated.

As for the active-subqueue notification, each non-leaf node can, of course, check after dequeuing each packet which of its subqueues is active, using the `is_empty()` operation. This, however, may be significantly after the subqueue-activating packet has arrived. A WFQ node needs to know the exact time when a subqueue becomes active both to record the GPS virtual-clock start time for the subqueue, and to know when to change the rate of its virtual clock.

Addition of this subqueue-activation notification to hierarchical queuing is straightforward. When a previously empty leaf node receives a packet, it must send a notification to its parent. Each interior node of the hierarchy must in turn forward any received subqueue-activation notification to *its* parent, provided that none of its other child subqueues were already active. If the interior node already had other active subqueues, then that node is itself active and no new notification needs to be sent. In this way, when a leaf node becomes active, the news will be propagated towards the root of the tree until either the root or an already-active interior node is reached.

To complete the hierarchical WFQ algorithm, we next describe how to modify the algorithm of *19.5.4 The GPS Model* to support subqueues of *any* type (*eg* FIFO, priority, or in our case hierarchical subtrees), provided inactive subqueues notify the WFQ parent when they become active.

19.8.1.1 WFQ with non-FIFO subqueues

Suppose we want to implement WFQ where the per-class subqueues, instead of being FIFO, can be arbitrary queuing disciplines; again, the case in which we are interested is when the subqueue represents a sub-hierarchy. The order of dequeuing from each subqueue might be changed by later arrivals (*eg* as in priority queuing), and packets in the subqueues might even disappear (as with random-drop queuing). (We *will* assume that nonempty subqueues can only become empty through a dequeuing operation; this holds in all the cases we will consider here.) The original WFQ algorithm envisioned labeling each packet P with its finishing time as follows:

$F_P = \max(VC(now), F_{prev}) + S/\alpha_i$

Clearly, this labeling of each packet upon its arrival is incompatible with subqueues that might place later-arriving packets ahead of earlier-arriving ones. The original algorithm must be modified.

It turns out, though, that a WFQ node need only calculate finishing times F_P for the packets P that are at the heads of each of its subqueues, and even that needs only to be done at the time the `dequeue()` operation is invoked. No waiting packet needs to be labeled. In fact, a packet P1 might be at the head of one subqueue, and be passed over in a `dequeue()` operation for too large an F_{P1}, only to be replaced by a different packet P2 during the next `dequeue()` call.

It is sufficient for the WFQ node to maintain, for each of its subqueues, a variable `NextStart`, representing the virtual-clock time (according to the WFQ node's own virtual clock) to be used as the start time for that subqueue's next transmitted packet. A subqueue's `NextStart` value serves as the $\max(VC(now), F_{prev})$ of the single-layer WFQ formula above. When the parent WFQ node is called upon to dequeue a packet, it calls the `peek()` operation on each of its subqueues and then calculates the finishing time F_P for the packet P *currently* at the head of each subqueue as `NextStart` + size(P)/α, where α is the bandwidth fraction assigned to the subqueue.

If a formerly inactive subqueue becomes active, it by hypothesis notifies the parent WFQ node. The parent node records the time on its virtual clock as the `NextStart` value for that subqueue. Whenever a subqueue is called upon to dequeue packet P, its `NextStart` value is updated to `NextStart` + size(P)/α, the virtual-clock finishing time for P.

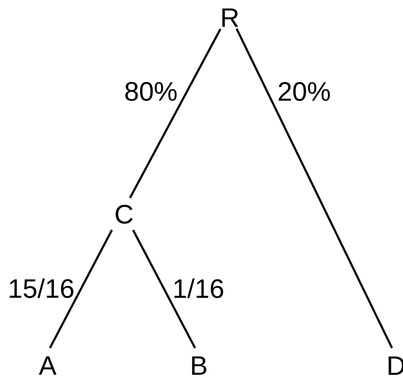
The active-subqueue notification is also exactly what is necessary for the WFQ node to maintain its virtual clock. If A is the set of active subqueues, and the ith subqueue has bandwidth share α_i, then the clock is to run at rate equal to the reciprocal of the sum of the α_i for i in A. This rate needs to be updated whenever a subqueue becomes active or inactive. In the first case, the parent node is notified by hypothesis, and the second case happens only after a `dequeue()` operation.

There is one more modification needed for non-root WFQ nodes: we must suspend their virtual clocks when they are not "transmitting", following the argument at the end of *19.5.4 The GPS Model*. Non-root nodes do not have real interfaces and do not physically transmit. However, we can say that an interior node N is **logically transmitting** when the root node R is currently transmitting a packet from leaf node L, and N lies on the path from R to L. Note that all interior nodes on the path from R to L will be logically transmitting simultaneously. For a specific non-root node N, whenever it is called upon at time T to dequeue a packet P, its virtual clock should run during the *wallclock* interval from T to T + size(P)/r, where r is the root node's physical bandwidth. The virtual finishing time of P need not have any direct correspondence to this actual finishing time T + size(P)/r. The rate of N's virtual clock in the interval from T to T + size(P)/r will depend, of course, on the number of N's active child nodes.

We remark that the `dequeue()` operation described above is relatively inefficient; each `dequeue()` operation by the root results in recursive traversal of the entire tree. There have been several attempts to improve the algorithm performance. Other algorithms have also been used; the mechanism here has been taken from *[BZ97]*.

19.8.1.2 A Hierarchical-WFQ Example

Let us consider again the example at the end of *19.8 Hierarchical Weighted Fair Queuing*:



When A is idle, B gets 4 times D's bandwidth
When A is active, B gets 1/4 D's bandwidth

Assume that all packets are of size 1 and R transmits at rate 1 packet per second. Initially, suppose FIFO leaf nodes B and D have long backlogs (*eg* b_1, b_2, b_3, ...) but A is idle. Both of R's subqueues are active, so R's virtual clock runs at the wall-clock rate. C's virtual clock is running 16× fast, though. R's `NextStart` values for both its subqueues are 0.

The finishing time assigned by R to d_i will be 5i. Whenever packet d_i reaches the head of the D queue, R's `NextStart` for D will be 5(i-1). (Although we claimed in the previous section that hierarchical WFQ nodes shouldn't need to assign finishing times beyond that for the current head packet, for FIFO subqueues this is safe.)

At least during the initial A-idle period, whenever R checks C's subqueue, if b_i is the head packet then R's `NextStart` for C will be 1.25(i-1) and the calculated virtual finishing time will be 1.25i. If ties are decided in B's favor then in the first ten seconds R will send

> b_1, b_2, b_3, b_4, $\mathbf{d}_1$, b_5, b_6, b_7, b_8, $\mathbf{d}_2$

During the ten seconds needed to send the ten packets above, all of the packets dequeued by C come from B. Having only one active subqueue puts C is in the situation of *19.5.4.4 GPS Example 2*, and so its packets' calculated finishing times will exactly match C's virtual-clock value at the point of actual finish. C dequeues eight packets, so its virtual clock runs for only those 8 of the 10 seconds during which one of the b_i is being transmitted. As a result, packet b_i finishes at time 16i by C's virtual clock. At T=10, C's virtual clock is 8×16 = 128.

Now, at T=10, as the last of the ten packets above completes transmission, let subqueue A become backlogged with a_1, a_2, a_3, *etc*. C will assign a finishing time of 128 + 1.0667i to a_i (1.0667 = 16/15); C has already assigned a virtual finishing time of 9×16=144 to b_9. None of the virtual finishing times assigned by C to the remaining b_i will change.

At this point the virtual finishing times for C's packets are as follows:

packet	C finishing time	R finishing time
a_1	128 + 1.0667	10 + 1.25
a_2	128 + 2×1.0667	10 + 2.50
a_3	128 + 3×1.0667	10 + 3.75
a_4	128 + 4×1.0667	10 + 5
...		
a_{15}	128 + 15×1.0667 = 144	10 + 15×1.25
b_9	144	10 + 16×1.25 = 30

During the time the 16 packets in the table above are sent from C, R will also send four of D's packets, for a total of 20.

The virtual finishing times assigned by C to b_9 and b_{10} have *not* changed, but note that the virtual finishing times *assigned to these packets by R* are now very different from what they would have been had A remained idle. With A idle, these finishing times would have been F_9 = 11.25 and F_{10} = 12.50, *etc*. Now, with A active, it is a_1 and a_2 that finish at 11.25 and 12.50; b_9 will now be assigned by R a finishing time of 30 and b_{10} will be assigned a finishing time of 50. R is still assigning successive finishing times at increments of 1.25 to packets dequeued by C, but B's contributions to this stream of packets have been bumped far back.

R's assignments of virtual finishing times to the d_i are immutable, as are C's assignments of virtual finishing times, but R can *not* assign a final virtual finishing time to any of C's packets (that is, A's or B's) until the packet has reached the head of C's queue. R assigns finishing times to C's packets in the order they are dequeued, and until a packet is dequeued by C it is subject to potential preemption.

19.9 Token Bucket Filters

Token-bucket filters provide an alternative to fair queuing for providing a traffic allocation to each of several groups. The main practical difference between fair queuing and token bucket is that if one sender is idle, fair queuing distributes that sender's bandwidth among the other senders. Token bucket does not: the bandwidth a sender is allocated is a **bandwidth cap**.

Suppose the outbound bandwidth is 4 packets/ms and we want to allocate to one particular sender, A, a bandwidth of 1 packet/ms. We could use fair queuing and give sender A a bandwidth fraction of 25%, but suppose we do not want A ever to get more bandwidth than 1 packet/ms. We might do this, for example, because A is paying a reduced rate, and any excess available bandwidth is to be divided among the *other* senders.

The catch is that we want the flexibility to allow A's packets to arrive at irregular intervals. We could simply wait 1 ms after each of A's packets begins transmission, before the next can begin, but this may be too strict. Suppose A has been dutifully submitting packets at 1ms intervals and then the packet that was supposed to arrive at T=6ms instead arrives at T=6.5. If the following packet then arrives on time at T=7, does this mean it should now be held until T=7.5, *etc*? Or do we allow A to send one late packet at T=6.5 and the next at T=7, on the theory that the *average* rate is still 1 packet/ms?

The latter option is generally what we want, and the solution is to define A's quota in terms of a **token-bucket specification**, which allows for specification of both an average rate and also a burst capacity.

If a packet does not meet the token-bucket specification, it is **non-compliant**; we can do any of the following things:

- **delay** the packet until the bucket is ready
- **drop** the packet
- **mark** the packet as non-compliant

The first option here is often called **shaping**; the second, more authoritarian option is sometimes known as **policing**.

Another use for token-bucket specifications is as a theoretical traffic description, rather than a rule to be enforced; in this context compliance is a non-issue.

A token-bucket filter can be thought of as a queuing discipline, with an underlying FIFO queue. If non-compliant packets are delayed, it is non-work-conserving. Dropping non-compliant packets can be viewed as an alternative to tail-drop. The queuing-discipline definition above in *19.4 Queuing Disciplines* does not provide for marking packets, but this is a straightforward extension.

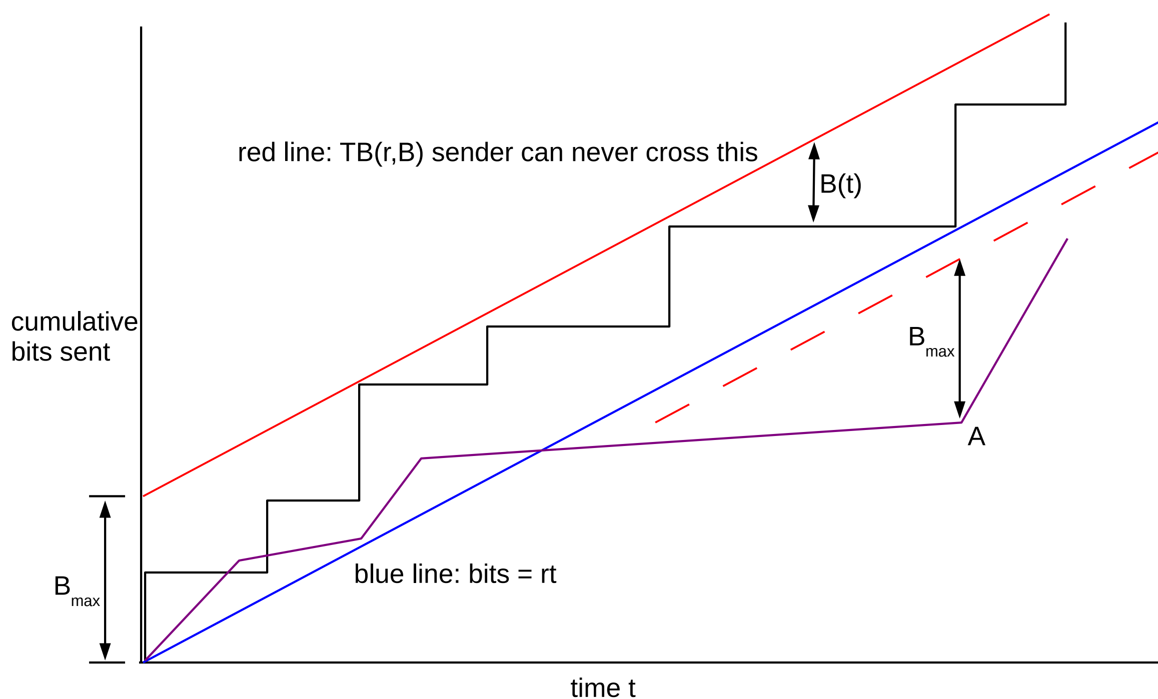
19.9.1 Token Bucket Definition

The idea behind a token bucket is that there is a notional bucket somewhere, being filled at a steady rate with tokens (or, if more divisibility is needed, with fluid); any overflow from the bucket is discarded. To send a packet, we need to be able to take one token from the bucket; if the bucket is empty then the packet is non-compliant and must suffer special treatment as above. If the bucket is full, however, then the sender may send a **burst** of packets corresponding to the bucket capacity (at which point the bucket will be empty).

A common variation is requiring one token per byte rather than per packet, with the fill rate correspondingly scaled; this allows packet size to be taken into account.

More precisely, a token-bucket specification TB(r,B_{max}) includes a **token fill rate** of r tokens/sec, representing the rate at which the bucket fills with tokens, and also a **bucket capacity** (or depth) $B_{max}>0$. The bucket fills at the rate specified, subject to a maximum of B_{max}; we will denote the current capacity by B, or by B(t) if we need to specify the time. In order for a packet of size S (possibly S=1 for counting size in units of whole packets) to be within the specification, the bucket must have at least S tokens; that is, $B \geqslant S$. Otherwise the packet is non-compliant. When the packet is sent, S tokens are removed from the bucket, that is, B=B–S. It is possible for the packets of a given flow all to be compliant with a given token-bucket specification at one point (*eg* one router) in the network but not at another point; this can happen, for example, if more than B_{max} packets pile up at a downstream router due to momentary congestion.

The following graph is a visual representation of a token-bucket constraint. The black and purple curves plotted are of cumulative bits sent as a function of time, that is, bits(t). When bits(t) is horizontal, the sender is idle.



Two token-bucket-compliant senders are shown, one black and one purple.
The black sender sends in discrete packets, and the graph is a sequence of steps;
the purple sender sends continuously at different rates on different intervals.
The blue line represents a sender sending steadily at rate r;
the solid red line is the "bucket limit" which a compliant sender may not cross.
The purple sender, by crossing below the blue line, cannot go back to the solid red line.
In fact the purple line cannot cross the dashed red line after falling "behind" at point A.

The blue line represents a sender sending linearly at the rate r, with no burstiness. At vertical distance B_{max} above the blue line is the red line. Graphs for compliant senders cannot cross this, because that would entail a burst of more than B_{max} above the blue line; we give a more formal argument below. As a sender's graph approaches the red line, the sender's current bucket contents decreases; the instantaneous bucket contents for the black sender is shown at one point as B(t).

The purple sender has fallen below the blue line at one point; as a result, it can never catch up. In fact, after passing through the vertex at point A the purple graph can never cross the dashed red line. A proof is in *19.11 Token Bucket Queue Utilization*, following some numeric token-bucket examples that illustrate how a token-bucket filter works.

Satellite Token Bucket

When I first got satellite Internet, my service was limited by a token-bucket filter with rate 56 Kbps and bucket 300 megabytes. When the bucket emptied, it took 12 hours to refill. The idea was that someone could use the Internet intensely but relatively briefly; satellite access is expensive. Within a year, the provider switched to a flat 300 MB cap per day; the token-bucket rule was apparently not well understood by customers.

If a packet arrives when there are not enough tokens in the bucket to send it, then as indicated above there are

three options. The sender can engage in shaping, making the packet wait until sufficient tokens accumulate. The sender can engage in policing, dropping the packet. Or the sender can send the packet immediately but mark it as noncompliant.

One common strategy is to send noncompliant packets – as marked in the third option above – with lower priority. Alternatively, marked packets may face a greater chance of being dropped by some downstream router. In ATM networks (*3.5 Asynchronous Transfer Mode: ATM*) the cell-loss priority bit is often used to mark noncompliant packets.

Token-bucket specifications supply a framework for making decisions about **admission control**: a router can decide whether to accept a new connection (or whether to accept the connection's quality-of-service request) based on the requested rate and bucket (queue) requirements.

Token-bucket specifications are the mirror-image equivalent to **leaky-bucket specifications**, in which the fluid leaks out of the leaky bucket at rate r and to send a packet we must add S units without overflowing. The two forms are completely equivalent.

So far we have been using token-bucket specifications to describe traffic; *eg* traffic *arriving* at a router. It is also possible to use token buckets to describe the router itself; in this setting, the leaky-bucket formulation may be clearer. The router's queue represents the bucket, and the router's packet transmissions represent tokens leaking out of the bucket. Arriving packets are added to the bucket; a bucket overflow represents lost packets. We will not pursue this interpretation further.

19.9.2 Token-Bucket Examples

Suppose the token-bucket specification is **TB(1/3 packet/ms, 4 packets)**, and packets arrive at the following times, with the bucket initially full:

> 0, 0, 0, 2, 3, 6, 9, 12

After all the T=0 packets are processed, the bucket holds 1 token. By the time the fourth packet arrives at T=2, the bucket volume has risen to 1 2/3; it immediately drops to 2/3 when packet 4 is sent. By T=3, the bucket volume has reached 1 and the fifth packet can be sent. The bucket is now empty, but fortunately the remaining packets arrive at 3-ms intervals and can all be sent.

In the next set of packet arrival times, again with TB(1/3,4), we have three bursts of four packets each.

> 0, 0, 0, 0, 12, 12, 12, 12, 24, 24, 24, 24

Each burst empties the bucket, which then takes 12 ms to refill. All packets are compliant.

In the following set of packet arrival times, still with TB(1/3,4), the burst of four packets at T=0 drains the bucket. At T=3 the bucket size has increased back to 1, allowing the packet that arrives then to be sent but also draining the bucket again.

> 0, 0, 0, 0, 3, 6, 12, 12

At T=6 the same thing happens. From T=6 to T=12 the bucket contents rise from 0 to 2, allowing the two packets arriving at T=12 to be sent.

Finally, suppose packets arrive at the following times at our TB(1/3,4) filter.

> 0, 1, 2, 3, 4, 5

Just after T=0 the bucket size is 3; just before T=1 it is 3 1/3.
Just after T=1 the bucket size is 2 1/3; just before T=2 it is 2 2/3
Just after T=2 the bucket size is 1 2/3; just before T=3 it is 2
Just after T=3 the bucket size is 1; just before T=4 it is 1 1/3
Just after T=4 the bucket size is 1/3; just before T=5 it is 2/3
At T=5 the bucket size is 2/3 and the arriving packet is **noncompliant**.

We can also represent this in tabular form as follows; note that for the noncompliant packet the bucket is not decremented.

packet arrival	0	1	2	3	4	5
bucket just before	4	3 1/3	2 2/3	2	1 1/3	2/3
bucket just after	3	2 1/3	1 2/3	1	1/3	2/3

19.9.3 Multiple Token Buckets

It often makes sense to require that a sender comply with two (or more) separate token-bucket specifications. We can think of these being applied to the traffic sequentially. Often one filter will specify a peak rate, with a small bucket size, and the other will specify an average rate, with a larger bucket size. Consider, for example, the following pair:

1. TB(1 packet/ms,   1.5 packets)
2. TB(1/5 packet/ms, 6 packets)

The first specification, meant to apply to the peak rate, mandates 1 ms on average between packets, but packets can be only 0.5 ms early without being noncompliant. The second specification, meant to apply over the longer term, states that *on average* there will be 5 ms between packets, subject to a burst of 6. The following is compliant, assuming both buckets are initially full.

0, 1, 2.5, 3, 4, 5, 6, 10, 15, 20

The first seven packets arrive at 1 ms intervals (the rate of the first filter) except for the packet that arrived at T=2.5 instead of T=2. The sender was allowed to send again at T=3 instead of waiting until T=3.5 because the bucket size in the first filter was 1.5 instead of 1.0. Here are the packet arrivals with the current size of each bucket **at the time of packet arrival**, just before the bucket is decremented. At T=2.0, the filter2 bucket would be 4.4.

arrival:	T=0	T=1	T=2.5	T=3	T=4	T=5	T=6	T=10	T=15	T=20
Filter 1:	1.5	1.5	1.5	1.0	1.0	1.0	1.0	1.5	1.5	1.5
Filter 2:	6	5.2	4.5	3.6	2.8	2	1.2	1.0	1.0	1.0

If we move up each packet in time to the first point when both buckets have reached 1.0, we get the **fastest compliant sequence** for this pair of filters. This is the sequence generated by a token-bucket *shaper* when there is a steady backlog of packets and each is sent as soon as the bucket capacity (or capacities, when applicable) is full enough to allow sending. After T=0, the filter1 bucket returns to capacity 1.0 at T=0.5. Continuing, the filter1 bucket allows for additional transmissions at T=1.5, T=2.5, T=3.5, T=4.5 and T=5.5. At this point filter2 becomes the limiting factor; its bucket is at 0.1 after the T=5.5 packet is sent and does not return to 1.0 until T=10.0. We get the following:

arrival:	T=0	T=0.5	T=1.5	T=2.5	T=3.5	T=4.5	T=5.5	T=10	T=15	T=20
Filter 1:	1.5	1.0	1.0	1.0	1.0	1.0	1.0	1.5	1.5	1.5
Filter 2:	6	5.1	4.3	3.5	2.7	1.9	1.1	1.0	1.0	1.0

19.9.4 GCRA

Another formulation of the token-bucket specifications is the **Generalized Cell Rate Algorithm**, or GCRA; this formulation is frequently used in classification of ATM traffic. A GCRA specification takes two parameters, a mean packet spacing time T, and an early-arrival allowance τ. For each packet we compute a *theoretical arrival time*, tat, initially zero. A packet may arrive earlier by amount at most τ. Specifically, if t is the time of actual arrival, we have two cases:

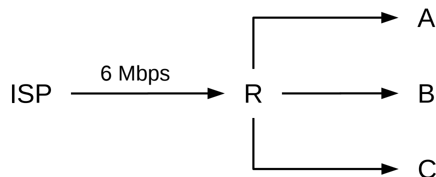
1. t $\geqslant$ tat–τ: the packet is compliant, and we update tat to max(t,tat) + T
2. t < tat–τ: the packet is too early and is noncompliant; tat is unchanged.

A flow satisfying GCRA(T,τ) is equivalent to a token-bucket specification with rate 1/T packets/unit time, and bucket size (τ+1)/T; tat represents the time the bucket would next be full. The time to fill an empty bucket is τ+1; if the bucket becomes full at time tat then, working backwards, it would contain enough to send one packet at time tat–τ.

For traffic flows with a more-or-less constant rate, τ represents the time by which one packet can be late without permanently falling behind its regular 1/T rate. The GCRA formulation is sometimes more convenient than the token-bucket formulation, particularly when τ<T.

19.10 Applications of Token Bucket

Unlike fair queuing, token-bucket filtering can be implemented at the downstream end of a link, though possibly with results not quite in agreement with expectations. Let us return to the final scenario of *19.6 Applications of Fair Queuing*:



While fair queuing cannot be applied at R to enforce equal shares to A, B and C, we *can* implement a token-bucket filter at R that limits each of A, B and C to 2 Mbps.

There are two drawbacks. First, the filter is not work-conserving: if A is idle, B and C will still only receive 2 Mbps. Second, in the absence of feedback there is no guarantee that limiting the traffic at R will eventually result in reduced utilization of the ISP⟶R link. While this is true for TCP traffic, due to the self-clocking property, it is conceivable that a sender D somewhere is trying to send 8 Mbps of real-time UDP traffic to A, via ISP and R. Three-quarters of the traffic would then fail to be compliant, and might be dropped by R, but unless D gets feedback from A that not much of the traffic is getting through, and that it should reduce its sending rate, the token-bucket filter at R will not achieve what we want. Most protocols *do* provide this kind of feedback, but not all.

19.10.1 Guaranteeing VoIP Bandwidth

As a particular instance of the previous situation, suppose we have an Internet connection from our ISP and want to begin using VoIP for telephony. We would like to reserve something like 64 Kbps of bandwidth for VoIP (plus room for headers), so that large downloads do not degrade voice quality. We can easily do this for the upstream direction, either with fair or priority queuing; priority queuing is an option here as the total VoIP traffic is limited by the number of lines.

However, the downstream direction may be more of a problem, if we are unable to enlist the ISP to apply fair queuing at the upstream end. As we argued in *19.6 Applications of Fair Queuing*, fair queuing at the downstream end has no effect.

Token-bucket at the downstream end might be an option. If we knew that the total link bandwidth was 500 bits/ms we might reserve 100 bits/ms, say, for VoIP traffic by limiting further delivery of non-VoIP traffic to 400 bits/ms. This is indeed sometimes done. Unfortunately, we encounter three problems. The first is that if no VoIP traffic is flowing then we probably do not want the 400 bits/ms cap on other traffic; we might arrange this by applying the cap only when the phone is in use, or by setting aside a small enough bandwidth fraction that it does not have a material affect on overall bulk bandwidth. The second problem is the (remote) possibility discussed in the previous example that the sender might keep sending anyway, at 500 bits/ms; our downstream router can throw away as many bits as it wants but the link itself will still be saturated. Finally, it is often quite difficult to determine exactly what the bandwidth of a particular Internet connection *is*. Especially if, as is often the case, it is shared, or configured to change with time, or subject to time-varying caps by the ISP.

If the competing downstream traffic is TCP, we theoretically could reduce the rate of upstream ACKs until the downstream VoIP traffic no longer encounters excessive losses, but that is largely hypothetical and likely to respond slowly.

Fortunately, typical VoIP bandwidth needs are low enough that one can often muddle through without providing any quality-of-service guarantees at all. This remains, however, a good example of the difficulties often faced by real-time traffic.

19.11 Token Bucket Queue Utilization

Suppose traffic meeting token-bucket specification TB(r,B_{max}) arrives at a router R, with no competition from other traffic. The bucket fill rate r corresponds to the minimum outbound link bandwidth needed by R to guarantee that the traffic does not build up; we do not want traffic on average arriving faster than it can depart.

Intuitively, the bucket size B_{max} corresponds to the amount of **queue space** at R that the flow can consume. To make this more precise, we will argue that, if the output rate from R is at least r, then the number of untransmitted bits stored at R is never more than B_{max}.

To show this more formally, we start by proving the "red line lemma" implicit in the discussion of the graph in *19.9.1 Token Bucket Definition* above, that the sender can never cross the red line. Specifically, assume the flow satisfies TB(r,B_{max}) and has a full bucket at time t=0. Let bits(t) be the cumulative number of bits sent (packetized or not) by time t. The blue line is the graph bits = rt and the red line is the graph bits = rt + B_{max}; we show the following:

$$\text{bits}(t) \le rt + B_{max}$$

We first prove this so long as the graph is above the blue line; that is, bits(t) >= rt. We claim that the right-hand side minus the left-hand side above, rt + B_{max} - bits(t), represents the volume B(t) of fluid (or tokens) in the bucket. Equating and rearranging slightly, we need to show B(t) + bits(t) - rt is always equal to B_{max}. This is true at t=0 when bits(t) = rt = 0 and the bucket is full. We next establish that its rate of change is also 0, and so it is constant.

While the bucket is not full, B(t) is always being filled at rate r. Correspondingly, rt is increasing at rate r, so B(t) - rt is not affected by the fill rate. Similarly, B(t) is being reduced at exactly the rate bits(t) is increasing. If we use the packet formulation, then when a packet arrives B(t) is reduced by the packet size and bits(t) increases by exactly the same amount.

The Calculus Version

For readers familiar with calculus it may help to note dB(t)/dt = r - bits'(t), at least if we assume, say, a fluid bit-arrival model where bits(t) is differentiable. That is, the bucket volume B(t) increases at rate r and also decreases at a rate equal to that of arriving data. Therefore, the rate of change of B(t) + bits(t) - rt is just r - bits'(t) + bits'(t) - r = 0.

This does not quite apply when bits(t) falls below the blue line. However, we have nothing to prove then. If bits(t) has a later interval above the blue line, starting at time t_1, we can reapply the argument above re-starting the clock and the bits counter at t=t_1.

In fact, we can argue that whenever the bits(t) graph passes through a point below the blue line, such as point A in the diagram above, then bits(t) cannot in the future climb above the new red line (the dashed red line in the diagram) B_{max} units above point A.

19.11.1 Token Bucket Through One Router

We now return to the claim about accumulation at a router R with outbound flow at least r; as before, let bits(t) represent the cumulative amount of data received. As long as bits(t) is above the blue line, the router can continuously transmit at rate r and the net number of bits held within the router is bits(r) - rt. By the argument above, this is bounded by B_{max}. If bits(t) falls below the blue line, the router's queue is empty and the router can transmit incoming data at least as fast as it is arriving.

While R can never be holding more than B_{max} bytes, at the instant just before a packet finishes transmission it can have B_{max} bytes in the queue, plus the currently transmitting packet still taking up an entire buffer. As a practical matter, then, we may need space equal to B_{max} plus one packet.

While a token-bucket specification does not include a delay bound specifically, we can compute an upper bound to the queuing delay at a router R as B_{max}/r; this is the time it takes for one full bucket's worth of packets to be transmitted.

If we have N flows each individually satisfying TB(r,B), then the collective traffic will satisfy TB(Nr, NB) (see exercise 12). However, a bucket size of NB will be needed only when all N individual flows have their bursts "gang up" at a particular instant. Often it is possible to take advantage of theoretical or empirical statistical information and conclude that the collective traffic "most of the time" meets a token-bucket specification TB(Nr, B_N) for B_N significantly less than NB.

19.11.2 Token Bucket Through Multiple Routers

If we have a single TB(r,B_{max}) flow through N routers, however, the queuing delay is *not* larger than for a single router, again assuming no competition. More specifically, assume that the traffic flow arrives at router R_1 satisfying TB(r,B), and passes in turn through R_1 to R_N. Each router R_i has an outbound bandwidth at least as large as r. Then the total queuing delay through all N routers remains B_{max}/r. If the packets pile up to the maximum size B_{max}, they only do so once.

To prove this we compare the TB sequence of packets with the same sequence of packets sent at a steady rate r through the same series of routers. If the last bit of packet k is the Nth bit since we began, then for the steady stream we send packet k at time N/r. We assume the link rates are all reduced to r.

Let t=0 represent the time we start counting bits. For every n, we established above that the nth bit of the TB packet flow can be transmitted at most B_{max}/r seconds ahead of the nth steady-stream bit, which is sent at time n/r. The steady-stream packets do not encounter queuing delays at all, as each router has always finished the previous one. The TB packets can each arrive no later than the steady-stream packets, as they were sent earlier and they cannot cross. Therefore, the maximum delay faced by any TB packet is B_{max}/r, exactly as for traffic through a single router.

19.11.3 Delay Constraints

If a traffic flow arriving at a router R is compliant for token-bucket specification TB(r,B), then as we showed above the amount of R's queue space used by the flow will be bounded by B so long as R can devote at least rate r to the flow's traffic.

Now let us add a **real-time delay constraint**: suppose that R is not to be allowed to delay any of the flow's packets by more than time D. For the time being, assume that there is no other traffic at R. We now need to make sure that R has sufficient bandwidth to forward a bucketful of size B within the time interval D. To send a burst of size B in time D, bandwidth B/D is needed. Therefore, to satisfy the real-time constraint, R needs outbound bandwidth

s = max(r, B/D)

Example 1: suppose the traffic specification is TB(1/3, 10), where the rate is in (equal-sized) packets/µsec, and D is 40 µsec. Then B/D is 1/4 packets/µsec, and the necessary outbound bandwidth s is simply r=1/3.

Example 2: now suppose in the previous example that the delay limit D is 20 µsec. In this case, we need s = B/D = 1/2 packets/µsec.

If there *is* other traffic, the delay constraint still holds, provided s represents the bandwidth allocated by R to the flow, and the flow's packets receive priority service at R, and we first subtract the largest-packet delay as in *5.3.2 Packet Size and Real-Time Traffic*.

Calculations of this sort often play a role in a router's decision on whether to accept a **reservation** for an additional TB(r,B) flow with associated delay constraint.

19.12 Hierarchical Token Bucket

Token-bucket filters can also be used to form a hierarchy, as in *19.7.1 Generic Hierarchical Queuing*. In this section we will assume that token bucket is used only for shaping; that is, delaying packets until the

bucket has sufficiently filled. As usual, packets will remain in the leaf FIFO queues until they are ready to be transmitted.

Central to the hierarchy is the conceptual time each internal token-bucket node **releases** its next packet; that is, becomes able to inform its parent node (when asked) that it has a packet ready to send, even if the packet physically remains waiting in one of the leaf queues. If a node N is informed by a child node that a packet has been released and N's bucket has sufficient capacity, then N releases the packet in turn to its parent immediately; otherwise N waits until its bucket fills sufficiently to make the packet compliant. When a packet arrives at a leaf node, it will be progressively released by each node along the path to the root; when it is released by the root node it can be sent.

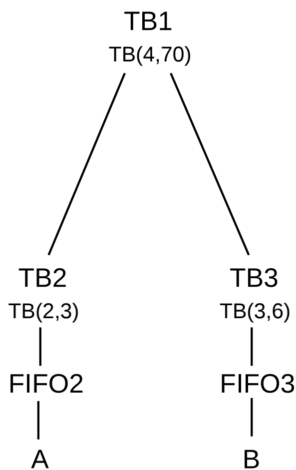
To make token-bucket filters classful, we will assume that each node may have multiple input subqueues, but treats these as if they were consolidated into a single FIFO subqueue. That is, the node releases packets to its parent in the order they were released to the node by its children.

Leaf nodes can mark each packet with its release time at the moment of arrival. Interior nodes may only be able to determine their release times for packets that have been released by their child nodes.

It is now straightforward to define the `peek()` operation of *19.7.1 Generic Hierarchical Queuing*: a node looks at the set of packets it has released and returns the one with the earliest release time.

In a token-bucket hierarchy it makes a sense to say that two child flows have bucket sizes of 200 and 300, respectively, while the combined flow is to be limited to a bucket size of 400.

The following diagram illustrates an example of a token-bucket hierarchy. The three token-bucket filters TB1, TB2 and TB3 have rates in packets/ms and bucket sizes in packets.



Hierarchical Token Bucket

If TB1's rate is, as here, less than the sum of its child rates, then as long as its children always have packets ready to send, the children will receive bandwidth in proportion to their token bucket rates. In the example above, TB1's rate is 4 packets/ms and yet the sum of the rates of its children is 5 packets/ms. Each child will therefore receive 4/5 its promised rate: TB2 will send at a rate of $2\times(4/5)$ packets/ms while TB3 will send at rate of $3\times(4/5)$ packets/ms.

To see this, assume FIFO2 and FIFO3 remain nonempty for a period long enough for their buckets to empty. TB2 and TB3 will then each release packets to TB1 at their respective rates of 2 packets/ms and 3

packets/ms. In the following sequence of release times to TB1, we assume TB3 starts at T=0 and TB2 at T=0.01, to avoid ties. Packets from A released by TB2 are shown in italic:

0, *0.01*, .33, *0.51*, .67, 1.0, *1.01*, 1.33, *1.51*, 1.67, 2.0, *2.01*

They will be dequeued by TB1 at 4 packets/ms, once TB1's bucket is empty. In the long run, TB3 has released three packets into this sequence for every two of TB2's, so sender B will receive 3/5 of the dequeuings, and thus 3/5 of the 4 packet/ms root bandwidth.

We can also have each token-bucket node physically forward released packets to FIFO queues attached to each parent node; we called this an internal-storage hierarchy in *19.7 Hierarchical Queuing*. In this particular case, the leaf-storage and internal-storage mechanisms function identically, provided the internal links are infinitely fast and the internal queues infinitely large. See exercise 18.

There is no point in having a node with a bucket larger than the sum of its child buckets and also a rate larger than the sum of its child rates. In the example above, in which the sum of the child rates exceeds the parent rates, A would be able to send at a sustained rate of 2 packets/ms provided B sends at only 2 packets/ms as well; reducing the child rates to 2×(4/5) and 3×(4/5) packets/ms respectively is not equivalent. If a node's rate is larger than the sum of the child rates, then it will be able to handle the child traffic without delay once the child buckets have emptied. Before that, though, the parent bucket may be the limiting factor.

19.13 Fair Queuing / Token Bucket combinations

At first glance, combining fair queuing with token bucket might seem improbable: the goal of fair queuing is to be *work-conserving*, allowing the bandwidth assigned to an idle input class to be divided among the active input classes, and the goal of token bucket is generally to limit a class to its token-bucket-defined maximum transmission rate. The usual approach to a hierarchy-based synthesis is to allow the administrator to decide, at each node of the hierarchy, whether or not the node can "borrow" (without repayment) bandwidth from inactive siblings. If it can, the set of siblings with mutual borrowing privileges resembles a fair-queuing scheduler; if not, the node is more like a token-bucket scheduler.

19.13.1 CBQ

CBQ was introduced in *[CJ91]* and analyzed in *[FJ95]*. It did not actually use the token-bucket mechanism, but instead implemented shaping by keeping track of the average idle time (more precisely, non-transmitting time) for a given input class. Input classes that tried to send too much were restricted, unless the node was permitted to "borrow" bandwidth from a sibling. When an input class sent less than it was allowed, its average utilization would fall; if a burst arrived then it would take some time for the average to "catch up" and thus the node could briefly send faster than its assigned rate. However, the size of the "bucket" could be controlled only indirectly.

19.13.2 Linux htb

The linux **htb** queuing discipline, part of the Traffic Control (tc) system, allows the same general functionality of CBQ, but replaces the average-idle calculations with token-bucket filters. This permits more direct control of burst sizes, and also avoids some technical timing issues that CBQ users had to watch out for. For

the sake of efficiency, htb uses the quantum algorithm for fair queuing; as noted in *19.5.5 The Quantum Algorithm*, this means less precise control over packet delay.

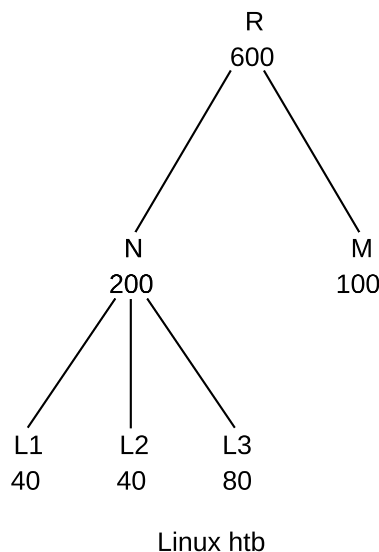
It is not uncommon to arrange for htb to apply token-bucket shaping only at the leaf nodes, and to configure the interior nodes do fair queuing only.

Each node in the tree has the following attributes:

- its guaranteed rate, r, corresponding to the token-bucket rate
- its burst allowance B, corresponding to the bucket size
- its ceiling rate r_{ceil}; nodes never send faster than this

In many cases r_{ceil} may simply be the output rate of the root node. For interior nodes, if the requested ceiling rate is less than the sum of the child rates, the actual ceiling rate is adjusted upwards to match the sum of the child rates; this means that interior nodes cannot do rate-limiting.

The most important attribute of each node is its guaranteed rate. The requested rate at each node should be at least as large as the sum of the child rates. In the following diagram, all rates are in Kbps; burst allowances are not shown. We will assume the root guaranteed rate, 600 Kbps, is also its ceiling rate.



Linux htb

Packets are marked **green**, **yellow** or **red** depending on their situation. Red packets are those that must wait; eventually they will turn yellow and then green.

Packets are considered **green** if they are now compliant (perhaps after waiting earlier) for one of the leaf token-bucket nodes; green packets are sent as soon as possible.

After L1, L2 and L3 have each emptied their buckets, they will not exhaust node N's rate. Similarly, after N and M have emptied their buckets they will use only half of R's rate. Nodes are allowed to "borrow" bandwidth – without payback – from their parent's rates; packets benefiting from such borrowed bandwidth are marked **yellow**, and may also be sent immediately if no green packets are waiting. Borrowing is always in proportion to a node's guaranteed rate, in the manner of fair queuing. That is, the guaranteed rates of the child nodes are treated as unnormalized fair-queuing weights; normalized weight fractions are obtained by dividing by their total. N above would have normalized weight fraction 200/(200+100) = 2/3.

If L1, L2 and L3 engage in borrowing from N, and each has traffic to send, then each gets a total bandwidth of 50, 50 and 100 Kbps, respectively. If L3 is idle, then L1 and L2 each would get 100 Kbps. If N and M borrow in turn from R, they each can send at 400 and 200 Kbps respectively, in which case L1, L2 and L3 (again assuming all are active) get 100, 100 and 200 Kbps. If M elects not do do any borrowing, because it has nothing to send, then N will get 600 Kbps and L1, L2 and L3 will get 150, 150 and 300.

If fair-queuing behavior is not desired, we can set r_{ceil} = r so that a node can never send faster than its guaranteed rate. This allows htb to model the token-bucket-only hierarchy of *19.12 Hierarchical Token Bucket*.

A working example of htb, with one parent and two child nodes, is constructed in *18.8 Linux Traffic Control (tc)*.

19.13.3 Parekh-Gallager Theorem

As a final example relating token-bucket specifications and fair queuing, we present the Parekh-Gallager Theorem, which provides a precise queuing-delay bound on traffic that enters a network meeting a token-bucket specification TB(r,B) and which has a guaranteed weighted-fair-queuing fraction through each router along the path.

Specifically, let us assume that the traffic travels from sender A to destination B through N routers R_1 ... R_N. The output rate of the ith router R_i is r_i, of which our flow is guaranteed rate $f_i \leqslant r_i$. Let f = min $\{f_i | i<N\}$. Suppose the maximum packet size for packets in our flow is S, and the maximum packet size including competing traffic is S_{max}. Then the total delay encountered by the flow's packets is bounded by the sum of the following:

1. propagation delay (total single-bit delay along all N+1 links)
2. B/f
3. The sum from 1 to N of S/f_i
4. The sum from 1 to N of S_{max}/r_i

The second term B/f represents the queuing delay introduced by a single burst of size B; we showed in *19.11.2 Token Bucket Through Multiple Routers* that this delay bound applied regardless of the number of routers.

The third term represents the total store-and-forward delay at each router for packets belonging to our flow under GPS; the delay at R_i is S/f_i.

The final term represents the degree to which fair-queuing may delay a packet beyond the theoretical GPS time expressed in the third term. If the routers were to use GPS, then the first three terms above would bound the packet delay; we established in *19.5.4.7 Finishing-Order Bound* that router R_i may introduce an additional delay above and beyond the GPS delay of at most S_{max}/r_i.

19.14 Epilog

If we want to use 100% the outbound bandwidth, but divide it among several senders according to a predetermined ratio, fair queuing is the tool to use. If we want to impose an absolute rather than a relative cap on traffic, token bucket is appropriate.

Fair queuing has applications to the routing of ordinary packets; for example, if routers implement fair queuing on a per-connection basis, then TCP senders will have no incentive to maximize queue utilization and TCP Reno will lose its competitive advantage.

It is for real-time traffic, however, that queuing disciplines such as fair queuing, token bucket and even priority queuing come into their own as fundamental building blocks. These tools allow us to guarantee a bandwidth fraction to VoIP traffic, or to allow such traffic to be sent with minimal delay. In the next chapter *20 Quality of Service* we will encounter fair queuing and token-bucket specifications repeatedly.

19.15 Exercises

1. Suppose a router uses fair queuing with three input classes, and uses the quantum algorithm of *19.5.2 Different Packet Sizes and Virtual Finishing Times*. The first class sends packets of size 900 bytes, the second sends packets of 400 bytes, and the third sends packets of 200 bytes. List what would be sent by each flow in each of the first five rounds.

2. Suppose we attempt to simulate BBRR as follows with the following strategy we will call SBBRR. Each subqueue has a bit-position marker that advances by one bit for each bit we have available to send from that queue. If three queues are active, then in the time it takes us to send three bits, each marker would advance by one bit. When all the bits of a packet have been ticked through, the packet is sent.

   (a). Explain why this is not the same as BBRR fair queuing (even with equal-sized packets).

   (b). Is it the same as BBRR if all input queues are active?

3. Suppose we modify the SBBRR strategy of the previous exercise so that, if the output link is ever idle, and no packet has yet had all its bits ticked through by the bit-position marker, then we immediately send the packet with the fewest bits remaining to be ticked through by the bit-position marker.

Suppose packets P1 of size 1000 and P2 of size 100 arrive on subqueue 1. Just after P1 begins transmission, packet Q1 of size 400 arrives on subqueue 2. Fair queuing should send the packets in the order P1, Q1, P2; show that the mechanism described here does not do that.

4. Suppose we attempt to implement fair queuing by calculating the finishing time F_j for P_j, the jth packet in subqueue i, as follows.

- $Start_{j+1} = \max(F_j, now)$ ("now" by wallclock time)
- $F_j = Start_j + N \times L_j$

where N is the total number of subqueues, active or not.

(a). Suppose a router has three subqueues; *ie* N=3. The outbound bandwidth is 1 size unit / 1 time unit. At T=0, packets P1, P2, P3, P4 and P5 arrive for subqueue 1, each of size 1 unit. At T=2 (by which point P1 and P2 will have finished transmission), packets Q1 and Q2 arrive on subqueue 2, also of size 1. What finishing times will all the packets be assigned? In what order will they be transmitted?

(b). Is this strategy approximately equivalent to fair queuing if we are given that all subqueues of the router are always active?

5. Suppose we modify the strategy of the previous exercise by letting N be the number of active subqueues at the time of arrival of packet P_j. What happens if we have three input subqueues, and at T=0 five packets arrive for subqueue 1, and at T=1 five packets arrive for subqueue 2. Assume all ten packets are of size 1, and the output bandwidth is again 1 size unit per time unit.

6. The following packets all arrive at time T=0 at a router with an output rate of one size unit per time unit.

Subqueue 1: P1 of size 100, P2 of size 500, P3 of size 400
Subqueue 2: Q1 of size 300, Q2 of size 200, Q3 of size 600
Subqueue 3: R1 of size 400, R2 of size

(a). Find the BBRR virtual finishing time of each packet
(b). Give the actual wallclock finishing time of each packet, if the packets were sent via BBRR

6.5. Is byte-by-byte round-robin the same as bit-by-bit round-robin, if someone found a way to implement these literally? Does byte-by-byte round-robin lead to the same transmission order as BBRR?

7. Calculate GPS finishing times for the following packets, all present at T=0. There are two subqueues, and their bandwidth fractions are α and β where $\alpha = (\sqrt{5}-1)/2 \simeq 0.618$ and $\beta = \alpha^2 = 1-\alpha$. The packet sizes for the two subqueues are as follows (they follow the Fibonacci sequence, except 2 appears twice):

α: 2, 3, 8, 21, 55, 144, 377, 987
β: 1, 2, 5, 13, 34, 89, 233, 610

Hint: you will have to evaluate α to more decimal places than is shown here.

8. Suppose a WFQ router has two subqueues, each with a bandwidth fraction of α=50%. The router transmits 1 byte per ms. Initially, the subqueues are empty and T=0 and the GPS virtual clock is 0. At that moment a packet P1 of size 1000 bytes arrives at the first subqueue. At T=500, a similarly sized packet P2 arrives at the second subqueue. Give, for each of P1 and P2,

(a). Its finishing time under the GPS virtual clock
(b). Its wallclock finishing time
(c). The value of the GPS virtual clock at the moment of WFQ finishing.

9. Suppose a router has three subqueues, and an outbound bandwidth of 1 packet per unit time. Twelve packets arrive at or after T=0, timed so that the router remains busy until finishing the packets at T=12.

(a). What packet arrival schedule leads to the minimum final BBRR clock value?
(b). What schedule leads to the maximum final BBRR clock value?

Hint: the rate of the BBRR clock depends only on the number of active subqueues.

10. Suppose packets from three subqueues are sent using the quantum algorithm of *19.5.5 The Quantum Algorithm*. The packets are listed below in order of arrival for each subqueue, along with their lengths L; the packets are all available at time T=0. The quantum is 1000 bytes. Give the order of transmission.

Subqueue 1	Subqueue 2	Subqueue 3
P1, L=700	Q1, L=400	R1, L=500
P2, L=700	Q2, L=500	R2, L=600
P3, L=700	Q3, L=1000	R3, L=200
P4, L=700	Q4, L=200	R4, L=900

11. At Disneyland, patrons often wait in a queue that winds slowly through one large waiting room, only to feed into another queue in another room. Is this an example of hierarchical queuing, *eg* of one FIFO queue feeding another, without classes?

12. If two traffic streams meet token-bucket specifications of TB(r1,b1) and TB(r2,b2) respectively, show their commingled traffic must meet TB(r1+r2,b1+b2). Hint: imagine a common bucket of size b1+b2, filled at rate r1 with red tokens and at rate r2 with blue tokens.

13. For each sequence of arrival times, indicate which packets are compliant for the given token-bucket specification. If a packet is noncompliant, go on to the next arrival without decrementing the bucket.

(a). TB(1/4,5): 0, 0, 0, 2, 3, 4, 5, 7, 9, 11, 15, 18
(b). TB(1/3,6): 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
(c). TB(1/3,6): 0, 2, 4, 6, 8, 10, 12, 14, 16, 18

14. Find the fastest sequence (see the end of *19.9.3 Multiple Token Buckets*) for the following flows. Both start at T=0, and all buckets are initially full.

(a). TB(1/4, 4); packets can depart at a minimum of 1 time unit apart. Continue the sequence to at least T=10
(b). TB(1/2, 4) and TB(1/8, 8); multiple packets can depart at the same instant. Continue to at least T=25.

15. Give the fastest sequence of packets compliant for all three of the following token-bucket specifications. Continue the sequence at least until T=60.

- TB(1/2, 1)
- TB(1/6, 4)
- TB(1/12, 8)

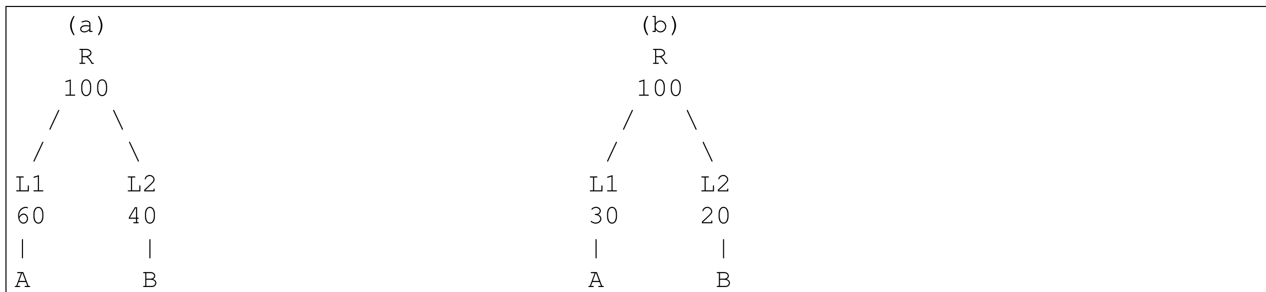
Hint: the first specification means arrival times must always be separated by at least 2. The middle specification should kick in by T=12.

16. Show that if a GPS traffic flow satisfies a token-bucket specification TB(r,B), then in any interval of time $t_1 \leqslant t \leqslant t_2$ the amount of traffic is at most $B + r\times(t_2-t_1)$. Hint: during the interval $t_1 \leqslant t \leqslant t_2$ the amount of fluid added to the bucket is exactly $r\times(t_2-t_1)$.

17. Show that a generic hierarchy of FIFO queuing disciplines, described in *19.7.1 Generic Hierarchical Queuing*, collapses to a single FIFO queue.

18. Show that the token-bucket leaf-storage hierarchy of *19.12 Hierarchical Token Bucket* produces the same result as an "internal-storage" hierarchy in which each intermediate token-bucket node contained a real, infinite-capacity FIFO queue, and each node instantaneously transmitted each packet to the parent's FIFO queue as soon as it was released. Show that packets are transmitted by each hierarchy at the same times. Hint: show that each node in the leaf-storage hierarchy "releases" a packet at the same time the corresponding internal-storage hierarchy forwards the packet upwards.

19. The following linux htb hierarchies are labeled with their guaranteed rates. Is there any difference in terms of the bandwidth allocations that would be received by senders A and B?



```
    (a)                                     (b)
     R                                       R
    100                                     100
   /   \                                   /   \
  /     \                                 /     \
L1      L2                              L1      L2
60      40                              30      20
|        |                              |        |
A        B                              A        B
```

20. Suppose we know that the real-time traffic through a given router R uses at most 1 Mbps of the total 10 Mbps bandwidth. Consider the following two ways of giving the real-time traffic special treatment:

i. Using priority queuing, and giving the real-time traffic higher priority.

ii. Using weighted fair queuing, and giving the real-time traffic a 10% share

(a). Show that, if the real-time traffic meets a token-bucket specification with rate 1 Mbps and negligible bucket size, then the two mechanisms are equivalent, in the sense that if the real-time and non-real-time traffic flows are sending at fractions α and β, respectively, of the 10 Mbps outbound rate, with α+β=1 (and with $\alpha \leqslant 10\%$), then the two methods above will actually send at the same rates.

(b). What differences can be expected if the bucket size is *not* negligible? Which approach will favor the real-time fraction?

21. In the previous exercise, now suppose we have *two* separate real-time flows, each guaranteed by a token-bucket specification not to exceed 1 Mbps. Is there a material difference between any pair of the following?

i. Sending the two real-time flows at priority 1, and the remaining traffic at priority 2.

ii. Sending the first real-time flow at priority 1, the second at priority 2, and the remaining traffic at priority 3.

iii. Giving each real-time flow a WFQ share of 10%, and the rest a WFQ share of 80%

22. Suppose a router uses priority queuing. There is one low-priority and one high-priority input. The outbound bandwidth is r.

(a). If the high-priority queue is currently empty, what is the maximum time that an arriving high-priority packet must wait?

(b). If the high-priority traffic follows a token-bucket description TB(rp,B), with rp < r, what is the maximum time an arriving high-priority packet must wait? Hint: use *19.11 Token Bucket Queue Utilization*.

Your answer may include symbolic representations of any necessary additional parameters.

So far, the Internet has been presented as a place where all traffic is sent on a **best-effort** basis and routers handle all traffic on an equal footing; indeed, this is often seen as a fundamental aspect of the IP layer. Delays and losses due to congestion are nearly universal. For bulk file-transfers this is usually quite sufficient; one way to look at TCP congestive losses, after all, is as part of a mechanism to ensure optimum utilization of the available bandwidth.

Sometimes, however, we may want some traffic to receive a certain minimum level of network services. We may allow some individual senders to negotiate such services in advance, or we may grant preferential service to specific protocols (such as VoIP). Such arrangements are known as **quality of service** (QoS) assurances, and may involve bandwidth, delay, loss rates, or any combination of these. Even bulk senders, for example, might sometimes wish to negotiate ahead of time for a specified amount of bandwidth.

While any sender might be interested in quality-of-service levels, they are an especially common concern for those sending and receiving **real-time** traffic such as voice-over-IP or videoconferencing. Real-time senders are likely to have not only bandwidth constraints, but constraints on delay and on loss rates as well. Furthermore, real-time applications may simply fail – at least temporarily – if these bandwidth, delay and loss constraints are not met.

In any network, large or small, in which bulk traffic may sometimes create queue backlogs large enough to cause unacceptable delay, quality-of-service assurances **must** involve the cooperation of the routers. These routers will then use the queuing and scheduling mechanisms of *19 Queuing and Scheduling* to set aside bandwidth for designated traffic. This is a major departure from the classic Internet model of "stateless" routers that have no information about specific connections or flows, though it is a natural feature of virtual-circuit routing.

In this chapter, we introduce some quality-of-service mechanisms for the Internet. We introduce the theory, anyway; some of these mechanisms have not exactly been adopted with warm arms by the ISP industry. Sometimes this is simply the chicken-and-egg problem: ISPs do not like to implement features nobody is using, but often nobody is using them because their ISPs don't support them. However, often an ISP's problem with a QoS feature comes down costs: routers will have more state to manage and more work to do, and this will require upgrades. There are two specific cost issues:

- it is not clear how to charge endpoints for their QoS requests (as with RSVP), particularly when the endpoints are not direct customers
- it is not clear how to compare special traffic like multicast, for the purpose of pricing, with standard unicast traffic

Nonetheless, VoIP at least is here to stay on a medium scale. Streaming video is here to stay on a large scale. Quality-of-service issues are no longer quite being ignored, or, at least, not blithely.

Note that a fundamental part of quality-of-service requests on the Internet is *sharing among multiple traffic classes*; that is, the transmission of "best-effort" and various grades of "premium" traffic *on the same network*. One can, after all, lease a SONET line and construct a private, premium-only network; such an approach is, however, expensive. Support for multiple traffic classes on the same network is sometimes referred to generically as **integrated services**, a special case of traffic engineering. It is not to be confused with the specific IETF protocol suite of that name (*20.4 Integrated Services / RSVP*). There are two separate

issues in an integrated network:

- how to make sure premium traffic gets the service it requires
- how to integrate different traffic classes on the same network

The first issue is addressed largely through the techniques presented in *19 Queuing and Scheduling*, and applies as well to networks with only a single service level. The present chapter addresses mostly the second issue.

Quality-of-service requests may be made for both TCP and UDP traffic. For example, an interactive TCP connection might request a minimum-delay path, while a bulk connection might request a maximum-bandwidth path and a streaming-prerecorded-video connection might request a specific guaranteed bandwidth. However, service quality is a particular concern of real-time traffic such as voice and interactive video, where delay can be a major difficulty. Such traffic is more likely to use UDP than TCP, because of the head-of-line blocking problem with the latter. We return to this in *20.3.3 UDP and Real-Time Traffic*.

20.1 Net Neutrality

There is a school of thought that says carriers must carry all traffic on an equal footing, and should be forbidden to charge extra for premium service. This is a strong formulation of the "net neutrality" principle, and it potentially complicates the implementation of some of the services described here. In principle, it may not matter whether the premium service involves interaction with routers, as is described in some of the mechanisms below, or simply involves improved access to best-effort carriage.

There are, however, many weaker formulations of net neutrality; for example, one is that traffic carriage must be "non-discriminatory". That is, a carrier could charge more for premium service so long as it did not single out individual providers for rate throttling.

Without taking sides on the net-neutrality debate, there are good reasons for arguing that additional charges for specific services from backbone routers are more appropriate than additional charges to avoid rate throttling.

20.2 Where the Wild Queues Are

We stated above that to ensure quality-of-service standards, it is necessary to have the participation of routers that have significant queue backlogs. It is not always clear, however, which routers these are. In the late 1990's, it was often claimed there was significant congestion at many (or at least some) "backbone" routers, where the word is in quotes here as a reminder that the term does not have a precise technical meaning. It is worth noting that this was also the era of 56 Kbps dialup access for most residential users.

In 2003, just a few years later, an analysis of the Internet in *[CM03]* concluded, "the Internet 'cloud' does not contribute heavily to congestion." At the time of this writing (2013), it appears that the Internet backbone continues to have excess capacity, and is seldom congested; delays, therefore, are more likely to be encountered more locally. A study in *[CBcDHL14]*, using timestamp pings to each end of suspected links, found that most congestion occurred in high-volume content-delivery networks (*1.12.2 Content-Distribution Networks*), such as those of Netflix and Google, and their interconnections to ISPs.

More concretely, suppose traffic from A to B goes through routers R1, R2 and R3:

If queuing delays (and losses) occur only at R1 and at R3, then there is no need to involve R2 in any bandwidth-reservation scheme; the same is true of R1 and R3 if the delays occur only at R2. Unfortunately, it is not always easy to determine the location of Internet congestion, and an abundance of bandwidth today may become a dearth tomorrow: bandwidth usage ineluctably grows to consume supply. That said, the early models for quality-of-service requests assumed that all (or at least most) routers would need to participate; it is quite possible that this is – practically speaking – no longer true. Another possible consequence is that adequate QoS levels might – *might* – be attainable with only the participation of ones immediate ISP.

20.3 Real-time Traffic

Real-time traffic is traffic with some sort of hard or soft **delay** bound, presumably larger than the one-way no-load propagation delay. Such traffic can be said to be **delay-intolerant**. For voice or video traffic, a packet arriving after the time at which it is to be played back might as well have been lost.

Fortunately, voice and video are also **loss-tolerant**, at least to a degree: a lost voice packet simply results in a momentary voice dropout; a lost video packet might result in replay of the previous video frame. Handling traffic that is both loss- and delay-*in*tolerant is very difficult; we will not consider that case further.

Much (but not all) real-time traffic is also **rate-adaptive**. For example, the online-video service hulu.com can send at resolutions of 288p, 360p, 480p and 720p, approximately corresponding to bandwidths of 480 Kbps, 700 Kbps, 1 Mbps, and 2.5 Mbps [2012 data]. Hulu's software can dynamically choose the appropriate rate. (Hulu transmissions, where the consequence of delay is a pause in the video replay rather than a loss, are not necessarily "real-time"; see *20.3.2 Streaming Video* below.)

As another example of rate-adaptiveness, the voice-grade audio-compression codec Opus (a successor to an earlier codec known as Speex) might normally be used at a 64 Kbps rate, but supports a more-or-less continuous range of rates down to 8 Kbps. While the lower rates have lower voice quality, they can be used as a fallback in the event that congestion prevents successful use of the 64 Kbps rate.

Generally speaking, rate-adaptivity notwithstanding, real-time traffic needs sufficient management that congestion becomes minimal. Real-time traffic should not be allowed to arrive at any router, for example, faster than it can depart. The most definitive way to achieve this is via some sort of reservation or admission-control mechanism, where new connections will not be accepted unless resources are available.

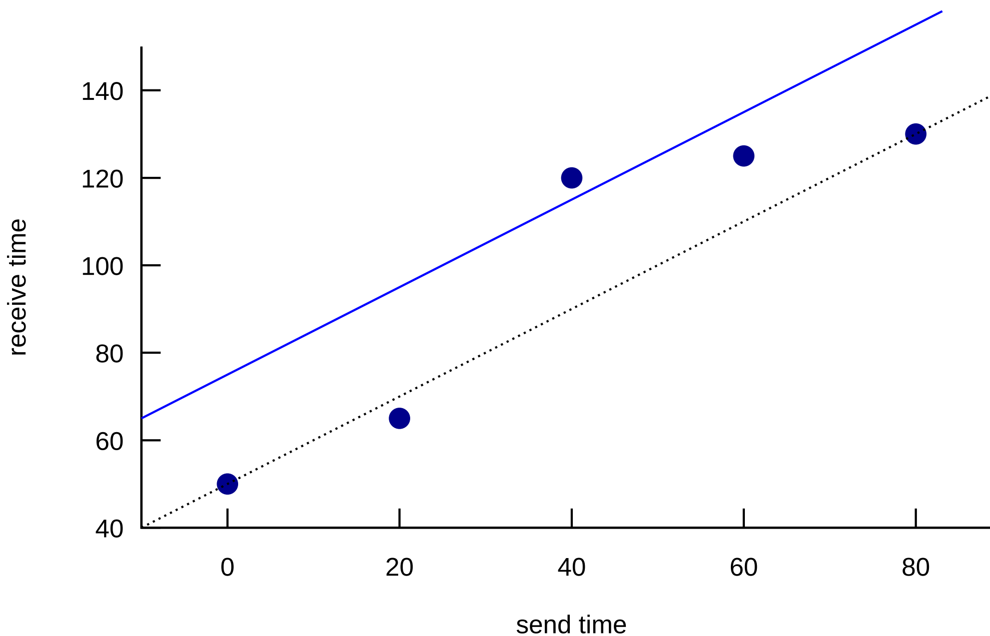
20.3.1 Playback Buffer

Real-time applications cannot avoid delay completely, of course, so the received stream will be delivered to the receiving application (for playback, if it is a voice or video stream) slightly behind the time when it was sent. Applications can intentionally increase this time by creating a **playback buffer** or **jitter buffer**; this allows the smoothing over of variations in delay (known as **jitter**).

For example, suppose a VoIP application sends a packet every 20 ms (a typical rate). If the delay is exactly 50 ms, then packets sent at times T=0, 20, 40, *etc* will arrive at times T=50, 70, 90, *etc*. Now suppose the delay is not so uniform, so packets sent at T=0, 20, 40, 60, 80 arrive at times 50, 65, 120, 125, 130.

packet	sent	expected	received	(rec'd – expected)
1	0	50	50	0
2	20	70	65	-5
3	40	90	120	30
4	60	110	125	15
5	80	130	130	0

The first and the last packet have arrived on time, the second is early, and the third and fourth are late by 30 and 15 ms respectively. Setting the playback buffer capacity to 25 ms means that the third packet is not received in time to be played back, and so must be discarded; setting the buffer to a value at least 30 ms means that all the packets are received.



The diagram above plots the points of the table. The dashed line represents the expected arrival time for the corresponding send time; packets are late by the amount they are above this line. The solid blue line represents a playback buffer corresponding to 25 ms; packets arriving at points above the line are lost. The higher the line, the more playback delay, but also the more that playback can accommodate late packets

For non-interactive voice and video, there is essentially no penalty to making the playback buffer quite long. But for telephony, if one speaker stops and the other starts, they will perceive a gap of length equal to the RTT including playback-buffer delay. For voice, this becomes increasingly annoying if the RTT delay reaches about 200-400 ms.

20.3.2 Streaming Video

Streaming video is something of a gray area. On the one hand, it is not really real-time; viewers do not necessarily care how long the playback-buffer delay is as long as it is consistent. Playback-buffer delays of ten seconds to up to a minute are not uncommon. As long as the sender can stay ahead of the playback application, all is well. The real issue is bandwidth; a playback-buffer delay in the tens of seconds is two orders of magnitude larger than the delay bounds that a genuine real-time application might request.

For very large videos, the sender probably coordinates with the playback application to limit how far ahead it gets; this amounts to using a fixed-size playback buffer. That playback buffer can accommodate *some* fluctuation in the delivery rate, but in the long run the delivery bandwidth must be at least as large as the playback rate. For smaller videos, *eg* traditional ten-minute YouTube clips, the sender sends as fast as it can without stopping. If the delivery bandwidth is less than the playback rate then the viewer can hit "pause" and wait until the entire video has been downloaded.

On the other hand, viewers do not particularly like pauses, especially during long videos. If a viewer starts a two-hour movie, average network congestion levels may change materially many times during that interval. The viewer would prefer a more-or-less consistent bandwidth, even if competing traffic ramps up; rate-adaptive playback is fine until one has signed up for HD-quality viewing. What the viewer here wants is an average-bandwidth guarantee. This can be supplied by overbuilding the network, by implementing some of the mechanisms of this chapter, or by various intermediate approaches.

For example, a large-enough network-content provider N might negotiate with a residential Internet carrier C so that there is sufficient long-haul bandwidth from N to the end-user viewers. If the problem is backbone congestion, then N might arrange to use BGP's MED option (*10.6.6.1 MED values and traffic engineering*) to carry the traffic as far as possible in its own network; this may also entail the creation of a large number of peering points with C's network (*10.4.1 Internet Exchange Points*). Alternatively, C might be persuaded to set aside one very large traffic category in its own backbone network (perhaps using *20.7 Differentiated Services*) that is reserved for N's traffic.

20.3.3 UDP and Real-Time Traffic

The main difficulty with using TCP for real-time traffic is **head-of-line blocking**: when a loss does occur, the TCP layer will hold any later data until the lost segment times out and is retransmitted. Even if the receiving application is able simply to ignore the lost packet, it is not granted that option.

In theory, the TCP timeout interval may be only slightly more than the RTT, which is often well under 100 ms. In practice, however, it is often much larger, due in part to the use of a coarse-grained clock to keep track of timeouts. TCP implementations are actually discouraged from using smaller timeout intervals; the following is from **RFC 6298** (2011); RTO stands for Retransmission TimeOut:

> Whenever RTO is computed, if it is less than 1 second, then the RTO SHOULD be rounded up to 1 second.

