An Introduction to Computer Networks

Release 1.0

Peter L Dordal

March 16, 2014

CONTENTS

Peter L Dordal
Department of Computer Science
Loyola University Chicago

Contents:

0 PREFACE

"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 book, 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. In this it is in good company: there is Wikipedia, there is Gnu/Linux, and there is an increasing number of other free online textbooks out there. The market inefficiencies of traditional publishing are sobering: the return to authors of advanced textbooks is at best modest, and costs to users are quite high.

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 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. 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.

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, earlier versions of this material have been used – 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 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 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 a chapter on using the ns-2 simulator to explore actual TCP behavior. 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 does not make an attempt to cover is the application layer. There are simply too many directions there in which to go; my recommendation is that, to the extent that application coverage is desired, the instructor can combine this text with appropriate application documentation.

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 *3.3 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* and *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.

Peter Dordal
Shabbona, Illinois

0.2 Progress Notes

As of March 2014 I am declaring that the first edition is complete. There remains some things I would like to add, but for now they will wait.

Going forward, the intronetworks.cs.luc.edu website will carry both the completed first edition and also the current 1.x edition, x>0. My plan is to make sure, as far as is possible, that the two editions are classroom-compatible: existing exercises will not be renumbered (though new exercises may be introduced with fractional numbers), and section renumbering will be avoided to the extent practical.

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

0.3 Technical considerations

In order to read this book online in html format you will need the appropriate unicode character set installed. At a minimum, the following characters must display properly:

⟨⟩ √∞≃≤≥×–≠ αβγμφ→←−−│┌┐└┘┬┴├┤┼

The diagrams in the body of the text are now all in bitmap .png format, although a few diagrams rendered with line-drawing characters still appear in the exercises. I would prefer to use the vector-graphics .svg format, but as of January 2014 most browsers do not appear to support zooming in on .svg images, which is really the whole point.

The book was prepared in reStructuredText using the linux Sphinx package.

1 AN OVERVIEW OF NETWORKS

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 (though some of the layers are not accessible to ordinary users): 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.

The LAN layer is often conceptually subdivided into the "Physical layer" dealing with, eg, the electrical signaling mechanisms involved, and above that an abstracted "Logical link layer" that describes how packets can be addressed from one LAN node to another. 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 link layer. 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*.

1.2 Bandwidth and Throughput

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.

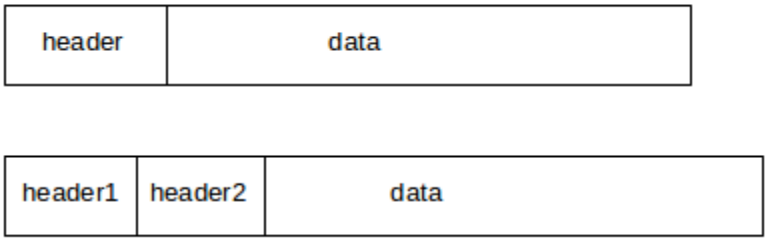
The term **bandwidth** can be used to refer to either of these, though we here try to use it mostly as a synonym for data rate. The term comes from radio transmission, where the width of the frequency band available is proportional, all else being equal, to the data rate that can be achieved.

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); in the context of data rates, a kilobit is 10^3 bits (not 2^{10}) and a megabit is 10^6 bits. The use of the lowercase "b" means bits; data rates expressed in terms of bytes often use an upper-case "B".

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 *18.11.4 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 make sure 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".

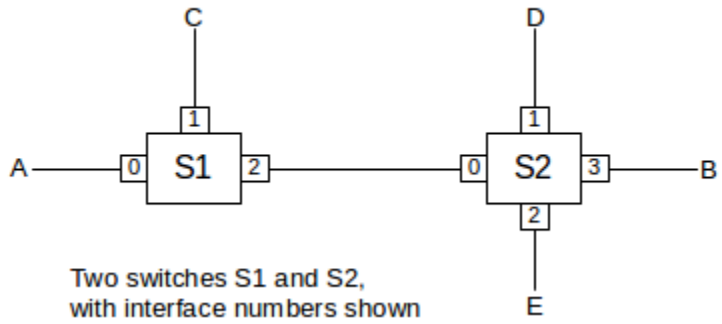
It is perhaps worth noting that packets are buffers built of 8-bit *bytes*, and all hardware today agrees what a byte is (hardware agrees *by convention* on the order in which the bits of a byte are to be transmitted). 8-bit bytes are universal now, but it was not always so. Perhaps the last great non-byte-oriented 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. **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 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 P to B, S1 must know that P must be forwarded out interface 2 and S2 must know P must be forwarded out interface 3. 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 this table, and finds the **next_hop** information: the address to which or interface by which the packet should be forwarded in order to bring it one step closer to its final destination. In the network below, a complete forwarding table for S1 (using interface numbers as next_hop values) 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

Alternatively, we could replace the interface information with next-node, or **neighbor**, information, as all the links above are point-to-point and so each interface connects to a unique neighbor. In that case, S1's table might 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.

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, many ISPs ignore quality-of-service information in the IP header, 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:

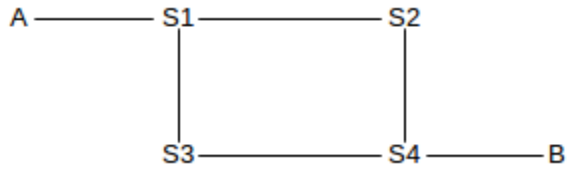
S1	
destination	**next_hop**
A	0
C	1
default	2

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.

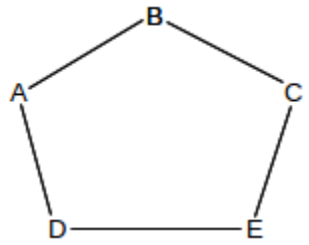
1.6 Routing Loops

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 to agree 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 v 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.

Congestion *is* a sign of a problem in real-time networks, which we will consider in *18 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 *3.7 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 the data transfer rate to a typical faster hard disk is about 1000 Mbps.) Wireless ("wi-fi") LANs are gaining popularity, and in some 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 *2.1.2 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" (*12.9.1 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; Ethernet switches do this by a passive **learning** algorithm. (IP routers, by comparison, use "active" protocols.) 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 forwarding 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 risk of eavesdropping is 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. The network/host division point is *not* carried within the IP header; in fact, nowadays the division into network and host is dynamic, and can be made at different positions in the address at different levels of the network.

IP addresses, unlike Ethernet addresses, are **administratively assigned**. You would get your Class B network portion, say, from the Internet Assigned Numbers Authority, or IANA, 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.

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).

The network portion of an IP address is sometimes called the **network number** or **network address** or **network prefix**; it 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) *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*.

IP routers use datagram forwarding, described in *1.4 Datagram Forwarding* above, to deliver packets, but the "destination" values listed in their 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 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. When network prefixes are used 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".)

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.

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. 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.

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 *3.7 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* connection 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. The primary advantage of connection-oriented networking, however, 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.

Perhaps 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 decide to send traffic through one router, in which case queue overflows are hard to avoid.

Now let us look at a simple example of how IP routing (or forwarding) 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.)

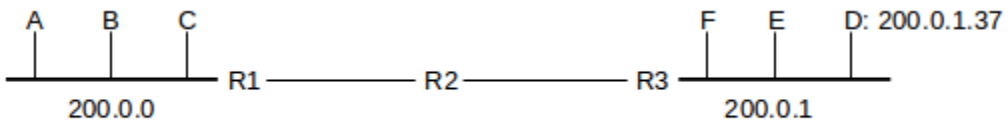
Suppose A is the sending host, sending a packet to a destination host D. The IP header of the packet will contain D's IP address in the "destination address" field (it will also contain A's own address as the "source address"). The first step is for A to determine whether D is on the same LAN as itself or not. 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, *7.7 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, 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 *7.10 Unnumbered Interfaces*.

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.

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 "broadcast" 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 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. 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.

Unrelated organizations exchange information through the Border Gateway Protocol, BGP (*10 Large-Scale IP Routing*), which allows for a fusion of technical information (which sites are reachable at all, and through where) with "policy" information representing legal or commercial agreements: which outside routers are "preferred", whose traffic we will carry even if it isn't to one of our customers, etc.

Internet 2 is a consortium of research sites with very-high-bandwidth internal interconnections. Before Internet 2, Loyola's largest router table consisted of maybe a dozen internal routes, plus one "default" route to the outside Internet. After Internet 2, the default route still pointed to the commercial Internet, but the master forwarding table now had to have an entry for every Internet-2 site so this traffic would take the Internet-2 path. See *7 IP version 4*, exercise 5.

As of the end of January 2011, the IANA has run out of Class-A blocks to allocate 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

There is a table at http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xmlof IANA assignments of /8 blocks; examination of the table shows all /8 blocks have now been allocated. A few months after the IANA 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 appear to have retreated only slightly.

Hopefully, this turn of events will accelerate implementation of IPv6, which has 128-bit addresses.

1.11 DNS

IP addresses are hard to remember (nearly impossible in IPv6). The **domain name system**, or 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; indeed, it is the classic example of a widely distributed database. In looking up `www.cs.luc.edu` three different DNS servers may be queried: for `cs.luc.edu`, for `luc.edu`, and for `.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.

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, so DNS must either return a list, or return one name that has been designated the *canonical* name.

1.12 Transport

Think about what types of communications one might want over the Internet:

- Interactive communications such as via ssh or telnet, with long idle times between short bursts
- Bulk file transfers
- Request/reply operations, *eg* to query a database or to make DNS requests
- Real-time voice traffic, at (without compression) 8KB/sec, with constraints on the *variation* in delivery time (known as *jitter*; see *18.11.3 RTP Control Protocol* for a specific numeric interpretation)

- Real-time video traffic. Even with substantial compression, video generally requires much more bandwidth than voice

While separate protocols might be used for each of these, the Internet has standardized on the Transmission Control Protocol, or **TCP**, for the first three (though there are periodic calls for a new protocol addressing the third item above), and TCP is sometimes pressed into service for the last two. TCP is thus the most common "transport" layer for application data.

The IP layer is not well-suited to transport. IP routing is a "best-effort" mechanism, which means packets can and do get lost sometimes. Data that does arrive can arrive out of order. The sender has to manage division into packets; that is, buffering. 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!

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, and 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 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.
- **stream-orientation**: The application 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; clients *initiate* those connections. 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 number. Each call 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 en route; if the window

size is 10, for example, then at any one time 10 packets are out there (perhaps 5 data packets and 5 returning acknowledgments). As each acknowledgment arrives, the window "slides forward" and the data packet 10 packets ahead is sent. For example, consider the moment when the ten packets 20-29 are in transit. When ACK[20] is received, Data[30] is sent, and so now packets 21-30 are in transit. When ACK[21] is received, Data[31] is sent, so packets 22-31 are in transit.

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 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 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.

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

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** 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 program 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 pretty well; when necessary (*eg* for games) certain ports can be selectively unblocked.

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

1.14 Network Address Translation

A related way of providing firewall protection for ordinary machines is Network Address Translation, or **NAT**. This technique also conserves IP addresses. Instead of assigning each host at a site a "real" IP address, just one address is assigned to a border router. Internal machines get "internal" IP addresses, typically of the form or 192.168.x.y or 10.x.y.z (addresses beginning with 192.168.0.0/16 or with 10.0.0.0/8 cannot be used on the Internet). Inbound connections to the internal machines are banned, as with some firewalls. When an internal machine wants to connect to the outside, the NAT router intercepts the connection, allocates a new port, and forwards the connection on as if it came from that new port and its own IP address. The remote machine responds, and the NAT router remembers the connection and forwards data to the correct internal host, rewriting the address and port fields of the incoming packets. Done properly, NAT improves the security of a site. It also allows multiple machines to share a single IP address, and so is popular with home users who have a single broadband connection but who wish to use multiple computers. A typical NAT router sold for residential or small-office use is commonly simply called a "router", or (somewhat more precisely) a "residential gateway".

Suppose the NAT router is NR, and internal hosts A and B each connect from port 3000 to port 80 on external hosts C and D, respectively. Here is what NR's NAT table might look like. No column for NR's IP

address is given, as it is fixed.

remote host	remote port	outside source port	inside host	inside port
C	80	3000	A	3000
D	80	3000	B	3000

A packet to C from ⟨A,3000⟩ would be rewritten by NR so that the source was ⟨NR,3000⟩. A packet from ⟨C,80⟩ addressed to ⟨NR,3000⟩ would be rewritten and forwarded to ⟨A,3000⟩. Similarly, a packet from ⟨D,80⟩ addressed to ⟨NR,3000⟩ would be rewritten and forwarded to ⟨B,3000⟩; the NAT table takes into account the sending socket address as well as the destination.

Now suppose B opens a connection to ⟨C,80⟩, also from inside port 3000. This time NR must remap the port number, because that is the only way to distinguish between packets from ⟨C,80⟩ to A and to B. The new table is

remote host	remote port	outside source port	inside host	inside port
C	80	3000	A	3000
D	80	3000	B	3000
C	80	3001	B	3000

Typically NR would not create TCP connections between itself and ⟨C,80⟩ and ⟨D,80⟩; the NAT table does forwarding but the endpoints of the connection are still at the inside hosts. However, NR might very well *monitor* the TCP connections to know when they have closed, and so can be removed from the table.

It is common for Voice-over-IP (VoIP) telephony using the SIP protocol (**RFC 3261**) to prefer to use UDP port 5060 at *both* ends. If a VoIP server is outside the NAT router (which must be the case as the server must generally be publicly visible) and a telephone is inside, likely port 5060 will pass through without remapping, though the telephone will have to initiate the connection. But if there are two phones inside, one of them will appear to be connecting to the server *from* an alternative port.

VoIP systems run into a much more serious problem with NAT, however. A call ultimately between two phones is typically first negotiated between the phones' respective VoIP servers. Once the call 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 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.

If a site wants to make it possible to allow 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 Network*. 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.

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, where your computer and a friend's, each behind a different NAT router, wish to establish a connection. 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 (*11.3 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⟩.

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.

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. However, most of the features at this level have ended up generally being implemented in the Application layer of the IETF model with no inconvenience. Indeed, there are few examples in the TCP/IP world of data format, compression or encryption being handled as a "sublayer" of the application, in which the sublayer is responsible for connections to the Transport layer (SSL/TLS might be an example for encryption; applications read and write data directly to the SSL endpoint). Instead, applications usually read and write data directly from/to the TCP layer, and then invoke libraries to handle things like data conversion, compression and encryption.

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 grap

IP ⟵ UDP ⟵ RTP
IP ⟵ TCP

(with the arrows pointing *downwards* in the graph) and

Ethernet ⟵ IP
Ethernet ⟵ ARP ⟵ IP
ATM ⟵ IP

1.16 Berkeley Unix

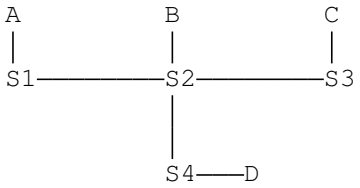
Though not officially tied to the IETF, the Berkeley Unix releases became *de facto* reference implementations for most of the TCP/IP protocols. 4.1BSD (BSD for Berkeley Software Distribution) was released in 1981, 4.2BSD in 1983, 4.3BSD in 1986, 4.3BSD-Tahoe in 1988, 4.3BSD-Reno in 1990, and 4.4BSD 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

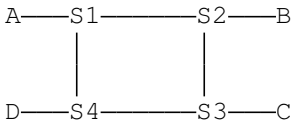
This completes our tour of the basics. In the remaining chapters we will expand on the material here.

1.18 Exercises

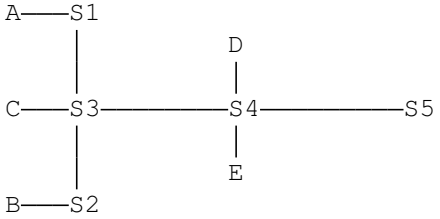
1. Give forwarding tables for each of the switches S1-S4 in the following network with destinations A, B, C, D. For the next_hop column, give the *neighbor* on the appropriate link rather than the interface number.



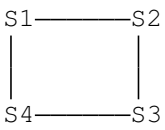
2. Give forwarding tables for each of the switches S1-S4 in the following network with destinations A, B, C, D. Again, use the neighbor form of next_hop rather than the interface form. Try to keep the route to each destination as short as possible. What decision has to be made in this exercise that did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



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

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

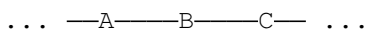
From the above we can conclude that S2 must be directly connected to both S1 and S3 as its table lists them as next_hops; similarly, S3 must be directly connected to S2 and S4.

(a). Must S1 and S4 be directly connected? If so, explain; if not, give a network in which there is no direct link between them, consistent with the tables above.
(b). Now suppose S3's table is changed to the following. In this case must S1 and S4 be directly connected? Why or why not?

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.

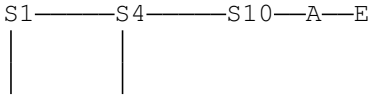
6. (a) Suppose a network is as follows, with the only path from A to C passing through B:

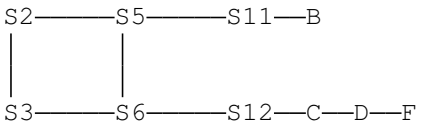


Explain why a single routing loop cannot include both A and C.

(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. Consider the following arrangement of switches:





Suppose S1-S6 have the forwarding tables below. For each destination A,B,C,D,E,F, suppose a packet is sent to the destination *from S1*. Give the switches it passes 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)

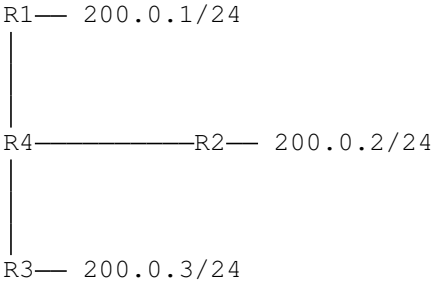
8. In the previous exercise, the routes taken by packets A-D are reasonably direct, but the routes for E and F are rather circuitous.

Some routing applications assign *weights* to different links, and attempt to choose a path with the lowest total link weight.

(a). Assign weights to the seven links S1–S2, S2–S3, S1–S4, S2–S5, S3–S6, S4–S5 and S5–S6 so that packet E's route in the previous exercise becomes the optimum (lowest total link weight) path.
(b). Assign (different!) weights to the seven links that make packet F's route in the previous exercise optimal.

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. Suppose we have the following three Class C IP networks, joined by routers R1–R4. 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**.



2 ETHERNET

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 Ethernet standard in server rooms and backbones, but because Ethernet speed scales 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.

2.1 10-Mbps classic Ethernet

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. In its original form, an Ethernet was a **broadcast bus**, which meant that all packets were, at least at the physical level, broadcast onto the shared medium and could be seen, theoretically, by all other nodes. If two nodes transmitted at the same time, there was a **collision**; proper handling of collisions was an important part of the access-mediation strategy for the shared medium. Data was transmitted using Manchester encoding; see *4.1.3 Manchester*.

The linear bus structure could be modified with **repeaters** (below), into an arbitrary tree structure, though loops remain something of a problem even with today's Ethernet.

Whenever two stations transmitted at the same time, the signals would collide, and interfere with one another; both transmissions would fail as a result. 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 happens, then 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.

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. 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.

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 likely consulted to determine to where the packet is forwarded. Originally, switches were seen as providing interconnection ("bridging") between separate Ethernets, but later a switched Ethernet was seen as one large "virtual" Ethernet. We return to switching below in *2.4 Ethernet 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.

In coaxial-cable installations, one long run of coax snaked around the computer room or suite of offices; each computer connected somewhere along the cable. Thin coax allowed the use of T-connectors to attach hosts; connections were made to thick coax via **taps**, often literally drilled into the coax central conductor. In a standalone installation one run of coax might be the entire Ethernet; otherwise, somewhere a repeater would be attached to allow connection to somewhere else.

Twisted-pair does not allow mid-cable attachment; it is only used for point-to-point links between hosts, switches and hubs. In a twisted-pair installation, each cable runs between the computer location and a central wiring closest (generally much more convenient than trying to snake coax all around the building). Originally each cable in the wiring closet plugged into a hub; nowadays the hub has likely been replaced by a switch.

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.

Here is the format of a typical Ethernet packet (DIX specification):

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.

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.

2.1.1 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.

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.

The second-to-lowest-order bit of the Ethernet address indicates, in the case of physical addresses, whether the address is believed to be globally unique or if it is only locally unique; this is known as the **Universal/Local** bit. When (global) Ethernet IDs are assigned by the manufacturer, the first three bytes serve to indicate the manufacturer. 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.

2.1.2 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.

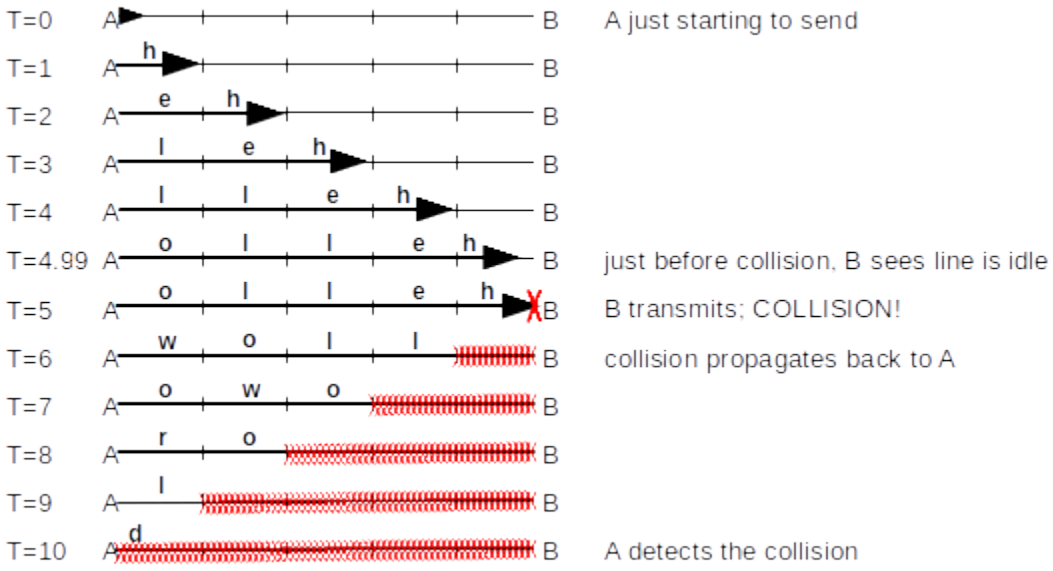
The value of the slot time determines several subsequent aspects of Ethernet. If a station has transmitted for one slot time, then no collision can occur (unless there is a hardware error) for the remainder of that packet. This is because one slot time is enough time for any other station to have realized 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 $c=3\times10^8$ 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 $0.77\times c\times10^{-7}$ = 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.

2.1.3 Exponential Backoff Algorithm

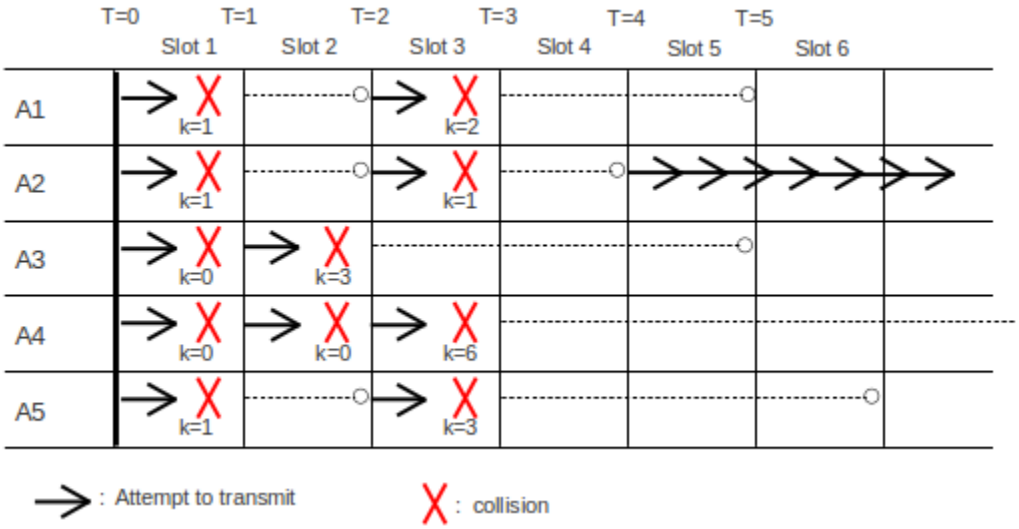
Whenever there is a collision the exponential backoff algorithm is used to determine when each station will retry its transmission. Backoff here is called *exponential* because the range from which the backoff value is chosen is doubled after every successive collision involving the same packet. Here 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, $1\le N\le10$ (N=0 represents the original attempt), choose k randomly with $0 \le k < 2^N$. Wait k slot times ($k\times51.2$ μsec). Then check if the line is idle, waiting if necessary for someone else to finish, and then retry step 3. For $11\le N\le15$, choose k randomly with $0 \le k < 1024$ ($= 2^{10}$)
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≥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 $0 \leq k < 1024$. 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.

2.1.4 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 full-duplex mode is not required for using Fast Ethernet, it was assumed that any sites concerned enough about performance to be worried about the capture effect would opt for full-duplex.

2.1.5 Hubs and topology

Ethernet hubs (multiport repeaters) change the topology, but not the fundamental constraints. Hubs allow much more branching; typically, each station in the office now has its own link to the wiring closet. Loops are still forbidden. Before inexpensive switches were widely available, 10BASE-T (twisted pair Ethernet) used hubs heavily; with twisted pair, a device can only connect to the endpoint of the wire. Thus, typically, each host is connected directly to a hub. The maximum diameter of an Ethernet consisting of multiple segments, joined by hubs, is constrained by the round-trip-time, and the need to detect collisions before the sender has completed sending, as before. However, twisted-pair links are required to be much shorter, about 100 meters.

2.1.6 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.

2.1.7 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 $p \leq 1$. 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 *3.3 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.

2.1.8 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, even if the network is idle. 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.

2.1.8.1 The ALOHA models

We get very similar throughput values when we analyze the Ethernet contention interval using the ALOHA model that was a precursor to Ethernet, and assume a very large number of active senders, each transmitting at a very low rate.

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). 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

$$S = Ns*(1–s)^{2(N–1)}$$
$$= G(1–G/N)^{2N}$$
$$\longrightarrow Ge^{-2G} \text{ as } N\longrightarrow\infty.$$

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.

2.1.8.2 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 speed up at all, 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, and "longer" networks could be formed by chaining together shorter ones with switches. 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 can 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 **collision-free**.

Fast Ethernet uses 4B/5B encoding, covered in *4.1.4 4B/5B*.

Fast Ethernet 100BASE-TX does not particularly support links between buildings, due to the network-diameter limitation. However, fiber-optic point-to-point links are quite effective 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

If we continue to maintain the same slot time but raise the transmission rate to 1000 Mbps, the network diameter would now be 20-40 meters. Instead of that, Gigabit Ethernet moved to a 4096-bit (512-byte) 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.

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.

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 100BASE-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** (*4.1.4 4B/5B*) 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 cable often dwarf the costs of the electronics.

2.4 Ethernet Switches

Switches join separate physical Ethernets (or Ethernets and token rings). 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.

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, below, which automatically "prunes" the network topology to make it loop-free.

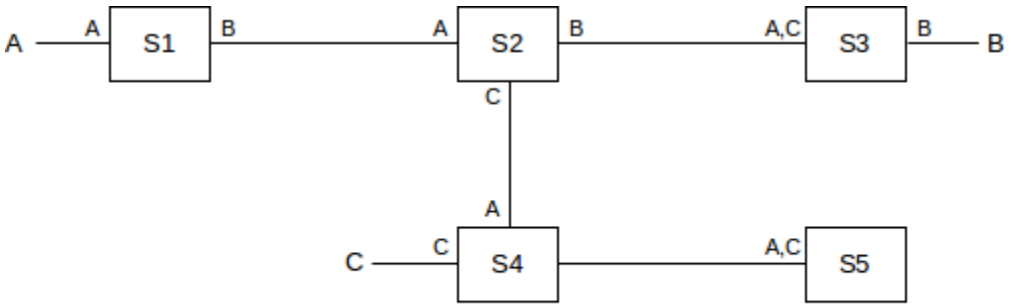
And while a switch does not propagate collisions, it must maintain a queue for each outbound interface in case it needs to forward a packet at a moment when the interface is busy; on occasion packets are lost when this queue overflows.

Ethernet switches use datagram forwarding as described in *1.4 Datagram Forwarding*. They start out with empty forwarding tables, and build them through a "learning" process. If a switch does not have an entry for a particular destination, it will fall back on broadcast: it will forward the packet out every interface other than the one on which the packet arrived.

A switch **learns** address locations as follows: for each interface, the switch maintains a table of physical 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 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 forward the packet out all interfaces J with J$\neq$I; this represents the fallback to broadcast. As time goes on, this fallback to broadcast is needed less and less often.

If the destination address D is the broadcast address, or, for many switches, a multicast address, broadcast is required.



Five learning bridges after three packet transmissions

In the diagram above, each switch's tables are indicated by listing near each interface the destinations 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.

Once all the switches have learned where all (or most of) the hosts are, packet routing becomes optimal. 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 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 only sees the traffic actually addressed to it; other traffic remains inaccessible.

Typical switches have room for table with 10^4 - 10^6 entries, though maxing out at 10^5 entries may be more common; this is usually enough to learn about all hosts in even a relatively large organization. A switched Ethernet can fail when total traffic becomes excessive, but excessive total traffic would drown *any* network (although other network mechanisms might support higher bandwidth). The main 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 IP Subnets*).

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

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

In practice, however, loops allow a form of 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 as a **spanning tree**. The switch-connections graph is the graph with nodes consisting of both switches and of the unswitched Ethernet **segments** and isolated individual hosts connected to the switches. Multi-host Ethernet segments are most often created via Ethernet hubs (repeaters). Edges in the graph represent switch-segment and switch-switch connections; each edge attaches to its switch via a particular, numbered interface. The goal is to disable redundant (cyclical) paths while remaining able to deliver to any segment. The algorithm is due to Radia Perlman, [RP85].

Once the spanning tree is built, all packets are sent only via edges in the tree, which, as a tree, has no loops. Switch ports (that is, edges) that are not part of the tree are not used at all, even if they would represent the most efficient path for that particular destination. 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 to that root

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

- a switch with a lower ID (thus becoming the new root)
- a shorter path to the existing root
- an equal-length path to the existing root, but via a switch or port with a lower ID (the tie-breaker rule)

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 is complete, 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:

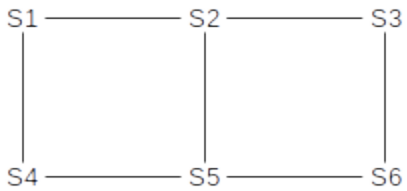
1. It enables the port via which it reaches the root
2. It enables any of its ports that further-out switches use to reach the root
3. If a remaining port connects to a segment to which other "segment-neighbor" switches connect as well, the port is enabled if the switch has the minimum cost to the root 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.



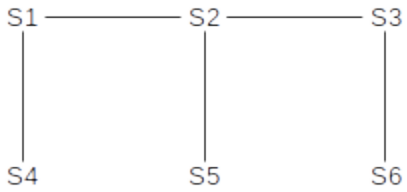
S1 has the lowest ID, and so becomes the root. S2 and S4 are directly connected, so they will enable the interfaces by which they reach S1 (Rule 1) while S1 will enable its interfaces by which S2 and S4 reach it (Rule 2).

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.

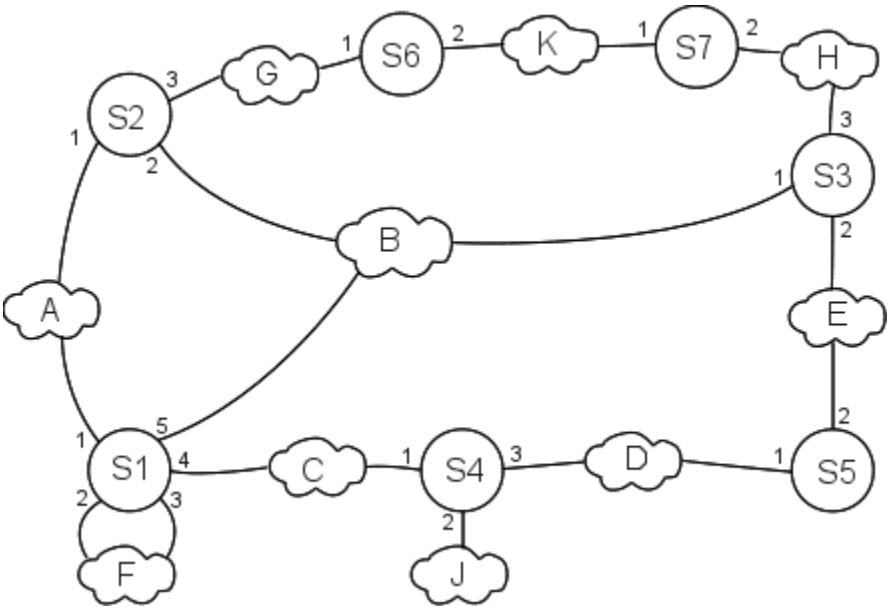
Similarly, S6 has two choices; it selects S3.

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.

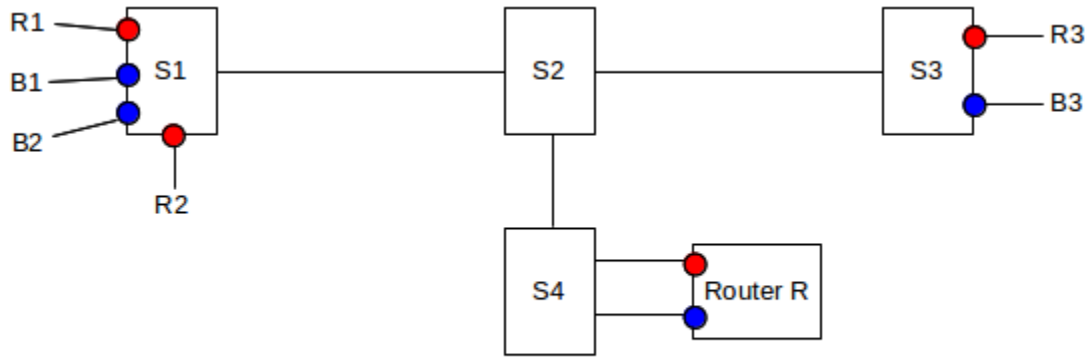
Another issue with the spanning-tree algorithm is that a rogue switch can announce an ID of 0, thus likely becoming the new root; this leaves that switch well-positioned to eavesdrop on a considerable fraction of the traffic. One of the goals of the Cisco "Root Guard" feature is to prevent this; another 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.

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 IP 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 Epilog

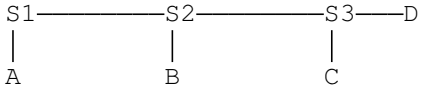
Ethernet dominates the LAN layer, but is not one single LAN protocol: it comes in a variety of speeds and flavors. Higher-speed Ethernet seems to be moving towards fragmenting into a range of physical-layer options for different types of cable, but all based on switches and point-to-point linking; different 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.

2.8 Exercises

1. Simulate the contention period of five Ethernet stations that all attempt to transmit at T=0 (presumably when some sixth station has finished transmitting). 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. Suppose we have Ethernet switches S1 through S3 arranged as below. All forwarding tables are initially empty.



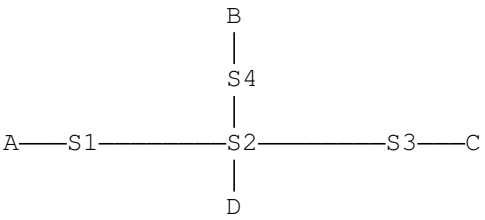
(a). If A sends to B, which switches see this packet?

(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?

3. 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*.

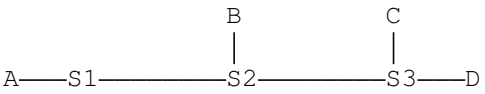


Now suppose the following packet transmissions take place:

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

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

4. In the switched-Ethernet network below, find two packet transmissions so that, when a third transmission A⟶D occurs, the packet *is* delivered to B (that is, it is forwarded out all ports of S2), but is *not* similarly delivered to C. 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.

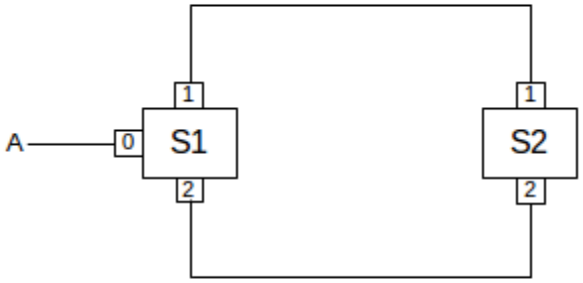
5. Given the Ethernet network with learning switches below, with (disjoint) unspecified parts represented by ?, explain why it is impossible for a packet sent from A to B to be forwarded by S1 only to S2, but to be forwarded by S2 out all of S2's other ports.

6. In the diagram of *2.4 Ethernet Switches*, suppose node D is connected to S5, and, with the tables as shown below the diagram, D sends to B.

(a). Which switches will see this packet, and thus learn about D?

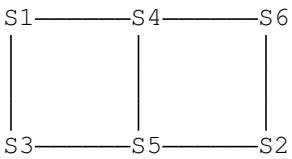
(b). Which of the switches in part (a) do *not* already know where B is (and will thus forward the packet out all non-arrival interfaces)?

7. 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 forward the packet out interfaces 1 and 2. What happens then? How long will A's packet circulate?

8. The following network is like that of *2.5.1 Example 1: Switches Only*, except that the switches are numbered differently. What is the end result of the spanning-tree algorithm in this case?



9. 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. 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 happens if a port formerly reserved for connection to another switch is now used for a host?

3 OTHER LANS

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. A return to being tethered by wires is almost unthinkable.

3.1 Virtual Private Network

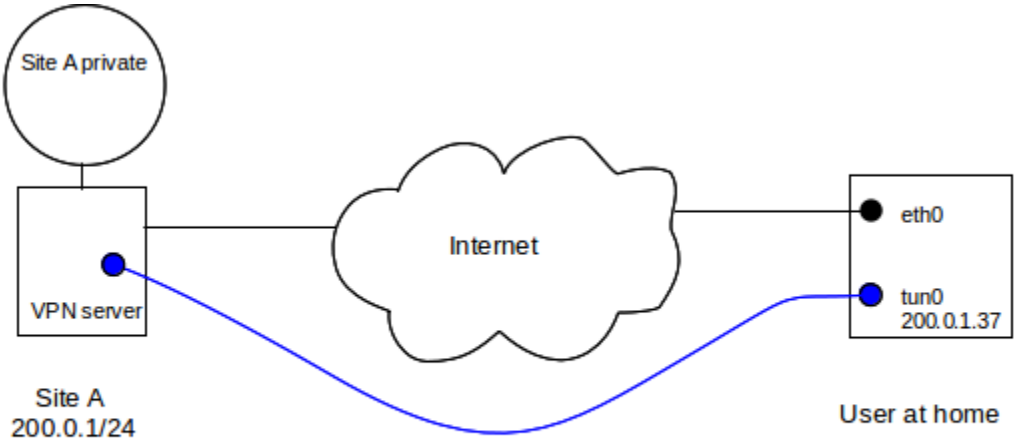
Suppose you want to connect to your workplace network from home. Your workplace, however, has a security policy that does not allow "outside" IP addresses to access essential internal resources. How do you proceed, without leasing a dedicated telecommunications line to your workplace?

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; see *7.11 Mobile IP*). Each end of the connection is associated with a software-created **virtual network interface**; each of the two virtual interfaces is assigned an IP address. 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 usually encrypted as well as encapsulated, though that is a separate issue. One example of a tunneling protocol 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**.

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.



VPN: blue link represents *tunnel*. Actual connection is made via eth0
The tun0 interface is a virtual network interface with a Site-A address

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.

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.

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 layer 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 Wi-Fi

Wi-Fi is a trademark 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. 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. Wi-Fi is designed to interoperate freely with Ethernet at the logical LAN layer; that is, Ethernet and Wi-Fi traffic can be freely switched from the wired side to the wireless side.

Band Width

To radio engineers, "band width" means the frequency range used by a signal, not data rate; in keeping with this we will in this section and *3.4 WiMAX* use the term "data rate" instead of "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.

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. 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 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 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 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 unrelated Wi-Fi transmitters can and do interact with and interfere with each other.

The United States requires users of the 5 GHz band to avoid interfering with weather and military applications in the same frequency range. Once that is implemented, however, there are more 5 MHz channels at this frequency than in the 2.4 GHz ISM band, which is one of the reasons 802.11ac can run faster (below).

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

- improved radio modulation techniques
- improved error-correcting codes
- smaller guard intervals between symbols
- increasing the channel width
- allowing multiple spatial streams via multiple antennas

The first two in this list seem by now to be largely tapped out; the third reduces the range but may increase the data rate by 11%.

The largest speed increases are obtained by increasing the number of 5 MHz channels used. 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 (150 Mbps with a smaller guard interval). 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).

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 $(4/3)\times 54 = 72$ Mbps, sent in symbols consisting of six bits as a unit.

3.3.1 Multiple Spatial Streams

The latest innovation in improving Wi-Fi data rates is to support multiple spatial 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. While the antennas are each more-or-less omnidirectional, careful signal analysis at the receiver can in principle recover each data stream. In practice, overall data-rate improvement over a single antenna can be considerably less than N.

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 $C \leq \min(A,B)$ is the number of spatial streams.

3.3.2 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. While there *is* a largely-collision-free mode for Wi-Fi operation (*3.3.7 Wi-Fi Polling Mode*), it is not commonly used, and collision management has a significant impact on ordinary Wi-Fi performance.

Wi-Fi, like almost any other wireless protocol, cannot detect collisions in progress. 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 1/100,000 of the transmitted signal, and it is simply overwhelmed during transmission. Thus, while the Ethernet algorithm was known as CSMA/CD, where CD stood for Collision Detection, Wi-Fi uses CSMA/**CA**, where CA stands for Collision **Avoidance**.

The basic collision-avoidance algorithm is as follows. There are three parameters applicable here (values are for 802.11b/g in the 2.4 GHz band); the value we call IFS is more formally known as DIFS (D for "distributed"; see *3.3.7 Wi-Fi Polling Mode*).

- slot time: 20 µsec
- IFS, the "normal" InterFrame Spacing: 50 µsec
- SIFS, the *short* IFS: 10 µsec

A 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 waits for the other traffic to finish and then for one IFS time after. If the station had to wait for other traffic, even briefly, it must then 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. Under this rule, the backoff-wait clock can expire only when the clock is running and therefore the medium is idle, at which point the transmission begins.

If there is a collision, the station retries, 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 in *3.3.2.2 Hidden-Node Problem*.

Because Wi-Fi cannot detect collisions directly, the protocol adds **link-layer ACK packets** (unrelated to the later TCP ACK), at least for unicast transmission. Once a station has received a packet addressed to it,

it waits for time SIFS and sends the ACK; at the instant when the end of the SIFS interval is reached, the receiver will be the only station authorized to send. Any other stations waiting the longer IFS period will see the ACK before the IFS time has elapsed and will thus not interfere with the ACK; similarly, any stations with a running backoff-wait clock will continue to have that clock suspended.

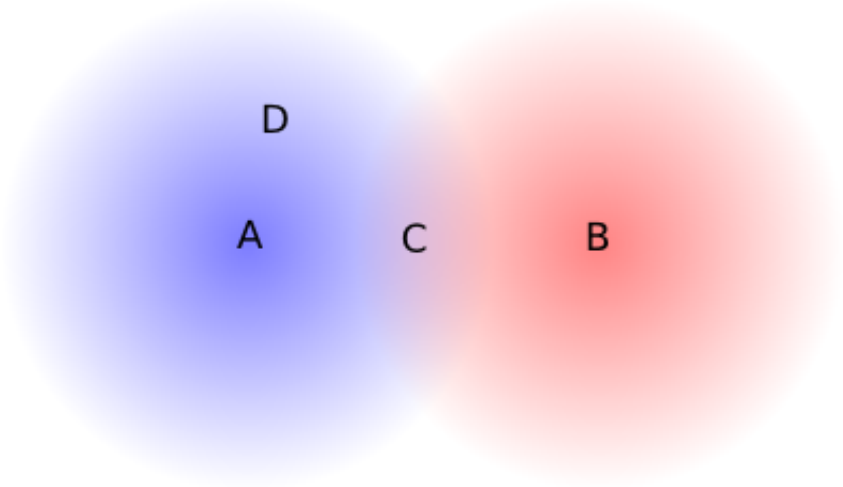
3.3.2.1 Wi-Fi RTS/CTS

Wi-Fi stations optionally also use a request-to-send/clear-to-send (**RTS/CTS**) protocol. Usually this is used only for larger 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) to be the maximum packet size, effectively disabling this feature. The idea here 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 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.

3.3.2.2 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.

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.

3.3.2.3 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.

3.3.3 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 $36 \times 75\% = 27$ 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 (described below), 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. A variety of algorithms have been proposed; a typical strategy involves maintaining a running average of the transmission success rate for each bit-rate in recent use. See [JB05] for a summary. While the signal-to-noise ratio has a strong influence on the transmission success rate, the exact correlation is sometimes obscure, and drivers do not use the signal-to-noise ratio directly.

While lowering the bit rate likely does increase transmission reliability in the face of noise, the longer packet-transmission times caused by the lower bit rate may *increase* the frequency of hidden-node collisions.

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.

3.3.4 Wi-Fi Configurations

There are two common Wi-Fi configurations: **infrastructure** and **ad hoc**. 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 *3.3.8 MANETs*.

The more-common **infrastructure** configuration involves designated **access points** to which individual Wi-Fi stations must **associate** before general communication can begin. The association process – carried out by an exchange of special management packets – may be restricted to stations with hardware ("MAC") addresses on a predetermined list, or to stations with valid cryptographic credentials. Stations may regularly re-associate to their Access Point, especially if they wish to communicate some status update.

Access Points

Generally, a Wi-Fi access point has special features; Wi-Fi-enabled "station" devices like phones and workstations do not act as access points. However, it may be 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. Under linux, the `hostapd` package is one option.

Stations in an infrastructure network communicate directly *only* with their access point. If B and C share access point A, and B wishes to send a packet to C, then B first forwards the packet to A and A 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. In an ad hoc network, by comparison, it is quite common for two nodes to be able to reach each other only by forwarding through an intermediate third node; this is in fact exactly the hidden-node scenario.

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.

3.3.5 Wi-Fi Roaming

Wi-Fi access points are generally identified by their **SSID**, an administratively defined string such as "linksys" or "loyola". These are periodically broadcast by the access point in special **beacon** packets (though for pseudo-security reasons beacon packets can be suppressed). Large installations can create "roaming" access among multiple access points 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, 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.

In order for this to work, traffic *to* wireless node B must find B's current access point AP. This is done in much 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. The **distribution network** is the underlying wired network (*eg* Ethernet) to which all the access points connect. If the distribution network is a switched Ethernet supporting the usual learning mechanism (*2.4 Ethernet Switches*), then Wi-Fi location update is straightforward. Suppose B

is a wireless node that has been exchanging packets via the distribution network with 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 involved will then learn the route through the switched Ethernet from C to B's current AP, and thus to B.

This process may leave other switches – not currently communicating with B – still holding in their forwarding tables the old location for B. This is not terribly serious, but can be avoided entirely if, after moving, B sends out an Ethernet broadcast packet.

Ad hoc networks also have SSIDs; these are generated pseudorandomly at startup. Ad hoc networks have beacon packets as well; all nodes participate in the regular transmission of these via a distributed algorithm.

3.3.6 Wi-Fi Security

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 Wired-Equivalent Privacy, or WEP, encryption standard, 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.

WEP has been replaced with Wi-Fi Protected Access, or **WPA**; the current version is WPA2. WPA2 encryption is believed to be quite secure, although there was a vulnerability in the associated Wi-Fi Protected Setup protocol.

Key management, however, can be a problem. Many smaller sites use "pre-shared key" mode, known as **WPA-Personal**, in which a single key is entered into the Access Point (ideally not over the air) and into each connecting station. The key is usually a hash of a somewhat longer passphrase. The use of a common key for multiple stations makes changing the key, or revoking the key for a particular user, difficult.

A more secure approach is **WPA-Enterprise**; this allows each station to have a separate key, and also for each station key to be recognized by all access points. It is part of the IEEE 802.1X security framework (technically so is WPA-Personal, though a much smaller part). In this model, the client node is known as the **supplicant**, the Access Point is known as the **authenticator**, and there is a third system involved known as the **authentication server**. One authentication server can support multiple access points.

To begin the association process, the supplicant contacts the authenticator using the Extensible Authentication Protocol, or **EAP**, with what amounts to a request to associate 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 an authentication server, which may exchange (via the authenticator) several challenge/response messages with the supplicant. 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. WPA-Enterprise is sometimes known as 802.1X mode, EAP mode or RADIUS mode.

One peculiarity of EAP is that EAP communication takes place before the supplicant is given an IP address (in fact before the supplicant has completed associating itself to the access point); 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.

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, or some form of security certificate, or often 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.

When this authentication strategy is used, 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. The access point allows the supplicant to associate into the network once it receives permission to do so from the authentication server.

3.3.7 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 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.

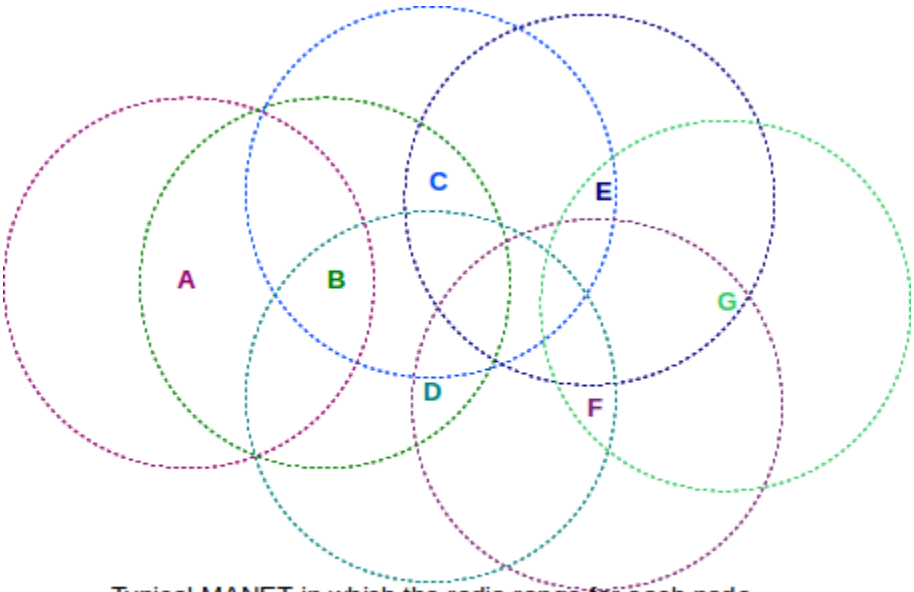
The Access Point generally maintains a **polling list** of stations that wish to be polled during the CFP. Stations request inclusion on this list by an indication when they associate or (more likely) reassociate to the Access Point. A polled station with nothing to send simply remains quiet.

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.

3.3.8 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, 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, unless the forwarding of broadcast messages throughout the MANET is built in to the routing mechanism. This can complicate the assignment of IP addresses; the common IPv4 mechanism we will describe in *7.8 Dynamic Host Configuration Protocol (DHCP)* relies on broadcast and so usually needs some adaptation.