Such a policy makes TCP unsuitable, except in environments with vanishingly small loss rates, whenever any genuine interactive response is needed; this includes telephony, video telephony, and most forms of video conferencing that involve real-time feedback between participants. (TCP *does* work well with video streaming.) While it is true that the popular video-telephony package Skype does in fact use TCP, this is not because Skype has figured a way around this limitation; Skype sessions are not infrequently plagued by congestion-related difficulties. The use of UDP allows an application the option of deciding that lost or late data should simply be skipped over.

20.4 Integrated Services / RSVP

Integrated Services, or **IntServ**, was developed by the IETF in the late 1990's as a first attempt at providing quality-of-service guarantees for selected Internet traffic. The IntServ model assumed all routers would

participate, or at least all routers along the connection path, though this is not strictly necessary. Connections were to make reservations using the Resource ReSerVation Protocol **RSVP**.

Note that this is a major retreat from the datagram-routing stateless-router model. Were virtual circuits (*3.4 Virtual Circuits*) the better routing model all along? For better or for worse, the marketplace appears to have answered this question with an unambiguous "no"; IntServ has seen very limited adoption in the core Internet.

However, many of the ideas of IntServ are implementable on the internet using an appropriate Content Distribution Network. After outlining IntServ itself, we describe this CDN alternative in *20.6.1 A CDN Alternative to IntServ*.

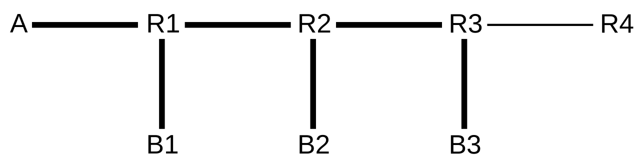
Under IntServ, routers maintain **soft state** about a connection, meaning that reservations must be refreshed by the sender at regular intervals (*eg* 30 seconds); if not, the reservation can be discarded. This also means that reservations can be recovered if the router crashes (though with some small probability of failure). Traditional virtual-circuit switches maintain **hard state**, so that if a sender stops sending but fails to "hang up" properly then the connection is still maintained (and perhaps charged for), and a router crash means the connection is lost.

IntServ has mostly not been supported by the backbone ISP industry. Partly this is because of practical difficulties in figuring out how to charge for reservations; if the charge is zero then everyone might make reservations for every connection. Another issue is that a busy router might have to maintain thousands of reservations; IntServ thus adds a genuine expense to the ISP's infrastructure.

At the time IntServ was developed, the dominant application envisioned was teleconferencing, in which one speaker's audio/video stream would be sent to a large number of receivers. Because of this, IntServ was based on **IP multicast**, to which we therefore turn next. This decision made IntServ somewhat more complex than would be necessary if point-to-point VoIP were all that was required, and future Internet reservation mechanisms may jettison multicast support (see *20.9 NSIS*). However, multicast and IntServ do share something fundamental in common: both require the participation of intermediate routers; neither can be effectively implemented solely in end systems.

20.5 Global IP Multicast

The idea behind IP multicast, following up on *7.3.1 Multicast addresses*, is that sender A transmits a stream of packets (real-time or not) to a *set* of receivers {B1, B2, ..., Bn}, in such a way that *no one packet of the stream is transmitted more than once on any one link*. This means that it is up to routers on the way to **duplicate** packets that need to be forwarded on multiple outbound links. For example, suppose A below wishes to send to B1, B2 and B3 in the following diagram:



Then R1 will receive packet 1 from A and will forward it to both B1 and to R2. R2 will receive packet 1 from R1 and forward it to B2 and R3. R3 will forward only to B3; R4 does not see the traffic at all. The

set of paths used by the multicast traffic, from the sender to all destinations, is known as the **multicast tree**; these are shown in bold.

We should acknowledge upfront that, while IP multicast is potentially a very useful technology, as with IntServ there is not much support for it within the mainstream ISP industry. The central issues are the need for routers to maintain complex new information and the difficulty in figuring out how to charge for this kind of traffic. At larger scales ISPs normally charge by total traffic carried; now suppose an ISP's portion of a multicast tree is a single path all across the continent, but the tree branches into ten different paths near the point of egress. Should the ISP consider this to be like one unicast connection, or more like ten?

Once Upon A Time, an ideal candidate for multicast might have been the large-scale delivery of broadcast television. Ironically, the expansion of the Internet backbone has meant that large-scale video delivery is now achieved with an individual unicast connection for every viewer. This is the model of YouTube.com, Netflix.com and Hulu.com, and almost every other site delivering video. Online education also once upon a time might have been a natural candidate for multicast, and here again separate unicast connections are now entirely affordable. The bottom line for the future of multicast, then, is whether there is an application out there that really needs it.

Note that IP multicast is potentially straightforward to implement within a single large (or small) organization. In this setting, though, the organization is free to set its own budget rules for multicast use.

Multicast traffic will consist of UDP packets; there is no provision in the TCP specification for "multicast connections". For large groups, acknowledgment by every receiver of the multicast UDP packets is impractical; the returning ACKs could consume more bandwidth than the outbound data. Fortunately, complete acknowledgments are often unnecessary; the archetypal example of multicast traffic is loss-tolerant voice and video traffic. The RTP protocol (*20.11 Real-time Transport Protocol (RTP)*) includes a response mechanism from receivers to senders; the RTP response packets are intended to at least give the sender some idea of the loss rate. Some effort is expended in the RTP protocol (more precisely, in the companion protocol RTCP) to make sure that these response packets, from multiple recipients, do not end up amounting to more traffic than the data traffic.

In the original Ethernet model for LAN-level multicast, nodes agree on a physical multicast address, and then receivers *subscribe* to this address, by instructing their network-interface cards to forward on up to the host system all packets with this address as destination. Switches, in turn, were expected to treat packets addressed to multicast addresses the same as broadcast, forwarding on *all* interfaces other than the arrival interface.

Global broadcast, however, is not an option for the Internet as a whole. Routers must receive specific instructions about forwarding packets. Even on large switched Ethernets, newer switches generally try to avoid broadcasting every multicast packet, preferring instead to attempt to figure out where the subscribers to the multicast group are actually located.

In principle, IP multicast routing can be thought of as an extension of IP unicast routing. Under this model, IP forwarding tables would have the usual ⟨udest,next_hop⟩ entries for each **unicast** destination, and ⟨mdest,**set**_of_next_hops⟩ entries for each **multicast** destination. In the diagram above, if G represents the multicast group of receivers {B1,B2,B3}, then R1 would have an entry ⟨G,{B1,R2}⟩. All that is needed to get multicast routing to work are extensions to distance-vector, link-state and BGP router-update algorithms to accommodate multicast destinations. (We are using G here to denote both the actual multicast group and also the multicast *address* for the group; we are also for the time being ignoring exactly how a group would be assigned an actual multicast address.)

These routing-protocol extensions can be done (and in fact it is quite straightforward in the link-state case,

as each node can use its local network map to figure out the optimal multicast tree), but there are some problems. First off, if any Internet host might potentially join any multicast group, then each router must maintain a separate entry for each multicast group; there are no opportunities for consolidation or for hierarchical routing. For that matter, there is no way even to support for multicast the basic unicast-IP separation of addresses into network and host portions that was a crucial part of the continued scalability of IP routing. The Class-D multicast address block contains $2^{28} \simeq 270$ million entries, far to many to support a routing-table entry for each.

The second problem is that multicast groups, unlike unicast destinations, may be **ephemeral**; this would place an additional burden on routers trying to keep track of routing to such groups. An example of an ephemeral group would be one used only for a specific video-conference speaker.

Finally, multicast groups also tend to be of interest only to their members, in that hosts far and wide on the Internet generally do not send to multicast groups to which they do not have a close relationship. In the diagram above, the sender A might not actually be a member of the group {B1,B2,B3}, but there is a strong tie. There may be no reason for R4 to know anything about the multicast group.

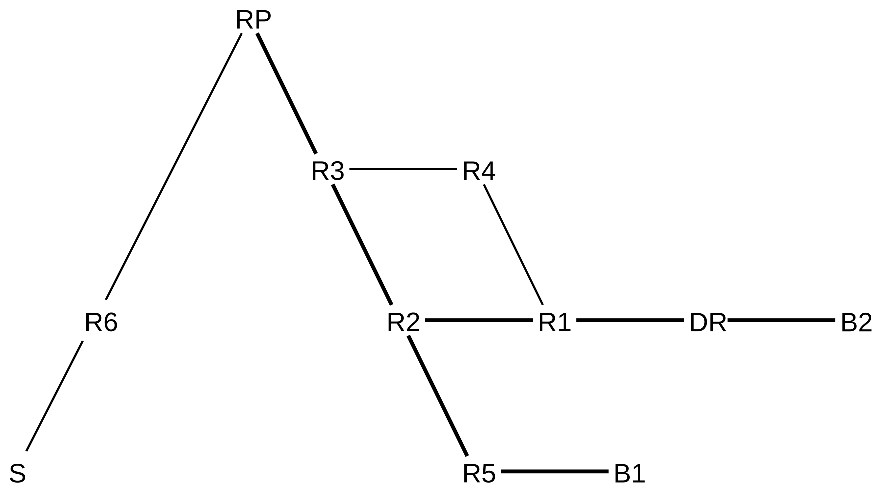
So we need another way to think about multicast routing. Perhaps the most successful approach has been the **subscription** model, where senders and receivers join and leave a multicast group dynamically, and there is no route to the group except for those subscribed to it. Routers update the multicast tree on each join/leave event. The optimal multicast tree is determined not only by the receiving group, {B1,B2,B3}, but also by the *sender*, A; if a different sender wanted to send to the group, a different tree might be constructed. In practice, sender-specific trees may be constructed only for senders transmitting large volumes of data; less important senders put up with a modicum of inefficiency.

The multicast protocol most suited for these purposes is known as PIM-SM, defined in **RFC 2362**. PIM stands for **Protocol-Independent Multicast**, where "protocol-independent" means that it is not tied to a specific routing protocol such as distance-vector or link-state. SM here stands for **Sparse Mode**, meaning that the set of members may be widely scattered on the Internet. We acknowledge again that, while PIM-SM is a reasonable starting point for a realistic multicast implementation, it may be difficult to find an ISP that implements it.

The first step for PIM-SM, given a multicast group G as destination, is for the designation of a router to serve as the **rendezvous point**, RP, for G. If the multicast group is being set up by a particular sender, RP might be a router near that sender. The RP will serve as the default root of the multicast tree, up until such time as sender-specific multicast trees are created. Senders will send packets to the RP, which will forward them out the multicast tree to all group members.

At the bottom level, the Internet Group Management Protocol, IGMP, is used for hosts to inform one of their local routers (their **designated router**, or DR) of the groups they wish to join. When such a designated router learns that a host B wishes to join group G, it forwards a **join** request to the RP.

The join request travels from the DR to the RP via the usual IP-unicast path from DR to RP. Suppose that path for the diagram below is ⟨DR,R1,R2,R3,RP⟩. Then every router in this chain will create (or update) an entry for the group G; each router will record in this entry that traffic to G will need to be sent to the previous router in the list (starting from DR), and that traffic *from* G must come from the next router in the list (ultimately from the RP).



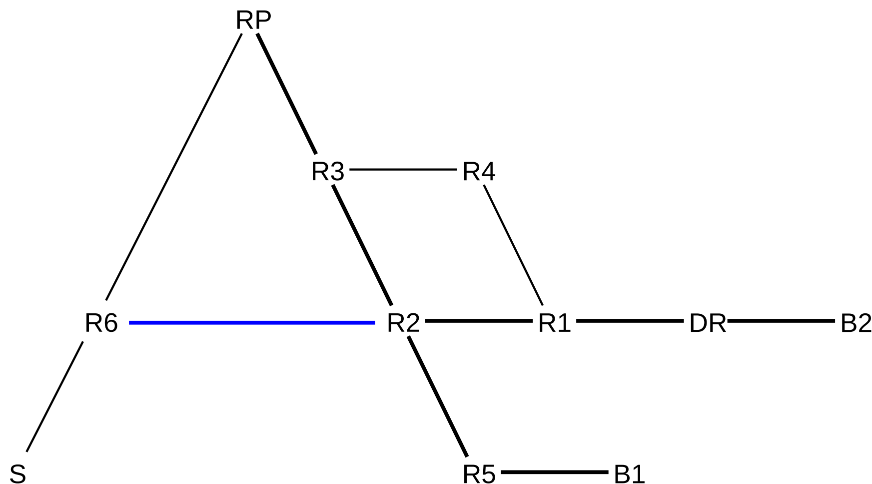
In the above diagram, suppose B1 is first to join the group. B1's designated router is R5, and the join packet is sent R5→R2→R3→RP. R5, R2 and R3 now have entries for G; R2's entry, for example, specifies that packets addressed to G are to be sent *to* {R5} and must come *from* R3. These entries are all tagged with ⟨*,G⟩, to use RFC 2362's notation, where the "*" means "any sender"; we will return to this below when we get to sender-specific trees.

Now B2 wishes to join, and its designated router DR sends its join request along the path DR→R1→R2→R3→RP. R2 updates its entry for G to reflect that packets addressed to G are to be forwarded to the set {R5,R1}. In fact, R2 does not need to forward the join packet to R3 at all.

At this point, a sender S can send to the group G by creating a multicast-addressed IP packet and **encapsulating** it in a unicast IP packet addressed to RP. RP opens the encapsulation and forwards the packet down the tree, represented by the bold links above.

Note that the data packets sent by RP to DR will follow the path RP→R3→R2→R1, as set up above, even if the normal *unicast* path from R3 to R1 were R3→**R4**→R1. The multicast path was based on R1's preferred next_hop to RP, which was assumed to be R2. Traffic here from sender to a specific receiver takes the exact reverse of the path that a unicast packet would take from that receiver to the sender; as we saw in *10.4.3 Hierarchical Routing via Providers*, it is common for unicast IP traffic to take a different path each direction.

The next step is to introduce **source-specific** trees for high-volume senders. Suppose that sender S above is sending a considerable amount of traffic to G, and that there is also an R6–R2 link (in blue below) that can serve as a shortcut from S to {B1,B2}:



We will still suppose that traffic from R2 reaches RP via R3 and not R6. However, we would like to allow S to send to G via the more-direct path R6→R2. RP would initiate this through special join messages sent to R6; a message would then be sent from RP to the group G announcing the option of creating a source-specific tree for S (or, more properly, for S's designated router R6). For R1 and R5, there is no change; these routers reach RP and R6 through the same next_hop router R2.

However, R2 receives this message and notes that it can reach R6 directly (or, in general, at least via a different path than it uses to reach RP), and so R2 will send a join message to R6. R2 and R6 will now each have general entries for ⟨*,G⟩ but also a source-specific entry ⟨S,G⟩, meaning that R6 will forward traffic addressed to G **and coming from S** to R2, and R2 will accept it. R6 may still also forward these packets to RP (as RP does belong to group G), but RP might also by then have an ⟨S,G⟩ entry that says (unless the diagram above is extended) not to forward any further.

The tags ⟨*,G⟩ and ⟨S,G⟩ thus mark two different trees, one rooted at RP and the other rooted at R6. Routers each use an implicit closest-match strategy, using a source-specific entry if one is available and the wildcard ⟨*,G⟩ entry otherwise.

As mentioned repeatedly above, the necessary ISP cooperation with all this has not been forthcoming. As a stopgap measure, the multicast backbone or **Mbone** was created as a modest subset of multicast-aware routers. Most of the routers were actually within Internet leaf-customer domains rather than ISPs, let alone backbone ISPs. To join a multicast group on the Mbone, one first identified the nearest Mbone router and then connected to it using tunneling. The Mbone gradually faded away after the year 2000.

We have not discussed at all how a multicast address would be allocated to a specific set of hosts wishing to form a multicast group. There are several large blocks of class-D addresses assigned by the IANA. Some of these are assigned to specific protocols; for example, the multicast address for the Network Time Protocol is 224.0.1.1 (though you can use NTP quite happily without using multicast). The 232.0.0.0/8 block is reserved for source-specific multicast, and the 233.0.0.0/8 block is allocated by the GLOP standard; if a site has a 16-bit Autonomous System number with bytes x and y, then that site automatically gets the multicast block 233.x.y.0/24. A fuller allocation scheme waits for the adoption and development of a broader IP-multicast strategy. This may never happen.

20.6 RSVP

We next turn to the RSVP (ReSerVation) protocol, which forms the core of IntServ.

The original model for RSVP was to support multicast, so as to support teleconferencing. For this reason, reservations are requested not by senders but by receivers, as a multicast sender may not even know who all the receivers are. Reservations are also for one direction; bidirectional traffic needs to make two reservations.

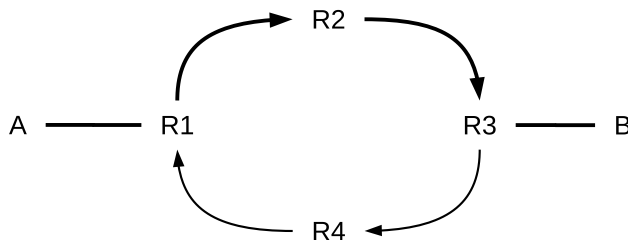
Like multicast, RSVP generally requires participation of intermediate routers.

Reservations include both a **flowspec** describing the traffic flow (*eg* a unicast or multicast destination) and also a **filterspec** describing how to identify the packets of the flow. We will assume that filterspecs simply define unidirectional unicast flows, *eg* by specifying source and destination sockets, but more general filter-specs are possible. A component of the flowspec is the **Tspec**, or traffic spec; this is where the token-bucket specification for the flow appears. Tspecs do not in fact include a bound on total delay; however, the degree of *queuing* delay at each router can be computed from the TB(r,B_{max}) token-bucket parameters as B_{max}/r.

The two main messages used by RSVP are **PATH** packets, which move from sender to receiver, and the subsequent **RESV** packets, which move from receiver to sender.

Initially, the sender (or senders) sends a PATH message to the receiver (or receivers), either via a single unicast connection or to a multicast group. The PATH message contains the sender's Tspec, which the receivers need to know to make their reservations. But the PATH messages are not just for the ultimate recipients: every router on the path examines these packets and learns the identity of the next_hop RSVP router in the *upstream* direction. The PATH messages inform each router along the path of the **path state** for the sender.

As an example, imagine that sender A sends a PATH message to receiver B, using normal unicast delivery. Suppose the route taken is A→R1→**R2**→R3→B. Suppose also that if B simply sends a unicast message to A, however, then the route is the rather different B→R3→**R4**→R1→A.



Arrows show path of normal unicast traffic between A and B
RSVP traffic, however, follows the bold links in both directions

As A's PATH message heads to B, R2 must record that R1 is the next hop back to A along this particular PATH, and R3 must record that R2 is the next reverse-path hop back to A, and even B needs to note R3 is the next hop back to A (R1 presumably already knows this, as it is directly connected to A). To convey this reverse-path information, each router inserts its own IP address at a specific location in the PATH packet, so that the next router to receive the PATH packet will know the reverse-path next hop. All this path state stored at each router includes not only the address of the previous router, but also the sender's Tspec. All these path-state records are for this particular PATH only.

The PATH packet, in other words, tells the receiver what the Tspec is, and prepares the routers along the way for future reservations.

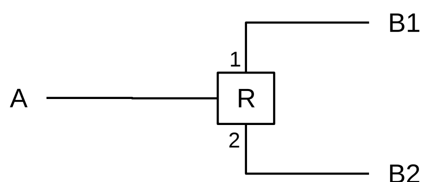
The actual traffic from sender A to receiver B will eventually be forwarded along the standard unicast path (or multicast-tree branch) from A to B, just like PATH messages from A to B. Routers will still have to check, though, whether each packet forwarded is part of a reservation or not, in order to give reserved traffic preferential drop rates and reduced queuing delays, and in order to police the reservation to ensure that the sender is not sending more reserved traffic than agreed. Reserved traffic is identified by source and destination IP addresses and port numbers. This IP-address-and-port combination can be likened to the VCI of virtual-circuit routing (*3.4 Virtual Circuits*), though it is constant along the path.

After receiving the sender's PATH message, each receiver now responds with its RESV message, requesting its reservation. The RESV packets are passed back to the sender not by the default unicast route, but along the reverse path created by the PATH message. In the example above, the RESV packet would travel B→R3→R2→R1→A. Each router (and also B) must look at the RESV message and look up the corresponding PATH record in order to figure out how to pass the reservation message back up the chain. If the RESV message were sent using normal unicast, via B→R3→R4→R1→A, then R2 would not see it.

Each router seeing the RESV path must also make a decision as to whether it is able to grant the reservation. This is the **admission control** decision. RSVP-compliant routers will typically set aside some fraction of their total bandwidth for their reservations, and will likely use priority queuing to give preferred service to this fraction. However, as long as this fraction is well under 100%, bulk unreserved traffic will not be shut out. Fair queuing can also be used.

Reservations must be resent every so often (*eg* every ~30 seconds) or they will time out and go away; this means that a receiver that is shut down without canceling its reservation will not continue to tie up resources.

If the RESV messages are moving up a multicast tree, rather than backwards along a unicast path, then they are likely to reach a router that already has granted a reservation of equal or greater capacity. In the diagram below, router R has granted a reservation for traffic from A to receiver B1, reached via R's interface 1, and now has a similar reservation from receiver B2 reached via R's interface 2.



Assuming R is able to grant B2's reservation, it does not have to send the RESV packet upstream any further (at least not as requests for a *new* reservation); B2's reservation can be **merged** with B1's. R simply will receive packets from A and now forward them out both of its interfaces 1 and 2, to the two receivers B1 and B2 respectively.

It is not necessary that *every* router along the path be RSVP-aware. Suppose A sends its PATH messages to B via A→R1→R2a→R2b→R3→B, where every router is listed but R2a and R2b are part of a non-RSVP "cloud". Then R2a and R2b will not store any path state, but also will not mark the PATH packets with their IP addresses. So when a PATH packet arrives at R3, it still bears R1's IP address, and R3 records R1 as the reverse-path next hop. It is now possible that when R3 sends RESV packets back to R1, they will take a different path R3→R2c→R1, but this does not matter as R2a, R2b and R2c are not accepting

reservations anyway. Best-effort delivery will be used instead for these routers, but at least part of the path will be covered by reservations. As we outlined in *20.2 Where the Wild Queues Are*, it is quite possible that we do not *need* the participation of the various R2's to get the quality of service wanted; perhaps only R1 and R3 contribute to delays.

In the multicast multiple-sender/multiple-receiver model, not every receiver must make a reservation for all senders; some receivers may have no interest in some senders. Also, if rate-adaptive data transmission protocols are used, some receivers may make a reservation for a sender at a lower rate than that at which the sender is sending. For this to work, some node between sender and receiver must be willing to decode and re-encode the data at a lower rate; the RTP protocol provides some support for this in the form of **RTP mixers** (*20.11.1 RTP Mixers*). This allows different members of the multicast receiver group to receive the same audio/video stream but at different resolutions.

From an ISP's perspective, the problems with RSVP are that there are likely to be a *lot* of reservations, and the ISP must figure out how to decide who gets to reserve what. One model is simply to charge for reservations, but this is complicated when the ISP doing the charging is not the ISP providing service to the receivers involved. Another model is to allow anyone to ask for a reservation, but to maintain a cap on the number of reservations from any one site.

These sorts of problems have largely prevented RSVP from being implemented in the Internet backbone. That said, RSVP is apparently quite successful within some corporate intranets, where it can be used to support voice traffic on the same LANs as data.

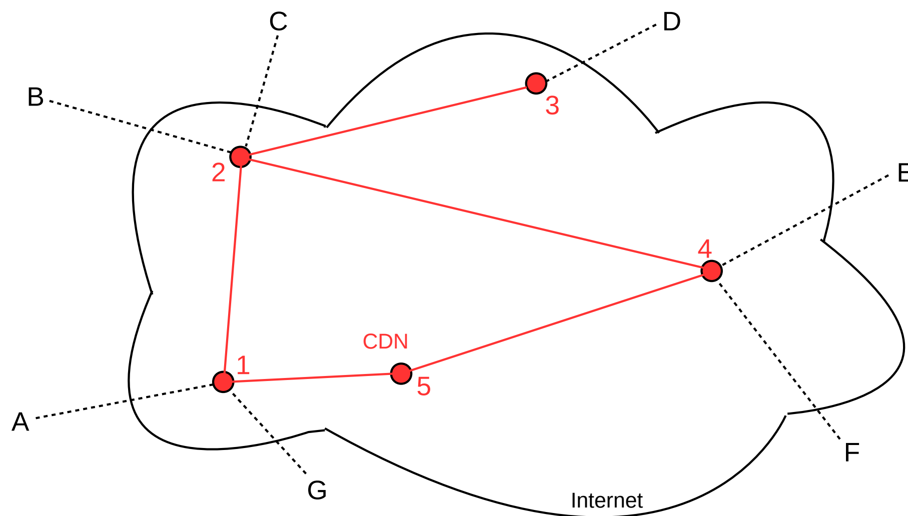
20.6.1 A CDN Alternative to IntServ

The core components of IntServ are, arguably,

- IP multicast
- Traffic reservations

While "true" IntServ implementations of these may never come to widespread (or even narrow) adoption, for many purposes a **content-distribution network** (*1.12.2 Content-Distribution Networks*) can achieve the same two goals of multicasting and reservations, without requiring any cooperation from the backbone routing infrastructure. Reservations may require private links between CDN nodes, but application-layer multicasting is straightforward to implement in software alone.

Imagine a videoconference presenter at node A, below. Nodes B through G represent video receivers. According to the IntServ strategy, we would create a multicast tree rooted at A, and then A's video presentation would be multicast to nodes B through G. The links on this multicast tree – consisting of both the red and the dashed links in the image below – would each (or mostly each) maintain an appropriate traffic reservation.



But instead of that, imagine that we have a network – that is, a CDN – consisting of multiple publicly accessible nodes (called **points of presence**, or PoPs), all connected by high-speed links (virtual or physical). These links between nodes are represented by the red lines in the diagram above. Each of the users A-G then connects to the CDN, using the usual public Internet; these are the dashed lines. These connections will probably be made under the aegis of a videoconferencing provider that has leased the services of the CDN. Each user – using software from the provider – attempts to connect to the closest, or at least to a reasonably close, access point of the CDN.

The CDN will be made aware of A as data source and B-G as subscribers, and will forward A's incoming data from 1 to 2, 3 and 4. A's data will, in keeping with the idea of multicast, not be forwarded along any link twice. This forwarding will most likely be done within the videoconferencing application layer rather than by IP-layer multicasting. That is, CDN node 1 will receive A's data stream and distribute it within the CDN to nodes 2, 3 and 4 as the CDN sees fit.

For real-time traffic, performance may depend on the implementation of the red intra-CDN links. The gold standard, in terms of support for real-time streaming, is for the red links to be private leased lines – and routers – over which the CDN has full control. In this case, load and queuing delay can be regulated as necessary, and true reservations (internal to the CDN) can be granted. The Google internal wide-area network (as of 2017), known as B4, has this sort of architecture; it carries huge volumes of traffic. See *[JKMOPSVWZ13]*.

At the other end, these red links may simply be paths through the public Internet, making the CDN an "overlay" network vaguely resembling the original Mbone. In this case the CDN's real-time traffic will compete with other traffic. This isn't necessarily bad, however, if the red links – which probably are Internet "backbone" links – have high capacity. Perhaps more importantly, a CDN is a large customer for the ISPs that connect it, and likely has a close business relationship with them. If congestion is a business problem for the CDN, in that it wants to attract videoconferencing providers as clients, it is in a strong position to negotiate appropriate service guarantees and even partial isolation of its real-time traffic flows.

Netflix Open Connect

The Netflix CDN used to deliver streamed video consists largely of so-called Open Connect Appliances (OCAs), each containing upwards of 100 TB (and colored Netflix red). These are distributed – rather widely – among ISPs and IXPs. Each OCA gets its daily updates (of multiple TB) delivered over the public Internet. There are no private links. Netflix has no need for them; none of their content is real-time.

However the intra-CDN links are implemented, data will flow in our multicast scenario as follows:

- From source A to CDN node 1, using the Internet
- Throughout the CDN's internal links as necessary
- From CDN nodes 1, 2, 3 and 4 to subscribers B through G, using each subscriber's Internet link

Many teleconferencing services use this sort of architecture. Subscribers can be charged for the privilege of using the network, solving the multicast and reservation pricing issues. In principle (though very seldom in practice) subscribers can be told the network is busy if congestion levels are high, addressing the reservation admission issue. More likely, the provider would handle high-use periods by allowing performance to degrade, while also attempting to provision the network so that it is adequate "most" of the time.

If the CDN is "small" – perhaps a dozen points of presence – then the dashed normal-Internet links from users to the CDN might end up handling most of the traffic carriage. However, if the CDN is reasonably large, then the dashed links may shrink to intra-city distances, and the red links will handle the long-haul traffic.

20.7 Differentiated Services

Differentiated Services, or DiffServ, was created as a low-overhead alternative to IntServ. The idea behind DiffServ is to replace IntServ's many reservations with just two service classes: **regular** (for everyone's bulk traffic) and **premium** (for what in IntServ would have been everyone's reserved traffic). For this reason, DiffServ is sometimes describe as providing "coarse-grained" resource allocation versus RSVP's "fine-grained" per-flow allocations. Like IntServ, DiffServ is easiest to implement within a single ISP or Autonomous System; however, DiffServ may be reasonably practical to implement across ISP boundaries as well. A set of routers agreeing on a common DiffServ policy is called a **DS domain**; a DS domain might consist of a single ISP but might also comprise a larger "backbone" ISP and some "regional" customer ISPs that have negotiated a single DiffServ policy.

Packets are marked at (and only at) the border routers of each DS domain, as the traffic in question enters that domain. The DS domain's interior routers do no marking and no traffic policing. At these interior routers, priority queuing is generally used to give premium service to the marked premium packets.

Packets are marked using the six bits of the DS field of the IPv4 header (*7.1 The IPv4 Header*); these were part of what was originally the IP Type-of-Service bits. DiffServ traffic is divided into several classes called Per Hop Behaviors, or **PHBs**; PHBs are perhaps best thought of as service classes. The simplest PHB is the **default PHB**, meaning the packet gets no special processing.

The **Class Selector** PHB is for backwards compatibility with the three-bit precedence subfield of the old IPv4 Type-of-Service field.

The **Expedited Forwarding** (EF) PHB (*20.7.1 Expedited Forwarding*) is arguably the best service. It is meant for traffic that has delay constraints in addition to rate constraints. The newer **Voice Admit** PHB is closely related and has been recommended as the preferred PHB for voice telephony.

Assured Forwarding, or AF (*20.7.2 Assured Forwarding*), is really four separate PHBs, corresponding to four **classes** 1 through 4. It is meant for providing (soft) service guarantees that are contingent on the sender's staying within a certain rate specification. Each AF class has its own rate specification; these rate specifications are entirely at the discretion of the implementing DS domain. AF uses the first three bits to denote the AF class, and the second three bits to denote the **drop precedence**.

Here are the six-bit patterns for the above PHBs; the AF drop-precedence bits are denoted "ddd".

- 000 000: default PHB (best-effort delivery)
- 001 ddd: AF class 1 (the lowest priority)
- 010 ddd: AF class 2
- 011 ddd: AF class 3
- 100 ddd: AF class 4 (the best)
- 101 110: Expedited Forwarding
- 101 100: Voice Admit
- 11x 000: Network control traffic (**RFC 2597**)
- xxx 000: Class Selector (traditional IPv4 Type-of-Service)

The goal behind premium PHBs such as EF and AF is for the DS domain to set some rules on admitting premium packets, and hope that their total numbers to any given destination are small enough that high-level service targets can be met. This is not exactly the same as a guarantee, and to a significant degree depends on statistics. The actual specifications are written as *per-hop* behaviors (hence the PHB name); with appropriate admission control these per-hop behaviors will translate into the desired larger-scale behavior.

One possible Internet arrangement is that a leaf domain or ISP would support RSVP, but hands traffic off to some larger-scale carrier that runs DiffServ instead. Traffic with RSVP reservations would be marked on entry to the second carrier with the appropriate DS class.

20.7.1 Expedited Forwarding

The goal of the EF PHB is to provide low queuing delay to the marked packets; the canonical example is VoIP traffic, even though the latter now has its own PHB. EF is generally considered to be the best DS service, though this depends on how much EF traffic is accepted by the DS domain.

Each router in a DS domain supporting EF is configured with a committed rate, R, for EF traffic. Different routers can have different committed rates. At any one router, **RFC 3246** spells out the rule this way (note that this rule does indeed express a per-*hop* behavior):

> Intuitively, the definition of EF is simple: the rate at which EF traffic is served at a given output interface should be at least the configured rate R, over a suitably defined interval, independent of the offered load of non-EF traffic to that interface.

To the EF traffic, in other words, each output interface should *appear* to offer bandwidth R, with no competing non-EF traffic. In general this means that the network should appear to be lightly loaded, though that appearance depends very much on strict control of entering EF traffic. Normally R will be well below the physical bandwidths of the router's interfaces.

RFC 3246 goes on to specify how this apparent service should work. Roughly, if EF packets have length L then they should be sent at intervals L/R. If an EF packet arrives when no other EF traffic is waiting, it can be held in a queue, but it should be sent soon enough so that, when physical transmission has ended, no more than L/R time has elapsed in total. That is, if R and L are such that L/R is 10µs, but the physical bandwidth delay in sending is only 2µs, then the packet can be held up to 8µs for other traffic.

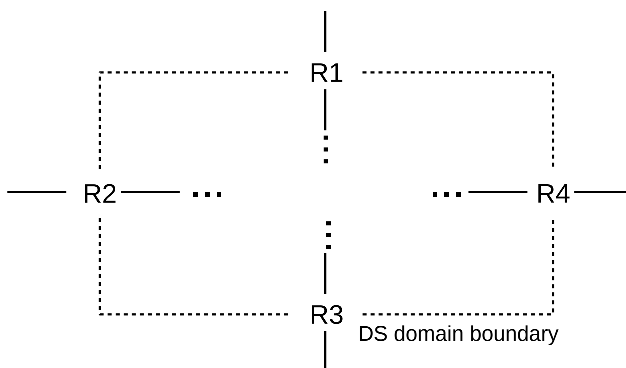
Note that this does *not* mean that EF traffic is given strict priority over all other traffic (though implementation of EF-traffic processing via priority queuing is a reasonable strategy); however, the sending interface must provide service to the EF queue at intervals of no more than L/R; the EF rate R must be in effect at per-packet time scales. Queuing ten EF packets and then sending the lot of them after time 10L/R is not allowed. Fair queuing can be used instead of priority queuing, but if quantum fair queuing is used then the quantum must be small.

An EF router's committed rate R means simply that the router has promised to reserve bandwidth R for EF traffic; if EF traffic arrives at a router faster than rate R, then a queue of EF packets may build up (though the router *may* be in a position to use some of its additional bandwidth to avoid this, at least to a degree). Queuing delays for EF traffic may mean that someone's application somewhere fails rather badly, but the router cannot be held to account. As long as the total EF traffic arriving at a given router is limited to that routers' EF rate R, then at least that router will be able to offer good service. If the arriving EF traffic meets a token-bucket specification TB(R,B), then the maximum number of EF packets in the queue will be B and the maximum time an EF packet should be held will be B/R.

So far we have been looking at individual routers. A DS domain controls EF traffic only at its border; how does it arrange things so none of its routers receives EF traffic at more than its committed rate?

One very conservative approach is to limit the *total* EF traffic entering the DS domain to the common committed rate R. This will likely mean that individual routers will not see EF traffic loads anywhere close to R.

As a less-conservative, more statistical, approach, suppose a DS domain has four border routers R1, R2, R3 and R4 through which all traffic must enter and exit, as in the diagram below:



Suppose in addition the domain knows from experience that exiting EF traffic generally divides equally between R1-R4, and also that these border routers are the bottlenecks. Then it might allow an EF-traffic entry rate of R at *each* router R1-R4, meaning a total entering EF traffic volume of 4×R. Of course, if on

some occasion all the EF traffic entering through R1, R2 and R3 happened to be addressed so as to exit via R4, then R4 would see an EF rate of $3\times$R, but hopefully this would not happen often.

If an individual ISP wanted to provide end-user DiffServ-based VoIP service, it might mark VoIP packets for EF service as they entered (or might let the customer mark them, subject to the ISP's policing). The rate of marked packets would be subject to some ceiling, which might be negotiated with the customer as a certain number of voice lines. These marked VoIP packets would receive EF service as they were routed within the ISP.

For calls also terminating within that ISP – or switching over to the traditional telephone network at an interface within that ISP – this would be all that was necessary, but some calls will likely be to customers of other ISPs. To address this, the original ISP might negotiate with its ISP neighbors for continued preferential service; such service might be at some other DS service class(*eg* AF). Packets would likely need to be re-marked as they left the original ISP and entered another.

The original ISP may have one larger ISP in particular with which it has a customer-provider relationship. The larger ISP might feel that with its high-volume internal network it has no need to support preferential service, but might still agree to carry along the original ISP's EF marking for use by a third ISP down the road.

20.7.2 Assured Forwarding

AF, documented in **RFC 2597**, is simpler than EF, but with little by way of a delay guarantee. The macroscopic goal is to grant specific rate assurances to certain traffic classes, *eg* the traffic from a certain set of customers. Distinct traffic classes are represented by distinct AF classes, which are limited to four. If a contributor to a traffic class sends more traffic than that particular contributor is permitted, the excess traffic is simply marked with a higher drop precedence. Further on in the network, the traffic may or may not be dropped.

The drop-precedence values for the second three bits for the AF classes are as follows:

- 010: do not drop
- 100: medium
- 110 high

Different priority-queuing levels may used for the different AF classes; this would ensure that all the traffic needs of AH class 4 are met before AH class 3, and down the line to AH class 1 at the lowest AH priority, just above bulk (default) traffic. Provided careful limits are placed on how much traffic is admitted to each class, strict priority queuing need not lead to the starvation of bulk traffic. Use of priority queuing is not mandatory; another option is fair queuing where the higher-precedence AH classes get a greater fair-queuing-guaranteed fraction of the total bandwidth, at least relative to their traffic volumes.

Traffic with different drop-precedence values within a single AF class, however, is *not* assigned to different subqueues (Priority, Fair or otherwise). Doing that would almost certainly lead to significant reordering of the overall traffic, and reordering tends to be bad for TCP traffic. If different priority-queuing levels were used here, for example, higher-drop-precedence packets would be delayed until lower-drop-precedence queues were emptied; almost any form of queuing using multiple queues for a different output interface would likely lead to at least some reordering. So long as one TCP connection remains in a single AF class

(*eg* because all traffic to or from a given site remains in a single AF class), the packets of that connection should not be reordered.

A classic application of AF is for an ISP to be able to grant different performance levels to different customers: gold, silver and bronze (again from the Appendix to **RFC 2597**). Customers would pay an ISP more to carry their traffic in a higher category. Along with their AF level, each customer would negotiate with the ISP their average rate and also their "committed" and "excess" burst (bucket) capacities. As the customer's traffic entered the network, it would encounter two token-bucket filters, with rate equal to the agreed-upon rate and with the two different bucket sizes: TB(r,$B_{committed}$) and TB(r,B_{excess}). Traffic compliant with the first token-bucket specification would be marked "do not drop"; traffic noncompliant for the first but compliant for the second would be marked "medium" and traffic noncompliant for either specification would be marked with a drop precedence of "high". (The use of B_{excess} here corresponds to the sum of "committed burst size" and "excess burst size" in the Appendix to **RFC 2597**.)

Customers would thus be free to send faster than their agreed-upon rate, subject to the excess traffic being marked with a lower drop precedence.

This process means that an ISP can have many different Gold customers, each with widely varying rate agreements, all lumped together in the same AF-4 class. The individual customer rates, and even the sum of the rates, may have only a tenuous relationship to the actual internal capacity of the ISP, although it is not in the ISP's long-term interest to oversubscribe any AF level.

If the ISP keeps the AF class 4 traffic sparse enough, it might outperform EF traffic in terms of delay. The EF PHB rules, however, explicitly address delay issues while the AF rules do not.

20.8 RED with In and Out

Differentiated Services Assured Forwarding (AF) fits nicely with **RIO** routers: RED with In and Out (or In, Middle, and Out); see *14.8.3 RED*. In RIO routers, each drop-precedence value (the "In", "Middle" and "Out" here) is subject to a different drop threshold. Packets with higher drop-precedence values would experience RED signaling losses at correspondingly lower degrees of RED queue utilization. This means that TCP connections with a significant fraction of higher-drop-precedence values would encounter RED-induced losses sooner – and would thus reduce their congestion window sooner – than connections that stayed within their committed rate. Because each AF class is sent at a single priority, TCP connections within a single AF class should not experience problems(*eg* reordering) other than these RED signaling losses. All this has the effect of gently encouraging customers to stay within their committed traffic limits.

RIO is likely to have a meaningful effect only on TCP connections; real-time UDP traffic may be affected slightly or not at all by RED signaling-type losses.

20.9 NSIS

The practical problem with RSVP is the need for routers to participate. One approach to gaining ISP cooperation might be a lighter-weight version of RSVP, though that is speculative; Differentiated Services was supposed to be just that and it too has not been widely adopted within the commercial Internet backbone.

That said, work has been underway for quite some time now on a replacement protocol suite. One candidate is Next Steps In Signaling, or **NSIS**, documented in **RFC 4080** and **RFC 5974**.

NSIS breaks the RSVP design into two separate layers: the **signal transport** layer, charged with figuring out how to reach the intermediate routers, and the higher **signaling application** layer, charged with requesting actual reservations. One advantage of this two-layer approach is that NSIS can be adapted for other kinds of signaling, although most NSIS signaling can be expected to be related to a specific network flow. For example, NSIS can be used to tell NAT routers to open up access to a given inside port, in order to allow a VoIP (or other) connection to proceed.

Generally speaking, NSIS also abandons the multicast-centric approach of RSVP. Signaling traffic travels hop-by-hop from one **NSIS Element**, or NE, to the next NE on the path. In cases when the signaling traffic follows the same path as the data (the "path-coupled" case), the signaling packet would likely be addressed to the ultimate destination, but *recognized* by each NE router on the path. NE routers would then add something to the packet, and perhaps update their own internal state. Nonparticipating (non-NE) routers would simply forward the signaling packet, like any other IP packet, further towards its ultimate destination.

20.10 Comcast Congestion-Management System

The large ISP Comcast introduced a DiffServ-like scheme to reduce internal congestion; this was eventually documented in **RFC 6057**. The effect of the scheme is to reduce bandwidth for those subscribers who are using the largest amount of bandwidth; this reduction is done only when the ISP's local network is experiencing significant congestion. The goal is to avoid saturation of the ISP-level backbone, thus perhaps improving fairness for other users.

By throttling bulk traffic throughout its network before maximum capacities are reached, this strategy has the potential to improve the performance of real-time protocols without IntServ- or DiffServ-type mechanisms.

During non-congested operation, all user traffic is tagged within Comcast's network as **Priority Best Effort** (PBE). This is akin to a medium drop precedence for a particular DiffServ Assured Forwarding class (*20.7.2 Assured Forwarding*), though **RFC 6057** does not claim that DiffServ is actually used. When congestion occurs, traffic of some high-volume users may be tagged instead as **Best Effort** (BE), meaning that it has a lower priority and a higher drop precedence.

The mechanism first defines a **Near-Congestion State** for ports on routers at a certain level of Comcast's network. These routers are known as Cable Modem Termination Systems, or CMTS's. Each CMTS node serves about 5,000 subscribers. Near-Congestion State is declared when the port utilization exceeding a specific percentage of its maximum for a specific amount of time; typically the utilization threshold is 70-80% and the time interval is 15 minutes. One CMTS port typically serves 200-300 customers. Previous experience had established CMTS ports as the likely locus for congestion.

When a particular CMTS port enters the Near-Congestion State, the port's traffic is monitored on a per-subscriber basis to see if any subscribers are in an **Extended High Consumption State**, or EHCS, again triggered when a subscriber exceeds a given percentage of their subscriber rate for a given number of minutes. The threshold for entering EHCS state is typically 70% of the subscriber rate – either the upstream rate or the downstream rate – for 15 minutes.

When the switch port is in Near-Congestion State *and* the subscriber's traffic is in EHCS, then all that subscriber's traffic is marked **Best Effort** (BE) rather than PBE, corresponding to a higher AF drop precedence. Downstream traffic is marked (at the CMTS or even higher up in the network) if it is the user's downstream utilization that is high; upstream traffic is marked at the user's connection to the network if upstream utilization is high.

Other routers may then drop BE-marked traffic preferentially, versus PBE-marked traffic. This will occur only at routers that are actively experiencing congestion, perhaps using a mechanism like RIO (*20.8 RED with In and Out*), though RIO was originally intended only for TCP traffic.

A subscriber leaves the EHCS state when the subscriber's bandwidth drops below 50% of the subscription rate for 15 minutes, where presumably the 50% rate is measured over some very short timescale. Note that this means that a user with a TCP sawtooth ranging from 30% to 60% of the maximum might remain in the EHCS state indefinitely.

Also note that *all* the subscriber's traffic will be marked as BE traffic, not just the overage. The intent is to provide a mild disincentive for sustained utilization within 70% of the subscriber maximum rate.

Token bucket specifications are not used, except possibly over very small timescales to define utilizations of 50% and 70%. The (larger-scale) token-bucket alternative might be to create a token-bucket specification for each customer TB(r,B) where r is some fraction of the subscription rate and B is a bucket of modest size. All compliant traffic would then be marked PBE and noncompliant traffic would be marked BE. Such a mechanism might behave quite differently, as only traffic actually over the ceiling would be marked.

20.11 Real-time Transport Protocol (RTP)

RTP is a convenient framework for carrying real-time traffic. It runs on top of UDP, largely to avoid head-of-line blocking (*11.1 User Datagram Protocol – UDP*), and offers several advantages over raw UDP. It does not, however, involve interactions with the intervening routers, and therefore cannot offer any service guarantees.

RTP headers include a sequence number, an application-provided timestamp representing when the attached data was recorded, and basic mechanisms for handling traffic that is a merger of several original sources ("contributing sources") of related content. RTP also includes support for multicast transmission, *eg* for the voice and video streams that make up a teleconference. Indeed, teleconferencing is arguably the application for which RTP was developed, although it is also widely used for point-to-point applications such as VoIP.

Perhaps the most striking feature of RTP is the absence of frequent acknowledgments. There is indeed a provision for receiver responses, but they are often sent only at several-second intervals, and are frequently omitted entirely. These responses use the companion RTCP protocol, and the "receiver report" format. RTCP receiver-report packets are often thought of not as acknowledgments but as a source of *statistics* about how the RTP flow is being delivered; in particular, the RTCP packets contain a ratio of how many sender packets arrived since the previous RTCP report, the highest sequence number received, and a measure of the degree of jitter.

For multicast, the infrequent acknowledgments make sense. If every receiver of a multicast group sent acknowledgments totaling 3% of the received-content bandwidth (about the TCP ratio), and if there were 100 receivers in the group (not large for a teleconference), then the total acknowledgment traffic arriving at the sender would be triple the content traffic. There are, in fact, mechanisms in place to limit the total RTCP bandwidth to no more than 5% of the outbound content stream; if a multicast stream has thousands of receivers then those receivers will each respond relatively infrequently (*eg* at intervals of many seconds, if not many minutes).

Even in one-to-one transmission settings, though, RTP may not send many acknowledgments. Many VoIP systems are configured not to send RTCP responses at all by default; when these *are* enabled, the rate has a typical minimum of one RTCP response per second. In one second, a VoIP sender may transmit 50 packets.

One explanation for this is that when a voice call is encountering significant congestion, the participants *are expected to hang up*, rather than keep the line open.

For two-way voice calls, **symmetric RTP** is often used (**RFC 4961**). This means that each party uses the same port for sending and receiving. This is done only for convenience and to handle NAT routers; the two separate directions are completely independent and do *not* serve as acknowledgments for one another, as would be the case for bidirectional TCP traffic. Indeed, one can block one direction of a VoIP symmetric-RTP stream and the call continues on indefinitely, transmitting voice only in the other direction. When the block is removed, the blocked voice flow simply resumes.

20.11.1 RTP Mixers

Mixers are network nodes that take one or more RTP source streams and perform any combination of the following operations

- consolidation of multiple streams into one
- translation to a different audio or video format
- re-encoding to a lower-bandwidth format

As an example of consolidation, video of several panelists might be consolidated into a single video frame, in which the current speaker has a larger window. At the audio level, a mixer might combine the streams from several individual microphones. If all parties are at the same location this kind of consolidation can be performed by the primary sender, but mixers may be useful if conference participants are geographically dispersed. As for translation, some participants may prefer to receive in a different format.

Perhaps the most important task for mixers, however, is rate-adaptation. With a unicast connection, if congestion is experienced then the sender itself can adapt to it by switching the encoding. This is not appropriate for multicast, however; if one receiver is experiencing congestion it is not unlikely that other receivers may be doing just fine. Therefore, if a multicast receiver is having trouble receiving at a high data rate, it is up to it to find a mixer that offers a lower-rate encoding, and switch its subscription/reservation to that mixer instead. Mixers offering a range of rates might be set up near the sender (or at least logically near); they might also be geographically distributed to be nearer to clusters of likely receivers. One host might provide several mixers offering a range of bandwidth options.

One thing mixers do *not* do is mix audio and video together; separate audio and video streams should be carried by separate RTP sessions. RTP packets carry timestamps to allow a receiver to synchronize audio and video playback, so mixing is not necessary. But more importantly, mixing of audio and video makes it difficult to re-encode to a different format, difficult for a receiver to subscribe only to the audio, and difficult for the RTCP reports to indicate which original stream was encountering more losses.

An individual sending point in RTP, complete with timing information, is called a **synchronization source** or SSRC. At the point of origin, each camera and microphone might be an SSRC. SSRCs are identified by a 32-bit identifier. Actual SSRC identifiers are to be chosen pseudorandomly (and there is a provision in the protocol for making sure two different sources do not choose the same SSRC identifier), but for many practical purposes the SSRC can be thought of as a host identifier, like an IP address.

When a mixer combines multiple SSRCs into one stream (or even when a mixer takes as input only a single SSRC), the mixer becomes the new synchronization source and creates a new SSRC identifier for all its output packets; the mixer also generates a new series of synchronization timestamps. However, the SSRC

identifiers for the streams that the mixer used as input are attached to all RTP packets from the mixer; each of these is now known as a **contributing source** or CSRC.

20.11.2 RTP Packet Format

The basic RTP header has the following format, from **RFC 3550**:

Ver	P	X	CC	M	Payload Type	Sequence Number
Timestamp						
Synchronization Source (SSRC) Identifier						
Contributing Source (CSRC) Identifiers						

The **Ver** field holds the version, currently 2. The **P** bit indicates that the RTP packet includes at least one padding byte at the end; the final padding byte contains the exact count.

The **X bit** is set if an extension header follows the basic header; we do not consider this further.

The **CC field** contains the number of contributing source (CSRC) identifiers (below). The total header size, in 32-bit words, is 3+CC.

The **M bit** is set to allow the marking, for example, of the first packet of each new video frame. Of course, the actual video encoding will also contain this information.

The **Payload Type** field allows (or, more precisely, originally allowed) for specification of the audio/visual encoding mechanism (*eg* µ-law/G.711), as described in **RFC 3551**. Of course, there are more than 2^7 possible encodings now, and so these are typically specified via some other mechanism, *eg* using an extension header or the separate Session Description Protocol (**RFC 4566**) or as part of the RTP stream's "announcement". **RFC 3551** put it this way:

> During the early stages of RTP development, it was necessary to use statically assigned payload types because no other mechanism had been specified to bind encodings to payload types.... Now, mechanisms for defining dynamic payload type bindings have been specified in the Session Description Protocol (SDP) and in other protocols....

The **sequence-number** field allows for detection of lost packets and for correct reassembly of reordered packets. Typically, if packets are lost then the receiver is expected to manage without them; there is no timeout/retransmission mechanism in RTP.

The **timestamp** is for synchronizing playback. The timestamp should be sufficiently fine-grained to support not only smooth playback but also the measurement of *jitter*, that is, the degree of variation in packet arrival.

Each encoding mechanism chooses its own timestamp granularity. For most telephone-grade voice encodings, for example, the timestamp clock increments at the canonical sampling rate of 8,000 times a second, corresponding to one DS0 channel (*4.2 Time-Division Multiplexing*). **RFC 3551** suggests a timestamp clock rate of 90,000 times a second for most video encodings.

Many VoIP applications that use RTP send 20 ms of voice per packet, meaning that the timestamp is incremented by 160 for each packet. The actual amount of data needed to send 20 ms of voice can vary from 160 bytes down to 20 bytes, depending on the encoding used, but the timestamp clock always increments at the 8,000/sec, 160/packet rate.

The **SSRC identifier** identifies the primary data source (the "synchronization source") of the stream. In most cases this is either the identifier for the originating camera/microphone, or the identifier for the mixer that repackaged the stream.

If the stream was processed by a mixer, the SSRC field identifies the mixer, and the SSRC identifiers of the original sources are now listed in the **CSRC** ("contributing sources") section of the header. If there was no mixer involved, the CSRC section is empty.

20.11.3 RTP Control Protocol

The RTP Control Protocol, or RTCP, provides a mechanism for exchange of a variety of extra information between RTP participants. RTCP packets may be Sender Reports (SR packets) or Receiver Reports (RR packets); we are mostly interested in the latter.

RTCP receiver reports are the only form of acknowledgments for RTP transmission. These are, however, unlike any other form of acknowledgment we have considered; for one thing, there is generally no question of retransmitting any lost data. RTCP RR packets should perhaps be thought of as statistical summaries regarding delivery rather than acknowledgments *per se*; in some cases they may in effect simply relay to senders the information "multicast is not working today".

RTCP packets contain a list of several SSRCs; for each SSRC, the message includes

- the fraction of RTP packets that were lost in the most recent interval
- the cumulative number of packets lost since the session began
- the highest sequence number received
- a measure of the interarrival jitter (below)
- for RR packets, information about the most recent RTCP SR packet

Jitter is measured as the mean deviation in actual arrival times, versus theoretical arrival times, and is measured in units of the RTP timestamp clock (above). The actual formula is as follows. For each packet P_i, we can record the actual time of arrival in RTP timestamp units as R_i and we can save the RTP timestamp included in the packet as S_i. We first calculate the ith deviation as follows:

$$Dev_i = (R_i - R_{i-1}) - (S_i - S_{i-1})$$

For evenly spaced packets, the deviation will be zero. For VoIP packets, $S_i - S_{i-1}$ is likely exactly 160; $R_i - R_{i-1}$ is the actual arrival interval in eighths of milliseconds.

We then calculate the cumulative jitter as follows, where α = 15/16:

$$J_i = \alpha \times J_{i-1} + (1-\alpha) \times Dev_i$$

In a multicast setting, if an RTP receiver is experiencing excessive losses, its most practical option is probably to find a mixer offering a lower-bandwidth encoding, or that is otherwise more likely to provide low-congestion service. RTCP packets are not needed for this.

In a unicast setting, an RTP sender can – at least theoretically – use the information provided by RTCP RRs to adapt its transmission rate downwards, to better make use of the available bandwidth. A sender *might* also use TFRC (*14.6.1 TFRC*) to calculate a TCP-friendly sending rate; there are Internet drafts for this (*[GP11]*), but as yet no RFC. TFRC also allows for a gradual adjustment in rate to accommodate the new conditions.

Of course, TFRC can only be used to adjust the sending rate if the sending rate is in fact adaptive, at the application level. This, of course, requires a rate-adaptive encoding.

RFC 3550 specifies a mechanism to make sure that RTCP packets do not consume more than 5% of the total RTP bandwidth, and, of that 5%, RTCP RR packets do not account for more than 75%. This is achieved by each receiver learning from RTCP SR packets how many other receivers there are, and what the total RTP bandwidth is. From this, and from some related information, each RTP receiver can calculate an acceptable RTCP reporting interval. This interval can easily be in the hundreds of seconds.

Mixers are allowed to consolidate RTCP information from their direct subscribers, and forward it on to the originating sources.

20.11.4 RTP and VoIP

Voice-over-IP data is often carried by RTP, although VoIP calls are initially set up using the Session Initiation Protocol, SIP (**RFC 3261**), or some other setup protocol such as H.323. As part of the call set-up process, the SIP nodes organizing the call exchange IP addresses and port numbers for the actual endpoints. Each endpoint then is informed of the address and port for the other endpoint, and the endpoints more-or-less immediately begin sending one another RTP packets. If an endpoint is behind a NAT firewall, its associated SIP server may have to continue forwarding packets.

VoIP calls typically send voice data in packets containing 20ms of audio input. If the standard DS0 G.711 encoding is used, 8 bytes are needed for each millisecond of voice and so each packet has 160ms of data. The G.711 encoding includes both μ-law and A-law encoders; these are both variations on the idea of having the 8-bit samples be proportional to the logarithm of the actual sound intensity.

Other voice encoders provide a greater degree of compression. For example, when G.729 voice encoding is used for VoIP then each packet will carry 20 ms of voice in 20 bytes of data. Each such packet will also, typically, have 54 bytes of header (14 bytes of Ethernet header, 20 bytes of IP header, 8 bytes of UDP header and 12 bytes of RTP header). Despite a payload efficiency of only 20/74 $\simeq$ 27%, this is generally considered acceptable, if not ideal.

VoIP connections often do not make much use of the data in the RTCP packets; it is not uncommon, in fact, for RTCP packets not even to be sent. The general presumption is that if the voice quality is not adequate, the users involved should simply hang up. The most common voice encoders (*eg* G.711 and, for that matter, G.729) are not rate-adaptive, and so if a call is encountering congestion there is nothing that can be done. That said, RTCP packets *may* provide useful information regarding jitter experienced by the call. Furthermore, as we mentioned above in *20.3 Real-time Traffic*, some voice codecs – such as Opus and Speex – do support rate-adaptiveness.

20.12 Multi-Protocol Label Switching (MPLS)

Currently, MPLS is a way of tagging certain classes of traffic, and perhaps routing them altogether differently from other classes of traffic. While IPv4 has always allowed routing based on tagged QoS levels, this feature was seldom used, and a common assumption for both Integrated and Differentiated Services was that the priority traffic essentially took the same route as everything else. MPLS allows priority traffic to take an entirely different route.

MPLS started out as a way of routing IP and non-IP traffic together, and later became a way to avoid the then-inefficient lookup of an IP address at every router. It has, however, now evolved into a way to support the following:

- creation of **explicit routes** for IP traffic, either in bulk or by designated class.
- large-scale virtual private networks (VPNs)

We are mostly interested in MPLS for the first of these; as such, it provides an alternative way for ISPs to handle real-time traffic. The explicit routes are defined using virtual circuits (*3.4 Virtual Circuits*).

In effect, virtual-circuit tags are added to each packet, on entrance to the routing domain, allowing packets to be routed along a predetermined path. The virtual circuit paths need not follow the same route that normal IP routing would use, though note that link-state routing protocols such as OSPF already allow different routes for different classes of service.

We note also that the MPLS-labeled traffic might very well use the same internal routers as bulk traffic; priority or fair queuing would then still be needed at those routers to make sure the MPLS traffic gets the desired level of service. However, the use of MPLS at least makes the **classification** problem easier: internal routers need only look at the tag to determine the priority or fair-queuing class, and deep-packet inspection can be avoided. MPLS would also allow the option that a high-priority flow would travel on a special path through its own set of routers that do *not* also service low-priority traffic.

Generally MPLS is used only *within* one routing domain or administrative system; that is, within the scope of one ISP. Traffic enters and leaves looking like ordinary IP traffic, and the use of MPLS internally is completely invisible. This local scope of MPLS, however, has meant that it has seen relatively widespread adoption, at least compared to RSVP and IP multicast: no coordination with other ISPs is necessary.

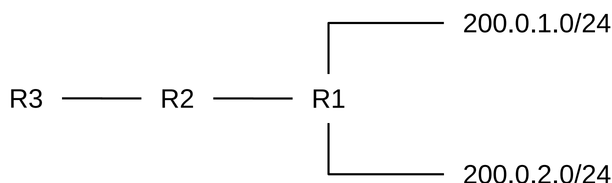
To implement MPLS, we start with a set of participating routers, called **label-switching routers** or LSRs. (The LSRs can comprise an entire ISP, or just a subset.) Edge routers partition (or *classify*) traffic into large flow classes; one distinct flow (which might, for example, correspond to all VoIP traffic) is called a **forwarding equivalence class** or FEC. Different FECs can have different quality-of-service targets. Bulk traffic not receiving special MPLS treatment is not considered to be part of any FEC.

A one-way virtual circuit is then created for each FEC. An MPLS header is prepended to each IP packet, using for the VCI a **label** value related to the FEC. The MPLS label is a 32-bit field, but only the first 20 bits are part of the VCI itself. The last 12 bits may carry supplemental connection information, for example ATM virtual-channel identifiers and virtual-path identifiers (*3.5 Asynchronous Transfer Mode: ATM*).

It is likely that some traffic (perhaps even a majority) does not get put into any FEC; such traffic is then delivered via the normal IP-routing mechanism.

MPLS-aware routers then add to their forwarding tables an MPLS table that consists of $\langle \text{label}_{\text{in}}$, $\text{interface}_{\text{in}}$, $\text{label}_{\text{out}}$, $\text{interface}_{\text{out}} \rangle$ quadruples, just as in any virtual-circuit routing. A packet arriving on interface $\text{interface}_{\text{in}}$ with label label_{in} is forwarded on interface $\text{interface}_{\text{out}}$ after the label is altered to $\text{label}_{\text{out}}$.

Routers can also build their MPLS tables incrementally, although if this is done then the MPLS-routed traffic will follow the same path as the IP-routed traffic. For example, downstream router R1 might connect to two customers 200.0.1/24 and 200.0.2/24. R1 might assign these customers labels 37 and 38 respectively.



R1 might then tell its upstream neighbors (*eg* R2 above) that any arriving traffic for either of these customers should be labeled with the corresponding label. R2 now becomes the "ingress router" for the MPLS domain consisting of R1 and R2.

R2 can push this further upstream (*eg* to R3) by selecting its own labels, *eg* 17 and 18, and asking R3 to label 200.0.1/24 traffic with label 17 and 200.0.2/24 with 18. R2 would then rewrite VCIs 17 and 18 with 37 and 38, respectively, before forwarding on to R1, as usual in virtual-circuit routing. R2 might not be able to continue with labels 37 and 38 because it might already be using those for inbound traffic from somewhere else. At this point R2 would have an MPLS virtual-circuit forwarding table like the following:

$interface_{in}$	$label_{in}$	$interface_{out}$	$label_{out}$
R3	17	R1	37
R3	18	R1	38

One advantage here of MPLS is that labels live in a flat address space and thus are easy and simple to look up, *eg* with a big array of 65,000 entries for 16-bit labels.

MPLS can be adapted to multicast, in which case there might be two or more $label_{out}$, $interface_{out}$ combinations for a single input.

Sometimes, packets that already have one MPLS label might have a second (or more) label "pushed" on the front, as the packet enters some designated "subdomain" of the original routing domain.

When MPLS is used throughout a domain, ingress routers attach the initial label; egress routers strip it off. A label information base, or LIB, is maintained at each node to hold any necessary packet-handling information (*eg* queue priority). The LIB is indexed by the labels, and thus involves a simpler lookup than examination of the IP header itself.

MPLS has a natural fit with Differentiated Services (*20.7 Differentiated Services*): the ingress routers could assign the DS class and then attach an MPLS label; the interior routers would then need to examine only the MPLS label. Priority traffic could be routed along different paths from bulk traffic.

MPLS also allows an ISP to create multiple, mutually isolated VPNs; all that is needed to ensure isolation is that there are no virtual circuits "crossing over" from one VPN to another. If the ISP has multi-site customers A and B, then virtual circuits are created connecting each pair of A's sites and each pair of B's sites. A and B each probably have at least one gateway to the whole Internet, but A and B can communicate with each other only through those gateways.

MPLS sometimes interacts somewhat oddly with traceroute (*7.11.1 Traceroute and Time Exceeded*). If a packet's TTL reaches 0 at an MPLS router, the router will usually generate the appropriate ICMP Time Exceeded message, but then attach to it the same MPLS label that had been on the original packet. As a result, the ICMP message will continue on to the *downstream* egress router before being sent back to the

original sender. If nothing else, this results in an unusually large round-trip-time report. To the traceroute initiator it will appear as if all the internal routers in the MPLS domain are the same distance from the initiator as the egress router.

20.13 Epilog

Quality-of-service guarantees for real-time and other classes of traffic have been an area of active research on the Internet for over 20 years, but have not yet come into the mainstream. The tools, however, are there. Protocols requiring global ISP coordination – such as RSVP and IP Multicast – may come slowly if at all, but other protocols such as Differentiated Services and MPLS can be effective when rolled out within a single ISP.

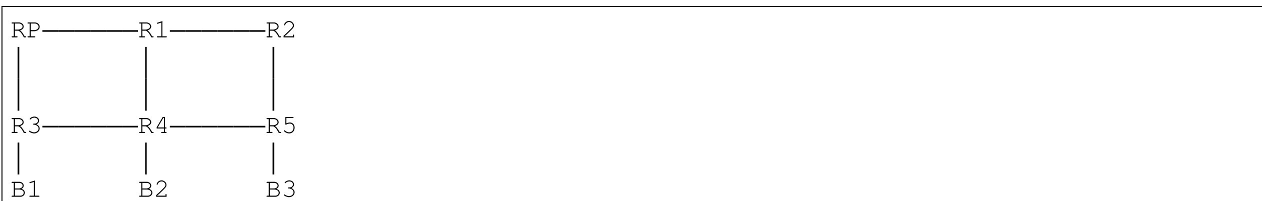
Still, after twenty years it is not unreasonable to ask whether integrated networks are in fact the correct approach. One school of thought says that real-time applications (such as VoIP) are only just beginning to come into the mainstream, and integrated networks are sure to follow, or else that video streaming will take over the niche once intended for real-time traffic. Perhaps IntServ was just ahead of its time. But the other perspective is that the marketplace has had plenty of opportunity to make up its mind and has answered with a resounding "no", and it is time to move on. Perhaps having separate networks for bulk traffic and for voice is not unreasonable or inefficient after all. Or perhaps the Internet will evolve into a network where *all* traffic is handled by real-time mechanisms. Time, as usual, may tell, but not, perhaps, quickly.

20.14 Exercises

1. Suppose someone proposes TCP over multicast, in which each router collects the ACKs returning from the group members reached through it, and consolidates them into a single ACK. This now means that, like the multicast traffic itself, no ACK is duplicated on any single link.

What problems do you foresee with this proposal? (Hint: who will send retransmissions? How long will packets need to be buffered for potential retransmission?)

2. In the following network, suppose traffic from RP to R3-R5 is always routed first right and then down, while traffic from R3–R5 to RP is always routed first left and then up. What is the multicast tree for the group G = {B1,B2,B3}?



3. What should an RSVP router do if it sees a PATH packet but there is no subsequent RESV packet?

4. In *20.7.1 Expedited Forwarding* there is an example of an EF router with committed rate R for packets with length L. If R and L are such that L/R is 10µs, but the physical bandwidth delay in sending is only 2µs, then the packet can be held up to 8µs for other traffic.

How large a bulk-traffic packet can this router send, in between EF packets? Your answer will involve L.

5. Suppose, in the diagram in *20.7.1 Expedited Forwarding*, EF was used for voice telephony and at some point calls entering through R1, R2 and R3 were indeed all directed to R4.

(a). How might the problem be perceived by users?
(b). How might the ISP respond to reduce the problem?

Note that the ISP has no control over who calls whom.

Network management, broadly construed, consists of all the administrative actions taken to keep a network running efficiently. This may include a number of non-technical considerations, *eg* staffing the help desk and negotiating contracts with vendors, but we will restrict attention exclusively to the technical aspects of network management.

The ISO and the International Telecommunications Union have defined a formal model for telecommunications and network management. The original model defined five areas of concern, and was sometimes known as FCAPS after the first letter of each area:

- fault
- configuration
- accounting
- performance
- security

Most non-ISP organizations have little interest in network accounting (the A in FCAPS is often replaced with "administration" for that reason, but that is a rather vague category). Network security is arguably its own subject entirely. As for the others, we can identify some important subcategories:

fault:

- **device management**: monitoring of all switches, routers, servers and other network hardware to make sure they are running properly.
- **server management**: monitoring of the network's application layer, that is, all network-based software services; these include login authentication, email, web servers, business applications and file servers.
- **link management**: monitoring of long-haul links to ensure they are working.

configuration:

- network **architecture**: the overall design, including topology, switching vs routing and subnet layout.
- **configuration management**: arranging for the consistent configuration of large numbers of network devices.
- **change management**: how does a site roll out new changes, from new IP addresses to software updates and patches?

performance:

- **traffic management**: using the techniques of *19 Queuing and Scheduling* to allocate bandwidth shares (and perhaps bucket sizes) among varying internal or external clients or customers.
- **service-level management**: making sure that agreed-upon service targets – *eg* bandwidth – are met (depending on the focus, this could also be placed in the fault category).

While all these aspects are important, technical network management narrowly construed often devolves to an emphasis on fault management and its companion, reliability management: making sure nothing goes wrong, and, when it does, fixing it promptly. It is through fault management that some network providers achieve the elusive availability goal of 99.999% uptime.