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.

3.3.8.1 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 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 Network*.

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

3.4 WiMAX

WiMAX is a wireless network technology standardized by IEEE 802.16. It supports both stationary subscribers (802.16d) and mobile subscribers (802.16e). The stationary-subscriber version is often used to provide residential Internet connectivity, in both urban and rural areas. The mobile version is sometimes referred to as a "fourth generation" or 4G networking technology; its similar primary competitor is known as LTE. WiMAX is used in many mobile devices, from smartphones to traditional laptops with wireless cards installed.

As in the sidebar at the start of *3.3 Wi-Fi* 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. WiMAX also supports a number of options for the width of its frequency band; the wider the band, the higher the data rate. Wider bands also allow the opportunity for multiple independent frequency channels. Downlink (base station to subscriber) data rates can be well over 100 Mbps (uplink rates are usually smaller).

Like Wi-Fi, WiMAX **subscriber stations** connect to a central access point, though the WiMAX standard prefers the term **base station** which we will use henceforth. Stationary-subscriber WiMAX, however, operates on a much larger scale. The coverage radius of a WiMAX base station can be tens of kilometers if larger antennas are provided, versus less (sometimes much less) than 100 meters for Wi-Fi; mobile-subscriber WiMAX might have a radius of one or two kilometers. Large-radius base stations are typically mounted in towers. Subscriber stations are not generally expected to be able to hear other stations; they interact only with the base station. As WiMAX distances increase, the data rate is reduced.

As with Wi-Fi, the central "contention" problem is how to schedule transmissions of subscriber stations so they do not overlap; that is, collide. The base station has no difficulty broadcasting transmissions to multiple different stations sequentially; it is the transmissions of those stations that must be coordinated. Once a station completes the **network entry** process to connect to a base station (below), it is assigned regular (though not necessarily periodic) transmission slots. These transmission slots may vary in size over time; the base station may regularly issue new transmission schedules.

The centralized assignment of transmission intervals superficially resembles Wi-Fi PCF mode (*3.3.7 Wi-Fi Polling Mode*); however, assignment is not done through polling, as propagation delays are too large (below). Instead, each WiMAX subscriber station is told in effect that it may transmit starting at an assigned time T and for an assigned length L. The station synchronizes its clock with that of the base station as part of the network entry process.

Because of the long distances involved, synchronization and transmission protocols must take account of speed-of-light delays. The round-trip delay across 30 km is 200 μsec which is ten times larger than the basic Wi-Fi SIFS interval; at 160 Mbps, this is the time needed to send 4 KB. If a 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; a special **ranging** mechanism allows stations to figure out this delay.

A subscriber station begins the network-entry connection process to a base station by listening for the base station's transmissions (which may be organized into multiple channels); these message streams contain regular management messages containing, among other things, information about available data rates in each direction.

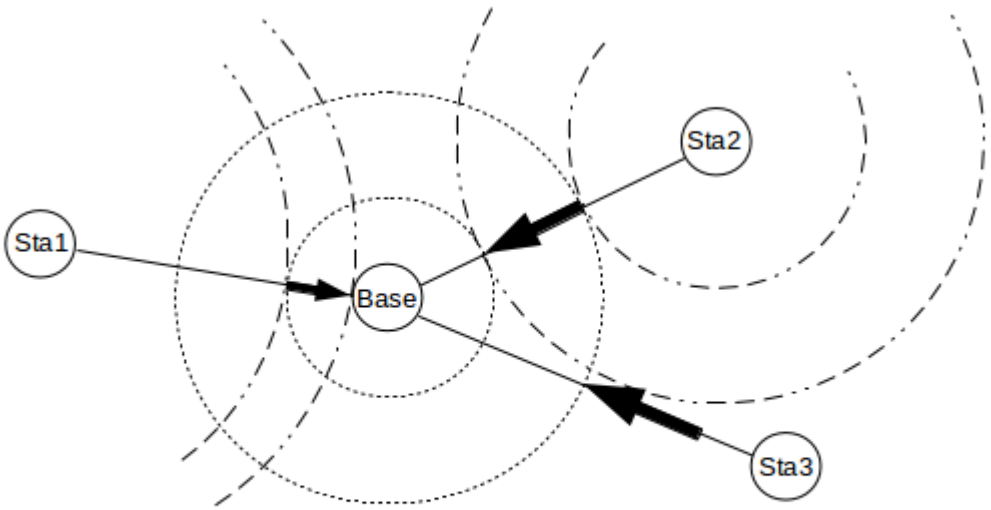
Also included in the base station's message stream is information about start times for **ranging intervals**.

The 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. However, network entry is only done once (for a given base station) and so the likelihood of a collision in any one ranging interval is small. An Ethernet/Wi-Fi-like exponential-backoff process is used if a collision does occur. Ranging intervals are the only times when collisions can occur; afterwards, all station transmissions are scheduled by the base station.

If there is no collision, the base station responds, and the station now knows the propagation delay and thus can determine when to transmit so that its data arrives at the base station exactly at a specified time. The station also determines its transmission signal strength from this ranging process.

Finally, and perhaps most importantly, the station receives from the base station its first timeslot for a **scheduled** transmission. These timeslot assignments are included in regular **uplink-map** packets broadcast by the base station. Each station's timeslot includes both a start time and a total length; lengths are in the range of 2 to 20 ms. Future timeslots will be allocated as necessary by the base station, in future uplink-map packets. Scheduled timeslots may be periodic (as is would be appropriate for voice) or may occur at varying intervals. WiMAX stations may request any of several quality-of-service levels and the base station may take these requests into account when determining the schedule. The base station also creates a downlink schedule, but this does not need to be communicated to the subscriber stations; the base station simply uses it to decide what to broadcast when to the stations. When scheduling the timeslots, the base station may also take into account availability of multiple transmission channels and of directional antennas.

Through the uplink-map schedules and individual ranging, each station transmits so that one transmission finishes arriving just before the next transmission begins arriving, as seen from the perspective of the base station. Only minimal "guard intervals" need be included between consecutive transmissions. Two (or more) consecutive 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 following diagram illustrates this for stations separated by relatively large physical distances.



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.

Mobile stations will need to update their ranging information regularly, but this can be done through future scheduled transmissions. The distance to the base station is used not only for the mobile station's transmission timing, but also to determine its power level; signals from each mobile station, no matter where located, should arrive at the base station with about the same power.

When a station has data to send, it includes in its next scheduled transmission a request for a longer transmission interval; if the request is granted, the station may send the data (or at least some of the data) in its *next* scheduled transmission slot. When a station is done transmitting, its timeslot shrinks 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.

3.5 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**.

3.5.1 Terrestrial Wireless

For non-satellite systems, access points are usually tower-mounted and serve multiple subscribers, though point-to-point "microwave links" are also available. A multi-subscriber access point may serve an area with radius up to a hundred miles, depending on the technology. WiMAX 802.16d is one form of fixed wireless, but there are several others. Generally frequencies are 900 MHz and up, meaning that line-of-sight transmission is used. Frequencies may be either licensed or unlicensed. Some frequencies are better than others at "seeing" through trees and other obstructions.

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 40 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".

3.5.2 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. Transmitter power is typically 1-2 watts, remarkably low for a signal that travels 35,786 km.

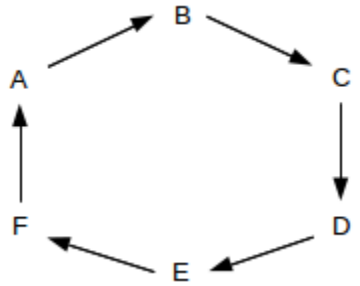
The primary problem associated with satellite Internet is very long RTTs. The the speed-of-light round-trip propagation delay is about 500 ms to which must be added queuing delays for the often-backlogged access point (my own personal experience suggested that RTTs of close to 1,000 ms were the norm). 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.

3.6 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.

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, a premium price was charged (there was also the matter of licensing fees). But due to a combination of lower hardware costs and higher bitrates (even taking collisions into account), Ethernet 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.

3.7 Virtual Circuits

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

Virtual circuits are The Road Not Taken by IP.

Virtual-circuit switching 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.

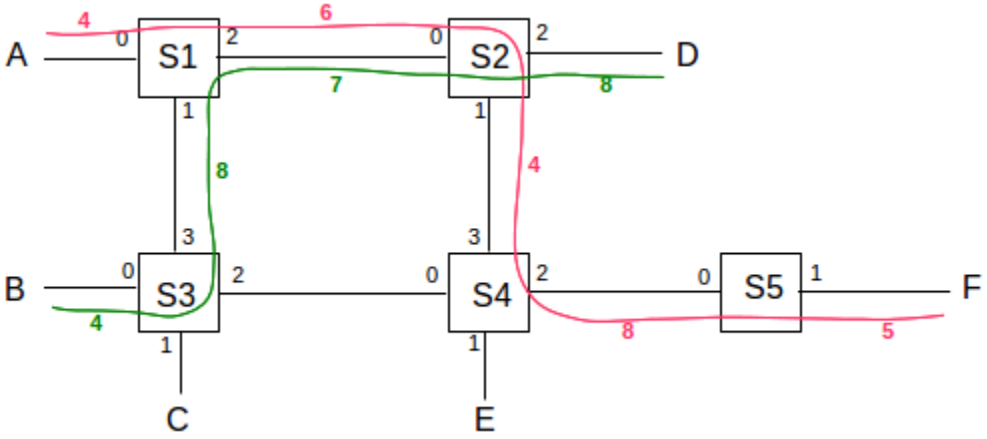
Before any packets can be sent, a connection needs to be established first. For that connection, the route is computed and then each link along the path is assigned a connection ID, traditionally called the **VCI**, for Virtual Circuit Identifier. In most cases, VCIs are only *locally* unique; that is, 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)

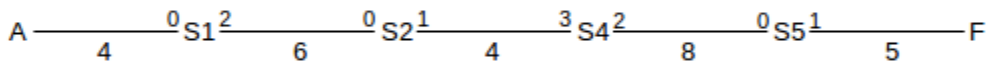
- A and F again (a separate connection)

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 $\langle VCI_{in}, port_{in} \rangle$ or the $\langle VCI_{out}, port_{out} \rangle$ 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 (*18.6 RSVP*) does exactly this. However, the fast-lookup and small-header advantages of a compact namespace are then lost.

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)

3.8 Asynchronous Transfer Mode: ATM

ATM is a LAN mechanism intended to accommodate real-time traffic as well as bulk data transfer. It was particularly intended to support voice. A significant source of delay in voice traffic is the packet **fill time**: at DS0 speeds, a voice packet fills 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. Voice delay also can introduce an annoying echo. 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, 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 forwarding incoming cells to 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.

3.8.1 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.

3.9 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 *18 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, *18.12 Multi-Protocol Label Switching (MPLS)*), but they remain a tantalizing strategy.

3.10 Exercises

1. 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. 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 (*1.14 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. 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 token were 64 bytes long (the 10-Mbps Ethernet minimum packet size), how long would it take on average to send a packet on a network with 40 stations? Ignore the spacing between packets.

(b). Repeat part (a) assuming the tokens were only 16 bytes long.

(c). Sketch a protocol by which stations could 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. The IEEE 802.11 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 Wi-Fi ACK frame would fail to be delivered in the absence of this rule, but succeed with it. Hint: this is another example of the hidden-node problem.

5. Suppose the average backoff 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 $(8\times1000)/54 = 148$ µsec to transmit a packet.

(a). How long is the average backoff interval, in µsec?
(b). What fraction of the total potential bandwidth is lost to backoff?

6. WiMAX 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).

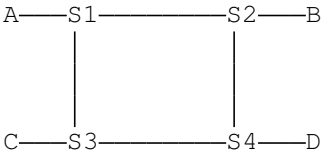
7. Suppose WiMAX subscriber stations can be moving, at speeds of up to 33 meters/sec (the maximum allowed under 802.16e).

(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. [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. In the example in *3.7 Virtual Circuits*, give the VCI table for switch S5.

10. 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.

Switch **S1**:

$\mathbf{VCI_{in}}$	$\mathbf{port_{in}}$	$\mathbf{VCI_{out}}$	$\mathbf{port_{out}}$
1	A	2	S3
2	A	2	S2
3	A	3	S2

Switch **S2**:

$\mathbf{VCI_{in}}$	$\mathbf{port_{in}}$	$\mathbf{VCI_{out}}$	$\mathbf{port_{out}}$
2	S4	1	B
2	S1	3	S4
3	S1	4	S4

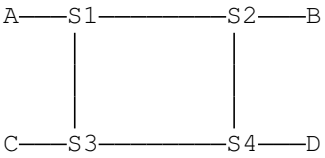
Switch **S3**:

$\mathbf{VCI_{in}}$	$\mathbf{port_{in}}$	$\mathbf{VCI_{out}}$	$\mathbf{port_{out}}$
2	S1	2	S4
3	S4	2	C

Switch **S4**:

$\mathbf{VCI_{in}}$	$\mathbf{port_{in}}$	$\mathbf{VCI_{out}}$	$\mathbf{port_{out}}$
2	S3	2	S2
3	S2	3	S3
4	S2	1	D

11. 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

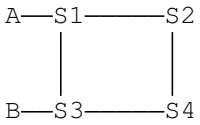
(c). B–D

(d). A–D, via whichever of S2 or S3 was *not* used in part (a)

12. 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*.



4 LINKS

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:

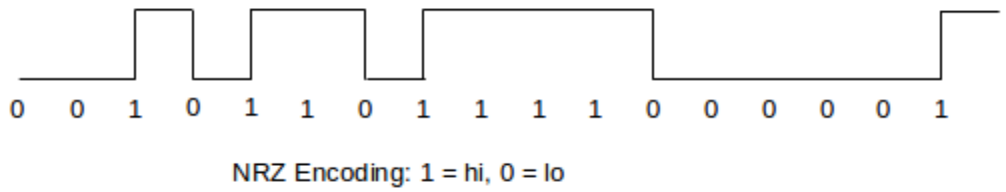
- encoding: correctly recognizing all the bits in a stream
- framing: recognizing packet boundaries

These are related, though not the same.

For long (multi-kilometer) electrical serial lines, among other things we in addition 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.



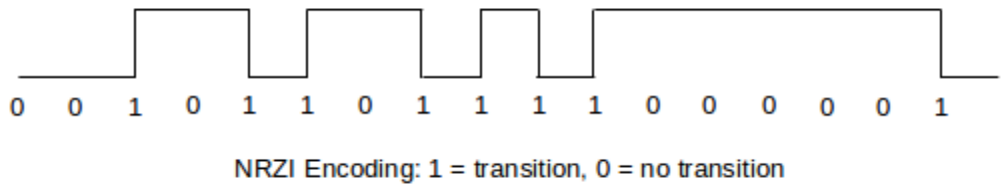
NRZ Encoding: 1 = hi, 0 = lo

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 5 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 5 1-bits or 6 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:

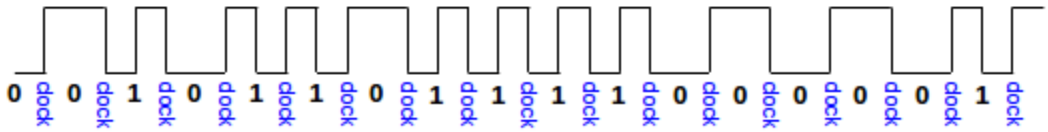


NRZI Encoding: 1 = transition, 0 = no transition

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.

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 01111100 011111010 (stuffed bits in **bold**)

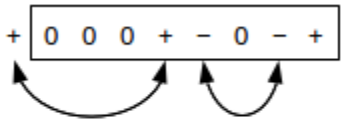
Note that bit stuffing is used by HDLC to solve two unrelated problems: the synchronization problem where long runs of the same bit cause the receiver to lose count, and the framing problem, where the transmitted bit pattern 0111 1110 now represents a flag that can never be mistaken for a data byte.

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

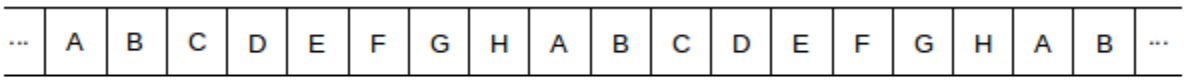
Classical **circuit switching** means a separate wire for each connection. This is still in common use for residential telephone connections: each subscriber has a dedicated wire to the Central Office. But a separate physical line for each connection is not a solution that scales well.

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 1KB 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.

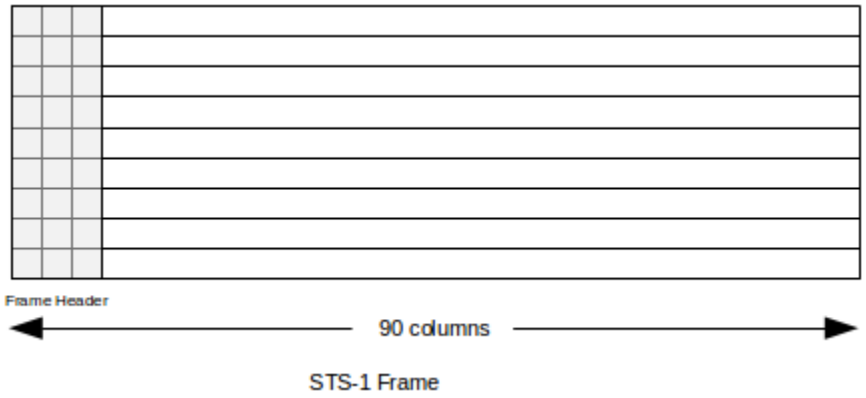
4.2.2 SONET

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 generally within reach. This mechanism solves "most" of the clock-synchronization problem, though SONET also has a resynchronization protocol in case the receiver gets lost.

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.



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 resynchronization 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 9×90-byte STS-1 frame, the payload envelope is the 9×87 region nominally following the three header columns; this payload region has its own three reserved columns meaning that there are 84 columns (9×84 bytes) available for data. This 9×87-byte payload envelope can "float" within the physical 9×90-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 9×90 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 9×270 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	bandwidth
STS-1	STM-0	51.84 Mbps
STS-3	STM-1	
STS-12	STM-4	622.08 Mbps (=12*51.84, exactly)
STS-48	STM-16	
STS-192	STM-64	
STS-768	STM-256	

Faster SONET lines have been defined, but a simpler way to achieve very high data rates over optical fiber is to use **wavelength-division multiplexing** (that is, frequency-division multiplexing at optical frequencies); this means we have separate SONET "channels" at different wavelengths of light.

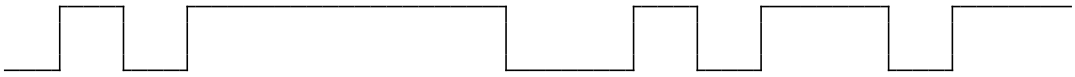
SONET provides a wide variety of leasing options at various bandwidths. High-volume customers can lease an entire STS-1 or larger unit. Alternatively, the 84 columns of an STS-1 frame can be divided into seven **virtual tributary** groups, each of twelve columns; these groups can be leased individually or in multiples, or be further divided into as few as three columns (which works out to be just over the T1 data rate).

4.3 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

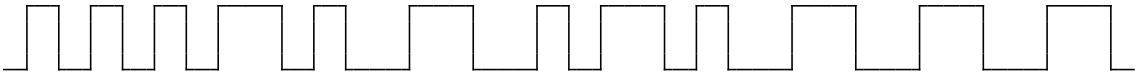
1. What is encoded by the following NRZI signal? The first bit is a 0-bit.



0

2. 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.(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.

4. What three ASCII letters (bytes) are encoded by the following 4B/5B pattern? (Be careful about uppercase vs lowercase.)

    01011010100111010101011111110

5.(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).

5 PACKETS

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 private microwave-relay links between trading centers, say New York and Chicago, in order to reduce delay. In computerized trading, milliseconds count. At 1 Gbps, 1 ms is the bandwidth delay for 125 KB. Propagation delay, though, is more interesting. 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 program trading, this 2 ms savings appears to be of significant 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 is 240 µsec = 200 µsec + 40 µsec

Case 2: A———————B

> Like the previous example except that the propagation delay is increased to 4 ms

The total transmit time is now 4200 µsec = 200 µsec + 4000 µsec

Case 3: A———R———B

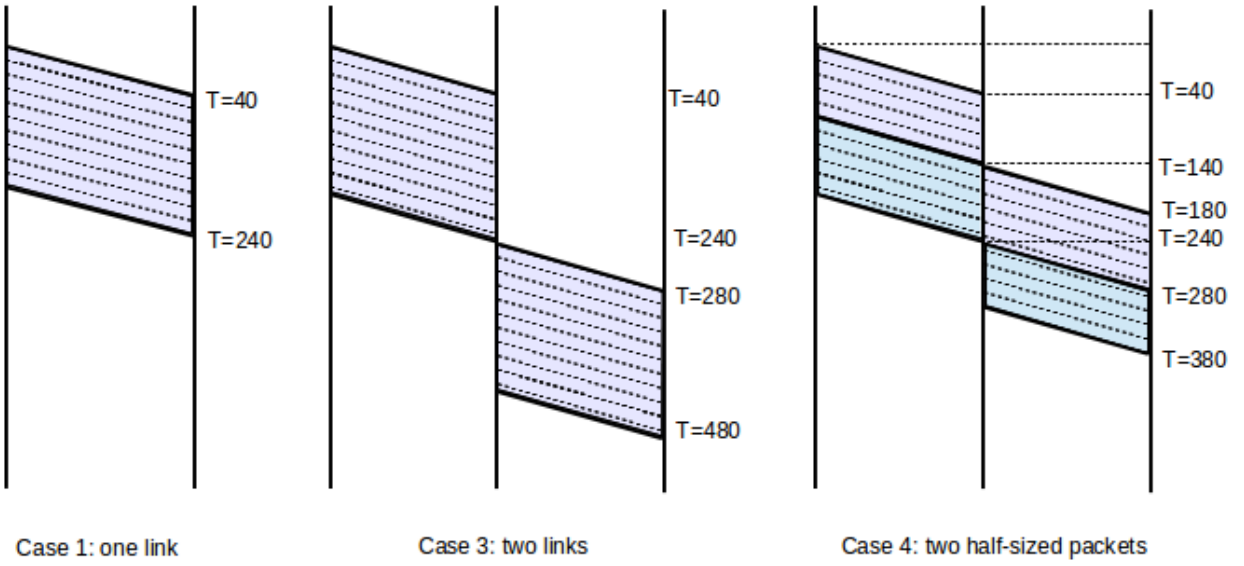
> We now have two links, each with propagation delay 40 µsec; bandwidth and packet size as in Case 1

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.



Bandwidth delay 200 µsec, per-link propagation delay 40 µsec

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.

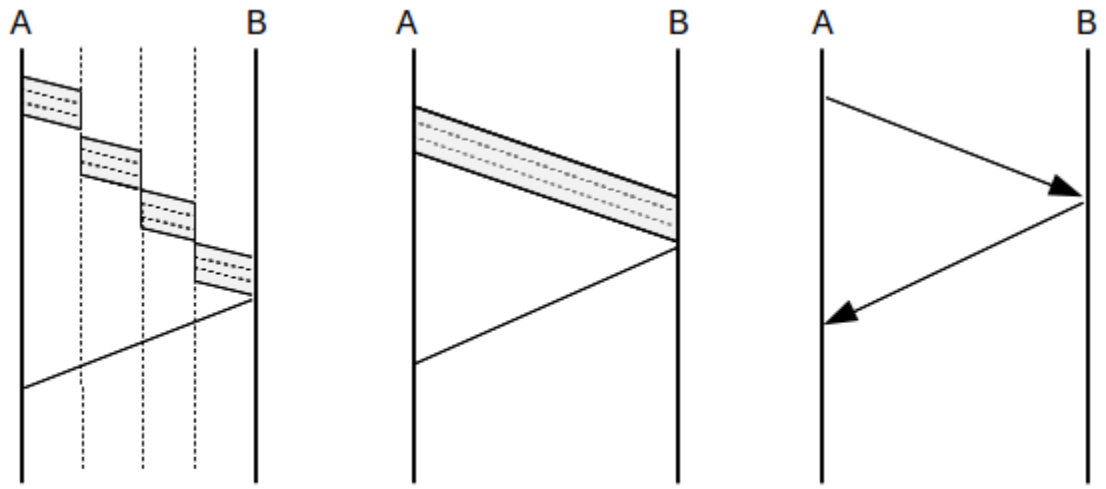
Now let us consider the situation when the propagation delay is the most significant component. The cross-continental US roundtrip delay is typically around 50-100 ms (propagation speed 200 km/ms in cable, 5,000-10,000 km cable route, or about 3-6000 miles); we will use 100 ms in the examples here. At 1.0 Mbit, 100ms is about 12KB, 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, 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

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 values

RTT	bandwidth	bandwidth × delay
1 ms	10 Mbps	1.2 KB
100 ms	1.5 Mbps	20 KB
100 ms	600 Mbps	8,000 KB

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 $\text{RTT}_{\text{noLoad}}$ 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.

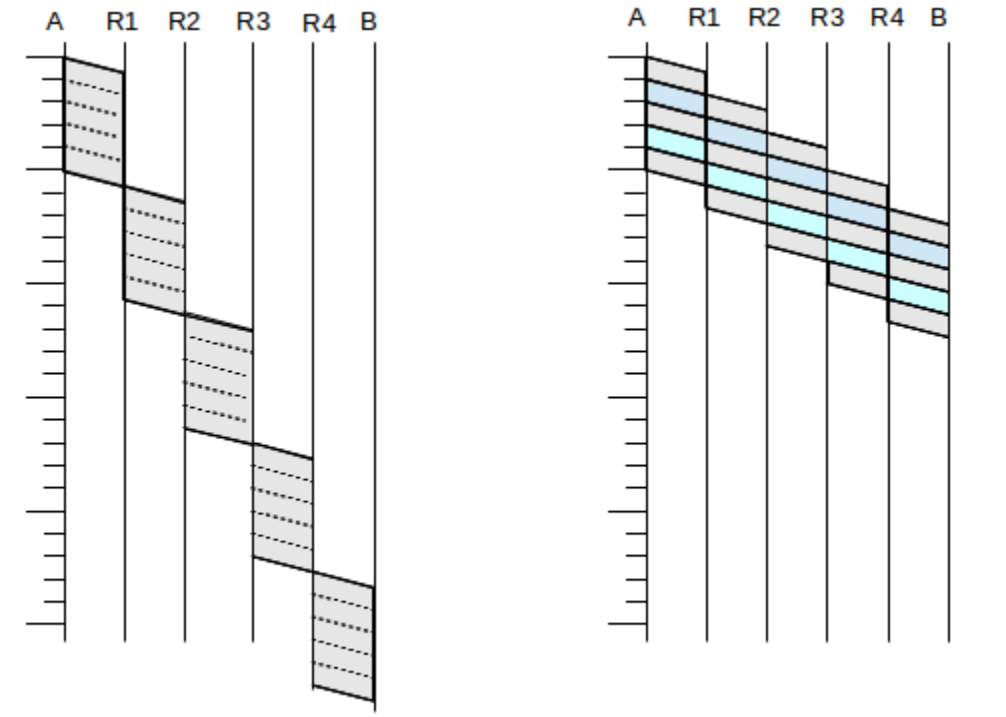
On the other hand, large packets waste a smaller percentage of bandwidth on headers. However, in most of the cases we will consider, this percentage does not exceed 10% (the VoIP/RTP example in *1.3 Packets* is an exception).

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:

A————R1————R2————R3————R4————B

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.

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%. 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 *18 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.

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 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 used to permit.

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.

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. 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.

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 $\times$ 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, which tends to sound complicated but which in fact 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 32 (at least for CRC-**32** codes); the full specification of a given CRC code requires giving this polynomial. We divide m(X) by p(X); our "checksum" is the remainder r(X), of maximum degree 31 (that is, 32 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 mathematical 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 1110. 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.

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. 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 take considerable time to compute; 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$\times$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$\times$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.

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**11**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 (*3.3.2.3 Wi-Fi Fragmentation*) is directly tied to *5.3.1 Error Rates and Packet Size*.

5.6 Exercises

1. 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. 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. Repeat parts (a) and (b) of the previous exercise, except change the per-link propagation delay from 60 µsec to 600 µsec.

4. 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?

5. 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. Now how long does it take to send two back-to-back 300-byte packets from A to B?

6. 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. 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.

A —— S_1 —— ... —— S_{N-1} —— B

Let BD be the bandwidth delay of a single packet on a single link, and let PD be the propagation delay on a single link. Show that the total bandwidth delay is (M+N-1)×BD, and the total propagation delay is N×PD. You do not have to be rigorous, but your argument should work both when M is significantly larger than N and also when N is significantly larger than M.

8. 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. 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. In the description in the text of the Internet checksum, the overflow bit was added back in after each ones-complement addition. Show that the same final result will be obtained if we add up the 16-bit words using 32-bit twos-complement arithmetic (the normal arithmetic on all modern hardware), and then add the upper 16 bits of the sum to the lower 16 bits. (If there is an overflow at this last step, we have to add that back in as well.)

11. 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. 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. 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. (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. 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. (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

6 ABSTRACT SLIDING WINDOWS

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.

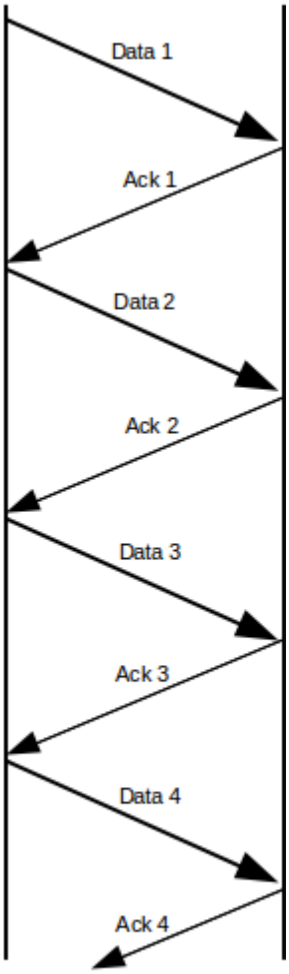
In addition to reliability, we also want to keep as many packets in transit as the network can support. The strategy used to achieve this is known as **sliding windows**. It turns out that the sliding-windows algorithm is also the key to managing congestion; we return to this in *13 TCP Reno and Congestion 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). 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].

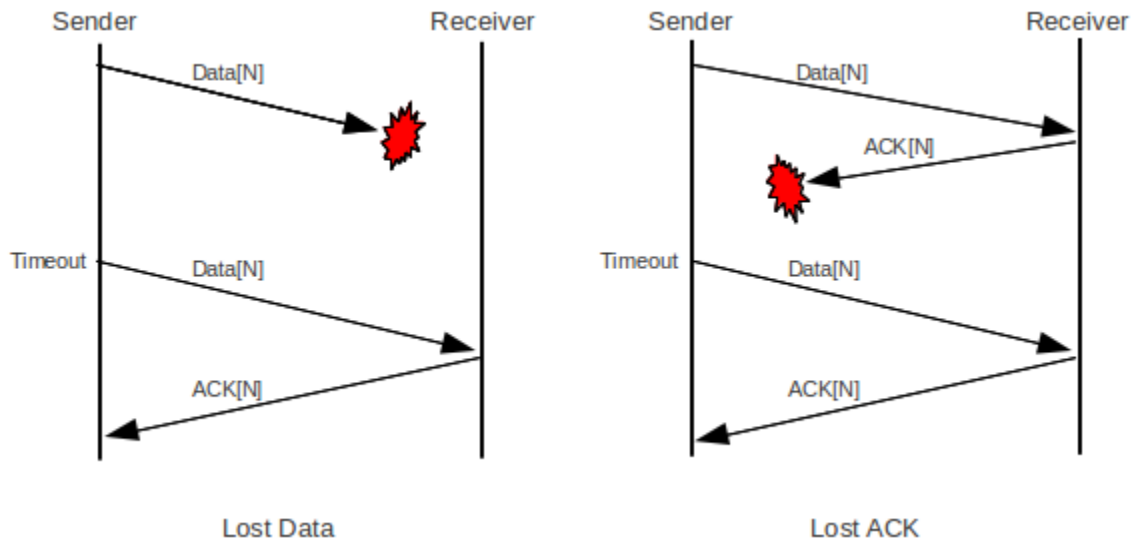
In the **stop-and-wait** version of retransmit-on-timeout, the sender sends only one outstanding packet at a time. If there is no response, the packet may be retransmitted, but the sender does not send Data[N+1] until it has received ACK[N]. Of course, the receiving side will not send ACK[N] until it 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

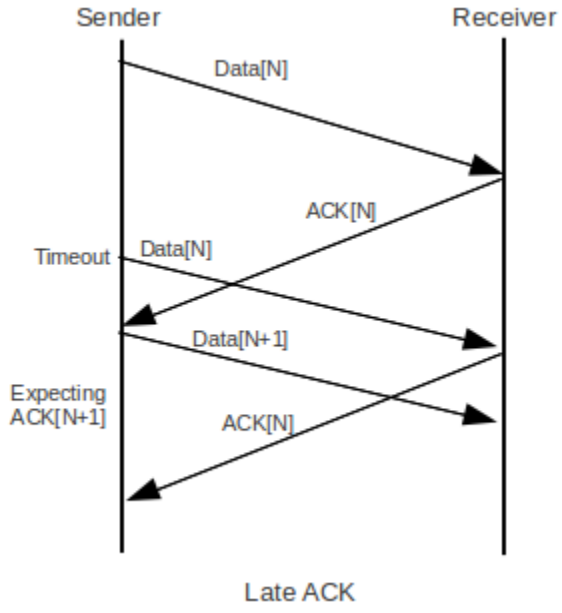
6.1.1 Packet Loss

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* ACK[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]).

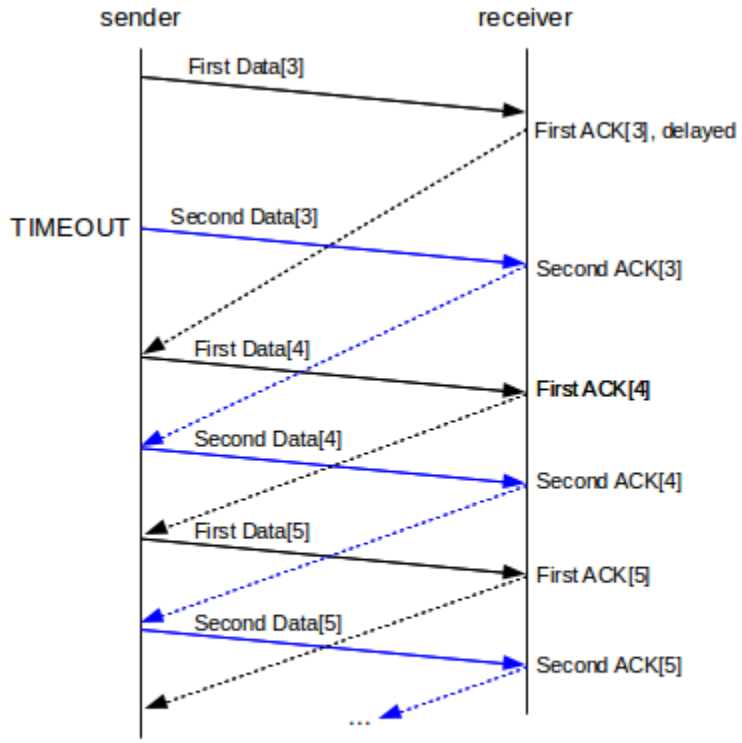
At least one side *must* implement retransmit-on-timeout; otherwise a lost packet leads to deadlock as the sender and the receiver both wait forever. The other side *must* implement at least one of retransmit-on-duplicate or retransmit-on-timeout; usually the former alone. If both sides implement retransmit-on-timeout 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, http://www.youtube.com/watch?v=XChxLGnIwCU, 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, 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 retransmit-on-duplicate strategy.



The Sorcerer's Apprentice bug
First transmissions are in black
Second transmissions are in blue

All packets are sent *twice* from Data[3] on. The transfer completes normally, but takes double the normal bandwidth.

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 *12.16 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 $K \leq N$.

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.

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.

Here is a video of sliding windows in action, with winsize = 5. The second link, R–B, has a capacity of five packets in transit either way; the A–R link has a capacity of one packet in transit either way.

The packets in the very first RTT represent connection setup. This particular video also demonstrates TCP "Slow Start": in the first *data-packet* RTT, two packets are sent, and in the second data RTT four packets are sent. The full window size of five is reached in the third data RTT. For the rest of the connection, at any moment (except those instants where packets have just been received) there are five packets "in flight", 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 the video).

6.2.1 Bandwidth × Delay

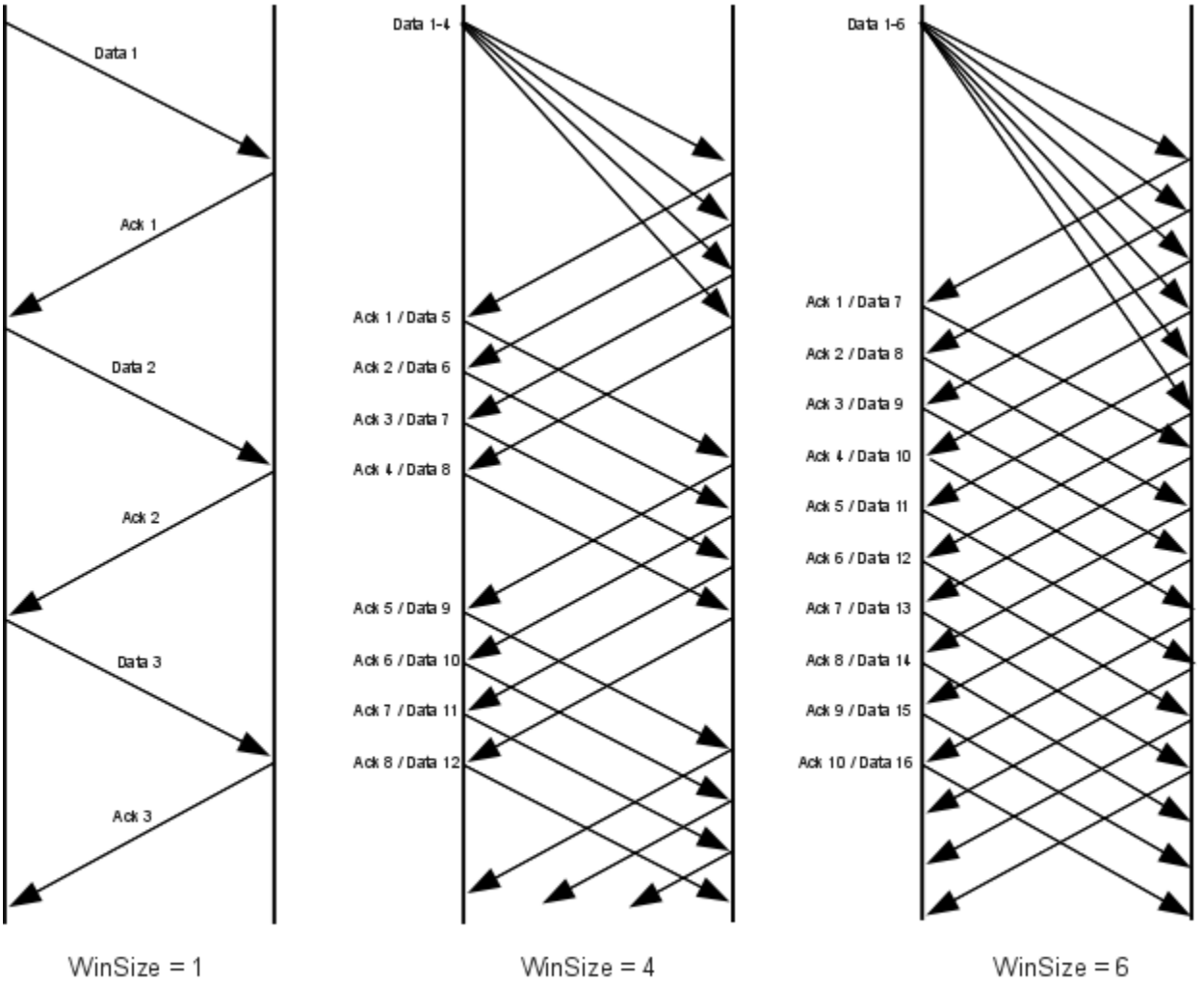
As indicated previously (*5.1 Packet Delay*), the bandwidth × RTT product represents the amount of data that can be sent before the first response is received. This is generally the optimum size for the window size.

There is, however, one catch: if a sender chooses winsize larger than the bandwidth × RTT product, 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 $\text{RTT}_{\text{actual}}$ to well above $\text{RTT}_{\text{noLoad}}$; see *6.3.1.3 Case 3: winsize = 6* below for an example. For this reason, a sender is often more interested in bandwidth × $\text{RTT}_{\text{noLoad}}$, 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 × $\text{RTT}_{\text{noLoad}}$ 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 $\times$ 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 $\times$ RTT matches winsize; that is, a connection's own traffic can inflate $\text{RTT}_{\text{actual}}$ to well above $\text{RTT}_{\text{noLoad}}$. 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 occurs, 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[lastACKed+1] through Data[lastACKed+winsize]. For any but the first of these, the receiver must buffer the arriving packet. If Data[lastACKed+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

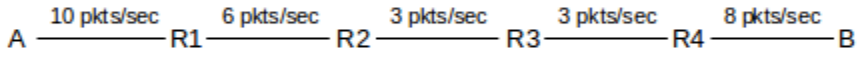
Suppose winsize = 4 and packet 5 is lost. It is quite possible that packets 6, 7, and 8 may have been received. However, the only (cumulative) acknowledgment that can be sent back is ACK[4]; the sender does not know how much of the windowful made it through. Because of the possibility that *only* Data[5] (or more generally Data[lastACKed+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 time-out, 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.



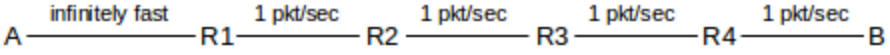
The slow links are R2–R3 and R3–R4. We will refer to the slowest link as the **bottleneck** link; if there are (as here) ties for the slowest link, then the first such link is the bottleneck. The bottleneck link is where the queue will form. If traffic is sent at a rate of 4 packets/ms from A to B, it 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.

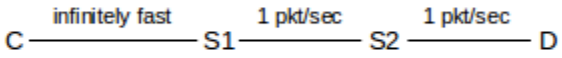
6.3.1 Simple fixed-window-size analysis

We will analyze the effect of window size on overall throughput and on RTT. Consider the following network path, with bandwidths 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 an 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.

This model 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.

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.

We assume a single connection is made; *ie* there is no competition. Bandwidth $\times$ delay here is 4 packets (1 packet/sec $\times$ 4 sec RTT)

6.3.1.1 Case 1: winsize = 2

In this case winsize < bandwidth$\times$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

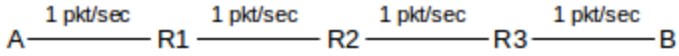
T	A sends	R1 queues	R1 sends	R2 sends	R3 sends	R4 sends	B ACKs
0	1,2,3,4,5,6	2,3,4,5,6	1				
1		3,4,5,6	2	1			
2		4,5,6	3	2	1		
3		5,6	4	3	2	1	
4	7	6,7	5	4	3	2	1
5	8	7,8	6	5	4	3	2
6	9	8,9	7	6	5	4	3
7	10	9,10	8	7	6	5	4
8	11	10,11	9	8	7	6	5
9	12	11,12	10	9	8	7	6
10	13	12,13	11	10	9	8	7

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$\times RTT_{noLoad}$ 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$\times$RTT product.

In general, whenever winsize > bandwidth$\times RTT_{noLoad}$, 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$\times RTT_{actual}$. 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 isolated 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 $\times$ 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 $T_D \leq T < T_A$ 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.

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$\times$queue_time (where throughput is measured in packets/sec):

3. queue_usage = throughput $\times$ ($RTT_{actual} - RTT_{noLoad}$)
   = winsize $\times$ (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 $\times$ ($RTT_{actual} - RTT_{noLoad}$), 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 $\times$ RTT_{noLoad} – then winsize is the limiting factor in throughput. Finally, if there is no competition and winsize $\geq$ bandwidth $\times$

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

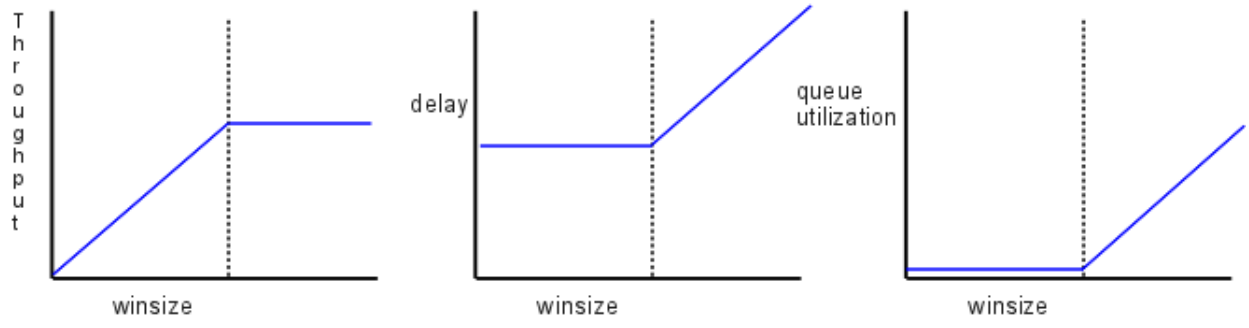
Regardless of the value of winsize, in the steady state the sender never sends faster than the bottleneck bandwidth. This is because the bottleneck bandwidth determines the rate of packets arriving at the far end, which in turn determines the rate of ACKs arriving back at the sender, which in turn determines 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:

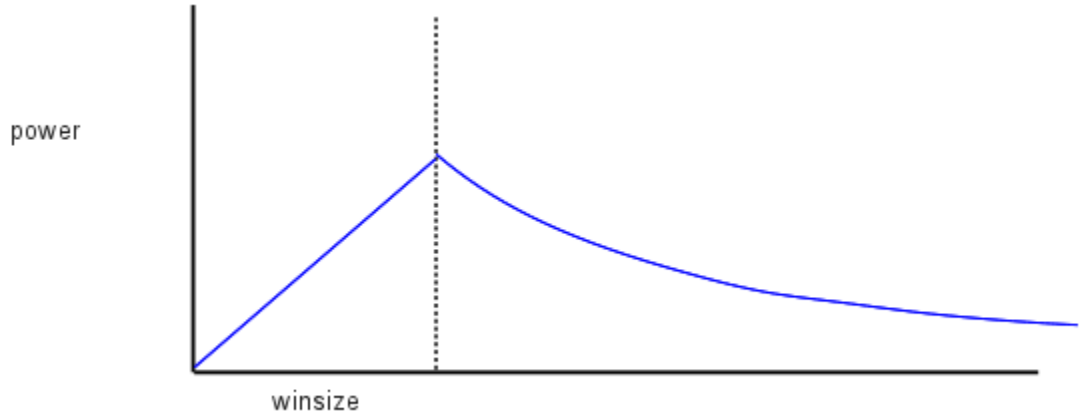
- throughput is proportional to winsize
- delay is constant
- queue utilization in the steady state is zero

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. Winsize is W; we abbreviate lastACKed by LA. Thus, the next packet expected is LA+1 and the window is LA+1 ... LA+W.

We have a pool EA of "early arrivals": packets buffered for future use. When packet Data[M] arrives:

```
if M≤LA or M>LA+W, ignore the packet
if M>LA+1, put the packet into EA.
if M==LA+1,
        output the packet (that is, Data[LA+1])
        LA = LA+1 (slide window forward by 1)
        while (Data[LA+1] is in EA) {
                output Data[LA+1]
                LA = LA+1
        }
        send ACK[LA]
```

There are a couple details omitted.

A possible implementation of EA is as an array of packet objects, of size W. We always put packet Data[K] into position K % W.

At any point between packet arrivals, Data[LA+1] is not in EA, 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:

```
if M≤LA or M>LA+W, ignore the packet
otherwise:
    set K = LA+1
    set LA = M, the new bottom edge of the window
    for (i=K; i≤LA; i++) send Data[i]
```

Note that new ACKs may arrive while we are in the loop at that last line. We assume here that the sender stolidly sends what it may send and only after that does it start to process 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.

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 TFTP Stop-and-Wait* (stop-and-wait) and in *12.13 TCP Sliding Windows*; the latter is one of the most important mechanisms on the Internet.

6.5 Exercises

1. Sketch a ladder diagram for stop-and-wait if Data[3] is lost the first time it is sent. Continue the diagram to the point where Data[4] is successfully transmitted. Assume an RTT of 1 second, no sender timeout, and a *receiver* timeout of 2 seconds.

2. 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 it arrives. What is the first ACK[N] that it will be able to determine is incorrect? Assume no packets are lost.

(b). Is there anything the transmitter can do to detect this receiver problem earlier?

3. 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–R link and all reverse links (from B to A) are infinitely fast. Carry out the table for 10 seconds.

4. 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.

5. 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. How many packets are in the queue, at the steady state, and what is RTT_{actual}?

6. 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. Carry out the table for 10 seconds. Note that in this exercise you will need to show the queue for both R1 and R2. (If you carry out the pattern at least partially until T=18, you can verify that RTT_{actual} for packet 8 is as calculated in the previous exercise. 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 literally, "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. 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).

8. 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?

⋆9. 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. 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. 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?

7 IP VERSION 4

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 somewhere or if you want to run your own transport protocol, the Internet backbone will still work for you. But if you want to change the IP layer, you may encounter difficulty. (Just talk to the IPv6 people, or the IP-multicasting or IP-reservations groups.)

IP is, in effect, a universal **routing and addressing** protocol. The two 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 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.

Currently the Internet uses (almost 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 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.

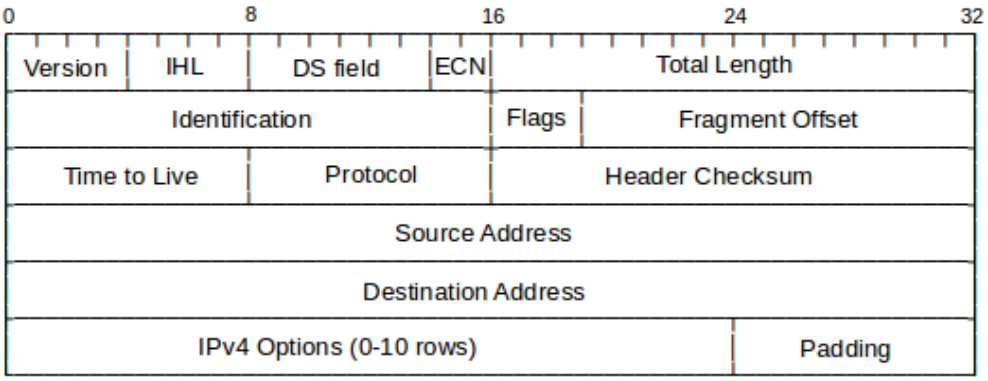
If you want to provide a universal service for delivering any packet anywhere, what else do you need besides routing and addressing? Every network (LAN) needs to be able to carry any packet. The protocols spell out the use of octets (bytes), so the only possible compatibility issue is that a packet is *too large* for a given network. IP handles this by supporting **fragmentation**: a network may break a too-large packet up into units it can transport successfully. While IP fragmentation is inefficient and clumsy, it does guarantee that any packet can potentially be delivered to any node.

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.

Here is how it is all laid out in the header. Each row is 32 bits wide.



The IP header, and basics of IP protocol operation, were 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 IP Header Length, in 32-bit words; an IP header can thus be at most 15 words long. The base header takes up five words, so the IP 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 IP packet may be smaller than the minimum LAN packet size (see Exercise 1) or larger than the maximum (if the IP packet has been fragmented over several LAN packets. The IP packet length, in other words, cannot be inferred from the LAN-level packet size. Because the Total Length field is 16 bits, the maximum IP packet size is 2^{16} bytes. This is probably much too large, even if fragmentation were not something to be avoided.