SNMP *versus* Management

While SNMP is a very important *tool* for network management, it *is* just a tool. Network management is the process of making decisions to achieve the goals outlined above, subject to resource constraints. SNMP simply provides some input for those decisions.

By far the most common device-monitoring protocol, and the primary focus for this chapter, is the **Simple Network Management Protocol** or **SNMP** (*21.2 SNMP Basics*). This protocol allows a device to report information about its current operational state; for example, a switch or router may report the configuration of each interface and the total numbers of bytes and packets sent via each interface.

Implicit in any device-monitoring strategy is initial device **discovery**: the process by which the monitor learns of new devices. The ping protocol (*7.11 Internet Control Message Protocol*) is common here, though there are other options as well; for example, it is possible to probe the UDP port on a node used for SNMP – usually 161. As was the case with router configuration (*9 Routing-Update Algorithms*), manual entry is simply not a realistic alternative.

SNMP and the Application Layer

SNMP can be studied entirely from a network-management perspective, but it also makes an excellent self-contained case study of the application layer. Like essentially all applications, SNMP defines rules for client and server roles and for the format of requests and responses. SNMP also contains its own authentication mechanisms (*21.11 SNMPv1 communities and security* and *21.15 SNMPv3*), generally unrelated to any operating-system-based login authentication.

It is a practical necessity, for networks of even modest size, to automate the job of checking whether everything is working properly. Waiting for complaints is not an option. Such a monitoring system is known as a **network management system** or **NMS**; there are a wide range of both proprietary and open-source NMS's available. At its most basic, an NMS consists of a library of scripts to discover new network devices and then to poll each device (possibly but not necessarily using SNMP) at regular intervals. Generally the data received is recorded and analyzed, and alarms are sounded if a failure is detected.

When SNMP was first established, there was a common belief that it would soon be replaced by the OSI's Common Management Interface Protocol. **CMIP** is defined in the International Telecommunication Union's X.711 protocol and companion protocols. CMIP uses the same ASN.1 syntax as SNMP, but has a richer operations set. It remains the network management protocol of choice for OSI networks, and was once upon a time believed to be destined for adoption in the TCP/IP world as well.

But it was not to be. TCP/IP-based network-equipment vendors never supported CMIP widely, and so any network manager had to support SNMP as well. With SNMP support essentially universal, there was never a need for a site to bother with CMIP.

CMIP, unlike SNMP, is supported over TCP, using the CMIP Over Tcp, or **CMOT**, protocol. One advantage of using TCP is the elimination of uncertainty as to whether a request or a reply was received.

21.1 Network Architecture

Before turning to SNMP in depth, we offer a few references to other parts of this book relating to **network architecture**. At the LAN and Internetwork layers local to a site, perhaps the main issues are redundancy, bandwidth and cost. Cabling between buildings, in particular, needs to provide redundancy. See *2.5 Spanning Tree Algorithm and Redundancy* and *2.6 Virtual LAN (VLAN)* for some considerations at the Ethernet level, and *7.6.3 Subnets versus Switching*.

Next, a site must determine what sort of connection to the Internet it will have. ISP contracts vary greatly in terms of bandwidth, burst bandwidth, and agreed-upon responses in the event of an outage. Some aspects of service-level specification appear in *19.9 Token Bucket Filters* and *20.7.2 Assured Forwarding*.

Organizations with geographically dispersed internal networks – ISPs and larger corporations – must decide how their internal sites should be connected. Should they communicate over the public Internet? Should a VPN be used (*3.1 Virtual Private Networks*)? Or should private lines (such as SONET, *4.2.2 SONET*, or carrier Ethernet, *3.2 Carrier Ethernet*) be leased between sites? If private lines are used, link monitoring becomes essential.

One increasingly important architectural decision at the application layer is the extent to which network software services are outsourced to the cloud, and run on remote servers managed by third parties.

21.2 SNMP Basics

SNMP is far and away the most popular protocol for supporting network device monitoring. At its most basic level, SNMP allows polling of individual designated device attributes, such as the system name or the number of packets received via interface `eth0`. Attributes may, however, be organized into records, sets and tables. Tables may be indexed contiguously, like an array – *eg* interface[1], interface[2], interface[3], *etc* – or sparsely – *eg* interface[1], interface[32767].

While the simplest routers and switches may have quite limited provisions for SNMP, virtually all "serious" networking hardware provides extensive support, for both standard sets of "basic" device attributes and for proprietary attributes as well. Devices with significant SNMP support are sometimes referred to as **managed** devices.

An SNMP node that replies to requests for information is known as an **SNMP agent**. The network node doing the SNMP querying is known as the **manager**; it may be part of an NMS or – more simply – be a standalone tool known as an SNMP browser or MIB browser (where MIB stands for Management Information Base, below). While most MIB browsers are understood to have a graphical user interface, there are also command-line tools to make SNMP queries, such as the `snmpget` and `snmpwalk` commands of the Net-SNMP project at net-snmp.org. These can be invoked by scripting languages to build a simple if rudimentary NMS.

SNMP runs exclusively over UDP. The choice of UDP was made to avoid the connection overhead of what was envisioned to be a simple request-reply protocol; if a manager polls 1,000 devices once a minute, that is 2,000 packets in all over UDP but might easily be 8,000 packets with TCP and the necessary SYN/FIN packets. This may be especially significant when the network is congested to near the point of failure.

The use of UDP does raise two problems: lost packets and having more data than will fit in one packet. For the simplest case of manager-initiated data requests, a manager can handle packet loss by polling a device

until a response is received. If a response (or even a request) is too big, the usual strategy is to use IP-layer fragmentation (*7.4 Fragmentation* and *8.5.4 IPv6 Fragment Header*). This is not ideal, but as most SNMP data stays within local and organizational networks it is at least workable.

Another consequence of the use of UDP is that every SNMP device needs an IP address. For devices acting at the IP layer and above, this is not an issue, but an Ethernet switch or hub has no need of an IP address for its basic function. Devices like switches and hubs that are to use SNMP must be provided with a suitable IP interface and address. Sometimes, for security, these IP addresses for SNMP management are placed on a private, more-or-less hidden subnet.

SNMP also supports the **writing** of attributes to devices, thus implementing a remote-configuration mechanism. As writing to devices has rather more security implications than reading from them, this mechanism must be used with care. Even for read-only SNMP, however, security is an important issue; see *21.11 SNMPv1 communities and security* and *21.15 SNMPv3*.

Writing to agents may be done either to configure the network behavior of the device – *eg* to bring an interface up or down, or to set an IP address – or specifically to configure the SNMP behavior of the agent.

Finally, SNMP supports asynchronous notification through its **traps** mechanism: a device can be configured to report an error immediately rather than wait until asked. While traps are quite important at many sites, we will largely ignore them here.

SNMP is sometimes used for server monitoring as well as device monitoring; alternatively, simple scripts can initiate connections to services and discern at least something about their readiness. It is one thing, however, to verify that an email server is listening on TCP port 25 and responds with the correct **RFC 5321** (originally **RFC 821**) EHLO message; it is another to verify that messages are accepted and then actually delivered.

21.2.1 SNMP versions

SNMP has three official versions, SNMPv1, SNMPv2 and SNMPv3. SNMPv1 made its first appearance in 1988 in a collection of RFCs starting with **RFC 1065** (updated in **RFC 1155**); the current definition for "core" attribute reporting was released as **RFC 1213** in 1991. We will return to this below in *21.10 MIB-2*.

SNMPv2 was introduced in 1993 with **RFC 1441**. Loosely speaking, SNMPv2 expanded on the basic information, starting with **RFC 1442** (currently **RFC 2578**), and also introduced improved techniques for managing tables.

SNMPv2 also included a proposed security mechanism, but it was largely rejected by the marketplace. Ultimately, a version of SNMPv2 that used the SNMPv1 "community" security mechanism (*21.11 SNMPv1 communities and security*) was introduced; see **RFC 1901**. This "community"-security version became known as SNMPv2**c**.

SNMPv3 then finally delivered a model for reasonably effective security. The "User-based Security Model" or **USM** was first proposed in 1998 in **RFC 2264**; the final 2002 version is in **RFC 3414**.

21.3 SNMP Naming and OIDs

A central part of the SNMP protocol is how to **name** each device attribute in a consistent manner. The naming scheme chosen was the **Object Identifier**, or **OID**, hierarchy.

Starting in 1985, the International Standards Organization (ISO) and the standardization sector of the International Telecommunications Union (ITU-T, then known as CCITT) developed the Object Identifier scheme for naming anything in the world that needed naming. Names, or OIDs, consist of strings of non-negative integers. In textual representation these component integers are often written separated by periods, *eg* 1.3.6.1; the notation { 1 3 6 1 } is also used. The scheme is standardized in ITU-T X.660.

All OIDs used by SNMP begin with the prefix 1.3.6.1. For example, the `sysName` attribute corresponds to OID 1.3.6.1.2.1.1.5. The prefix 1.3.6.1.2.1 is known as `mib-2`, and the one-step-longer prefix 1.3.6.1.2.1.1 is `system`. Occasionally we will abuse notation and act as if names referred to single additional levels rather than full prefixes, *eg* the latter two names in `mib-2.system.sysName`.

The basic SNMP read operation is `Get()` (sent via the `GetRequest` protocol message), which takes an OID as parameter. The agent receiving the `Get` request will, if authentication checks out and if the OID corresponds to a valid attribute, return a pair consisting of the OID and the attribute value. We will return to this in *21.8 SNMP Operations*, and see how multiple attribute values can be requested in a single operation.

OIDs form a tree or hierarchy; the immediate child nodes of, say, 1.3.6.1 of length 4 are all nodes 1.3.6.1.N of length 5. The root node, with this understanding, is anonymous; OIDs are sometimes rendered with a leading "." to emphasize this: .1.3.6.1. Alternatively, the numbers can be thought of as labels on the *arcs* of the tree rather than the nodes.

There is no intrinsic way to distinguish internal OID nodes – prefixes – from leaf OID nodes that correspond to actual named objects. Context is essential here.

It is common to give the numeric labels at any specific level human-readable string equivalents. The three nodes immediately below the root are, with their standard string equivalents

- itu-t(0)
- iso(1)
- joint-iso-itu-t(2)

The string equivalents can be thought of as external data; when OIDs are encoded only the numeric values are included.

OID naming has been adopted in several non-SNMP contexts as well. ISO and ITU-T both use OIDs to identify official standards documents. The X.509 standard uses OID naming in encryption certificates; see **RFC 5280**. The X.500 directory standard also uses OIDs; the related **RFC 4524** defines several OID classes prefixed by 0.9.2342.19200300.100.1. As a non-computing example, Health Level Seven names US healthcare information beginning with the OID prefix 2.16.840.1.113883.

As mentioned above, SNMP uses OIDs beginning with 1.3.6.1; in the OID tree these levels correspond to

- iso(1)
- org(3): organizations
- dod(6): the US Department of Defense, the original sponsor of the Internet
- internet(1), apparently the only sublevel of DOD

Here is a portion of the OID tree showing a few other assignments in the iso subtree, and highlighting the 1.3.6.1 prefix above. More entries are available from oid-info.com/get.

itu-t(0)

iso(1)
- standard(0)
- registration-authority(1)
- member-body(2)
- **identified-organization(3)**
  - nist(5)
  - **dod(6)**
  - icd-ecma(12) (European Computer Manufacturers Association)
  - oiw(14) (OSE Implementers Workshop)
  - edifactboard(15)
  - ewos(16) (European Workshop on Open Systems)
  - osf(22) (Open Software Foundation)
  - nordunet(23)
  - nato(26)
  - icao(27) (International Civil Aviation Organization)
  - ...
  - amazon(187) (the author has no idea if Amazon currently uses this)

joint-iso-itu-t(2)

This SNMP 1.3.6.1 prefix is spelled out in **RFC 1155** via the syntax

```
internet       OBJECT IDENTIFIER ::= { iso org(3) dod(6) 1 }
```

which means that the string `internet` has the type `OBJECT IDENTIFIER` and its actual value is the list to the right of `::=`: 1.3.6.1. The use of `iso` here represents the OID *prefix* .1, conceptually different from just the level iso(1). The formal syntax here is **ASN.1**, as defined by the ITU-T standard X.680. We will expand on this below in *21.6 ASN.1 Syntax and SNMP*, though the presentation will mostly be informal; further details can be found at http://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx.

After the above, **RFC 1155** then defines the OID prefixes for management information, `mgmt`, and for private-vendor use, `private`, as

```
mgmt           OBJECT IDENTIFIER ::= { internet 2 }
private        OBJECT IDENTIFIER ::= { internet 4 }
```

Again, `internet` represents the newly defined prefix 1.3.6.1. Most general-purpose SNMP OIDs begin with the `mgmt` prefix. The `private` prefix is for OIDs representing vendor-specific information; for example, the prefix 1.3.6.1.4.1.**9** has been delegated to Cisco Systems, Inc. **RFC 1155** states that both the mgmt and private prefixes are delegated by the IAB to the IANA.

Responsibility for assigning names that begin with a given OID prefix can easily be delegated. The Internet Activities Board, for example, is in charge of 1.3.6.1. (According to **RFC 1155**, this delegation by the Department of Defense to the IAB was never made officially; the IAB just began using it.)

21.4 MIBs

A MIB, or **Management Information Base**, is a set of definitions that associate OIDs with specific attributes, and, along the way, associates each numeric level of the OIDs with a symbolic name. For example, below is the set of so-called `System` definitions in the core **RFC 1213** MIB known as MIB-2 (*21.10 MIB-2*). The `mib-2` and `system` prefixes are first defined as

```
mib-2          OBJECT IDENTIFIER ::= { mgmt 1 }
system         OBJECT IDENTIFIER ::= { mib-2 1 }
```

that is, 1.3.6.1.2.1 and 1.3.6.1.2.1.1 respectively. The `system` definitions, which represent actual attributes that can be retrieved, are as follows

```
sysDescr       { system 1 }
sysObjectID    { system 2 }
sysUpTime      { system 3 }
sysContact     { system 4 }
sysName        { system 5 }
sysLocation    { system 6 }
sysServices    { system 7 }
```

Most of these attributes represent string values that need to be administratively defined; we will return to these in *21.10 MIB-2*.

The MIB is *not* the actual attributes themselves; the set of all ⟨OID,value⟩ pairs stored by an SNMP manager – perhaps the result of a single set of queries, perhaps the result of a sequence of queries over time – is sometimes known as the management database, or MDB, though that term is not as universal.

Colloquially, a MIB can be either the abstract set of OID definitions or a particular ASCII file that defines the set. The latter must be run through a **MIB compiler** to be used by other software; users of a MIB browser generally find the information much more useful if the appropriate MIB file is loaded before making SNMP queries. Some RFCs can be read directly by the MIB compiler, which knows to edit out the non-MIB discussion.

The Net-SNMP package contains SNMP agents for linux and Macintosh computers (Microsoft has their own SNMP-agent software), and command-line tools for making SNMP queries; these latter tools are also available for Windows. The sysName value, for example, can be retrieved with the `snmpget` command

```
snmpget -v1 -c public localhost 1.3.6.1.2.1.1.5.0
```

The OID here corresponds to { system 5 0 }; the final 0 represents a SNMP convention that indicates this is a scalar value and is not part of a table. See *21.9 MIB Browsing* for further examples.

MIBs serve as both documentation and data. As documentation, they tell the implementer of a given SNMP agent what information is to be returned for each OID request. As data, they serve to translate numeric OIDs into human-readable data. For example, if the above `snmpget` command understands the appropriate MIB (*eg* because the MIB file is in the right place), we can type

```
snmpget -v1 -c public localhost sysName.0
```

It is not uncommon for an installation to involve several hundred different (possibly overlapping) MIBs, each defining a different portion of the OID tree. This is particularly true of the `private` subtree 1.3.6.1.4.1, for which one might have a separate MIB file for each manufacturer and device.

21.5 SNMPv1 Data Types

SNMP uses OIDs as the indexes for retrieval of attributes. The attributes themselves can have the following types, as of **RFC 1155**:

Datatype	Description
INTEGER	A 32-bit integer
OCTET STRING	Either a text string or a network address
Counter	A non-negative INTEGER that increases from 0, and can wrap around.
Gauge	An INTEGER that can rise and fall but that never wraps around
TimeTicks	An INTEGER used to measure time in 1/100ths of a second
OBJECT IDENTIFIER	An OID appearing as data
IpAddress	An IPv4 address, as an OCTET STRING of fixed length 4
NetworkAddress	An IpAddress, in theory allowing for other kinds of addresses later
Opaque	Any other data, ultimately an OCTET STRING

The type of the `sysName` attribute, for example, is OCTET STRING.

When SNMP data values are returned by an agent, they are tagged with their type. Thus, the receiver can See *21.12 SNMP and ASN.1 Encoding* for further details.

When the INTEGER type is used to represent an enumerated type, the value of zero must not be used; an example is the status type `up(1)`, `down(2)`, `testing(3)`.

21.6 ASN.1 Syntax and SNMP

We have already seen the ASN.1 definition of some OBJECT IDENTIFIERs in *21.3 SNMP Naming and OIDs*.

The attribute corresponding to any single OID is always a scalar type, *eg* one of the above. However, SNMP also uses some composite type definitions:

> SEQUENCE { *fieldname1 type1*, *fieldname2 type2*, ... }: This defines a **record** type.
>
> SEQUENCE OF *type*: This defines a homogeneous list – an array or table – of objects of type *type*. In most cases, *type* represents a SEQUENCE type, and so the list is a list of records, or table in the database sense.

Note the importance of the keyword OF here to distinguish records from lists.

The OBJECT-TYPE macro is heavily used in MIB definitions. It has the format

> *objname* `OBJECT-TYPE`
>
> `SYNTAX` *type*
>
> `ACCESS` read-write *or* read-only *or* write-only *or* not-accessible
>
> `STATUS` mandatory *or* optional *or* obsolete
>
> `DESCRIPTION` *descriptive string*

INDEX *OID used for table indexing*

::= *OID assigned to objname*

The *objname*, sometimes called the OBJECT DESCRIPTOR, is intended to be globally unique. The value for SYNTAX must be a valid ASN.1 syntax specification; see *21.10 MIB-2* for examples. The values for ACCESS and STATUS are, in each case, one of the literal strings shown above. The value for DESCRIPTION is optional.

Many of the objects so defined will represent tables or other higher-level structures; in this case the Object ID of the last line will represent an internal node of the OID tree rather than a leaf node, and the ACCESS will be not-accessible. It is not that the table is actually inaccessible, but rather that the attributes must be retrieved one by one rather than collectively.

Here is a concrete simple example; other examples appear in the following section. The OID value of ifEntry here is 1.3.6.1.2.1.2.2.1.

```
ifInOctets OBJECT-TYPE
    SYNTAX  Counter
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
            "The total number of octets received on the
            interface, including framing characters."
    ::= { ifEntry 10 }
```

The data type definitions and the above ASN.1 syntax used for are collectively known as the **structure of management information**, or SMI. The SMI in effect defines all the types and structures needed by the various MIBs. For SNMPv1, the SMI is defined in **RFC 1155**.

21.7 SNMP Tables

We have seen in *21.4 MIBs* how SNMP organizes the OID names for the system group, which consists of scalar values. Each agent attribute is assigned a permanent OID name; *eg* { system 5 0 } is the full OID for the sysName attribute. OIDs for scalar (non-table) attributes always include a final ".0".

Most agent data, however, comes in the form of **tables**, that is, lists of records. In SNMP, they are sometimes referred to as *conceptual* tables, as the SNMP agent offering the table is often not responsible for its storage or physical organization. In this sense, SNMP tables resemble database "views".

These tables can be accessed by linear search or by primary key lookup; the key can be a single attribute or a set of attributes and is known as the **index**. SNMP does not support secondary keys on tables, though it would in principle be possible for an agent to present the same data in two tables, each with a different index (a related example of this trick appears in *21.14.1.5 Matrix*).

SNMP tables are almost always small enough that the agent keeps all the records in main memory; database-style concerns about disk-storage organization and indexing are not an issue.

As in databases, an SNMP table index acts as a *constraint*, disallowing two distinct rows with identical index values. When an SNMP table mirrors a "real" table (*eg* the IP forwarding table), problems sometimes occur when the SNMP index is too "small" – has too few attributes – and so the SNMP table may not be able to accurately represent the original table. See *21.13.11.2 IP-Forward MIB* for an example.

In the world of databases, a key with too *many* attributes may fail to capture an important constraint; it may also be less efficient. Neither is a concern with SNMP, with the exception of tables that support row creation, which we may take to be a special case. SNMP table indexes may thus sometimes include attributes that from a database perspective would likely be omitted from the key; again, see *21.13.11.2 IP-Forward MIB* for examples.

We will look here at three examples of SNMPv1 tables, each part of MIB-2. The goal right now is to present how the tabular structure is mapped onto the underlying OID-tree structure; we will return to the syntactic definition of tables in *21.10.2 Table definitions and the interfaces Group* and to the semantics of each table in *21.10 MIB-2*. SNMPv2 has made some modest improvements to how tables are defined.

The first example will be the SNMP **interface table**, `ifTable`. For each network interface (*eg* loopback, Ethernet 0, Ethernet 1, Wireless 0, Point-to-Point 0, *etc*) a collection of attributes including the device name, MTU, bitrate, physical address, and the number of bytes and packets received and sent. The index here (as we shall see later) is an abstractly assigned interface number, known as `ifIndex`. Here is some sample data, *not* showing all columns that the actual MIB-2 table includes, and not (yet) showing any OIDs. The header of the index column, ifIndex, is in italic; the value is a single integer. The `eth0` interface has a higher number for packets than bytes (octets) because the 32-bit `inOctets` value has wrapped around.

ifIndex	name	MTU	bitrate	inOctets	inPackets
1	lo	16436	10,000,000	171	3
2	eth0	1500	100,000,000	37155014	1833455677
3	eth1	1500	100,000,000	0	0
4	ppp0	1420	0	2906687015	2821825

Routes

Although the data here is mostly a mash-up of records from different sources, the 10.38.0.0 route below was a VPN route using the `ppp0` tunnel above. Otherwise, workstation forwarding tables often have just two entries, for the local subnet and the default route.

The SNMP **routing table**, `ipRouteTable`, based on the IP forwarding table of *1.10 IP - Internet Protocol*, has a column representing the destination network, a variety of status and information columns (*eg* for RouteAge), and a NextHop column; the index is the destination-network attribute. Again, here is some sample data with the index column again in italic. The index here – an IP address – is a compound object: a list of four integers. We return to this below.

dest	mask	metric	next_hop	type
0.0.0.0	0.0.0.0	1	192.168.1.1	indirect(4)
10.0.0.0	255.255.255.0	0	0.0.0.0	direct(3)
10.38.0.0	255.255.0.0	1	192.168.4.1	indirect(4)

(The `type` column indicates whether the destination is local or nonlocal; the host is on subnet 10.0.0.0/24 and so the middle entry involves local delivery. The use of `indirect(4)` and `direct(3)` is an example of an SNMP enumerated type.)

The indexing of the interfaces table is (usually) **dense**: the values for `ifIndex` are typically consecutive numbers. The routing table, on the other hand, has a **sparse** index: the index values are likely nowhere near one another.

The **TCP Connections table** `tcpConnTable` lists every TCP connection together with its connection

state as in *12.7 TCP state diagram*; one can obtain this on most linux, windows or macintosh systems with the command `netstat -a`. The index in this case consists of the *four* attributes of the connection-defining socketpair: the local address, the local port, the remote address and the remote port. In this table, the only attribute not part of the index is an integer representing the connection state (again represented by an SNMP enumerated type).

localAddr	*localPort*	*remoteAddr*	*remotePort*	state
10.0.0.3	31895	147.126.1.209	993	established(5)
10.0.0.3	40113	74.125.225.98	80	timeWait(11)
10.0.0.3	20459	10.38.2.42	22	established(5)

SNMP has adopted conventions for how tabular data such as the above is to be encoded in the OID tree. The first step in this strategy is to define, statically, an OID prefix for the subtree representing the table; in this section we will denote this by **T** (later, in *21.10.2 Table definitions and the interfaces Group*, we will see that **T** often represents two OID levels of the form Table.Entry, or T.E). Each attribute of the table (that is, each column) is then assigned a non-leaf OID by appending successive integers to the table prefix, starting at 1. For example, the root prefix for the interfaces table is **T** = 1.3.6.1.2.1.2.2.1, known as `ifEntry`. The columns shown in the fraction of the interfaces table above have the following OIDs:

OID	table column
`{ ifEntry 1 }` or **T**.1	The interface number, `ifIndex`
`{ ifEntry 2 }` or **T**.2	The interface name or description
`{ ifEntry 4 }` or **T**.4	The interface MTU
`{ ifEntry 5 }` or **T**.5	The interface bitrate
`{ ifEntry 10 }` or **T**.10	The number of inbound octets (bytes)
`{ ifEntry 11 }` or **T**.11	The number of inbound unicast packets

The second step is to establish a convention for converting the index attributes to a list of integers that can be used as an **OID suffix** identifying a particular row. This index OID suffix is then appended to the attribute OID prefix (identifying the column) to obtain the full OID (with no trailing .0 this time) that represents the name of the table data value.

For the case of the interfaces table, the interface number is used as a single-level OID suffix. The `eth1` value for `inOctets` thus has OID name **T**.10.3: 10 is the number assigned to the `inOctets` column and 3 is the `ifIndex` value for the `eth1` row. Written in full, this is 1.3.6.1.2.1.2.2.1.**10.3**. Note this numbering has the form **T**.col.row, the transposition of the more common programming-language row-first convention T[row][col].

For the routing and TCP tables presented here, IPv4 addresses are written in dotted-decimal notation, *eg* 10.38.0.0, and then converted to four levels of OID suffix, *eg* .10.38.0.0. (Note that 10.38.0.0 has the same *textual* representation as an IPv4 address and as a four-level OID suffix (though a leading '.' has been added here to the latter), but *logically* they are entirely different things).

This OID-suffix encoding may seem obvious, but this is deceptive, and in *21.13.12 TCP-MIB* we will see a newer convention in which the IPv4 address 10.38.0.0 would be encoded as the OID suffix .**1**.**4**.10.38.0.0, where the **1** denotes an IPv4 address (2 for IPv6) and the **4** denotes the address length as an OCTET STRING. The present rule, for the .10.38.0.0 four-level encoding, comes from **RFC 2578**, §7.7; the `IpAddress` type (for IPv4 only) is defined as `IMPLICIT OCTET STRING (SIZE (4))` (in **RFC 1155**), of predetermined length. An IP address *might* have been encoded as a single OID level using the address as a single unsigned 32-bit integer, but this option was not chosen.

In the routing table above, the nextHop column is assigned the number 7, so the nextHop for 10.38.0.0 thus has the OID **T**.7.10.38.0.0.

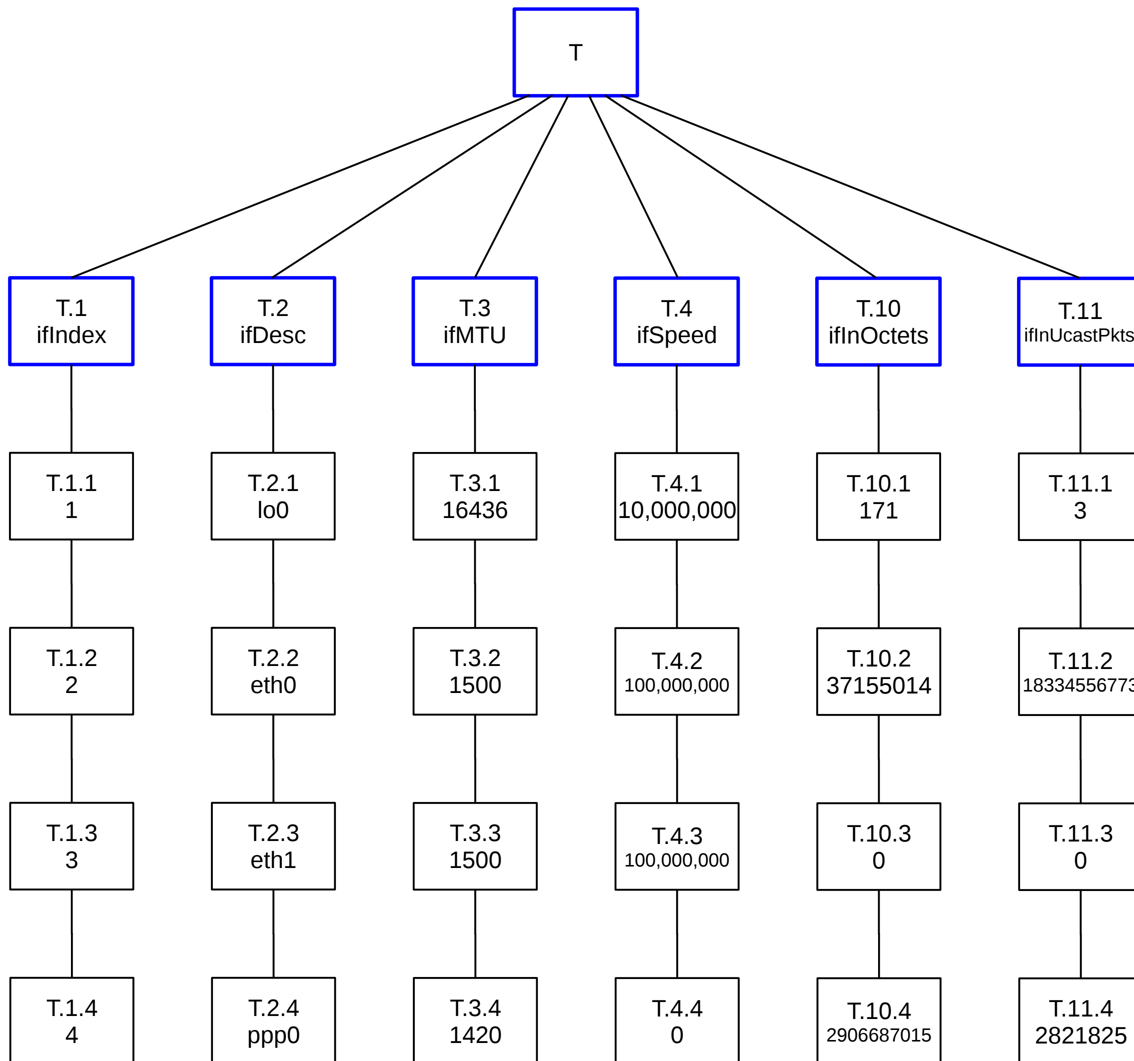
The OID suffix encoding used for indexing has nothing to do with the ASN.1 BER encoding used for data values, *21.12 SNMP and ASN.1 Encoding*.

For the TCP connections table, the two IP addresses involved each translate to four-level OID chunks, as in the routing table, and the two port numbers each translate to one-level chunks; the resultant OID suffix for the state of the first connection in the table above would be

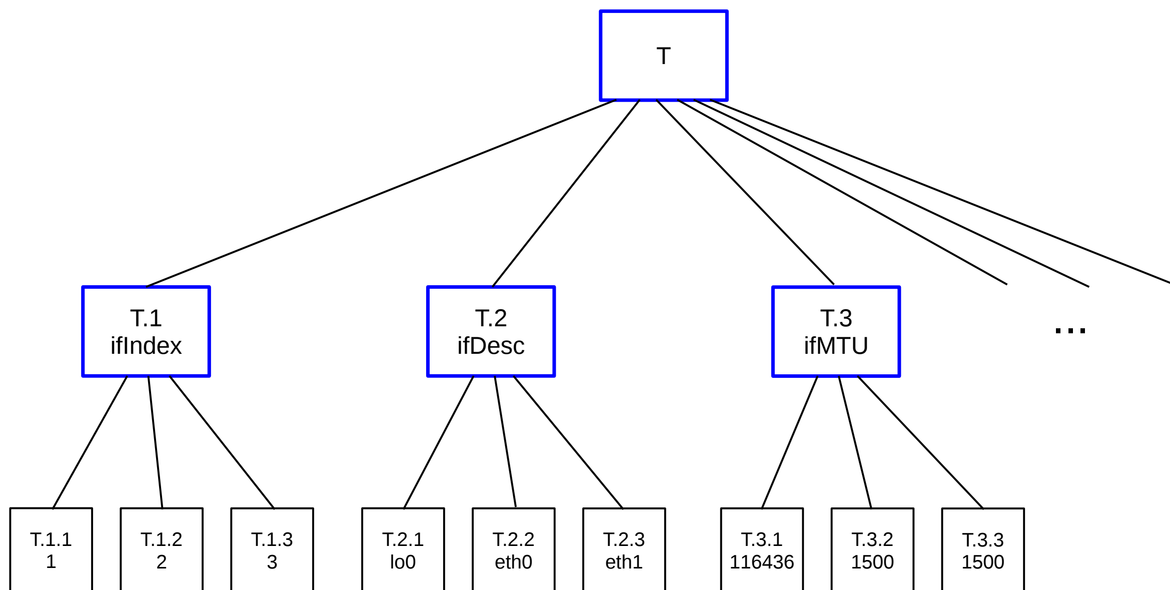
> .10.0.0.3.*31895*.147.126.1.209.*993*

where, for clarity, the port-number levels are shown in italic. The state column is assigned the identifier 1, so this would all be appended to **T**.1 (it is common, but, as we see here, not universal, for column-number assignment to begin with the index columns).

Here is a column-oriented diagram of the interfaces-table fragment from above. As we have been doing, the table prefix is denoted **T** and nodes are labeled **T**.col.row. The topmost T and the first row (with the **T**.col OIDs) are internal nodes; these are drawn with the heavier, blue boxes. The lower four rows, with black boxes, are leaf nodes.



The diagram above emphasizes the arrangement into columns. The actual OID tree structure, however, is as in the diagram below; all the leaf nodes in one column above are actually sibling nodes.



In all three tables here, the index columns are part of the table data. This is unnecessary; all the index columns are of necessity encoded in the full OID of any table entry. In SNMPv2 it is required (though not necessarily enforced) to in effect omit the index columns from the table as presented by the agent (though they are still declared); see *21.13.4 SNMPv2 Indexes*.

We also note that, in all three tables here (and in most SNMP tables that serve purely as sources of information), new rows cannot be added via the SNMP manager. Some SNMP tables, however, do support row creation; see *21.14 Table Row Creation*.

21.8 SNMP Operations

As mentioned earlier, the fundamental read operation is `Get()`; the protocol message itself is known as `GetRequest`, and also contains the authentication information (that is, the community string for SNMPv1 and SNMPv2c). A request-id is included so multiple outstanding `GetRequests` aren't mixed up.

The `GetRequest` message can contain a *list* of OIDs (actually encoded as a list of ⟨OID,null⟩ pairs called a `VarBind` list). A single-OID get simply entails a `VarBind` list of length 1.

The agent receiving the `GetRequest` validates the authentication and then, if successful, looks up each OID in the received list to find the corresponding value. The filled-in `VarBind` list of all ⟨OID,value⟩ pairs is returned.

If any value cannot be found, *eg* because the OID represented an interior node of the OID tree instead of an actual value, or because the OID lay outside the portion of the OID tree that the request was authorized to access, then in SNMPv1 *no* values are returned. Only if every OID in the list corresponds to a valid value is a list of all the ⟨OID,value⟩ pairs returned. SNMPv2 later relaxed this rule to allow the return of whatever individual requests were successful.

If the exact OID is not known – which is often the case for tabular data – the `GetNext()` operation can

be used. It works like `Get()`, except that for each OID *oid* in the request list the agent finds the **next leaf OID** strictly following *oid* in lexicographical order, which we will call *oid′*. The agent then includes the pair ⟨*oid′*,*value*⟩ in its response, where *value* is the value corresponding to *oid′*. Note that in the `GetNext()` case *oid′* is not previously known to the manager. Note also that SNMP *always* returns, with each value, the corresponding OID, which for tabular data encodes the full index for the value.

The `GetNext()` operation allows a manager to **walk** through any subtree of the OID tree. If the root of the subtree is prefix **T**, then the first call is to GetNext(**T**). This returns $\langle \text{oid}_1,\text{value}_1 \rangle$. The next call to GetNext has parameter oid_1; the agent returns $\langle \text{oid}_2,\text{value}_2 \rangle$. The manager continues with the series of GetNext calls until, finally, the subtree is exhausted and the agent returns either an error or else (more likely) an $\langle \text{oid}_N,\text{value}_N \rangle$ for which oid_N is no longer an extension of the original prefix p.

As an example, let us start with the prefix 1.3.6.1.2.1.1, the start of the `system` group. The first call to `GetNext()` will return the pair

> ⟨1.3.6.1.2.1.1.**1.0**, sysDescr_value⟩

The OID 1.3.6.1.2.1.1.**1.0** is the first leaf node below the interior node 1.3.6.1.2.1.1.

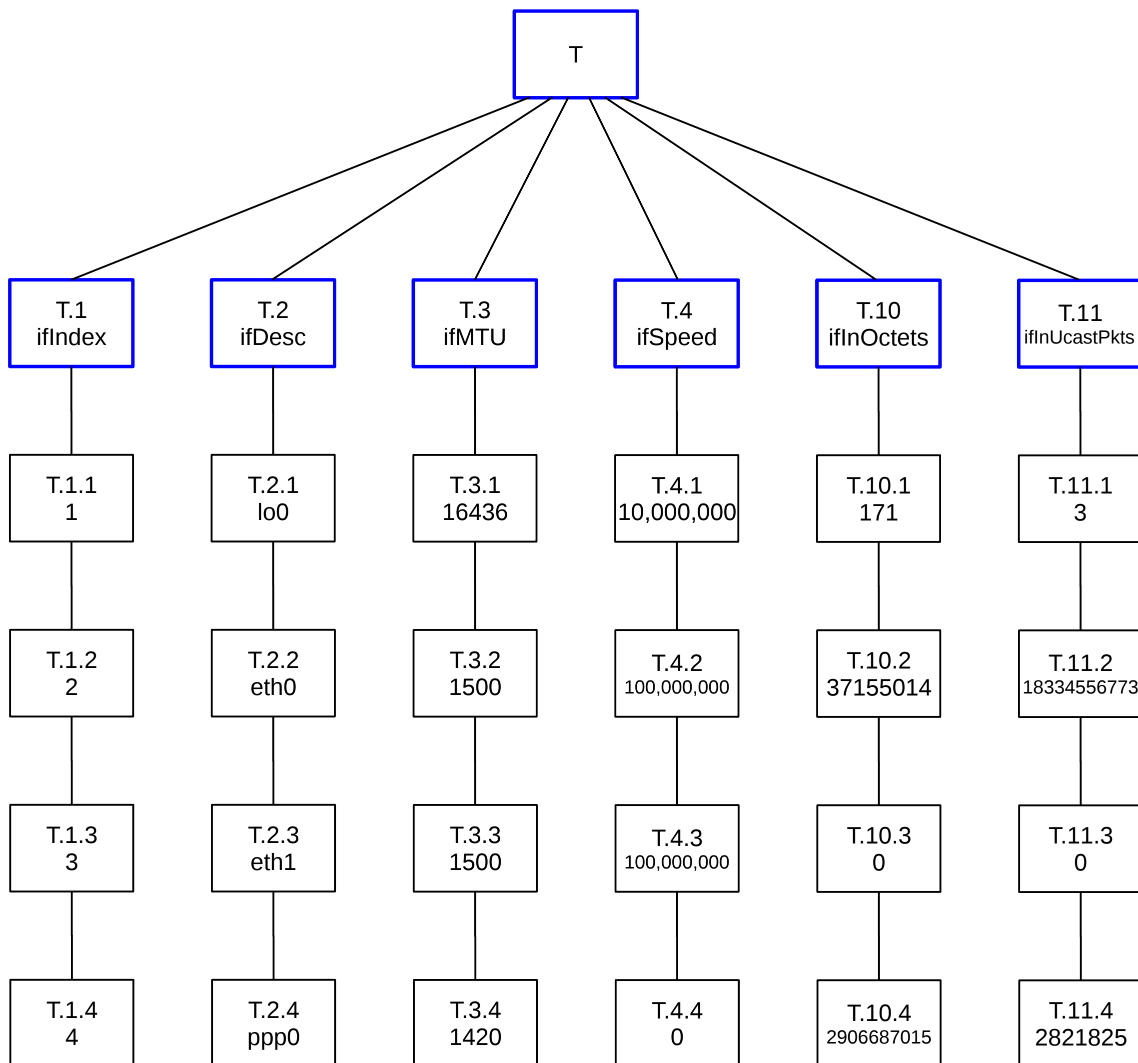
The second call to GetNext will take OID 1.3.6.1.2.1.1.**1.0** as parameter and `GetNext()` will return

> ⟨1.3.6.1.2.1.1.**2.0**, sysObjectID_value⟩

Here, the OID returned is the next leaf node strictly following 1.3.6.1.2.1.1.**1.0**. The process will continue on through {system 3 0}, {system 4 0}, *etc*, until an OID is found that is *not* part of the system group. As we have seen, this will likely be the first entry of the interfaces group, `ifNumber.0`, with OID 1.3.6.1.2.1.**2**.1.0.

The action of `GetNext()` is particularly useful when retrieving a table, such as the interfaces table presented in the preceding section. In that example the index values are consecutive integers, and so an SNMP manager could likely guess them. However, guessing the index values for the other two tables – the IP forwarding table and the TCP connections table – would be well nigh impossible. And without the index values, we do not have a complete OID.

Consider again the partial-column version of the interfaces table as diagrammed in this format:



Let us initially call GetNext(**T**). The next leaf node is the upper left black box, with OID **T**.1.1. The call to GetNext(**T**) returns the pair ⟨**T**.1.1,1⟩. We now continue with a call to GetNext(**T**.1.1); this returns ⟨**T**.1.2,2⟩ and represents the black box immediately below the first one. The next two calls to `GetNext()` return, successively, ⟨**T**.1.3,3⟩ and ⟨**T**.1.4,4⟩.

Now we call GetNext(**T**.1.4). The next leaf node following is the first leaf node in the *second* column, **T**.2.1; the value returned is ⟨**T**.2.1,"lo"⟩. The next three calls to GetNext, each time supplied with parameter the value returned by the previous `GetNext()`, return the next three values making up the second, `ifDesc` column of the table.

At the bottom of the second column, the call to GetNext(**T**.2.4) returns the ⟨OID,value⟩ pair that is the first leaf node of the third column, ⟨**T**.3.1,16436⟩. At the bottom of the third column, `GetNext()` jumps to the top of the fourth column, and so on. In this manner `GetNext()` iterates through the entire table, one column at a time.

When we finally get to the last leaf node of the table, shown here as the lower-right **T**.11.4 (though the actual `ifTable` has additional columns). The call to GetNext(**T**.11.4) returns something outside the table. It will

either return an error – in the event that there is no next OID that the manager is authorized to receive – or the next leaf node up and to the right of T. (In the normal MIB-2 collection, this is would be the first entry of the `atTable` table.)

Either way, the manager receiving the data can tell that its request for GetNext(**T**.11.4) has gone past the end of table **T**, and so **T** has been completely traversed.

Here is a diagram of the above sequence of `GetNext()` operations involved in traversing our partial interfaces table; it shows the GetNext(**T**.11.4) at the table's lower right returning the beyond-the-table ⟨OID,value⟩ pair ⟨**A**,avalue⟩:

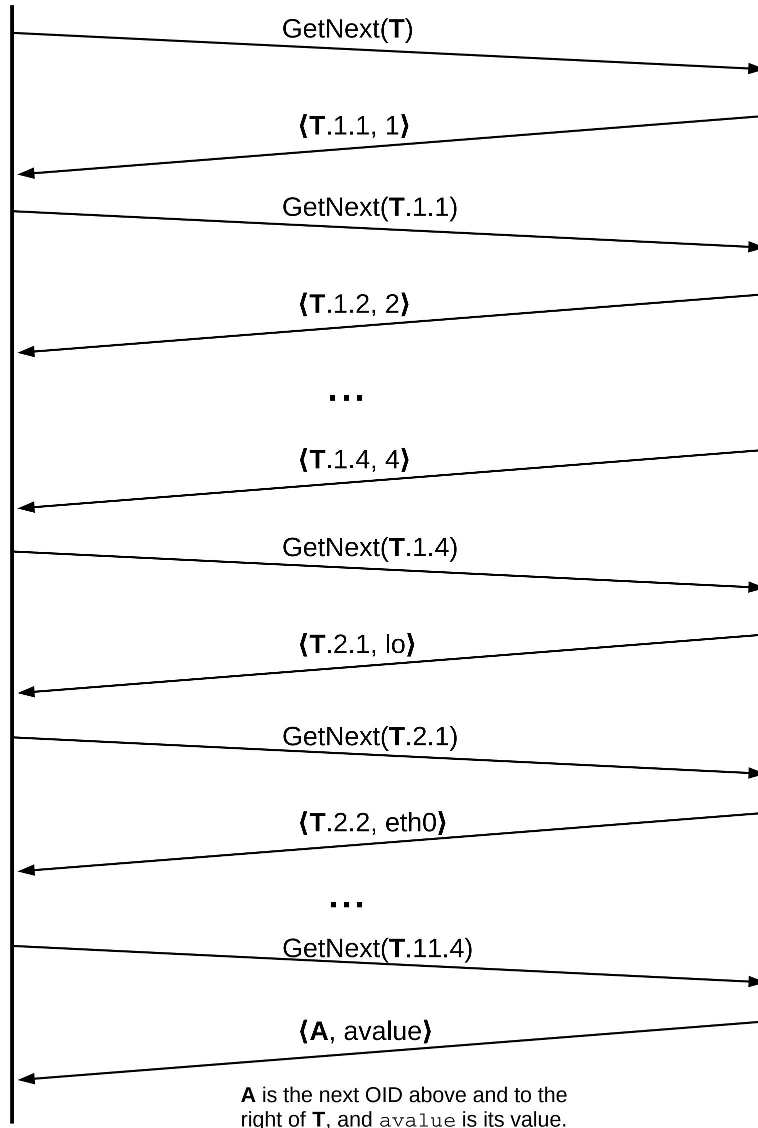


Diagram of traversing the (partial) interfaces table with `GetNext()`

Some manager utilities performing this kind of table retrieval – often called a "walk" – will present the data in the order retrieved, one column at a time, and some will (perhaps as an option) format the data visually

to look more like a table. See Net-SNMP's `snmpwalk` and `snmptable` in *21.9 MIB Browsing*.

21.8.1 Multi-attribute `Get()`

A single `Get()/GetNext()` request can in fact include a list of attributes to be retrieved. This provides an efficient way to request entire rows of a table.

A manager in general does not know all the index values of a table, but likely *does* know all the column OIDs (as in the blue full-length row above). The manager can speed up the table-retrieval process by asking for entire rows at a time. The first such `GetNext()` might be

GetNext(**T**.1, **T**.2, **T**.3, **T**.4, **T**.10, **T**.11)

These are the OIDs of the blue full-length row; these are all non-leaf OIDs. The result is the list of pairs

⟨**T**.1.1,1⟩, ⟨**T**.2.1,"lo"⟩, ⟨**T**.3.1,16436⟩, ⟨**T**.4.1,10000000⟩, ⟨**T**.10.1,171⟩, ⟨**T**.11.1,3⟩

That is, the table's entire first row is returned (where we are assuming again that the only columns are 1, 2, 3, 4, 10 and 11). The next call to `GetNext()` would use the OIDs returned above:

GetNext(**T**.1.1, **T**.2.1, **T**.3.1, **T**.4.1, **T**.10.1, **T**.11.1)

This would then return the table's entire second row, and so on. When we got to the very last row, the call

GetNext(**T**.1.4, **T**.2.4, **T**.3.4, **T**.4.4, **T**.10.4, **T**.11.4)

would return (assuming now that the columns shown are the only columns that exist)

⟨**T**.2.1,"lo"⟩, ⟨**T**.3.1,16436⟩, ⟨**T**.4.1,10000000⟩, ⟨**T**.10.1,171⟩, ⟨**T**.11.1,3⟩, ⟨**A**,avalue⟩

where **A** is the next leaf OID above and to the right of **T** (assuming no error). At this point the manager knows, as in the single-attribute-get case, that the entire table has been retrieved.

It does not matter whether the manager uses this multi-attribute form of `GetNext()` to return all the columns of the relevant table, or only the subset of columns in which the manager is interested.

We will see in *21.13.3 SNMPv2 GetBulk()* that SNMPv2 introduced an operation `GetBulk()` that can return multiple rows of multiple columns in a single operation.

Tables are not necessarily static. New dynamic interfaces may be added, and new routes and TCP connections are added all the time. A manager reading a table using the single-attribute form of `GetNext()` may find that only the latter part of a new row is retrieved, or, alternatively, that the first part of a deleted row is retrieved. The multi-attribute `GetNext()` offers a little more protection from this.

21.8.2 `Set()`

SNMP also allows a `Set()` operation. Not all attributes are writable, of course, and a manager must have write authority for those that *are* writable. And there is no `SetNext()`; the exact OID of each value must be supplied.

In the system group, the `sysName` attribute is writable. It has OID 1.3.6.1.2.1.1.5.0; to update this we would invoke

Set(1.3.6.1.2.1.1.5.0, "newsysname")

Like `Get()`, `Set()` can be invoked on multiple attributes. Suppose table **T** has three columns **T**.1, **T**.2 and **T**.3, and two rows:

T.1	**T**.2	**T**.3
10	eth0	3.14159
11	eth1	2.71828

We can then change the second `eth1` row to ⟨11,ppp1,0.577216⟩ with

> Set((**T**.2.11,ppp1), (**T**.3.11,0.577216))

We cannot change the indexing this way, but see *21.14 Table Row Creation*.

The semantics for multi-attribute `Set()` explicitly require – and still require – that either all the assignments succeed, or all fail; that is, multi-attribute `Set()` operations are **atomic**. (The usual reason for a failure is that the manager issuing the `Set()` lacked write permission for some item.) SNMPv1 had the same all-or-nothing rule for multi-attribute `Get()`, but that requirement was relaxed in SNMPv2. The all-or-nothing `Set()` semantics mean that certain updates cannot end up completed only half-way.

The multi-attribute `Set()` semantics do *not* mean, however, that near-simultaneous assignments by different managers are **serialized**. Suppose, for example, two separate managers both attempt to update attributes specified by OIDs **X** and **Y**, one to 10 and 100 and the other to 20 and 200. If the updates are scheduled at about the same time,

> Set((**X**,10), (**Y**,100))
> Set((**X**,20), (**Y**,200))

then it is possible for the Set()s to be performed in the order (**X**,20), (**X**,10), (**Y**,100), (**Y**,200), leaving **X** = 10 and **Y** = 200. See *21.13.5 TestAndIncr* for a workaround.

21.9 MIB Browsing

Tools for individual SNMP reading are widely available, and are very helpful for viewing what is going on.

We have already mentioned the Net-SNMP package. The command `snmpget` issues a single (perhaps multi-attribute) `Get()` request; the very similar `snmpgetnext` issues a GetNext request. The command `snmpwalk` takes an OID representing the root of a subtree, and returns everything in that subtree. Here is an example that assumes the snmp agent on localhost is configured to use community string "tengwar" (see *21.11 SNMPv1 communities and security*):

```
snmpwalk -v 1 -c tengwar localhost 1.3.6.1.2.1.2.2
```

The OID here is that of `ifTable`. Multiple OIDs are permitted for `snmpget` and `snmpgetnext` but not for `snmpwalk`.

If the appropriate MIB files are loaded, the above command can also be entered as

```
snmpwalk -v 1 -c tengwar localhost ifTable
```

This entails putting the RFC1213-MIB file into a special directory (*eg* $HOME/.snmp/mibs), and then either adding a line like

```
mibs +RFC1213-MIB
```

to the snmp.conf file, or by including the MIB name on the command line:

```
snmpwalk -v 1 -c tengwar -m RFC1213-MIB localhost ifTable
```

Note that `RFC1213-MIB` is the identifier assigned to the MIB file in its first line, as below, and *not* necessarily the name of the file containing the MIB.

```
RFC1213-MIB DEFINITIONS ::= BEGIN
```

If the appropriate MIB file is loaded, the OIDs may be displayed symbolically rather than numerically, but the data presentation will not change:

```
   iso.3.6.1.2.1.1.5.0 = STRING: "valhal"       ;; without MIB
RFC1213-MIB::sysName.0 = STRING: "valhal"       ;; with MIB
```

(Actually, if the data is of type Object ID, then without the MIB the OID will be displayed numerically, and with the appropriate MIB all or part of it may be displayed symbolically.)

Finally, the Net-SNMP manager package includes `snmptable`, which is like `snmpwalk` except that the data is displayed as a table rather than one column at a time. For the `snmptable` command, the appropriate MIB file *must* be installed. Net-SNMP comes with a mib browser with a graphical interface, `tkmib`.

The personal edition of the iReasoning MIB browser is not open-source, but it *is* free, and works well on windows, macs and linux; SNMPv3 is not supported. The license does prohibit the publication of benchmark tests without consent.

21.10 MIB-2

We can now turn to the MIB that represents the core of SNMPv1 data, known as MIB-2, and defined in **RFC 1213**. The "2" here – often represented with a Roman numeral: MIB-II – represents the second iteration of the definition, but it is still part of SNMPv**1**. The predecessor MIB-1 was first documented in **RFC 1066**, 1988.

As we saw in *21.4 MIBs*, the MIB-2 OID prefix is 1.3.6.1.2.1.

MIB-2 has since been extended. We look at a few extensions below in *21.13.7 SNMPv2 MIB Changes*, *21.13.9 IF-MIB and ifXTable*, *21.13.12 TCP-MIB* and *21.13.11.2 IP-Forward MIB*. In general, serious network management should make use of these newer versions. However, MIB-2 is still an excellent place to get started, even if partly obsolete.

The original MIB-2 is divided into ten groups, not all of which are in current use:

- `system(1)`: above
- `interfaces(2)`: above, in brief
- `at(3)`: the ARP cache, *7.9 Address Resolution Protocol: ARP*
- `ip(4)`: including the IP forwarding table used as an example above
- `icmp(5)`: information about ICMP, *7.11 Internet Control Message Protocol*

- `tcp(6)`: including the TCP connections table used as an example above
- `udp(7)`: information about UDP activity
- `egp(8)`: information about the now-obsolete EGP protocol, replaced by BGP
- `transmission(10)`: never used
- `snmp(11)`: about the SNMP agent itself

Originally group 9 was for CMOT, above, but this entry is commented out in **RFC 1213** (though it does not appear at all in MIB-1).

The `at` group is officially deprecated; the same ARP-cache information is available within the `ip` group. The EGP protocol has been entirely replaced by BGP, and management of BGP routers is highly specialized; **RFC 4273** contains a BGP MIB but private MIBs are almost universally used here as well.

The transmission group was included anticipating the day when "definitions for managed transmission media are defined". This day has not come to pass.

The Net-SNMP agent implementation returns nothing for the egp and transmission groups.

21.10.1 The system Group

We have already looked at this in *21.4 MIBs* and will not say much more, except to note that several of the entries in this group must be manually configured. This can be done either through editing of the configuration file, *eg* `snmpd.conf`, or else (in the case of `read-write` attributes) by using SNMP itself.

The `sysObjectID` value represents a vendor-specific OID for this particular system's SNMP agent. For example, the Net-SNMP package installed on my system returns 1.3.6.1.4.1.8072.3.2.10. The 1.3.6.1.4.1 is the root of the `private`, vendor-specific, OID tree, and 8072 has been assigned to the Net-SNMP project. This often represents, in practice, the way an SNMP manager can figure out the vendor of an agent, and thus determine what vendor-specific information to query. One can use the 8072 discovered here to ask for the subtree 1.3.6.1.4.1.8072, and find all sorts of Net-SNMP-specific information. Similarly, on one particular Windows XP installation with Microsoft Network Monitoring installed, the sysObjectID value is 1.3.6.1.4.1.**311**.1.1.3.1.1; 311 is the OID level assigned to Microsoft in the private subtree 1.3.6.1.4.1.

The system group has been expanded in SNMPv2 with an Object Resource table, `sysORTable`; see *21.13.7 SNMPv2 MIB Changes*.

21.10.2 Table definitions and the interfaces Group

The interfaces group consists of a single INTEGER value `ifNumber` representing the number of current network interfaces (including "down" interfaces), and the interfaces table partially introduced earlier. We take the opportunity here to outline exactly how MIB table definitions – and enumerated types – are structured using ASN.1.

Some of the interfaces-group definitions have later been updated. See, for example, **RFC 2863**, which also redefines the group to use the additional features of the SNMPv2 SMI. We will return to an extension of the interfaces group in *21.13.9 IF-MIB and ifXTable*.

MIB table definitions almost always involve a two-level process: an OID is defined for the table, and then a second OID is defined for a **table entry**, that is, for one row of the table. This second OID is usually generated from the first by appending ".1", and it is this second OID that represents the table prefix in the sense of *21.7 SNMP Tables*, denoted there by **T**.

The actual ASN.1, slightly annotated, is as follows:

```
ifTable OBJECT-TYPE
    SYNTAX  SEQUENCE OF IfEntry       -- note UPPER-CASE-I IfEntry
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
            "A list of interface entries.  The number of
            entries is given by the value of ifNumber."
    ::= { interfaces 2 }              -- that is, 1.3.6.1.2.1.2.2

ifEntry OBJECT-TYPE                   -- note lower-case-i ifEntry
    SYNTAX  IfEntry                   -- note UPPER-CASE-I IfEntry
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
            "An interface entry containing objects at the
            subnetwork layer and below for a particular interface."
    INDEX   { ifIndex }
    ::= { ifTable 1 }                 -- that is, 1.3.6.1.2.1.2.2.1
```

Both these entries are `not-accessible` as they do not represent leaf nodes. The second declaration above is for the lower-case-i `ifEntry`; the next definition in **RFC 1213** is for the UPPER-CASE-I version. The latter represents the complete list of all columns of an ifEntry/IfEntry object, together with their types from (in most cases) *21.5 SNMPv1 Data Types*. The `PhysAddress` type is defined in **RFC 1213** as a synonym for `OCTET STRING`. Definitions for the OID associated with each column comes later.

It is `ifEntry` that represents an actual row, and thus includes an `INDEX` entry to specify the attribute or attributes that make up the primary key for that row.

We now define **I**fEntry:

```
IfEntry ::=
    SEQUENCE {
        ifIndex            INTEGER,
        ifDescr            DisplayString,
        ifType             INTEGER,
        ifMtu              INTEGER,
        ifSpeed            Gauge,
        ifPhysAddress      PhysAddress,
        ifAdminStatus      INTEGER,
        ifOperStatus       INTEGER,
        ifLastChange       TimeTicks,
        ifInOctets         Counter,
        ifInUcastPkts      Counter,
        ifInNUcastPkts     Counter,
        ifInDiscards       Counter,
        ifInErrors         Counter,
        ifInUnknownProtos  Counter,
```

```
        ifOutOctets         Counter,
        ifOutUcastPkts      Counter,
        ifOutNUcastPkts     Counter,
        ifOutDiscards       Counter,
        ifOutErrors         Counter,
        ifOutQLen           Gauge,
        ifSpecific          OBJECT IDENTIFIER
    }
```

Attributes

There can be a certain eye-glazing tedium in some of SNMP's lengthy attribute lists, such as the one above. This can be replaced by panic in a hurry, though, when a problem has arisen and the proper attribute to diagnose it doesn't seem to be included anywhere.

It would have been possible to plug in the above definition of **I**fEntry into the `SYNTAX` specification of the previous **i**fEntry, but that is cumbersome. The **I**fEntry definition is not a stand-alone `OBJECT-TYPE` and does not have its own OID.

If all we wanted to do was to implement the interfaces table using the shortest possible OIDs, we would not have created separate ifTable and ifEntry OIDs. This would mean, however, that we could not use the `OBJECT-TYPE` macro to define `ifTable`, which would have been less consistent (especially as the ASN.1 syntax also determines the packet encoding, as in *21.12 SNMP and ASN.1 Encoding*). Essentially every table prefix in SNMP is defined using two additional OID levels, as here, rather than one.

We now turn to the 22 specific interface attributes. Here is the definition for the first, `ifIndex`; it defines column 1 of the interfaces table to be, in effect, ifEntry.1. As we are now talking about a leaf node, once the OID suffix is appended to represent the index, the `ACCESS` is no longer `not-accessible`.

```
ifIndex OBJECT-TYPE
    SYNTAX  INTEGER
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
            "A unique value for each interface.  Its value ranges
            between 1 and the value of ifNumber.  The value for each
            interface must remain constant at least from one
            reinitialization of the entity's network management system
            to the next reinitialization."
    ::= { ifEntry 1 }
```

The `DESCRIPTION` clearly indicates that the values for `ifIndex`, used to specify interfaces, are to be consecutive integers. Compliance with this rule has been an early casualty, for various reasons, and is formally withdrawn in **RFC 2863**. Some vendors simply number physical interfaces non-consecutively. In other cases, there is some underlying issue with consecutive numbering. For example, one of the author's systems running Net-SNMP returns `ifNumber` = 3, and then the following table values:

`ifIndex`	`ifName`
1	lo
2	eth0
549	ppp0

It turns out that ppp0 is a virtual interface corresponding to a VPN tunnel, *3.1 Virtual Private Networks*, and the underlying tunnel regularly fails and is then automatically re-instantiated. Each time it does so, the `ifIndex` is incremented by 1.

The rule that interfaces be numbered consecutively was formally deprecated in **RFC 2863**, an SNMPv2 update of the interface group. This, in turn, makes the `ifNumber` value rather less useful than it might be; most SNMP tables are not associated with a count attribute and seem to do just fine.

Most SNMP data values correspond straightforwardly with attributes defined by the hardware and the underlying operating system. The `ifIndex` value does not, at least not necessarily. Generally the agent must maintain at least some state to keep the `ifIndex` value consistent. In some cases, the `ifIndex` value may be taken from the relative position of the interface in some internal operating-system table. This is not, however, universally the case, as with the `ifIndex` value of 549 for ppp0 in the table above.

The `ifIndex` value is widely used throughout SNMP, and is often referenced in other tables. SNMPv2 even defines a special type (a TEXTUAL CONVENTION) for it, named `InterfaceIndex` (*21.13.1 SNMPv2 SMI and Data Types*).

As we mentioned earlier, there is no reason to include `ifIndex` as an actual column in the table; the value of `ifIndex` can always be calculated from the OID of any component of the row. The SNMPv2 approach here – basically to define `ifIndex` as `not-accessible` – is described below in *21.13.4 SNMPv2 Indexes*.

The `ifDescr` is a textual description of the interface; it is usually the device name associated with the interface. **RFC 1213** states "this string should include the name of the manufacturer, the product name and the version of the hardware interface", but this is inconsistent with linux device-naming conventions. The current rule is that it is merely unique.

The `ifType` attribute is our first example of how ASN.1 handles **enumerated types**. The value is a small integer and the hardware type associated with each integer is spelled out as follows. The majority of the networking technologies in this 1991 list have pretty much vanished from the face of the earth.

```
ifType OBJECT-TYPE
    SYNTAX  INTEGER {
                other(1),          -- none of the following
                regular1822(2),
                hdh1822(3),
                ddn-x25(4),
                rfc877-x25(5),
                ethernet-csmacd(6),
                iso88023-csmacd(7),
                iso88024-tokenBus(8),
                iso88025-tokenRing(9),
                iso88026-man(10),
                starLan(11),
                proteon-10Mbit(12),
                proteon-80Mbit(13),
                hyperchannel(14),
                fddi(15),
                lapb(16),
                sdlc(17),
                ds1(18),           -- T-1
                e1(19),            -- European equiv. of T-1
```

```
                basicISDN(20),
                primaryISDN(21),    -- proprietary serial
                propPointToPointSerial(22),
                ppp(23),
                softwareLoopback(24),
                eon(25),            -- CLNP over IP [11]
                ethernet-3Mbit(26),
                nsip(27),           -- XNS over IP
                slip(28),           -- generic SLIP
                ultra(29),          -- ULTRA technologies
                ds3(30),            -- T-3
                sip(31),            -- SMDS
                frame-relay(32)
            }
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
            "The type of interface, distinguished according to
            the physical/link protocol(s) immediately 'below'
            the network layer in the protocol stack."
    ::= { ifEntry 3 }
```

A more serious problem is that over two hundred new technologies are unlisted here. To address this, the values for `ifType` have been placed under control of the IANA, as defined in `IANAifType`; see **RFC 2863** and https://www.iana.org/assignments/ianaiftype-mib/ianaiftype-mib. The IANA can then add new types without formally updating any RFC.

For the meaning of `ifMtu`, the interface MTU, see *7.4 Fragmentation*. The 32-bit `ifSpeed` value will be unusable once speeds exceed 2 Gbps; **RFC 2863** defines an `ifHighSpeed` object with speed measured in units of Mbps. This entry is part of the `ifXTable` table, *21.13.9 IF-MIB and ifXTable*. **RFC 2863** also clarifies that, for virtual interfaces that do not really have a bandwidth, the value to be reported is zero (though in the example earlier the loopback interface `lo` was reported to have a bandwidth of 10 Mbps). The `ifPhysAddress` is, on Ethernets, the Ethernet address of the interface.

The `ifAdminStatus` and `ifOperStatus` attributes are enumerated types: up(1), down(2), testing(3). If the `ifAdminStatus` is up(1), and the `ifOperStatus` disagrees, then there is a likely hardware malfunction. The `ifLastChange` attribute reflects the last time, in TimeTicks, there was a change in `ifOperStatus`.

The next eleven entries count bytes, packets and errors. The `ifInOctets` and `ifOutOctets` count bytes received and sent (including framing bytes, *4.1.5 Framing*, if applicable). The problem with these is that they may wrap around too fast: in 34 seconds at 1 Gbps. The MIB-2 values are still used, but are generally supplemented with 64-bit counters defined in `ifXTable`, *21.13.9 IF-MIB and ifXTable*.

The packet counters are for unicast packets, non-unicast packets, discarded packets, errors, and, for inbound packets only, packets with unknown protocols. The count of non-unicast packets is separated in `ifXTable` into separate counts of broadcast and multicast packets.

The interface queue length is available in `ifOutQLen`. It takes a considerable amount of traffic to make this anything other than 0. **RFC 1213** says nothing about the timescale for averaging the queue length.

The last member of the classic MIB-2 interfaces group is `ifSpecific`, which has type ObjectID. It was to return an OID that may be queried for additional ifType-specific information about the interface. It was

formally deprecated in **RFC 2233**.

21.10.3 The ip Group

The ip group contains several scalar attributes and three tables.

The first two attributes are writable:

- `ipForwarding`: a Boolean attribute to enable or disable forwarding
- `ipDefaultTTL`: the default TTL in outgoing packets (*7.1 The IPv4 Header*)

The next scalar attributes are as follows. Here and later, we summarize these informally.

- `ipInReceives`: the number of received IP packets
- `ipInHdrErrors`: the number of apparent IP packets received with header errors
- `ipInAddrErrors`: the number of IP packets received with bad destination addresses
- `ipForwDatagrams`: the number of forwarded IP packets
- `ipInUnknownProtos`: the number of received IP packets with unknown higher-level protocol
- `ipInDiscards`: the number of received IP packets that had no errors, but which were still discarded due to resource limits
- `ipInDelivers`: the number of received IP packets delivered locally
- `ipOutRequests`: the number of IP packets originated by this node for delivery elsewhere
- `ipOutDiscards`: the number of outbound IP packets discarded due to resource limits
- `ipOutNoRoutes`: the number of outbound IP packets for which there was no route in the IP forwarding table
- `ipReasmTimeout`: the value, in seconds, of the IP fragment-reassembly timer
- `ipReasmReqds`: the number of IP proper fragments received that needed reassembly at this node (forwarded fragments are reassembled at their destination)
- `ipReasmOKs`: the number of fragmented IP packets successfully reassembled
- `ipReasmFails`: the number of fragmented IP packets *not* successfully reassembled
- `ipFragFails`: primarily, packets that needed fragmentation but their Dont Fragment bit (*7.4 Fragmentation*) was set
- `ipFragCreates`: the number of IP fragments created by this node
- `ipRoutingDiscards`: the number of packets that should have been forwarded, but were discarded due to resource limits

The first table is the `ipAddrTable`, a list of all IP addresses assigned to this node together with interfaces and netmasks.

The second table is the `ipRouteTable`, that is, the forwarding table. We looked briefly at this in *21.7 SNMP Tables*. Note that the index consists solely of the destination IP address `ipRouteDest`,

which is not sufficient for CIDR-based forwarding in which the forwarding-table key is the ⟨dest,netmask⟩ pair. Traffic to 10.38.0.0/16 might be routed differently than traffic to 10.38.0.0/24!

This table contains four attributes `ipRouteMetric1` through `ipRouteMetric4`. **RFC 1213** states that unused metrics should be set to -1, but many agents simply omit such columns entirely. Agents may similarly omit `ipRouteAge`.

The third table is `ipNetToMediaTable`, which is the ARP-cache table and which replaces a similar now-deprecated table in the at group. The `ipNetToMedia` table adds one column, `ipNetToMediaType`, not present in the at-group table; this column indicates whether an physical-to-IP-address mapping is invalid(2), dynamic(3) or static(4). Most ARP entries are dynamic.

For an updated version of the ip group, see *21.13.11.1 IP-MIB* and *21.13.11.2 IP-Forward MIB*.

21.10.4 The icmp Group

The icmp group consists of 26 counters for various ICMP events. ICMP Echo Request (ping request) packets have their own counter.

21.10.5 The tcp Group

The tcp group includes the following scalars:

- `tcpRtoAlgorithm`: the method for calculating the retransmission timeout, normally vanj(4) for the Jacobson-Karels algorithm in *12.19 TCP Timeout and Retransmission*.
- `tcpRtoMin`: the minimum TCP retransmission timeout, in milliseconds
- `tcpRtoMax`: the maximum TCP retransmission timeout, again in ms
- `tcpMaxConn`: if the system imposes a static cap on the number of allowed TCP connections, that is returned. Most systems have no static cap, though, and return -1.
- `tcpActiveOpens`: the number of TCP connections opened from this node; specifically, the number of TCP state transitions CLOSED → SYN_SENT
- `tcpPassiveOpens`: literally, the number of TCP state transitions LISTEN → SYN_RECD
- `tcpAttemptFails`: the number of TCP connections that went to CLOSED without ever being ESTABLISHED
- `tcpEstabResets`: the number of TCP connections that went from ESTABLISHED or CLOSE_WAIT directly to CLOSED, via RST packets
- `tcpCurrEstab`: the number of TCP connections currently in either state ESTABLISHED or state CLOSE_WAIT
- `tcpInSegs`: the count of received TCP packets, including errors and duplicates (though duplicate *reception* is relatively rare)
- `tcpOutSegs`: the count of sent TCP packets, *not* including retransmissions
- `tcpRetransSegs`: the count of TCP packets with at least one retransmitted byte; the packet may also contain new data

- tcpInErrs: the number of TCP packets received with errors, including checksum errors
- tcpOutRsts: the number of RST packets sent

Perhaps surprisingly, there is no counter provided for total number of TCP *bytes* sent or received. There is also no entry for the congestion-management strategy, *eg* Reno v NewReno v TCP Cubic (*15 Newer TCP Implementations*).

The tcp group also includes the tcpConnTable table, which lists, for each connection (identified by ⟨localAddress,localPort,remoteAddress,remotePort⟩) the connection state. We looked at this earlier in *21.7 SNMP Tables*. The table is noteworthy in that four of its five columns are part of the INDEX. A consequence is that to extract all the information from the table, a manager need only retrieve the tcpConnState column: the other four attributes will all be encoded in the OID index that is returned with each tcpConnState value.

We will look at the newer SNMPv2 replacement in *21.13.12 TCP-MIB*.

21.10.6 The udp Group

The udp group contains the following scalars:

- udpInDatagrams: a count of the number of UDP packets received and deliverable to a socket
- udpNoPorts: a count of UDP packets received but undeliverable because the port was not open
- udpInErrors: a count of the number of UDP packets undeliverable due to any other errors
- udpOutDatagrams: a count of the number of UDP packets sent

The udp group also contains the table udpTable. This table lists those UDP ports that the node has open. The table rows have the form ⟨localAddr,localPort⟩, with both columns included in the index. If a UDP socket accepts connections from anywhere, the localAddr will appear as 0.0.0.0.

21.10.7 The snmp Group

The SNMP group consists of 28 SNMP status and error attributes, numbered from 1 to 30 with 7 and 23 not used.

21.11 SNMPv1 communities and security

An SNMPv1 manager authenticates itself to an agent by providing a string known as a **community** string that is a combination of both userid and password. That is, the single string identifies the manager (or manager group) and at the same time authenticates the manager.

The community string identifies a manager as a member of a designated "community", in the conventional use of the word, of managers. Of course, at many sites there will be only one manager that interacts with any given agent.

Community strings are sent unencrypted, and so are vulnerable to sniffing. Even sniffing is not always necessary; far and away the most popular value – the default for many agents – is the string "public".

While the community string can be made obscure, and changed frequently, these are not enough; the only appropriate security practice today is to use SNMPv3 authentication (*21.15 SNMPv3*).

If a manager sends an incorrect community string, then in SNMPv1 and SNMPv2c there is no reply. SNMPv3 relaxed this rule somewhat, *21.15.3 SNMPv3 Engines*.

A single agent can support multiple community strings. Each community has an associated subset of the MIB (a *view*) that it allows. For example, an agent can be configured so that the community string "system" allows the manager access to the `system` group, the community string "tengwar" allows access to the entire MIB-2 group, and the community string "galadriel" allows access to the preceding plus the `private` group(s). Using Net-SNMP (the most common agent on linux and Macintosh machines) this would be achieved by the following entries in the `snmpd.conf` file:

```
rocommunity  system    default   1.3.6.1.2.1.1
rocommunity  tengwar   default   1.3.6.1.2.1
rocommunity  galadriel default   -V mib2+private

view   mib2+private included  1.3.6.1.2.1
view   mib2+private included  1.3.6.1.4.1
```

In order that the galadriel community could contain two disconnected OID subtrees, it was necessary to make use of the View-based Access Control Model (VACM). Community galadriel has access to the OID-tree view named "mib2+private"; this view is in turn defined in the last two lines above.

For our purposes here, VACM can be seen as an implementation mechanism for specifying what portion of the OID tree is accessible to a given community. VACM allows read and write permissions to be granted to specific OID trees, and also to be excluded from specific subtrees (*eg* table columns). Using the "mask" mechanism, access can even be granted to specific rows of a table. We return to VACM for SNMPv3 very briefly in *21.15.9 VACM for SNMPv3*.

On Microsoft operating systems, an SNMP agent is generally included but must be enabled, *eg* from "Windows components" or "Programs and Features". After that, the agent must still be configured. This is done by accessing "Services" (*eg* through Control Panel → Administrative Tools or by launching services.msc), selecting "SNMP service", and clicking on "Properties". This applet only allows setting the community name and read vs write permissions; specifying collections of visible OID subtrees (views) is not supported here (though it may be via SNMP itself).

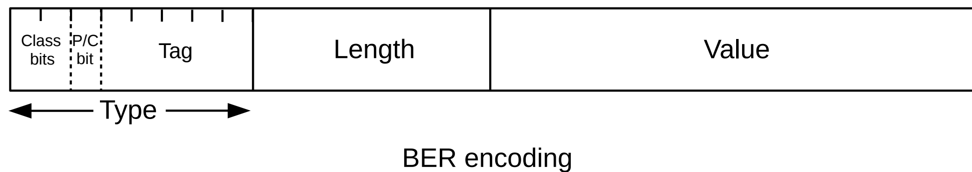
The community mechanism *can* offer a reasonable degree of security, if community names are changed frequently and if eavesdropping is not a concern. Perhaps the real problem with community-based security is that just how much information can leak out if an attacker knows the community string is not always well understood. SNMP access can reveal a site's complete network and host structure, including VPNs, subnets, TCP connections, host-to-host trust relationships, and much more. Most sites block the SNMP port 161 at their firewall; some even go so far as to run SNMP only on a "hidden" network largely invisible even within the organization.

The view model for OID-tree access is formalized in **RFC 3415** as part of SNMPv3; it is called the View-based Access Control Model or **VACM**. It allows the creation of named views. The `vacmAccessTable` spells out the viewing rights assigned to a given VACM **group**, which, in a rough sense, corresponds to an SNMPv1/SNMPv2c community. VACM supports, in addition to views consisting of disjoint unions of OID subtrees, table views that limit access to a specific set of columns.

21.12 SNMP and ASN.1 Encoding

When SNMP data is entered into a packet, it is encoded according to the ASN.1 Basic Encoding Rules, or **BER**. This is a hierarchical syntax-driven binary encoding strategy in which each atomic value (INTEGER, OCTET STRING, OBJECT IDENTIFIER, *etc*) is *tagged* with its type, and each compound structure is also tagged. This means that the receiver can understand a complex packet format *without* prior knowledge of its structure. The BER rules are part of the ITU standard X.690. (ASN.1 also supports several other encoding-rule formats, including the human-readable XML Encoding Rules, but SNMP uses BER.)

The general representation of an object is a "type-length-value" triple of the following form; the `type` tag makes the data self-identifying.



BER encoding

The first two bits of the `type` field, the `class` bits, identify the context. **Universal** types such as INTEGER and OBJECT IDENTIFIER have class bits 00; **application-specific** types such as Counter32 and TimeTicks have class bits 01.

The third bit is 0 for **p**rimitive types and 1 for **c**onstructed types such as STRUCTURE.

The rest of the first byte is the type tag. If a second byte is needed, the tag bits of the first byte are 11111.

21.12.1 Primitive Types

The standard ASN.1 universal-type primitive tag values used by SNMP are as follows:

00000	NULL
00010	INTEGER
00100	OCTET STRING
00110	OBJECT IDENTIFIER

SNMP also defines several application-specific tags (the following are from **RFC 2578**):

00000	IpAddress
00001	Counter32
00010	Gauge32
00011	TimeTicks
00100	Opaque
00110	Counter64

The second field of the type-length-value structure is the length of the `value` portion, in bytes. Most lengths of primitive types will fit into a single byte. If a data item is longer than 127 bytes (true for many composite types), the multi-byte integer encoding below is used.

The actual data is then encoded into the `value` field. For nonnegative integers, the integer is converted to a twos-complement bitstring and then encoded in as few bytes as possible, provided there is at least

one leading 0-bit to represent the sign. Similarly, negative numbers must have at least one leading 1-bit to represent the sign.

For example, 127 can be encoded as a length=1 INTEGER with `value` byte 0111 1111, while 128 must be encoded as a length=2 integer with `value` bytes 0000 0000 and 1000 0000. The second `value` byte represents 128, but the first byte represents the sign; without the first byte the INTEGER represented by a length of 1 and a `value` byte of 1000 0000 is –128. Similarly, to encode decimal 10,000,000 (0x989680), four `value` bytes are needed as otherwise the leading bit would be 1 and the number would be interpreted as negative.

For OCTET STRINGs, the string bytes are placed in the `value` portion and the number of bytes is placed in the `length` portion.

Object IDs are generally encoded using one byte per level; there are two exceptions. First, the initial two levels x.y of the OID are encoded using a single level with value $40 \times x + y$; all SNMP OIDs begin with 1.3 and so the first byte is 43 (0x2b). Second, if a level is greater than 127, it is encoded as multiple bytes. The first bit of the last byte is 0; the first bit of each of the preceding bytes is 1. The seven remaining bits of each byte contain the bits of the OID level. For example, 1.3.6.1.2.1.**128**.9 would have a `value` encoding of the following bytes in hexadecimal:

> 2b 06 01 02 01 **81 00** 09

where **128** is represented as the two seven-bit blocks 0000001 0000000 and the first block is prefixed by 1 and the second block by 0. Note that the encoding of 128 in an OID is quite different from the encoding of 128 as an INTEGER.

21.12.2 Composite Types

Now that we have the encodings of the primitive types we can build structures. For composite types the P/C bit will be set to 1. The only universal composite type we will consider is

10000	SEQUENCE and SEQUENCE OF

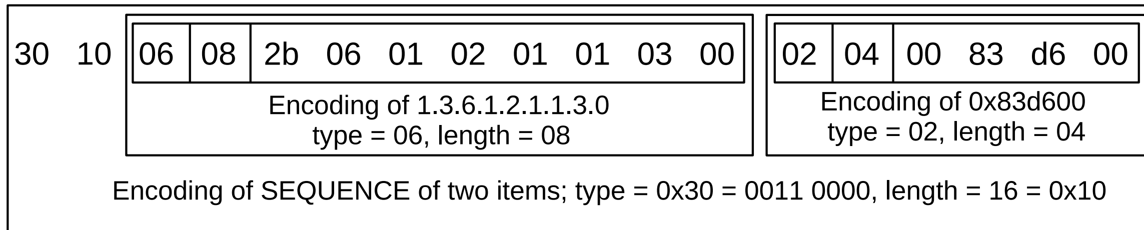
There are also several application-specific composite types; the first three bits of the tag field for these will be 101.

00000	Get-Request
00001	Get-Next-Request
00010	Get-Response
00011	Set-Request
00100	Trap

A `VarBind` pair is encoded in SNMPv1 as

```
SEQUENCE { name OBJECT IDENTIFIER, value ObjectSyntax }
```

`ObjectSyntax` is specified as a `CHOICE` that can contain any of the tagged SNMP primitive types above; the `CHOICE` syntax adds no bytes so the `value` is simply encoded as above. The `VarBind` pair ⟨1.3.6.1.2.1.1.3.0, 8650000⟩ – the OID is `sysUpTime` and the value is 24 hours – would then be represented in hexadecimal bytes as below; the hexadecimal representation of 8650000 is 0x83d600.



The first byte of 0x30 marks this as a SEQUENCE; recall that the `type` byte for a SEQUENCE has a P/C bit of 1 and five low-order bits of 10000, for a numeric value of 48 decimal or 0x30.

The `VarBindList` is defined to be a `SEQUENCE OF VarBind`. If we have only the single `VarBind` above, the resultant enclosing `VarBindList` would be as follows; the `length` field is 0x12 = 18.

30 12 30 10 06 08 2b 06 01 02 01 01 03 00 02 04 00 83 d6 00

The BER encoding rules do not stop with the `VarBindList`. A slightly simplified ASN.1 specification for an entire SNMPv1 Get-Request packet portion (or "protocol data unit") is

```
SEQUENCE {
       request-id   INTEGER,
       error-status INTEGER,
       error-index  INTEGER,
       variable-bindings VarBindList
}
```

The encoding of the whole packet would also be by the above BER rules.

The BER encoding mechanism represents a very different approach from the fixed-field layout of, say, the IP and TCP headers (*7.1 The IPv4 Header* and *12.2 TCP Header*). The latter approach is generally quite a bit more compact, as only four bytes are needed for a larger integer versus six under BER, and no bytes are used for SEQUENCE specifications. The biggest advantage of the SNMP BER approach, however, is that all objects, from entire packets down to individual values, are **self-describing**. Given the variety of types used by SNMP, the fact that many are of variable length, and the fact that value "readers" such as MIB browsers often operate without having all the type-specifying MIBs loaded, this self-describing feature is quite useful.

21.13 SNMPv2

SNMPv2 introduced multiple evolutionary changes: to the SMI, to various MIBs, and to the basic protocol operation. Many new MIBs were added. SNMPv2 also contained a proposal for improved security, but this was not widely adopted. Eventually most of the SNMP community settled on SNMPv2**c**, the version of SNMPv2 that stayed with the community-based security model.

Most of the specification of SNMPv2c is in **RFC 1901** through **RFC 1909**.

Generally SNMPv1 and SNMPv2c agents and managers can interoperate quite easily. Essentially all SNMPv2 agents also support SNMPv1 queries, and answer according to the version of the request received. There is slightly greater confusion between SNMPv1 and SNMPv2 MIB files, but almost all browsers and managers support both.

21.13.1 SNMPv2 SMI and Data Types

SNMPv2 introduced the `OBJECT-IDENTITY` macro, which acts like the `OBJECT-TYPE` macro except that it leaves out the SYNTAX clause; it serves as a way to define OIDs separately from syntax. In the `OBJECT-TYPE` macro, the `ACCESS` attribute is renamed `MAX-ACCESS`, and a new access option `read-create` (for reading plus row creation) is added.

SNMPv2 also provided new 64-bit versions of some of the basic types, and added some other types. These definitions are in **RFC 2578**, originally **RFC 1442**. A very practical problem with SNMPv1 is that, for example, the 32-bit `inOctets` counter can wrap around in 34 seconds at 1 Gbps.

The INTEGER type remains, now declared synonymous with Integer32. The SNMPv1 Counter and Gauge types have been replaced with Counter32 and Gauge32, and 64-bit versions of both – Counter64 and Gauge64 – were added (Gauge64 was added somewhat later, in **RFC 2856**).

A BITS type, for representing individual bits in a word, was also added.

The OBJECT IDENTIFIER type was formally limited to 128 levels.

SNMPv2 also introduced the `TEXTUAL-CONVENTION` macro, which is an alternative name for a type (or a primary name for an enumerated type) together with a description of what the type is actually to represent. Examples include `OwnerString`, which is an OCTET STRING intended to describe a management station and perhaps its human operator, `EntryStatus` which is an enumerated type meant to be used for row additions (*21.14.1 RMON*), and `InterfaceIndex`, which is meant to be the number of an interface appearing in the MIB-2 `ifTable`, *21.10.2 Table definitions and the interfaces Group*, but used in another table. For casual MIB reading, textual conventions can simply be thought of as types.

21.13.2 SNMPv2 Get Semantics

When an SNMPv1 agent receives a list of OIDs as part of a `Get()` request, and one or more of them is unavailable, then an error is returned. Under SNMPv2, the agent returns a list containing the appropriate value for each valid OID. For request OIDs for which the agent is not able to return an actual value, the special value `noSuchInstance` (for missing table entries with legal column specifications) or `noSuchObject` (for other missing values) is substituted.

The `Set()` semantics remain unchanged: an SNMPv2 agent does not update any of the OID values in the request unless it is able to update all of them.

21.13.3 SNMPv2 GetBulk()

SNMPv2 introduced the GetBulk operation as an extension of GetNext. A manager includes an integer N in its request and the agent then iterates the action of `GetNext()` N times. All N results (which can each represent an entire row) can then be returned in a single operation.

For example, suppose a table **T** has five rows indexed by 11-15, and three columns with OIDs **T**.1 through **T**.3. The OIDs for the table are as follows:

T.1.11	**T**.2.11	**T**.3.11
T.1.12	**T**.2.12	**T**.3.12
T.1.13	**T**.2.13	**T**.3.13
T.1.14	**T**.2.14	**T**.3.14
T.1.15	**T**.2.15	**T**.3.15

Then a GetBulk request for (**T**.1,**T**.2,**T**.3) with a repetition count of 3 will return the first three rows, with the following OIDs:

> **T**.1.11 **T**.2.11 **T**.3.11
>
> **T**.1.12 **T**.2.12 **T**.3.12
>
> **T**.1.13 **T**.2.13 **T**.3.13

To continue, the next such request would include OIDs (**T**.1.13, **T**.2.13, **T**.3.13) and the result (again assuming a count of 3) would be of values with these OIDs:

> **T**.1.14 **T**.2.14 **T**.3.14
>
> **T**.1.15 **T**.2.15 **T**.3.15
>
> **T**.2.11 **T**.3.11 **A**

where **A** is the next leaf OID above and to the right of **T**.

Note the third row of the second request: the first leaf OID following **T**.1.15 – the last row of column 1 – is **T**.2.11, that is, the first row of column 2. Similarly, **T**.3.11 follows **T**.2.15. As **T**.3.15 is the last leaf OID in the table, its leaf-OID successor (**A**) is outside the table.

The GetBulk request format actually partitions the list of requested OIDs into two parts: those OIDs that are to be requested only once and those for which the request is repeated. Two additional parameters besides the OID list are included: `non-repeaters` indicating the number of OIDs to be requested only once and `max-repetitions` indicating the number of times the remaining OIDs are retrieved.

As with GetNext, it is possible that a request for rows of the table will return ⟨OID,value⟩ pairs outside the table, that is, following the table in the OID-tree order.

If the total number of OIDs in the request is N ⩾ `non-repeaters`, then the return packet will contain a list of ⟨OID,value⟩ variable bindings of up to length

> `non-repeaters` + (N – `non-repeaters`)×`max-repetitions`

GetBulk has considerable potential to return more data than there is room for, so an agent may return fewer repetitions as it sees fit.

21.13.4 SNMPv2 Indexes

As we saw in *21.7 SNMP Tables*, in SNMPv1 the index columns were included in the table. Because every attribute returned by an agent comes paired with the attribute's OID, and because the OID of a table item encodes the row and thus the index value, index columns are unnecessary except in rare cases (one such case is when *all* table columns are part of the index, as in `udpTable`).

SNMPv2 deals with this by requiring the usual INDEX clause in the ASN.1 tableEntry definition, but then when the tableIndex attribute is defined with OBJECT-TYPE it is assigned a MAX-ACCESS of

not-accessible. Here is an example from the sysORTable of *21.13.7 SNMPv2 MIB Changes*; it is the second "not-accessible" that represents the change.

```
sysOREntry OBJECT-TYPE
    SYNTAX     SysOREntry
    MAX-ACCESS not-accessible
    STATUS     current
    DESCRIPTION
            "An entry (conceptual row) in the sysORTable."
    INDEX      { sysORIndex }
    ::= { sysORTable 1 }

 ...

sysORIndex OBJECT-TYPE
    SYNTAX     INTEGER (1..2147483647)
    MAX-ACCESS not-accessible
    STATUS     current
    DESCRIPTION ...
    ::= { sysOREntry 1 }
```

The `INDEX` attribute appears in the declaration of `sysOREntry` in the usual way. But it is now classed as an **auxiliary object**, and access to `sysORIndex` is `not-accessible`; in SNMPv1 it would have been `read-only` and thus an ordinary column.

When direct access to index attributes is suppressed this way, the index data is still available in the accompanying OID, but it is no longer tagged by type as in *21.12 SNMP and ASN.1 Encoding*. The `sysORIndex` value above, for example, would appear as a single OID level, but the manager would have to use the MIB to determine that it was meant as an INTEGER and not a TimeTicks or Counter. An IPv4 address used as an index would appear in the OID as four levels, readily recognizable as an IPv4 address *provided* the manager knew where to look. When STRING values appear in the index, the lack of an index column can be a particular nuisance; for example, the only indication of the username "alice" in the `usmUserTable` of *21.15.9.1 The usmUserTable* might be the OID fragment 97.108.105.99.101, representing the ASCII codes for the letters a.l.i.c.e.

Generally, an SNMPv2 agent will send back the `noSuchObject` special value (see *21.13.2 SNMPv2 Get Semantics*) when asked for a `not-accessible` auxiliary object.

21.13.5 `TestAndIncr`

SNMPv2 introduced the `TestAndIncr` textual convention, which introduces something of an aberration to the usual semantics of `Set()`. The underlying type is INTEGER, in the range $0..2^{31}–1$, and `Get()` works the usual way. However, if **TI** is the OID name of a `TestAndIncr` object, then Set(**TI**,*val*) never sets the value of **TI** to *val*. Instead, if the value of the object is *already* equal to *val*, then the `Set()` succeeds and the value of the object is incremented, that is, set to *val* + 1. If the value of the object is not equal to *val*, an error occurs and no change is made.

A `TestAndIncr` object acts like a kind of semaphore, though not exactly as there is no way to decrement the object (though the value does wrap around from 2^{31}-1 back to 0). The goal here is to provide a voluntary mechanism to enforce **serialization** when more than one manager may be writing to the same set of attributes.

As we saw at the example at the end of *21.8.2 Set()*, such serialization is not automatic. But now let us revisit that example using `TestAndIncr`. Recall that two managers are updating attributes with OIDs **X** and **Y**; this time, however, they agree also to include a `TestAndIncr` object with OID **TI**. Then serialization is assured as long as each manager executes each multi-attribute `Set()` in the following form, where *val* := Get(**TI**) means that the manager uses `Get()` to retrieve the value of **TI** and stores it in its own local variable *val*.

val := Get(**TI**)

Set((**TI**,*val*), (**X**,*xval*), (**Y**,*yval*))

To see this, suppose manager B's Get(**TI**) occurs after manager A's Set(**TI**,*val*) has incremented *val*. Then manager B's `Set()` operations occur even later, after A's `Set()` has completed. The alternative is that both managers `Get()` the same value *val*. Let A be the manager who first succeeds with Set(**TI**,*val*). Now the other manager – B – will have its Set(**TI**,*val*) fail, as the value stored at **TI** is now *val*+1. Thus *all* of B's `Set()` operations will fail.

A consequence here is that manager B will have to try again, probably immediately. However, the likelihood of conflict is low, and B can expect to succeed soon.

Usually only one `TestAndIncr` object needs to be provided for an entire table, not one per row. For an actual example, see *21.15.9.1 The usmUserTable*.

21.13.6 Table Augmentation

SNMPv2 introduced a mechanism for extending an existing table by adding, in effect, more columns, known as **augmentation**. In the new table-entry definition, instead of an `INDEX` clause there is an `AUGMENTS` clause; the argument to which is a table-entry name for the existing table to be extended. The new table-entry row will be indexed by whatever attributes indexed the table-entry in the AUGMENTS clause.

For example, the `ifXTable` augments the MIB-2 `ifTable`, meaning in essence that the `ifXTable` is automatically indexed by `ifTable`'s index, `ifIndex`. Here is the `ifXEntry` definition that establishes that:

```
ifXEntry         OBJECT-TYPE
   SYNTAX       IfXEntry
   MAX-ACCESS   not-accessible
   STATUS       current
   DESCRIPTION
           "An entry containing additional management information
           applicable to a particular interface."
   AUGMENTS    { ifEntry }
   ::= { ifXTable 1 }
```

The intent here is that every ifTable row is now extended to include the nineteen additional values defined for `IfXEntry`; that is, there is a one-to-one correspondence between rows if `ifTable` and `ifXTable`.

If, on the other hand, the new table extends only a few rows of the original table, *ie* is a **sparse** extension, then the new table entry should have an INDEX clause that repeats that of the original table. An example is the EtherLike-MIB of *21.13.10 ETHERLIKE-MIB*, in which the `dot3StatsTable` extends the MIB-2 `ifTable` by providing additional information for those interfaces that behave like Ethernets. The `dot3StatsEntry` definition is

```
dot3StatsEntry OBJECT-TYPE
    SYNTAX     Dot3StatsEntry
    MAX-ACCESS not-accessible
    STATUS     current
    DESCRIPTION "Statistics for a particular interface to an ethernet-like medium."
    INDEX       { dot3StatsIndex }
    ::= { dot3StatsTable 1 }
```

The INDEX is `dots3StatsIndex`, which is then defined as follows; note the statement in the DESCRIPTION (and the REFERENCE) that the dot3StatsIndex is to correspond to `ifIndex`.

```
dot3StatsIndex OBJECT-TYPE
    SYNTAX      InterfaceIndex
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION "An index value that uniquely identifies an interface
                to an ethernet-like medium.  The interface identified
                by a particular value of this index is the same interface
                as identified by the same value of ifIndex."
    REFERENCE   "RFC 2863, ifIndex"
    ::= { dot3StatsEntry 1 }
```

Finally, it is possible that a new table has a many-to-one, or **dense**, correspondence to the rows (or a subset of the rows) of an existing table. In this case, the new table will have an INDEX clause that will include the index attributes of the original table, and one or more additional attributes. An example is the EtherLike-MIB `dot3CollTable`, which keeps, for each interface, a set of histogram buckets giving, for each N, a count of the number of packets that experienced exactly N collisions before successful transmission. The `dot3CollEntry` definition is as follows:

```
dot3CollEntry OBJECT-TYPE
    SYNTAX      Dot3CollEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION ...
    INDEX       { ifIndex, dot3CollCount }
    ::= { dot3CollTable 1 }
```

The `ifIndex` entry in the INDEX represents the original table, as before except that here there is also a second, new INDEX attribute, `dot3CollCount`.

21.13.7 SNMPv2 MIB Changes

The core MIB for SNMPv2 – essentially the SNMPv2 version of MIB-2 – is **RFC 3418**, originally **RFC 1450**.

Some changes appear under the MIB-2 OID prefix: 1.3.6.1.2.1. SNMPv2 also defines a prefix 1.3.6.1.6 specifically for SNMPv2 information.

This MIB adds the Object Resource table, `sysORTable`, to the system group. It also slightly modifies the mib-2.snmp group and provides a new snmp group under the SNMPv2 1.3.6.1.6 prefix.

21.13.8 sysORTable

The original system group contained the attribute `sysObjectID` that identifies the agent and at the same time suggests a private OID tree that could provide additional information about the agent (*21.10.1 The system Group*).

The `sysORTable`, 1.3.6.1.2.1.1.9 or mib-2.system.9, is an attempt to extend this. It consists of a list of OIDs that can be queried for further agent information; each OID also has an associated description string and a `sysORUpTime` value indicating the time that OID was added.

For example, my system lists the following (where snmpModules = 1.3.6.1.6.3 and mib-2 = 1.3.6.1.2.1):

snmpModules.11.3.1.1	Message Processing MIB
snmpModules.15.2.1.1	User-based security MIB
snmpModules.10.3.1.1	SNMP management architecture
snmpModules.1	SNMPv2 information
mib-2.49	TCP
mib-2.4	IP
mib-2.50	UDP
snmpModules.16.2.2.1	VACM
snmpModules.13.3.1.3	SNMP notification
mib-2.92	SNMP notification logging

Each of the above OID prefixes can theoretically then be accessed for further information. Unfortunately, on my system several of them are not configured, and a query returns nothing, but `sysORTable` does not know that.

21.13.9 IF-MIB and ifXTable

RFC 2863 has addressed the problems with `ifSpeed` and the byte counters by introducing a new table, `ifXTable` (formerly `ifExtnsTable` with OID mib-2.31.1.2. As such, it is still under the aegis of the MIB-2 OID, but outside the traditional interfaces group. It does use the SNMPv2 SMI.

The `ifXTable` table includes definitions for "high capacity" 64-bit counters for in and out octets, in and out unicast packets, in and out multicast packets and in and out broadcast packets. It also includes the `ifHighSpeed` attribute for interface speed, still a 32-bit quantity but now measured in units of 1 Mbps.

The full definition of an `ifXTable` row is as follows:

```
IfXEntry ::=
   SEQUENCE {
       ifName                   DisplayString,
       ifInMulticastPkts        Counter32,
       ifInBroadcastPkts        Counter32,
       ifOutMulticastPkts       Counter32,
       ifOutBroadcastPkts       Counter32,
       ifHCInOctets             Counter64,
       ifHCInUcastPkts          Counter64,
       ifHCInMulticastPkts      Counter64,
       ifHCInBroadcastPkts      Counter64,
```

```
        ifHCOutOctets            Counter64,
        ifHCOutUcastPkts         Counter64,
        ifHCOutMulticastPkts     Counter64,
        ifHCOutBroadcastPkts     Counter64,
        ifLinkUpDownTrapEnable   INTEGER,
        ifHighSpeed              Gauge32,
        ifPromiscuousMode        TruthValue,
        ifConnectorPresent       TruthValue,
        ifAlias                  DisplayString,
        ifCounterDiscontinuityTime TimeStamp
    }
```

The original MIB-2 interfaces group counted multicast and broadcast packets together, *eg* in `ifInNUcastPkts`.

21.13.10 ETHERLIKE-MIB

RFC 3635 (originally **RFC 1650**) defines a MIB for "Ethernet-like" interfaces. The primary goal is to enable the collection of statistics on collisions and other Ethernet-specific behaviors. Several new tables are defined.

The table `dot3StatsTable` contains additional per-interface attributes; the name refers to the IEEE designation for Ethernet of 802.3. The table represents a sparse extension of the original `ifTable`, in the sense of *21.13.6 Table Augmentation* (where this table was used as the example).

The rows of the table mostly consist of counters for various errors and other noteworthy conditions:

- Alignment errors: the number of bits in the frame is not divisible by 8
- CRC checksum failure
- Frames that experienced exactly one collision
- Frames that experienced more than one collision
- Signal quality errors. SQE is specific to 10 Mbps Ethernet
- Deferred transmissions; when the station tried to send, the line was not idle
- Late collisions: the only way a collision can occur after the slot time is passed is if the physical Ethernet is too big or if collision-detection is failing. See *2.1.5 The Slot Time and Collisions*
- Excessive collisions: the frame experienced 16 collisions and the sender gave up
- Other hardware errors (`dot3StatsInternalMacTransmitErrors` and `dot3StatsInternalMacReceiveErrors`)
- Carrier sense errors ("carrier sense" refers to the collision-detection mechanism; there is no actual carrier)
- Frames longer than 1500 octets
- For Ethernets that encode data as symbols (*eg* 100 Mbps Ethernet's 4B/5B), frames arriving with a corrupted symbol

There is also an attribute to describe the Ethernet chipset.

The dot3HCStatsTable is like the dot3StatsTable except its counters are 64 bits instead of 32 bits.

The table dot3CollTable lists, for each N, how many packets experienced exactly N collisions before successful transmission or discard. Ethernet generally allows a maximum N of 16. The table is indexed by the pair ⟨ifIndex,N⟩. The dot3StatsTable contains this information for N=0,1,16.

The table dot3ControlTable contains information on those Ethernet-like interfaces that also support the so-called MAC Control sublayer; the dot3PauseTable is similar.

21.13.11 IP-MIB and IP-Forward MIB

The MIB-2 ip group is really about two separate things: strictly local information about IP packets and delivery, and the forwarding table, which affects a much larger collection of nodes. All nodes have local IP information, but only routers do significant forwarding. It did not take long for these two subgroups to separate.

The IP-Forward MIB began in **RFC 1354**, just over a year after the original MIB-2 in **RFC 1312**; this created a new home for forwarding-table information. By the time the first SNMPv2 MIB for the ip group came out in **RFC 2011**, the ipRouteTable once part of that group was declared obsolete.

21.13.11.1 IP-MIB

The current version of the IP-MIB is **RFC 4293**. It allows enabling or disabling of forwarding (ipForwarding, setting the default TTL (ipDefaultTTL), and, for IPv4 only (as it is not a configurable parameter for IPv6) the fragment-reassembly timeout (ipReasmTimeout).

When IP addresses appear in these tables, the defining entry is almost always prefaced by an object of type InetAddressType, which can take values ipv4(1) and ipv6(2) (and a few other values for special cases).

Two tables are devoted to IP statistics: the ipSystemStatsTable for system-wide IP information and the ipIfStatsTable for interface-specific information. Each table includes the IP address type (that is the IP version) in its index, meaning that separate statistics are kept for IPv4 and IPv6 traffic. The system table is indexed by the IP address type alone; the interface table is indexed by that and the interface number (the MIB-2 ifIndex).

The ipIfStatsTable contains, for each interface, counts of packets (and octets) in and out for broadcast, multicast and unicast, counts of forwarded, reassembled and fragmented packets, counts of packets with any of several kinds of errors, and related additional counts. An entry for the counter refresh rate is also provided. When appropriate, 64-bit counters are provided; usually the equivalent 32-bit counter is *also* provided. The ipSystemStatsTable provides all-interface summaries of these same counts.

The tables ipv4InterfaceTable and ipv6InterfaceTable represent information about what IP addresses are assigned to each interface. These are indexed by the interface index alone, meaning that the tables cannot accurately represent an interface with more than one IP address of the same type.

Additional IP-address information is contained in the ipAddressPrefixTable primarily for IPv6 and the ipAddressTable for both IPv4 and IPv6. These tables contain information about what IP addresses are assigned to what interfaces and where these addresses came from (*eg* DHCP, *7.10 Dynamic Host*

Configuration Protocol (DHCP), or Prefix Discovery, *8.6.2 Prefix Discovery*). Indexed by the IP address itself (and also the `ipAddressAddrType`), these tables thus support the possibility that one interface has multiple IP addresses (this is particularly common for IPv6).

The `ipNetToPhysicalTable` represents the map from local IP addresses to physical LAN addresses, as created by either ARP for IPv4 or Neighbor Discovery for IPv6. In addition to the interface, the IP address and the physical address, tle table also contains a timestamp indicating when a given entry was last updated or refreshed, an indication of whether the address mapping is dynamic, static or invalid, and, finally, an attribute `ipNetToPhysicalState`. The values for this last are `reachable(1)`, `stale(2)` for expired reachability, `delay(3)` and `probe(4)` relating to active updates of the reachability, `invalid(5)`, `unknown(6)` and `incomplete(7)` indicating that ARP is in progress. See *7.9.1 ARP Finer Points*.

There is also a simple version of the forwarding table known as the Default Router Table. This contains a list of "default", or, more accurately, "initially configured" routes. While this does represent a genuine forwarding table, it is intended for nodes that do not act as routers and do not engage in routing-update protocols. The table represents a list of "default" routes by IP address and interface, and also contains route-lifetime and route-preference values.

The `ipv6RouterAdvertTable` is used for specifying timers and other attributes of IPv6 router advertisements, *8.6.1 Router Discovery*.

Finally, the IP-MIB contains two tables for ICMP statistics" `icmpStatsTable` and `icmpMsgTable`. The latter keeps track, for example, of how many pings (`ICMP Echo`) and other ICMP messages were sent out; see *7.11 Internet Control Message Protocol*.

21.13.11.2 IP-Forward MIB

Information specific to a host's IP-forwarding capability was first split out from the MIB-2 ip group in **RFC 1354**; it was updated to SNMPv2 in **RFC 2096** and the current version is **RFC 4292**. The original MIB-2 ip group left off at OID mib-2.ip.23; the new IP-Forward MIB begins at mib-2.ip.24.

There have been three iterations of an SNMP-viewable IP forwarding table since the original **RFC 1213** ip group's `ipRouteTable` at mib-2.ip.21. Here are all four:

- `ipRouteTable`
- `ipForwardTable`
- `ipCidrRouteTable`
- `inetCidrRouteTable`

Each new version has formally deprecated its predecessors.

The first replacement was `ipForwardTable`, described in **RFC 1354**. It defines the OID `ipForward` to be mib-2.ip.24; the new table is at `ipForward.2`. This table added several routing attributes, but perhaps more importantly changed the indexing. The index for `ipRouteTable` was the IP destination network `ipRouteDest`, alone. The new table's index also includes a quality-of-service attribute `ipForwardPolicy`, usually representing the IPv4 Type of Service field (now usually known as the DS field, *7.1 The IPv4 Header*). This inclusion allows the `ipForwardTable` to accurately represent routing based on ⟨dest,QoS⟩, as discussed in *9 Routing-Update Algorithms*. Such routing is sometimes called

"multipath" routing, because it allows multiple paths to a given destination based on different QoS values. However, the mask length is *not* included in the index, making `ipForwardTable` inadequate for representing CIDR routing.

The index also includes the next_hop, which for the actual forwarding table does not make sense – the next_hop is what one is looking up, given the destination – but which works fine for SNMP. See the comments about SNMP indexes with more attributes than expected in *21.7 SNMP Tables*. The index even includes an attribute `ipForwardProto` that represents the routing-update protocol that is the source of the table entry: `icmp(4)`, `rip(8)` (a common distance-vector implementation), `is-is(9)` and `ospf(13)` (two link-state implementations) and `bgp(14)`.

In addition to the next_hop, this table also includes attributes for `ipForwardType` (*eg* local vs remote), `ipForwardAge` (the time since the last update), `ipForwardInfo`, `ipForwardNextHopAS`, and several routing metrics. The purpose of `ipForwardInfo` is to provide an OID that can be accessed to provide additional information specific to the routing-update algorithm in use. The `ipForwardNextHopAS` allows the specification of the next_hop Autonomous System number, usually relevant only when BGP (*10.6 Border Gateway Protocol, BGP*) is involved. (If the AS number is not relevant, it is set to zero.)

The second iteration of the SNMP-viewable IP forwarding table is `ipCidrRouteTable`, appearing in **RFC 2096** and located at `ipForward`.4 (and returning to the practice of calling it a "route" rather than a "forward" table). This table adds the address mask, `ipCidrRouteMask`, to the index, finally allowing distinct routes to 10.38.0.0/16 and 10.38.0.0/24. The quality-of-service field `ipCidrRouteTos` remains in the index (as does the destination), and is now firmly identified with the IPv4 Type of Service (DS) field. The routing-update algorithm was dropped from the index.

This table also adds an attribute `ipCidrRouteStatus` of type `RowStatus` and used for the creation and deletion of entire rows (that is, forwarding table entries) under the control of SNMP. We will return to this process in *21.14 Table Row Creation*.

The third (and still current) version of the IP forwarding table is `inetCidrRouteTable`, introduced in **RFC 4292** and located at `ipForward`.7. The main change introduced by this table is the extension to support IPv6: the IP-address columns (*eg* for destination and next_hop) have companion columns for the address *type*: `ipv4(1)` and `ipv6(2)`. See *21.13.12 TCP-MIB* for further details.

The next_hop attribute (now two columns, with the addition of the address type) is still part of the index.

The address *mask* used in `ipCidrRouteTable` is now updated to be a prefix length, `inetCidrRoutePfxLen`. The quality-of-service field `inetCidrRoutePolicy` is an Object ID, declared to be "an opaque object without any defined semantics"; that is, it is at the implementer's discretion. The IPv4 ToS/DS field evolved in IPv6 to the Traffic Class field, *8.1 The IPv6 Header*.

Finally, routes are no longer required to list a single associated interface. The table makes use of the `InterfaceIndexOrZero` textual convention of **RFC 2863**, covering just this situation.

21.13.12 TCP-MIB

RFC 4022 contains some extensions to MIB-2's tcp group. The SNMPv2-based MIB embedded in **RFC 4022** repeats the MIB-2 tcp group and then adds new features within the mib-2.tcp (1.3.6.1.2.1.6) tree. **RFC 1213** stopped at mib-2.tcp.15; **RFC 4022** defines new objects starting at mib-2.tcp.17.

The new `tcpConnectionTable` is defined at mib-2.tcp.19, versus the original `tcpConnTable` at

mib-2.tcp.13. The newer table supports IPv6; like the inetCidrRouteTable above, each IP-address column now also has a companion address-type column, and addresses themselves are represented as OCTET STRINGs preceded by a length byte. This follows the rules for InetAddressType and InetAddress of **RFC 4001**, and the OID-suffix-encoding rules of **RFC 2851** §4.1. Because the length of an InetAddress isn't known in advance, the length must be included.

For IPv4 users used to the earlier tcpConnTable, this means that there are extra 1.4.'s prefixing the IPv4 addresses in the index, as in this example of host 10.0.0.5 connecting from port 54321 to web server 147.126.1.230:

tcpConnectionState.**1**.**4**.10.0.0.5.54321.**1**.**4**.147.126.1.230.80

The new table also includes a column representing the process ID of the process that has open the local end of the connection.

This table adheres to the SNMPv2 convention that index columns are not included in the data – the attributes are marked not-accessible. There are only two accessible columns, tcpConnectionState and tcpConnectionProcess.

TCP-MIB does *not* make available per-interface TCP statistics, *eg* the number of TCP bytes sent by eth0. Nor does it make available per-connection statistics such as packet-loss and retransmission counts or total bytes transmitted each way.

21.14 Table Row Creation

SNMPv2 also refined the mechanisms by which a manager can add a row to an agent's table. In principle, adding a row to a table is done with the Set() operation. Imagine a table **T** with three columns indexT, fruitT and primeT corresponding to **T**.1, **T**.2 and **T**.3. Imagine also that, right now, the table has three rows with indexT values 10, 11 and 12:

indexT	fruitT	primeT
T.1	**T**.2	**T**.3
10	apple	37
11	blueberry	59
12	cantaloupe	67

Then a new row ⟨13,durian,101⟩ might be added with the following multi-attribute Set() operation. The entries of the new row will have OIDs **T**.1.13, **T**.2.13 and **T**.3.13, and all we have to do to create the row is assign to these. Note that we are assigning to OIDs that, in the agent's current database, do not yet exist.

Set((**T**.1.13,13), (**T**.2.13,durian), (**T**.3.13,101))

Of course, this raises some questions. Will the agent actually allow this? What happens if these Set() operations are performed individually rather than as a group? Is there any way to *delete* this newly added row? And, more seriously, what happens if some *other* manager tries to insert at the same time the row ⟨13,feijoa,103⟩?

We now turn to the specific row-creation mechanism of RMON.

21.14.1 RMON

RMON, for Remote MONitoring, was an early attempt (first appearing in **RFC 1271** eight months after MIB-2's **RFC 1213**) at having an SNMP agent take on some monitoring responsibilities of its own. The current version is in **RFC 2819**. The original RMON, now often called RMON1, only implemented LAN-layer monitoring, but this was later extended to the IP and Transport layers in RMON2, **RFC 4502**. We will here consider only RMON1.

RMON implements only passive monitoring; there is no capability for the remote agent to send out its own SNMP queries, or even pings (though see *21.14.3 PING-MIB*). Monitoring is implemented by putting the designated interface into promiscuous mode (*2.1 10-Mbps Classic Ethernet*) and capturing all traffic. In modern fully-switched Ethernets, hosts simply do not see traffic not actually addressed to them, and so RMON would need to be implemented on a switch or router to be of much practical use.

An agent's RMON activity is controlled by an SNMP manager through the insertion of new rows in various **control tables**. The mechanism for doing this is our primary concern here.

RMON statistics are divided into ten groups, of which we will consider only the following:

- **statistics**: counts of errors and counts of packets in size ranges 0-64, 65-127, 128-255, 255-511, 512-1023 and 1024-1518 octets.
- **history**: The statistics group data, taken at regular intervals
- **hosts**: The Ethernet senders and receivers seen by an interface
- **host top N**: The top-N senders or receivers
- **matrix**: Information on traffic by ⟨sender,receiver⟩ pair

21.14.1.1 Statistics (and use of EntryStatus)

The `etherStatsTable` is, by default, empty, and is indexed by `etherStatsIndex` which is an opaque INTEGER. The column `etherStatsDataSource` represents the OID of a specific interface number as defined by `ifTable`; for example, the interface with `ifNumber` = 6 would be represented by mib-2.2.2.1.1.6. One column, `etherStatsStatus`, has type `EntryStatus` as follows:

```
valid(1),
createRequest(2),
underCreation(3),
invalid(4)
```

The attributes `etherStatsDataSource` and `etherStatsStatus` were initially `read-write`, which was later changed to status `read-create` as they can only be written to as part of the creation of a new row. There is one more `read-create` attribute, `etherStatsOwner`, which is a manager-supplied string identifying that particular manager (perhaps by IP address and hostname and, if a human is involved, appropriate additional identification).

To enable statistics collection, the SNMP manager creates a row in the agent's `etherStatsTable` by setting the three attributes `Status`, `DataSource` and `Owner` (where we have left off the attribute-name prefix `etherStats` for readability). Once this is done and the `Status` is set to `valid`, the agent begins collecting data about the desired interface. The manager can read the data as it desires, by accessing that

particular row. Data collection ends when the manager sets the `etherStatsStatus` for the row to `invalid`, though it is up to the agent whether the row is actually deleted at that point.

The `etherStatsIndex` column is not actually writable, but the manager must still select a value for the index. One approach is to read the entire table and identify the first unused index value. Other managers, however, may be creating new rows in the agent at the same moment, and so a row index that was available moments before may now be unavailable. We return below to how such conflicts are prevented. One common strategy for reducing the chance of row-creation collisions is to choose a value for the index at random.

It is often possible for a manager to create the desired row with a single `Set()` operation. However, **RFC 2819** requires that the the `Status` attribute in a request for creation of a new row must be `createRequest`, and a second `Set()` operation is therefore always required to transition to `valid`.

If **T** is the tree OID `etherStatsEntry` and 157 is the manager's chosen index, and the manager wants to monitor `ifIndex` = 6, it could send the following as a single operation.

> Set((**T**.Status.137,createRequest), (**T**.DataSource.137,mib-2.2.2.1.1.6), (**T**.Owner.137,*owner*))

at which point the agent will create the row with status `underCreation`. The value `createRequest` is used only in manager requests and never appears in actual rows on the agent. The manager will then follow with

> Set(**T**.Status.137,valid)

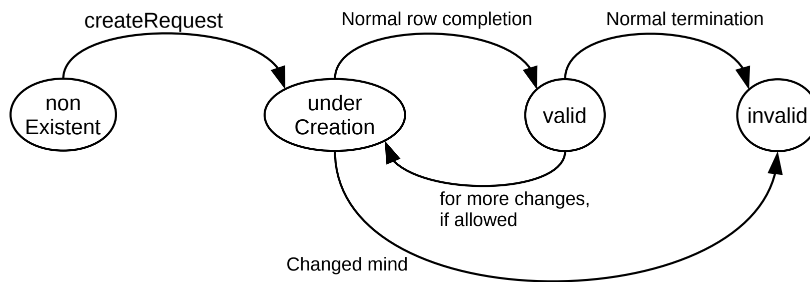
If the manager wants or needs to create the entry piecemeal, it can do so as follows:

> Set(**T**.Status.137, createRequest)
> Set(**T**.DataSource.137, mib-2.2.2.1.1.6)
> Set(**T**.Owner.137, *owner*)
> ...
> Set(**T**.Status.137, valid)

Immediately following the first Set(**T**.Status.137, createRequest) the agent will again create the row and mark it as `underCreation`, but this time the new row will be missing several columns.

The `Status` attribute must be specified in the very first `Set()` operation for the row.

Existing rows may not have their `Status` set to `createRequest`. The primary legal state transitions are as follows:



Transition diagram for EntryStatus. When a row is requested with `createRequest`, it is actually created as `underCreation`.

In some cases, a row can be edited by the manager by changing the status from `valid` back to `underCreation`. However, many specific row-creation implementations require that no changes can be made after a row is marked `valid`. To change an existing row in such a case, the row should be marked `invalid` and a new row created.

These transition rules for `createRequest` prevent two managers from simultaneously creating rows with the same index. Suppose, for example, that managers M_A and M_B each attempt to create a new row 137 by executing

Set(**T**.Status.137, createRequest)

If M_A's Set() request is the first to be acted upon, row 137's `Status` becomes `underCreation`. Later, when M_B's Set() request is processed, the row will no longer be nonexistent. So, from the diagram above, M_B's arriving `createRequest` is invalid: there is no `createRequest` arc from any node other than `nonexistent`. M_B's Set() operation above, and therefore any of M_B's other Set() operations, will fail.

This strategy is meant to prevent only accidental row-creation conflict; it will not prevent M_B from hijacking M_A's row, or from marking it as `invalid` (and thus effectively deleting it). But this is generally understood as an inconsequential risk; legitimate managers should be trustable.

Any authorized SNMP manager can read all the `etherStatsTable` records; there is no requirement that only the creator of a row can read that row.

It is quite possible for multiple rows to be created all referring to the same interface, *eg* by multiple managers. Although it is not forbidden, there is no reason for one manager to create two rows for the same interface.

The four-state `EntryStatus` type appeared in the original SNMPv1 **RFC 1271**. Updates to RMON to SNMPv2 have left `EntryStatus` alone, but SNMPv2 has also introduced the six-state `RowStatus` type which is more likely to be used by new MIBs. We will turn to this in *21.14.2 SNMPv2 RowStatus*.

21.14.1.2 History

The history group allows for collection of a series of samples of data from the Statistics group, above, at regular intervals. The group consists of two tables, `historyControlTable` where the manager creates rows to manage the process, and `etherHistoryTable` containing the actual data.

The `historyControlTable` contains entries for (again omitting the `historyControl` prefix from attribute names) `DataSource`, representing the interface to be examined, and the `Status` and `Owner`

attributes as above. As with the `etherStatsTable`, when the manager creates a new row it specifies a value for the index, with full name `historyControlIndex`, again perhaps chosen at random.

The manager must also specify two additional attributes: `Interval` representing the time in seconds between data-collection events, and `BucketsRequested` representing the *requested* maximum number of interface records to be kept by the agent. When the agent reaches the maximum, each new record replaces the oldest previous record.

The control table contains one agent-created attribute: `BucketsGranted` indicating the actual maximum number of interface records to be kept.

Once the control table row is marked `valid`, the agent starts accumulating one statistics record for the specified interface every `Interval` seconds. These records are numbered consecutively; the number of a given record is its `SampleIndex`. The agent keeps only `BucketsRequested` records, so once `SampleIndex` reaches that value then, whenever record N is added, record N – `BucketsRequested` is deleted.

All this data collected by the agent is stored in `etherHistoryTable`, which contains columns for all the statistics in `etherStatsTable` except for the counts of packets in the various size ranges. The table is indexed by the pair of values `Index` corresponding to `etherControlIndex` and the record number `SampleIndex`. As such, this table is the disjoint union of multiple independent series of consecutively numbered records, one series for each `Index` value in the control table, that is, one series for each manager request.

As an example, suppose the manager asked for history statistics to be kept for a given interface at a rate of once a minute, the `BucketsGranted` is 70, and the control-index value for this request is 491. After one hour, the table contains records with sample indexes 1-60; the full row-index values are ⟨491,1⟩ through ⟨491,60⟩. After one hour and eleven minutes, record ⟨491,71⟩ replaces record ⟨491,1⟩. After two hours, records with sample indexes of 51-120 are available. The manager might return once an hour and retrieve the most recent 60 records.

A manager might also create *two* control-table records, one holding 25 records taken at 1-hour intervals and another holding 60 records taken at 1-minute intervals. If all is well, that manager might download the first table once a day, and entirely ignore the second table. The manager always has available these 1-minute records for the last hour, though, and can access them as needed if a problem arises (perhaps signaled by something else entirely).

A manager can easily retrieve only its own rows from the `etherHistoryTable`. Let **T** be the root of the `etherHistoryTable`, which has columns 1-15. Suppose again a manager has created its controlTable row with a value for `historyControlIndex` of 491; the manager can then retrieve the first of its data rows with the following; note that each OID contains the column number and *part* of the row index.

> GetNext(**T**.1.491, **T**.2.491, ..., **T**.15.491)

If the history-table rows associated with 491 have sample-index values ranging from 37 to 66, the above GetNext() will return the row indexed by ⟨491,37⟩; that is, the values paired with the following OIDs:

> **T**.1.491.37, **T**.2.491.37, ..., **T**.15.491.37

Subsequent GetNext()s will return the subsequent rows associated with control entry 491: row ⟨491,38⟩, row ⟨491,39⟩, *etc*. If the `etherHistoryTable` had been indexed in the reverse order, with the sample index first and the `historyControlIndex` second, a substantial linear search would be necessary to locate the first row with a given value for the control index.

21.14.1.3 Hosts

The host group allows an agent to keep track of what other hosts – in RMON1 identified by their Ethernet address – are currently active.

Like the `historyControlTable`, the `hostControlTable` allows a manager to specify `DataSource`, `Status` and `Owner`; the manager also specifies `TableSize`.

Once the control-table entry is `valid`, the agent starts recording hosts in the `hostTable`, which is indexed by the control-table index and the host Ethernet address. The agent also records each new host's `CreationOrder` value, an integer record number starting at 1.

The `hostTable` also maintains counters for the following per-host attributes; these are updated whenever the agent sees another packet to or from that host. We will revisit these in the `hostTopNTable`, following.

- InPackets
- OutPackets
- InOctets
- OutOctets
- OutErrors
- OutBroadcastPkts
- OutMulticastPkts

When the number of host entries for a particular control-table index value exceeds `TableSize` – that is, the new host's `CreationOrder` would exceed `TableSize`, old entries are removed *and all hosts in the table are given updated* `CreationOrder` *values*. That is, if the table of size three contains entries for hosts A, B and C with `creationOrder` values of 1, 2 and 3, and host D comes along, then A will be deleted and B, C and D will be given `creationOrder` values of 1, 2 and 3 respectively. This is quite different from the record-number assignments in the `etherHistoryTable`, where the `SampleIndex` record numbers are immutable.

A consequence of this is that the `CreationOrder` values are always contiguous integers starting at 1.

Entries are deleted based on order of creation, not order of last update. The `hostTable` does not even have an attribute representing the time a given host's entry (or entries) was last updated.

As a convenience, the data in `hostTable` is also made available in `hostTimeTable`, but there indexed by the control-table index and the `CreationOrder` time. This makes for potentially faster lookup; for example, the manager always knows that the record with `CreationOrder` = 1 is the oldest record. More importantly, this alternative index allows the manager to download the most recent entries in a single step.

Whenever a host is deleted because the table is full, the `CreationOrder` values assigned to other hosts all change, and so the indexing to `hostTimeTable` changes. Thus, a manager downloading rows from `hostTimeTable` one at a time must be prepared for the possibility that what had earlier been row 3 is now row 2, and that a host might be duplicated or skipped. To help managers deal with this, the *control* table has an entry `LastDeleteTime` representing the time – in TimeTicks since startup – of the last deletion and thus `CreationOrder` renumbering. If a manager sees that this value changes, it can, for example, start the data request over from the beginning.

21.14.1.4 Host Top N

The hostTopN table is a report prepared by the agent about the top N entries (where N is manager-supplied) in the hostTable, over an interval of time. The manager specifies in the control table the Status and Owner attributes, the HostIndex control-table index value from hostControlTable, and also the Duration (in seconds) and the RateBase indicating which of the following hostTable statistics is to be used in the ranking:

- InPackets
- OutPackets
- InOctets
- OutOctets
- OutErrors
- OutBroadcastPkts
- OutMulticastPkts

The value of N is in the attribute RequestedSize; the agent *may* reduce this and communicates any change (or lack of it) through the attribute GrantedSize.

Once the control-table row becomes valid, the agent then starts maintaining counters for all the entries in the part of hostTable indexed by HostIndex, and at the end of the Duration sorts the data and places its top-N results in the hostTopN table. If, during the interval, some hosts were removed from the hostTable because the table was full, the results may be inaccurate.

All result statistics are inaccessible until the Duration has elapsed and the particular hostTopN report has run its course, at which point the results become read-only.

21.14.1.5 Matrix

The matrix group allows an agent to collect information on traffic flow indexed by the source and destination addresses. The manager begins the process by supplying attributes for Status, Owner, the usual Index, the interface DataSource, and the maximum table size.

Once the row is valid, the agent begins collecting, for every ⟨source,destination⟩ pair, counts of the number of packets, octets and errors. The record for ⟨A,B⟩ counts these things sent by A; the corresponding record for ⟨B,A⟩ counts the reverse direction. The actual matrixSDTable is indexed by the manager-supplied Index value and the source and destination Ethernet addresses in that order.

A companion table (or view) is also maintained called matrixDSTable, that lists the same information but indexed by destination first and then source. This view is not present to supply information about the reverse direction; that would be obtained by reversing source and destination in the matrixSDTable. Rather, the matrixDSTable allows a manager to extract all information about a single *destination* D in a single SNMP tree-walk operation of the prefix matrixDSTable.Index.D. This is similar to the indexing discussion at the end of *21.14.1.2 History*. See exercise 9.

21.14.2 SNMPv2 RowStatus

RMON made do with four values for its EntryStatus attribute: createRequest, underCreation, valid and invalid. SNMPv2 still supports EntryStatus, but the preferred form is the six-value RowStatus, **RFC 2579**, with options

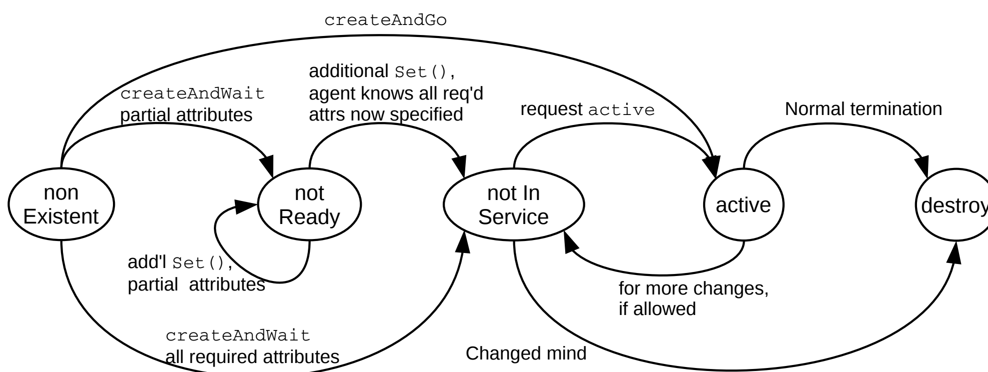
- active: like valid
- notReady: the agent knows that the row is not complete
- notInService: the row is complete, but has not yet been activated
- createAndGo: like createRequest followed by valid
- createAndWait: like createRequest
- destroy: like invalid

The createAndGo option allows a manager to make a single new-row request to the agent, and, if successful, the agent will immediately set the row status to active. Using EntryStatus requires two steps: first to set the new row to underCreation and then to set it to valid.

The manager can ask for any status except notReady. Every row created will, however, be marked at the agent with one of notReady, notInService or active.

As with EntryStatus, the manager must still choose an index value. Pseudorandom selection remains appropriate in many cases, but the agent may also supply, if the MIB file supports it, a recommendedNextIndex attribute.

Here is a simplified RowStatus state-transition diagram. Not all links to destroy, or from a node back to itself, are shown. See **RFC 2579**, p 9, for more details.



Simplified transition diagram for RowStatus

After the manager issues a createAndWait, the agent fills in the attributes provided by the manager, and any other default attributes it has available. The manager, if desired, can now Get() the entire row, and find out what values are still missing. These values will be reported as noSuchInstance. Alternatively, the manager may simply know that it has more attributes to Set().

If the agent knows the row is missing attributes necessary for activation, it will set the `RowStatus` to `notReady`, otherwise to `notInService`. A `notInService` row can, of course, still have undefined `read-only` attributes that the agent will later set after activation. A `notInService` row can also still have attributes the manager intends to set before activation, but the agent has given default values in the interim.

Once the manager has set all the attributes required for activation, it sets the `RowStatus` attribute to `active`, and agent activity begins.

As with `EntryStatus`, two managers cannot simultaneously create the same row. If they were to try to do so, the second would be attempting to set `RowStatus` to `createAndWait` or `createAndGo` for a row entry that already existed, but from the diagram such a transition is not allowed.

21.14.3 PING-MIB

The idea behind the ping MIB is to allow a manager to ask an agent to repeatedly ping a target, *7.11 Internet Control Message Protocol*, and then to report the success rate. Cisco Systems has a ping MIB at ftp://ftp.cisco.com/pub/mibs/v2/CISCO-PING-MIB.my, but we will use the IETF alternative in **RFC 2925**/**RFC 4560**. The latter is officially titled DISMAN-PING-MIB (DISMAN is short for DIStributed MANagement). Both ping MIBs make use of the `RowStatus` convention of the previous section.

The manager can ping the target itself, *if* there are no firewalls in the way, but the result may likely be different from that obtained by the agent.

The actual MIB supports multiple types of `ping` besides the usual ICMP Echo Request, but we will consider only the latter.

The DISMAN version has control table `pingCtlTable` and the results appear in `pingResultsTable`; in the Cisco version the same table is used for both control and results. The results include the minimum, maximum and average RTT, the number of pings sent and the number of responses. The results table also has an attribute `OperStatus` to indicate whether the test has stopped.

The `pingCtlTable` contains a wide range of options for the actual ping request, including options to specify the outgoing IP address for multihomed agents. The more familiar options, however, include

- `pingCtlTargetAddress`: and address type, as IPv6 is supported
- `pingCtlDataSize`: how big each ping packet is
- `pingCtlTimeout`: how long before the agent gives up on any one ping
- `pingCtlProbeCount`: the number of pings to be sent
- `pingCtlFrequency`: the interval between pings

The table also includes `pingCtlRowStatus` of type `RowStatus`, above, and `pingCtlOwnerIndex` of type `SnmpAdminString`, which is a fancier way of identifying the manager than the RMON `Owner` attributes, and which can include some SNMPv3 credentials as desired.

The new row is created, `RowStatus` is set to `active` (perhaps through `createAndGo`), and the test begins.

By setting the control table's `RowStatus` to `destroy`, a manager can attempt to halt a ping series in progress.

21.15 SNMPv3

SNMP version 3 added authentication and encryption to SNMPv2c, but made relatively few other changes except to nomenclature. The original definitions are in **RFC 3410** through **RFC 3415**; **RFC 3410** first appeared as **RFC 2570**. SNMPv3 introduced the User Security Model, or **USM**, in which agents allow manager access only if the manager has presented an appropriate **key**, derived ultimately from a passphrase. The agent's response can be either digitally signed or encrypted, as desired.

SNMPv3 did make several terminology changes. Any SNMP node – either manager or agent – is now known as an SNMP **entity**. An entity is divided into two parts; the first is the SNMP **engine** consisting of the message dispatcher, the message processor, and the security and access-control subsystems. The second part consists of the various SNMP **applications**, consisting, for agents, primarily of the command responder (responding to `Get()` and `GetNext()`, *etc*). One goal of this architectural division is to separate the applications from the authentication mechanisms. Another goal, however, is to provide a framework in which future new applications can easily be supported.

It is the SNMP engine that must implement all the new security provisions.

21.15.1 What Could Possibly Go Wrong?

RFC 3414 analyzes the the risks of inadequate SNMP security, and identifies two primary and two secondary threats. The primary threats are as follows:

- Unauthorized **modification** of information in transit: that is, **man-in-the-middle** attacks in which A's message to B is intercepted and modified by an intermediate node M.
- **Masquerade**: unauthorized operations by a manager (or agent) masquerading as an authorized entity. Use of guessed or cracked passwords would fall into this category.

The secondary threats are these:

- **Message Stream Modification**: This includes re-ordering messages, and, perhaps most importantly, **replay attacks**. For example, even if no keys are ever discovered, an attacker might learn that a certain message causes a certain server to reboot. Retransmitting that message could have serious consequences.
- **Disclosure**: reading by an eavesdropper of the management information in transit.

All but the last, disclosure, can be addressed by appropriate digital signatures and timestamps; encryption is only needed when disclosure is an active concern. Because disclosure is not always a significant concern, and perhaps because when **RFC 2574** was written in 1999 the unlicensed export of encryption technology from the United States was illegal, SNMPv3 defines two separate new levels of security:

- authentication-only, addressing modification, masquerade and replay
- encryption, addressing the above and disclosure

These correspond to three values for the `snmpSecurityLevel` textual convention (in which "priv" abbreviates "privacy"):

- `noAuthNoPriv`: no authentication
- `authNoPriv`: authentication, but no encryption

- `authPriv`: encryption of all messages

Encryption (privacy) implies authentication; encrypted messages also include the authentication signatures described below.

21.15.2 Cryptographic Fundamentals

SNMPv3 authentication is based on **cryptographic hash functions**: functions which take a data string of arbitrary length and return a fixed-length hash, in such a way that

- Knowing the hash value sheds no practical light on the original data
- Given a hash value, there is no feasible way to find a message yielding that hash

Further details are at *22.6 Secure Hashes*.

The Internet checksum of *5.4 Error Detection* fails as a cryptographic hash, for example, because given a message m and a hash value h′, one can calculate hash(m) = h and then append to m a two-byte string based on h′–h, yielding a message m′ for which hash(m′) = h′.

The two cryptographic hash functions originally supported by SNMPv3 in **RFC 3414** are MD5, which produces a 128-bit hash, and SHA-1, which produces a 160-bit hash. Since the publication of **RFC 3414** in 2002, vulnerabilities in each of these hash functions have been discovered. The SNMP framework in principle allows easy substitution of new hash functions, but **RFC 7860**, standardizing the use of the SHA-2 family, did not appear even in draft form until 2013.

If two parties share a secret key k, the basic hash-based way to sign a message m is to append to it the value hash(m⌒k), where m⌒k is the result of appending k to m. An eavesdropper will see m and hash(m⌒k) but this will provide no information about k, and, similarly, an attacker will, given a message m, not be able to generate the value hash(m⌒k) without knowing k. Because some hash functions are vulnerable to the so-called "length-extension attack" when used this way, the actual signature is often slightly more involved (*21.15.5 Passwords and Keys* and *22.6.1 Secure Hashes and Authentication*), but hash(m⌒k) is a good example of the basic concept.

The SNMP encryption mechanism is also based on shared secret keys (*22.7 Shared-Key Encryption*); that is, public-key encryption is not used. **RFC 3414** describes the use of the Data Encryption Standard cipher, or DES, which uses 56-bit keys. Later, **RFC 3826** introduced the use of the Advanced Encryption Standard cipher, or AES, which in SNMP users a 128-bit key. (Both DES and AES are discussed briefly in *22.7.2 Block Ciphers*.) DES is, if anything, even more vulnerable than MD5 and SHA-1, due to the limited key length; AES is a much stronger choice.

Shared-secret encryption is based, abstractly, on an encrypting function E(p,k) that takes a plaintext message p and a key k and returns the encrypted ciphertext. Similarly, there is a decrypting function D(c,k) that takes an encrypted ciphertext message c and the key k and returns the original plaintext.

21.15.3 SNMPv3 Engines

The SNMPv3 engine is the component of an SNMP entity charged with ensuring security. Each SNMPv3 engine – manager or agent – has an identifier known as the **snmpEngineID**. This is a string that, by default, incorporates the node's IP or Ethernet address and additional standard information. While it is common

to leave this default setting unchanged, it is also possible to replace it using a site-based naming scheme, perhaps including the organization name and a locally assigned serial number; see RFC 3411.

In any SNMPv3 exchange, one SNMPv3 engine is designated **authoritative**. For a `Get()` or `Set()` request, or any other request that requires a response, it is always the agent that is authoritative. It is the job of the authoritative engine to validate the message received from the other engine.

For either authentication-only or encrypting security levels, the nonauthoritative engine must present a pair consisting of the following:

- `userName`: a human-readable string identifying a person or manager-computer
- `authParameters`: a string holding data specific to the authentication/encryption mechanism in use that serves to authenticate the user. This might be thought of as a passphrase, except one goal is never to send passphrases in the clear.

The authoritative engine must keep a table of all the `userName` values it recognizes, and, for each, the appropriate key with which to validate the user.