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 indicate if the body of the IP packet represents a TCP packet or a UDP packet, or, in unusual cases, something else altogether.

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 **Source** and **Destination Address** fields contain, of course, the IP addresses. These would be updated only by NAT firewalls.

One option is the **Record Route** option, in which routers are to insert their own IP address into the IP header option area. Unfortunately, with only ten words available, there is not enough space to record most longer routes (but see *7.9.1 Traceroute and Time Exceeded*, below). Another option, now deprecated as security risk, is to support **source routing**. The sender would insert into the IP header option area a list of IP addresses; the packet would be routed to pass through each of those IP addresses in turn. With **strict** source routing, the IP addresses had to represent adjacent neighbors; no router could be used if its IP address were not on the list. With **loose** source routing, the listed addresses did not have to represent adjacent neighbors and ordinary IP routing was used to get from one listed IP 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.

7.2 Interfaces

IP addresses (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 address 127.0.0.1 (although the IPv6 address ::1 is also used). Delivering packets to the localhost address 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 convenient and fast. 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 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 Network*, 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*

Network). 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. Unfortunately, finding the default router 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.

Many workstations have both an Ethernet interface and a Wi-Fi interface. Both of these can be used simultaneously (with different IP addresses assigned to each), either on the same IP network or on different IP networks.

Routers always have at least two interfaces on two separate LANs. Generally this means a separate IP address for each interface, though some point-to-point interfaces can be used without being assigned an IP address (*7.10 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 LAN; the interface might be assigned one IP address for each IP network. Other times a single interface is assigned multiple IP addresses that are on the same LAN; 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 IP addresses represent special cases.

While the standard IPv4 loopback address is 127.0.0.1, any IP 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. 127.255.33.57 – should work, *eg* with `ping`.

Private addresses are IP 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 IP 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 (*7.8.1 DHCP and the Small Office*) built into NAT routers.

Broadcast addresses are a special form of IP 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 IP 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, **IP 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 IP 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 IP multicasting appears in *18 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? IP 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 IP 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 IP to join Token Ring (MTU = 4KB, at least

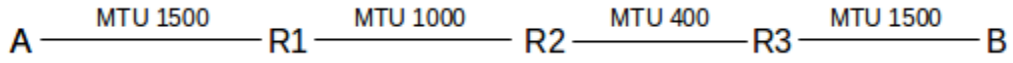
originally) to Ethernet (MTU = 1500B), the token-ring packets might be too large to deliver to the Ethernet side, or to traverse an Ethernet backbone *en route* to another Token Ring. (Token Ring, in its day, did commonly offer a configuration option to allow Ethernet interoperability.)

So, IP must support fragmentation, and thus also reassembly. There are two major 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 IP, partly because intermediate routers are too busy to do reassembly (this is as true today as it was thirty years ago), and partly because IP fragmentation is seen as the strategy of last resort.

An IP 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 IP 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 IP packet. Note that the start position can be a number up to 2^{16}, the maximum IP 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 IP 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 IP-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" bit. If it is set to 1 by the sender then a router must *not* fragment the packet and must drop it instead; see *12.12 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 IP 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 (*12.12 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 Sharing (NFS) protocol, for example, UDP is used to carry 8KB disk blocks. These are often sent as a single 8+ KB IP 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 *3.8.1 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 IP address can be divided into a net portion IP_{net} and a host portion IP_{host}; the division point was determined by whether the IP address was a Class A, a Class B, or a Class C. We also hinted 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 IP 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 IP 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 IP 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 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 bits). As with the 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 IP 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. In *10 Large-Scale IP Routing* we will see that in some cases multiple matches in the forwarding table may exist; the **longest-match** rule will be introduced to pick the best match.

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 network address 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.

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 *7.10 Unnumbered Interfaces*.

7.6 IP 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 IP 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.

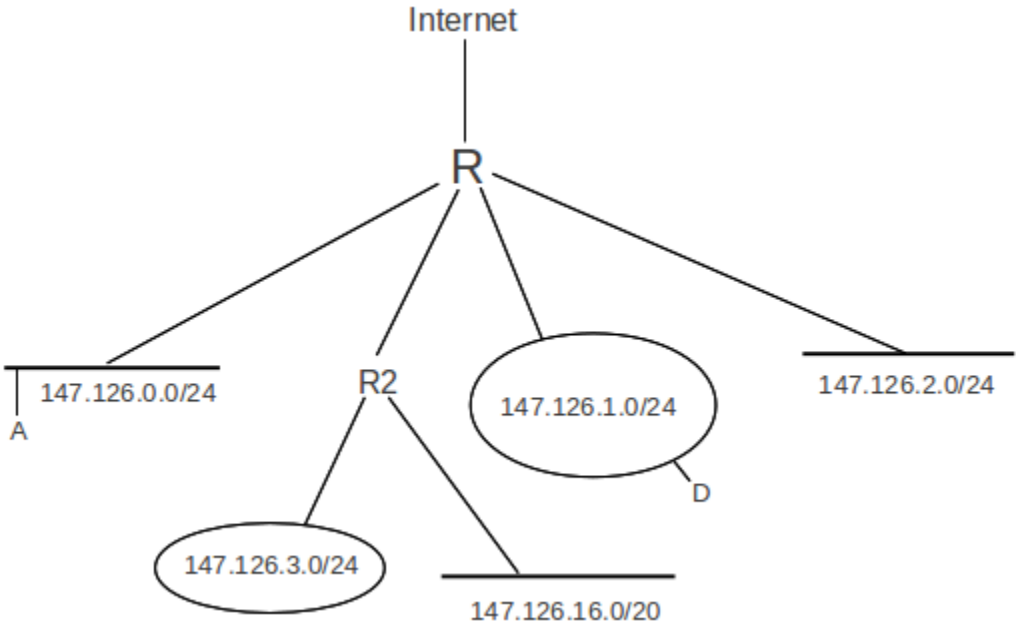
Subnets were first proposed in **RFC 917**, and became official with **RFC 950**.

Getting a separate IP 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 IP 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 IP network into some combination of physical LANs – the subnets

–, and assign each a **subnet address**: an IP network address which has the *site's* IP network address as prefix. To put this more concretely, suppose the site's IP 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 IP 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 IP address with the mask is the same as extracting the first 24 bits of the IP 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 IP 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 IP-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" IP 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 IP 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 IP 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 the same subnet and with hosts outside the university entirely. Both of these worked with the bad subnet mask; it is different-subnet local hosts that are 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 switching requires no configuration at all, 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. The industry has come up with the term "Layer 3 Switch" to describe routers that in effect do not support all the usual firewall bells and whistles; such streamlined routers may also be able to do most of the hard work in specialized hardware, another source of speedup. (It should be no surprise that Layer 3 switching is usually marketed in terms of the hardware speedup, not in terms of the reduced flexibility.)

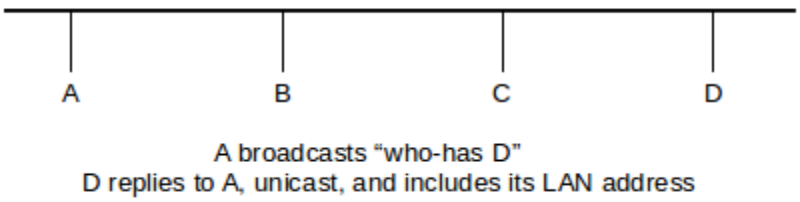
Switching scales remarkably well, but it does have limitations. First, broadcast packets must be forwarded throughout a switched network; they do not, however, pass to different subnets. Second, LAN networks do not like redundant links (that is, loops); while one can rely on the spanning-tree algorithm to eliminate these, that algorithm too becomes less efficient at larger scales.

7.7 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. This means looking up the LAN address D_{LAN} corresponding to D_{IP}.

One approach would be via a special server, but the spirit of early IP 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 IP 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.



Additionally, all hosts maintain an **ARP cache**, consisting of ⟨IP,LAN⟩ address pairs for other hosts on the network. After the exchange above, A has ⟨D_{IP},D_{LAN}⟩ in its table; anticipating that A will soon send it a packet to which it needs to respond, D also puts ⟨A_{IP},A_{LAN}⟩ 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 card replaced.

7.7.1 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 IP 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 IP address, spaced over a few seconds and leaving the ARP field for the sender's IP address filled with zeroes. This last step means that any other host with that IP 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 IP 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 *7.11 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 work quite well: 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

7.7.2 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, 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.

7.7.3 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.

7.7.4 Detecting Sniffers

Finally, here 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. 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.

7.8 Dynamic Host Configuration Protocol (DHCP)

DHCP is the most common mechanism by which hosts are assigned their IP 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 IP 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 IP address is assigned to it. The DHCP response is likely to contain several other essential startup options as well, including:

- IP address
- subnet mask
- default router
- DNS Server

These four items are a pretty standard **minimal network configuration**.

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 IP addresses to hand out, and maintains a database of which IP 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).

7.8.1 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 (*1.14 Network Address Translation*) and DHCP has made **autoconfiguration** close to a reality.

The typical home/small-office "router" is in fact a NAT firewall coupled with a switch (usually also coupled with a Wi-Fi access point); we will refer to this as a **NAT router**. One specially designated port, the **external** port, connects to the ISP's line, and uses DHCP to obtain an IP 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 IP address block, *eg* 192.168.0.0/24. The NAT router contains a DCHP server, usually enabled by default, that will hand out IP 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, as two different hosts may be assigned the same IP address, or a host's IP address may suddenly change if it gets a new IP address from the other server. Disabling one of the DHCP servers fixes this.

While omnipresent DHCP servers have made IP autoconfiguration work "out of the box" in many cases, in the era in which IP 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 (*8.13 Stateless Autoconfiguration (SLAAC)*) that does not require DHCP, though DHCPv6 may well end up displacing it.

7.8.2 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 IP address to assign.

This feature has to be specially enabled on the router.

7.9 Internet Control Message Protocol

The Internet Control Message Protocol, or ICMP, is a protocol for sending IP-layer error and status messages. It 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.

Here are the ICMP types, with subtypes listed in the description. The list has been pruned to include only the most common values.

Type	Description
Echo Request	`ping` queries
Echo Reply	`ping` responses
Destination **host** unreachable	Destination **network** unreachable
Destination **port** unreachable	
Fragmentation required but DF flag set	
Network administratively prohibited	
Source Quench	Congestion control
Redirect datagram for the **host**	Redirect datagram for the **network**
Redirect for TOS and network	
Redirect for TOS and host	
Router Solicitation	Router discovery/selection/solicitation
Fragment reassembly time exceeded	TTL expired in transit
Missing a required option	Pointer indicates the error
Bad length	

ICMP is perhaps best known for Echo Request/Reply, on which the `ping` tool 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 blocked by default by many firewalls, on the theory that revealing even the existence of computers is a security risk. While this may be an appropriate

decision, it does significantly impair the utility of ping. Most routers do still pass ping requests, but some site routers block them.

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 IP header was set, forbidding fragmentation. Later we will see how TCP uses this option as part of **Path MTU Discovery**, the process of finding the largest packet we can send *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.
- **Administratively Prohibited**: this is sent by a router that knows it can reach the network in question, but has configured 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.

7.9.1 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.

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.

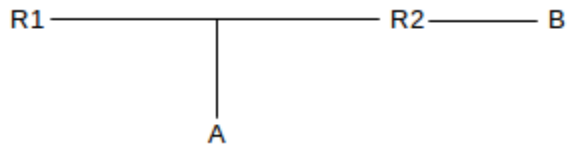
Route Efficiency

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).

7.9.2 Redirects

Most non-router hosts start up with an IP forwarding table consisting of a single (default) router, discovered along with their IP 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.

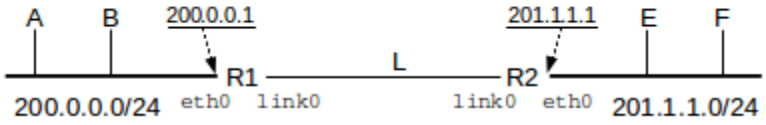
7.9.3 Router Solicitation

These ICMP messages are used by some router protocols to identify immediate neighbors. When we look at routing-update algorithms, below, these are where the process starts.

7.10 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.

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 IP 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.

7.11 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 IP 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 *3.3.5 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.

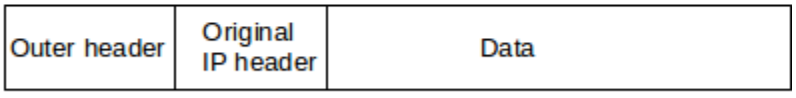
To maintain connectivity to the home address, a Mobile IP host needs to have a **home agent** back on the home network; the job of the home agent is to maintain an IP tunnel that always connects to the device's current care-of address. Packets arriving at the home network addressed to the home address 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 (*7.7.1 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.

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.

Original packet



Encapsulated packet with outer header

A special value in the IP-header Protocol field indicates that IP-in-IP tunneling was 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.

7.12 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 very limited adoption. The biggest issue fixed by IPv6 is IPv4's lack of address space, but there are also several other less dramatic improvements.

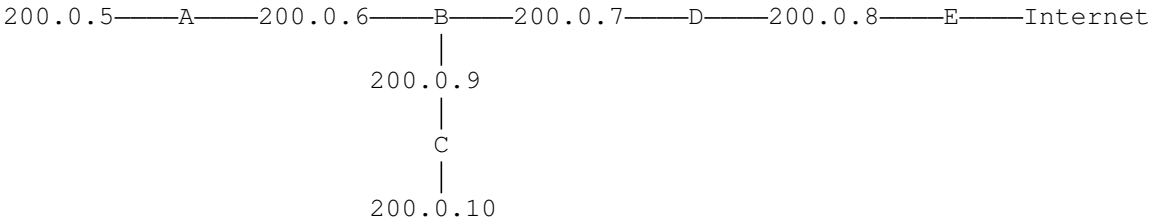
7.13 Exercises

1. Suppose an Ethernet packet represents a TCP acknowledgment; that is, the packet contains an IP header and a 20-byte TCP header but nothing else. Is the IP packet here smaller than the Ethernet minimum-packet size, and, if so, by how much?

2. How can a receiving host tell if an arriving IP packet is unfragmented?

3. How long will it take the IDENT field of the IP 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. 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.



5. (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 IP 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. For each IP network prefix given (with length), identify which of the subsequent IP 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

7. Suppose that the subnet bits below for the following five subnets A-E all come from the beginning of the fourth byte of the IP 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 IP addresses do not belong to any subnet?

8. In Section 4.6 it was stated that, in newer implementations, "repeat ARP queries about a timed out entry are first sent unicast". Why is this likely to succeed most of the time? Under what conditions would the unicast query fail?

9. Suppose A broadcasts an ARP query "who-has B?", receives B's response, and proceeds to send B a regular IP packet. If B now wishes to reply, is it likely that A will already be present in B's ARP cache? Why?

10. 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?

8 IP VERSION 6

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 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"), as the basis for IPv6.

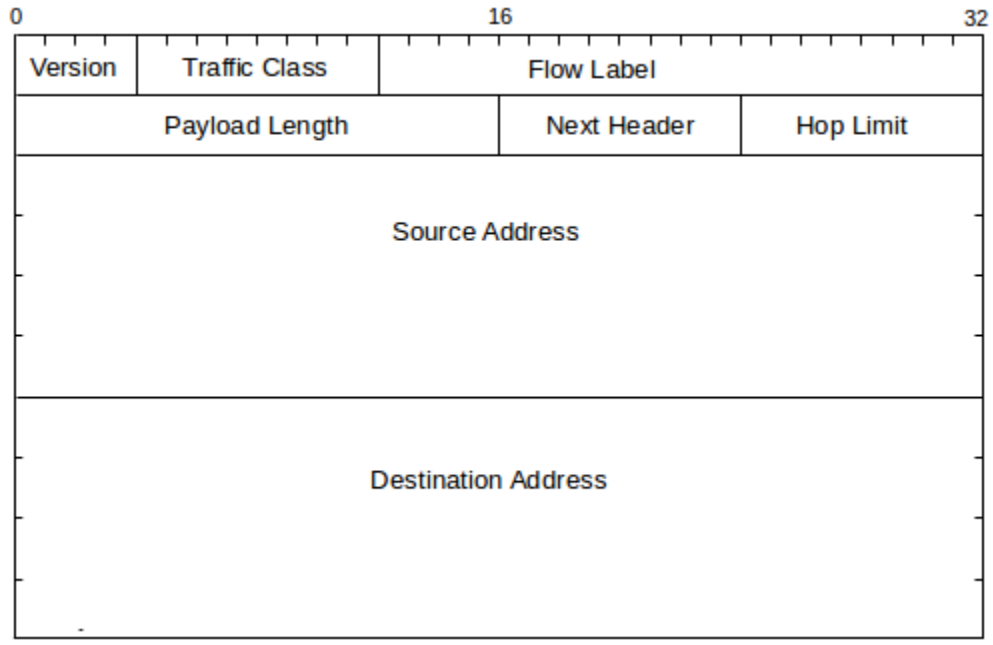
SIP addresses were originally to be 64 bits in length, but in the month leading up to adoption this was increased to 128. 64 bits would probably have been enough, but the problem is less the actual number than the simplicity with which addresses can be allocated; the more bits, the easier this becomes, 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, 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: in most ordinary-host cases, the first 64 bits of the address is the network portion (including any subnet portion) and the remaining 64 bits represent the host portion. While there are some configuration alternatives here, and while the IETF occasionally revisits the issue, at the present time the 64/64 split seems here to stay. Routing, however, can, as in IPv4, be done on different prefixes of the address at different points of the network. Thus, it is misleading to think of IPv6 as a return to Class A/B/C address allocation.

IPv6 is now twenty years old, and yet usage 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** looks like the following; 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 below. Some fixed-header header fields have 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, below. Note that 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 been used at various other scales: a single TCP connection, multiple *related* TCP connections, or even all traffic from a particular subnet (*eg* the "computer-lab flow").

8.1.1 IPv6 addresses

IPv6 addresses are written in eight groups of four hex digits, separated by colons, and with leading 0's optionally removed, eg

> FEDC:BA98:1654:310:FEDC:BA98:7654:3210

If an address contains a long run of 0's, as would be the case if the IPv6 address had an embedded IPv4 address, then when writing the address the string "::" may 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 the standard IPv6 format for representing IPv4 addresses. A separate representation of IPv4 addresses, with the "FFFF" block replaced by 0-bits, is used for tunneling IPv6 traffic over IPv4. 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) and a 64-bit host identifier. See *8.8.1 Network Prefixes*.

8.2 Host identifier

If desired, the second 64 bits of an IPv6 address can be a **host identifier** derived from the LAN address. To create the host identifier from a 48-bit Ethernet address, for example, 0xFFFE is inserted 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) is then set to 1. The result is officially known as the Modified EUI-64 Identifier, where EUI stands for Extended Unique Identifier; details can be found in **RFC 4291**. For example, for a host with Ethernet address 00:a0:cc:24:b0:e4, the EUI-64 address would be 02a0:ccff:fe24:b0e4 (note the leading 00 becomes 02 when the seventh bit is turned on).

8.3 Link-local addresses

IPv6 defines **link-local** addresses, intended to be used only on a single LAN (and never routed). These begin with the link-local prefix of the ten bits 1111 1110 10 followed by 54 more zero bits; that is, FE80::/64. The final 64 bits are the host identifier for the link interface in question, above. The Ethernet link-local address of the machine in the previous paragraph with Ethernet address 00:a0:cc:24:b0:e4 is fe80::2a0:ccff:fe24:b0e4.

When sending to a link-local address, one must include somewhere an identification of which link to use. IPv4 addresses, on the other hand, almost always implicitly identify the link by virtue of the network prefix. See *8.16 ping6* and *8.17 TCP connections with link-local addresses* for examples of sending to link-local addresses.

Once the link-local address is created, it must pass the **duplicate-address detection** test before being used; see *8.12 Duplicate Address Detection*.

8.4 Anycast addresses

IPv6 also introduces **anycast** addresses. An anycast address might be assigned to each of a set of routers (in addition to each router's own unicast address); 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.

8.4.1 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 straightforward; it is general IP-layer multicast (*18.5 Global IP Multicast*) that is problematic. See *2.1.1 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; 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.

8.4.2 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 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 routing header is an exception.

8.5 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.

8.6 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 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.

8.7 Routing Header

The routing header contains a list of IPv6 addresses through which the packet should be routed. These do not have to be contiguous: if the list to be visited en route to destination D is ⟨R1,R2,...,Rn⟩, then the IPv6 option header contains ⟨R2,R3,...,Rn,D⟩ and the initial destination address is R1; R1 then updates this header to ⟨R1,R3,...,Rn,D⟩ (that is, the old destination R1 and the current next-router R2 are swapped), and sends the packet on to R2. This continues on until Rn addresses the packet to the final destination D. The header contains a Segments Left pointer indicating the next address to be processed, so that when the packet arrives at D the Routing Header contains 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 do not need to examine this header; they process the packet based only on its current destination address.

8.8 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 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.

8.8.1 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 host portion. As of this writing this is 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 we will assume here that it remains in force. This 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/host division point is fixed, routers may still use CIDR and may still route on any shorter prefix than /64.

A typical IPv6 site may involve a variety of specialized network prefixes, including the link-local prefix and prefixes for anycast and multicast addresses. Private IPv6 address prefixes, corresponding to IPv4's 10.0.0.0/8, may also be in use.

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. 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 often 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 formally deprecated in **RFC 3879**. The first 8

bits of such a prefix are 1111 1101 (FD00::/8); the related prefix 1111 1100 (FC00::/8) is reserved for future use. The last 16 bits of the 64-bit 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 prefix is "almost certainly" globally unique, although it is not supposed to be routed and 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.

8.8.2 Neighbor Discovery

IPv6 Neighbor Discovery, or **ND**, is a protocol that replaces (and combines) several IPv4 tools, most notably ARP, ICMP redirects and most non-address-assignment parts of DHCP. The original specification for ND is in **RFC 2461**, later updated by **RFC 4861**. ND provides the following services:

- Finding a local host's LAN address, given its IPv6 address
- Finding the local router(s)
- Finding the set of network address prefixes that can be reached via local delivery (IPv6 allows there to be more than one)
- Determining that some neighbors are now unreachable
- Detecting duplicate IPv6 addresses

8.9 Router Advertisement

IPv6 routers periodically send **Router Advertisement** packets; these are also sent in response to **Router Solicitation** requests sent by hosts to the all-routers multicast group. Router Advertisement packets serve to identify the routers; this process is sometimes called **Router Discovery**. These RA packets also contain a list of all network address prefixes in use on the LAN. These packets may contain other important information about the LAN as well, such as an agreed-on MTU; much of the information included in RA packets is, in IPv4, sent via DHCP.

These IPv6 router messages represent a marked change from IPv4, in which routers need not send anything besides forwarded packets, and in which hosts typically discover their default routers via DHCP. IPv6 routers may even interact directly with ordinary hosts, if the SLAAC configuration mechanism (below) is used. To become an IPv4 router, a node need only enable IPv4 forwarding in the kernel. To become an IPv6 router, by comparison, in addition to forwarding a node "MUST" (**RFC 4294**) also run software to support Router Advertisement; the linux **radvd** package is one option here.

8.10 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 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 externally routable 64-bit prefix. 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 have a common prefix of length N<64. For multihomed sites the prefix list may contain multiple unrelated prefixes corresponding to the different address blocks. Finally, private and 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 *8.13 Stateless Autoconfiguration (SLAAC)* below.

8.11 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.

Hosts may also use the Authentication Header or the Encapsulated Security Payload Header to supply digital signatures for ND packets. If a node is statically configured to require such checks, then the IPv6 analogue of ARP spoofing (*7.7.2 ARP security*) can be prevented.

Finally, IPv4 ICMP Redirect messages have also been moved in IPv6 to the Neighbor Discovery protocol.

8.11.1 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 (*7.8 Dynamic Host Configuration Protocol (DHCP)*). In addition to DHCPv6, IPv6 also supports **StateLess Address AutoConfiguration**, or SLAAC. The original idea behind SLAAC was to support complete plug-and-play network setup: hosts on a completely 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.

8.12 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 *8.8.2 Neighbor Discovery*, asking if any other host has this address; if anyone answers, then the address is a duplicate. As with IPv4 ACD (*7.7.1 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.

8.13 Stateless Autoconfiguration (SLAAC)

To obtain an address via SLAAC, defined in **RFC 4862**, a host first generates its link-local address, as above in *8.3 Link-local addresses*, appending the standard 64-bit link-local prefix to its 64-bit host identifier (*8.2 Host identifier*) derived from the LAN address. 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 sends 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 (*8.10 Prefix Discovery*). 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, in the original version of SLAAC, appends the prefix to the front of its host identifier to create what should now be a routable address. The prefix length plus the host-identifier length *must* equal 128 bits; in the most common case each is 64 bits. The address so formed must now be verified through the duplicate-address-detection mechanism above.

An address generated in this way will, because of the embedded host identifier, uniquely identify the host for all time. This includes identifying the host even when it is connected to a new network and is given a different network prefix. Therefore, **RFC 4941** defines a set of **privacy extensions** to SLAAC: optional mechanisms for the generation of alternative host identifiers, based on pseudorandom generation using the original LAN-address-based host identifier as a "seed" value. The probability of two hosts accidentally choosing the same host identifier in this manner is very small; the Neighbor Solicitation mechanism with DAD must, however, still be used to verify that the address is in fact unique. DHCPv6 also provides an option for temporary address assignments, also 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 will make external tracking of a host slightly more difficult. However, for a residential "site" with only a handful of hosts, a considerable degree of tracking may be obtained simply by using the common 64-bit prefix.