Any SNMPv3 engine also keeps track of two 32-bit attributes

- `snmpEngineTime`: the number of seconds since the engine last rebooted
- `snmpEngineBoots`: the number of times the engine has rebooted

The pair of these, which we will abbreviate as ⟨Time,Boots⟩, uniquely identifies a point in time for an entity (to the nearest second). They are used to prevent replay attacks; two messages sent more than one second apart will never have the same ⟨Time,Boots⟩ timestamp.

Note that keeping track of `snmpEngineBoots` implies that the entity have some form of persistent storage.

When a nonauthoritative engine (for our purposes, the manager) wants to send a request to an authoritative engine (the agent), the first step is to find out the agent's `snmpEngineID` and then its current ⟨Time,Boots⟩ value. This is done through an initial two-step discovery process. The first request contains an empty `varBindList`, an empty engineID, a username of the empty string and a security level of `noAuthNoPriv`. The authoritative side sends a response containing its engineID. The second step is to send a request, again with an empty `varBindList` but now containing a valid ⟨username,key⟩ pair. The value for ⟨Time,Boots⟩ is ⟨0,0⟩. The authoritative engine now responds with a message including its actual ⟨Time,Boots⟩.

21.15.4 Message Authentication

After discovering the authoritative engine's ⟨Time,Boots⟩, the nonauthoritative engine stores the authoritative engine's `snmpEngineBoots` and also the difference between the authoritative engine's `snmpEngineTime` and its own clock time. It can now use this difference to approximate the authoritative engine's `snmpEngineTime` at any point in the future. Every message from the nonauthoritative engine will include its estimate of the authoritative engine's current ⟨Time,Boots⟩ value. Relative drift between the two engines' clocks will eventually mean this estimate fails, but, as we shall see, it can be expected to be close enough for quite a while.

The authoritative engine accepts a non-empty request only if all three of the following hold, where Time and Boots are the values submitted by the nonauthoritative engine and `snmpEngineTime` and

`snmpEngineBoots` are the authoritative engine's own values:

- Boots = `snmpEngineBoots`
- `snmpEngineBoots` < $2^{31} - 1$
- Time and `snmpEngineTime` differ by less than 150 seconds

The maximum allowable clock drift, in other words, is 150 seconds. If the two clocks drift by more than that, the nonauthoritative side must again go through the synchronization process outlined at the end of the previous section.

The actual message signing is based on a shared secret key, `authKey`, negotiated previously between user and agent. Because the key is usually specific to the agent, it is sometimes called a *local* key, **kl**.

21.15.5 Passwords and Keys

Users are identified by `userName` values, and also (usually) have human-readable passwords. These passwords must be converted first into keys, in such a way that each ⟨user,agent⟩ pair has its own unique key. We will start with a mechanism commonly used for authentication-only security; it is similar for both the MD5 and SHA-1 hashes. We will denote the hash function (either MD5 or SHA-1) by `hash()`.

The first step is to create a **digest** based on the password. It would suffice in principle to set the digest to hash(password). In order to make the password→key conversion process relatively slow, however, so as to hamper brute-force attacks, Appendix A of **RFC 3414** recommends repeating the password (logically, at least) to fill a buffer 2^{20} bytes in length. The `hash()` function is then applied to this megabyte string. (For another perspective on intentionally slow password operations, see *22.6.2 Password Hashes*.)

The next step is to take the engineID of the agent for which the user is generating this particular key and take the hash of the concatenation of the digest, engineID and digest again:

kl = hash(digest⁀engineID⁀digest)

This is now the ⟨user,agent⟩ shared key, or *local* key. It must be entered (or computed) on the agent, and stored there. For MD5 it is 16 bytes long; for SHA-1 it is 20 bytes long.

This mechanism is, strictly speaking, optional; an agent does not know how manager keys were generated, and thus cannot enforce any particular mechanism when a manager's key is later changed as below. Any secure way to generate a unique key **kl** for each user and each agent would be sufficient. The mechanism here, though, has the advantage that any manager node can compute a user's local key directly from the password and the agent engineID; no keys need be stored by the manager. This allows a manager to use one password for multiple agents; compromise of any one agent and its attendant local key **kl** should not affect the security of other agents or of the original password.

21.15.6 Message Signing

An SNMP message **mesg** is now signed by a local key **kl** using the **Hash Message Authentication Code**, or HMAC (**RFC 2104**), as follows:

- **kl** is extended to a 64-byte **ek** by padding with zeroes.
- A constant string **ipad** is formed by repeating the octet 0x36 64 times.

- A constant string **opad** is formed by repeating 0x5c 64 times.
- Set **k1** = **ek** XOR **ipad**.
- Set **k2** = **ek** XOR **opad**.

The `authParameter` field – the actual digital signature – is now set to be the first 12 bytes of

hash(**k2**⌒hash(**k1**⌒**mesg**))

This is slightly more complicated than the hash(**mesg**⌒**kl**) mechanism suggested in *21.15.2 Cryptographic Fundamentals*, in order to make it more resistant to potential vulnerabilities in the hash() function; see *22.6.1 Secure Hashes and Authentication*.

The receiver can replicate this `authParameter` calculation and verify that it matches the value transmitted in the packet.

21.15.7 Key Changes

Suppose a user wants to change his or her password, and thus the key **k**. The manager will need to communicate the new key to the agent in such a way that it is not exposed to eavesdroppers. Here is the mechanism, where, again, `hash()` is either MD5 or SHA-1 as appropriate; we will use N to denote the length in bytes of the result of `hash()` (either 16 or 20).

The original key is here denoted `oldkey` and the new key is `newkey`.

The manager first chooses a string `random` of N bytes chosen as randomly as possible. A second N-byte string `delta` is then calculated as follows:

`temp` = `hash(oldkey`⌒`random)`

`delta` = `temp` XOR `newkey`

At this point, `random` and `delta` are sent – in the clear – from the manager to the agent. The agent can use `random` and `oldkey` to compute `temp`, and thus `newkey` = `delta` XOR `temp`. An eavesdropper cannot use `random` to find out anything about `temp` without knowing `oldkey`, and cannot get anything useful out of `delta` unless either `newkey` or `temp` is known.

The actual process is to combine `random` and `delta` into a single 2N-byte **keyChange** string, written to one of the key-change columns of `usmUserTable`, *21.15.9.1 The usmUserTable*.

Note that if an eavesdropper saves `random` and `delta`, and later discovers the user's `oldkey`, then `newkey` can be calculated easily. Using the language of *22.9.2 Forward Secrecy*, forward secrecy fails badly. However, if an eavesdropper later discovers `newkey`, there is no obvious way to find `oldkey`; see exercise 10.

To improve forward secrecy, and to simplify mass key generation generally, **RFC 2786** outlines an experimental mechanism to use Diffie-Hellman-Merkle key exchange (*22.8 Diffie-Hellman-Merkle Exchange*) to create and change keys, instead of the process above.

21.15.8 Creating Additional Users

Suppose we want to add user "alice" to an agent, together with Alice's local key **kl**. The problem is that there is no pre-existing shared key, and so no way to transmit **kl** to the agent via SNMP without risk of

eavesdropping.

One solution is to require **out-of-band** account creation, that is, for Alice to log on to the agent via, perhaps, an ssh or direct serial-line connection, and enter there her name and password. This works, but is awkward on a large network.

The usual way is to create an account for Alice by **cloning** some other account on the agent. We will assume that an initial account has been created as in *21.15.5 Passwords and Keys* for user "master". Cloning the account just requires specification of the names "master" and "alice"; no keys need be sent. The second step is to update the password for the new "alice" account, which is done via the key-change mechanism of the previous section. We look at the details of the cloning procedure in the next section.

21.15.9 VACM for SNMPv3

For SNMPv1 and SNMPv2c, VACM created associations between community strings and security groups, an then between security groups and permitted views of the OID tree. For SNMPv3, user names replace community strings.

There are three primary VACM tables, defined in **RFC 3415**. The first is `vacmSecurityToGroupTable`, indexed by the "security model" and the "security name". For SNMPv3 the model is 3 and the security name is the username; the SNMPv3 rows of the table are thus effectively indexed by the username. It is not unreasonable to have a separate security group for each user, *eg* "grppld" for user "pld".

The second table is `vacmAccessTable`, indexed by security group, security model (community or SNMPv3) and security level (*eg* `authNoPriv`). The table can be used to look up a named view for each of "read" and "write" privileges (also "trap" privileges, if desired). (This table is also indexed by "contextName", but often there is only one contextName, the default.)

Finally, the named views themselves are stored in `vacmViewTreeFamilyTable`, indexed by view name and an OID prefix representing an OID subtree. A given view *v* consists of all the OID prefixes *op* for which $\langle v, op \rangle$ appears in the table. Some prefixes may have an associated "mask", typically to allow access to a designated table *row*, and some prefixes may represent exclusion of that subtree from the view. Typical view names are "_all_", "_none_" and "systemonly"; we created a view "mib2+private" in *21.11 SNMPv1 communities and security*.

21.15.9.1 The `usmUserTable`

Now it is time to look at the actual table in which agents store user names and their associated authentication credentials, `usmUserTable`. It has the attributes below, which should all be prefixed by `usmUser`; the index attributes are the first two, `EngineID` and `Name`. We will consider only those attributes related to authentication-only security.

EngineID	usually the agent's own engineID
Name	*eg* "alice"
SecurityName	usually the same as Name
CloneFrom	an indication of the row this entry is cloned from
AuthProtocol	MD5 or SHA-1
AuthKeyChange	the ⟨random, delta⟩ keyChange object of *21.15.7 Key Changes*
OwnAuthKeyChange	the same but with different permissions; see below
PrivProtocol	used for encryption
PrivKeyChange	used for encryption
OwnPrivKeyChange	used for encryption
Public	used for confirming key changes; see below
StorageType	hopefully permanent, meaning values persist between reboots
Status	of type RowStatus, for row insertion

One can think of the table as also having an implicit authKey column, representing the local key corresponding to Name, that is never directly readable or writable. **RFC 3414** states flatly "the authKey is not accessible via SNMP." However, the agent must still keep the authKey somewhere, tied to the Name, so it can validate a given user based on the Name and authParameter supplied in a request.

Because EngineID is usually the agent's own EngineID, the table is *de facto* indexed just by Name.

Recall that, as was discussed in *21.13.4 SNMPv2 Indexes*, the index attributes, EngineID and Name, will not be directly accessible, but will be encoded in the OID associated with every other retrieved attribute. The username "alice" will thus be encoded, using the ASCII string encoding, as 97.108.105.99.101, not necessarily easily readable by human managers.

We are now in a position explain the cloning process and the key-change process as they play out with this table.

A specific semantic rule for this table is that the use of Set() to assign a ⟨random, delta⟩ keyChange object to AuthKeyChange or OwnAuthKeyChange causes that user's hidden authKey to be updated via the process of *21.15.7 Key Changes*. The user cannot confirm directly that this change succeeded, as a read of these keyChange attributes returns the empty string, so the usual recommended strategy is also to write a random value to the Public attribute; because Set() operations are atomic, a change to the latter means the former change succeeded as well.

Because it is possible for multiple managers to be updating the table simultaneously, a single TestAndIncr object named usmUserSpinLock may be used to enforce serialization, as in *21.13.5 TestAndIncr*. So the full recommended sequence for updating a key is as follows, where *keyChange* is the ⟨random, delta⟩ keyChange object and *index* encodes the EngineID and the Name:

> *rand = random value chosen by the manager*
>
> *val* := Get(usmUserSpinLock)
>
> Set((usmUserSpinLock, *val*), (AuthKeyChange.*index*, *keyChange*), (Public.*index*, *rand*))

This is repeated as necessary until Get(Public) returns *rand*, indicating that the Set() operations all succeeded.

The above applies for a key change being done by someone with write privileges to the AuthKeyChange column of usmUserTable.

An alternative to granting ordinary users write access to the `AuthKeyChange` column is to have them use the column `OwnAuthKeyChange` instead. Any user may attempt to write to this column, but the write will only succeed if the userName by which the request is authenticated is equal to the `Name` representing the index for this particular keyChange. In other words, anyone with write access to the `OwnAuthKeyChange` column can change his or her *own* key, and *only* his or her own key. Write access to this column must still be granted, however. As we will see below in *21.15.9.2.1 Cloning in Net-SNMP*, this security option is of relatively little practical significance.

To clone a new user from an existing one, the first step is to choose the row to be cloned, represented by the OID of an index value. Call this *cloneRow*, and let *index* again encode the `EngineID` and `Name`:

> *val* := Get(usmUserSpinLock)
>
> Set((`usmUserSpinLock`, *val*), (`CloneFrom`.*index*, *cloneRow*), (`Status`.*index*, `createAndWait`))

Typically the key change is then executed before changing the `Status` to `active`, though this is not required.

21.15.9.2 Creating Accounts in Net-SNMP

To create an initial account, the configuration files are used. On the author's installation of Net-SNMP version 5.7.x (in 2016), the administratively written configuration file is /etc/snmp/snmpd.conf and the system-written configuration file is /var/lib/snmp/snmpd.conf; the latter contains `snmpEngineBoots` and other persistent SNMP data. The following line goes in this second file, with SNMP shut down (some earlier versions of Net-SNMP required that it be placed in /var/net-snmp/snmpd.conf). Its effect is to create an SNMPv3 user named "master" with MD5-authentication password "saskatchewan".

```
createUser master MD5 saskatchewan
```

When SNMP is then started, the line above is replaced by something like the following (on a single line) in /var/lib/snmp/snmpd.conf:

```
usmUser 1 3 0x80001f8880889cb038b1aca650 "master" "master" NULL
\
    .1.3.6.1.6.3.10.1.1.2 0x7293f49a82fc950f5c344efd94dbb7db
    .1.3.6.1.6.3.10.1.2.1 0x 0x
```

The new localized key is 0x7293f49a82fc950f5c344efd94dbb7db; the first hex string beginning 0x80001 is the engineID.

For this new account to be authorized to do anything, we must also add the following permissions entry to /etc/snmp/snmpd.conf:

```
rwuser master
```

The effect of this entry is to create an entry (on SNMP restart) in `vacmSecurityToGroupTable` associating user "master" with its own security group (Net-SNMP names it "grpmaster"), and then an entry in the `vacmAccessTable` granting this new group SNMPv3 read and write access to the entire OID tree. (In general we could also have used "rouser", except that we will need "master" to be able to create new users.)

After permissions (at least read permissions) are enabled, the following snmpget should work

```
snmpget -v 3 -u master -l authNoPriv -a MD5 -A saskatchewan
localhost 1.3.6.1.2.1.1.4.0
```

The hostname is "localhost" if this is being run on the same machine that the Net-SNMP agent is running on; it can also of course be run remotely.

We can make this a little shorter by editing the Net-SNMP per-user manager configuration file $HOME/.snmp/snmp.conf to add the following lines:

```
defSecurityName master
defAuthType MD5
defSecurityLevel authNoPriv
defAuthPassphrase saskatchewan
```

Now the snmpget command can be shortened to

```
snmpget -v 3 localhost 1.3.6.1.2.1.1.4.0
```

If we take the password, "saskatchewan", and repeat it to 2^{20} bytes, the MD5 checksum is 0x3da9dbfc3a78acb675e436746e1f4f8a; this is the "digest" of *21.15.5 Passwords and Keys*. From the /var/lib/snmp/snmpd.conf file (or from the created usmUser entry above) we find the engineID is 0x80001f8880889cb038b1aca650. If we convert these two strings to binary data, concatenate them as digest⌒engineID⌒digest, and take the MD5 checksum, we indeed get

7293f49a82fc950f5c344efd94dbb7db

which is exactly the key entry in the /var/lib/snmp/snmpd.conf file.

21.15.9.2.1 Cloning in Net-SNMP

We can now **clone** this account while SNMP is running, using the `snmpusm` command-line utility. The following line creates an account "pld" cloned from "master", using the master account (and assuming that master is an rwuser and not an rouser); we have abbreviated by *auth* the full credentials `-u master -l authNoPriv -a MD5 -A saskatchewan`. We have switched from "localhost" to *HOST* to emphasize that this can be run remotely.

`snmpusm -v3` *auth HOST* `create pld master`

We then change the password to "ramblers", *still using the authority of user "master"*; the `-Ca` option tells snmpusm to change only the authentication key.

`snmpusm -v3` *auth* `-Ca` *HOST* `passwd saskatchewan ramblers pld`

But before new user pld can do anything, we must grant access. We *can* again edit /etc/snmp/snmpd.conf on the machine running the Net-SNMP agent, adding the directive `rouser pld` (granting read-only access this time) and then restarting SNMP. We can also, however, manipulate the VACM tables of *21.15.9 VACM for SNMPv3* using the Net-SNMP `snmpvacm` command, which works remotely:

`snmpvacm -v3` *auth HOST* `createSec2Group 3 pld grppld`

`snmpvacm -v3` *auth HOST* `createAccess grppld 3 2 0 _all_ _none_ _none_`

These two commands create entries in the `vacmSecurityToGroupTable` and the `vacmAccessTable` respectively. The parameters following `grppld` in the second command include the USM security model (3), the authNoPriv security level (2), a placeholder (0) for the `contextmatch` flag which we are not using, and the views `_all_` and `_none_` predefined by Net-SNMP. The command here gives `grppld` read access to everything, and write access to nothing (the second `_none_` denies access for creation of notifications such as traps).

After all this, user pld can then issue an snmpget using pld's own credentials with

> `snmpget -v 3 -u pld -l authNoPriv -a MD5 -A ramblers` *HOST* `1.3.6.1.2.1.1.4.0`

If user pld is to be able to change pld's password (key), write-access must be granted at least to the `usmUserAuthKeyChange` column of `usmUserTable`. We do this by adding the appropriate OID to the write view for user pld. Views are maintained in the `vacmViewTreeFamilyTable` as an association between the view name and a list of OIDs.

The first step is to add the `usmUserAuthKeyChange` column, 1.3.6.1.6.3.15.1.2.2.1.6, to user pld's view. The command below can be applied alone, so that the `usmUserAuthKeyChange` column is the *only* object to which user pld can write, or as part of a series of `createView` statements adding a series of OID subtrees to `pldwriteview`. (An OID can be removed from a view with the `deleteView` option.) In the following commands we are still using the initial `master` account for *auth*.

> `snmpvacm -v3` *auth HOST* `createView pldwriteview 1.3.6.1.6.3.15.1.2.2.1.6`

Now we apply this new view, `pldwriteview`, to pld's group, grppld. We must first clear the previously granted access.

> `snmpvacm -v3` *auth HOST* `deleteAccess grppld 3 2`
>
> `snmpvacm -v3` *auth HOST* `createAccess grppld 3 2 0 _all_ pldwriteview _none_`

The password can now be changed from "ramblers" to "ignatius" as follows:

> `snmpusm -v3 -u pld -l authNoPriv -a MD5 -A ramblers -Ca` *HOST* `passwd ramblers ignatius pld`

We granted pld access here to the entire `usmUserAuthKeyChange` column. In principle this might be risky, as it allows user pld to write to the key-changing column for *any* user. One potential approach here is to grant user pld access only to pld's own row in `usmUserAuthKeyChange`; this has the following rather cumbersome OID (in which the last four levels 3.112.108.100 spell out the three-byte string "pld" in ASCII):

```
1.3.6.1.6.3.15.1.2.2.1.6.17.128.0.31.136.128.22.41.105.105.47.232.188.86.0.0.0.0.3.112.108
```

Another approach might be to grant pld write access to the `usmUserOwnAuthKeyChange` column. The semantics of this column, outlined above in *21.15.9.1 The usmUserTable*, are such that key-change requests are accepted only for the user who signed the request. This would prevent user pld from even attempting to change anyone else's password.

But the risk of full `usmUserAuthKeyChange` access is minimal: a request authenticated by user pld can only change the password of another user, say bob, *if bob's previous password is known to pld*, as keychange objects must incorporate the old key/password. But if that password is known, then the password can just

as easily be changed by authenticating the request as user bob rather than as user pld. In any event, the Net-SNMP `snmpusm` command does not support writing to `usmUserOwnAuthKeyChange`.

The command `snmpusm` does not support making general modifications; if a USM entry here is made incorrectly, it may be necessary to delete and re-create it.

21.16 Exercises

1. Consider the table below. The first column is the index.

index	count	veggie
1	401	kale
3	523	kohlrabi
57	607	mâche
92	727	okra

Give the OID for each data value, assuming this table were encoded in SNMP. Assume the columns are assigned OID levels 1, 2, and 3 in order (including the index column), and the root of the subtree (the `tableEntry` OID) is represented by **T**. Note that rows are not numbered consecutively.

2. Recall that `GetBulk()` acts like a repeated `GetNext()`. Why is there no `GetBulk()` equivalent of `Get()`?

3. What happens if we have a three-row, three-column table and ask, using `GetBulk()`, for the first two columns with a repetition count of four? What is retrieved? Assume entries are of the form **T**.col.row, for col and row each ranging from 1 to 3.

4. Consider the following multi-attribute `GetNext()` as presented in *21.8.1 Multi-attribute Get()*:

> GetNext(**T**.1.4, **T**.2.4, **T**.3.4, **T**.4.4, **T**.10.4, **T**.11.4)

The answer given in the text explicitly assumed that the only columns existing were those shown: 1, 2, 3, 4, 10 and 11. What would be the result if all columns of `ifTable` were present? Assume, as before, that row 4 is the final row.

5. Suppose you want an SNMP table `identTable` to hold ⟨idNum,userName⟩ pairs, where idnum is to be an INTEGER and username an OCTET STRING. The INDEX is idnum. Give ASN.1 definitions for the following:

- `identTable`
- `identEntry`
- `IdentEntry`
- `idNum`
- `userName`

Follow the SNMPv2 convention of *not* including the index column in the actual table data, *21.13.4 SNMPv2 Indexes*.

6. (a) What can an SNMP agent do to detect that a manager has created a row in state `underCreation` and then crashed, leaving the row abandoned?

(b). What can an SNMP agent do to detect that a manager has created a valid row, and has later crashed?

7. In the RMON hostTopN table (*21.14.1.4 Host Top N*), the agent does all the report-building work. Would it be possible for the manager to do this? If not, why not? If so, why do you think RMON provides this table?

8. In the RMON matrixSDTable (*21.14.1.5 Matrix*), the table constructed collects data from only a single interface on the agent (that interface specified by DataSource). Could we get additional Matrix information by considering other interfaces? Why or why not?

9. The RMON matrix group (*21.14.1.5 Matrix*) provides two tables with the same information, matrixSDTable and matrixDSTable.

(a). Suppose we want all the table data about a given destination D, and have only the matrixSDTable. Explain why every row of matrixSDTable would need to be examined.

(b). Now suppose we repeat the investigation of (a), but this time the manager has previously downloaded the complete list of all hosts on the LAN by using the RMON hosts tables. If N is the number of hosts on the LAN, explain how to find all hosts that communicated with D using only N retrieval requests.

(c). Explain how to find all hosts S that sent packets to D even more quickly using the matrixDSTable. If M⩽N is the number of such S, your answer should involve N+1 retrieval requests.

10. In the keychange operation of *21.15.7 Key Changes*, suppose the manager simply transmitted delta2 = oldkey XOR newkey to the agent.

(a). Suppose an eavesdropper discovers delta2 and also knows a few bits of oldkey. What can the eavesdropper learn about newkey? Would the same vulnerability apply to the mechanism of *21.15.7 Key Changes*?

(b). Suppose an eavesdropper later discovers newkey. Explain how to recover oldkey, and why this does not work when the mechanism of *21.15.7 Key Changes* is used.

11. List the OID prefixes for which a manager would need to be granted write permission if the manager were to be able to modify all settings in .1.3.6.1.2.1 and .1.3.6.1.4.1, and change their own local key, but *not* have access to columns in usmUserTable that would allow modification of other manager accounts. (The VACM table has an "exception" option to make this easier).

12. Suppose a manager has write permission only for the usmUserOwnAuthKeyChange column in usmUserTable, which allows change of only that manager's password. However, the manager has full write access to the VACM tables. Explain how the manager can modify the local keys of other managers.

13. Use Wireshark to monitor localhost traffic while you use the Net-SNMP snmpusm command to change a manager's own local key. Does the command use the usmUserAuthKeyChange column or the usmUserOwnAuthKeyChange column?

How do we keep intruders out of our computers? How do we keep them from impersonating us, or from listening in to our conversations or downloading our data? Computer security problems are in the news on almost a daily basis. In this chapter we take a look at just a few of the issues involved in building secure networks.

For our limited overview here, we will divide attacks into three categories:

1. Attacks that execute the intruder's code on the target computer
2. Attacks that extract data from the target, without code injection
3. Eavesdropping on or interfering with computer-to-computer communications

The first category is arguably the most serious; this usually amounts to a complete takeover, though occasionally the attacker's code is limited by operating-system privileges. A computer taken over this way is sometimes said to have been "owned". We discuss these attacks below in *22.1 Code-Execution Intrusion*. Perhaps the simplest form of such an attack is through stolen or guessed passwords to a system that offers remote login to command-shell accounts. More technical forms of attack may involve a virus, a buffer overflow (*22.2 Stack Buffer Overflow* and *22.3 Heap Buffer Overflow*), a protocol flaw (*22.1.2 Christmas Day Attack*), or some other software flaw (*22.1.1 The Morris Worm*).

In the second category are intrusions that do not involve remote code execution; a server application may be manipulated to give up data in ways that its designers did not foresee.

For example, in 2008 David Kernell gained access to the Yahoo email account of then-vice-presidential candidate Sarah Palin, by guessing or looking up the answers to the forgotten-password security questions for the account. One question was Palin's birthdate. Another was "where did you meet your spouse?", which, after some research and trial-and-error, Kernell was able to guess was "Wasilla High"; Palin grew up in and was at one point mayor of Wasilla, Alaska. Much has been made since of the idea that the answers to many security questions can be found on social-networking sites.

As a second example in this category, in 2010 Andrew "weev" Auernheimer and Daniel Spitler were charged in the "AT&T iPad hack". IPad owners who signed up for network service with AT&T had their iPad's ICC-ID recorded along with their other information. If one of these owners later revisited the AT&T website, the site would automatically request the iPad's ICC-ID and then populate the web form with the user's information. If a randomly selected ICC-ID were presented to the AT&T site that happened to match a real account, that user's name, phone number and email address would be returned. ICC-ID strings contain 20 decimal digits, but the individual-device portion of the identifier is much smaller and this brute-force attack yielded 114,000 accounts.

This attack is somewhat like a password intrusion, except that there was no support for running commands via the "compromised" accounts.

Auernheimer was convicted for this "intrusion" in November 2012, but his sentence was set aside on appeal in April 2014. Auernheimer's conviction remains controversial as the AT&T site never requested a password in the usual sense, though the site certainly released information not intended by its designers.

Finally, the third category here includes any form of eavesdropping. If the password for a login-shell account is obtained this way, a first-category attack may follow. The usual approach to prevention is the use of

encryption. Encryption is closely related to **secure authentication**; encryption and authentication are addressed below in *22.6 Secure Hashes* through *22.10 SSH and TLS*.

Encryption does not always work as desired. In 2006 intruders made off with 40 million credit-card records from **TJX Corp** by breaking the WEP Wi-Fi encryption (*22.7.7 Wi-Fi WEP Encryption Failure*) used by the company, and thus gaining access to account credentials and to file servers. Albert Gonzalez pleaded guilty to this intrusion in 2009. This was the largest retail credit-card breach until the Target hack of late 2013.

22.1 Code-Execution Intrusion

The most serious intrusions are usually those in which a vulnerability allows the attacker to run executable code on the target system.

The classic **computer virus** is broadly of this form, though usually without a network vulnerability: the user is tricked – often involving some form of social engineering – into running the attacker's program on the target computer; the program then makes itself at home more or less permanently. In one form of this attack, the user receives a file `interesting_picture.jpg.exe` or `IRS_deficiency_notice.pdf.exe`. The attack is made slightly easier by the default setting in Windows of not displaying the final file extension `.exe`.

Early viruses had to be completely self-contained, but, for networked targets, once an attacker is able to run some small initial executable then that program can in turn download additional malware. The target can also be further controlled via the network.

The reach of an executable-code intrusion may be limited by privileges on the target operating system; if I am operating a browser on my computer as user "pld" and an intruder takes advantage of a flaw in that browser, then the intruder's code will also run as "pld" and not as "root" or "Administrator". This may prevent the intruder from rewriting my kernel, though that is small comfort to me if my files are encrypted and held for ransom.

On servers, it is standard practice to run network services with the minimum privileges practical, though see *22.2.3 Defenses Against Buffer Overflows*.

Exactly what is "executable code" is surprisingly hard to state. Scripting languages usually qualify. In 2000, the ILOVEYOU virus began spreading on Windows systems; users received a file `LOVE-LETTER.TXT.vbs` (often with an enticing Subject: line such as "love letter for you"). The `.vbs` extension, again not displayed by default, meant that when the file was opened it was automatically run as a **v**isual **b**asic **s**cript. The ILOVEYOU virus was later attributed to Reonel Ramones and Onel de Guzman of the Philippines, though they were never prosecuted. The year before, the Melissa virus spread as an emailed Microsoft Word attachment; the executable component was a Word macro.

Under Windows, a number of configuration-file formats are effectively executable; among these are the program-information-file format .PIF and the screen-saver format .SCR.

22.1.1 The Morris Worm

The classic Morris Worm was launched on the infant Internet in 1988 by Robert Tappan Morris. Once one machine was infected, it would launch attacks against other machines, either on the same LAN or far away.

The worm used a number of techniques, including taking advantage of **implementation flaws** via stack buffer overflows (*22.2 Stack Buffer Overflow*). Two of the worm's techniques, however, had nothing to do with code injection. One worm module contained a dictionary of popular passwords that were used to try against various likely system accounts. Another module relied on a different kind of implementation vulnerability: a (broken) diagnostic feature of the `sendmail` email server. Someone could connect to the `sendmail` TCP port 25 and send the command `WIZ <password>`; that person would then get a shell and be able to execute arbitrary commands. It was the intent to require a legitimate `sendmail`-specific password, but an error in `sendmail`'s frozen-configuration-file processing meant that an empty password often worked.

22.1.2 Christmas Day Attack

The 1994 "Christmas day attack" (*12.10.1 ISNs and spoofing*) used a TCP **protocol weakness** combined with a common computer-trust arrangement to gain privileged login access to several computers at UCSD. Implementations can be fixed immediately, once the problem is understood, but protocol changes require considerable negotiation and review.

The so-called "rlogin" trust arrangement meant that computer A might be configured to trust requests for remote-command execution from computer B, often on the same subnet. But the ISN-spoofing attack meant that an attacker M could send a command request to A that would *appear* to come from the trusted peer B, at least until it was too late. The command might be as simple as "open up a shell connection to M". At some point the spoofed connection would fail, but by then the harmful command would have been executed. The only fix is to stop using rlogin. (Ironically, the ISN spoofing attack was discovered by Morris but was not used in the Morris worm above; see *[RTM85]*.)

Note that, as with the sendmail WIZ attack of the Morris worm, this attack did not involve network delivery of an executable fragment (a "shellcode").

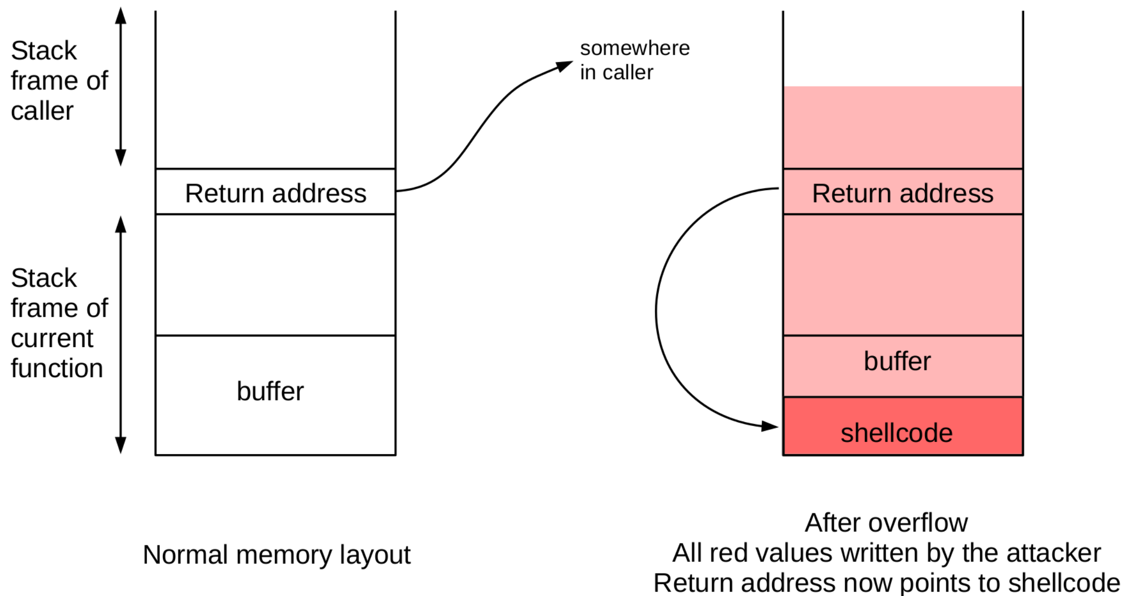
22.2 Stack Buffer Overflow

The stack buffer overflow is perhaps the classic way for an attacker to execute a short piece of machine code on a remote machine, thus compromising it. Such attacks are always due to an implementation flaw. A server application reads attacker-supplied data into a buffer, `buf`, of length `buflen`. Due to the flaw, however, the server reads more than `buflen` bytes of data, and the additional data is written into memory past the end of `buf`, corrupting memory. In the C language, there is no bounds checking with native arrays, and so such an overflow is not detected at the time it occurs.

In most memory layouts, the stack grows downwards; that is, a function call creates a new stack frame with a numerically lower address. Array indexing, however, grows upwards: `buf[i+1]` is at a higher address than `buf[i]`. As a consequence, overwriting the buffer allows rewriting the most recent return address on the stack. A common goal for the attacker is to supply an overflowing buffer that does two things:

1. it includes a **shellcode** - a small snippet of machine code that, when executed, does something bad (traditionally but not necessarily by starting a shell with which the attacker can invoke arbitrary commands).
2. it overwrites the stack return address so that, when the current function exits, control is returned not to the caller but to the supplied shellcode.

In the diagram below, the left side shows the normal layout: the current stack frame has a buffer into which the attacker's message is read. When the current function exits, control is returned to the address stored in `return_address`.



The right side shows the result after the attacker has written shellcode to the buffer, and, by overflow, has also overwritten the `return_address` field on the stack so that it now points to the shellcode. When the function exits, control will be passed to the code at `return_address`, that is, to the shellcode rather than to the original caller. The shellcode is here shown below `return_address` in memory, but it can also be above it.

22.2.1 Return to libc

A variant attack is for the attacker to skip the shellcode, but instead to overwrite the stack return address with the address of a known library procedure. The libc library is popular here, making this known as the **return-to-libc** approach. The goal of the attacker is to identify a library procedure that, in the current context of the server process being attacked, will do something that the attacker finds useful.

One version of this attack is to overwrite the stack address with the address of the `system()` call, and to place on the stack just above this return address a pointer to the string "/bin/sh" (often present in the environment strings of the attacked process). When the current function exits, control branches to `system()`, which now thinks it has been called with parameter `/bin/sh`. A shell is launched.

Return-to-libc attacks often involve no shellcode at all. The attack of *22.3.2 A JPEG heap vulnerability* uses some return-to-libc elements but also does involve injected shellcode.

A practical problem with any form of the stack-overflow attack is knowing enough about the memory layout of the target machine so that the attacker can determine exactly where the shellcode is loaded. This is made simpler by standardized versions of Windows in which many library routines are at fixed, well-known addresses.

22.2.2 An Actual Stack-Overflow Example

Why the code here?

In many accounts of computer vulnerabilities, there is an understandable reluctance to explain the actual mechanics, lest some attacker learn how to exploit the flaw. This secrecy, however, sometimes has the unfortunate side-effect of making vulnerabilities seem almost magical. The goal in presenting this twenty-year-old example in detail is simply to strip away the aura of mystery surrounding many exploits.

To put together an actual example, we modify a version of TCP simplex-talk (*12.6 TCP simplex-talk*) written in the C language. On the server side, all we have to do is read the input with the infamous `gets(char * buf)`, which reads in data up to a newline character into the array `buf`, with no size restrictions. To do this, we must arrange for the standard-input stream of the reading process to be the network connection; the version of `gets()` that reads from an arbitrary input stream is `fgets(char * buf, int bufsize, FILE *stream)`; this is much safer as it will not read more than `bufsize` characters into `buf`, though it is still up to the programmer to supply the correct value of `bufsize`.

Alternatively, we can also have the server its the data with the call

```
recv(int socket, char * buf, int bufsize, int flags)
```

but supply an *incorrect* (and much too large) value for the parameter `bufsize`. This approach has the practical advantage of allowing the attacker to supply a buffer with NUL characters (zero bytes) and with additional embedded newline characters.

On the client side – the attacker's side – we need to come up with a suitable shellcode and create a too-large buffer containing, in the correct place, the address of the start of the shellcode. Guessing this address used to be easy, in the days when every process always started with the same virtual-memory address. It is now much harder precisely to make this kind of buffer-overflow attack more difficult; we will cheat and have our server print out an address on startup that we can then supply to the client.

An attack like this depends on knowing the target operating system, the target cpu architecture (so as to provide an executable shellcode), the target memory layout, and something about the target server implementation (so the attacker knows what overflow to present, and when). Alas, all but the last of these are readily guessable. Once upon a time vulnerabilities in server implementations were discovered by reading source code, but it has long been possible to search for overflow opportunities making use only of the binary executable code.

22.2.2.1 The server

The overflow-vulnerable server, oserver.c, is more-or-less a C implementation of the tcp simplex-talk server of *12.6 TCP simplex-talk*; note the explicit call to `bind()` which was handled by the `ServerSocket()` constructor in the Java version. For each new connection, a new process to handle it is created with `fork()`; that new process then calls `process_connection()`.

The `process_connection()` function then reads a line at a time from the client into a buffer `pcbuf` of 80 bytes. Unfortunately for the server, it may read well more than 80 bytes into `pcbuf`.

For the stack overflow to work, it is essential that the function that read in the oversized buffer – thus corrupting the stack – must return. Therefore the protocol has been modified so that process_connection() returns if the arriving string begins with “quit”.

We must also be careful that the stack overflow does not so badly corrupt any local variables that the code fails afterwards to continue running, even before returning. All local variables in process_connection() are overwritten by the overflow, so we save the socket itself in the global variable gsock.

We also call setstdinout(gsock) so that the standard input and standard output within process_connection() is the network connection. This allows the use of gets(), which reads from the standard input (alternatively, recv() with an incorrect value for bufsize may be used). It also means that the shell launched by the shellcode will have its standard input and output correctly set up; we could, of course, make the appropriate dup()/fcntl() calls from the shellcode, but that would increase its complexity and size.

Because the standard output is now the TCP connection, the server prints incoming strings to the terminal via the standard-error stream.

On startup, the server prints an address (that of mbuf[]) within its stack frame; we will refer to this as **mbuf_addr**. The attacking client must know this value. No real server is so accommodating as to print its internal addresses, but in the days before address randomization, *22.2.3.2 ASLR*, the server’s stack address was typically easy to guess.

Whenever a connection is made, the server here also prints out the distance, in bytes, between the start of mbuf[] in main() and the start of pcbuf – the buffer that receives the overflow – in process_connection(). This latter number, which we will call **addr_diff**, is constant, and must be compiled into the exploit program (it does change if new variables are added to the server’s main() or process_connection()). The actual address of pcbuf[] is thus mbuf_addr – addr_diff. This will be the address where the shellcode is located, and so is the address with which we want to overwrite the stack. We return to this below in *22.2.2.3 The exploit*, where we introduce a “NOPslide” so that the attacker does not have to get the address of pcbuf[] exactly right.

Linux provides some protection against overflow attacks (*22.2.3 Defenses Against Buffer Overflows*), so the server must disable these. As mentioned above, one protection, address randomization, we defeat by having the server print a stack address. The server must also be compiled with the -fno-stack-protector option to disable the stack canary of *22.2.3.1 Stack canary*, and the -z execstack option to disable making the stack non-executable, *22.2.3.3 Making the stack non-executable*.

```
gcc -fno-stack-protector -z execstack -o oserver oserver.c
```

Even then we are dutifully warned by both the compiler and the linker:

```
warning: 'gets' is deprecated ....
warning: the 'gets' function is dangerous and should not be used.
```

In other words, getting this vulnerability still to work in 2014 takes a bit of effort.

The server here does work with the simplex-talk client of *12.6 TCP simplex-talk*, but with the #USE_GETS option enabled it does not handle a client exit gracefully.

22.2.2.2 The shellcode

The shellcode must be matched to the operating system and to the cpu architecture; we will assume linux running on a 32-bit Intel x86. Our goal is to create a shellcode that launches `/bin/sh`. The approach here is taken from *[SH04]*.

The shellcode must be a string of machine instructions that is completely self-contained; data references cannot depend on the linker for address resolution. Not only must the code be position-independent, but the code together with its data must be position-independent.

So-called "Intel syntax" is used here, in which the destination operand comes first, *eg* `mov eax,37`. The nasm assembler is one of many that supports this format.

System calls in linux are easily made using interrupts, using the `int 0x80` opcode and parameter. The x86 architecture has four general-purpose registers `eax`, `ebx`, `ecx` and `edx`; the first of these holds a code for the particular system routine to be invoked and the others hold up to three parameters. (Below, in *22.3.2 A JPEG heap vulnerability*, we will also make use of register `edi`.) The system call we want to make is

```
execve(char * filename, char *argv[], char *envp[])
```

We need `eax` to contain the numeric value 11, the 32-bit-linux syscall value corresponding to `execve` (perhaps defined in `/usr/include/i386-linux-gnu/asm/unistd_32.h`). (64-bit linux uses 59 as the syscall value for `execve`.) We load this with `mov al 11`; `al` is a shorthand for the low-order byte of register `eax`. We first zero `eax` by subtracting it from itself. We can also, of course, use `mov eax 11`, but then the 11 expands into four bytes 0x0b000000, and we want to avoid including NUL bytes in the code.

We also need `ebx` to point to the NUL-terminated string `/bin/sh`. The register `ecx` should point to an array of pointers ["/bin/sh", 0] in memory (the null-terminated `argv[]`), and `edx` should point to a null word in memory (the null-terminated `envp[]`). We include in the shellcode the string "/bin/shNAAAABBBB", and have the shellcode insert a NUL byte to replace the N and a zero word to replace the "BBBB", as the shellcode must contain no NUL bytes. The shellcode will also replace the "AAAA" with the address of "/bin/sh". We then load `ecx` with the address of "AAAA" (now containing the address of "/bin/sh" followed by a zero word) and `edx` with the address of "BBBB" (now just a zero word).

Loading the address of a string is tricky in the x86 architecture. We want to calculate this address relative to the current instruction pointer IP, but no direct access is provided to IP. The workaround in the code below is to jump to `shellstring` near the end, but then invoke `call start`, where `start:` is the label for our main code up at the top. The action of `call start` is to push the address of the byte following `call start` onto the stack; this happens to be the address of `shellstring:`. Back up at `start:`, the `pop ebx` pops this address off the stack and leaves it in `ebx`, where we want it.

Our complete shellcode is as follows (the actual code is in shellcode.s):

```
jmp short shellstring
start:
pop ebx                 ;get the address of the string in ebx
sub eax, eax            ;zero eax by subtracting it from itself
mov [ebx+7 ], al        ;put a NUL byte where the N is in the string
mov [ebx+8 ], ebx       ;put the address of the string where the AAAA is
mov [ebx+12], eax       ;put 4 null bytes into where the BBBB is
mov al, 11              ;execve is syscall 11
lea ecx, [ebx+8]        ;load the address of where the AAAA was
lea edx, [ebx+12]       ;load the address of the NULLS;
```

```
sub ecx,ecx
sub edx,edx
int 0x80                ;call the kernel, WE HAVE A SHELL!
shellstring:
call start              ;pushes address of string below and jumps to start:
db '/bin/shNAAAABBBB'   ;the string. AAAA and BBBB get filled in as above
```

We run this through the commands

```
nasm -f elf shellcode.s
ld -o shellcode shellcode.o
objdump -d shellcode
```

and come up with the following string:

```
char shellcode[] =
       "\xeb\x1a\x5b\x29\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c"
       "\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\x29\xc9\x29\xd2\xcd\x80"
       "\xe8\xe1\xff\xff\xff/bin/sh/NAAAABBBB";
```

We can test this with a simple C program defining the above and including

```
int main(int argc, char **argv) {
    int (*func)();
    func = (int (*)()) shellcode;
    (int)(*func)();
}
```

We can verify that the resulting shell has not inherited the parent environment with the `env` and `set` commands.

Additional shellcode examples can be found in *[AHLR07]*.

22.2.2.3 The exploit

Now we can assemble the actual attack. We start with a C implementation of the simplex-talk client, and add a feature so that when the input string is "doit", the client

- sends the oversized buffer containing the shellcode, terminated with a newline to make `gets()` happy
- sends "quit", terminated with a newline, to force the server's `process_connection()` to return, and thus to transfer control to the code pointed to by the now-overwritten `return_address` field of the stack
- begin a loop – `copylines()` – to copy the local terminal's standard input to the network connection (hopefully now with a shell at the other end), and to copy the network connection to the local terminal's standard output

On startup, the client accepts a command-line parameter representing the address (in hex) of a variable close to the start of the server's `main()` stack frame. When the server starts, it prints this address out; we simply copy that when starting the client.

The full client code is in netsploit.c.

All that remains is to describe the layout of the malicious oversized buffer, created by `buildbadbuf()`. We first compute our guess for the address of the start of the vulnerable buffer `pcbuf` in the server's `process_connection()` function: we take the address passed in on the command line, which is actually the address of `mbuf` in the server's `main()`, and add to it the known constant (`pcbuf` - `mbuf`). This latter value, 147 in the version tested by the author, is stored in netsploit.c's `BUF_OFFSET`.

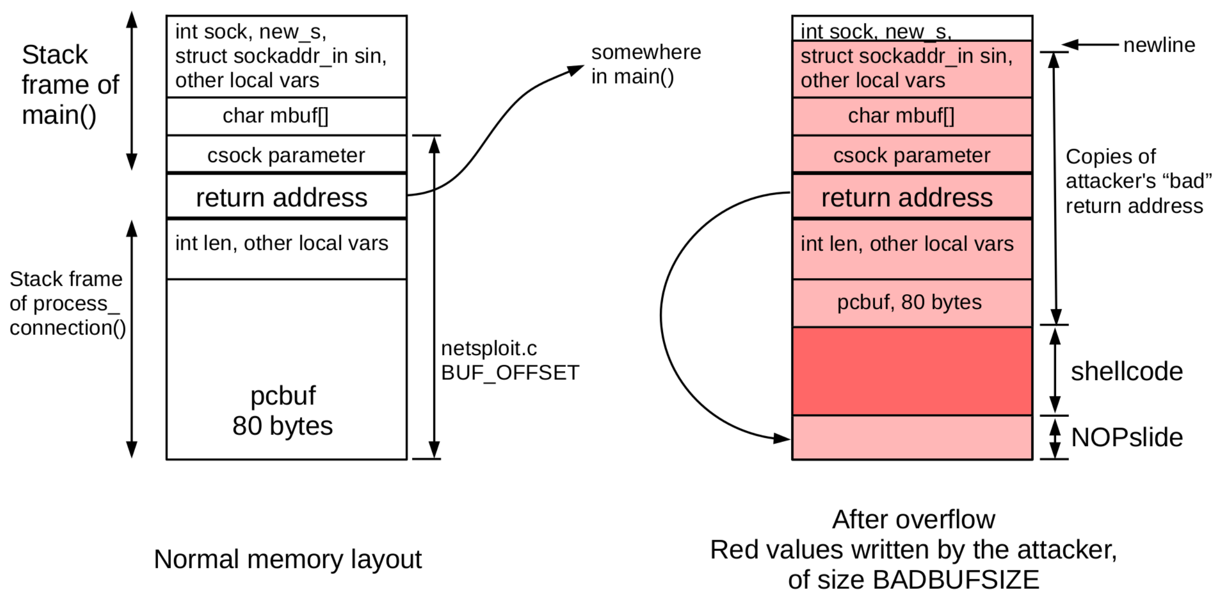
This calculation of the address of the server's `pcbuf` *should* be spot-on; if we now store our shellcode at the start of `pcbuf` and arrange for the server to jump to our calculated address, the shellcode should run. Real life, however, is not always so accommodating, so we introduce a **NOP slide**: we precede our shellcode with a run of NOP instructions. A jump anywhere into the NOPslide should lead right into the shellcode. In our example here, we make the NOPslide 20 bytes long, and add a fudge factor of between 0 and 20 to our calculated address (`FUDGE` is 10 in the actual code).

We need the attack buffer to be large enough that it overwrites the stack return-address field, but small enough that the server does not incur a segmentation fault when overfilling its buffer. We settle on a `BADBUFSIZE` of 161 (160 plus 1 byte for a final newline character); it should be comparable to but perhaps slightly larger than the `BUF_OFFSET` value of, in our case, 147).

The attack buffer is now

- 20 bytes of NOPs
- 50 bytes of shellcode
- 90 bytes worth of the repeated calculated address (`baseaddr-BUF_OFFSET+FUDGE` in the code
- 1 byte newline, as the server's `gets()` expects a newline-terminated string

Here is a diagram like the one above, but labeled for this particular attack. Not all memory regions are drawn to scale, and there are more addresses between the two stack frames than just `return address`.



We double-check the bad buffer to make sure it contains no NUL bytes and no other newline characters.

If we wanted a longer NOPslide, we would have to hope there was room *above* the stack's return-address

field. In that case, the attack buffer would consist of a bunch of repeated address guesses, then the NOPslide, and finally the shellcode.

After the command "doit", the netsploit client prompt changes to `1>`. We can then type `ls`, and, if the shellcode has successfully started, we get back a list of files on the server. As earlier, we can also type `env` and `set` to verify that the shell did not inherit its environment from any "normal" shell.

22.2.3 Defenses Against Buffer Overflows

How to prevent this sort of attack? The most basic approach is to make sure that **array bounds** are never violated (and also that similar rules for the use of dynamically allocated memory, such as not using a block after it has been freed, are never violated). Some languages enforce this automatically through "memory-safe" semantics; while this is not a guarantee that programs are safe, it does eliminate an important class of vulnerabilities. In C, memory- and overflow-related bugs can be eliminated through careful programming, but the task is notoriously error-prone.

Another basic approach, applicable to almost all remote-code-execution attacks, is to make sure that the server runs with the minimum **permissions** possible. The server may not have write permission to anything of substance, and may in fact be run in a so-called "chroot" environment in which any access to the bulk of the server's filesystem is disabled.

One issue with this approach is that a server process on a unix-derived system that wants to listen on a port less than 1024 needs special privileges to open that port. Traditionally, the process would start with root privileges and, once the socket was opened, would downgrade its privileges with calls to `setgid()` and `setuid()`. This is safe in principle, but requires careful attention to the `man` pages; use of `seteuid()`, for example, allows the shellcode to recover the original privileges. Linux systems now support assigning to an unprivileged process any of several "capabilities" (see `man capabilities`). The one most relevant here is `CAP_NET_BIND_SERVICE`, which, if set, allows a process to open privileged ports. Still, to assign these capabilities still requires some privileged intervention when the process is started.

22.2.3.1 Stack canary

The underlying system can also provide some defenses. One of these, typically implemented within the compiler, is the **stack canary**: an additional word on the stack near the return address that is set to a pseudo-random value. A copy of this word is saved elsewhere. When the function calls `return` (either explicitly or implicitly), this word is checked to make sure the stack copy still agrees with the saved-elsewhere copy; a discrepancy indicates that the stack was overwritten.

In the `gcc` compiler, a stack canary can be enabled with the compiler option `-fstack-protector`. To compile the stack exploit in *22.2.2.3 The exploit*, we needed to add `-fno-stack-protector`.

22.2.3.2 ASLR

Another operating-system-based defense is **Address-Space Layout Randomization**, or ASLR. In its simplest form, this means that each time the server process is restarted, the stack will have a different starting address. For example, restarting the oserver program above five times yields addresses (in hex) of bf8d22b7, bf84ed87, bf977d87, bfcc5cb7 and bfa302d7. There are at least four hex digits (16 bits) of entropy here; if the server did not print its stack address it might take 2^{16} guesses for the attacker to succeed.

Still, 2^{16} guesses might take an attacker well under an hour. Worse, the attacker might simply create a stack-buffer-overflow attack with a very long NOPslide; the longer the NOPslide the more room for error in guessing the shellcode address. With a NOPslide of length 2^{10} = 1024 bits, guessing the correct stack address will take only 2^6 = 64 tries. (Some implementations squeeze out 19 bits of address-space entropy, meaning that guessing the correct address increases to 2^9 = 512 tries.)

For 64-bit systems, however, ASLR is much more effective. Brute-force guessing of the stack address takes a prohibitively long time.

ALSR also changes the heap layout and the location of libraries each time the server process is restarted. This is to prevent return-to-libc attacks, *22.2.1 Return to libc*. For a concrete example of an attacker's use of non-randomized library addresses, see *22.3.2 A JPEG heap vulnerability*.

On linux systems, ASLR can be disabled by writing a 0 to /proc/sys/kernel/randomize_va_space; values 1 and 2 correspond to partial and full randomization.

Windows systems since Vista (2007) have had ASLR support, though earlier versions of the linker required the developer to request ASLR with the `/DYNAMICBASE` switch.

22.2.3.3 Making the stack non-executable

A more sophisticated idea, if the virtual-memory hardware supports it, is to mark those pages of memory allocated to the stack as *non-executable*, meaning that if the processor's instruction register is loaded with an address on those pages (due to branching to stack-based shellcode), a hardware-level exception will immediately occur. This immediately prevents attacks that place a shellcode on the stack, though return-to-libc attacks (*22.2.1 Return to libc*) are still possible.

In the x86 world, AMD introduced the per-page NX bit, for No eXecute, in their x86-64 architecture; CPUs with this architecture began appearing in 2003. Intel followed with its XD, for eXecute Disabled, bit. Essentially all x86-64 CPUs now provide hardware NX/XD support; support on 32-bit CPUs generally requires so-called Physical Address Extension mode.

The NX idea applies to all memory pages, not just stack pages. This sometimes causes problems with applications such as just-in-time compilation, where the code page is written and then immediately executed. As a result, it is common to support NX-bit disabling in software. On linux systems, the compiler option `-z execstack` disables NX-bit protection; this was used above in *22.2.2.1 The server*. Windows has a similar `/NXCOMPAT` option for requesting NX-bit protection.

While a non-executable stack prevents the stack-overflow attack described above, injecting shellcode onto the heap is still potentially possible. The OpenBSD operating system introduced **write or execute** in 2003; this is abbreviated **W^X** after the use of "^" as the XOR operator in C. A memory page may be writable or executable, but not both. This is strong protection against virtually all shellcode-injection attacks, but may still be vulnerable to some return-to-libc attacks (*22.2.1 Return to libc*).

See [AHLR07], chapter 14, for some potential attack strategies against systems with non-executable pages.

22.3 Heap Buffer Overflow

As with stack overflows, heap overflows all rely on some software flaw that allows data to be written beyond the confines of the designated buffer. A buffer on the heap is subject to the same software-failure overflow

prospects as a buffer on the stack. An important difference, however, is that buffers on the heap are not in clear proximity to an obvious return address. Despite that difference, heap overflows can also be used to enable remote-code-execution attacks.

Perhaps the simplest heap overflow is to take advantage of the fact that some heap pages contain executable code, representing application-loaded library pages. If the page with the overflowable buffer is pointed to by `p`, and the following page in memory pointed to by `q` contains code, then all an attacker has to do is to have the overflow fill the `q` page with a long NOPslide and a shellcode. When at some point a call is made to the code pointed to by `q`, the shellcode is executed instead. A drawback to this attack is that the layout of heap pages like this is seldom known. On the other hand, the fact that heap pages do sometimes legitimately contain executable code means that uniformly marking the heap as non-executable, as is commonly done with the stack, may not be an option.

22.3.1 A linux heap vulnerability

We now describe an actual linux heap vulnerability from 2003, following *[AB03]*, based on version 2.2.4 of the glibc library. The vulnerable server code is simply the following:

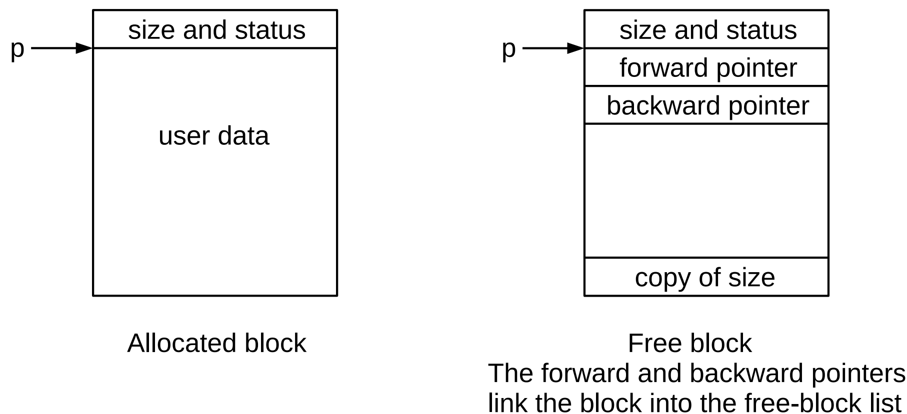
```
char *p = malloc(1024);
char *q = malloc(1024);
gets(p);              // read the attacker's input
free(q);              // block q is last allocated and first freed
free(p);
```

As with the stack-overflow example, the `gets(p)` results in an overflow from block p into block q, overwriting not just the data in block q but also the block headers maintained by `malloc()`. While there is no guarantee in general that block `q` will immediately follow block `p` in memory, in practice this usually happens unless there has been a great deal of previous allocate/free activity.

The vulnerability here relies on some features of the 2003-era (glibc-2.2.4) `malloc()`. All blocks are either allocated to the user or are free; free blocks are kept on a doubly linked list. We will assume the data portion of each block is always a power of 2; there is a separate free-block list for each block size. When two adjacent blocks are both free, `malloc()` coalesces them and moves the joined block to the free-block list of the next size up.

All blocks are prefixed by a four-byte field containing the size, which we will assume here is the actual size 1032 including the header and alignment rather than the user-available size of 1024. As the three low-order bits of the size are always zero, one of these bits is used as a flag to indicate whether the block is allocated or free. Crucially, another bit is used to indicate whether the *previous* block is allocated or free.

In addition to the size-plus-flags field, the first two 32-bit words of a free block are the forward and backward pointers linking the block into the doubly linked free-block list; the size-plus-flag field is also replicated as the last word of the block:



Allocated block

Free block
The forward and backward pointers
link the block into the free-block list

The strategy of the attacker, in brief, is to overwrite the `p` block in such a way that, when block `q` is freed, `malloc()` thinks block `p` is also free, and so attempts to coalesce them, in the process unlinking block `p`. But `p` was not in fact free, and the `forward` and `backward` pointers manipulated by `malloc()` as part of the unlinking process are in fact provided by the attacker. As we shall see, this allows writing two attacker-designated 32-bit words to two attacker-designated memory locations; if one of the locations holds a return address and is updated so as to point to attacker-supplied shellcode also in block `p`, the system has been compromised.

The normal operation of `free(q)`, for an arbitrary block `q`, includes the following:

- Get the block size (`size`) and flags at address `q-4`
- Check the following block at address `p+size` to see if it is free; if so, merge (we are not interested in this case)
- Check the flags to see if the *preceding* block is free; if so, load its size `prev_size` from address `q-8`, the address of the `copy of size` field in the diagram above; from that calculate a pointer to the previous block as `p = q - prev_size`; then unlink block `p` (as the coalesced block will go on a different free-block list).

For our specific block `q`, however, the attacker can overwrite the final `size` field of block `p`, `prev_size` above, and can also overwrite the flag at address `q-4` indicating whether or not block `p` is free. The attacker can *not* overwrite the header of block `p` that would properly indicate that block `p` was still in use, but the `free()` code did not double-check that.

We will suppose that the attacker has overwritten block `p` to include the following:

- setting the previous-block-is-free flag in the header of block `q` to true
- setting the final `size` field of block `p` to a desired value, `badsize`
- placing value `ADDR_F` at address `q-badsize`; this is where `free()` will believe the previous block's forward pointer is located
- placing the value `ADDR_B` at address `q-badsize+4`; this is where `free()` will believe the previous block's backward pointer is located

When the `free(q)` operation is now executed, the system will calculate the previous block as at address `p1 = q-badsize` and attempt to unlink the false "block" `p1`. The normal unlink is

```
(p->backward) -> forward  = p->forward;
(p->forward)  -> backward = p->backward;
```

Alas, when unlinking `p1` the result is

```
*ADDR_B       = ADDR_F
*(ADDR_F + 4) = ADDR_B
```

where, for the pointer increment in the second line, we take the type of `ADDR_F` to be `char *` or `void *`.

At this point the jig is pretty much up. If we take `ADDR_B` to be the location of a return address on the stack, and `ADDR_F` to be the location of our shellcode, then the shellcode will be executed when the current stack frame returns. Extending this to a working example still requires a fair bit of attention to details; see *[AB03]*.

One important detail is that the data we use to overwrite block `p` generally cannot contain NUL bytes, and yet a small positive number for `badsize` will have several NUL bytes. One workaround is to have `badsize` be a small *negative* number, meaning that the false previous-block pointer `p1` will actually come after `q` in memory.

22.3.2 A JPEG heap vulnerability

In 2004 Microsoft released vulnerability notice MS04-028 (and patch), relating to a **heap buffer overflow** in **Windows XP SP1**. Microsoft had earlier provided a standard library for displaying JPEG images, known as GDI for Graphics Device Interface. If a specially formed JPEG image were opened via the GDI library, an overflow on the heap would occur that would launch a shellcode. While most browsers had their own, unaffected, JPEG-rendering subroutines, it was often not difficult to convince users to open a JPEG file supplied as an email attachment or via an html download link.

The problem occurred in the processing of the JPEG "comment" section, normally invisible to the user. The section begins with the flag 0xFFFE, followed by a two-byte length field (in network byte order) that was to include its own two bytes in the total length. During the image-file loading, 2 was subtracted from the length to get the calculated length of the actual data. If the length field had contained 1, for example, the calculated length would be -1, which would be interpreted as the unsigned value 2^{32} - 1. The library, thinking it had that amount of space, would then blithely attempt to write that many bytes of comment into the next heap page, overflowing it. These bytes in fact would come from the image portion of the JPEG file; the attacker would place here a NOPslide, some shellcode, and, as we shall see, a few other carefully constructed values.

As with the linux heap described above in *22.3.1 A linux heap vulnerability*, blocks on the WinXP heap formed a doubly-linked list, with forward and backward pointers known as `flink` and `blink`. As before, the allocator will attempt to unlink a block via

```
blink -> forward  = flink;
flink -> backward = blink;
```

The first of these simplifies to `*blink = flink`, as the offset to field `forward` is 0; this action allows the attacker to write any word of memory (at address `blink`) with any desired value.

The JPEG-comment-overflow operation eventually runs out of heap and results in a segmentation fault, at which point the heap manager attempts to allocate more blocks. However, the free list has already been

overwritten, and so, as above, this block-allocation attempt instead executes `*blink = flink`.

The attacker's conceptual goal is to have `flink` hold an instruction that branches to the shellcode, and `blink` hold the address of an instruction that will soon be executed, or, equivalently, to have `flink` hold the address of the shellcode and `blink` represent a location soon to be loaded and used as the target of a branch. The catch is that the attacker doesn't exactly know where the heap is, so a variant of the return-to-libc approach described in *22.2.1 Return to libc* is necessary. The strategy described in the remainder of this section, described in *[JA05]*, is one of several possible approaches.

In Windows XP SP1, location `0x77ed73b4` holds a pointer to the entry point of the *Undefined Exception Filter* handler; if an otherwise-unhandled program exception occurs, Windows creates an `EXCEPTION_POINTERS` structure and branches to the address stored here. It is this address, which we will refer to as **UEF**, the attack will overwrite, by setting `blink = UEF`. A call to the Undefined Exception Filter will be triggered by a suitable subsequent program crash.

When the exception occurs (typically by the second operation above, `flink -> backward = blink`), before the branch to the address loaded from UEF, the `EXCEPTION_POINTERS` structure is created on the heap, overwriting part of the JPEG comment buffer. The address of this structure is stored in register `edi`.

It turns out that, scattered among some standard libraries, there are half a dozen instructions at known addresses of the form `call DWORD [edi+0x74]`, that is, "call the subroutine at 32-bit address `edi` + 0x74" (*[AHLR07]*, p 186). All these `call` instructions are intended for contexts in which register `edi` has been set up by immediately preceding instructions to point to something appropriate. In our attacker's context, however, `edi` points to the `EXCEPTION_POINTERS` structure; 0x74 bytes past `edi` is part of the attacker's overflowed JPEG buffer that is safely past this structure and will not have been overwritten by it. One such call instruction happens to be at address `0x77d92a34`, in `user32.dll`. This address is the value the attacker will put in `flink`.

So now, when the exception comes, control branches to the address stored in UEF. This address now points to the above `call DWORD [edi+0x74]`, so control branches again, this time into the attacker-controlled buffer. At this point, the processor lands on the NOPslide and ends up executing the shellcode (sometimes one more short jump instruction is needed, depending on layout).

This attack depends on the fact that a specific instruction, `call DWORD [edi+0x74]`, can be found at a specific, fixed address, `0x77d92a34`. Address-space layout randomization (*22.2.3.2 ASLR*) would have prevented this; it was introduced by Microsoft in Windows Vista in 2007.

22.3.3 Cross-Site Scripting (XSS)

In its simplest form, cross-site scripting involves the attacker posting malicious javascript on a third-party website that allows user-posted content; this javascript is then executed by the target computer when the victim visits that website. The attacker might leave a comment on the website of the following form:

```
I agree with the previous poster completely
<script> do_something_bad() </script>
```

Unless the website in question does careful html filtering of what users upload, any other site visitor who so much as *views* this comment will have the `do_something_bad()` script executed by his or her browser. The script might email information about the target user to the attacker, or might attempt to exploit a browser vulnerability on the target system in order to take it over completely. The script and its enclosing tags will not appear in what the victim actually sees on the screen.

The `do_something_bad()` code block will usually include javascript function definitions as well as function calls.

In general, the attacker can achieve the same effect if the victim visits the attacker's website. However, the popularity (and apparent safety) of the third-party website is usually important in practice; it is also common for the attack to involve obtaining private information from the victim's account on that website.

22.3.4 SQL Injection

SQL is the close-to-universal query language for databases; in a SQL-injection attack the attacker places a carefully chosen SQL fragment into a website form, in such a way that it gets executed. Websites typically construct SQL queries based on form data; the attacker's goal is to have his or her data treated as additional SQL. This is *supposed* to be prevented by careful quoting, but quoting often ends up not quite careful *enough*. A successful attacker has managed to run SQL code on the server that gives him or her unintended privileges or information.

As an example, suppose that the form has two fields, `username` and `password`. The system then runs the following sub-query that returns an empty set of records if the ⟨username,password⟩ pair is not found; otherwise the user is considered to be authenticated:

> select * from PASSWORDS p
> where p.user = '*username*' and p.password = '*password*' ;

The strings *username* and *password* are taken from the web form and spliced in; note that each is enclosed in single quotation marks *supplied by the server*. The attacker's goal is to supply username/password values so that a nonempty set of records will be returned, thus authenticating the attacker. The following values are successful here, where the quotation marks in *username* are supplied by the attacker:

> *username*: `'  OR 1=1 OR 'x'='y`
> *password*: `foo`

The spliced-together query built by the server is now

> select * from PASSWORDS p
> `where p.user = '' OR 1=1 OR 'x'='y' and p.password = 'foo';`

Note that of the eight single-quote marks in the where-clause, four (the first, sixth, seventh and eighth) came from the server, and four (the second through fifth) came from the attacker.

The where-clause here appears to SQL to be the disjunction of three OR clauses (the last of which is `'x'='y' and p.password = 'foo'`). The middle OR clause is `1=1` which is always true. Therefore, the login succeeds.

For this attack to work, the attacker must have a pretty good idea what query the server builds from the user input. There are two things working in the attacker's favor here: first, these queries are often relatively standard, and second, the attacker can often discover a great deal from the error messages returned for malformed entries. In many cases, these error messages even reveal the table names used.

See also xkcd.com/327.

22.4 Network Intrusion Detection

The idea behind **network intrusion detection** is to monitor ones network for signs of attack. Many newer network intrusion-detection systems (NIDS) also attempt to halt the attack, but the importance of simple monitoring and reporting should not be underestimated. Many attacks (such as password guessing, or buffer overflows in the presence of ASLR) take multiple (thousands or millions) of tries to succeed, and a NIDS can give fair warning.