In theory, if another host B on the LAN wishes to contact host A with a SLAAC-configured address containing the original host identifier, and B knows A's IPv6 address A_{IPv6}, then B might extract A's LAN address from the low-order bits of A_{IPv6}. This was never actually allowed, however, even before the **RFC 4941** privacy options, as there is no way for B to know that A's address was generated via SLAAC at all. B would always find A's LAN address through the usual process of IPv6 Neighbor Solicitation.

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 site-local address, not globally visible, then the host will likely want to use an address that is also site-local. **RFC 6724** also includes mechanisms

to allow a host with a permanent public address (*eg* corresponding to a DNS entry) to prefer alternative "temporary" or "privacy" addresses for outbound connections.

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 there is no obvious 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 the autoconfiguration process.

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. Hosts can also use **Dynamic DNS** (**RFC 2136**) to update their own DNS records.

8.14 DHCPv6

The job of the DHCPv6 server is to tell an inquiring host its network prefix(es) and also supply a 64-bit host-identifier. 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 DHCP request with the correct current subnet, and then to forward it to the actual DCHPv6 server. This allows the DHCP 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; use of the multicast group does not necessarily identify a requester's subnet. In fact, the All_DHCP_Relay_Agents_and_Servers multicast address scope is limited to the current link; relay agents then forward to the actual DHCP 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 DHCPv6 request, and 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. Another concern is that the DHCPv6 temporary-address sequence might have a flaw that would allow a remote observer to infer a relationship between different temporary addresses; with SLAAC, a host is responsible itself for the security of its temporary-address sequence and is not called upon to trust an external entity.

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; unlike for DHCPv4, there is no separate "address lease lifetime".

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 makes it very difficult for an outsider who knows one of a site's host addresses to guess the addresses of other hosts. Some DHCPv6 servers, however, do not yet support this; such servers make the SLAAC approach more attractive.

8.15 Manual Configuration

Like IPv4 addresses, IPv6 addresses can also be set up purely by manual configuration. In theory, this would be done only in the absence of an IPv6 router, in which case a unique-local prefix (*8.8.1 Network Prefixes*) would likely be appropriate. See *8.18 Manual address configuration* for one example. While it might be convenient to distribute only the /64 prefix via manual configuration, and have SLAAC supply the low-order 64 bits, this option is not described in the SLAAC RFCs and seems not to be available in common implementations.

8.15.1 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 (*1.14 Network Address Translation*). On an IPv6 network, on the other hand, *every* host is likely to 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 largely if not 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); one might need to configure the firewall with a list of inside addresses (or subnets) meant to be globally visible. Furthermore, given such a conventional firewall, it would then be 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 quite difficult. (It remains true, however, that as of this writing consumer-grade IPv6 firewalls of the type described here do not really exist.) 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 impractical as a /64 subnet has $2^{64} \simeq 18$ quintillion hosts.

As for nuisance, NAT has always broken protocols that involve negotiation of new connections (*eg* TFTP, or SIP, used by VoIP); IPv6 should make these much easier to manage.

8.15.2 ICMPv6

RFC 4443 defines an updated version of the ICMP protocol. It includes an IPv6 version of ICMP Echo Request / Echo Reply, upon which the "ping" command is based. It also handles 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 are special cases; 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 site-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 sent"; 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, *eg* an unrecognized Next Header value.

8.15.3 Routerless Connection Examples

Most IPv6 networks require at least one IPv6 router; both SLAAC and DHCPv6 configuration requires this. However, link-local addresses are quite serviceable, as long as one remembers that the interface must be specified; manually configured addresses are another option. Here are some examples. One practical problem with link-local addresses is that application documentation describing how to include a specification of the interface is sometimes sparse.

8.16 ping6

We will start with the linux version of `ping6`, the IPv6 analogue of the familiar `ping` command. It is used to send ICMPv6 Echo Requests. The ping6 command supports an option to specify the interface (`-I eth0`); as noted above, this is mandatory when sending to link-local addresses.

ping6 ::1: This allows me to ping my own loopback address.

ping6 -I eth0 ff02::1: This pings the all-nodes multicast group on interface `eth0`. I get these answers:

- 64 bytes from fe80::3e97:eff:fe2c:2beb (this is the host I am pinging *from*)
- 64 bytes from fe80::2a0:ccff:fe24:b0e4 (another linux host)

My VoIP phone – on the same subnet but apparently supporting IPv4 only – remains mute.

ping6 -I eth0 fe80::6267:20ff:fe72:8960: This pings the link-local address of the other linux host answering the previous query. Note the "ff:fe" in the host identifier. Also note the flipped seventh bit of the two bytes 02a0; the other linux host has Ethernet address 00:a0:cc:24:b0:e4.

8.17 TCP connections with link-local addresses

The next step is to create a TCP connection. Some commands, like ping6 above, may provide for a way of specifying the interface as an option. Failing that, many linux systems allow appending "*%interface*" to the address (which unfortunately usually is required to be in numeric form). The following, for example, allows me to connect using ssh to the other linux host above:

```
ssh fe80::2a0:ccff:fe24:b0e4%eth0
```

That the ssh service was 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 `ip.version == 6` may be useful).

If the connection fails, but ssh works for IPv4 connections and shows as listening in the tcp6 list from the netstat command, a blocked port is a likely suspect. In this case the commands `ufw` and `ip6tables --list` may be useful.

8.18 Manual address configuration

The use of manually configured addresses, *8.15 Manual Configuration*, is also possible; this avoids the need to specify an interface. The first step is to pick a unique-local prefix, *eg* fd37:dead:beef:cafe::0/64 (note that this particular prefix does *not* meet the randomness rules for unique-local address prefixes). Low-order bits can then be added and addresses assigned manually; on linux this is done with:

- host1: `ip -6 address add fd37:dead:beef:cafe::1 dev eth0`
- host2: `ip -6 address add fd37:dead:beef:cafe::2 dev eth0`

Now on host1 the command

```
ssh fd37:dead:beef:cafe::2
```

should work, again assuming `ssh` is listening for IPv6 connections. Because the address here is not link-local, /etc/host entries may also be created, *eg* the following on host1:

```
fd37:dead:beef:cafe::2 host2-6
```

Now to connect all that should be needed is

```
ssh host2-6
```

8.19 Node Information Messages

In addition to ICMPv6, IPv6 also has 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 DNS reverse-name lookups. Other NI queries allow a host to be asked for its other IPv6 addresses, or for its IPv4 addresses.

8.19.1 IPv6-to-IPv4 connectivity

What happens if you switch to IPv6 completely, perhaps because your ISP has run out of IPv4 addresses? Ideally you will only need to talk to IPv6 servers. For example, the DNS name `microsoft.com` corresponds to an IPv4 address, but also to an IPv6 address. If there is not IPv6 connectivity between you and the IPv6 server you are trying to reach, **tunneling** of IPv6 traffic over IPv4 can be used.

But what do you do if you have only an IPv6 address and want to reach an IPv4-only server? IPv4 and IPv6 cannot directly interoperate; while IPv6 could have been modified to support an interoperability feature, a change of such magnitude in IPv4 is politically and economically impossible. Thus, you will have to find some sort of **translator**, such as an IPv6-to-IPv4 NAT router. **RFC 2766** defines an IPv6-to-IPv4 flavor of NAT that also includes any necessary translation of the higher transport layer as well. The NAT translator *will* have an IPv4 address (otherwise it cannot talk to other IPv4 nodes), but that address can be shared among multiple IPv6 clients.

8.19.2 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 do not really exist. Time will tell how all this evolves. However, while IPv6 has now been around for twenty years, top-level IPv4 address blocks disappeared just three years ago. It is quite possible that this will prove to be just the catalyst IPv6 needs.

8.19.3 Exercises

1. Each IPv6 address is associated with a specific solicited-node multicast address. 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. 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. 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. Create an IPv6 ssh connection as in *8.15.3 Routerless Connection Examples*. 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. Create an IPv6 ssh connection using manually configured addresses as in *8.18 Manual address configuration*. Again use WireShark or the equivalent to monitor the connection. Is DAD (*8.12 Duplicate Address Detection*) used?

9 ROUTING-UPDATE ALGORITHMS

How do IP routers build and maintain their forwarding tables?

Ethernet bridges can always fall back on broadcast, 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**. Both 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). 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.

In the following chapter we will look at the Border Gateway Protocol, or BGP, in which no link-cost calculations are made, because links may be between unrelated organizations which have no agreement in place on determining link cost.

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. 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 *18.7 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. On unix-based systems the process in charge of this is often called "routed" (pronounced route-d).

Routers identify their router neighbors (through some sort of neighbor-discovery mechanism), and add a third column to their forwarding tables for **cost**; table entries are thus of the form ⟨destination,next_hop,cost⟩. The simplest case is to assign a cost of 1 to each link (the "hopcount" metric); it is also possible to assign more complex numbers.

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.

Actual destinations in IP would be *networks* 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 networks with the router itself. That is, we will build forwarding tables to reach every *router*. While it is possible that one destination network might be reachable by two or more routers, thus breaking our identification of a router with its set of attached networks, in practice this is of little concern.

9.1.1 Distance-Vector Update Rules

Let A be a router receiving a report $\langle D,c_D \rangle$ from neighbor N at distance 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 $\langle D,N,c_D+c_N \rangle$
2. **Lower cost**: D is a known destination with entry $\langle D,M,c \rangle$, but $c > c_D+c_N$. A updates the entry for D to $\langle D,N,c_D+c_N \rangle$. It is possible that M=N, meaning that N is now reporting a distance decrease to D.
3. **Next_hop increase**: A has a previous entry $\langle D,\mathbf{N},c \rangle$, but $c < c_D+c_N$. A updates the entry for D to $\langle D,N,c_D+c_N \rangle$. N is A's next_hop to D, and N is now reporting a distance increase.

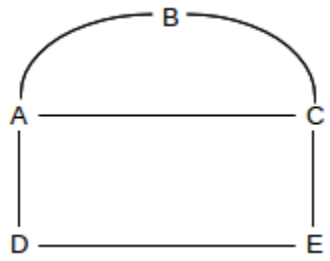
The first two rules are for new destinations and a shorter path to existing destinations. In these cases, the cost to each destination monotonically decreases (at least if we consider all unreachable destinations as being at distance ∞). Convergence is automatic, as the costs cannot decrease forever.

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**⟩

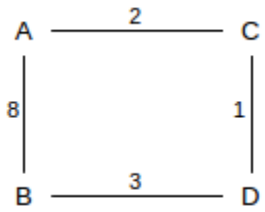
In this case we have 100% reachability but B routes to D via the longer-than-necessary path B–C–E–D. However, one more report will fix this: suppose A reports to B. B will received ⟨D,1⟩ from A, and will update its entry ⟨D,C,3⟩ to ⟨D,A,2⟩.

Note that A routes to E via D while E routes to A via C; this asymmetry was due to indeterminateness in the selection 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⟩

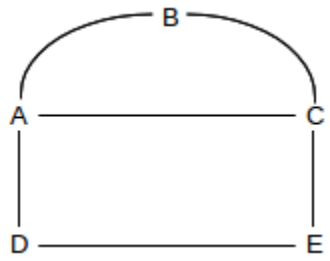
After A and C exchange, A has ⟨D,C,3⟩ and C has ⟨B,A,10⟩. After C and D exchange, C updates its ⟨B,A,10⟩ entry to ⟨B,D,4⟩ and D adds ⟨A,C,3⟩; D receives C's report of ⟨B,10⟩ but ignores it. Now finally suppose B and D exchange. D ignores B's route to A, as it has a better one. B, however, gets D's report ⟨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.

Initially, upon discovering the break, D and E update their tables to $\langle$E,-,$\infty\rangle$ and $\langle$D,-,$\infty\rangle$ 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 $\langle$E,-,$\infty\rangle$. 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 $\langle$D,2$\rangle$ to E, E will add the entry $\langle$D,C,3$\rangle$, and will again be able to reach D. When C reports to A, A will add the route $\langle$E,C,2$\rangle$. The final step will be when A next reports to D, and D will have $\langle$E,A,3$\rangle$. 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	cost	next_hop
A	3	S
B	4	T
C	5	S
D	6	U

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

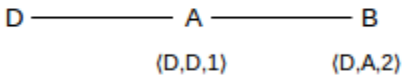
R then updates its table as follows:

destination	cost	next_hop	reason
A	3	S	No change; S probably sent this report before
B	4	T	No change; R's cost via S is tied with R's cost via T
C	6	S	Next_hop increase
D	5	S	Lower-cost route via S
E	3	S	New destination

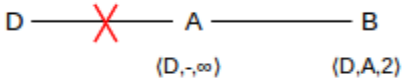
Whatever S's cost to a destination, R's cost to that destination via S is one greater.

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 (distance = ∞), 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 $\langle$D,B,3$\rangle$. 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 $\langle$D,3$\rangle$ to B, at which point B will update its entry to $\langle$D,A,4$\rangle$. Then B will report $\langle$D,4$\rangle$ to A, and A's entry will be $\langle$D,B,5$\rangle$, 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

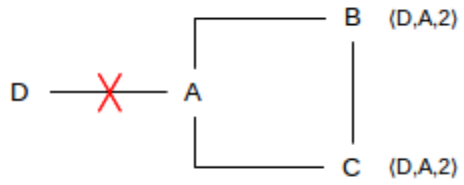
The simplest fix to this problem is to use a small value for infinity. Most flavors of the RIP protocol use infinity=16, with updates every 30 seconds. The drawback to so small an infinity is that no path through the network can be longer than this; this makes paths with weighted link costs difficult. 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 $\langle D,-,\infty \rangle$. A then reports $\langle D,\infty \rangle$ to B, which updates its table to $\langle D,-,\infty \rangle$. But then, before A can also report $\langle D,\infty \rangle$ to C, C reports $\langle D,2 \rangle$ to B. B then updates to $\langle D,C,3 \rangle$, and reports $\langle D,3 \rangle$ back to A; neither this nor the previous report violates split-horizon. Now A's entry is $\langle D,B,4 \rangle$. Eventually A will report to C, at which point C's entry becomes $\langle D,A,5 \rangle$, 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 $\langle D,\infty \rangle$ 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.

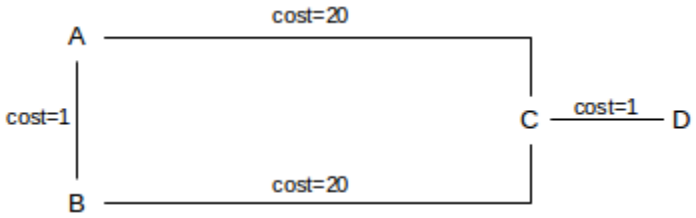
9.3 Observations on Minimizing Route Cost

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 ⟨D,B,c⟩, 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.

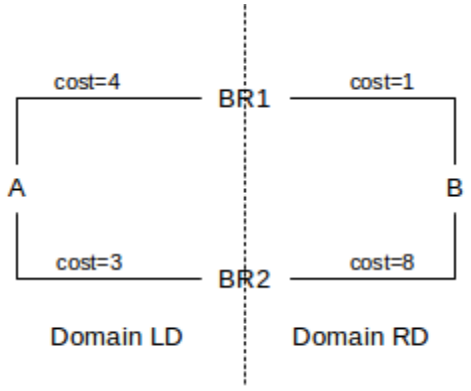
As we apply distance-vector routing, each router independently builds its tables. A router might have some notion of the path its packets would take to their destination; for example, in the case above A might believe that with forwarding entry ⟨D,B,22⟩ its packets would take the path A–B–C–D (though 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 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 (*3.3.8 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 distance ∞ 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.

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 $c(D,M) \geq C$. 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.

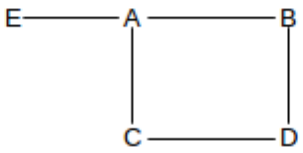
Link-state protocols distribute network map information through a modified form of broadcast of the status of each individual link. Whenever either side of a link notices the link has died (or if a node notices that a new link has become available), it sends out **link-state packets** (LSPs) that "flood" the network. This broadcast process is called **reliable flooding**; in general broadcast protocols work poorly with networks that have even small amounts of topological looping (that is, redundant paths). Link-state protocols must ensure

that 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:



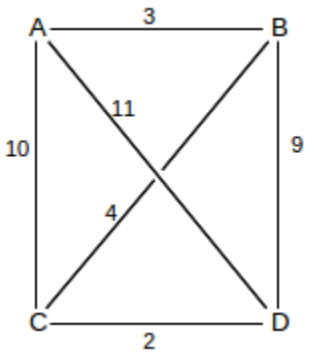
Suppose the A–E link status changes. A sends LSPs to C and B. Both these will forward the LSPs to D; suppose B's arrives first. Then D will forward the LSP to C; the LSP traveling C→D and the LSP traveling D→C might even cross on the wire. D will ignore the second LSP copy that it receives from C and 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. Here is our example network:



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**.

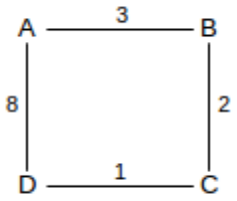
For the example above, we start with **current** = A and **R** = {⟨A,A,0⟩}. The set **T** will be {⟨B,B,3⟩, ⟨C,C,10⟩, ⟨D,D,11⟩}. The shortest entry is ⟨B,B,3⟩, so we move that to **R** and continue with **current** = B.

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 attributes other than the destination, or the destination and QoS bits. This kind of routing is decidedly nonstandard, and is generally limited to the imposition of very local routing policies. It is, however, often available.

On linux systems this 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, where L1 is faster and L2 serves more as a backup; in such a configuration the default route to the Internet will often be via L1. A site may wish to route some outbound traffic via L2, however, 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

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.

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 --source-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 IP precedence bits, so that a single standard IP forwarding table can be used, though support for the IP precedence bits is limited.

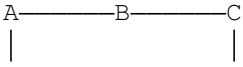
9.7 Epilog

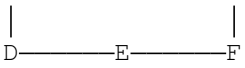
At this point we have concluded the basics of IP routing, involving routing within large (relatively) *homogeneous* organizations such as multi-site corporations or Internet Service Providers. Every router involved must agree to run the same protocol, and must agree to a uniform assignment of link costs.

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

1. 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⟩).



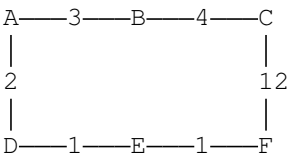


(a). Suppose each node creates a report from its initial configuration and sends that to each of its neighbors. What will each node's forwarding table be after this set of exchanges? The exchanges, in other words, are all conducted simultaneously and in parallel; no report contains new information learned by a router as part of this process.

(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. 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⟩?

3. Suppose 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*.

4. Suppose nodes A, B, C, D, E and F form a network in which all links have cost 1. The forwarding tables for A and F are as follows:

A's table

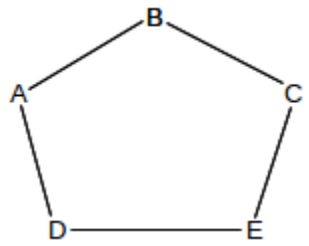
destination	cost	next hop
B	1	B
C	1	C
D	2	B
E	2	C
F	3	B

F's table

destination	cost	next hop
A	3	D
B	2	D
C	2	E
D	1	D
E	1	E

These tables imply that A is directly connected only to B and C, and that F is directly connected only to D and E, because these are the only destinations reachable with a cost of 1. Give all other direct connections that must exist, and draw the network.

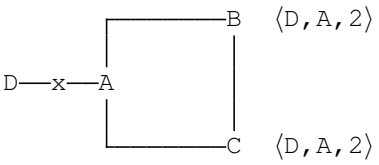
5. Suppose A, B, C, D and E are connected as follows. Each link has cost 1, and so each forwarding table is uniquely determined; B's table is ⟨A,A,1⟩, ⟨C,C,1⟩, ⟨D,A,2⟩, ⟨E,C,2⟩. Distance-vector routing update is used.



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.

6. Consider the network in *9.2.1.1 Split Horizon*:, using distance-vector routing updates.

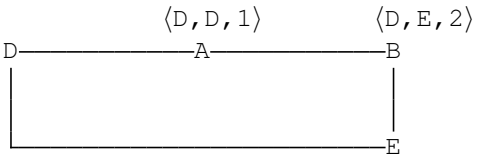


Suppose the D–A link breaks and then

- 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). What report will lead to the formation of the routing loop?
(b). What report will eliminate the possibility of the routing loop?

7. Suppose the network of *9.2 Distance-Vector Slow-Convergence Problem* is changed to the following. Distance-vector update is used; again, the A–D 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. 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

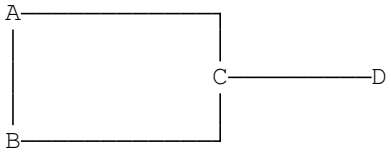
dest	cost	next_hop
B	1	B
C	1	C
D	2	C
E	2	C
F	3	B

F's table

dest	cost	next_hop
A	3	E
B	2	D
C	2	D
D	1	D
E	1	E

9. (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, 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 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).

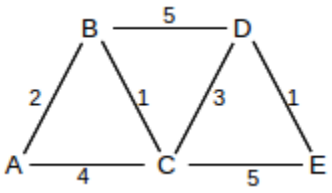
10. Suppose the distance-vector algorithm is run on a network and no links break (so by the last paragraph of *9.1.1 Distance-Vector Update Rules* the next_hop-increase rule is never applied).

(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. Give a scenario illustrating how a (very temporary!) routing loop might form in link-state routing.

12. 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. 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?

10 LARGE-SCALE IP ROUTING

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 not just to a common protocol, but also 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 the 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 path cost comparisons between the two sites simply cannot be done. In general, we cannot even "translate" costs from one site to another, because the paths themselves depend on the cost assignment strategy.

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 /27 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.

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.

The basic strategy of CIDR is to consolidate multiple networks 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* FF.FF.FC.00 (in hex) rather than the number 22; note 0xFC = 1111 1100 with 6 leading 1-bits, so FF.FF.FC.00 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].

It is possible to have multiple matches, and responsibility for avoiding this 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 three possible applications; the latter two involve hierarchical routing:

- Application 1 (legacy): CIDR allows IANA to allocate 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
- 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.

Hierarchical routing does introduce one new wrinkle: the routes chosen may no longer be globally optimal. Suppose, for example, we are trying to route to prefix 200.20.0.0/16. Suppose at the top level that traffic to 200.0.0.0/8 is routed to router R1, and R1 then routes traffic to 200.20.0.0/16 to R2. A packet sent to 200.20.1.2 by an independent router R3 would always pass through R1, *even if there were a shorter path R3→R4→R2 that bypassed R1*. This is addressed in further detail below in *10.4.3 Provider-Based Hierarchical Routing*.

We can avoid this by declaring that CIDR will only be used when it turns out that, in the language of the example of the preceding paragraph, the best path to reach 200.20.0.0/16 *is* always through R1. But this is seldom the case, and is even less likely to *remain* the case as new links are added. Such a policy also defeats much of the potential benefit of CIDR at reducing router forwarding-table size by supporting the creation of arbitrary-sized address blocks and then *routing to them as a single unit*. The Internet backbone might

be much happier if all routers simply had to maintain a single entry ⟨200.0.0.0/8, R1⟩, versus 256 entries ⟨200.x.0.0/16, R1⟩ for all but one value of x.

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

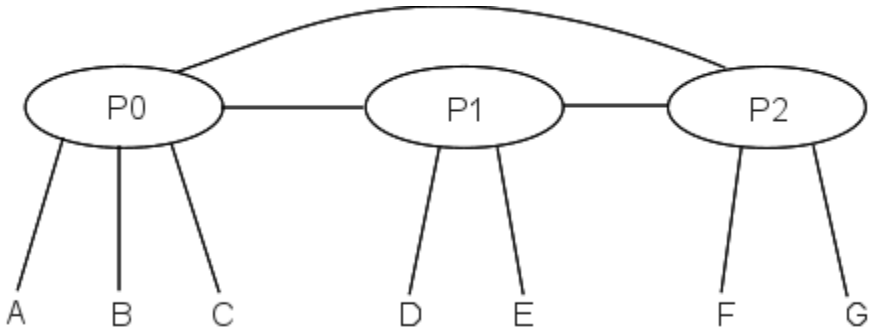
Let us now allocate addresses to the customers:

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:

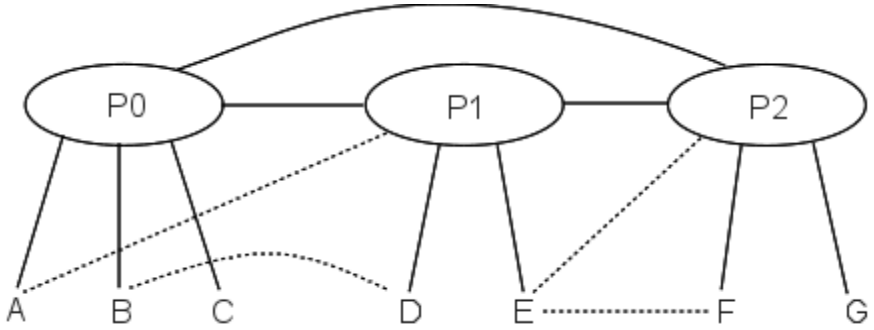
destination	next_hop
200.0.0.0/16	A
200.1.0.0/16	B
200.2.16.0/20	C
201.0.0.0/8	P1
202.0.0.0/8	P2

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.

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 are **multi-homed**; that is, they have connections to alternative providers: A–P1 and E–P2. Typically, though, while A and E may use these secondary 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.