There are also host-based intrusion-detection systems (HIDS) that run on and monitor a specific host; we will not consider these further.

Most NIDS detect intrusions based either on **traffic anomalies** or on pattern-based **signatures**. As an example of the former, a few pings (*7.11 Internet Control Message Protocol*) might be acceptable but a large number, or a modest number addressed to nonexistent hosts, might be cause for concern.

As for signatures, the attack in *22.3.2 A JPEG heap vulnerability* can be identified by the hex strings 0xFFFE0000 or 0xFFFE0001. What about the attack in *22.2.2.3 The exploit*? Using the shellcode itself as signature tends to be ineffective as shellcode is easy to vary. The NOPslide, too, can be replaced with a wide variety of other instructions that just happen to do nothing in the present context, such as `sub eax,eax`. One of the most common signatures used by NIDSs for overflow attacks is simply the presence of overly long strings; the false-positive rate is relatively low. In general, however, coming up with sufficiently specific signatures can be nontrivial. An attack that keeps changing in relatively trivial ways to avoid signature-based detection is sometimes said to be **polymorphic**.

22.4.1 Evasion

The NIDS will have to reassemble TCP streams (and sometimes sequences of UDP packets) in order to match signatures. This raises the possibility of **evasion**: the attacker might arrange the packets so that the NIDS reassembles them one way and the target another way. The possibilities for evasion are explored in great detail in *[PN98]*; see also *[SP03]*.

One simple way to do this is with overlapping TCP segments. What happens if one packet contains bytes 1-6 of a TCP connection as "help" and a second packet contains bytes 2-7 as "armful"?

```
h   e   l   p
    a   r   m   f   u   l
```

These can be reassembled as either "helpful" or "harmful"; the TCP specification does not say which is preferred and different operating systems routinely reassemble these in different ways. If the NIDS reassembles the packets one way, and the target the other, the attack goes undetected. If the attack is spread over multiple packets, there may be many more than two ways that reassembly can occur, increasing the probability that the NIDS and the target will differ.

Another possibility is that one of the overlapping segments has a header irregularity (in either the IP or TCP header) that causes it to be rejected by the target but not by the NIDS, or vice-versa. If the packets are

```
h   e   l   p
h   a   r   m   f   u   l
```

and both systems normally prefer data from the first segment received, then both would reassemble as "helpful". But if the first packet is rejected by the target due to a header flaw, then the target receives

"harmful". If the flaw is not recognized by the NIDS, the NIDS does not detect the problem.

A very similar problem occurs with IPv4 fragment reassembly, although IPv4 fragments are at this point intrinsically suspicious.

One approach to preventing evasion is to configure the NIDS with information about how the actual target does its reassembly, so the NIDS can match it. An even safer approach is to have the NIDS reassemble any overlapping packets and then forward the result on to the potential target.

22.5 Cryptographic Goals

For the remainder of this chapter we turn to the use of cryptographic techniques in networking, to protect packet contents. Different techniques address different issues; three classic goals are the following:

1. Message **confidentiality**: eavesdroppers should not be able to read the contents.
2. Message **integrity**: the recipient should be able to verify that the message was received correctly, even in the face of a determined adversary along the way.
3. Sender **authentication**: the recipient should be able to verify the identity of the sender.

Briefly, confidentiality is addressed through encryption (*22.7 Shared-Key Encryption* and *22.9 Public-Key Encryption*), integrity is addressed through secure hashes (*22.6 Secure Hashes*), and authentication is addressed through secure hashes and public-key signatures.

Encryption by itself does not ensure message integrity. In *22.7.4 Stream Ciphers* we give an example using the message "Transfer $ **02**000 to Mallory". It is encrypted by XORing with the corresponding number of bytes of the keystream (*22.7 Shared-Key Encryption*), and decrypted by XORing again. If the attacker XORs the two bytes in bold with the byte ('0' XOR '2'), the message becomes "Transfer $ **20**000 to Mallory"; if the attacker XORs those bytes in the encrypted message with ('0' XOR '2') then the result will decrypt to the $20,000 transfer.

Similarly, integrity does not automatically ensure authentication. Two parties can negotiate a temporary key to guarantee message integrity without ever establishing each other's identities as is necessary for authentication. For example, if Alice connects to a website using SSL/TLS, but the site never purchased an SSL certificate (*22.10.2 TLS* and *22.10.2.1 Certificate Authorities*), or Alice does not trust the site's certificate authority (*22.10.2.1 Certificate Authorities*), then Alice has message integrity for her session, but not authentication.

To the above list we might add resistance to message **replay**. Consider messages such as the following:

1. "I, Alice, authorize the payment to Bob of $1000 from my account"
2. My facebook.com authentication cookie is Zg8yPCDwbzJ-59Hc-DvHt67qrS

Alice does not want the first message to be executed by her bank more than once, though doing so would violate none of the three rules above. The owner of the second message might be happy to replay it multiple times, but would not want someone *else* to be able to do so. One straightforward way to prevent replay attacks is to introduce a message timestamp or count.

We might also desire resistance to **denial-of-service attacks**. Such attacks are generally related to implementation failures. Attacks in which SSL users end up negotiating a very early version of the protocol,

and thus a much less secure encryption mechanism, are arguably of this type; see the POODLE sidebar at *22.10 SSH and TLS*.

Finally, one sometimes sees *message non-repudiation* as a goal. However, in the technical – as opposed to legal – realm we can never hope to prove Alice herself signed a message; the best we can do is to prove that Alice's *key* was used to sign the message. At this point non-repudiation is, for our purposes here, quite similar to authentication. See, however, the final example of *22.6.1 Secure Hashes and Authentication*.

22.5.1 Alice and Bob

Cryptographers usually use Alice and Bob, instead of A and B, as the names of the parties sending each other encrypted messages. This tradition dates back at least to *[RSA78]*. Other parties sometimes include Eve the eavesdropper and Mallory the active attacker (and launcher of man-in-the-middle attacks). (In this article the names Aodh and Bea are introduced, though not specifically for cryptography; the Irish name Aodh is pronounced, roughly, as "Eh".)

22.6 Secure Hashes

How can we tell if a file has been changed? One way is to create a record of the file's checksum, using, for the sake of argument, the Internet checksum of *5.4 Error Detection*. If the potential file corruption is random, this will fail to detect a modified file with probability only 1 in 2^{16}.

Alas, if the modification is *intentional*, it is trivial to modify the file so that it has the same checksum as the original. If the original file has checksum c_1, and after the initial modification the checksum is now c_2, then all we have to do to create a file with checksum c_1 is to append the 16-bit quantity $c_2 - c_1$ (where the subtraction is done using ones-complement; we need to modify this slightly if the number of bytes in the first-draft modified file is odd). The CRC family of checksums is almost as easily tricked.

The goal of a **cryptographically secure hash** is to provide a hash function – we will not call it a "checksum" as hashes based on simple sums all share the insecurity above – for which this trick is well-nigh impossible. Specifically, we want a hash function such that:

- Knowing the hash value should shed no practical light on the message
- Given a hash value, there should be no feasible way to find a message yielding that hash

A slightly simpler problem than the second one above is to find two messages that have the same hash; this is sometimes called the **collision problem**. When the collision problem for a hash function has been solved, it is (high) time to abandon it as potentially no longer secure.

If a single bit of the input is changed, the secure-hash-function output is usually entirely different.

Hashes popular in the 1990s were the 128-bit MD5 (**RFC 1321**, based on MD4, *[RR91]*) and the 160-bit SHA-1 (developed by the NSA); SNMPv3 originally supported both of these (*21.15.2 Cryptographic Fundamentals*). MD5 collisions (two messages hashing to the same value) were reported in 2004, and collisions where both messages were meaningful were reported in 2005; such **collision attacks** mean it can no longer be trusted for security purposes.

Hash functions currently (2014) believed secure include the SHA-2 family, which includes variants ranging from 224 bits to 512 bits (and known individually as SHA-224 through SHA-512).

A common framework for constructing n-bit secure-hash functions is the Merkle-Dåmgard construction (*[RM88]*, *[ID89]*); it is used by MD5, SHA-1 and SHA-2. The initial n-bit state is specified. To this is then applied a set of transformations $H_i(x,y)$ that each take an n-bit block x and some additional bits y and return an updated n-bit block. These transformations are usually similar to the **rounds functions** of a block cipher; see *22.7.2 Block Ciphers* for a typical example. In encryption, the parameter y would be used to hold the key, or a substring of the key; in secure-hash functions, the parameter y holds a substring of the input message. The set of transformations is applied repeatedly as the process iterates over the entire input message; the result of the hash is the final n-bit state.

In the MD5 hash function, the input is processed in blocks of 64 bytes. Each block is divided into sixteen 32-bit words. One such block of input results in 64 iterations from a set of sixteen rounds-functions H_i, each applied four times in all. Each 32-bit input word is used as the "key" parameter to one of the H_i four times. If the total input length is not a multiple of 512 bits, it is padded; the padding includes a length attribute so two messages differing only by the amount of padding should not hash to the same value.

While this framework in general is believed secure, and is also used by the SHA-2 family, it does suffer from what is sometimes called the **length-extension** vulnerability. If h = hash(m), then the value h is simply the final state after the above mechanism has processed message m. An attacker knowing only h can then initialize the above algorithm with h, and continue it to find the hash $h' = \text{hash}(m \frown m')$, for an arbitrary message m' concatenated to the end of m, *without knowing m*. If the original message m was padded to message m_p, then the attacker will find $h' = \text{hash}(m_p \frown m')$, but that is often enough. This vulnerability must be considered when using secure-hash functions for message authentication, below.

The SHA-3 family of hash functions does not use the Merkle-Dåmgard construction and is believed not vulnerable to length-extension attacks.

22.6.1 Secure Hashes and Authentication

Secure hash functions can be used to implement message authentication. Suppose the sender and receiver share a secret, pre-arranged "key", K, a random string of length comparable to the output of the hash. Then, in principle, the sender can append to the message m the value $h = \text{hash}(K \frown m)$. The receiver, knowing K, can recalculate this value and verify that the h appended to the message is correct. In theory, only someone who knew K could calculate h.

This *particular* $\text{hash}(K \frown m)$ implementation is undermined by the length-extension vulnerability of the previous section. If the hash function exhibits this vulnerability and the sender transmits message m together with $\text{hash}(K \frown m)$, then an attacker can modify this to message $m \frown m'$ together with $\text{hash}(K \frown m \frown m')$, *without knowing K*.

This problem can be defeated by reversing the order and using $\text{hash}(m \frown K)$, but this now introduces potential collision vulnerabilities: if the hash function has the length-extension vulnerability and two messages m_1 and m_2 hash to the same value, then so will $m_1 \frown K$ and $m_2 \frown K$.

Taking these vulnerabilities into account, **RFC 2104** defines the **Hash Message Authentication Code**, or HMAC, as follows; it can be used with any secure-hash function whether or not it has the length-extension vulnerability. The 64-byte length here comes from the typical input-block length of many secure-hash functions; it may be increased as necessary. (SHA-512, of the SHA-2 family, has a 128-byte input-block length and would be a candidate for such an increase.)

- The shared key K is extended to 64 bytes by padding with zeroes.

- A constant string **ipad** is formed by repeating the octet 0x36 (0011 0110) 64 times.
- A constant string **opad** is formed by repeating 0x5c (0101 1100) 64 times.
- We set **K1** = K XOR **ipad**.
- We set **K2** = K XOR **opad**.

Finally, the HMAC value is

HMAC = hash(**K2** ⌒ hash(**K1**⌒**mesg**))

The values 0x36 (0011 0110) and 0x5c (0101 1100) are not critical, but the XOR of them has, by intent, a reasonable number of both 0-bits and 1-bits; see *[BCK96]*.

The HMAC algorithm is, somewhat surprisingly, believed to be secure even when the underlying hash function is vulnerable to some kinds of collision attacks; see *[MB06]* and **RFC 6151**. That said, a hash function vulnerable to collision attacks may have other vulnerabilities as well, and so HMAC-MD5 should still be phased out.

Negotiating the pre-arranged key K can be a significant obstacle, just as it can be for ciphers using pre-arranged keys (*22.7 Shared-Key Encryption*). If Alice and Bob meet in person to negotiate the key K, then Alice can use HMAC for authentication of Bob, as long as K is not compromised: if Alice receives a message signed with K, she knows it must have come from Bob.

Sometimes, however, K is negotiated on a per-session basis, without definitive "personal" authentication of the other party. This is akin to Alice and someone claiming to be "Bob" selecting an encryption key using the Diffie-Hellman-Merkle mechanism (*22.8 Diffie-Hellman-Merkle Exchange*); such key selection is secure and does not require that Alice or Bob have any prior knowledge of one another. In the HMAC setting, Alice can be confident that any HMAC-signed message was sent by the same "Bob" that negotiated the key, and not by a third party (assuming neither side has leaked the key K). This is true even if Alice is not sure "Bob" is the real Bob, or has no idea who "Bob" might be. The signature guarantees the message *integrity*, but also serves as *authentication* that the sender is the same entity who sent the previous messages in the series.

Finally, if Alice receives a message from Bob signed with HMAC using a pre-arranged secret key K (*22.6.1 Secure Hashes and Authentication*), Alice may herself trust the signature, but she cannot prove to anyone else that K was used to sign the message without divulging K. She also cannot prove to anyone else that Bob is the only other party who knows K. Therefore this signature mechanism provides authentication but not non-repudiation.

22.6.2 Password Hashes

A site generally does not store user passwords directly; instead, a hash of the password, h(p), is stored. The cryptographic hash functions above, *eg* MD5 and SHA-n, work well as long as the password file itself is not compromised. However, these functions all execute very quickly – by design – and so an attacker who succeeds in obtaining the password file can usually extract passwords simply by calculating the hash of every possible password. There are about 2^{14} six-letter English words, and so there are about 2^{38} passwords consisting of two such words and three digits. Such passwords are usually considered rather strong, but

brute-force calculation of h(p) for 2^{38} possible values of p is quite feasible for the hashes considered so far. Checking 10^8 MD5-hashed passwords per second is quite feasible; this means $2^{38} = 256 \times 2^{30} \simeq 256 \times 10^9$ passwords can be checked in under 45 minutes. Use of a GPU increases the speed at least 10-fold.

Special hashes have been developed to address this. Two well-known ones are scrypt (*[CP09]* and this Internet Draft) and bcrypt. A newer entrant is Argon2, while PBKDF2 is an old stalwart. The goal here is to develop a hash that takes – ideally – the better part of a second to calculate even once, and which cannot easily be sped up by large precomputed tables. Typically these are something like a thousand to a million times slower than conventional hash functions like MD5 and the SHA-2 family; a millionfold-slower hash function is the equivalent of making the passwords longer by 20 random bits ($10^6 \simeq 2^{20}$). See *[ACPRT16]* for a theoretical analysis of the effectiveness of scrypt in this context. See also **RFC 2898**, section 4.2, though the proposal there is only a thousandfold slower.

(Good password tables also incorporate a *salt*, or *nonce*: a random string s is chosen and appended to the password before hashing; the password record is stored as the pair $\langle$s,h(p$\frown$s)$\rangle$. This means that cracking the password for one user will not expose a second user who has chosen exactly the same password, so long as the second user has a different salt s. Salting, however, does not change the scenarios outlined above.)

22.6.3 CHAP

Secure hashes can also be used to provide secure password-based login authentication in the presence of eavesdropping. Specific implementations include CHAP, the Challenge-Handshake Authentication Protocol (**RFC 1994**) and Microsoft's MS-CHAP (version 1 in **RFC 2433** and version 2 in **RFC 2759**). The general idea is that the server sends a random string (the "challenge") and the client then creates a response consisting of a secure hash of the challenge string concatenated with the user's password (and possibly other information). Assuming the secure hash is actually secure, only someone in possession of the user password could have created this response. By the same token, an eavesdropper cannot figure out the password from the response.

While such protocols can be quite secure in terms of verifying to the server that the client knows the password, the CHAP strategy has a fundamental vulnerability: the server must store the plaintext password rather than a secure hash of it (as in the previous section). An attacker who makes off with the server's password file then has everything; no brute-force password cracking is needed. For this reason, newer authentication protocols often will create an encrypted channel first (*eg* using TLS, *22.10.2 TLS*), and then use that secure channel to exchange credentials. This may involve transmission of the *plaintext* password, which clearly allows the server to store only hashed passwords. However, as the next example shows, it *is* possible to authenticate without having the server store plaintext passwords and without having the plaintext password transmitted at all.

22.6.4 SCRAM

SCRAM (Salted Challenge-Response Authentication Mechanism), **RFC 5802**, is an authentication protocol with the following features:

- The client password is not transmitted in the clear
- The server does not store the plaintext client password
- If the server's hashed credentials are compromised, an attacker still cannot authenticate

We outline a very stripped-down version of the protocol: password salting has been removed for clarity, and the exchange itself has been greatly simplified. The **ClientKey** is a hashed version of the password; what the server stores is a secure hash of the ClientKey called **StoredKey**:

```
StoredKey = hash(ClientKey)
```

The exchange begins with the server sending a random **nonce** string to the client. The client now calculates the **ClientSignature** as follows:

```
ClientSignature = hash(StoredKey, nonce)
```

Only an attacker who has eavesdropped on this exchange can replicate the ClientSignature, but the server can compute it. The client then sends the following to the server:

```
ClientKey XOR ClientSignature
```

Because the server can calculate ClientSignature, it can extract ClientKey, and verify that h(ClientKey) = StoredKey. An attacker who has obtained StoredKey from the server can generate ClientSignature, but cannot generate ClientKey.

However, an attacker who has *both* exfiltrated StoredKey from the server *and* eavesdropped on a client-server exchange is able to generate the ClientSignature and thus extract the ClientKey. The attacker can then authenticate at a later time using ClientKey. For this reason, a secure tunnel is still needed for the authentication. Given the presence of such a tunnel, the SCRAM approach appears to offer a relatively modest security improvement over sending the password as plaintext. Note, though, that with SCRAM the server does not see the password at all, so if the client mistakenly connects to the wrong server, the password is not revealed.

22.7 Shared-Key Encryption

Secure hashes can provide authentication, but to prevent eavesdropping we need encryption. While **public-key** encryption (*22.9 Public-Key Encryption*) is perhaps more glamorous, the workhorse of the encryption world is the **shared-key cipher**, or **shared-secret** or **symmetric** cipher, in which each party has possession of a key K, used for both encryption and decryption.

Shared-key ciphers are often quite fast; public-key encryption is generally slower by several orders of magnitude. As a result, if two parties without a shared key wish to communicate, they will almost always used public-key encryption only to exchange a key for a shared-key cipher, which will then be used for the actual message encryption.

Typical key lengths for shared-key ciphers believed secure range from 128 bits to 256 bits. For most shared-key ciphers there are no known attacks that are much faster than brute force, and $2^{256} \simeq 10^{77}$ is quite a large number.

Shared-key ciphers can be either **block ciphers**, encrypting data in units of blocks that might typically be 8 bytes long, or **stream** ciphers, which generate a pseudorandom **keystream**. Each byte (or even bit) of the message is then encrypted by (typically) XORing it with the corresponding byte of the keystream.

22.7.1 Session Keys

The more messages a key is used to encrypt, the more information a potential eavesdropper may have with which to launch a codebreaking attack. If Alice and Bob have a pre-arranged shared secret key K, is therefore quite common for them to encrypt the bulk of their correspondence with temporary **session keys**, each one used for a time period that may range from minutes to days. The secret key K might then be used only to exchange new session keys and to forestall man-in-the-middle attacks (*22.9.3 Trust and the Man in the Middle*) by signing important messages (*22.6.1 Secure Hashes and Authentication*). If Diffie-Hellman-Merkle key exchange (ref:*diffie-hellman*) is used, K might be reserved *only* for message signing.

Sometimes session keys are entirely different keys, chosen randomly; if such a key is discovered, the attacker learns nothing about the secret key K. Other times, the session key is a mash-up of K and some additional information (such as an initialization vector, *22.7.3 Cipher Modes*), that is transmitted in the clear. The session key might then be the concatenation of K and the initialization vector, or the XOR of the two. Either approach means that an eavesdropper who figures out the session key also has the secret key K, but the use of such session keys still may make any codebreaking attempt much more difficult.

In *22.9.2 Forward Secrecy* we consider the reverse problem of how Alice and Bob might keep a session key private even if the secret key is compromised.

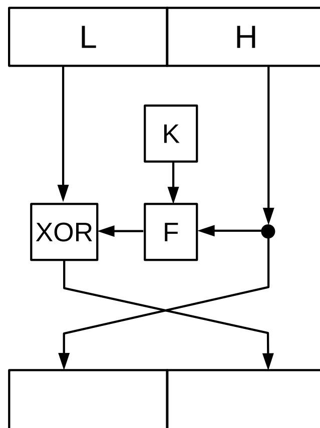
22.7.2 Block Ciphers

As mentioned, a block cipher encrypts data one block at a time, typically with a key length rather longer than the block size. A typical block cipher proceeds by the iterated application of a sequence of **round functions**, each updating the block and using some round-dependent substring of the key as auxiliary input. If we start with a block of plaintext P and apply all the round functions in turn, the result is the ciphertext block C = E(P,K) = E_K(P). The process can be reversed so that P = D(C,K) = D_K(C).

A common framework is that of the **Feistel network**. In such an arrangement, the block is divided into two or more words sized for the machine architecture; an 8-byte block is typically divided into two 32-bit words which we can call L and H for Low and High. A typical round function is now of the following form, where K is the key and F(x,K) takes a word x and the key K and returns a new word:

$$\langle L,H\rangle \longrightarrow \langle H, L \text{ XOR } F(H,K)\rangle$$

Visually, this is often diagrammed as



One round here scrambles only half the block, but the other half gets scrambled in the next round (sometimes the operation above is called a half-round for that reason). The total number of rounds is typically somewhere between 10 and 50. Much of the art of designing high-quality block ciphers is to come up with round functions that result in overall very fast execution, without introducing vulnerabilities. The more rounds, the slower.

The internal function F, often different for each round, may look at only a designated subset of the bits of the key K. Note that the operation above is invertible – that is, can be decrypted – regardless of F; given the right-hand side the receiver can compute F(H,K) and thus, by XORing this with L XOR F(H,K), can recover L. This remains true if, as is sometimes the case, the XOR operation is replaced with ordinary addition.

Crypto Law

The Salsa20 cipher mentioned here is a member of Daniel Bernstein's "snuffle" family of ciphers based on secure-hash functions. In the previous century, the US government banned the export of ciphers but not secure-hash functions. They also at one point banned the export (and thus publication) of one of Bernstein's papers; he sued. In 1999, the US Court of Appeals for the Ninth Circuit found in his favor. The decision, 176 F.3d 1132, is the only appellate decision of which the author is aware that contains (in the footnotes) not only samples of C code, but also of Lisp.

A simple F might return the result of XORing H and a subset of the bits of K. This is usually a little *too* simple, however. Ordinary modulo-32 addition of H and a subset of K often works well; the interaction of addition and XORing introduces considerable bit-shuffling (or *diffusion* in the language of cryptography). Other operations used in F(H,K) include Boolean AND and OR operations. 32-bit multiplication introduces considerable bit-shuffling, but is often computationally more expensive. The **Salsa20** cipher of *[DB08]* uses only XOR and addition, for speed.

It is not uncommon for the round function also to incorporate "bit rotation" of one or both of L and H; the result of bit-rotation of 1000 **111**0 two places to the left is 00**11 1**010.

If a larger blocksize is desired, say 128 bits, but we still want to avail ourselves of the efficiency of 32-bit operations, the block can be divided into ⟨A,B,C,D⟩. The round function might then become

⟨A,B,C,D⟩ ⟶ ⟨B,C,D, (A XOR F(B,C,D,K)) ⟩

As mentioned above, many secure-hash functions use block-cipher round functions that then use successive chunks of the message being hashed in place of the key. In the MD5 algorithm, block A above is transformed into the 32-bit sum of input-message fragment M, a constant K, and G(B,C,D) which can be any of several Boolean combinations of B, C and D.

An alternative to the Feistel-network framework for block ciphers is the use of so-called substitution-permutation networks.

The first block cipher in widespread use was the federally sanctioned Data Encryption Standard, or **DES** (commonly pronounced "dez"). It was developed at IBM by 1974 and then selected by the US National Bureau of Standards (NBS) as a US standard after some alterations recommended by the National Security Agency (NSA). One of the NSA's recommendations was that a key size of 56 bits was sufficient; this was in an era when the US government was very concerned about the spread of strong encryption. For years, many people assumed the NSA had intentionally introduced other weaknesses in DES to facilitate government eavesdropping, but after forty years no such vulnerability has been discovered and this no longer seems so likely. The suspicion that the NSA had in the 1970's the resources to launch brute-force attacks against DES, however, has greater credence.

In 1997 an academic team launched a successful brute-force attack on DES. The following year the Electronic Frontier Foundation (EFF) built a hardware DES cracker for about US$250,000 that could break DES in a few days.

In an attempt to improve the security of DES, triple-DES or **3DES** was introduced. It did not become an official standard until the late 1990's, but a two-key form was proposed in 1978. 3DES involves three applications of DES with keys $\langle$K1,K2,K3$\rangle$; encrypting a plaintext P to ciphertext C is done by C = $E_{K3}(D_{K2}(E_{K1}(P)))$. The middle deciphering option D_{K2} means the algorithm reduces to DES when K1 = K2 = K3; it also reduces exposure to a particular vulnerability known as "meet in the middle" (no relation to "man in the middle"). In *[MH81]* it is estimated that 3DES with three distinct keys has a strength roughly equivalent to 2×56 = 112 bits. That same paper also uses the meet-in-the-middle attack to show that straightforward "double-DES" encryption C = $E_{K2}(E_{K1}(P))$ has an effective keystrength of only 56 bits – no better than single DES – if sufficient quantities of plaintext and corresponding ciphertext are known.

As concerns about the security of DES continued to mount, the US National Institute of Standards and Technology (NIST, the new name for the NBS) began a search for a replacement. The result was the Advanced Encryption Standard, **AES**, officially approved in 2001. AES works with key lengths of 128, 192 and 256 bits. The algorithm is described in *[DR02]*, and is based on the Rijndael family of ciphers; the name Rijndael ("rain-dahl") is a combination of the authors' names.

Earlier non-DES ciphers include **IDEA**, the International Data Encryption Algorithm, described in *[LM91]*, and **Blowfish**, described in *[BS93]*. Both use 128-bit keys. The IDEA algorithm was patented; Blowfish was intentionally not patented. A successor to Blowfish is Twofish.

22.7.3 Cipher Modes

The simplest encryption "mode" for a block cipher is to encrypt each input block independently. That is, if P_i is the ith plaintext block, then the ith ciphertext block C_i is $E(P_i,K)$. This is known as electronic codebook or **ECB** mode.

ECB is vulnerable to known-plaintext attacks. Suppose the attacker knows that the message is of the following form, where the vertical lines are the block boundaries:

```
Bank Transfer  |  to MALLORY  |  Amount $1000
```

If the attacker also knows the ciphertext for "Amount $100,000", and is in a position to rewrite the message (or to inject a new message), then the attacker can combine the first two blocks above with the third $100,000 block to create a rather different message, without knowing the key. At http://en.wikipedia.org/wiki/Block_cipher_mode_of_operation there is a remarkable example of the failure of ECB to fail to effectively conceal an encrypted image.

As a result, ECB is seldom used. A common alternative is cipher block chaining or **CBC** mode. In this mode, each plaintext block is XORed with the previous ciphertext block before encrypting:

$C_i = E(K, C_{i-1} \text{ XOR } P_i)$

To decrypt, we use

$P_i = D(K, C_i) \text{ XOR } C_{i-1}$

If we stop here, this means that if two messages begin with several identical plaintext blocks, the encrypted messages will also begin with identical blocks. To prevent this, the first ciphertext block C_0 is a random string, known as the **initialization vector**, or IV. The plaintext is then taken to start with block P_1. The IV is sent in the clear, but contains no private information.

See exercise 5.0.

22.7.4 Stream Ciphers

Conceptually, a stream cipher encodes one byte at a time. The cipher generates a pseudorandom **keystream**. If K_i is the ith byte of the keystream, then the ith plaintext byte P_i is encrypted as $C_i = P_i$ XOR K_i. A stream cipher can be viewed as a special form of pseudorandom number generator or PRNG, though PRNGs used for numeric methods and simulations are seldom secure enough for cryptography.

Simple XORing of the plaintext with the keystream may not seem secure, but if the keystream is in fact truly random then this cipher is unbreakable: to an attacker, all possible plaintexts are equally likely. A truly random keystream means that the entire keystream must be shared ahead of time, and no portion may ever be reused; this cipher is known as the **one-time pad**.

Stream ciphers do not, by themselves, provide authenticity. An attacker can XOR something with the encrypted message to change it. For example, if the message is known to be

```
Transfer $ 02000 to Mallory
           ^^
```

then the attacker can XOR the two bytes over the "^" with '0' XOR '2', changing the character '0' to a '2' and the '2' to a '0'. The attacker does this to the encrypted stream, but the decrypted plaintext stream retains the change. Appending an authentication code such as HMAC, *22.6.1 Secure Hashes and Authentication*, prevents this.

Stream ciphers are, in theory, well suited to the encryption of data sent a single byte at a time, such as data from a telnet session. The ssh protocol (*22.10.1 SSH*), however, generally uses block ciphers; when it has to send a single byte it pads the block with random data.

22.7.4.1 RC4

The RC4 stream cipher was quite widely used. It was developed by Ron Rivest at RSA Security in 1987, but never formally published. The code was leaked, however, and so the internal details are widely available. "Unofficial" implementations are sometimes called **ARC4** or ARCFOUR, the "A" for "Alleged".

RC4 was designed for very fast implementation in software; to this end, all operations are on whole bytes. RC4 generates a keystream of pseudorandom bytes, each of which is XORed with the corresponding plaintext byte. The keystream is completely independent of the plaintext.

The key length can range from 5 up to 256 bytes. The unofficial protocol contains no explicit mechanism for incorporating an initialization vector, but an IV is well-nigh essential for stream ciphers; otherwise an identical keystream is generated each time. One simple approach is to create a session key by concatenating the IV and the secret RC4 key; alternatively (and perhaps more safely) the IV and the RC4 key can be XORed together.

RC4 was the basis for the ill-fated WEP Wi-Fi encryption, *22.7.7 Wi-Fi WEP Encryption Failure*, due in part to WEP's requirement that the 3-byte IV precede the 5-byte RC4 key. The flaw there did not affect other applications of RC4, but newer attacks have suggested that RC4 be phased out.

Internally, an RC4 implementation maintains the following state variables:

> An array S[] representing a permutation i→S[i] of all bytes
>
> two 8-bit integer indexes to S[], i and j

S[] is guaranteed to represent a permutation (*ie* is guaranteed to be a 1-1 map) because it is initialized to the identity and then all updates are transpositions (involving swapping S[i] and S[j]).

Initially, we set S[i] = i for all i, and then introduce 256 transpositions. This is known as the **key-scheduling algorithm**. In the following, `keylength` is the length of the key `key[]` in bytes.

```
J=0;
for I=0 to 255:
    J = J + S[I] + key[I mod keylength];
    swap S[I] and S[J]
```

As we will see in *22.7.7 Wi-Fi WEP Encryption Failure*, 256 transpositions is apparently not enough.

After initialization, `I` and `J` are reset to 0. Then, for each keystream byte needed, the following is executed, where `I` and `J` retain their values between calls:

```
I = (I+1) mod 256
J = J + S[I] mod 256
swap S[I] and S[J]
return S[ S[I] + S[J] mod 256 ]
```

For `I` = 1, the first byte returned is `S[ S[1] + S[S[1]] ]`.

22.7.5 Block-cipher-based stream ciphers

It is possible to create a stream cipher out of a block cipher. The first technique we will consider is **counter mode**, or CTR. The sender and receiver agree on an initial block, B, very similar to an initialization vector. The first block of the keystream, K_0, is simply the encrypted block E(B,K). The ith keystream block, for

i>0, is K_i = E(B+i,K), where "B+i" is the result of treating B as a long number and performing ordinary arithmetic.

Going from B+1 to B+(i+1) typically changes only one or two bits, but no significant vulnerability based on this has ever been found, and this form of counter mode is now widely believed to be as secure as the underlying block cipher.

A related technique is to generate the keystream by starting with a secret key K and taking a secure hash of each of the successive concatenations K⌒1, K⌒2, *etc*. The drawback to this approach is that many secure-hash functions have not been developed with this use in mind, and unforeseen vulnerabilities may exist.

The keystream calculated by counter mode is completely independent of the plaintext. This is not always the case. In **cipher feedback** mode, or CFB, we initialize C_0 to the initialization vector, and then define the keystream blocks K_i for i>0 as follows:

$K_i = E_K(C_{i-1})$

As usual, $C_i = P_i$ XOR K_i for i>0, so the right-hand side above is $E_K(P_{i-1}$ XOR $K_{i-1})$. The evolution of the keystream thus depends on earlier plaintext. Note that, by the time we need to use K_i, the earlier plaintext P_{i-1} has been entirely received.

Stream ciphers face an error-recovery problem, but we will ignore that here as we will assume the transport layer provides sufficient assurances that the ciphertext is received accurately.

22.7.6 Encryption and Authentication

If the ciphertext is modified in transit, most decryption algorithms will happily "decrypt" it anyway, though perhaps to gibberish; encryption provides no implicit validity check. Worse, it is sometimes possible for an attacker to modify the ciphertext so as to produce a meaningful, intentional change in the resultant plaintext. Examples of this appear above in *22.7.4 Stream Ciphers* and *22.7.3 Cipher Modes*; for CBC block ciphers see exercise 5.0.

It is therefore common practice to include HMAC authentication signatures (*22.6.1 Secure Hashes and Authentication*) with each encrypted message; the use of HMAC rather than a simple checksum ensures that an attacker cannot modify messages and have them still appear to be valid. HMAC may not identify the sender as positively as public-key digital signatures, *22.9.1.1 RSA and Digital Signatures*, but it *does* positively identify the sender as the same entity with whom the receiver negotiated the key. This is all that is needed.

Appending an HMAC signature does more than prevent garbled messages; there are attacks that, in the absence of such signatures, allow full decryption of messages (particularly if the attacker can create multiple ciphertexts and find out which ones the recipient was able to decrypt successfully). See *[SV02]* for an example.

There are three options for combining encryption with HMAC; for consistency with the literature we replace HMAC with the more general MAC (for Message Authentication Code):

- MAC-then-encrypt: calculate the MAC signature on the plaintext, append it, and encrypt the whole
- encrypt-and-MAC: calculate the MAC signature on the plaintext, encrypt the message, and append to that the MAC

- encrypt-then-MAC: encrypt the plaintext and calculate the MAC of the ciphertext; append the MAC to the ciphertext

These are analyzed in *[BN00]*, in which it is proven that **encrypt-then-MAC** in general satisfies some stronger cryptographic properties than the others, although these properties may hold for the others in special cases. Encrypt-then-MAC means that no decryption is attempted of a forged or spoofed message.

22.7.7 Wi-Fi WEP Encryption Failure

The WEP (Wired-Equivalent Privacy) mechanism was the first Wi-Fi encryption option, introduced in 1999; see *3.7.5 Wi-Fi Security*. It used RC4 encryption with either a 5-byte or 13-byte secret key; this was always appended to a 3-byte initialization vector that, of necessity, was sent in the clear. The RC4 session key was thus either 8 or 16 bytes; the shorter key was more common.

There is nothing inherently wrong with that strategy, but the designers of WEP made some design choices that were, in retrospect, infelicitous:

- the secret key was appended to the IV (rather than, say, XORed with it)
- the IV was only three bytes long
- a new RC4 keystream and new IV was used for each packet

The third choice above was perhaps the more serious mistake. One justification for this choice, however, was that otherwise lost packets (which are common in Wi-Fi) would represent gaps in the received stream, and the receiver might have trouble figuring out the size of the gap and thus how to decrypt the later arrivals.

A contributing issue is that the first byte of the plaintext – a header byte from the Wi-Fi packet – is essentially known. This first byte is generally from the Logical Link Control header, and contains the same value as the Ethernet Type field (see *3.7.4 Access Points*). All IP packets share the same value. Thus, by looking at the first byte of the encrypted packet, an attacker knows the first byte of the keystream.

A theoretical WEP vulnerability was published in [FMS01] that proved all-too-practical in the real world. We will let K[] denote the 8-byte key, with K[0],K[1],K[2] representing the three-byte initialization vector and K[i] for 3⩽i<8 representing the *secret* key. The attacker monitors the airwaves for IVs of a particular form; after about 60 of these are collected, the attacker can make an excellent guess at K[3]. Now a slightly different group of IVs is collected, allowing a guess at K[4] and so on. The bytes of the secret key thus fail sequentially. The attack requires about the same (modest) time for each key byte discovered, so a 16-byte total key length does not add much more security versus the 8-byte key.

Because the IV is only three bytes, the time needed to collect packets containing the necessary special IV values is modest.

We outline the attack on the first secret byte, K[3]. The IV's we are looking for have the form

⟨3,-1,X⟩

where -1 = 255 for 8-bit bytes; these are sometimes called **weak IVs**. One IV in 2^{16} is of this form, but a more careful analysis (which we omit) can increase the usable fraction of IVs substantially.

To see why these IVs are useful, we need to go back to the RC4 key-scheduling algorithm, *22.7.4.1 RC4*. We start with an example in which the first six bytes of the key is

```
K[] = [3,-1,5,4,-14,3]
```

The IV here is ⟨3,-1,5⟩.

Recall that the array S is initialized with S[i] = i; this represents S = S_0. At this point, I and J are also 0. We now run the following loop, introducing one transposition to S per iteration:

```
for I=0 to 255:
    J = J + S[I] + K[I mod keylength];
    swap S[I] and S[J]
```

The first value of `J`, when `I` = 0, is `K[0]` which is 3. After the first transposition, S is as follows, where the swapped values are in bold:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	1	2	**0**	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

For the next iteration, `I` is 1 and `J` is 3+1+(-1) = 3 again. After the swap, S is

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	**0**	2	**1**	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Next, `I` is 2 and `J` is 3+2+5 = 10. In general, if the IV were ⟨3,-1,X⟩, `J` would be 5+X.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	0	**10**	1	4	5	6	7	8	9	**2**	11	12	13	14	15	16	17	18	19	20

At this point, we have processed the three-byte IV; the next iteration involves the first secret byte of `K`. `I` is 3 and `J` becomes 10+1+`K[3]` = 15:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	0	10	**15**	4	5	6	7	8	9	2	11	12	13	14	**1**	16	17	18	19	20

Recall that the first byte returned in the keystream is `S[ S[1] + S[S[1]] ]`. At this point, that would be `S[0+3]` = 15.

From this value, 15, and the IV ⟨3,-1,5⟩, the attacker can calculate, by repeating the process above, that `K[3]` = 4.

If the IV were ⟨3,-1,X⟩, then, as noted above, in the `I`=2 step we would have `J` = 5+X, and then in the `I`=3 step we would have `J` = 6 + X + `K[3]`. If Y is the value of the first byte of the keystream, using `S[]` as of

this point, then `K[3]` = Y – X – 6 (assuming that X is not -5 or -4, so that `S[0]` and `S[1]` are not changed in step 3).

If none of `S[0]`, `S[1]` and `S[3]]` are changed in the remaining 252 iterations, the value we predict here after three iterations – *eg* 15 – would be the first byte of the actual keystream. Treating future selections of the value `J` as random, the probability that one iteration does *not* select `J` in the set {0,1,3} – and thus leaves these values alone – is 253/256. The probability that the remaining iterations do not change these values in `S[]` is thus about $(253/256)^{252} \simeq 5\%$

A 5% success rate is not terribly impressive, on the face of it. But 5% of the time we identify `K[3]` correctly, and the other 95% of the time the (incorrect) values we calculate for `K[3]` are uniformly, randomly distributed in the range 0 to 255. With 60 guesses, we expect about three correct guesses. The other 57 are spread about with, most likely, no more than two guesses for any one value. If we look for the value guessed most often, it is likely to be the true value of `K[3]`; increasing the number of ⟨3,-1,X⟩ IVs will increase the certainty here.

For the sake of completeness, in the next two iterations, `I` is 4 and 5 respectively, and `J` is 15+4+(-14) = 5 and 5+4+3=12 respectively. The corresponding contents of `S[]` are

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	0	10	15	**5**	**4**	6	7	8	9	2	11	12	13	14	1	16	17	18	19	20

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	0	10	15	5	**12**	6	7	8	9	2	11	**4**	13	14	1	16	17	18	19	20

Now that we know `K[3]`, the next step is to find `K[4]`. Here, we search the traffic for IVs of the form ⟨4,-1,X⟩, but the general strategy above still works.

The *[FMS01]* attack, as originally described, requires on average about 60 weak IVs for each secret key byte, and a weak IV turns up about every 2^{16} packets. Each key byte, however, requires a different IV, so any one IV has one chance per key byte to be a weak IV. To guess a key thus requires $60\times65536 \simeq 4$ million packets.

But we can do quite a bit better. In *[TWP07]*, the number of packets needed is only about 40,000 on average. At that point the attack is entirely practical.

This vulnerability was used in the attack on TJX Corp mentioned in the opening section.

22.8 Diffie-Hellman-Merkle Exchange

How do two parties agree on a secret key K? They can meet face-to-face, but that can be expensive or even not possible. We would like to be able to encrypt, for example, online shopping transactions, which often occur at the spur of the moment.

Diffie, Hellman and Merkle

Diffie and Hellman's paper *[DH76]* appeared in 1976, but they reference Merkle's later-published paper *[RM78]* as "submitted". In *[MH04]*, Hellman suggested that the name of the key-exchange algorithm here should bear the names of all three contributors. The algorithm was apparently discovered in 1974 by Malcolm Williamson at GCHQ, but Williamson's work was classified.

One approach is to use public-key encryption, below, but that came two years later. The Diffie-Hellman-Merkle approach, from *[DH76]* and *[RM78]*, allows private key exchange without public keys.

The goal here is for Alice and Bob to exchange information over a public network so that, at the end, Alice and Bob have determined a common key, and no eavesdropper can determine it without extraordinary computational effort.

As a warmup, we begin with a simple example. Alice and Bob each choose a number; say Alice chooses 7 and Bob chooses 9. Alice sends to Bob the value A = 3^7 = 2187, and Bob sends to Alice the value B = 3^9 = 19683. Alice then computes $B^7 = 3^{9\times7}$, and Bob computes $A^7 = 3^{7\times9}$. If they each use 32-bit arithmetic, they have each arrived at 2111105451 (the exact value is 1144561273430837494885949696427).

The catch here is that any intruder with a calculator can recover 7 from A and 9 from B, by factoring. But if we make the arithmetic a little more complicated, this becomes extremely difficult.

In the actual protocol, Alice and Bob agree, publicly, on a large prime p (typically several hundred digits) and a small value g. Alice then chooses a<p, and sends to Bob the value A = g^a mod p. Similarly, Bob chooses b<p and sends to Alice the value B = g^b mod p.

As before, Alice and Bob now compute B^a mod p and A^b mod p, which both equal $g^{a\times b}$ mod p. This becomes their shared key.

The question is whether an eavesdropper who knows A and B and g can compute a and b. This is the **discrete logarithm problem**, as in some mod-p sense a = $\log_g(A)$. It is indeed believed to be computationally intractable, provided that g is a so-called primitive root modulo p (meaning that the values g^1 mod p, g^2 mod p, ..., g^{p-1} mod p are all distinct). A naive attempt to compute a from A is to try each g^i for i<p until one finds a match; this is much too slow to be practical.

There is some work involved in finding a suitable primitive root g given p, but that can be public, and is usually not *too* time-consuming. However, p and g should be updated at regular intervals, and the same p (and g) should not be shared with other sites. Otherwise one may be vulnerable to the **logjam** attack described in *[ABDGHSTVWZ15]*. For a fixed prime p, it turns out that relatively fast discrete-logarithm computation is possible if an attacker is able to pre-compute a very large table. For 512-bit primes, the average logarithm-calculation time in *[ABDGHSTVWZ15]* was 90 seconds, once the table was constructed. Time for calculation of the table itself took almost 10^5 core-hours, though that could be compressed into as little as one week with highly parallel hardware. By comparison, the authors were able to factor 512-bit RSA keys in about 5,000 core-hours; see *22.9.1.2 Factoring RSA Keys*.

The authors of *[ABDGHSTVWZ15]* conjecture that the NSA has built such a table for some well-known (and commonly used) 1024-bit primes. However, for those with fewer computational resources, the logjam attack also includes strategies for forcing common server implementations to downgrade to the use of 512-bit primes using TLS implementation vulnerabilities; *cf* the POODLE attack (first sidebar at *22.10 SSH and TLS*).

The Diffie-Hellman-Merkle exchange is vulnerable to a **man-in-the-middle attack**, discussed in greater

detail in *22.9.3 Trust and the Man in the Middle*. If Mallory intercepts Alice's g^a and sends Bob his own $g^{a'}$, and similarly intercepts Bob's g^b and replaces it with his own $g^{b'}$, then the Alice–Mallory link will negotiate key $g^{a \times b'}$ and the Mallory-Bob link will negotiate $g^{a' \times b}$. Mallory then remains in the middle, intercepting, decryping, re-encrypting and forwarding each message between Alice and Bob. The usual fix is for Alice and Bob to *sign* their g^a and g^b; see *22.9.2 Forward Secrecy*.

22.8.1 Fast Arithmetic

It is important to note that the other arithmetic operations here – *eg* calculating g^a mod p – can be done in polynomial time with respect to the number of binary digits in p, sometimes called the **bit-length** of p. This is not completely obvious, as the naive approach of multipying out $g \times g \times \ldots \times g$, a times, takes O(p) steps when $a \simeq p$, which is exponential with respect to the bit-length of p. The trick is to use repeated squaring: to compute g^{41}, we note 41 = 101001 in binary, that is, $41 = 2^5 + 2^3 + 1$, and so

$$g^{41} = ((((g^2)^2)^2)^2)^2 \times ((g^2)^2)^2 \times g$$

A simple python3 implementation of this appears at *22.12 RSA Key Examples*.

It is perhaps also worth noting that even *finding* large primes is not obviously polynomial in the number of digits. We can, however, use fast probabilistic primality testing, and note that the Prime Number Theorem guarantees that the number of candidates we must test to find a prime near a given number n is O(log n).

22.9 Public-Key Encryption

In public-key encryption, each party has a public encryption key K_E and a private decryption key K_D. Alice can publish her K_E to the world. If Bob has Alice's K_E, he can encrypt a message with it and send it to her; only someone in possession of Alice's K_D can read it. The fundamental idea behind public-key encryption is that knowledge of K_E should not offer any meaningful information about K_D.

22.9.1 RSA

Public-key encryption was outlined in *[DH76]*, but the best-known technique is arguably the RSA algorithm of *[RSA78]*. (The RSA algorithm was discovered in 1973 by Clifford Cocks at GCHQ, but was classified.)

To construct RSA keys, one first finds two very large primes p and q, perhaps 1024 binary digits each. These primes should be chosen at random, by choosing random N's in the right range and then testing them for primality until success. Let $n = p \times q$.

It now follows from Fermat's little theorem that, for any integer m, $m^{(p-1)(q-1)} = 1$ mod n (it suffices to show $m^{(p-1)(q-1)} = 1$ mod p and $m^{(p-1)(q-1)} = 1$ mod q).

One then finds positive integers e and d so that $e \times d = 1$ mod (p-1)(q-1); given any e relatively prime to both p-1 and q-1 it is possible to find d using the Extended Euclidean Algorithm (a simple Python implementation appears below in *22.12 RSA Key Examples*). From the claim in the previous paragraph, we now know $m^{e \times d} = (m^e)^d = m$ mod p.

If we take m as a message (that is, as a bit-string of length less than the bit-length of n, rather than as an integer), we encrypt it as $c = m^e$ mod n. We can decrypt c and recover m by calculating c^d mod n = $m^{e \times d}$ mod n = m.

The public key is the pair ⟨n,e⟩; the private key is d.

> **Elliptic-curve cryptography**
>
> One of the concerns with RSA is that a faster factoring algorithm will someday make the encryption useless. A newer alternative is the use of *elliptic curves*; for example, the set of solutions modulo a large prime p of the equation $y^2 = x^3 + ax + b$. This set has a natural (but nonobvious) product operation completely unrelated to modulo-p multiplication, and, as with modulo-p arithmetic, finding n given $B = A^n$ appears to be quite difficult. Several cryptographic protocols based on elliptic curves have been proposed; see Wikipedia.

As with Diffie-Hellman-Merkle (*22.8.1 Fast Arithmetic*), the operations above can all be done in polynomial time with respect to the bit-lengths of p and q.

The theory behind the security of RSA is that, if one knows the public key, the only way to find d in practice is to factor n. And factoring is a hard problem, with a long history. There are much faster ways to factor than to try every candidate less than $\sqrt{n}$, but it is still believed to require, in general, close-to-exponential time with respect to the bit-length of n. See *22.9.1.2 Factoring RSA Keys* below.

RSA encryption is usually thousands of times slower than comparably secure shared-key ciphers. However, RSA needs only to be used to encrypt a secret key for a shared-key cipher; it is this latter key that actually protects the document.

For this reason, if a message is encrypted for multiple recipients, it is usually not much larger than if it were encrypted for only one: the additional space needed for each additional recipient is just the encrypted key for the shared-key cipher.

Similarly, for digital-signature purposes Alice doesn't have to encrypt the entire message with her private key; it is sufficient (and much faster) to encrypt a secure hash of the message.

If Alice and Bob want to negotiate a session key using public-key encryption, it is sufficient for Alice to know Bob's public key; Alice does not even need a public key. Alice can choose a session key, encrypt it with Bob's public key, and send it to Bob.

We will look at an actual calculation of an RSA key and then an encrypted message in *22.12 RSA Key Examples*.

22.9.1.1 RSA and Digital Signatures

RSA can also be used for strong digital signatures (as can any public-key system in which the roles of the encryption and decryption keys are symmetric, and can be reversed). Suppose, as above, Alice's public key is the pair ⟨n,e⟩ and her private key is the exponent d. If Alice wants to sign a message m to Bob, she simply encrypts it using the exponent d from her *private* key; that is, she computes $c = m^d \bmod n$. Bob, along with everyone else in the world, can then decrypt the resulting ciphertext c with Alice's *public* exponent e (that is, c^e). If this yields a valid message, then Alice (or someone in possession of her key) must have been the sender.

Generally, for signature purposes Alice will encrypt not her entire message but simply a secure hash of it, and then send both the message and the RSA-encrypted hash. Bob decrypts the encrypted hash, and the signature checks out if the result matches the hash of the corresponding message.

The advantage of a public-key signature over an HMAC signature is that the former uses a key with long-term persistence, and can be used to identify the *individual* who sent the message. The drawback is that HMAC signatures are much faster.

It is common for Alice to sign her message to Bob and then encrypt the message plus signature – perhaps with Bob's public key – before sending it all to Bob.

Just as Alice probably prefers not to sign undated documents with her handwritten signature, she will likely prefer to apply her digital signature only to documents containing a timestamp. This helps deter replay attacks.

22.9.1.2 Factoring RSA Keys

The security of RSA depends largely on the difficulty in factoring the key modulus n = pq (though it is theoretically possible that an RSA vulnerability exists that does not entail factoring). As of 2015, the factoring algorithm that appears to be the fastest for larger keys is the so-called number-field sieve; an early version of this technique was introduced in *[JP88]*. A heuristic estimate of the time needed to factor a number n via this algorithm is $\exp(1.923\times\log(n)^{1/3}\times\log(\log(n))^{2/3})$, where $\exp(x) = e^x$ and log(x) is the natural logarithm. To a first approximation this is $\exp(k\times L^{1/3})$, where L is the bit-length of N. This is sub-exponential (because of the exponent 1/3) but still extremely slow.

Side Channels

Those placing serious reliance on the relative security levels outlined here should also be aware of the existence of so-called side-channel attacks, *eg* making use of electromagnetic leakage from a computer to infer a key.

The first factorization of an RSA modulus with 512 bits occurred in 1999 and was published in *[CDLMR00]*; this was part of the RSA factoring challenge. It took seven calendar months, using up to 300 processors in parallel. Fifteen years later, *[ABDGHSTVWZ15]* reported being able to do such a factorization in eight days on a 24-core machine (~5,000 core-hours), or in seven hours using 1800 cores. Some of the speedup is due to software implementation improvements, but most is due to faster hardware.

If we normalize the number-field-sieve factoring time of a 512-bit n to 1, we get the following table of estimated relative costs for factoring larger n:

key length in bits	relative factoring time
512	1
1024	8,000,000
1536	8×10^{11}
2048	8×10^{15}
3072	3×10^{22}
4096	8×10^{27}

From this we might conclude that 1024-bit keys are potentially breakable by a *very* determined and well-funded adversary, while the 2048-bit key length appears to be much safer. In 2011 NIST's recommended key length for RSA encryption increased to 2048 bits (publication 800-131A, Table 6). This is quite a bit larger than the 128-to-256-bit recommendation for shared-key ciphers, but RSA factoring attacks are *much* faster than trying all keys by brute force.

Note that some parts of the factoring algorithm are highly parallelizable while other parts are less so; relative factoring times when using highly parallel hardware may therefore differ quite a bit. See also *22.12.1 Breaking the key*.

22.9.2 Forward Secrecy

Suppose Alice and Bob exchange a shared-key-cipher session key K_S using their RSA keys. Later, their RSA keys are compromised. If the attacker has retained Alice and Bob's prior communications, the attacker can go back and decrypt K_S, and then use K_S to decrypt the entire session protected by K_S.

This is not true, however, if Alice and Bob had used Diffie-Hellman-Merkle key exchange. In that case, there is no encryption used in the process of negotiating K_S, so no later encryption compromise can reveal K_S.

This property is called **forward secrecy**, or, sometimes, *perfect* forward secrecy (other times, perfect forward secrecy adds the further requirement that the compromise of any other session key negotiated by Alice and Bob does not reveal information about K_S).

The advantage of public-key encryption, however, is that Alice can sign the key K_S she sends to Bob. Assuming Bob is confident he has Alice's real public key, a man-in-the-middle attack (*22.9.3 Trust and the Man in the Middle*) becomes impossible.

It is now common for public-key encryption to be used to sign all the transactions that are part of the Diffie-Hellman-Merkle exchange. When this is done, Alice and Bob gain both forward secrecy *and* protection from man-in-the-middle attacks.

22.9.3 Trust and the Man in the Middle

Suppose Alice wants to send Bob a message. Where does she find Bob's public key E_B?

If Alice goes to a public directory that is not completely secure and trustworthy, she may find a key that in fact belongs to Mallory instead. Alice may now fall victim to a **man-in-the-middle** attack, like that at the end of *22.8 Diffie-Hellman-Merkle Exchange*:

- Alice encrypts Bob's message using Mallory's public key E_{Mal}, thinking it is Bob's
- Mallory intercepts and decrypts the message, using his own decryption key D_{Mal}
- Mallory re-encrypts the message with Bob's real public key E_{Bob}
- Bob decrypts the message with D_{Bob}

Despite Mallory's inability to break RSA directly, he has read (and may even modify) Alice's message.

Alice can, of course, get Bob's key directly from Bob. Of course, if Alice met Bob, the two could also exchange a key for a shared-key cipher.

Wi-Fi in the Middle

One of the easiest settings at which to launch a man-in-the-middle attack is a public Wi-Fi hotspot, either as the hotspot owner or by setting up a rogue access point to which some customers mistakenly connect.

We now come to the mysterious world of trust. Alice might trust Charlie for Bob's key, but not Dave. Alice doesn't have to meet Charlie to get Bob's key; all she needs is

- a trusted copy of Charlie's public key
- a copy of Bob's public key, together with Bob's name, *signed by* Charlie

At the same time, Alice might trust Dave but not Charlie for Evan's key. And Bob might not trust Charlie *or* Dave for Alice's key. Mathematically, the trust relationship is neither symmetric nor transitive. To handle the possibility that trust might erode with time, signed keys often have an expiration date.

At the small scale, **key-signing parties** are sometimes held in which participants exchange some keys directly and others indirectly through signing. This approach is sometimes known as the **web of trust**. At the large scale, **certificate authorities** (*22.10.2.1 Certificate Authorities*) are entities built into the TLS framework (*22.10.2 TLS*) that verify that a website's public key is as claimed; you are implicitly trusting these certificate authorities if your browser vendor trusts them. Both the web of trust and certificate authorities are examples of "public-key infrastructure" or **PKI**, which is, broadly, any mechanism for reliably tying public keys to their owners. For applications of public-key encryption that manage to avoid the need for PKI, by use of **cryptographically generated addresses**, see *8.6.4 Security and Neighbor Discovery* and the discussion of .onion addresses at the end of *7.8 DNS*.

22.9.4 End-to-End Encryption

Many communications applications are based on the model of encryption from each end-user to the central server. For example, Alice and Bob might both use https (based on TLS, *22.10.2 TLS*) to encrypt their interactions with their email provider. This means Alice and Bob are now trusting that provider, who decrypts messages from Alice, stores them, and re-encrypts them when delivering them to Bob.

This model does protect Alice and Bob from Internet eavesdroppers who have not breached the security of the email provider. However, it also allows government authorities to order the email provider to turn over Alice and Bob's correspondence.

If Alice and Bob do not wish to trust an intermediary, or their (or someone else's) government, they need to implement **end-to-end** encryption. That is, Alice and Bob must negotiate a key, use that key to encrypt messages between them, and not divulge the key to anyone else. This is quite a bit more work for Alice and Bob, and even more complicated if Alice wishes to use end-to-end encryption with a large number of correspondents.

Of course, even with end-to-end encryption Alice may still be compelled by subpoena to turn over her correspondence with Bob, but that is a different matter. Alice's private key may also be seized under a search warrant. It is common (though not universal) to protect private keys with a password; this is good practice, but protecting a key having an effective length of 256 bits with a password having an effective length of 32 bits leaves something to be desired. The mechanisms of *22.6.2 Password Hashes* provide only limited relief. The mechanism of *22.9.2 Forward Secrecy* may be more useful here, assuming Alice can communicate to Bob that her previous key is now compromised; see also *22.10.2.3 Certificate revocation*.

22.10 SSH and TLS

We now look at two encryption mechanisms in popular use on the Internet. The first is the Secure Shell, **SSH**, used to allow login to remote systems, remote command execution, file transfer and even some forms of VPN tunneling. Public-key-encryption is optional; if it is used, the public keys are generally transported manually.

POODLE

The 2014 POODLE attack was based in part on a long-known flaw in SSL 3.0. But SSL 3.0 is fifteen years obsolete (though it was not officially deprecated until 2015); the other flaw was a broken version-negotiation implementation in which the disruption of a few packets by an attacker could force both client and server to settle on SSL 3.0 even when both supported the latest TLS version. See *22.10.2.4 TLS Connection Setup*.

The second example is the Transport Layer Security protocol, **TLS**, which is a successor of the Secure Sockets Layer, or **SSL**. Many people still refer to TLS by the SSL name even though TLS replaced SSL in 1999, though, to be fair, TLS is effectively an update of SSL, and TLS v1.0 could easily have been named SSL v4.0. TLS is used to encrypt web traffic for the HTTP Secure protocol, **https**; it is also used to encrypt traffic for several other applications and *can* be used, with appropriate programming, for any application.

If Alice wants to contact a server S using either SSH or TLS, at some point she will have to trust a public key claimed to be from S. A common approach with SSH is for Alice to accept the key on faith the first time it is presented, but then to save it for all future verifications. Under TLS, the key from S that Alice receives will have been signed by a certificate authority (*22.10.2.1 Certificate Authorities*); Alice presumably trusts this certificate authority. (SSH does now support certificate authorities too, but their use with SSH seems not yet to be common.)

Both SSH and TLS eventually end up negotiating a shared-secret session key, which is then used for most of the actual data encryption.

22.10.1 SSH

The SSH protocol establishes an encrypted communications channel between two hosts, after establishing the identities of each endpoint. Its primary use is to enable secure remote-command execution with input and output via the secure channel; this includes the remote execution of an interactive shell, which is in effect a telnet-style terminal login with encryption. The companion program `scp` uses the SSH protocol to implement secure file transfer. Finally, ssh supports secure port forwarding from one machine (and port) to another; unrelated applications can then connect to one machine and find themselves securely talking to another. The current version of the SSH protocol is 2.0, and is defined in **RFC 4251**. The authentication and transport sub-protocols are defined in **RFC 4252** and **RFC 4253** respectively.

One of the first steps in an SSH connection is for the endpoints to negotiate which secret-key cipher (and mode) to use. Support of the following ciphers is "recommended" and there is a much longer list of "optional" ciphers (which include RC4 and Blowfish); the table below includes those added by **RFC 4344**:

cipher	modes	nominal keylength
3DES	CBC, CTR	168 bits
AES	CBC, CTR	192 bits
AES	CTR	128 bits
AES	CTR	256 bits

SSH supports a special name format for including new ciphers for local use.

The SSH protocol also allows the endpoints to negotiate a public-key-encryption mechanism, a secure-hash function, and even a key-exchange algorithm although only minor variants of Diffie-Hellman-Merkle key exchange are implemented.

If Alice wishes to connect to a server S, the server clearly wants to verify Alice's identity, either through a password or some other means. But it is just as important for Alice to verify the identity of S, to prevent man-in-the-middle attacks and the possibility that an attacker masquerading as S is simply collecting Alice's password for later use.

The SSH protocol is not designed to minimize the number of round-trip packet exchanges, making SSH connection setup quite a bit slower than TLS connection setup (*22.10.2.4 TLS Connection Setup*). A connection may involve quite a few round trips to get started: the TCP three-way handshake, the protocol version exchange, the "Key Exchange Init" exchange, the actual Diffie-Hellman-Merkle key exchange, the "NewKeys" exchange, and the "Service Request" exchange (all but the first are documented in **RFC 4253**). Still, multi-second delays can usually be reduced through performance tuning; enabling diagnostic output (*eg* with the `-v` option) is often helpful. A common delay culprit is a server-side DNS lookup of the client, fixable with a server-side configuration setting of `UseDNS no` or the equivalent.

In the following subsections we focus on the "common" SSH configuration and ignore some advanced options.

22.10.1.1 Server Authentication

To this end, one of the first steps in SSH connection negotiation is for the server to send the public half of its **host key** to Alice. Alice verifies this key, which is typically in her `known_hosts` file. Alice also asks S to sign something with its host key. If Alice can then decrypt this with the public host key of S, she is confident that she is talking to the real S.

If this is Alice's first attempt to connect to S, however, she should get a message like the one below:

> The authenticity of host 'S (10.2.5.1)' can't be established.
>
> RSA key fingerprint is da:2e:e3:94:84:6b:bf:6d:2f:e4:c3:76:68:72:a5:a0.
>
> Are you sure you want to continue connecting (yes/no)?

If Alice is cautious, she may contact the administrator of S and verify the key fingerprint. More likely, she will simply choose "yes", in which case the host key of S will be added to her own `known_hosts` file; this latter strategy is sometimes referred to as **trust on first use**.

If she later re-connects to S after the host key of S has been changed, she will get a rather more dire message, and, under the default configuration, she will not be allowed to continue until she manually removes the old, now-incorrect, host key for S from her `known_hosts` file.

See also the ARP-spoofing scenario at *7.9.2 ARP Security*, and the comment there about how this applies to SSH.

SSH v2.0 now also supports a certificate-authority mechanism for verifying server host keys, replacing the known_hosts file.

22.10.1.2 Key Exchange

The next step is for Alice's computer and S to negotiate a session key. After the cipher and key-exchange algorithms are negotiated, Alice's computer and S use the chosen key-exchange algorithm to agree on a session key for the chosen cipher.

The two directions of the connection generally get different session keys. They can even use different encryption algorithms.

At this point the channel is encrypted.

22.10.1.3 Client Authentication

The server now asks Alice to authenticate herself. If password authentication is used, Alice types in her password and it and her username are sent to the server, over the now-encrypted connection. This does expose the server to brute-force password-guessing attacks, and is not infrequently disabled.

Alice may also have set up **RSA authentication** (other types of public-key authentication are also possible). For this, Alice must create a public/private key pair (often in files id_rsa.pub and id_rsa), and the public key must have been previously installed on S. On Unix-based systems it is often installed on S in the authorized_keys file in the .ssh subdirectory of Alice's home directory. If Alice now sends a message to S signed by her private key, S is in a position to verify the signature. If this succeeds, Alice can log in without supplying a password to S.

It is common though not universal practice for Alice's private-key file (on her own computer) to be protected by a password. If this is the case, Alice will need to supply that password, but it is used only on Alice's end of the connection. (The default password hash is MD5, which may not be a good choice; see *22.6.2 Password Hashes*.)

Public-key authentication is tried before password authentication. If Alice has created a public key, it is likely to be tried even if she has not copied it to S.

If S is set up to *require* public-key authentication, Alice may not have any direct way to install her public key on S herself, and may need the cooperation of the administrators of S. It is possible that the latter will send Alice a public/private keypair chosen by them specifically for S, rather than allowing Alice to choose her own keypair. The standard SSH user configuration does support different private keys for different servers.

In several command-line implementations of ssh, the various stages of authentication can be observed from the client side by using the ssh or slogin command with the -v option.

22.10.1.4 The Session

Once an SSH connection has started, a new session key is periodically negotiated. **RFC 4253** recommends this after one hour or after 1 GB of data is exchanged.

Data is then sent in packets generally with size a multiple of the block size of the cipher. If not enough data is available, *eg* because only a single keystroke (byte) is being sent, the packet is padded with random data

as needed. *Every* packet is required to be padded with at least four bytes of random data to thwart attacks based on known plaintext/ciphertext pairs. Included in the *encrypted* part of the packet is a byte indicating the length of the padding.

22.10.2 TLS

Transport Layer Security, or TLS, is an IETF extension of the Secure Socket Layer (SSL) protocol originally developed by Netscape Communications. SSL went through published versions 2.0 and 3.0; the latter was introduced in 1996 and was officially deprecated by **RFC 7568** in 2015 (see the POODLE sidebar above at *22.10 SSH and TLS*). TLS 1.0 was introduced in 1999, as the IETF took ownership of the specification from Netscape. The current version of TLS is 1.2, specified in **RFC 5246** (plus updates listed there). A draft of TLS 1.3 is in progress (2018); see draft-ietf-tls-tls13.

The original and still primary role for TLS is encrypting web connections and verifying for the client the authenticity of the server. TLS can, however, be embedded in any network application.

Unlike SSH, *client* authentication, while possible, is not common; web servers often have no pre-existing relationship with the client. Also unlike SSH, the public-key mechanisms are all based on established **certificate authorities**, or CAs, whereas the most common way for an SSH server's host key to end up on a client is for it to have been accepted by the user during the first connection. Browsers (and other TLS applications as necessary) have embedded lists of certificate authorities, known as **trust stores**, trusted by the browser vendor. SSH requires no such centralized trust.

If Bob wishes to use TLS on his web server S_{Bob}, he must first arrange for a certificate authority, say CA_1, to sign his **certificate**. A certificate contains the full DNS name of S_{Bob}, say bob.int, a public key K_S used by S_{Bob}, and also an expiration date. The use of the ITU-T X.509 certificate format is common.

The `.int` domain

For Bob to actually have a domain name in the `.int` top-level domain, as in the example here, Bob's organization must be established by international treaty.

Now imagine that Alice connect to Bob's server S_{Bob} using TLS. Early in the process S_{Bob} will send her its signed certificate, claiming public key K_S. Alice's browser will note that the certificate is signed by CA_1, and will look up CA_1 on its list of trusted certificate authorities. If found, the next step is for the browser to use CA_1's public key, also on the list, to verify the signature on the certificate S_{Bob} sent.

If everything checks out, Alice's browser now knows that CA_1 certifies bob.int has public key K_S. As S_{Bob} has presented this key K_S, and is able to verify that it possesses the matching private key, this is proof that S_{Bob} is legitimately the server with domain name bob.int.

Assuming, of course, that CA_1 is correct.

As with SSH, once Alice has accepted the public key K_S of S_{Bob}, a secret-key cipher is negotiated and the remainder of the exchange is encrypted.

22.10.2.1 Certificate Authorities

A **certificate authority**, or CA, is just an entity in the business of signing certificates: messages that include a name S_{Bob} and a verified public key K_S. The purpose of the certificate authority is to prevent man-in-the-middle attacks (*22.9.3 Trust and the Man in the Middle*); Alice wants to be sure she is not really connected to S_{Bad} instead of to S_{Bob}.

What the CA is actually signing is that the particular public key K_S belongs to domain bob.int, just like Charlie might sign Bob's public key on an individual basis. The difference here is that the CA is likely to be a large organization, and the CA's public key is likely to be embedded in the network application software somewhere. (Real CA's usually have a two-layer key signing arrangement, in which Bob's public key would be signed by one of the CA's subsidiary keys, but the effect is the same.)