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 is going 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

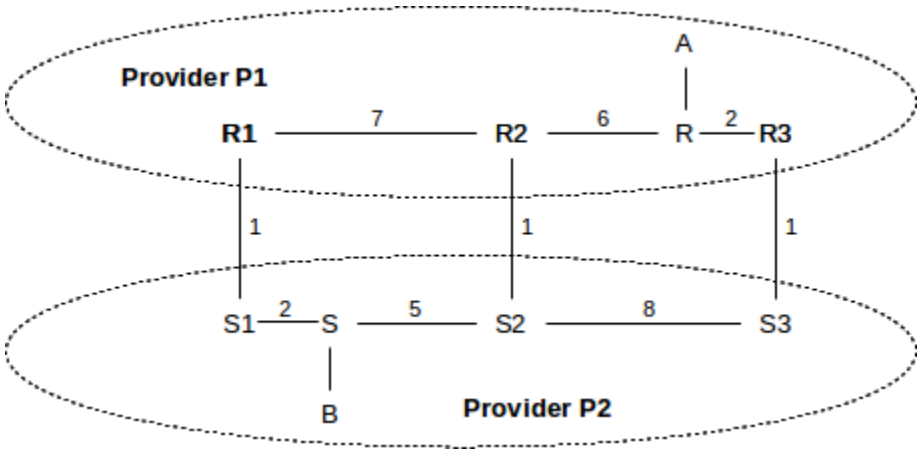
- A can send outbound traffic via P1
- P1 forwards A's traffic to A via the private link

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 Provider-Based Hierarchical Routing

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, and then from that provider entry point to B; the second strategy may not yield the same result. This second strategy mirrors the two-step routing process of first routing on the address bits that identify the provider, and then routing on the address bits including the subscriber portion.

Consider the following example, in which providers P1 and P2 have three interconnection links (r1–s1, r2–s2 and r3–s3), each with cost 1. 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–s2 crossover, with total length 5+1+5=11. However, traffic from A to B will be routed by P1 to its closest crossover to P2, namely the r3–s3 link. The total path is 2+1+8+5=16. Traffic from B to A will be routed by P2 via the r1–s1 crossover, for a length of 2+1+7+6=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 of little consequence.

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 picture 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. 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.

But when systems under different administration (*eg* two large ISPs) talk to each other, comparing metrics simply does not work. Each side's metrics may be based on any of the following:

- hopcount
- bandwidth
- cost
- congestion

One provider's metric may even use larger numbers for better routes (though if this is done then total path costs, or preference, cannot be obtained simply by adding the per-link values). Any attempt at comparison of values from the different sides is a comparison of apples and oranges.

The **Border Gateway Protocol**, or BGP, is assigned the job of handling exchange of routing information between neighboring independent organizations; this is sometimes called **exterior** routing. The current version is BGP-4, documented in **RFC 4271**. The BGP term for a routing domain under coordinated administration, and using one consistent 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.

BGP also has a second goal, in addition to the purely technical problem of finding routes in the absence of cost information: BGP also provides support for **policy-based routing**; that is, for making routing decisions based on *managerial* or *administrative* input (perhaps regarding who is paying what for the traffic carried).

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.

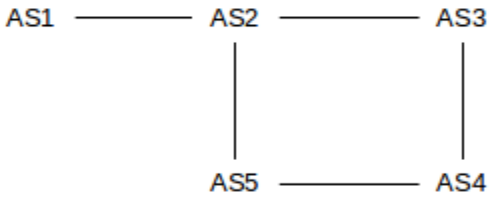
The BGP speakers must maintain a database of all routes received, not just of the routes actually used. However, the speakers exchange with neighbors only the routes they (and thus their AS) use themselves; this is a firm BGP rule.

The current BGP standard is **RFC 4271**.

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 avoid 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 too 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 will include the entire AS-path ⟨AS5,AS4,AS3,AS2,AS1⟩, and AS2 would know not to use this route because it would 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 some degree of **administrative** input to what, for interior routing,

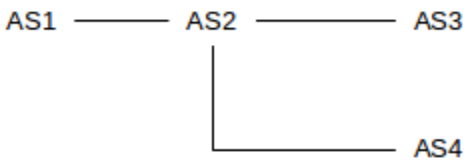
is largely a technical calculation (though an interior-routing administrator can set link costs). BGP is the interface between large ISPs, and can be used to implement *contractual* agreements made regarding which ISPs will carry other ISPs' traffic. If ISP2 tells ISP1 it has a good route to destination D, but ISP1 chooses not to send traffic to ISP2, BGP can be used to implement this.

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 policy-based routing.

10.6.4 BGP Filtering and Policy Routing

As stated above, one of the goals of BGP is to support **policy routing**; 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. Finally, while it is not commonly useful, import filtering can involve rather strange criteria; for example, in *10.6.8 Examples of BGP Instability* we will consider examples where AS1 prefers routes with AS-path ⟨AS3,AS2⟩ to the strictly shorter path ⟨AS2⟩.

The next stage is best-path selection, for which 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 that go through their customers, as these are "cheaper". A smaller ISP that connects to two or more larger ones might be paying to route almost all 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. Some choices may lead to instability, below, so domains are encouraged to set their rules in accordance with some standard principles, also below.

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.

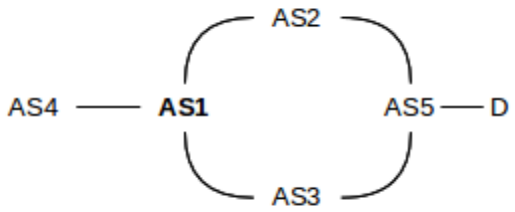
Local_preference values are communicated internally via the LOCAL_PREF path attribute, below. They are not shared with other Autonomous Systems.

The final significant step of the route-selection phase is to apply the Multi_Exit_Discriminator value; we postpone this until below. A site may very well choose to ignore this value entirely. There may then be additional trivial tie-breaker rules; note that if a tie-breaker rule assigns significant traffic to one AS over another, then it may have significant economic consequences and shouldn't be considered "trivial". If this situation is detected, it would probably be addressed in the local-preferences phase.

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 AS1 switches to reaching D via AS3, but A is *forbidden by policy* to announce AS3-paths to AS4. Then A 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.

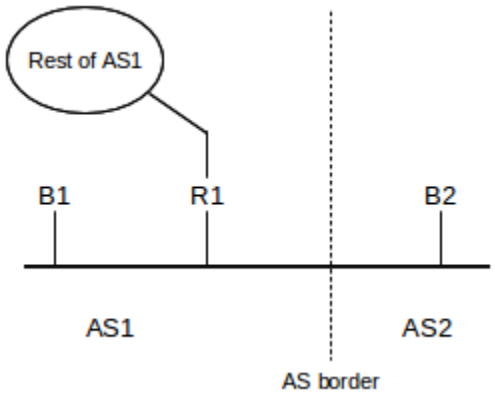
10.6.5 BGP Path attributes

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.

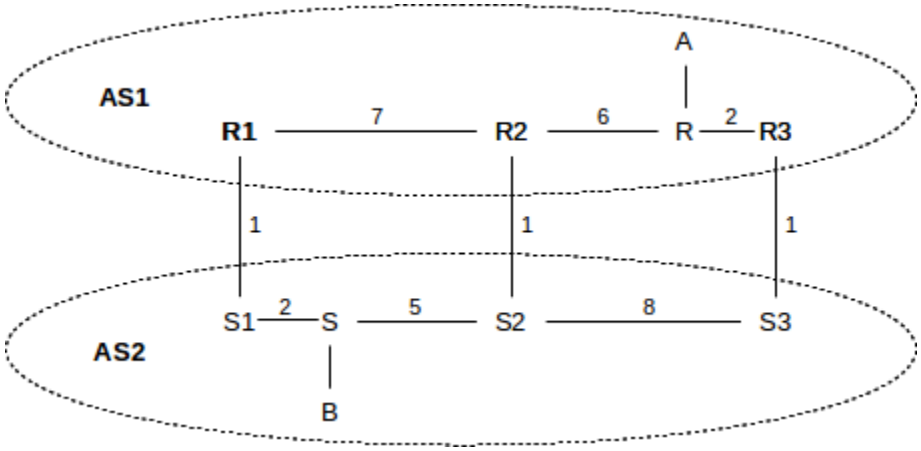
10.6.5.3 MULTI_EXIT_DISC

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 Provider-Based Hierarchical Routing*, with providers now replaced by Autonomous Systems); the numeric values on links are their relative costs. We will assume that border routers R1, R2 and R3 are also AS1's BGP speakers.



In the absence of the MED, AS1 will send traffic from A to B via the R3–S3 link, and AS2 will return the traffic via S1–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 R1, R2 and R3 can provide MED values to AS2 when advertising destination A, indicating a preference for AS2→AS1 traffic to use the rightmost link:

- R1: destination A has MED 200
- R2: destination A has MED 150

- R3: 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 the S3–R3 link; that is, via the link 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 R3–S3 link becomes disabled, in which case the numeric MED values above would mean that AS2 should then prefer the R2–S2 link for reaching A.

There is no way to use MED values to cause A–B traffic to take the R2–S2 link; that link is the minimal-cost link 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 cross-links, they are used only in ranking precedence, not as numeric measures of cost (though they are sometimes derived from that).

As a hypothetical example of why a site might use the MED, suppose site B in the diagram above wants its customers to experience high-performance downloads. It contracts with AS2, which advertises superior quality in the downloads experienced by users connecting to servers in its domain. In order to achieve this superior quality, it builds a particularly robust network S1–S–S2–S3. It then agrees to accept MED information from other providers so that it can keep *outbound* traffic in its own network as long as possible, instead of handing it off to other networks of lower quality. AS2 would thus want B→A traffic to travel via S3 to maximize the portion of the path under its own control.

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
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.

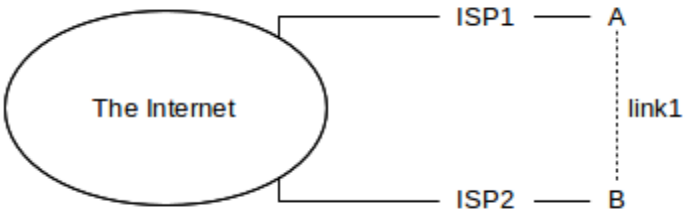
Here are some of the community values once supported by a no-longer-extant ISP that we shall call AS1. The full community value would have included AS1's AS-number.

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 Policy and Transit Traffic

Perhaps the most common source of simple policy decisions is whether a site wants to accept **transit traffic**.

As a first example, let us consider the case of configuring a private link, such as the dashed link1 below between "friendly" but unaffiliated sites A and B:



If A and B fully advertise link1, by exporting to their respective ISPs routes to each other, then ISP1 (paid by A) may end up carrying much of B's traffic or ISP2 (paid by B) may end up carrying much of A's traffic. Economically, these options are not desirable unless fully agreed to by both parties. 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.

Three common options A and B might agree to regarding link1 are **no-transit**, **backup**, and **load-balancing**.

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

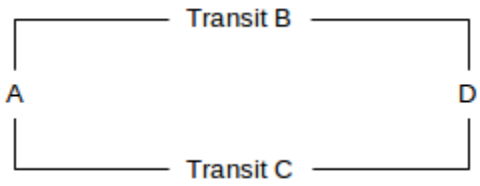
and begin advertising it to the Internet. The link1 route will be the primary route to B until ISP2's service is restored.

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.

For *outbound* traffic, A and B will need a way to send through one another if their own ISP link is down. One approach is to consider their default-route path (*eg* to 0.0.0.0/0) to be a concrete destination within BGP. ISP1 advertises this to A, using A's interior routing protocol, but so does B, and A has configured things so B's route 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.

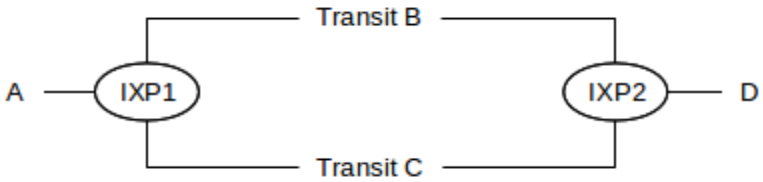
For inbound **load balancing**, there is no easy fix, in that if ISP1 and ISP2 both export routes to A, then A has lost all control over how other sites will prefer one to the other. A may be able to make one path artificially appear more expensive, and keep tweaking this cost until the inbound loads are comparable. Outbound load-balancing is up to A and B.

Another basic policy question is which of the two available paths site (or regional AS) A uses to reach site D, in the following diagram. B and C are Autonomous Systems.



How can A express preference for B over C, assuming B and C both advertise to A their routes to D? Generally A will use a local_preference setting to make the carrier decision for A⟶D traffic, though it is D that makes the decision for the D⟶A traffic. It is possible (though not customary) for one of the transit providers to advertise to A that it can reach D, but not advertise to D that it can reach A.

Here is a similar diagram, showing two transit-providing Autonomous Systems B and C connecting at Internet exchange points IXP1 and IXP2.



B and C each have routers within each IXP. B would probably 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 can avoid this problem by not advertising to C that it can reach A and D. In general, transit providers are often quite careful about advertising reachability to any other AS for whom they do not intend to provide transit service, because to do so may implicitly mean getting stuck with that traffic.

If B and C were both to try try to get away with this, a routing loop would be created within IXP1! But in that case in B's next advertisement to C at IXP1, B would state that it reaches D via AS-path ⟨C⟩ (or ⟨C,D⟩

if D were a full-fledged AS), and C would do similarly; the loop would not continue for long.

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. However, most AS policies are determined by a few simple AS business relationships, as identified in [LG01].

> **Customer-Provider**: A provider contracts to transit traffic from the customer to the rest of the Internet. A customer may have multiple providers, for backup or for more efficient routing, but a customer **does not** accept transit traffic between the providers.
>
> **Siblings**: Siblings are ISPs that have a connection between them that they intend as a backup route or for internal traffic. Two siblings may or may not have the same upstream ISP provider as parent. The siblings **do not** use their connection for transit traffic except when one of their primary links is down.
>
> **Peers**: Peers are two providers that agree to exchange all their customer traffic with each other; thus, peers **do** use their connection for transit traffic. Generally 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 there would likely be no exchange of cash, but if the volume flow is significantly asymmetric then compensation can certainly be negotiated.

These business relationships can be described in terms of – or even inferred from – what routes are accepted [LG01]. Every AS has, first of all, its **customer** routes, that is, the routes of its direct customers, and its customers' customers, etc. These might be referred to as the ISP's *own* routes. Other routes are learned from peers or, sometimes, providers. Essentially every AS exports its customer routes to all its AS neighbors; the issue is with provider routes and peer routes. Here are the rules of [LG01]:

- Customers **do not** export to their providers routes they have learned from peers or other providers. This would make them into transit providers, which is not their job.
- Providers **do** export provider/peer routes to their customers (though a provider may just provide a single consolidated default route to a single-homed customer).
- Peers **do not** export peer or provider routes to other peers; that is, if ISP P1 has peers P2 and P3, and P1 learns of a route to destination D through P2, then P1 does not export that route to P3. Instead, P3 would be expected to have its own relationship with P2, or be a customer of a provider that had a relationship with P2. Peers do not (usually) provide transit services to third parties.
- Siblings **do** export provider/peer routes to each other, though likely with an understanding about preference values. If S1 and S2 are two siblings with a mutual-backup relationship, and S1 loses its normal connectivity and must rely on the S1–S2 link, then S2 must know all about S1's routes.
- All AS's **do** export their own routes, and their customers' routes, to customers, siblings, peers and providers.

There may be more 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.

Following these rules creates a simplified BGP world. Special cases for special situations have the potential to introduce non-convergence or instability.

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.

A consequence of the use of the above classification and attendant export rules is the **no-valley theorem** [LG01]: if every AS has BGP policies consistent with the scheme above, then when we consider the full AS-path from A to B, there is at most one peer-peer link. Those to the left of the peer-peer link are (moving from left to right) either customer⟶provider links or sibling⟶sibling links; that is, they are non-downwards (*ie* upwards or level). To the right of the peer-peer link, we see provider⟶customer or sibling⟶sibling links; that is, these are non-upwards. If there is no peer-peer link, then we can still divide the AS-path into a non-downwards first part and a non-upwards second part.

The above constraints 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, unstable BGP arrangements appear rare on the Internet; most actual relationships and configurations are consistent with the rules above.

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:

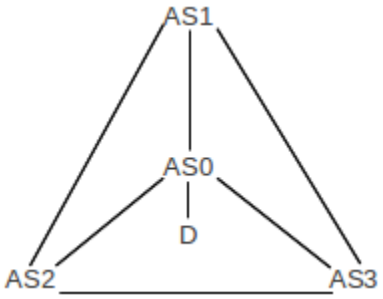
⟨AS2,AS0⟩, ⟨AS0⟩

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.

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 adopt ⟨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.

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.

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

1. Consider the following IP forwarding table that uses CIDR. IP address bytes are in **hexadecimal**, so each hex digit corresponds to four address bits.

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, indicate to what destination it is 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. Consider the following IP forwarding table, using CIDR. As in exercise 1, IP address bytes are in **hexadecimal**.

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.

3. Give an IPv4 forwarding table – using CIDR – that will route all Class A addresses to next_hop A, all Class B addresses to next_hop B, and all Class C addresses to next_hop C.

4. 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
37.119.0000 0000.0/18	A
37.119.0100 0000.0/18	A
37.119.1000 0000.0/18	A
37.119.1100 0000.0/18	B

These four entries cannot be consolidated into a single /16 entry, because they don't all go to the same next_hop. How could they be consolidated into *two* entries?

5. 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

A: 51.10.0.0/16
B: 51.23.0.0/16

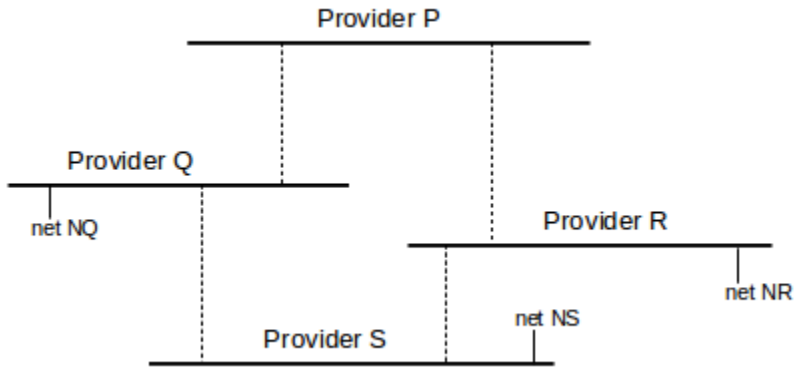
Q has customers C and D and assigns them address blocks as follows:

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 forwarding tables for P, Q and R; the longest-match rule will be needed to resolve conflicts.
(c). Now suppose *in addition* to A switching from P to Q, C switches from provider Q to provider R. Give the forwarding tables.

6. Suppose P, Q and R are ISPs as in the previous problem. P and R do not connect directly; they route traffic to one another via Q. In addition, customer B is multi-homed and has a secondary connection to provider R; customer D is also multi-homed and has a secondary connection to provider P; these secondary connections are not advertised to other providers however. Give forwarding tables for P, Q and R.

7. Consider the following network of providers P-S, all using BGP. The providers are the horizontal lines; each provider is its own AS.

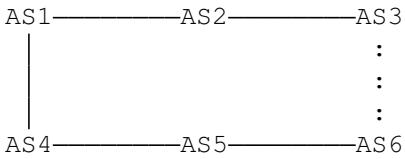


(a). What routes to network NS will P receive, assuming there is no export filtering? For each route, list the AS-path.

(b). What routes to network NQ will P receive? For each route, list the AS-path.

(c). Suppose R uses export filtering so as not to advertise to P any of its routes except those that involve S in their AS-path. What routes to network NR will P receive, with AS-paths?

8. 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 ⟨AS1⟩.



(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 AS2? 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. 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.

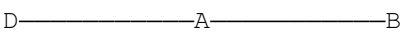
(a). Assuming all Megacorp traffic must enter and exit in Chicago, what is the route of traffic to and from the San Diego office to a client also in San Diego?

(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. What must it do to ensure that its internal traffic is still routed over its own network?

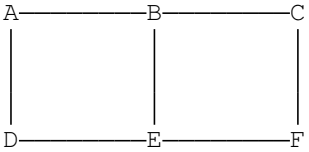
10. Suppose we try to use BGP's strategy of exchanging destinations plus paths as an interior routing-update strategy, perhaps replacing distance-vector routing. No costs or hop-counts are used, but routers attach to each destination a list of the routers used to reach that destination. Routers can also 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 distance-vector with the hopcount metric, if routers are not allowed to have preferences and if the router-path length is used as a tie-breaker.

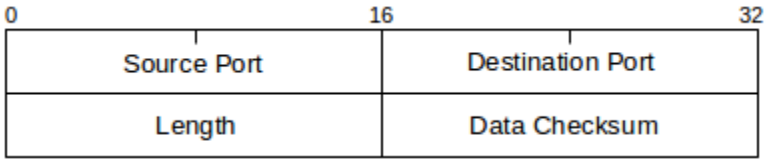
11 UDP TRANSPORT

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** on the data. 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.

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 *11.7 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 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 works much better for voice and video, where small losses simply degrade the received signal slightly, but where

larger delays are intolerable. This 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.

Sometimes UDP is used simply because it allows new or experimental protocols to run entirely as user-space applications; no kernel updates are required. Google has created a protocol named QUIC (Quick UDP Internet Connections) that appears to be in this category, though QUIC also takes advantage of UDP's freedom from head-of-line blocking. QUIC's goals include supporting multiplexed streams in a single connection (*eg* for the multiple components of a web page), and for eliminating the RTT needed for setting up a TCP connection. Google can achieve widespread web utilization of QUIC simply by distributing the client side in its Chrome browser; no new operating-system support (as would be required for adding a TCP mechanism) is then needed.

Finally, UDP is well-suited for "request-reply" semantics; 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, presumably 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 packets use the 16-bit Internet checksum (*5.4 Error Detection*) on the data. While it is seldom done now, the checksum can be disabled and the field set to the all-0-bits value, which never occurs as an actual ones-complement sum.

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 typically include some form of **flow control** to prevent this; receiver-side ACKs often are pressed into service for this role too.

11.1.1 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. "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⟩

11.1.1.1 The Server

We will start with the server side, presented here in Java. We will use port 5432; 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. We also need a `DatagramPacket` object that contains the packet data and source ⟨IP_address,port⟩ for arriving packets. The server application does not acknowledge anything sent to it, or in fact send any response at all.

The server application needs no parameters; it just starts. (That said, we could make the port number a parameter, to allow easy change. The port we use here, 5432, has also been adopted by PostgreSQL for TCP connections.) 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 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
}
```

11.1.1.2 UDP and IP addresses

The line `s = new DatagramSocket(destport)` creates a `DatagramSocket` object **bound** to the given port. If a host has multiple IP addresses, 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 standard socket library does not provide a way to find out to which these an arriving UDP packet was actually sent. 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.

11.1.1.3 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.

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 connect to an stalk server on the same host using IPv6 (and the loopback interface). If greater IPv4-versus-IPv6 control is desired, one can to replace the `getByName()` call with `getAllByName()`, which returns an array of all addresses (`InetAddress[]`) associated with the given name. One can then find the IPv6 addresses by searching this array for addresses `addr` for which `addr instanceof Inet6Address`.

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:
            buf = input("> ")
            if len(buf) == 0: return           # an empty line exits
            s.sendto(bytes(buf + "\n", 'ascii'), addr)

talk()
```

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

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.

11.2 Fundamental Transport Issues

As we turn to actual transport protocols, including eventually TCP, we will encounter the following standard problematic cases that must be addressed if the integrity of the data is to be ensured.

Old duplicate packets: These packets can be either **internal** – from an earlier point in the *same* connection instance – or **external** – from a *previous* instance of the connection. For the internal case, the receiver must make sure that it does not accept an earlier duplicated and delayed packet as current data (if the earlier packet was not duplicated, in most cases the transfer would not have advanced). Usually internal old duplicates are prevented by numbering the data, either by block or by byte. However, if the numbering field is allowed to wrap around, an old and a new packet may have the same number.

For the external case, the connection is closed and then reopened a short time later, using the same port numbers. (A connection is typically defined by its endpoint socket addresses; thus, 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 separate **incarnations** of the connection.) Somehow a delayed copy of a packet from the first instance (or incarnation) of the connection arrives while the second instance is in progress. This old duplicate must not be accepted, incorrectly, as valid data, as that would corrupt the second transfer.

Both these 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 *12.11 TCP Faster Opening*.

Lost final ACK: Most packets will be acknowledged. The final packet (typically but not necessarily an ACK) 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?

Duplicated connection request: How do we distinguish between two different connection requests and a single request that was retransmitted? Does it matter?

Reboots: What if one side reboots while the other side is still sending data? How will the other side detect this? Are there any scenarios that could lead to corrupted data?

11.3 Trivial File Transport Protocol, TFTP

As an actual protocol based on UDP, we consider the Trivial File Transport Protocol, TFTP. While TFTP supports clients sending files to the server, we will restrict attention to the more common case where the client requests a file from the server.

Although TFTP is a very simple protocol, it addresses all the fundamental transport issues listed above, to at least some degree.

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 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.

Because TFTP uses UDP it must take care of packetization itself, and thus must fix a block size small enough to be transmitted successfully everywhere.

In the absence of packet loss or other errors, TFTP file requests proceed as follows.

1. The client sends a RRQ to server port 69, from client port c_port

2. The server obtains a new port, s_port, from the OS
3. The server sends Data[1] from s_port
4. The client receives Data[1], and thus learns the value of s_port
5. The client sends ACK[1] (and all future ACKs) to the server's s_port
6. The server 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
8. An optional but recommended step of server-side **dallying** is used to address the lost-final-ACK issue

11.3.1 Port Changes

In the above, the server changes to a new port s_port when answering. While this change plays a modest role in the reliability of the protocol, below, it also makes the implementer's life much easier. When the server creates the new port, it is assured that the only packets that will arrive at that port are those related to the original client request; other client requests will be associated with other server ports. The server can create a new process for this new port, and that process will be concerned with only a single transfer even if multiple parallel transfers are taking place.

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 I know 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. Does this represent noncompliance with the RFC? I have no idea.

If the server answered all requests from port 69, it would have to distinguish among multiple concurrent transfers by looking at the client socket address; each client transfer would have its own state information including block number, open file, and the time of the last successful packet. This considerably complicates the implementation.

This port-change rule does break TFTP when the server is on the public side of a NAT firewall. When the client sends an RRQ to port 69, the NAT firewall will now allow the server to respond from port 69. However, the server's response from s_port is generally blocked, and so the client never receives Data[1].