A certificate specific for bob.int will not work for **www**.bob.int, even though both DNS names might direct to the same server S_{Bob}. If S_{Bob} presents a certificate for bob.int to a browser that has arrived at S_{Bob} via the url http://www.bob.int, the user should see a warning. It is straight forward, however, for the server either to support two certificates, or (if the certificate authority supports this) a single certificate for www.bob.int with a Subject Alternative Name also entered on the certificate for bob.int alone. It is also possible to obtain "wildcarded" certificates for *.bob.int (though this does not match bob.int). **RFC 6125**, §7.2, recommends against wildcard certificates.

> **Firefox Certificates**
>
> As of this writing, certificate authorities for the popular Firefox browser are found in Preferences → Advanced → View Certificates. It is an interesting list, if only because, for such highly trusted organizations, few people have heard of more than a handful of them. The policy for becoming a Firefox CA is here.

Popular browsers all use preset lists of CAs provided by (and presumably trusted by) either the browser vendor or the host operating-system vendor. **If** Alice is also willing to trust these CAs, she can feel comfortable using the key she receives to send private messages to Bob. That "if", however, is sometimes controversial. Just because Alice trusts her browser and operating system (*eg* not to contain malware), that does *not* automatically imply that Alice should trust these vendors' judgment when it comes to CAs.

On the face of it, Bob's certificate authority CA_1 is just signing that domain name bob.int has public key K_S. This isn't *quite* the same, however, as attesting that the certificate-signing request actually came from Bob. All depends on how thorough CA_1 is in checking the identity of its customer, and since those customers typically choose the least expensive CA, there is sometimes an incentive to cut corners.

A certificate authority's failure to verify the identity of the party making the certificate-signing request can enable a man-in-the-middle attack (*22.9.3 Trust and the Man in the Middle*). Suppose, for example, that Mallory is able to obtain a signed certificate from another certificate authority CA_2 linking bob.int to key K_{bad} controlled by Mallory. If Mallory can now position his server S_{Bad} in the middle between Alice and S_{Bob},

Alice —— S_{Bad} —— S_{Bob}

then Alice can be tricked into connecting to S_{Bad} instead of S_{Bob}. Alice will request a certificate, but from S_{Bad} instead of S_{Bob}, and get Mallory's from CA_2 instead of Bob's actual certificate from CA_1. Mallory opens a second connection from S_{Bad} to S_{Bob} (this is easy, as Bob makes no attempt to verify Alice's

identity), and forwards information from one connection to the other. As far as TLS is concerned everything checks out. From Alice's perspective, Mallory's false certificate vouches for the key K_{bad} of S_{Bad}, CA_2 has signed this certificate, and CA_2 is trusted by Alice's browser. There is no "search" at any point through the other CAs to see if any of them have any contrary information about S_{Bob}. In fact, there is not necessarily even contact with CA_2, though see *22.10.2.3 Certificate revocation* below.

If the certificate authority CA_1 were also the domain registrar with whom Bob registered the DNS name bob.int, it would be especially well-positioned to verify that Bob is really the owner of the bob.int domain. But this is not generally the case. Quite often, the CA's primary verification method is to send an email to, say, bob@bob.int (or perhaps to the DNS administrative contact listed in the WHOIS database for domain bob.int). If someone responds to this email, it is assumed they must be legitimately part of the bob.int domain. Note, however, that in the man-in-the-middle attack above, it does not matter what CA_1 does; what matters is CA_2's verification policy.

If Alice is very careful, she may click on the "lock" icon in her browser and see that the certificate authority signing her connection to S_{Bad} is CA_2 rather than CA_1. But if Alice has a secure way of finding Bob's "real" certificate authority, she might as well use it to find Bob's key K_S. As she often does not, this is of limited utility in practice.

The second certificate authority CA_2 might be a legitimate certificate authority that has been tricked, coerced or bribed into signing Mallory's certificate. Alternatively, it might be Mallory's own creation, inserted by Mallory into Alice's browser through some other vulnerability.

Superfish

The "Superfish" vulnerability of 2015 involved an operating-system module (based on LSP, WFP or, in principle, iptables) that could intercept all browser TLS connections (thus acting as the man in the middle), together with software that could act as a certificate authority, generating new certificates (like Mallory's from CA_2) on the fly. The apparent goal was to inject advertising into TLS-secured connections.

Mallory's machinations here do require both the man-in-the-middle attack and the bad certificate. If Alice is able to establish a direct connection with S_{Bob}, then the latter will send its true key K_S signed by CA_1.

As another attack, Mallory might obtain a certificate for b0b.int and hope Alice doesn't notice the spelling difference between B0B and BOB. When this is done, Mallory often also sends Alice a disguised link to b0b.int in the hope she will click on it. Unicode domain names make this even easier, as Unicode provides many character pairs that are different but which look identical.

Extended-Validation certificates were introduced in 2007 as a way of providing greater assurances that the certificate issued to bob.int was in fact generated by a request from Bob himself; Mallory should in theory have a much harder time obtaining an EV certificate for bob.int from CA_2. Browsers that have secured a TLS connection using an EV certificate typically add the name of the domain owner, highlighted in green and/or with a green padlock icon, to the address bar. Financial institutions often use EV certificates. So does mozila.org. Of course, if Alice does not know Bob is using an EV certificate, she can still be tricked by Mallory as above.

22.10.2.2 Certificate pinning

Another strategy intended as a more direct prevention of man-in-the-middle attacks is **certificate pinning**. Conceptually, Alice (or her browser) makes a note the first time she connects to S_{Bob} that the certificate authority is CA_1 or that Bob's public key is K_S, or both. Future connections to S_{Bob} must match at least one of these credentials. This form of pinning depends on Alice's having reached the real S_{Bob} on the first connection, sometimes called "trust on first use". A similar trust-on-first-use strategy is often (though not always) used with SSH, *22.10.1.1 Server Authentication*.

In the pinning protocol described in **RFC 7469**, it is S_{Bob} that initiates the pinning by including a "pin directive" in its initial HTTP connection. This requests Alice's browser to pin the desired certificates, though the pinned correspondence between a site and its keys is always maintained at the browser side. The pin directive also specifies which keys (K_S or CA_1 or both) are pinned, and for how long.

If S_{Bob} sends a pin directive for K_S, then Alice's browser remembers K_S, and any new certificate at S_{Bob} will be a mismatch. If S_{Bob} pins CA_1, then S_{Bob} can have CA_1 issue a new key and it will still pass pin-validation. If CA_1 is later compromised, though, S_{Bob} is not protected against any new rogue certificates it issues. (This is generally a minor concern, as CA_1 compromise will always mean that *new* visitors to S_{Bob} will be vulnerable to man-in-the-middle attacks.)

Bob can also obtain, along with K_S, a backup certificate $K_{S\text{-backup}}$, and have S_{Bob}'s pin directives pin both of these. Then, if K_S is compromised, $K_{S\text{-backup}}$ is ready to go. Hopefully, key compromise is a rare event, so there is a very small chance that both K_S and $K_{S\text{-backup}}$ are compromised within the pin's lifetime. Key compromise pretty much follows from any server compromise, however, and if intruders break in, $K_{S\text{-backup}}$ cannot be put into production until Bob is sure the site is secure. That may take some time.

If K_S and $K_{S\text{-backup}}$ *are* both compromised, and CA_1 wasn't pinned, S_{Bob} may simply become inaccessible to returning visitors. Automatic unpinning of revoked certificates would help, but certificate revocation (see following section) has its own difficulties. The user may have an option to disable pin validation for this particular site, but it's not supposed to be as simple as clicking "ok", and in any event if the site is your bank you may be loath to do this.

22.10.2.3 Certificate revocation

Suppose the key r underlying the certificate for bob.int has been compromised, and Mallory has the private key. Then, if Mallory can trick Alice into connecting to S_{Bad} instead of S_{Bob}, the original CA_1 will validate the connection. S_{Bad} can prove to Alice that it has the secret key corresponding to K_S, and all the certificate does is to attest that bob.int has key K_S. Mallory's deception works even if Bob noticed the compromise and updated S_{Bob}'s key to K_2; the point is that Mallory still has the original key, and CA_1's certificate attesting to that original key is still valid.

There is a mechanism by which a certificate can be revoked. Revocation information, however, must be kept at some central directory; a server can continue to serve up a revoked certificate and unless the clients actively check, they will be none the wiser. This is the reason certificates have **expiration dates**.

The original revocation mechanism was the global **certificate revocation list**. A newer alternative is the Online Certificate Status Protocol, **OCSP**, described in **RFC 6960**. If Alice receives a certificate signed by CA_1, she can send the serial number of the certificate to a designated "OCSP responder" run by or on behalf of CA_1. If the certificate is still valid, the responder site will return a signed confirmation of that.

Of course, an eavesdropper watching Alice's traffic arriving at the OCSP responder – and the OCSP responder itself – now knows that Alice is visiting bob.int. An eavedropper closer to Alice, however, knows that anyway.

More seriously, someone running a man-in-the-middle attack close to Alice can probably intercept and block Alice's OCSP request. If Alice receives no response, what should she do? Maybe there is a problem, but maybe the responder site is simply down or the Internet is simply slow. Most browsers that actually do revocation checks assume the latter – known as **soft fail** – making revocation checks of dubious value. The alternative of refusing to allow access to the original site – **hard fail** – leads to many false positives.

As of 2016, there is no generally accepted solution. One minimal approach, in the event of OCSP soft fail, is to use hard fail with EV certificates, or at least to downgrade EV certificates to ordinary ones upon OCSP non-response. Another approach is **OCSP stapling**, in which the server site periodically (perhaps daily) requests a signed and dated update from its CA's OCSP server indicating that the site's certificate is still valid. This OCSP report is then "stapled" to the `ServerHello` message (below), if requested by the client. This allows the client to verify that the certificate was still valid quite recently.

Of course, if the client *is* the victim of a man-in-the-middle attack then the (fake) server will not staple an OCSP validity report, and the client must fall back to the regular OCSP lookup process. But this case can be expected to be infrequent, making a hard fail after OCSP non-response a reasonable option.

Google's Chrome browser implements **CRLSets** in lieu of checking OCSP servers, described here. This is a list of revoked certificates downloaded regularly to the browser. Unfortunately, the full list is much too large, so CRLSets are limited to emergency revocations and certificate revocations due to key compromise; even then the list is not complete.

22.10.2.4 TLS Connection Setup

The typical TLS client-side user is interested in viewing a web page as quickly as possible, placing a premium on rapid negotiation of the TLS connection. If sites were to load noticeably more slowly when encryption was used, encryption might not be used routinely. As a result, the connection-setup process, known as the TLS **handshake protocol**, is designed to complete in two RTTs. The goals of the handshake protocol are to agree on the encryption and authentication mechanisms to be used, to provide authentication as necessary, and to negotiate the encryption keys. Here is an outline of a typical exchange, with some options omitted:

client (Alice)	server (Bob)
ClientHello • preferred TLS version • session ID (empty for new connections) • list of ciphers acceptable to client • start of key exchange (maybe)	
	ServerHello • chosen version • chosen cipher • server certificate • server key exchange (optional)
(client now knows the key) optional client certificate optional other data (may be encrypted) Finished message	
	Finished message (encrypted)

The client initiates the connection by sending its ClientHello message, which contains its preferred TLS/SSL-protocol version number, a session identifier, some random bytes (a cryptographic nonce), and a list of cipher *suites*: tuples consisting of a key-exchange algorithm, a shared-key encryption algorithm and a secure-hash algorithm. The list of cipher suites is ranked by the client according to preference.

Choosing your cipher

In theory, any TLS client can choose which cipher suites it will accept. In practice, for most software applications this is non-trivial. In the Firefox browser, see about:config, under security.ssl3 (yes, ssl3 is in theory not the same as tls).

The server then responds with its ServerHello. This contains the final protocol-version number, generally the maximum of the version proposed by the client and the highest version supported by the server. The ServerHello also contains the server's choice of a cipher suite from the client's list, and some more random bytes (the server's nonce). The server also sends its certificate, if requested (which it almost always is). The certificate is considered to be a separate "message" at the TLS protocol level, not part of the ServerHello, but everything is sent together.

Having received the server's certificate, the client application's next step is to validate this certificate, by checking its signature against the client's list of trusted certificate authorities, the client's "trust store". This step does not involve any network communication. While most operating systems maintain a list of generally trusted certificates, the client application can trust all, some or none of this list; the client can also load application-specific certificates. If the certificate does *not* check out, the client is free to continue the connection, perhaps pausing to add the server certificate to its trust store (hopefully after user interaction confirming this).

The client then responds with its key-exchange response, and its own certificate if applicable (which it

seldom is). It immediately follows that with its `Finished` message, the first message that is encrypted with the just-negotiated cipher suite. The server then replies with *its* encrypted `Finished` message, and encrypted application-layer communication can then begin.

Most browsers allow the user to click on the padlock icon for the TLS-secured connection to find out what cipher suite was actually agreed upon.

A client starting a brand-new connection leaves the Session ID field empty in the `ClientHello` message. However, if a client wishes to resume a previous session, it includes here the Session ID from that previous session. The server must, of course, also recognize that session.

Once upon a time, some broken TLS servers failed to respond properly if a client proposed a version number greater than what the server supported; the server would close the connection instead of returning a lower version number. As a result, many clients were programmed to try again with the next-lower version in the event of *any* connection-setup failure. This meant that an attacker could force a client to propose an obsolete version (*eg* SSL 3.0) simply by interrupting earlier connection-setup attempts, perhaps with a TCP RST packet. A consequence was the POODLE vulnerability mentioned in the sidebar in *22.10 SSH and TLS*.

An application's choice of TLS (versus unencrypted communication) is often signaled by the use of a special port number; for example, the standard http port is 80 while the standard https port is 443. Alternatively, there may be some initial negotiation; for example, in the SMTP email protocol (**RFC 5321**) it is common for the client to connect to the server on the standard port 25 without encryption, but then to negotiate the use of TLS using the STARTTLS extension to SMTP (**RFC 3207**). This adds multiple extra RTTs, but for non-interactive protocols this is usually of only minor concern.

22.10.2.4.1 TLS key exchange

Perhaps the most important part of the TLS handshake is to negotiate the encyption keys. The keys themselves are all derived from the TLS **master secret**. The key derivations are done in deterministic ways (*eg* using secure hashes, or a stream-cipher algorithm), and can be done by either side once the master secret is known.

In all cases, the client should know the master secret after receiving the `ServerHello` and related packets. At this point – after one RTT – the client can begin sending encrypted messages to the server. At this point, the client can even begin sending encrypted data, although it is possible that the client is not yet fully authenticated to the server.

Negotiation of the master secret depends on the cipher suite chosen. If RSA is being used for key negotiation, then the client chooses a random **pre-master secret**. This is sent to the server in the third leg of the exchange, after the client has received the server's certificate in the `ServerHello` stage. The pre-master secret is encrypted with the public RSA key from the server's certificate, thus conveying it securely to the server. Both sides calculate the master secret from the pre-master secret and both sides' cryptographic nonces, using a secure hash or the pseudorandom keystream of a stream cipher (*22.7.4 Stream Ciphers*). The inclusion of the server nonce means tha the client does not unilaterally specify the key.

If Diffie-Hellman-Merkle key-exchange is used (*22.8 Diffie-Hellman-Merkle Exchange*), then the server proposes the prime p and primitive root g during the `ServerHello` phase, as part of its selection of one of the cipher suites listed in the `ClientHello`. The server (Bob) also includes g^b mod p. The client chooses its exponent a, and sends g^a mod p to the server. The pre-master secret is $g^{a \times b}$ mod p, which the client, as in

the RSA case, knows at the end of the first RTT. The client's g^a mod p cannot be encrypted using the master key, but everything else sent by the client at that point can be.

22.10.2.4.2 TLS version 1.3

Version 1.3 of TLS prunes the list of acceptable cipher suites of all algorithms no longer considered secure, and also of all algorithms that do not support forward secrecy, *22.9.2 Forward Secrecy*. The version-negotiation process has been improved, to prevent version-downgrade attacks.

TLS v1.3 requires that all messages after the `ServerHello` be encrypted (except possibly one last client-to-server key-exchange message that should be intrinsically secure). The client knows the master secret at this point, and the server will know it as soon as it hears from the client.

Finally, TLS v1.3 adds a so-called **0-RTT** mode, in which the client can send encrypted messages from the beginning, assuming an appropriate master secret has been negotiated during a *previous* connection. For HTTPS interactions, this is a common case. The 0-RTT mode, in particular, allows the client to send secure *data* along with its `ClientHello`. The server can respond with 0-RTT-protected data along with its `ServerHello`, if it chooses to. After that, everything should be protected with the new session's master secret; this is known as **1-RTT** protection.

To send 0-RTT data, the client must resume the earlier session by including the previous Session ID in its `ClientHello` message. The server *can* refuse 0-RTT data, and demand 1-RTT protection, but usually will not.

The advantage of 0-RTT data is that, if the server accepts it, an answer can return to the client in a single RTT. This is particularly significant in QUIC (*11.1.1 QUIC* and *12.22.4 QUIC Revisited*), in which the TLS handshake is carried out in parallel with the QUIC connection handshake (a replacement for the TCP three-way handshake). This means that the response comes just one RTT after the very first message from the client. QUIC requires TLS v1.3 (or later).

0-RTT protection is not quite as strong ast 1-RTT protection. For one, forward secrecy does not apply. More seriously, a participant receiving 0-RTT data is vulnerable to replay attacks. An attacker cannot modify a previously intercepted 0-RTT message (without breaking the cipher), but can resend it.

At a minimum, the recipient of a 0-RTT request should accept it only if it is **idempotent**: yielding the same results and side-effects whether executed once or twice (*11.5.2 Sun RPC*). This is generally automatic for simple HTTP GET requests. Idempotency is not itself a security guarantee – after all, the request "delete all files" is idempotent. The point, however, is that the particular request must, if part of a replay attack, have been executed before, and if it was safe once, it *should* be safe twice. A TLS server can also suppport other anti-replay mechanisms, such as a database of past requests.

Another consequence of mixing 0-RTT and 1-RTT data is that the recipient needs to be able to tell which requests received 0-RTT protection and which received full 1-RTT protection.

22.10.3 A TLS Programming Example

In this section we introduce a simple pair of programs, tlsserver.c and tlsclient.c, that communicate via TLS. They somewhat resemble the simplex-talk program in *12.6 TCP simplex-talk*, except that neither reads from the command line. Instead, the client sends a message to the server (which may ignore it), and then the server sends a message back. This structure will allow us later to point the client at a "real" TLS-using

(that is, HTTPS-using) web server, instead of `tlsserver`, have the client send an HTTP GET request (*12.6.2 netcat again*), and obtain the web server's response.

Both programs are written in C, in order to use the OpenSSL library, a descendant of the original Netscape SSL implementation. The openssl package also includes some important command-line utilities for certificate creation. While OpenSSL library documentation remains notoriously spare, parts of the library have now been audited, and OpenSSL remains the most widely used implementation of TLS.

The code shown here is intended simply as a demonstration; **it should not be considered a model of how to implement secure connections**.

22.10.3.1 Making a certificate

The first step is to create the appropriate application certificate, and create our own certificate authority to sign it. For each step we will use the `openssl` command, also used below at *22.12 RSA Key Examples* with additional explanation. We start with the certificate authority. The first step is to create the key our certificate authority will use; this is what the `genrsa` option below achieves. We choose a key length of 4096 bits.

```
openssl genrsa -out CAkey.pem 4096
```

The resultant file says it is a "private key", but it is in fact a public/private key pair.

The next step is to create a **self-signed certificate**. The `req` option asks for a signing "request", the `-new` option indicates this is a new request, and the `-x509` option tells openssl to forget making a "request" and instead make a self-signed certificate. X.509 is actually a format standard.

```
openssl req -new -x509 -key CAkey.pem -out CAcert.pem -days 10000
```

The certificate here is set to expire in 10,000 days; real *published* certificates are not supposed to be valid for more than about three years. The signing process (self- or not) triggers a series of questions that will form the "Distinguished Name" of the certificate:

```
Country Name (2 letter code): US
State or Province Name (full name): Illinois
Locality Name (eg, city):Shabbona
Organization Name (eg, company): An Introduction to Computer Networks
Organizational Unit Name (eg, section): Security
Common Name (e.g. server FQDN or YOUR name): Peter L Dordal
Email Address []:
```

If we were requesting a certificate for a web server, we would use the hostname as Common Name, but we are not. The output here, CAcert.pem, represents the concatenation of the above information with the public key from CAkey.pem, and then signed by the private key from CAkey.pem. The private key itself, however, is *not* present in CAcert.pem; this is the file that the TLS client will receive as its certificate authority. We can read the information from the file as follows:

```
openssl x509 -in CAcert.pem -text
```

This produces the following, somewhat edited for space. We can see that the certificate is self-signed because the `Issuer:` is the same as the `Subject:`

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 14576542390929842281 (0xca4a481320cbe069)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, ST=Illinois, L=Shabbona, O=An Introduction to Computer Networks, OU=S
        Validity
            Not Before: Jan 10 20:17:55 2017 GMT
            Not After : May 28 20:17:55 2044 GMT
        Subject: C=US, ST=Illinois, L=Shabbona, O=An Introduction to Computer Networks, OU=
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:b0:70:06:7e:38:1d:29:35:a7:ca:40:bf:fd:6e:
                    e5:26:7b:ee:0d:e7:d7:c2:61:8e:42:5f:b9:85:8c:
                    ...
                    d4:12:cf
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                D1:74:15:F5:31:CB:DD:FA:D6:AE:81:7A:40:AA:64:7A:55:96:2E:08
            X509v3 Authority Key Identifier:
                keyid:D1:74:15:F5:31:CB:DD:FA:D6:AE:81:7A:40:AA:64:7A:55:96:2E:08

            X509v3 Basic Constraints:
                CA:TRUE
    Signature Algorithm: sha256WithRSAEncryption
         2a:d2:35:43:c2:5d:1c:5d:e2:88:ed:4e:aa:d2:b5:d6:e9:26:
         60:f0:37:ea:29:56:14:62:58:01:78:b0:6f:ee:ab:40:17:36:
         ...
         eb:3d:da:79:5c:90:4d:c9
-----BEGIN CERTIFICATE-----
MIIF8TCCA9mgAwIBAgIJAMpKSBMgy+BpMA0GCSqGSIb3DQEBCwUAMIGOMQswCQYD
VQQGEwJVUzERMA8GA1UECAwISWxsaW5vaXMxETAPBgNVBAcMCFNoYWJib25hMS0w
...
ywhRBNEO1XXB7bFrkkv93q4G3Re2zyw2/5BDEn7rPdp5XJBNyQ==
-----END CERTIFICATE-----
```

We now create the application key and then the application **signature request**. This request we will then sign with the above certificate CAfile.pem to generate the application certificate.

```
openssl genrsa -out appkey.pem 2048
openssl req -new -key appkey.pem -out appreq.pem -days 10000
```

The certificate request here again requires entering a Distinguished Name. This time we enter as follows; the Organization Name must match the CAcert.pem above:

```
Country Name (2 letter code): US
State or Province Name (full name): Illinois
Locality Name (eg, city): no fixed abode
Organization Name (eg, company): An Introduction to Computer Networks
Organizational Unit Name (eg, section): TLS server
Common Name (e.g. server FQDN or YOUR name): Odradek
```

```
Email Address []:
```

We are also asked for a "challenge password", but we leave this blank.

Now we come to the final step: the actual signing. Unlike any of the previous openssl commands, this requires root privileges. It also makes use of the global openssl configuration file (/etc/ssl/openssl.cnf), which, among other things, references a file "serial" to assign the certificate a serial number. (We did not have to do any of this to create the *self*-signed certificate.)

```
openssl ca -in appreq.pem -out appcert.pem -days 10000 -cert CAcert.pem -keyfile CAkey.pem
```

We now have our application certificate in appcert.pem, which we deliver to the application section, below. If we read it with `openssl x509 -in appcert.pem -text`, we find that the Issuer is that of CAfile.pem, but the Subject is as above, with Common Name = Odradek.

22.10.3.2 TLSserver

The complete server is at tlsserver.c, along with the certificate and key files appcert.pem and appkey.pem (both stored with an additional `.text` suffix to prevent an accidental click from loading them directly into a browser). To compiler the server under linux, with the OpenSSL library installed in the usual place, we use

```
gcc -o tlsserver tlssserver.c -lssl -lcrypto
```

The server should be run in the same directory with its certificate and key files.

In the following we go through the important lines of the full program stripped of any error checking. Most OpenSSL library methods return 1 on success and 0 on failure. Different kinds of failure may require different error-message libraries.

```
SSL_library_init();
```

This does what it sounds like: initializes the SSL subsystem. Loading error strings may also be useful. The next steps commit us to the versions of SSL we agree to accept.

```
method = SSLv23_server_method();
ctx = SSL_CTX_new(method);
SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2 | SSL_OP_NO_SSLv3 | SSL_OP_NO_TLSv1 | SSL_OP_NO_TI
```

Somewhat paradoxically, `SSLv23_server_method()` accepts *all* SSL and TLS versions. In the third line above, we then disable everything earlier than TLS version 1.1. In openssl version 1.1.0 (the numbering is unrelated to the TLS versioning), `SSLv23_server_method()` can be replaced with the more appropriately named `TLS_server_method()`.

The variable `ctx` represents a TLS *context*, which is a set of TLS state information. Our server will use the same context for all incoming connections.

Next we load the server certificate and key files into our newly created TLS context. Recall that the server side gets our application certificate `appcert.pem`; the client side will get our certificate-authority certificate `CAcert.pem`.

```
int cfile_result = SSL_CTX_use_certificate_file(ctx, "./appcert.pem", SSL_FILETYPE_PEM);
int kfile_result = SSL_CTX_use_PrivateKey_file (ctx, "./appkey.pem",  SSL_FILETYPE_PEM);
```

The server needs the key file to sign messages. Next we create an ordinary TCP socket listening on port 4433; `createSocket()` is defined at the end of the tlsserver.c file.

```
sock = createSocket(4433);
```

Finally we get to the main loop. We accept a TCP connection, create a new connection-specific SSL object from our context, and tie the new SSL object to the socket.

```
while(1) {
    int childsock = accept(sock, (struct sockaddr*)&addr, &len);
    SSL* ssl = SSL_new(ctx);
    SSL_set_fd(ssl, childsock);
    SSL_accept(ssl);
    SSL_write(ssl, reply, strlen(reply));
}
```

`SSL_accept()` is where the handshaking described in *22.10.2.4 TLS Connection Setup* takes place. At this point, the server writes its `reply` message over the now-encrypted channel, and is done.

22.10.3.3 TLSclient

The complete client is at tlsclient.c; its certificate is CAcert.pem (again with a `.text` suffix). Again we go through the sequence of SSL library calls with error-checking removed. As with the server, we start with initialization; this time, we also load error strings:

```
SSL_library_init();   // "SSL_library_init() always returns '1'
ERR_load_crypto_strings();
SSL_load_error_strings();
```

Next we again choose what TLS versions we will allow, this time starting with `SSLv23_`**client**`_method()`:

```
method = SSLv23_client_method();
ctx = SSL_CTX_new(method);
SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2 | SSL_OP_NO_SSLv3|SSL_OP_NO_TLSv1|SSL_OP_NO_TLSv1_
```

If we instead use `method = TLSv1_1_client_method()`, the connection should fail, this call allows *only* TLS version 1.1, and the server requires TLS version 1.2 or better.

The next step is to load the **trust store**, that is, the certificates from the certificate authorities we have elected to trust. If we do nothing, the trust store will be empty. We first load a standard certificate directory (directories are supplied to `SSL_CTX_load_verify_locations()` as the third parameter and individual files as the second). Certificates in a directory must be named (possibly using symbolic links) by their hash values; see the c_rehash utility. If all we wanted was to be able to trust our own server's `appcert.pem`, we could just load our own certificate-authority certificate `CAcert.pem`, but we will need the standard certificate directory if we want to point `tlsclient` at a real HTTPS server rather than at `tlsserver`.

```
SSL_CTX_load_verify_locations(ctx, NULL, "/etc/ssl/certs")
```

Next we load our own certificate `CAcert.pem`, which is then *added* to the trust store, in addition to the standard certificates. We can add multiple individual certificates by making multiple calls, or by concatenating all the certificates into a single file. The certificates must be separated by their `BEGIN CERTIFICATE` and `END CERTIFICATE` lines.

```
SSL_CTX_load_verify_locations(ctx, "CAcert.pem", NULL);
```

Now we're ready to connect to the server. The last line below initiates the TLS handshake, starting with `ClientHello`.

```
sock = openConnection(hostname, port);
ssl = SSL_new(ctx);
SSL_set_fd(ssl, sock);
SSL_connect(ssl);
```

Next the client retrieves (and, in our case prints) the certificate supplied by the server. If the server is `tlsserver`, this will normally be `appcert.pem`, with `CN=Odradek`.

```
cert = SSL_get_peer_certificate(ssl);
certname = X509_get_subject_name(cert);
// print certname
```

The printed certname does indeed show `CN=Odradek`, from `appcert.pem`. If we were writing a web browser, this is the point where we would verify that the site hostname matches the `CN` field of the certificate.

After the client receives the application certificate, it must **verify** its signature, with the call below. This is where the client uses its trust store.

```
ret = SSL_get_verify_result(ssl);
```

If one of the certificate authorities in the trust store vouches for the signature on the application certificate, the return value above is `X509_V_OK`, and all is well. If we comment out the loading of `CAcert.pem`, however, we get "unable to get local issuer certificate". If, with tlsclient still not loading `CAcert.pem`, we have the *server* send `CAcert.pem` (and `CAkey.pem`), instead of its proper certificate, we get an error "self-signed certificate". A full list of certificate-verification errors is listed with the verify command.

If validation fails, the connection is still encrypted, but is vulnerable to a man-in-the-middle attack. Regardless of what happened during validation, our particular `tlsclient` goes on to write an HTTP GET request to the server, and then read the server's response (the companion `tlsserver` program does not in fact read the GET request). Generally speaking, however, **continuing the TLS session after a certificate validation failure is a very bad idea**.

```
SSL_write(ssl, request, req_len);     // ignored by tlsserver program
do {
    bytesread = SSL_read(ssl, buf, sizeof(buf));
    fwrite(buf, bytesread, 1, stdout);
} while(bytesread > 0);
```

The written request here is ignored by `tlsserver`; it is an HTTP GET request of the form `GET / HTTP/1.1\r\nHost:` *hostname*`\r\n\r\n`. If we point `tlsclient` at a real webserver, say

```
tlsclient google.com 443
```

then we should again get an X509_V_OK verification result because we loaded the default certificate-authority library.

We can also point the built-in openssl client at tlsserver; by default it connects to localhost at port 4433:

```
openssl s_client
```

Of course, verification fails. This is because s_client doesn't know about our certificate authority. We can add it, however, on the command line:

```
openssl s_client -CAfile CAcert.pem
```

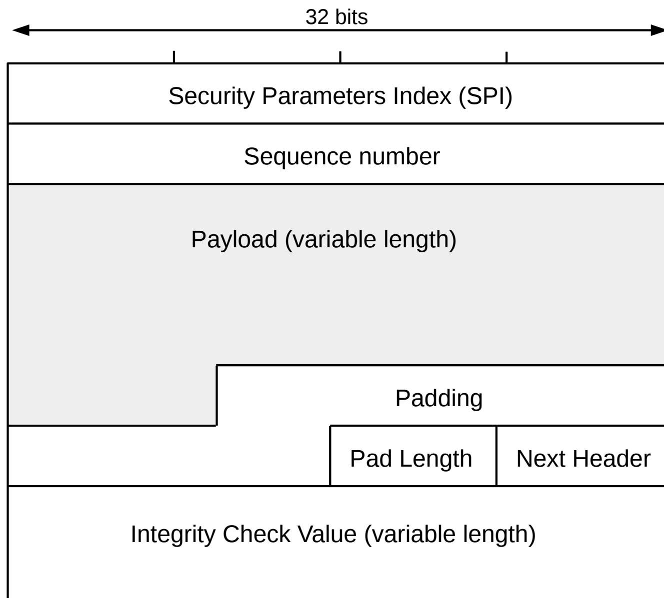
Now the verification is successful.

22.11 IPsec

The SSH software package was built from the ground up to implement the SSH protocol. All modern web browsers incorporate TLS libraries to enable secure web connections. What can you do if you want to add encryption (or authentication) to a network application that doesn't have it built in? Or, alternatively, how can you as a system administrator ensure that everyone's traffic is protected, regardless of what software they are using?

IPsec, for "IP security", is one answer. It is a general-purpose security protocol which typically behaves as if it were a network sublayer *below* the IP layer (or, in transport mode, below the Transport layer). In this it is akin to Wi-Fi (*3.7.5 Wi-Fi Security*), which implements encryption within the LAN layer; in both Wi-Fi and IPsec the encryption is transparent to the communicating applications. In terms of actual implementation it is most often incorporated within the IP layer, but can be implemented as an external network appliance.

IPsec can be used to protect anything from individual TCP (or UDP) connections to all traffic between a pair of routers. It is often used to implement VPN-like access from "outside" hosts to private subnets behind NAT routers. It is easily adapted to support any encryption or authentication mechanism. IPsec supports two packet formats: the **authentication header**, AH, for authentication only, and the **encapsulating security payload**, ESP, below, for either authentication or encryption or both. The ESP format is much more common and is the only one we will consider here. The AH format dates from the days when most export of encryption software from the United States was banned (see the sidebar 'Crypto Law' at *22.7.2 Block Ciphers*), and, in any event, the ESP format can be used for authentication only. The ESP packet format is as follows:



ESP packet layout

The SPI identifies the security association, below. The sequence number is there to prevent replay attacks. Senders must increment it on every transmission, but receivers care only if the received numbers are not strictly increasing; gaps due to lost packets do not matter. The cryptographic algorithm applied to the payload and the integrity-check algorithm are negotiated at connection set-up. The Padding field is used first to bring the Payload length up to a multiple of the applicable encryption blocksize, and then to round up the total to a multiple of four bytes. The Next Header field describes the data that is *inside* the Payload, *eg* TCP or UDP for Transport mode or IP for Tunnel mode. It corresponds to the Protocol field of *7.1 The IPv4 Header* or the Next Header field of *8.1 The IPv6 Header*.

IPsec has two primary modes: **transport** and **tunnel**. In transport mode, the IPsec endpoints are also typically the traffic endpoints, and only the transport-layer header (*eg* TCP header) and data are encrypted or protected. In the more-common tunnel mode one of the IPsec endpoints is often a router (or "security gateway"); encryption or protection includes the original IP headers, so that an eavesdropper cannot necessarily identify the actual traffic endpoints.

IPsec is documented in a wide range of RFCs. A good overview of the architectural principles is found in **RFC 4301**. The ESP packet format is described in **RFC 4303**.

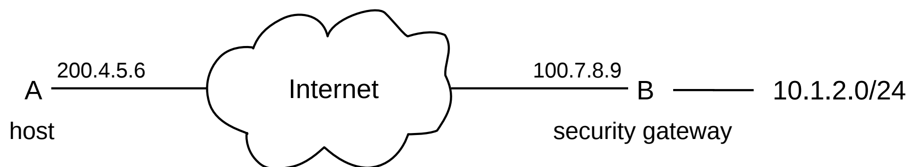
A word of warning: while IPsec does support modern encryption, it also continues to support outdated algorithms as well; users must take care to ensure that the encryption negotiated is sufficient. IPsec has also attracted, in recent years, rather less attention from the security community than SSH or TLS, and "many eyes make all bugs shallow". Or at least some bugs.

22.11.1 Security Associations

In order for a given connection or node-to-node path to receive IPsec protection, it is first necessary to set up a pair of **security associations**. A security association consists of all necessary encryption/authentication attributes – algorithms, keys, rekeying rules, *etc* – together with a set of **selectors** to identify the covered

traffic. A given security association covers traffic in one direction only; bidirectional traffic requires a separate security association for each direction. For outbound traffic – that is, traffic going from unprotected (internal) to IPsec-protected status – the selector consists of the destination IP address (or set of addresses) and possibly also the source IP address (or set of source addresses) and port or protocol values. Inbound ESP packets carry a 32-bit Security Parameters Index, or SPI, that for unicast traffic identifies the security association. However, that security association must still be checked against the packet for an actual match.

The destination and source IP addresses need not be the same as the IP addresses of the IPsec endpoints. As an example, consider the following tunnel-mode arrangement, in which traveling host A wants to connect to private subnet 10.1.2.0/24 through security gateway B. IPv4 addresses are shown, but the same arrangement can be created with IPv6.



The A-to-B IPsec security association's selector will include the entire subnet 10.1.2.0/24 in its set of destination addresses. A packet from A to 10.1.2.3 arriving at A's IPsec interface will match this selector, and will be encapsulated and sent (via normal Internet routing) to B at 100.7.8.9. B will de-encapsulate the packet, and then forward it on to 10.1.2.3 using its normal IP forwarding table. B might actually *be* the NAT router at its site, with external address 10.7.8.9 and internal subnet 10.1.2.0/24, or B might simply be a publicly visible host at its site that happens to have a route to the private 10.1.2.0/24 subnet.

The action of forwarding the encapsulated packet from A to B closely resembles IP forwarding, but isn't quite. It is unlikely A will have a true forwarding-table entry for 10.1.2.0/24 at all; it will very likely have only a single default route to its local ISP connection. Delivery of the packet cannot be understood simply by examining A's IP forwarding table. A might even have a forwarding-table entry for 10.1.2.0/24 to somewhere else, but the IPsec "pseudo-route" to B's 10.1.2.0/24 is still the one taken. This can easily lead to confusion; for complex arrangements with multiple overlapping security associations, this can lead to nontrivial difficulties in figuring out just how a packet is forwarded.

A second routing issue exists at B's end. Host 10.1.2.3 will see the packet from A arrive with address 200.4.5.6. Its reply back to A will be delivered to A using the tunnel only if the B-site routing infrastructure routes the packet back to B. If B is the NAT router, this will happen as a matter of course, but otherwise some deliberate action may need to be taken to avoid having 10.1.2.3-to-A traffic take an unsecured route. Additionally, the B-to-A security association needs to list 10.1.2.0/24 in its list of *source* addresses. The IPsec "pseudo-route" now resembles the policy-based routing of *9.6 Routing on Other Attributes*, with routing based on both destination and source addresses. A packet for A arriving at B with source address 10.**2.4**.3 should *not* take the IPsec tunnel. (To add confusion, linux IPsec pseudo-routes do not actually show up in the linux policy-based routing tables.)

Security associations are created through a software **management interface**, *eg* via the linux ipsec command and the associated configuration file ipsec.conf. It is possible for an application to request creation of the necessary security associations, but it is more common for these to be set up *before* the IPsec-protected application starts up.

A request for the creation of a security association typically triggers the invocation of the **Internet Key Exchange**, IKE, protocol; the current version 2 is often abbreviated IKEv2. IKEv2 is described in **RFC**

7296. IKEv2 typically uses public keys to negotiate a session key (*22.7.1 Session Keys*); IKEv2 may then renegotiate the session key at intervals. In the simplest (and not very secure) case, both sides have been manually configured with a session key, and IKEv2 has little to do beyond verifying that the two sides have the same key.

NAT traversal of IPsec packets is particularly tricky. For AH packets it is impossible, because the cryptographic authentication code in the packet covers the original IP addresses, as well as the packet transport data. That is not an issue for ESP packets, but even there the incoming packet must match the receivers's security-association selector, which it will not if that was negotiated using the sender's original IP address. An additional problem is that many NAT routers fail to forward (or fail to forward properly) packets outside of protocols ICMP, UDP and TCP.

As a result, IPsec has its very own NAT-traversal mechanism, outlined in **RFC 3715**, **RFC 3947** and **RFC 3948**. IPsec packets are encapsulated in UDP packets, with their original headers. After de-encapsulation at the IPsec receiving end, it is these original headers that are used in the security-association check. Additionally, a keepalive mechanism is defined in which the IPsec nodes send regular small packets to make sure the NAT mapping for the connection does not time out.

22.12 RSA Key Examples

In this section we create a short RSA key, using the `openssl` package, available for Windows, Macs and linux. We then break it, via factoring.

The first step is to create an RSA key with length 96 bits; this length was chosen for easy factorability.

```
openssl genrsa -out key96.pem 96
```

The resultant file is as follows:

```
-----BEGIN RSA PRIVATE KEY-----
MFICAQACDQCo1hzP6/gTzbNAEHcCAwEAAQINAJzcEHi8aYSO0iizgQIHANkzC28P
jwIHAMb/VQH8mQIHALc+9ZqRyQIHAKkfQ43msQIGJxhmMMOs
-----END RSA PRIVATE KEY-----
```

This is in the so-called PEM format, which means that the two lines in the middle are the base64 encoding of the ASN.1 encoding (*21.12 SNMP and ASN.1 Encoding*) of the actual data. Despite the `PRIVATE KEY` label, the file in fact contains both private and public keys. SSH private keys, typically generated with the `ssh-keygen` command, are also in PEM format.

We can extract the PEM-file data with the next command:

```
openssl rsa -in key96.pem -text
```

The output is the following:

```
Private-Key: (96 bit)
modulus:
    00:a8:d6:1c:cf:eb:f8:13:cd:b3:40:10:77
publicExponent: 65537 (0x10001)
privateExponent:
    00:9c:dc:10:78:bc:69:84:8e:d2:28:b3:81
prime1:
```

```
    00:d9:33:0b:6f:0f:8f
prime2:
    00:c6:ff:55:01:fc:99
exponent1:
    00:b7:3e:f5:9a:91:c9
exponent2:
    00:a9:1f:43:8d:e6:b1
coefficient:
    27:18:66:30:c3:ac
```

The default OpenSSL encryption exponent, denoted e in *22.9.1 RSA*, is 65537 = 2^{16}+1. (The default exponent used to be 3, but see exercise 10.0)

We next convert all these hex numbers to decimal; the corresponding notation of *22.9.1 RSA* is in parentheses.

modulus (n)	52252327837124407964427358327
privateExponent (d)	48545702997494592199601992577
prime1 (p)	238813258387343
prime2 (q)	218799945153689
exponent1	201481036403145
exponent2	185951742453425
coefficient	42985747170220

We now verify some arithmetic, using any tool that supports large integers (*eg* python3, used here, or the unix `bc` command). First we check n = pq:

```
>>> 238813258387343 * 218799945153689
52252327837124407964427358327
```

Next we check that ed = 1 mod (p-1)(q-1):

```
>>> e=65537
>>> d=48545702997494592199601992577
>>> p=238813258387343
>>> q=218799945153689
>>> (p-1)*(q-1)
52252327837123950351223817296
>>> e*d % 52252327837123950351223817296
1
```

To encrypt a message m, we *must* use efficient mod-n calculations; here is an implementation of the repeated-squaring algorithm (mentioned above in *22.8.1 Fast Arithmetic*) in python3. (This function is built into python as `pow(x,e,n)`.)

```
def power(x,e,n):   # computes x^e mod n
    pow = 1
    while e>0:
       if e%2 == 1: pow = pow*x % n
       x = x*x % n
       e = e//2     # // denotes integer division
    return pow
```

Let m be the string "Rivest". In hex this is 0x526976657374; in decimal, 90612911403892.

```
>>> m=0x526976657374
>>> c=power(m,e,n)
>>> c
38571433489059199500953769621
>>> power(c,d,n)
90612911403892
```

What about the last three numbers in the PEM file, `exponent1`, `exponent2` and `coefficient`? These are pre-computed values to speed up decryption. Their values are

- `exponent1` = d mod (p-1)
- `exponent2` = d mod (q-1)
- `coefficient` is the solution of coefficient $\times$ q = 1 mod p

22.12.1 Breaking the key

Finally, let us break this 96-bit key and decrypt the message with ciphertext c above. The hard part is factoring n; we use the Gnu/linux `factor` command:

```
> factor 52252327837124407964427358327
52252327837124407964427358327: 218799945153689 238813258387343
```

The factors are indeed the values of p and q, above. Factoring took 2.584 seconds on the author's laptop. Of course, 96-bit RSA keys were never secure; recall that the current recommendation is to use 2048-bit keys.

The Gnu/linux `factor` command uses Pollard's rho algorithm, and, while serviceable, is not especially well suited to factoring the product of two large primes. The author was able to factor a 200-bit modulus in just over 5 seconds using the msieve program, one of several large-number-factoring programs available on the Internet. Msieve implements a version of the number-field-sieve algorithm mentioned in *22.9.1.2 Factoring RSA Keys*.

We are almost done; we now need to find the decryption key d, knowing e, p-1 and q-1. For this we need an implementation of the extended Euclidean algorithm; the following Python implementation is taken from WikiBooks:

```
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)
```

A call to `egcd(a,b)` returns a triple (g,x,y) where g is the greatest common divisor of a and b, and x and y are solutions to g = ax + by. From *22.9.1 RSA*, we need d to be positive and to satisfy 1 = de + (p-1)(q-1)y. The x value (the second value) returned by `egcd(e, (p-1)*(q-1))` satisfies the second part, but it may be negative in which case we need to add (p-1)(q-1) to get a positive value which is congruent mod (p-1)(q-1). This x value is -3706624839629358151621824719; after adding (p-1)(q-1) we get d=48545702997494592199601992577.

This is the value of d we started with. If c is the ciphertext, we now calculate m = pow(c,d,n) as before, yielding m=90612911403892, or, in hex, 52:69:76:65:73:74, “Rivest”.

22.13 Exercises

Exercises are given fractional (floating point) numbers, to allow for interpolation of new exercises.

1.0 Modify the netsploit.c program of *22.2.2.3 The exploit* so that the NOPslide is about 1024 bytes long, and the NOPslide and shellcode are now *above* the overwritten return address in memory.

2.0 Disable ASLR on a linux system by writing the appropriate value to /proc/sys/kernel/randomize_va_space. Now get the netsploit attack to work *without* having the attacked server print out a stack address. You *are* allowed beforehand to run a test copy of the server that prints its address.

3.0 Below are a set of four possible TCP-segment reassembly rules. These rules (and more) appear in *[SP03]*. Recall that, once values for all positions j⩽i are received, these values are acknowleged and released to the application; there must be some earlier “hole” for segments to be subject to reassembly at all.

1. Always accept the first value received for each position.
2. Always accept the last value received for each position, up until the value is released to the application.
3. [”BSD”] For each new segment, discard each position from that segment if that position was supplied by an earlier packet with a left edge less than or equal to that of the new packet. Values in any remaining positions replace any previously received values.
4. [”Linux”] Same as the previous rule, except that we only discard positions that were supplied by an earlier packet with a left edge *strictly smaller* than that of the new packet.

For each of the following two sets of packet arrivals, give the reassembled packet. Values in the same column have the same position. Reassembled packets will all begin with “daa” and “ebb” respectively.

(a).

```
| |a a a a a
|  |    b b b b b b b
|  |    c c c c
|d|
```

(b).

```
| |     a a a a a
| |b b b b b
| |     c c c c c c
| |       d d d
|e|
```

4.0 Suppose a network intrusion-detection system is 10 hops from the attacker, but the target is 11 hops, behind one more router R. Outline a strategy of tweaking the TTL field so that the NIDS receives TCP stream “helpful” and the target receives “harmful”. Segments should either be disjoint or cover exactly the same range of bytes; you may assume that the target accepts the first segment for any given range of bytes.

5.0 Suppose Alice encrypts blocks P1, P2 and P3 using CBC mode (*22.7.3 Cipher Modes*). The initialization vector is C0. The encrypt and decrypt operations are E(P) = C and D(C) = P. We have

- C1 = E(C0 XOR P1)
- C2 = E(C1 XOR P2)
- C3 = E(C2 XOR P3)

Suppose Mallory intercepts C2 in transit, and replaces it with C2' = C2 XOR M; C1 and C3 are transmitted normally; that is, Bob receives [C1',C2',C3'] where C1' = C1 and C3' = C3. Bob now attempts to decrypt; C1' decrypts normally to P1 while C2' decrypts to gibberish.

Show that C3' decrypts to P3 XOR M. (This is sometimes used in attacks to flip a specific bit or byte, in cases where earlier gibberish does not matter.)

6.0 Suppose Alice uses a block-based stream cipher (*22.7.5 Block-cipher-based stream ciphers*); block i of the keystream is Ki and Ci = Ki XOR Pi. Alice sends C1, C2 and C3 as in the previous exercise, and Mallory again replaces C2 by C2 XOR M. What are the three plaintext blocks Bob deciphers?

7.0 Show that if p and q are primes with p = 2q + 1, then g is a primitive root mod p if $g \neq 1$, $g^2 \neq 1$, and $g^q \neq 1$. (This exercise requires familiarity with modular arithmetic and primitive roots.)

8.0 Suppose we have a *short* message m, *eg* a bank PIN number. Alice wants to send message M to Bob that, without revealing m immediately, can be used later to verify that Alice knew m at the time M was sent. During this later verification, Alice may reveal m itself.

(a). Suppose Alice simply sends M = hash(m). Explain how Bob can quickly recover m.

(b). How can Alice construct M using a secure-hash function, avoiding the problem of (a)? Hint: as part of the later verification, Alice can supply additional information to Bob.

9.0 In the example of *22.7.7 Wi-Fi WEP Encryption Failure*, suppose the IV is ⟨4,-1,5⟩ and the first two bytes of the key are ⟨10,20⟩. What is the first keystream byte `S[ S[1] + S[S[1]] ]`?

10.0 Suppose Alice uses encryption exponent e=3 to three friends, Bob, Charlie and Deborah, with respective encryption moduli n_B, n_C and n_D, all of which are relatively prime. Alice sends message m to each, encrypted as

- $C_B = m^3 \bmod n_B$
- $C_C = m^3 \bmod n_C$
- $C_D = m^3 \bmod n_D$

If Mallory intercepts all three encrypted messages, explain how he can efficiently decrypt m. Hint: the Chinese Remainder Theorem implies that Mallory can find $C < n_B n_C n_D$ such that

- $C = C_B \bmod n_B$
- $C = C_C \bmod n_C$
- $C = C_D \bmod n_D$

(One simple way to avoid this risk is for Alice to include a timestamp and the recipient's name in each message, ensuring that she never sends exactly the same message twice.)

11.0 Repeat the key-creation of *22.12 RSA Key Examples* using a 110-bit key. Extract the modulus from the key file, convert it to decimal, and attempt to factor it. Can you do the factoring in under a minute?

12.0 Below are a series of public RSA keys and encrypted messages; the encrypted message is c and the modulus is n=pq. In each case, find the original message, using the methods of *22.12.1 Breaking the key*; you will have to factor n and then find d. For some keys, the Gnu/Linux `factor` command will be sufficient; for the larger keys consider msieve or some other fast factorer.

Each number below is in decimal. The encryption exponent e is always 65537; the encryption is c = power(message,e,n). Each message is an ASCII string; that is, after the numeric message is converted to a string, the byte values are each in the range 32-127. The following Python function may be useful in converting numeric messages to strings:

```
def int2ascii(n):
    if n==0: return ""
    return int2ascii(n // 256) + chr(n % 256)
```

(a) [64 bits] c=13467824835325843134
n=15733922878520524621

(b) [96 bits] c=8007751471156136764029275727
n=57644199986835279860947893727

(c) [104 bits] c=6642328489179330732282037747645
n=17058317327334907783258193953123

(d) [127 bits] c=95651949760509273124353927897611145475
n=122096824047754908887766043915630626757
Limit for Gnu/linux `factor` without the GMP library

(e) [185 bits] c=14898070767615435522751082309577192810119252877170446296
n=36881105206579952723396854897336450502002018818436622611

(f) [210 bits] c=1030865591241775131254813948981782525587720826169501849049177362
n=1089313781487492651628882855744766776820791642308868127824231021

(g) [280 bits]

c=961792929180423930042975913300802531765775361218349440433358416557620430721970697783

n=1265365011260907658483984328995361695181000868013851878620685943648084220336740539017

(h) [304 bits]

c=17860252858059565448950681133486824440185752167054796213786228492658586864179401029486173539

n=26294146550372428669569992924076340150116542153388301312743129088600749884420889043685502979

Note that RFCs are not included here.

In this chapter we present solutions (in some cases partial solutions) to exercises marked with a ♢.

24.1 Solutions for *An Overview of Networks*

1.18 Exercises

Exercise 2.7

A	
B,C,D	direct
E	D (or C)

B	
A,C	direct
D	A
E	C

C	
A,B,E	direct
D	E (or A)

D	
A,E	direct
B	A
C	E (or A)

E	
C,D	direct
A	C (or D)
B	C

Exercise 7.0(d) (destination D):

According to S1's forwarding table, the next_hop to D is S2.
According to S2's table, the next_hop to D is S5.
According to S5's table, the next_hop to D is S6.
According to S6's table, the next_hop to D is S12.

The path from S1 to D is thus S1–S2–S5–S6–S12.

Exercise 7.5(a)

The shortest path from A to F is A–S1–1–S2–2–S5–1–S6–F, for a total cost of 1+2+1 = 4.

24.2 Solutions for *Ethernet*

2.9 Exercises

Exercise 2.7

When A sends to D, all switches use fallback-to-flooding as no switch knows where D is. All switches S1-S4, though, learn where A is.

When D sends to A, S2 knows where A is and so routes the packet directly to S1, which also knows where A is. S3 and S4 do not learn where D is.

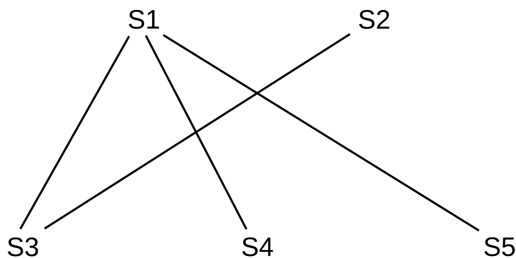
When A sends to B, all switches again use fallback-to-flooding, but no switch learns anything new.

When B sends to D, S4 uses fallback-to-flooding as it does not know where D is. However, S2 does know where D is, and so S2 forwards the packet only to D. S2 and S4 learn where B is.

switch	known destinations
S1	AD
S2	ABD
S3	A
S4	AB

Exercise 8.5

(a). S1, as root, keeps all three of its links to S3, S4 and S5. Of S2's three links in the direction of the root (that is, to S3, S4 and S5), it keeps only its link to S3 as S3 has the lowest ID.



The path from S2 to S5 is S2–S3–S1–S5.

Exercise 12.0

1. h1→h2: this packet is reported to C, as destination h2 is not known by S. C installs on S the rule that h1 can be reached via port 1. The packet is then flooded.

2. h2→h1: this packet is *not* reported to C, as destination h1 *is* known by S. The packet is not flooded.

3. h3→h1: this packet is again not reported to C. S delivers the packet normally, without flooding.

4. h2→h3: this packet is reported to C, as destination h3 is not known by S. C installs on S the rule that h2 can be reached via port 2. The packet is then flooded.

Exercise 13.0

Table for S1 only:

h1→h2: S1 reports to C and learns where h1 is
h2→h1: S1 does not report, and forwards normally; destination h1 is known
h1→h3: S1 reports to C as h3 is not known, but S1 already knows where h1 is
h3→h1: S1 does not report, and forwards normally; destination h1 is known
h2→h3: S1 reports to C and learns where h2 is
h3→h2: S1 does not report, and forwards normally; destination h2 is known

S1's table ends up with forwarding entries for h1 and h2, but not h3.

24.3 Solutions for *Other LANs*

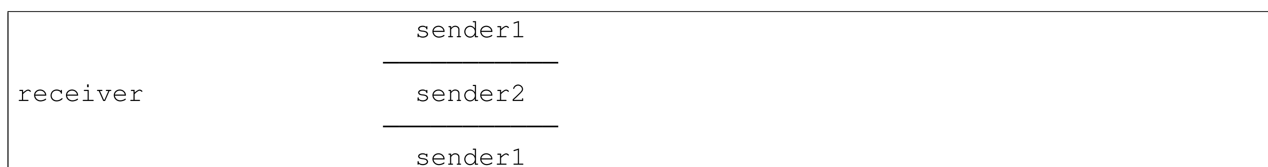
3.11 Exercises

Exercise 3(b)

One simple strategy is for stations to compare themselves to one another according to the numeric value of their MAC addresses. The station with the smallest MAC address becomes S_0, the station with the next-smallest address becomes S_1, *etc*.

Exercise 4.5

The three sending nodes can be laid out at the vertices of an equilateral triangle, with the receiving node in the center. Alternatively, with impermeable walls the following arrangement works:



The latter arrangement generalizes to almost any number of senders. For the former, senders can be laid out at the vertices of a square, or tetrahedron, or possibly a pentagon, but geometry enforces an eventual limit.

Exercise 5(b)

If T is the length of the contention interval, in µsec, then transmission and contention alternate with lengths 151–T–151–T–.... The fraction of bandwidth lost to contention is T/(T+151).

Exercise 10.5

The connections are as follows, where the VCI number is shown between the endpoints of each link:

A–1–S1–2–S2–3–S4–2–D
B–1–S2–2–S4–1–S3–3–S1–2–A

24.4 Solutions for *Links*

Exercise 3.5

The binary ASCII for the three letters is

N	0100 1110
e	0110 0101
t	0111 0100

If we look up the 4-bit "nybbles" in the right column above in the 4B/5B table at *4.1.4 4B/5B*, we get

data	symbol
0100	01010
1110	11100
0110	01110
0101	01011
0111	01111
0100	01010

Putting all these symbols together, the encoding is (with spaces added for readability)

01010 11100 01110 01011 01111 01010

24.5 Solutions for *Packets*

5.6 Exercises

Exercise 3

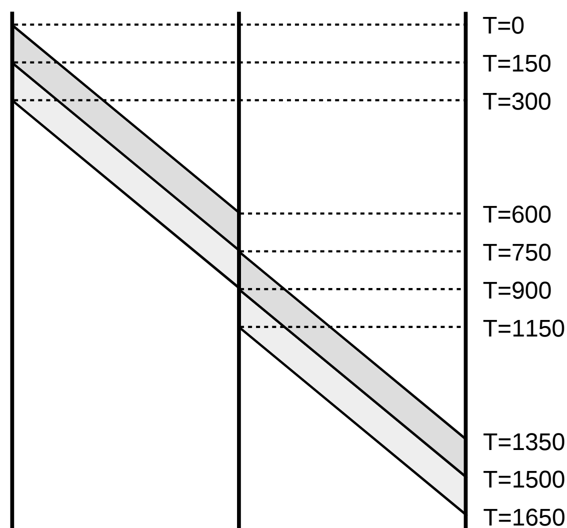
(a). The bandwidth delay for a 600-byte packet is 300 µsec. The packet has a 300 µsec bandwidth delay and a 600 µsec propagation delay for each link, for a total of 2×900 = 1800 µsec. We have the following timeline:

T=0	A begins sending the packet
T=300	A finishes sending the packet
T=600	S begins receiving the packet
T=900	S finishes receiving the packet and begins sending to B
T=1200	S finishes sending the packet
T=1500	B begins receiving the packet
T=1800	B finishes receiving the packet

(b). The bandwidth delay for each 300-byte packet is now 150 µsec, so the first packet arrives at S at T=750. The second packet arrives one bandwidth delay – 150 µsec – later, at T=900 µsec. The first packet takes another 750 µsec to travel from S to B, arriving at T=1500; the second packet arrives 150 µsec later at T=1650. Here is the timeline:

T=0	A begins sending packet 1
T=150	A finishes sending packet 1, starts on packet 2
T=300	A finishes sending packet 2
T=600	S begins receiving packet 1
T=750	S finishes receiving packet 1 and begins sending it to B
	S begins receiving packet 2
T=900	S finishes receiving packet 2 and begins sending it to B
T=1350	B begins receiving packet 1
T=1500	B has received all of packet 1 and starts receiving packet 2
T=1650	B finishes receiving packet 2

Here's the data for (b) in ladder-diagram format (not quite to scale):



The long propagation delay means that A finishes all transmissions before S begins receiving anything, and

S finishes all transmissions before B begins receiving anything. With a smaller propagation delay, these A/S/B events are likely to overlap.

Exercise 15(a)

The received data is 10100010 and the received code bits are 0111. We first calculate the four code bits for the received data:

1: parity over all bits 1: parity over second half: 1010**0010** 0: parity over these bits: 10**10**00**10** 0: parity over these bits: 1**0**1**0**0**0**1**0**

So the incorrect received code bits are **0**111.

The first bit of the received code tells us that the error is in the data (rather than the code bits). The second – correct – bit of the received code tells us that the error is in the first half, that is, in the non-bold bits of 1010**0010**.

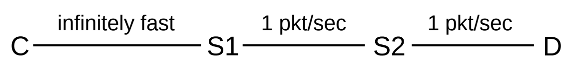
The third bit of the received code tells us that the error is in the bold bits of 10**10**00**10**, so we're down to the third or fourth bit. Finally, the fourth bit of the received code tells us the error is in the bold bits of 1**0**1**0**0**0**1**0**, which means the fourth bit is it, and the corrected data is 10110010 (making the data the same as that in the example in *5.4.2.1 Hamming Codes*).

24.6 Solutions for *Sliding Windows*

6.5 Exercises

Exercise 2.5

The network is



(a). The second part of formula 4 of *6.3.2 RTT Calculations* is queue_usage = winsize – bandwidth × RTT_{noLoad}. We know winsize = 6, bandwidth = 1 packet/sec and RTT_{noLoad} = 4, so this works out to 2 packets in the queue.

(b). At T=0, C sends packets 1-6 to S1. S1 begins sending packet 1 to S2; packets 2-6 are queued.

At T=1, S1 begins sending packet 2 to S2; S2 begins sending packet 1 to D

At T=2, S1 begins sending packet 3 to S2 and S2 begins sending packet 2 to D. Packet 1 has reached D, which begins sending ACK1 to S2.

At T=3, S1 begins sending packet 4 to S2, S2 begins sending packet 3 to D, D begins sending ACK2 to S2, and S2 begins sending ACK1 to S1.

At T=4, S1 begins sending packet 5 to S1, S2 begins sending packet 4 to D, D begins sending ACK3 to S2, and S2 begins sending ACK2 to S1. ACK1 has arrived at S1, and is instantly forwarded to C. **The window**

slides forward by one, from [1-6] to [2-7], and C sends new packet 7, which instantly arrives at S1 and is placed in the queue there.

Here is the table. The column "S1→S2" is the data packet S1 is currently sending to S2; the column "S2→S1" is the ACK that S2 is currently sending to S1; similarly for "S2→D" and "D→S2".

T	C sends	S1 queues	S1→S2	S2→D	ACK D→S2	S2→S1
0	1,2,3,4,5,6	2,3,4,5,6	1			
1		3,4,5,6	2	1		
2		4,5,6	3	2	1	
3		5,6	4	3	2	1
4	7	6,7	5	4	3	2
5	8	7,8	6	5	4	3
6	9	8,9	7	6	5	4
7	10	9,10	8	7	6	5
8	11	10,11	9	8	7	6

Exercise 7.5

(a). The formula, from *6.2.1 Bandwidth × Delay*, is winsize = bandwidth × RTT_{noLoad} = 1000 pkts/sec × 0.1 sec = 100 packets.

(b). This is the first line of formula 3 of *6.3.2 RTT Calculations*. For convenience, we switch to units of ms. Queue_usage = throughput × (RTT_{actual} – RTT_{noLoad}) = 1 pkt/ms × (130ms – 100ms) = 30 packets.

(c). We use formula 4 of *6.3.2 RTT Calculations*. We have RTT_{actual} = winsize / bandwidth = (100+50)/(1 pkt/ms) = 150 ms.

Exercise 9.0

If Data[8] is in the receiver's window, then Data[4] must have been received. For the sender to have sent Data[4] it must have previously received ACK[0]. Therefore, all transmissions of Data[0] must precede the first transmission of Data[4], and so must precede the first successful transmission of Data[4]. Because of non-reordering, no Data[0] can arrive after Data[4].

24.7 Solutions for *IPv4*

7.15 Exercises

Exercise 4.0

Forwarding table for A:

destination	next_hop
200.0.5.0/24	direct
200.0.6.0/24	direct
default	B

Exercise 6.0(d)

The subnet prefix is 10.0.168.0/21. The /21 means that the prefix consists of the first two bytes (16 bits) and the first 5 (=21-16) bits of the third byte.

Expressing the third byte, 168, in binary, we get 10101|000.

We now convert the third byte of each of the four addresses to binary, and see which match this to the first five bits. Bits after the "|" mark are ignored.

> 10.0.166.1: 166 = 10100|110; not a match
> 10.0.170.3: 170 = 10101|010; this is a match
> 10.0.174.5: 174 = 10101|110; this is a match
> 10.0.177.7: 177 = 10110|001; not a match

Exercise 6.5

(a). 240 is 11110000 in binary, and so adds 4 1-bits. Together with the 16 1-bits from the first two bytes of the mask, this is /20
(d). /20 is 16 1-bits (two bytes) plus 4 additional 1-bits. Four 1-bits starting a byte is 11110000, or 240; the full mask is 255.255.240.0

24.8 Solutions for *Routing-Update Algorithms*

9.8 Exercises

Exercise 2.5

For destination A, R1's report is likely the same as it sent earlier, when R's route to A was first established. There is no change.

For destination B, R's total distance using R1 would be 2+1 = 3. This is tied with R's route to B using next_hop R2, and so there is no change.

For destination C, R1 is reporting an increase in cost. R's next_hop to C is R1, and so R must increase its cost to C to 4+1 = 5.

For destination D, R's total cost using R1 is 3+1 = 4, cheaper than R's previous cost of 5 using next_hop R3. R changes to the R1 route.

The updated table is

destination	cost	next hop	
A	2	R1	no change; same next_hop
B	3	R2	no change; tie
C	5	R1	source increase
D	4	R1	lower-cost route found

24.9 Solutions for *Large-Scale IP Routing*

10.8 Exercises

Exercise 0.5

(i) 200.63.1.1 matches only A, as 63 = 0011 1111 in binary and 64 = 0100 0000. The first two bits do not match, ruling out the /10 prefix.

(ii) 200.80.1.1 matches A and B, but not C, as 80 = 0101 0000. Destination B is the longer match.

(iii) 200.72.1.1 matches A, B and C, but not D, as 72 = 0100 1000. Destination C is the longest match.

(iv) 200.64.1.1 matches A, B, C and D; D is the longest match.

Exercise 5(a)

P's table

destination	next hop
52.0.0.0/8	Q
53.0.0.0/8	R
51.10.0.0/16	A
51.23.0.0/16	B

Q's table

destination	next hop
51.0.0.0/8	P
53.0.0.0/8	R
52.14.0.0/16	C
52.15.0.0/16	D

R's table

destination	next hop
51.0.0.0/8	P
52.0.0.0/8	Q

Exercise 7(a)

P will receive a route from Q with AS-path ⟨Q,S⟩, and a route from R with AS-path ⟨R,S⟩.

24.10 Solutions for *UDP*

Exercise 2(a)

Two seconds after Data[3] was sent and lost, both sender and receiver will time out. The sender will retransmit Data[3]; the receiver will retransmit ACK[2]. The latter will be ignored, as the sender is not retransmitting on duplicates. When the retransmitted Data[3] arrives at the receiver, the receiver will send ACK[3] and the transfer will continue.

24.11 Solutions for *TCP Reno*

12.24 Exercises

Exercise 12.5(b)

Let t be the transit capacity; then the total $\texttt{cwnd}_{max}$ is $t(1+f)$. At this point it is convenient to normalize the units of capacity measurement so $t(1+f) = 2$. Then the link-unsaturated and queue-filling phases have lengths in the proportion t–1 to ft. The average height of the tooth during the link-unsaturated phase is $(t+1)/2$, and the average utilization percentage is thus $(t+1)/2t$. This gives a utilization of

$$(t+1)/2t \times (t-1) + ft$$

We next eliminate t. After normalization, we have $t(1+f) = 2$, and so $t = 2/(1+f)$. Plugging this in above and simplifying gives

$$\text{utilization} = (3 + 6f - f^2)/4(1+f)$$

24.12 Solutions for *Dynamics of TCP Reno*

14.13 Exercises

Exercise 2(a)

Because the window size of 50 is larger than the bandwidth×delay product is 40 packets, packets are sent at the bottleneck rate of 5 packets/ms. As packets spend 1 ms on the C–R1 link, on average this link must be carrying 5 packets. The same applies to the R2–D link; for the R1–R2 link with a 2.0 ms propagation delay, the number of packets carried will be 2.0×5 = 10.

Alternatively, from the bandwidth×delay product of 40 packets we conclude 40 packets are in transit. Of these, half are acknowledgments. The other half are distributed in proportion to the link propagation delays, yielding 5, 10, and 5 packets.

Exercise 2.5

At the point when A's bandwidth is 2 packets/ms, B's bandwidth is 4 packets/ms, and so B has 4×20 = 80 packets in transit. As B's winsize is 120, this leaves 120–80 = 40 packets in R's queue.

For A to have half as much bandwidth, it must have half the queue capacity, or 20 packets. With a bandwidth of 2 packets/ms and an $\text{RTT}_{\text{noLoad}}$ of 40 ms, A will have 2×40 = 80 packets in transit. A's winsize is then 20+80 = 100.

We can confirm this using the following formulas from *14.2.3 Example 3: competition and queue utilization*:

$$\alpha Q = w_A - 2\alpha(d_A+d)$$
$$\beta Q = w_B - 2\beta(d_B+d)$$

However, these formulas were derived under the assumption that the bottleneck bandwidth was normalized to 1. To apply them here, we need to measure time in units of 1/6 ms; that is, all times need to be multiplied

by 6. We have:

$d_A = 90$
$d_B = 30$
$d = 30$
$w_B = 120$
α=1/3, β=2/3

Substituting these into the second formula, we get the total queue utilization Q = 180 – 120 = 60, in agreement with our earlier answer. From the first formula we now solve for $w_A = \alpha(Q + 2(d_A+d)) = (1/3)\times(60 + 240) = 100$, also in agreement with our earlier answer.

Exercise 3(a)

The bottleneck bandwidth is 5 packets/ms, and as the bottleneck link is saturated, this will be the transmission rate on all links. For 5 packets to be in transmission, we need a propagation delay of 5 packets / (5 packets/ms) = 1.0 ms.

Exercise 13.5(a)

The `cwnd` will vary between 500 and 1000, with an average of 750 packets. Losses occur at intervals of 500 RTTs, during which 500*750 packets have been sent.

24.13 Solutions for *Mininet*

Exercise 6.0(a)

When h1 sends its broadcast ARP, the controller learns where h1 is relative to s1, s2 and s3.

When h2 replies to h2, there is not yet a rule for h1 at s2, and so s2 sends the packet to the controller, which installs rules for destinations h1 and h2 in s2. The packet is then flooded. When it arrives at s1, the controller installs rules for h1 and h2 and s1. The packet also arrives at s3, by flooding, so the controller installs rules for h1 and h2 in s3.

When h3 sends its broadcast packet, all of s1-s3 see the packet. Because the packet is broadcast, all three switches report the packet to the controller.

When h1 replies to h3, no switches have a rule for destination h3, and the controller knows how to reach h1 and h3 from each switch, so a new rule for reaching h3 is installed at each switch.

- genindex
- search

BIBLIOGRAPHY

[ABDGHSTVWZ15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, Paul Zimmermann, "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice", preprint at weakdh.org, May 2015.

[AEH75] Eralp Akkoyunlu, Kattamuri Ekanadham and R V Huber, "Some constraints and tradeoffs in the design of network communications", SOSP '75: Proceedings of the fifth ACM symposium on Operating systems and principles, November 1975.

[AO96] Aleph One (Elias Levy), "Smashing The Stack For Fun And Profit", Phrack volume 7 number 49, 1996, available at http://insecure.org/stf/smashstack.html.

[AGMPS10] Mohammad Alizadeh, Albert Greenberg, David Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta and Murari Sridharan, "Data Center TCP (DCTCP)", Proceedings of ACM SIGCOMM 2010, September 2010.

[AP99] Mark Allman and Vern Paxson, "On Estimating End-to-End Network Path Properties", Proceedings of ACM SIGCOMM 1999, August 1999.

[ACPRT16] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin and Stefano Tessaro, "Scrypt is Maximally Memory-Hard", Cryptology ePrint Archive, Report 2016/989, 2016, available at https://eprint.iacr.org/2016/989.

[AHLR07] Chris Anley, John Heasman, Felix "FX" Linder and Gerardo Richarte, "The Shellcoder's Handbook", second edition, Wiley, 2007.

[AKM04] Guido Appenzeller, Isaac Keslassy and Nick McKeown, "Sizing Router Buffers", ACM SIGCOMM Computer Communication Review, October 2004

[JA05] John Assalone, "Exploiting the GDI+ JPEG COM Marker Integer Underflow Vulnerability", Global Information Assurance Certification technical note, January 2005, available at www.giac.org/paper/gcih/679/exploiting-gdi-plus-jpeg-marker-integer-underflow-vulnerability/106878.

[PB62] Paul Baran, "On Distributed Computing Networks", Rand Corporation Technical Report P-2626, 1962.

[BCL09] Steven Bauer, David Clark and William Lehr, "The Evolution of Internet Congestion", Telecommunications Policy Research Conference (TPRC) 2009, August 2009, available at ssrn.com/abstract=1999830.

[MB06] Mihir Bellare, “New Proofs for NMAC and HMAC: Security without Collision-Resistance”, Advances in Cryptology - CRYPTO ‘06 Proceedings, Lecture Notes in Computer Science volume 4117, Springer-Verlag, 2006.

[BCK96] Mihir Bellare, Ran Canetti and Hugo Krawczyk, “Keying Hash Functions for Message Authentication”, Advances in Cryptology - CRYPTO ‘96 Proceedings, Lecture Notes in Computer Science volume 1109, Springer-Verlag, 1996.

[BN00] Mihir Bellare and Chanathip Namprempre, “Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm”, Advances in Cryptology — ASIACRYPT 2000 / Lecture Notes in Computer Science volume 1976, Springer-Verlag, 2000; updated version July 2007.

[BZ97] Jon Bennett and Hui Zhang, “Hierarchical Packet Fair Queueing Algorithms”, IEEE/ACM Transactions on Networking, volume 5, October 1997.

[DB08] Daniel Bernstein, “The Salsa20 family of stream ciphers”, Chapter, New Stream Cipher Designs, Matthew Robshaw and Olivier Billet, editors, Springer-Verlag, 2008.

[JB05] John Bickett, “Bit-rate Selection in Wireless Networks”, MS Thesis, Massachusetts Institute of Technology, 2005.

[BCTCU16] Timm Böttger, Felix Cuadrado, Gareth Tyson, Ignacio Castro and Steve Uhlig, “Open Connect Everywhere: A Glimpse at the Internet Ecosystem through the Lens of the Netflix CDN”, ARXIV, arXiv e-print (arXiv:1606.05519), June 2016.

[BP95] Lawrence Brakmo and Larry Peterson, “TCP Vegas: End to End Congestion Avoidance on a Global Internet”, IEEE Journal on Selected Areas in Communications, volume 13 number 8, 1995.

[BBGO08] Vladimir Brik, Suman Banerjee, Marco Gruteser and Sangho Oh, “Wireless Device Identification with Radiometric Signatures”, Proceedings of the 14th ACM International Conference on Mobile Computing and Networking (MobiCom ‘08), September 2008.

[AB03] Andreis Brouwer, “Hackers Hut”, http://www.win.tue.nl/~aeb/linux/hh/hh.html, April 1, 2003

[CF04] Carlo Caini and Rosario Firrincieli, “TCP Hybla: a TCP enhancement for heterogeneous networks”, International Journal of Satellite Communications and Networking, volume 22, pp 547-566, 2004.

[CGYJ16] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh and Van Jacobson, “BBR Congestion-Based Congestion Control”, ACM Queue, volume 14 number 5, September-October 2016.

[CM03] Zehra Cataltepe and Prat Moghe, “Characterizing Nature and Location of Congestion on the Public Internet”, Proceedings of the Eighth IEEE International Symposium on Computers and Communication, 2003.

[CDLMR00] Stefania Cavallar, Bruce Dodson, Arjen K Lenstra, Walter Lioen, Peter Montgomery, Brian Murphy, Herman te Riele, Karen Aardal, Jeff Gilchrist, Gerard Guillerm, Paul Leyland, Joel Marchand, François Morain, Alec Muffett, Craig Putnam, Chris Putnam and Paul Zimmermann, “Factorization of a 512-bit RSA Modulus”, Advances in Cryptology — EUROCRYPT 2000, Lecture Notes in Computer Science volume 1807, Springer-Verlag, 2000.

[CK74] Vinton G Cerf and Robert E Kahn, “A Protocol for Packet Network Intercommunication”, IEEE Transactions on Communications, volume 22 number 5, May 1974.

[CJ89] Dah-Ming Chiu and Raj Jain, "Analysis of the Increase/Decrease Algorithms for Congestion Avoidance in Computer Networks", Journal of Computer Networks volume 17, pp. 1-14, 1989.

[CJ91] David Clark and Van Jacobson, "Flexible and efficient resource management for datagram networks", Presentation, September 1991

[CSZ92] David Clark, Scott Shenker and Lixia Zhang, "Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism", Proceedings of ACM SIGCOMM 1992, August 1992.

[CBcDHL14] David Clark, Steven Bauer, kc claffy, Amogh Dhamdhere, BBradley Huffaker, William Lehr, and Matthew Luckie, "Measurement and Analysis of Internet Interconnection and Congestion", Telecommunications Policy Research Conference (TPRC) 2014, September 2014.

[DR02] Joan Daemen and Vincent Rijmen, "The Design of Rijndael: AES – The Advanced Encryption Standard.", Springer-Verlag, 2002.

[DS78] Yogen Dalal and Carl Sunshine, "Connection Management in Transport Protocols", Computer Networks 2, 1978.

[ID89] Ivan Dåmgard, "A Design Principle for Hash Functions", Advances in Cryptology - CRYPTO '89 Proceedings, Lecture Notes in Computer Science volume 435, Springer-Verlag, 1989.

[DKS89] Alan Demers, Srinivasan Keshav and Scott Shenker, "Analysis and Simulation of a Fair Queueing Algorithm", ACM SIGCOMM Proceedings on Communications Architectures and Protocols, 1989.

[DH76] Whitfield Diffie and Martin Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory, volume IT-22, November 1976.

[EGMR05] Mihaela Enachescu, Ashish Goel, Yashar Ganjali, Nick McKeown and Tim Roughgarden. "Part III: Routers with Very Small Buffers", ACM SIGCOMM Computer Communication Review, volume 35 number 2, July 2005.

[FF96] Kevin Fall and Sally Floyd, "Simulation-based Comparisons of Tahoe, Reno and SACK TCP", ACM SIGCOMM Computer Communication Review, July 1996.

[FRZ13] Nick Feamster, Jennifer Rexford and Ellen Zegura, "The Road to SDN: An Intellectual History of Programmable Networks", ACM Queue, December 2013.

[FGMPC02] Roberto Ferorelli, Luigi Grieco, Saverio Mascolo, G Piscitelli, P Camarda, "Live Internet Measurements Using Westwood+ TCP Congestion Control", IEEE Global Telecommunications Conference, 2002.

[F91] Sally Floyd, "Connections with Multiple Congested Gateways in Packet-Switched Networks, Part 1", ACM SIGCOMM Computer Communication Review, October 1991.

[FJ92] Sally Floyd and Van Jacobson, "On Traffic Phase Effects in Packet-Switched Gateways", Internetworking: Research and Experience, volume 3, pp 115-156, 1992.

[FJ93] Sally Floyd and Van Jacobson, "Random Early Detection Gateways for Congestion Avoidance", IEEE/ACM Transactions on Networking, volume 1, August 1993.

[FJ95] Sally Floyd and Van Jacobson, "Link-sharing and Resource Management Models for Packet Networks", IEEE/ACM Transactions on Networking, volume 3, June 1995.

[FP01] Sally Floyd and Vern Paxson, "Difficulties in Simulating the Internet", IEEE/ACM Transactions on Networking, volume 9, August 2001.

[FMS01] Scott Fluhrer, Itsik Mantin and Adi Shamir, "Weaknesses in the Key Scheduling Algorithm of RC4", SAC '01 Revised Papers from the 8th Annual International Workshop on Selected Areas in Cryptography, Springer-Verlag, 2001.

[FL03] Cheng Fu and Soung Liew, "TCP Veno: TCP Enhancement for Transmission over Wireless Access Networks", IEEE Journal on Selected Areas in Communications, volume 21 number 2, February 2003.

[LG01] Lixin Gao, "On Inferring Autonomous System Relationships in the Internet", IEEE/ACM Transactions on Networking, volume 9, December 2001.

[GR01] Lixin Gao and Jennifer Rexford, "Stable Internet Routing without Global Coordination", IEEE/ACM Transactions on Networking, volume 9, December 2001.

[JG93] Jose J Garcia-Lunes-Aceves, "Loop-Free Routing Using Diffusing Computations", IEEE/ACM Transactions on Networking, volume 1, February 1993.

[GP11] L Gharai and C Perkins, "RTP with TCP Friendly Rate Control", Internet Draft, http://tools.ietf.org/html/draft-gharai-avtcore-rtp-tfrc-00.

[GV02] Sergey Gorinsky and Harrick Vin, "Extended Analysis of Binary Adjustment Algorithms", Technical Report TR2002-39, Department of Computer Sciences, University of Texas at Austin, 2002.

[GM03] Luigi Grieco and Saverio Mascolo, "End-to-End Bandwidth Estimation for Congestion Control in Packet Networks", Proceedings of the Second International Workshop on Quality of Service in Multiservice IP Networks, 2003.

[GM04] Luigi Grieco and Saverio Mascolo, Performance Evaluation and Comparison of Westwood+, New Reno, and Vegas TCP Congestion Control, ACM SIGCOMM Computer Communication Review, volume 34 number 2, April 2004.

[GG03] Marco Gruteser and Dirk Grunwald, "Enhancing Location Privacy in Wireless LAN Through Disposable Interface Identifiers:A Quantitative Analysis", Proceedings of the 1st ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots (WMASH '03), September, 2003.

[HRX08] Sangtae Ha, Injong Rhee and Lisong Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant", ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel, volume 42 number 5, July 2008.

[SH04] Steve Hanna, "Shellcoding for Linux and Windows Tutorial", http://www.vividmachines.com/shellcode/shellcode.html, July 2004

[MH04] Martin Hellman, "Oral history interview with Martin Hellman", Charles Babbage Institute, 2004. Retrieved from the University of Minnesota Digital Conservancy, http://purl.umn.edu/107353.

[JH96] Janey Hoe, "Improving the Start-up Behavior of a Congestion Control Scheme for TCP", ACM SIGCOMM Symposium on Communications Architectures and Protocols, August 1996.

[HVB01] Gavin Holland, Nitin Vaidya and Paramvir Bahl, "A rate-adaptive MAC protocol for multi-Hop wireless networks", MobiCon '01: Proceedings of the 7th annual International Conference on Mobile Computing and Networking, 2001.

[CH99] Christian Huitema, *Routing in the Internet*, second edition, Prentice Hall, 1999.

[HBT99] Paul Hurley, Jean-Yves Le Boudec and Patrick Thiran, "A Note on the Fairness of Additive Increase and Multiplicative Decrease", Proceedings of ITC-16, 1999.

[JK88] Van Jacobson and Michael Karels, "Congestion Avoidance and Control", Proceedings of the Sigcomm '88 Symposium, volume 18(4), 1988.

[JKMOPSVWZ13] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart and Amin Vahdat, "B4: Experience with a Globally-Deployed Software Defined WAN", Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM, August 12-16, 2013.

[JWL04] Cheng Jin, David Wei and Steven Low, "FAST TCP: Motivation, Architecture, Algorithms, Performance", IEEE INFOCOM, 2004.

[KM97] Ad Kamerman and Leo Monteban, "WaveLAN-II: A high-performance wireless LAN for the unlicensed band", AT&T Bell Laboratories Technical Journal, volume 2 number 3, 1997.

[SK88] Srinivasan Keshav, "REAL: A Network Simulator" (Technical Report), University of California at Berkeley, 1988.

[KKCQ06] Jongseok Kim, Seongkwan Kim, Sunghyun Choi and Daji Qiao, "CARA: Collision-Aware Rate Adaptation for IEEE 802.11 WLANs", IEEE INFOCOM 2006 Proceedings, April 2006.

[LK78] Leonard Kleinrock, "On Flow Control in Computer Networks", Proceedings of the International Conference on Communications, June 1978.

[LH06] Mathieu Lacage and Thomas Henderson, "Yet Another Network Simulator", Proceedings of WNS2 '06: Workshop on ns-2: the IP network simulator, 2006.

[LM91] Xuejia Lai and James L. Massey, "A Proposal for a New Block Encryption Standard", EUROCRYPT '90 Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology, Springer-Verlag, 1991.

[LKCT96] Eliot Lear, Jennifer Katinsky, Jeff Coffin and Diane Tharp, "Renumbering: Threat or Menace?", Tenth USENIX System Administration Conference, Chicago, 1996.

[LSL05] DJ Leith, RN Shorten and Y Lee, "H-TCP: A framework for congestion control in high-speed and long-distance networks", Hamilton Institute Technical Report, August 2005.

[LSM07] DJ Leith, RN Shorten and G McCullagh, "Experimental evaluation of Cubic-TCP", Extended version of paper presented at Proc. Protocols for Fast Long Distance Networks, 2007.

[LBS06] Shao Liu, Tamer Basar and R Srikant, "TCP-Illinois: A Loss and Delay-Based Congestion Control Algorithm for High-Speed Networks", Proceedings of the 1st international conference on performance evaluation methodologies and tools, 2006.

[AM90] Allison Mankin, "Random Drop Congestion Control", ACM SIGCOMM Symposium on Communications Architectures and Protocols, 1990.

[MCGSW01] Saverio Mascolo, Claudio Casetti, Mario Gerla, MY Sanadidi, Ren Wang, "TCP westwood: Bandwidth estimation for enhanced transport over wireless links", MobiCon '01: Proceedings of the 7th annual International Conference on Mobile Computing and Networking, 2001.

[McK90] Paul McKenney, "Stochastic Fairness Queuing", IEEE INFOCOM '90 Proceedings, June 1990.

[MABPPRST08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker and Jonathan Turner, "OpenFlow: Enabling innovation in campus networks", ACM SIGCOMM Computer Communications Review, April 2008.

[RM78] Ralph Merkle, "Secure Communications over Insecure Channels", Communications of the ACM, volume 21, April 1978.

[RM88] Ralph Merkle, "A Digital Signature Based on a Conventional Encryption Function", Advances in Cryptology — CRYPTO '87, Lecture Notes in Computer Science volume 293, Springer-Verlag, 1988.

[MH81] Ralph Merkle and Martin Hellman, "On the Security of Multiple Encryption", Communications of the ACM, volume 24, July 1981.

[MB76] Robert Metcalfe and David Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", Communications of the ACM, volume 19 number 7, 1976.

[MW00] Jeonghoon Mo and Jean Walrand, "Fair End-to-End Window-Based Congestion Control", IEEE/ACM Transactions on Networking, volume 8 number 5, October 2000.

[JM92] Jeffrey Mogul, "Observing TCP Dynamics in Real Networks", ACM SIGCOMM Symposium on Communications Architectures and Protocols, 1992.

[MM94] Mart Molle, "A New Binary Logarithmic Arbitration Method for Ethernet", Technical Report CSRI-298, Computer Systems Research Institute, University of Toronto, 1994.

[RTM85] Robert T Morris, "A Weakness in the 4.2BSD Unix TCP/IP Software", AT&T Bell Laboratories Technical Report, February 1985.

[NJ12] Kathleen Nichols and Van Jacobson, "Controlling Queue Delay", ACM Queue, May 2012.

[OKM96] Teunis Ott, JHB Kemperman and Matt Mathis, "The stationary behavior of ideal TCP congestion avoidance", Technical Report, 1996.

[PFTK98] Jitendra Padhye, Victor Firoiu, Don Towsley and Jim Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation", ACM SIGCOMM conference on Applications, technologies, architectures, and protocols for computer communication, 1998.

[PG93] Abhay Parekh and Robert Gallager, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks - The Single-Node Case", IEEE/ACM Transactions on Networking, volume 1 number 3, June 1993.

[PG94] Abhay Parekh and Robert Gallager, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks - The Multiple Node Case", IEEE/ACM Transactions on Networking, volume 2 number 2, April 1994.

[VP97] Vern Paxson, "End-to-End Internet Packet Dynamics", ACM SIGCOMM conference on Applications, technologies, architectures, and protocols for computer communication, 1997.

[PWZMTQ17] Changhua Pei, Zhi Wang, Youjian Zhao, Zihan Wang, Yuan Meng, Dan Pei, Yuanquan Peng, Wenliang Tang and Xiaodong Qu, "Why It Takes So Long to Connect to a WiFi Access Point", IEEE International Conference on Computer Communications, May 2017.

[CP09] Colin Percival, "Stronger Key Derivation Via Sequential Memory-Hard Functions", BSDCan - The Technical BSD Conference, May 2009.

[PB94] Charles Perkins and Pravin Bhagwat, “Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers”, ACM SIGCOMM Computer Communications Review, volume 24 number 4, October 1994.

[PR99] Charles Perkins and Elizabeth Royer, “Ad-hoc On-Demand Distance Vector Routing”, Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications, February 1999.

[RP85] Radia Perlman, “An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN”, ACM SIGCOMM Computer Communication Review 15(4), 1985.

[JP88] John M Pollard, “Factoring with Cubic Integers”, unpublished manuscript circulated 1988; included in “The Development of the Number Field Sieve”, Lecture Notes in Mathematics volume 1554, Springer-Verlag, 1993.

[PDG12] Balaji Prabhakar, Katherine N Dektar and Deborah M Gordon, “The Regulation of Ant Colony Foraging Activity without Spatial Information”, PLoS Computational Biology 8(8), http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1002670

[PN98] Thomas Ptacek and Timothy Newsham, “Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection”, Technical report, Secure Networks Inc, January 1998.

[RJ90] Kadangode Ramakrishnan and Raj Jain, “A Binary Feedback Scheme for Congestion Avoidance in Computer Networks”, ACM Transactions on Computer Systems, volume 8 number 2, May 1990.

[RX05] Injong Rhee and Lisong Xu, “Cubic: A new TCP-friendly high-speed TCP variant,” 3rd International Workshop on Protocols for Fast Long-Distance Networks, February 2005.

[RR91] Ronald Rivest, “The MD4 message digest algorithm”, Advances in Cryptology - CRYPTO ‘90 Proceedings, Springer-Verlag, 1991.

[RSA78] Ronald Rivest, Adi Shamir and Leonard Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”, Communications of the ACM, volume 21, February 1978.

[SRC84] Jerome Saltzer, David Reed and David Clark, “End-to-End Arguments in System Design”, ACM Transactions on Computer Systems, volume 2 number 4, November 1984.

[BS93] Bruce Schneier, “Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)”, Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993), Springer-Verlag, 1994.

[SM90] Nachum Shacham and Paul McKenney, “Packet recovery in high-speed networks using coding and buffer management”, IEEE INFOCOM ‘90 Proceedings, June 1990.

[SP03] Umesh Shankar and Vern Paxson, “Active Mapping: Resisting NIDS Evasion without Altering Traffic”, Proceedings of the 2003 IEEE Symposium on Security and Privacy, 2003.

[SV96] M Shreedhar and George Varghese, “Efficient Fair Queuing Using Deficit Round Robin”, IEEE/ACM Transactions on Networking, volume 4 number 3, June 1996.

[SKS06] Rade Stanojević, Christopher Kellett and Robert Shorten, “Adaptive Tuning of Drop-Tail Buffers for Reducing Queuing Delays”, IEEE Communications Letters, volume 10 number 7, August 2006.

[TSZS06] Kun Tan, Jingmin Song, Qian Zhang and Murari Sidharan, “Compound TCP: A Scalable and TCP-friendly Congestion Control for High-speed Networks”, 4th International Workshop on Protocols for Fast Long-Distance Networks (PFLDNet), 2006.

[TWHL05] Ao Tang, Jintao Wang, Sanjay Hegde and Steven Low, "Equilibrium and Fairness of Networks Shared by TCP Reno and Vegas/FAST", Telecommunications Systems special issue on High Speed Transport Protocols, 2005.

[TWP07] Erik Tews, Ralf-Philipp Weinmann and Andrei Pyshkin, "Breaking 104-bit WEP in less than 60 seconds", WISA'07 Proceedings of the 8th International Conference on Information Security Applications, Springer-Verlag, 2007.

[VMCCP16] Mathy Vanhoef, Célestin Matte, Mathieu Cunche, Leonardo Cardoso and Frank Piessens, "Why MAC Address Randomization is not Enough: An Analysis of Wi-Fi Network Discovery Mechanisms", ACM Asia Conference on Computer and Communications Security, May 2016.

[VGE00] Kannan Varadhan, Ramesh Govindan and Deborah Estrin, "Persistent Route Oscillations in Interdomain Routing", Computer Networks, volume 32, January, 2000.

[SV02] Serge Vaudenay, "Security Flaws Induced by CBC Padding – Applications to SSL, IPSEC, WTLS...", EUROCRYPT '02 Proceedings, 2002.

[WJLH06] David Wei, Cheng Jin, Steven Low and Sanjay Hegde, "FAST TCP: Motivation, Architecture, Algorithms, Performance", ACM Transactions on Networking, December 2006.

[WM05] Damon Wischik and Nick McKeown, "Part I: Buffer Sizes for Core Routers", ACM SIGCOMM Computer Communication Review, volume 35 number 2, July 2005.

[LZ89] Lixia Zhang, "A New Architecture for Packet Switching Network Protocols", PhD Thesis, Massachusetts Institute of Technology, 1989.

[ZSC91] Lixia Zhang, Scott Shenker and David Clark, "Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic", ACM SIGCOMM Symposium on Communications Architectures and Protocols, 1991.

INDEX

Symbols

.onion addresses, 193
/etc/resolv.conf, 193
0-RTT protection, QUIC, 367
0-RTT protection, TLS v1.3, 765
1-RTT protection, QUIC, 367
1-RTT protection, TLS v1.3, 765
127.0.1.1, 193
2-D parity, 147
2.4 GHz, 93
3DES, 742
4B/5B, 127
4G, 112
5 GHz, 93
6LoWPAN, 92
6in4, IPv6, 237
802.11, 92
802.11i, IEEE, 105
802.16, 112
802.1Q, 64
802.1X, IEEE, 106
802.3, IEEE Ethernet, 44

A

A and AAAA records, DNS, 231
A record, DNS, 192
accelerated open, TCP, 353
access point, Wi-Fi, 100
accurate costs, 252
ACD, IPv4, 197
ACK compression, 454
ACK, TCP, 331
acknowledgment, 28
acknowledgment number, TCP, 331
acknowledgments, QUIC, 366
ACKs of unsent data, TCP, 344
ACK[N], 153
active close, TCP, 346
active queue management, 425
active subqueue, 583
ad hoc configuration, Wi-Fi, 100
ad hoc wireless network, 110
adaptive droptail algorithm, 428
additive increase, multiplicative decrease, 376
address, 12
address configuration, manual IPv6, 236
address ownership, QUIC, 368
address randomization, 726
Address Resolution Protocol, 195
AddressFamily, 307
Administratively Prohibited, 203
admission control, RSVP, 634
ADT, 428
advertised window size, 355
AES, 742
AF drop precedence, 640
AfriNIC, 27
agent configuration, SNMP, 680
agent, SNMP, 655
AH, IPsec, 771
AIMD, 376, 423
algorithm, distance-vector, 246
algorithm, DSDV, 254
algorithm, EIGRP, 255
algorithm, Ethernet learning, 57
algorithm, exponential backoff, 49
algorithm, fair queuing bit-by-bit round-robin, 585
algorithm, fair queuing GPS, 587
algorithm, fair queuing, quantum, 593
algorithm, Floyd-Warshall, 574
algorithm, hierarchical weighted fair queuing, 602
algorithm, Karn/Partridge, 357
algorithm, link-state, 256
algorithm, loop-free distance vector, 254

algorithm, Nagle, 355
algorithm, Shortest-Path First, 258
algorithm, spanning-tree, 59
Alice, 735
alignment, memory, 176
all-nodes multicast address, 217
all-routers multicast address, 217
ALOHA, 53
AMI, 129
anternet, 373
anycast address, 215
Aodh, 735
AODV, 254, 541
APNIC, 27
AQM, 425
ARC4, 743
ARCFOUR, 743
architecture, network, 654
argon2, 738
ARIN, 27
arithmetic, fast, 750
ARP, 195
ARP cache, 196
ARP failover, 198
ARP spoofing, 198
ARPANET, 35
ARQ protocols, 153
AS-path, 277
AS-set, 279
ASLR, 726
ASN.1, 658, 660
ASN.1 enumerated type, 676
association, Wi-Fi, 100
Assured Forwarding, 640
Assured Forwarding PHB, 638
asymmetric routes, 274
Asynchronous Transfer Mode, 86
at-least-once semantics, 322
ATM, 13, 86
augmentation, SNMP, 688
authentication header, IPsec, 771
authentication with secure-hash functions, 736
authenticator, WPA, 106
authoritative nameserver, 192
autoconfiguration, IPv4, 200
autonomous system, 245, 267, 276

B

B8ZS, 129
backbone, 26
backoff, Ethernet, 49
backup link, BGP, 285
backwards compatibility, TCP, 444
bad news, distance-vector, 247
band width, wireless, 89
bandwidth, 12
bandwidth $\times$ delay, 140, 158
bandwidth delay, 137
bandwidth guarantees, 610
base station, WiMAX, 113
basic encoding rules, 681
BBR TCP, 467
BBRR, 585
bcrypt, 738
BDP, 158
beacon packets, Wi-Fi, 101
beacon, Wi-Fi, 104
beefcafe, 236
BER, 681
Berkeley Packet Filter, 562
Berkeley Unix, 37
best-effort, 23, 28
best-path selection, BGP, 280
BGP, 275, 290
BGP relationships, 288
BGP speaker, 277
big-endian, 13
binary data, 309
bind(), 304
bit stuffing, 128
bit-by-bit round robin, 585
BLAM, 51
Blowfish, 742
Bluetooth, 92
Bob, 735
boot counter, 323
border gateway protocol, 275
border routers, 253
bottleneck link, 161, 402
BPDU message, spanning tree, 60
bpf, 562
bps, 12, 20
broadcast IPv4 address, 178
broadcast, Ethernet, 21

broadcast, MANETs, 111
BSD, 37
buffer overflow, 32
buffer overflow, heap, 728
bufferbloat, 392, 595
bugs, 772
byte stuffing, 128

C

CA, 758
cache, DNS, 192
CAM table, 59
canonical name, DNS, 195
Canopy, 118
capture effect, Ethernet, 51
care-of address, 206
carrier Ethernet, 80
CBC mode, 743
CDN and IntServ, 635
CDNs, 31
cell-loss priority bit, ATM, 86
certificate authorities, 754
certificate authority, 758
certificate pinning, 760
certificate revocation, 761
certificate revocation list, 761
CFB mode, 745
CGA, 223
challenge-handshake authentication, 738
channel width, 89
channel, Wi-Fi, 93
chap authentication, 738
checksum offloading, 56, 337
checksum, TCP, 330
checksum, UDP, 299
Christmas day attack, 352
CIDR, 267
cipher feedback mode, 745
cipher modes, 742
Cisco, 64, 255, 284
class A/B/C addressing, 22
Class Selector PHB, 637
class, queuing discipline, 581
classful queuing discipline, 581
Classless Internet Domain Routing, 267
clear-to-send, Wi-Fi, 96
client, 28
client-server, 31
ClientHello, TLS, 763
ClientKey, 738
ClientSignature, 738
cliff, 18, 375
CLNP, 36
clock recovery, 125
clock synchronization, 125
close, TCP, 333, 347
CMIP, 654
CMNS, 36
CMOT, 654
CNAME, DNS, 195
CoDel, 428
collision, 20, 43
collision attack, MD5, 735
collision avoidance, 95
collision detection, 44, 52, 54
collision detection, wireless, 88
collision domain, 56
collision problem, hash, 735
collision, Wi-Fi, 94
commands, Mininet, 546
community attribute, BGP, 284, 285
community, SNMP, 680
concave cwnd graph, 447
configuring FreeRADIUS, 107
configuring WPA2-Enterprise, 107
congestion, 18, 24, 29
congestion avoidance phase, 375
congestion bit, 425
congestion window, 374
connect(), 307, 309
connection, 12
connection table, virtual circuit, 83
connection-oriented, 329
connection-oriented networking, 23
connectionless networking, 23
conservative, 35
Content-Distribution Networks, 31
contention, 18
contention interval, 52
contributing source, RTP, 644
control packet, Wi-Fi, 92
control packet, Wi-Fi ACK, 94
control packet, Wi-Fi RTS/CTS, 96
convergence to TCP fairness, 424, 444
convex cwnd graph, 447
counter mode, 744

CRC code, 145
cross-site scripting, 731
cryptographically generated address, 193, 223
CSMA, 52
CSMA persistence, 52
CSMA/CA, 95
CSMA/CD, 44
CSRC, 644
CTR mode, 744
CTS, Wi-Fi, 96
Cubic, TCP, 463
cumulative ACK, 157
customer, BGP, 288
cut-through, 19
cwnd, 374
CWR, 331, 426
cyclical redundancy check, 145

D

DAD, IPv6, 224
DALLY, TFTP, 316
Data Center TCP, 459
data rate, 12
data types, SNMPv1, 659
datacenter, 69
Datagram Congestion Control Protocol, 301
datagram forwarding, 12
Data[N], 153
DCCP, 301
DCCP congestion control, 422
DCCP connection establishment, 361
DCTCP, 459
deadbeef, 294
DECbit, 425
DECnet, 425
default forwarding entry, 181
default route, 15, 26
deflation, cwnd, 386
delay constraints, 613
delay, bandwidth, 137
delay, largest-packet, 143
delay, propagation, 137
delay, queuing, 137
delay, store-and-forward, 137
delay-based congestion control, 449
delayed ACKs, 355, 380
denial of service attack, 175
dense wavelength-division multiplexing, 134
DES, 742
Destination Unreachable, 203
DHCP, 199
DHCP Relay, 201
DHCPv6, 224, 227
Differentiated Services, 174, 637
Diffie-Hellman-Merkle key exchange, 748
DiffServ, 637
DIFS, Wi-Fi, 94, 110
dig, 33
digital signatures, RSA, 751
discrete logarithm problem, 749
distance vector, loop-free versions, 254
distance-vector, 26
distance-vector routing update, 246
distribution network, Wi-Fi, 104
DIX Ethernet, 44
DNS, 27, 191, 200, 305, 307, 308, 320, 341
DNS A and AAAA records, 231
DNS and IPv6, 224, 226, 231
DNS, round-robin, 193
Domain Name System, 191
domain name system, 27
Dont Fragment bit, 179
doze mode, Wi-Fi, 102
draft standard, 36
drop precedence, DiffServ AF, 640
DS, 174
DS domain, 637
DS field, IPv4 header, 637
DS1 line, 130
DS3 line, 131
DSDV, 254, 541
DSO channel, 130
dual stack, 231
DuckDuckGo, 193
dumbbell network topology, 410
duplicate address detection, IPv4, 197
duplicate connection request, 316
duplicate-address detection, IPv6, 224
durian, 695
DWDM, 134
dynamic rate scaling, 97

E

EAP, 106
EAPOL, 106
ECB mode, 742

ECE, 331, 426
Echo Request/Reply, 202
ECMP, 571
ECN, 174, 426
ECN and VPNs, 80
ECT, 426
EFS, 384
EGP, 278
EIGRP, 255
elephant flow, 260
elevator algorithm, 323
elliptic-curve cryptography, 751
eNB, LTE, 113
encapsulated security payload, IPsec, 771
encoding, 309
encrypt-and-MAC, 745
encrypt-then-MAC, 745
encryption, 739
end-to-end encryption, 754
End-to-End principle, 153, 330
engines, SNMPv3, 705
enumerated type, SNMP, 676
equal-cost multipath routing, 571
error message, ICMP, 202
error-correcting code, 146
error-detection code, 143
ESP, IPsec, 771
ESS, Wi-Fi, 104
estimated flightsize, 384
Ethernet, 19
Ethernet address, 20
Ethernet hub, 44
Ethernet repeater, 44
Ethernet switch, 56
ethtool, linux, 337
Euclidean algorithm, 750
EUI-64 identifier, 213
evasion, intrusion detection, 733
exactly-once semantics, 322
Expedited Forwarding, 638
Expedited Forwarding PHB, 637
expiration date, certificate, 761
Explicit Congestion Notification, 174, 426
exponential backoff, Ethernet, 49
exponential backoff, Wi-Fi, 95
exponential growth, 378
export filtering, BGP, 280
extended interface table, SNMP, 690
extended-validation certificates, 760
Extensible Authentication Protocol, 106
extension headers, 217
exterior routing, 275
extreme TCP unfairness, 416

F

face:b00c, 232
facebookcorewwwi, 193
factoring RSA keys, 752
fading, 90
fair queuing, 582
fair queuing and AF, 640
fair queuing and EF, 639
fairness, 29
fairness, TCP, 402, 410, 417, 444
fallback to flooding, 57
fast arithmetic, 750
Fast Ethernet, 54
Fast Open, TCP, 354
fast primality testing, 750
Fast Recovery, 384
fast retransmit, 382
fastest sequence, token-bucket, 609
FCAPS, 653
Feistel network, 740
Fibonacci sequence, 619
FIFO queuing, 401
fill time, voice, 130
filtering, BGP, 279
filterspec, 633
FIN, 331
FIN packet, 333
finishing order, WFQ, 589
firewall, 32, 65, 68
fixed wireless, 116
flights of packets, 375
flightsize, 468
flightsize, estimated, 384
flow control, 157
flow control, QUIC, 366
flow control, TCP, 356
Flow Label, 212
flow specification, 605
flow tables, 66
flow, IPv6, 212
flow, TCP, 569
flowspec, RSVP, 633

Floyd-Warshall algorithm, 574
fluid model, fair queuing, 587
foraging, 373
foreign agent, 206
fortune, 193
forward secrecy, 753
forwarding delay, 19
forwarding table, 13
forwarding, IP, 24
forwarding, MANET, 112
four-way handshake, Wi-Fi, 105
foxes, 373
fragment header, IPv6, 219
fragment offset, 179
fragmentation, 23
Fragmentation Required, 203
fragmentation, IP, 178
fragmentation, Wi-Fi, 97
frame, 12
frames, QUIC, 365
framing, 128
FreeRADIUS, 107
frequency band, 89
Friendliness, TCP, 420
Frost, Robert, 82
full-duplex Ethernet, 55
fwmark, 260, 562

G

generalized processor sharing, 587
generic hierarchical queuing, 597
geographical routing, 274
GET request, HTTP, 343
getaddrinfo(), 231
getAllByName(), Java, 305
getAllByName(), java, 307
GetBulk, SNMPv2, 685
getByName(), 232
getByName(), Java, 231
getByName(), java, 307
gethostbyname(), 231
gigabit Ethernet, 55
glibc-2.2.4, 728
global scope, IPv6 addresses, 228
goodput, 12, 479
GPS, fair queuing, 587
granularity, loss counting, 507
gratuitous ARP, 197
greediness in TCP, 414
group, OpenFlow, 571
GTC, EAP, 107

H

H-TCP, 462
half-closed, TCP, 346
half-open, TCP, 346
Hamilton TCP, 462
Hamming code, 147
Handshake packet, QUIC, 367
handshake protocol, TLS, 762
Happy Eyeballs, IPv6, 233
hard fail, OCSP, 762
Hash Message Authentication Code, 736
hash, password, 737
HDLC, 128
head-of-line blocking, 29, 300, 627
header, 12
header, Ethernet, 45
header, IPv4, 173
header, IPv6, 211
header, TCP, 330
header, UDP, 299
heap buffer overflow, 727
heap vulnerability, 728
Hellman (Diffie-Hellman-Merkel), 748
Hello, spanning-tree, 60
henhouse, 373
hidden node problem, 89
hidden-node collisions, 96
hidden-node problem, 96
hierarchical routing, 182, 267, 269
hierarchical token bucket, 613
hierarchical token bucket, linux, 615
high-bandwidth TCP problem, 429
Highspeed TCP, 445
history, RMON, 698
HMAC, 707, 736
hold down, 251
home address, 206
home agent, 206
host command, 33
host key, ssh, 756
Host Top N, RMON, 700
Host Unreachable, 203
host-specific forwarding, 80
hot-potato routing, 274

htb, linux, 560, 615
htonl, 310
htons, 310
HTTP, 343, 350
https, 755
hub, Ethernet, 44
hulu, 625
Hybla, TCP, 459

I

IAB, 34
IANA, 22, 27, 677
ICANN, 192
ICMP, 201
ICMPv6, 229
IDEA, 742
idempotency and 0-RTT, 765
idempotent, 322
IDENT field, 179
IEEE, 44, 47
IEEE 802.11, 92
IEEE 802.1Q, 64
IEEE 802.1X, 106
IEEE 802.3 Ethernet, 44
IETF, 34
ifconfig, 33
ifDescr, 676
ifIndex, 675
IFS, Wi-Fi, 94
ifType, SNMP, 676
ifXTable, SNMP, 690
IKEv2, 773
Illinois, TCP, 455
implementations, at least two, 36
import filtering, BGP, 280
incarnation, connection, 313
incast, TCP, 462
inetCidrRouteTable, 694
inflation, cwnd, 386
infrastructure configuration, Wi-Fi, 100
Initial packet, QUIC, 367
initial sequence number, TCP, 331, 335, 350
initialization vector, 743
instability, BGP, 292
integrated services, 627
integrated services, generic, 623
interconnection fabric, 69
interface, 176
interface identifier, ipv6, 213
interface table, extended, SNMP, 690
interior routing, 245
Internet Architecture Board, 34
Internet checksum, 144
Internet Engineering Task Force, 34
Internet exchange point, 272
Internet Key Exchange, 773
Internet Society, 34
intersymbol interference, 90
intrusion detection, 732
IntServ, 627
IP, 11, 21
ip command (linux), 33
IP forwarding, 24
IP fragmentation, 178
IP multicast, 628
IP network, 23
IP-in-IP encapsulation, 207
ipconfig, 33
IPsec, 771
iptables, 260, 562
IPv4 header, 173
IPv6, 211, 216
IPv6 address configuration, manual, 236
IPv6 addresses, 212
IPv6 connections, link-local, 236
IPv6 extension headers, 217
IPv6 header, 211
ipv6 interface identifier, 213
IPv6 link-local connections, 236
IPv6 multicast, 216
IPv6 Neighbor Discovery, 220
IPv6 programming, 305
IPv6 tunnel, 237
irregular prime, 25
ISM band, 93
ISN, 335, 350
ISOC, 34
ISP, 15
IV, 743
IXP, 272

J

jail, staying out of, 272
Java getAllByName(), 305
java getAllByName(), 307
Java getByName(), 231

java getByName(), 307
javascript, 731
jitter, 30, 392, 625, 646
join, multicast, 630
joining a Wi-Fi network, 101
JPEG heap vulnerability, 730
JSON, 311
jumbogram, IPv6, 218

K

Karn/Partridge algorithm, 357
KB, 12
KeepAlive, TCP, 358
key exchange, TLS, 764
key-scheduling algorithm, RC4, 744
key-signing parties, 754
keystream, 743
KiB, 12
kings, 35
knee, 18, 375, 449
known_hosts, ssh, 756

L

LACNIC, 27
ladder diagram, 139
LAN, 11, 19
LAN layer, 47
LARTC, 259
latching on, TFTP, 312
layer 3 switch, 187
layers, 11
leaky bucket, alternative formulation, 608
learning, Ethernet switch, 21, 57, 67
legacy routing, 270
length-extension vulnerability, 736
liberal, 35
licensing this book, 3
lightning, 134
link-layer ACK, 94
link-local address, 214
link-state packets, 257
link-state routing update, 256
linux, 100, 196, 199, 205, 235, 236, 441, 455, 463, 581
linux advanced routing, 259
linux htb, 615
linux IPv6 routing, 220
linux sfq, 594
Lisp, 741
listen, 28
little-endian, 13
load balancer, Pox, 571
load-balancing, 286
load-balancing, SDN, 70
load-balancing, traditional, 71
local traffic, 279
local-area network, 19
logical link layer, 11
logjam attack, 749
lollipop numbering, 257
longest-match rule, 269, 273
loopback address, 177
loopback interface, 176
loss recovery, sliding windows, 160
loss synchronization, 416
loss synchronization, TCP, 414
loss-based congestion control, 449
loss-tolerant, 300, 624
lossy-link TCP problem, 431
lost final ACK, 315
lost final ACK, TFTP, 316
LRO, TCP, 337
LSA, 257
LSO, TCP, 337
LSP, 257
LTE, 112

M

MAC address, 20
MAC address randomization, 103
MAC layer, 47
MAC-then-encrypt, 745
MAE, 272
MAE-East, 272
man-in-the-middle attack, 749, 753
managed device, SNMP, 655
management frame protection, Wi-Fi, 109
management packet, Wi-Fi, 92, 101
manager, SNMP, 655
Manchester encoding, 126
MANET, 110
mangling, 67, 260
manual IPv6 address configuration, 236
MapReduce, 462
market, IPv4 addresses, 27
master secret, TLS, 764

Matrix, RMON, 701
max-min fairness, 417
Maximum Transfer Unit, 178
MB, 12
Mbone, 632
Mbps, 12, 20
MD5, 735
MED, 274, 281, 282
media-access control, 47
Merkle (Diffie-Hellman-Merkel), 748
Merkle-Dåmgard construction, 735
mesh network, 110
MiB, 12
MIB browsers, 671
MIB-2, 672
MIC, 106
Michael, 106
Microsoft SNMP agent, 681
middlebox, 190
middleboxes, 335
middleboxes and ECN, 426
MIMO, 98
minimal network configuration, 200
minimizing route cost, 252
minimum RTO, TCP, 461
mininet, 542
Mininet commands, 546
MISO, 99
Mitnick, Kevin, 20, 352
mixer, RTP, 644
mobile IP, 206
mobile wireless network, 110
modified EUI-64 identifier, 213
MPLS, 647
MPTCP, 359
msieve, 776
MTU, 178
multi-exit discriminator, 281, 282
multi-protocol label switching, 647
multicast, 216
multicast address allocation, 632
multicast IP address, 178
multicast programming, 552
multicast subscription, 630
multicast tree, 628
multicast, Ethernet, 46
multicast, IP, 22, 628
multihomed, 177, 272
multihoming and ARP, 199
multihoming and TCP, 338
multihoming and UDP, 304
multipath interference, 90
Multipath TCP, 359
multiple flow tables, 68
multiple losses, 444
multiple token buckets, 609
MUST, 34
MX records, DNS, 195

N

Nagle algorithm, 355
nameserver, 192
NAT, 187, 207, 238, 299, 330
NAT and ICMP, 203
NAT and IPsec, 774
NAT problems, 189
NAT-PT, IPv6-to-IPv4, 240
NAT64, 241
nc, 308
Neighbor Advertisement, IPv6, 222
neighbor discovery security, 222
Neighbor Discovery, IPv6, 220
Neighbor Solicitation, IPv6, 221
net neutrality, 624
Net-SNMP, 655, 659, 671, 673, 681
Net-SNMP and SNMPv3, 711
netcat, 34, 308, 342
netcat HTTP GET request, 343
netem queue, 555
netsh (Windows), 33
netstat, 34, 345
network address, 23
network address translation, 187
network architecture, 654
network entry, WiMAX, 116
Network File System, 322
network interface, Ethernet, 20
network management, 580
Network Management System, 654
network model
    five layer, 11
    four layer, 11
    seven layer, 35
network number, 23
network prefix, 23
network prefix, IPv6, 215

Network Unreachable, 203
NewReno, TCP, 387
next_hop, 13
NEXT_HOP attribute, BGP, 282
NFS, 322
NMS, 654
no-transit, BGP, 285
no-valley theorem, BGP, 291
node information message, IPv6, 230
non-compliant, token-bucket, 605
non-congestive loss, 431
non-executable stack, 727
non-recursive DNS lookup, 193
non-repudiation, 735, 737
nonce, 738
nonce, cryptographic, 763
nonpersistence, 52
NOPslide, 725
noSuchObject, SNMP, 685
NoTCP Manifesto, 300
NRZ, 125
NRZI, 126
NS record, DNS, 192
ns-2 trace file, 482
ns-2 tracefiles, reading with python, 485
NSFNet, 35
NSIS, 641
nslookup, 33, 194, 232
ntohl, 310
ntohs, 310
number-field sieve, 752
NX page bit, 727

O

Object ID, SNMP, 656
OBJECT-IDENTITY, SNMP, 685
OBJECT-TYPE, SNMP, 660
OC-3, 132
OCSP, 761
OCSP stapling, 762
OFDM, 89
offloading, TCP, 337
OID, SNMP, 656
old duplicate packets, 313
old duplicates, TCP, 348, 349
OLSR, 541
one-time pad, 743
ones-complement, 144
onion addresses, 193
OpenBSD, 727
OpenFlow, 66
openssl, 108
openSSL programming, 765
Optical Transport Network, 133
optimistic DAD, 225
opus, 625
orthogonal frequency-division multiplexing, 89
OSI, 34
OSPF, 257
OTN, 133
overhead, ns-2, 499

P

packet loss rate, 418
packet pairs, 409
packet size, 141
PAP, EAP, 107
Parekh-Gallager claim, 592
Parekh-Gallager theorem, 617
parking-lot topology, 417
partial ACKs, 387
passive close, TCP, 346
password hash, 737
password sniffing, 20, 199
path attributes, BGP, 281
path bandwidth, 161
Path MTU Discovery, 180, 203
path MTU discovery, TCP, 354
PATH packet, RSVP, 633
PAWS, TCP, 353
PBKDF2, 738
PCF Wi-Fi, 109
PEAP, 107
peer, BGP, 288
peer-to-peer, 31
per-hop behaviors, DiffServ, 637
perfect forward secrecy, 753
persist timer, TCP, 356
persistence, 52
phase effects, TCP, 496
PHBs, DiffServ, 637
physical address, Ethernet, 20
physical layer, 11
PIFS, Wi-Fi, 110
PIM-SM, 630
ping, 32, 202

ping6, 235
pinning, TLS certificates, 760
pipe drain, 160
pipelining, SMTP, 148
PKI, 754
playback buffer, 625
PMK, 802.1X, 106
Pogonomyrmex, 373
Point of Presence, 31
point-to-point protocol, 128
poison reverse, 251
policing, 605
policy-based routing, 66, 259
Pollard's rho algorithm, 776
polling mode, Wi-Fi, 109
polling, TCP, 356
POODLE vulnerability, 755
Port Control Protocol, 190
port exhaustion, TCP, 350
port numbers, UDP, 299
Port Unreachable, 203
potatoes, hot, 274
power and wireless, 92
power management, Wi-Fi, 102
Pox controller, 563
PPP, 128
PPPoE, 128
pre-master secret, TLS, 764
prefix information, IPv6, 221
presentation layer, 35
presidents, 35
primality testing, fast, 750
Prime Number Theorem, 750
primitive root modulo p, 749
priority for small packets, WFQ, 589
priority queuing, 402, 580
priority queuing and AF, 640
privacy, 88
privacy extensions, SLAAC, 226
private IPv4 address, 177
private IPv6 address, 216
PRNG, 743
probe request, Wi-Fi, 103
probing, IPv6 networks, 214
promiscuous mode, 46
propagation delay, 137
proportional fairness, 418
protection against wrapped segments, TCP, 353
protocol graph, 36
protocol-independent multicast, 630
provider, BGP, 288
provider-based routing, 270
proxy ARP, 197
pseudorandom number generator, 743
PSH, 331
PSK, WPA2, 105
PTK, 105
public-key encryption, 750
public-key infrastructure, 754
pulse stuffing, TDM, 131
push, TCP, 331
python tracefile script, 488–490, 513
python, reading ns-2 tracefiles, 485

Q

QoS, 623
QoS and routing, 245
quality of service, 15, 23, 245, 623
quantum algorithm, fair queuing, 593
Query Identifier, ICMP, 202
query, ICMP, 202
queue capacity, typical, 392
queue overflow, 24
queue utilization, token bucket, 611
queue-competition rule, 403
queuing delay, 137
queuing discipline, 581
queuing theory, 401
queuing, priority, 580
QUIC, 300, 363
quiet time on startup, TCP, 353

R

race condition, Pox, 570
RADIUS, 106
radvd, 220, 239
random drop queuing, 401
Random Early Detection, 427
randomization of MAC addresses, 103
ranging intervals, WiMAX, 115
ranging, WiMAX and LTE, 114
rate control, Wi-Fi, 97
rate scaling, Wi-Fi, 97
rate-adaptive traffic, 625
RC4, 743
Real-Time Protocol, 422

real-time traffic, 623, 625
Real-time Transport Protocol, 30
reassembly, 23
reboot and RPC, 323
reboots, 317
Record Route, IP option, 175
recursive DNS lookup, 193
RED, 427
Reed-Solomon codes, 134
regional registry, 27
registry, regional, 27
reliable, 329
reliable flooding, 257
Remote Procedure Call, 320
rendezvous point, multicast, 630
Reno, 37, 375
Reno vs Vegas, 516
repeater, Ethernet, 44
request for comment, 34
request-to-send, Wi-Fi, 96
request/reply, 30, 320, 329
reservations, 23
reset, TCP, 331
resolver, DNS, 192
RESV packet, RSVP, 633
retransmit-on-duplicate, 155
retransmit-on-timeout, 153
return-to-libc attack, 720
reverse DNS, 195
RFC, 34
- RFC 1034, 191
- RFC 1065, 656
- RFC 1066, 672
- RFC 1122, 15, 174, 197, 199, 299, 304, 336, 338, 348, 350, 355, 357, 358, 371
- RFC 1123, 318
- RFC 1155, 656, 658, 660, 661, 663
- RFC 1213, 656, 659, 672–674, 676, 677, 679, 693, 694, 696
- RFC 1271, 696, 698
- RFC 1312, 692
- RFC 1321, 735
- RFC 1323, 353
- RFC 1350, 311, 314, 318
- RFC 1354, 692, 693
- RFC 1441, 656
- RFC 1442, 656, 685
- RFC 1450, 689
- RFC 1518, 268
- RFC 1519, 268
- RFC 1550, 211
- RFC 1644, 353
- RFC 1650, 691
- RFC 1661, 79
- RFC 1700, 13, 310
- RFC 1812, 205
- RFC 1831, 322
- RFC 1854, 148
- RFC 1883, 211
- RFC 1884, 211
- RFC 1901, 656, 684
- RFC 1909, 684
- RFC 1948, 352
- RFC 1981, 354
- RFC 1994, 738
- RFC 2001, 386
- RFC 2003, 207
- RFC 2011, 692
- RFC 2026, 36
- RFC 2096, 693, 694
- RFC 2104, 707, 736
- RFC 2119, 35
- RFC 2131, 199
- RFC 2136, 226
- RFC 2233, 678
- RFC 2264, 656
- RFC 2309, 428
- RFC 2362, 630
- RFC 2386, 245
- RFC 2433, 738
- RFC 2453, 246, 252, 552
- RFC 2460, 144, 211, 212, 218, 219, 330
- RFC 2461, 220
- RFC 2464, 217
- RFC 2473, 218
- RFC 2474, 174
- RFC 2481, 174, 426
- RFC 2529, 237
- RFC 2570, 704
- RFC 2574, 704
- RFC 2578, 656, 663, 682, 685
- RFC 2579, 702
- RFC 2581, 355, 381
- RFC 2582, 387
- RFC 2597, 638, 640, 641
- RFC 2616, 343

RFC 2675, 218
RFC 2759, 738
RFC 2766, 241
RFC 2780, 218
RFC 2786, 708
RFC 2819, 696, 697
RFC 2827, 175, 352
RFC 2851, 695
RFC 2856, 685
RFC 2863, 673, 675–677, 690, 694
RFC 2865, 106
RFC 2898, 738
RFC 2925, 703
RFC 3022, 188, 203
RFC 3056, 237
RFC 3168, 426
RFC 3207, 764
RFC 3246, 638, 639
RFC 3261, 189, 647
RFC 3360, 335, 426
RFC 3410, 704
RFC 3411, 706
RFC 3414, 656, 704, 705, 707, 710
RFC 3415, 681, 704, 709
RFC 3418, 689
RFC 3448, 421
RFC 3465, 380, 535
RFC 3513, 213
RFC 3519, 207
RFC 3540, 427
RFC 3550, 645, 647
RFC 3551, 645
RFC 3561, 254
RFC 3635, 691
RFC 3649, 430, 445–447
RFC 3715, 774
RFC 3748, 106
RFC 3756, 222
RFC 3775, 221
RFC 3826, 705
RFC 3833, 195
RFC 3879, 216
RFC 3927, 215
RFC 3947, 774
RFC 3948, 774
RFC 3971, 223
RFC 3972, 223
RFC 4001, 695
RFC 4007, 213, 236
RFC 4022, 694
RFC 4033, 195
RFC 4034, 195
RFC 4080, 641
RFC 4193, 216
RFC 4213, 237
RFC 4251, 755
RFC 4252, 755
RFC 4253, 755–757
RFC 4271, 276
RFC 4273, 673
RFC 4291, 211, 213–215
RFC 4292, 693, 694
RFC 4293, 692
RFC 4294, 220
RFC 4301, 772
RFC 4303, 219, 772
RFC 4340, 301
RFC 4341, 423
RFC 4342, 423
RFC 4344, 755
RFC 4380, 176
RFC 4429, 225
RFC 4443, 229
RFC 4451, 284
RFC 4472, 232
RFC 4502, 696
RFC 4524, 657
RFC 4560, 703
RFC 4566, 645
RFC 4620, 230
RFC 4681, 401
RFC 4861, 214, 220, 222, 223
RFC 4862, 224, 225
RFC 4919, 92
RFC 4941, 226
RFC 4953, 335
RFC 4961, 644
RFC 4966, 241
RFC 5095, 218
RFC 5116, 368
RFC 5227, 197
RFC 5246, 758
RFC 5247, 106
RFC 5280, 657
RFC 5321, 656, 764
RFC 5348, 421

RFC 5508, 203
RFC 5694, 31
RFC 5802, 738
RFC 5925, 352
RFC 5944, 206
RFC 5952, 213
RFC 5961, 335
RFC 5974, 641
RFC 6040, 80
RFC 6052, 241
RFC 6057, 642
RFC 6093, 336
RFC 6105, 224
RFC 6106, 226
RFC 6125, 759
RFC 6146, 241
RFC 6147, 241
RFC 6151, 737
RFC 6164, 230
RFC 6182, 359
RFC 6275, 218
RFC 6282, 92
RFC 6298, 462, 627
RFC 6356, 359
RFC 6472, 279
RFC 6553, 218
RFC 6554, 218
RFC 6564, 218, 219
RFC 6582, 387
RFC 6633, 203, 427
RFC 6724, 226, 232, 233
RFC 6742, 232
RFC 6824, 359
RFC 6887, 190
RFC 6891, 195
RFC 6918, 203
RFC 6960, 761
RFC 7045, 219
RFC 7217, 214, 215, 225, 226
RFC 7296, 80, 773
RFC 7413, 354
RFC 7421, 211
RFC 7469, 761
RFC 7540, 343
RFC 7567, 392, 428
RFC 7568, 758
RFC 7686, 193
RFC 7707, 214
RFC 7721, 227
RFC 783, 311, 318
RFC 7844, 103
RFC 7860, 705
RFC 791, 22, 174, 179
RFC 792, 201
RFC 793, 330, 332, 335, 336, 343, 345, 369
RFC 821, 656
RFC 8257, 459, 460
RFC 8305, 233
RFC 854, 336
RFC 896, 355, 356
RFC 917, 182
RFC 950, 182, 183
RFC 970, 582
RFC 988, 22
Rinjdael, 742
RIPE, 27
RMON, 695
roaming, Wi-Fi, 104
root nameserver, 192
round-robin DNS, 193
round-trip time, 139
route, 34
router, 15
Router Advertisement, IPv6, 220
Router Discovery, IPv6, 220
Router Solicitation, IPv6, 220
routerless IPv6 examples, 235
routing and addressing, 173
routing domain, 245, 267
routing header, IPv6, 218
routing loop, 17, 250
routing loop, ephemeral, 257
routing policies, BGP, 279
routing policy database, 260
routing update algorithms, 243
routing, IP, 24
routing, MANET, 112
RPC, 320
RSA, 750
RSA factoring challenge, 752
RST, 331
RSVP, 627, 632
RTCP, 646
RTCP measurement of, 646
RTO, TCP, 357
RTO, TCP minimum, 461

RTP, 30, 422
RTP and VoIP, 645
RTP mixer, 644
RTS, Wi-Fi, 96
RTT, 139
RTT bias in TCP, 414
RTT inflation, 163
RTT-noLoad, 140

S

SACK TCP, 389
SACK TCP in ns-2, 511
Salsa20, 741
salt, password, 738
satellite Internet, 118
satellite-link TCP problem, 431
sawtooth, TCP, 376, 380, 430, 481, 507, 518
scalable routing, 173
scanning, IPv6 networks, 214
scope, IPv6 address, 213
scope, IPv6 link-local, 214
SCRAM authentication, 738
scrypt, 738
SCTP, 360
SDN, 65
search warrant, 754
secure hash functions, 735
secure neighbor discovery, 223
secure shell, 755
security, 715
security association, IPsec, 772
segment, 12
segmentation, 23
segments, TCP, 331
select group, 571
select() call, 554
Selective ACKs, TCP, 389
selective export property, 290
selector, IPsec, 772
self-ARP, 197
self-clocking, 158
SEND, 223
sequence number, TCP, 331
serial execution, RPC, 323
server, 28
ServerHello, TLS, 763
session key, 739
session layer, 35
sfq, 594
SHA-1, 735
SHA-2, 735
Shannon-Hartley theorem, 89
shaping, 605
shared-key ciphers, 739
shellcode, 722
Shortest-Path First algorithm, 258
SHOULD, 34
shutdown, TCP, 346, 347
sibling family, BGP, 290
sibling, BGP, 288
SIFS, Wi-Fi, 94
signaling losses, 427
signatures, intrusion, 733
signatures, RSA, 751
silly window syndrome, TCP, 356
SIMO, 99
Simple Network Management Protocol, 654
SimpleHTTPServer, 549
simplex-talk, TCP, 338
simplex-talk, UDP, 301
simultaneous open, TCP, 345
single link-state, 26
single-responsibility principle, 302
singlebell network topology, 405
site-local IPv6 address, 216
size, packet, 141
SLAAC, 224, 225
SLAAC privacy extensions, 226
sliding windows, 29, 157
sliding windows, TCP, 354
slot time, Wi-Fi, 94
slow convergence, 250
small-packet priority, WFQ, 589
SMI, 661
SMTP, 148, 764
SNMP, 654, 694
SNMP agent configuration, 680
SNMP agents and managers, 655
SNMP enumerated type, 676
SNMP versions, 656
SNMPv1 data types, 659
SNMPv3 engines, 705
Snorri Sturluson, 108
SO_LINGER, TCP, 348
socket, 28
socket address, 28

soft fail, OCSP, 762
soft state, 627
software-defined networking, 65
SONET, 131
Sorcerer's Apprentice bug, 156
Source Quench, 202, 426
source-specific multicast tree, 631
spanning-tree algorithm, 59
sparse-mode multicast, 630
spatial streams, MIMO, 100
speex, 625
split horizon, 251
spoofing, IP, 175
spoofing, TCP, 335
SQL injection, 732
SSH, 198
ssh, 754, 755
ssh host key, 756
SSID, Wi-Fi, 100, 104
ssl, 754
SSL programming, 765
SSRC, 644
stack canary, 726
stalk, TCP, 338
stalk, UDP, 301
star topology, 44
STARTTLS, 764
state diagram, TCP, 343
stateless autoconfiguration, 224
stateless forwarding, 15
STM-1, 132
stochastic fair queuing, 594
stop-and-wait transport, 153
stop-and-wait, TFTP, 318
store-and-forward, 19
store-and-forward delay, 137
StoredKey, 738
stream ciphers, 743
Stream Control Transmission Protocol, 360
stream-oriented, 329
streaming video, 30, 626
streams, QUIC, 365
STS-1, 131
STS-3, 132
subnet mask, 183
subnets, 182
subnets, IPv6, 216, 230
subnets, vs switching, 186
subpoena, 754
subqueue, 581
subscription, multicast, 630
Sun RPC, 322
superfish, 760
supplicant, WPA, 106
switch, 15
switch fabrics, 59
switching, vs subnets, 186
switchline.py, 547
symbol, data, 56, 127
symmetric ciphers, 739
SYN, 331
SYN flooding, 332
SYN packet, 332
synchronization source, RTP, 644
synchronized loss hypothesis, TCP, 415
synchronized loss, TCP, 378, 410, 414
synchronized states, TCP, 344

T

T/TCP, 353
T1 line, 130
T3 line, 131
tables, SNMP, 661
Tahoe, 37, 375
tail drop, 401
tangle, cords, 92
tbf, linux, 560
tc, linux, 259, 560, 615
TCO, TCP, 337
TCP, 11, 28, 327
TCP accelerated open, 353
TCP application close, 347
TCP BBR, 467
TCP checksum offloading, 337
TCP close, 333
TCP Cubic, 463
TCP fairness, 410, 417
TCP Fast Open, 354
TCP Friendliness, 420
TCP Hamilton, 462
TCP header, 330
TCP Hybla, 459
TCP Illinois, 455
TCP minimum RTO, 461
TCP NewReno, 387
TCP NewReno in ns-2, 511

TCP old duplicates, 348
TCP Reno, 375, 384
TCP sawtooth, 376, 380, 430, 481, 507, 518
TCP segmentation offloading, 337
TCP state diagram, 343
TCP Tahoe, 375
TCP timeout interval, 627
TCP Vegas, 516, 595
TCP Westwood, 453
TCP Westwood+, 454
TCP, Highspeed, 445
TCP, SACK, 389
TCP_NODELAY, 356
TCP_QUICKACK, 355
TDM, 129
Teredo tunneling, 176
terrestrial broadband, 116
terrestrial wireless, 116
TestAndIncr, 687
TEXTUAL-CONVENTION, SNMP, 685
TFTP, 311
thepiratebay, 192
thermonuclear, 35
three-way handshake, 332
three-way handshake, TCP, 350
threshold slow start, 380
throughput, 12
tier-1 provider, 289
Time to Live, 174
time-division multiplexing, 129
timeout and retransmission, 28
timeout interval, TCP, 357, 627
Timestamp, IP option, 175
TIMEWAIT, TCP, 349
TJX attack, 88, 718
tls, 754
TLS client example, 769
TLS connection setup, 762
TLS handshake protocol, 762
TLS programming, 765
TLS server example, 768
TLS version 1.3, 765
token bucket, 605
token bucket queue utilization, 611
token bus Ethernet, 82
token ring, 81
token-bucket applications, 610
token-bucket, RSVP, 633
topology, 16
topology table, EIGRP, 256
Tor project, 193
ToS and routing, 245
TP4, 36
trace file, ns-2, 482
tracefiles, ns-2, reading with python, 485
traceroute, 33, 203
tracking, Wi-Fi, 103
trading, 137
traffic amplification, QUIC, 368
traffic amplification, UDP, 300
traffic anomalies, 733
traffic engineering, 16, 259, 285, 623
traffic management, 580
tragedy of the commons, 373
Trango, 118
transient queue peak, 509
transit capacity, 159
transit traffic, 279, 285
Transmission Control Protocol, 28
transmission, Ethernet, 49
Transport layer, 28
transport mode, IPsec, 772
traps, SNMP, 656
tree, 16
triggered updates, 251
triple DES, 742
Trivial File Transport Protocol, 311
trust anchors, 223
trust and public keys, 753
trust on first use, SSH, 756
trust on first use, TLS, 760
TSO, TCP, 337
Tspec, 633
TTL, 174
tunnel mode, IPsec, 772
tunnel, IPv6, 237
tunneling, 79, 190
two implementations, 36
two-generals problem, 315
twos-complement, 144
Type of Service, 174

U

u32, 562
UDP, 29, 299, 307, 309
UDP advisory, 300

UDP, for real-time traffic, 627
unbounded slow start, 380
unicast, 21
unique-local IPv6 address, 216
unknown destinations, Ethernet, 57
unlicensed spectrum, 93
unnumbered IP interface, 205
upgrades, network, 18
uplink scheduling, WiMAX and LTE, 113
URG, 331
User Datagram Protocol, 29
usmUserTable, 709
utilities, network, 32

V

VACM, 681
VarBind list, 666
VCI, 83
video, streaming, 626
virtual circuit, 12, 23, 82
virtual hosting, 195
Virtual LANs, 64
virtual link, 79
virtual private network, 79
virtual tributary, 133
VLANs, 64
voice over IP, 23
VoIP, 23
VoIP and RTP, 645
VoIP bandwidth guarantees, 610
voting, 35
VPN, 79
VPNs and ECN, 80

W

W^X, 727
wavelength-division multiplexing, 134
web of trust, 754
web server, 549
weighted fair queuing, 582
WEP encryption failure, 718
WEP, Wi-Fi, 105
Westwood, TCP, 453
Wi-Fi, 92
Wi-Fi fragmentation, 97
Wi-Fi polling mode, 109
Wi-Fi security, 88
WiMAX, 112
window, 158
window scale option, TCP, 354
window size, 29, 157
Windows, 176
Windows XP SP1 vulnerability, 730
winsize, 157
wireless, 92
wireless LANs, 88
wireless, fixed, 116
wireless, satellite, 118
wireless, terrestrial, 116
WireShark, 234, 365, 533
wireshark, 34
WireShark, TCP example, 336
work-conserving queuing, 582
WPA authenticator, 106
WPA supplicant, 106
WPA, Wi-Fi, 105
WPA-Enterprise, 106
WPA-Personal, 105
WPA2-Enterprise, configuring, 107
write-or-execute, 727

X

XD page bit, 727
xkcd, 275, 732
XML, 311
XSS, 731

Z

ZigBee, 92
zone identifier, IPv6, 236
zones, DNS, 192