11.4 TFTP Stop-and-Wait

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.

11.4.1 Lost Final ACK

The receiver, after receiving the final DATA packet and sending the final ACK, might exit. But if it does so, and the final ACK is lost, the sender will continue to timeout and retransmit the final DATA packet until it gives up; it will never receive confirmation that the transfer succeeded.

TFTP addresses this by recommending that the receiver enter into a **DALLY** state when it has sent the final ACK. In this state, it responds only to received 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 will receive the final ACK and may then exit.

The dally interval should be at least twice the sender's timeout interval. Note that the receiver has no direct way to determine this value.

The TCP analogue of dallying is the TIMEWAIT state, though TIMEWAIT also has another role.

11.4.2 Duplicated Connection Request

Suppose the first RRQ is delayed. The client times out and retransmits it.

One approach would be for the server 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 would significantly complicate the design of a TFTP implementation, because having the server create a new process for each RRQ that arrived would no longer be possible.

So TFTP allows the server to start *two* sender processes, 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 received, recording its source port. The second Data[1] now appears 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"). Were it not for this duplicate-RRQ scenario, packets from an unknown port could probably be simply ignored.

What this means in practice is that the first of the two sender processes above will successfully connect to the receiver, and the second will receive the "Unknown Transfer ID" message and will exit.

A more unfortunate case related to this is below, example 4 under "TFTP Scenarios".

11.4.3 TFTP States

The TFTP specification is relatively informal; more recent protocols are often described using finite-state terminology. In each allowable state, the specification spells out the appropriate response to all packets.

Above we defined a DALLYING 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.

11.5 TFTP scenarios

1. **Duplicated RRQ**: This was addressed above. Usually two child processes will start on the server. The one that the client receives Data[1] from first is the one the client will "latch on" to; the other will be sent an ERROR packet.

2. **Lost final ACK**: This is addressed with the DALLYING state.

3. **Old duplicate**: From the *same* connection, this is addressed by not allowing the 16-bit sequence number to wrap around. This limits the maximum TFTP transfer to 65,535 blocks, or 32 megabytes.

For external old duplicates, involving an earlier instance of the connection, the only way this can happen is if both sides choose the same port number for both instances. If either side chooses a new port, this problem is prevented. If ports are chosen at random as in the sidebar above, the probability that both sides will chose the same 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. Note that this issue represents a second, more fundamental and less pragmatic, reason for having the server choose a new port for each transfer.

After enough time, port numbers will eventually be recycled, but we will assume old duplicates have a limited lifetime.

4. **Getting a different file than requested**: Suppose the client sends RRQ("foo"), but transmission is delayed. In the meantime, the client reboots or aborts, and then sends RRQ("bar"). This second RRQ is lost, but the server sends Data[1] for "foo".

At this point the client believes it is receiving file "bar", but is in fact receiving file "foo".

In practical terms, this scenario seems to be of limited importance, though "diskless" workstations often did use TFTP to request their boot image file when restarting.

If the *sender* reboots, the transfer simply halts.

5. **Malicious flooding**: A malicious application aware that client C is about to request a file might send repeated copies of bad Data[1] to likely ports on C. When C *does* request a file (*eg* if it requests a boot image upon starting up, from port 1024), it may receive the malicious file instead of what it asked for.

This is a consequence of the server handoff from port 69 to a new port. Because the malicious application must guess the client's port number, this scenario too appears to be of limited importance.

11.6 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 for a moment 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.

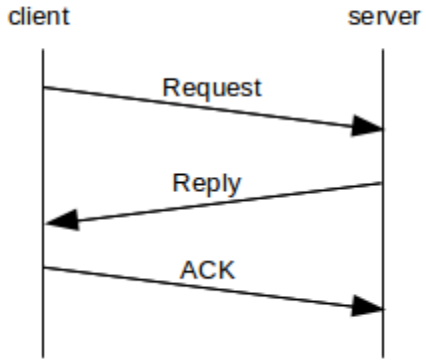
11.7 Remote Procedure Call (RPC)

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.

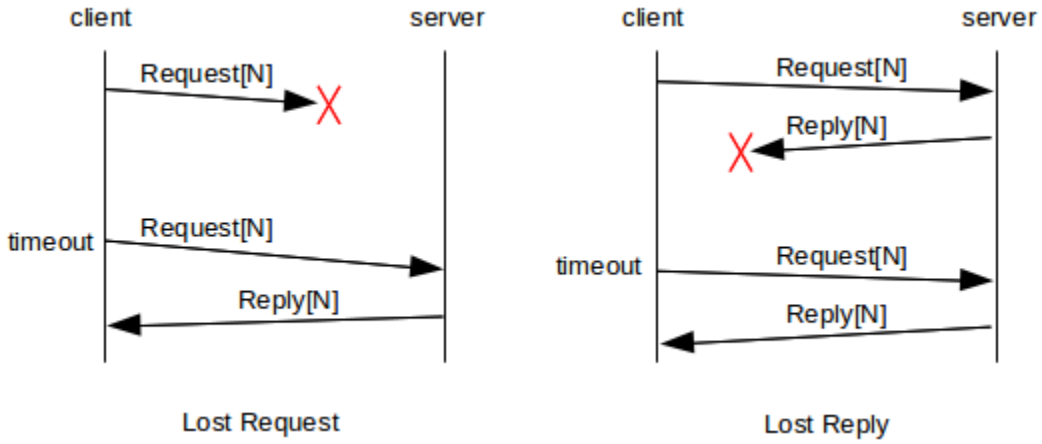
While TCP can be used for processes like these, this adds the overhead of creating and tearing down a connection; in many cases, the RPC exchange consists of nothing further beyond the request and reply and so the TCP overhead would be nontrivial. RPC over UDP is particularly well suited for transactions 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 *11.7.2 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].

11.7.1 Network File Sharing

In terms of total packet volume, the application making the greatest use of early RPC was Sun's **Network File Sharing**, 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/IP packet, using IP fragmentation for transmission over Ethernet.

11.7.2 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 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.

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.

11.7.3 Serialization

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 is to allow multiple outstanding requests and out-of-order replies.

11.7.4 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 server reboots. If a client *really* wants to be sure that its request is executed only once, it needs to be sure that the server did not reboot between the original request and the client's retransmission following a timeout. One way to achieve this is for the server to maintain a "reboot counter", written to the disk and incremented after each restart, and then to include the value of the reboot counter in each reply. Requests contain the client's expected value for the server reboot counter; if at the server end there is not a match, the client is notified of the potential error. Full recovery from what may have been a partially executed request, however, requires some form of application-layer "journal log" like that used for database servers.

11.8 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.

11.9 Exercises

1. Perform the UDP simplex-talk experiments discussed at the end of *11.1.1 UDP Simplex-Talk*. Can multiple clients have simultaneous sessions with the same server?

2. What would happen in TFTP if both sides implemented retransmit-on-timeout and neither side implemented retransmit-on-duplicate? Assume the actual transfer time is negligible. Assume Data[3] is sent but the first instance is lost. Consider these cases:

- sender timeout = receiver timeout = 2 seconds
- sender timeout = 1 second, receiver timeout = 3 seconds
- sender timeout = 3 seconds, receiver timeout = 1 second

3. In the previous problem, how do things change if ACK[3] is the packet that is lost?

4. Spell out plausible responses for a TFTP receiver upon receipt of a Data[N] packet for each of the states UNLATCHED, ESTABLISHED, and DALLYING. Your answer may depend on N.

Example: upon receipt of an ERROR packet, TFTP would in all three states exit.

5. In the TFTP-receiver code in *11.4.3 TFTP States*, explain why we must check `thePacket.getLength()` *before* extracting the opcode and block number.

6. Outline a TFTP scenario in which the TFTP receiver of *11.4.3 TFTP States* sets a socket timeout interval but never encounters a "hard" timeout – that is, a `SocketTimeoutException` – and yet must timeout and retransmit. Hint: the only way to avoid a hard timeout is constantly to receive *some* packet before the timeout timer expires.

7. In *11.5 TFTP scenarios*, under "Old duplicate", we claimed that if either side 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 the old-duplicate problem?

(b). If the server changes its port number on a subsequent connection, but the client does not, what prevents the old-duplicate problem?

8. In the simple RPC protocol at the beginning of *11.7 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. 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. 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]. Can ACKs be cumulative? If not, what should happen if an ACK is lost?

12 TCP TRANSPORT

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. 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 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 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, S will also create a new thread or process to handle communication with s_C. Typically the server S will have multiple connected children of 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.

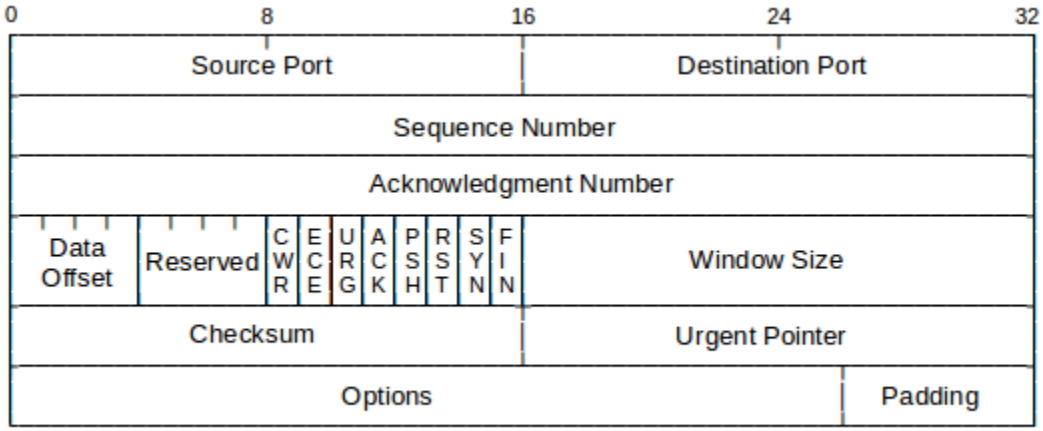
Saltzer, Reed and Clark categorized functions that were subject to the End-to-End principle this way:

> 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 checksum prevents delivery of corrupted data. The Data Offset is for specifying the number of words of Options.



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**.

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.

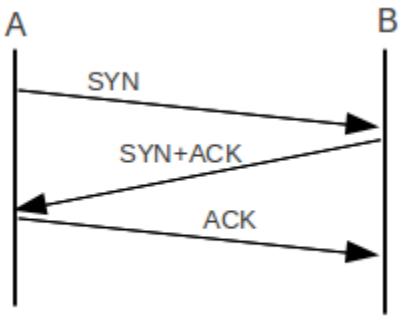
The TCP header defines six important flag bits; the brief definitions here are expanded upon in the sequel:

- **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



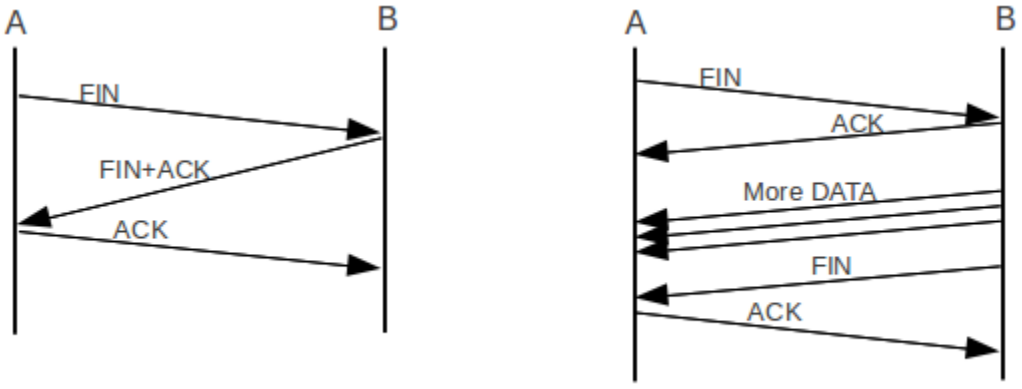
TCP three-way handshake

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 earlier data in a more useful manner, so as to achieve request/reply turnaround comparable to that with RPC (*11.7 Remote Procedure Call (RPC)*). We return to this in *12.11 TCP Faster Opening*.

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
- 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



Two TCP close scenarios

The FIN handshake is really more like two separate two-way FIN/ACK handshakes. If B is ready to close immediately, it may send its FIN along with its ACK of A's FIN, as is shown in the above diagram at the left. In theory this is rare, however, 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. On the other hand, it is possible for B to send a considerable amount of data back to A after A sends its FIN, as is shown at the right. The FIN is, in effect, a promise not to *send* any more, but that side of the connection must still be prepared to receive data. 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

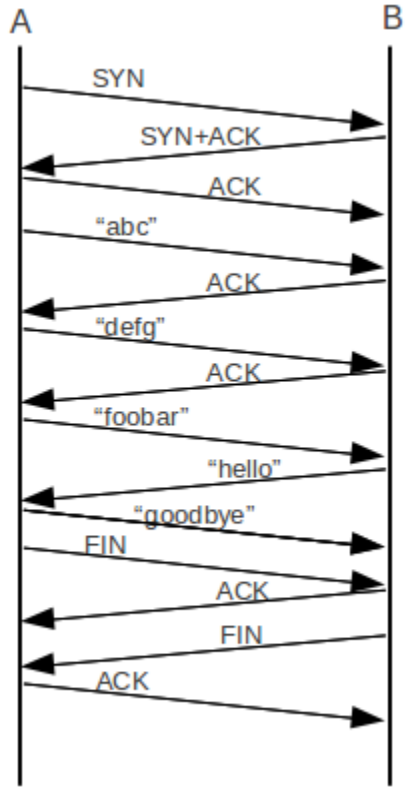
is done sending, and only then does B sort the data and begin sending it back to A. This can be generated with the command, on A, `cat thefile | ssh B sort`.

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	**seq=21**, ack=6, FIN	seq=6, **ack=21** ;; ACK of "goodbye"
12		seq=6, **ack=22** ;; ACK of FIN
13		seq=6, **ack=22**, FIN
14	**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, either side can abort the connection at any time by sending RST.

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.)

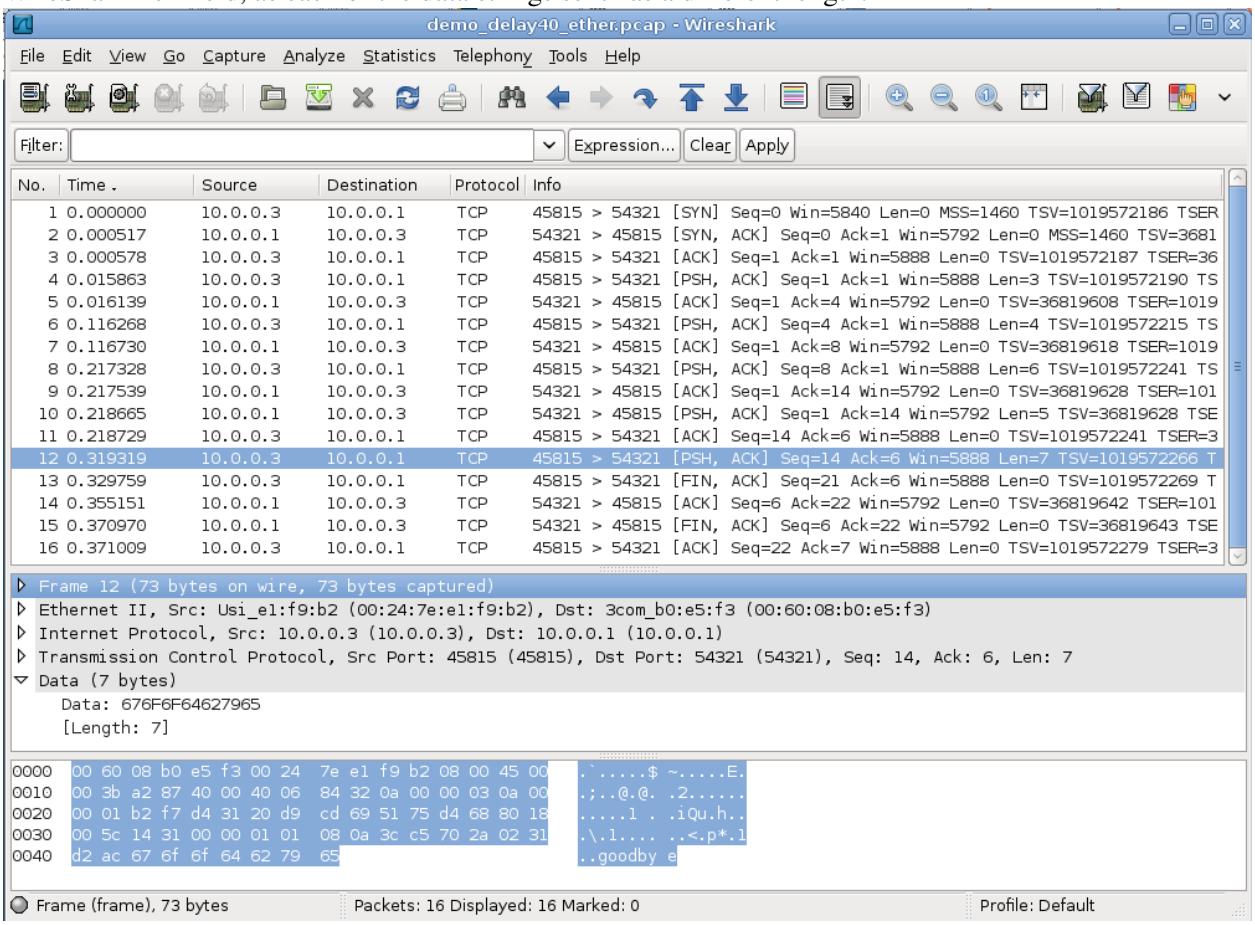
As for the **URG** bit, imagine an ssh connection, in which A has sent a large amount of data to B, which is momentarily stalled processing it. The user at A wishes to abort the connection by sending the interrupt character CNTL-C. Under normal processing, the application at B would have to finish processing 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 far ahead in the normal data stream. The receiver then skips processing the arriving data stream in

first-come-first-served order and processes the urgent data first. For this to work, the receiving process must have signed up to receive an asynchronous signal when urgent data arrives.

The urgent data does appear in the ordinary TCP data stream, and it is up to the protocol to determine the length of the urgent data substring, and what to do with the unread, buffered data sent ahead of the urgent data. For the CNTL-C example, the urgent data consists of a single character, and the earlier data is simply discarded.

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 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 determined 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). 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.

Packets 11-14 in the table and 13-16 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 here was 551144795 and B's ISN was 1366676578. The actual pcap packet-capture file is at demo_tcp_connection.pcap. The hardware involved used **TCP checksum offloading** to have the network-interface card do the actual checksum calculations; as a result, the checksums are wrong in the actual capture file. WireShark has an option to disable the reporting of this.

12.5 TCP simplex-talk

Here is a Java version of the simplex-talk server for TCP. The main `while` loop has the call `ss.accept()` at the start; `ss` is the `ServerSocket` object and is in LISTEN state. This call blocks until an incoming connection is established, at which point it returns the connected child socket. Connections will be accepted from *all* IP addresses of the server host, *eg* the "normal" IP address and also the loopback address 127.0.0.1. Unlike the UDP case (*11.1.1.2 UDP and IP addresses*), the server response packets will always be sent from the same server IP address that the client first used to reach the server.

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`). In the former mode, if a second client connection is made while the first is connected, 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. The main loop is within the `line_talker()` call, and does not return to `ss.accept()` until the connection has closed.

In the latter 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 w
            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
}
```

Here is the 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.

```
// 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
    }
}
```

Here are some things to try with THREADING=false:

- start up two clients while the server is running. Type some lines into both. Then exit the first client.

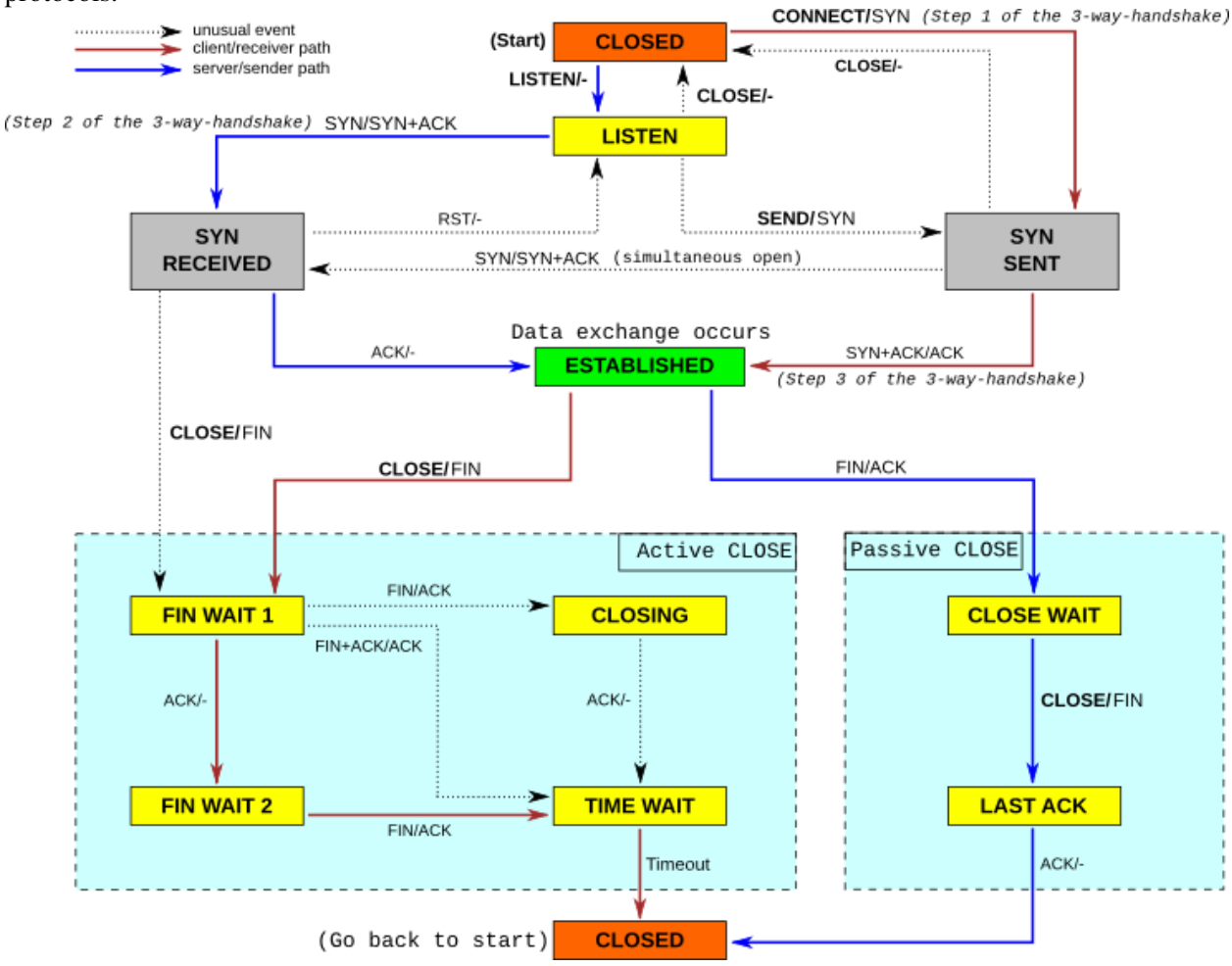
- start up the client before the server.
- start up the server, and then the client. Type some text to the client. Kill the server. What happens to the client? (It may take a couple lines of input)
- start the server, then the client. Kill the server and restart it. Now what happens to the client?

With THREADING=true, try connecting multiple clients simultaneously to the server. How does this behave differently from the first example above?

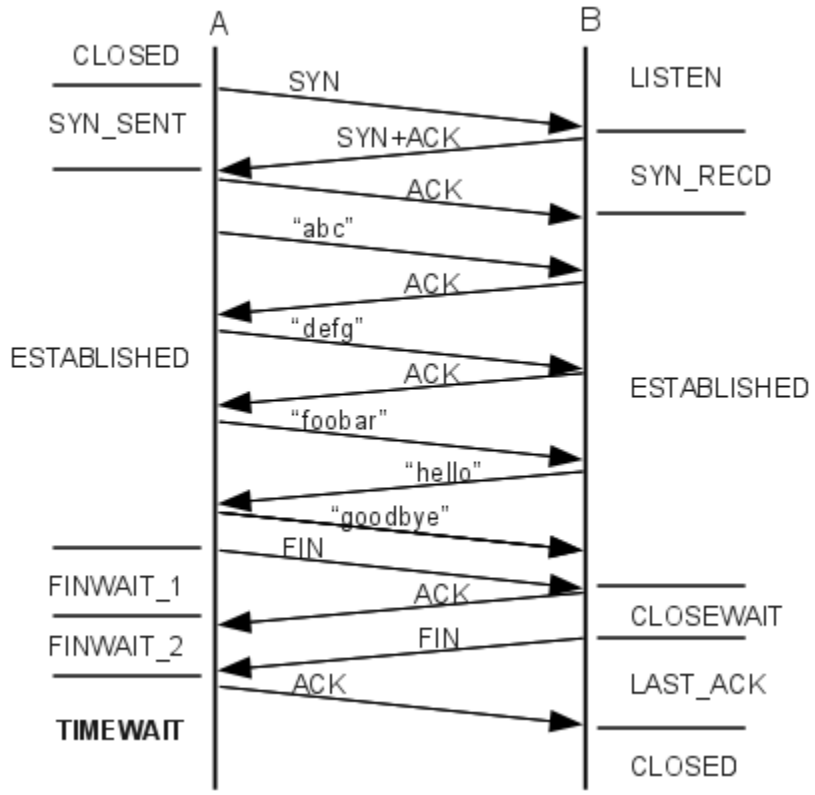
12.6 TCP state diagram

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, the state-diagram approach to protocol design has proven very effective, and is used for most protocols.



Here is the ladder diagram for the 14-packet connection described above, this time labeled with TCP states.



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 opening and closing connections.

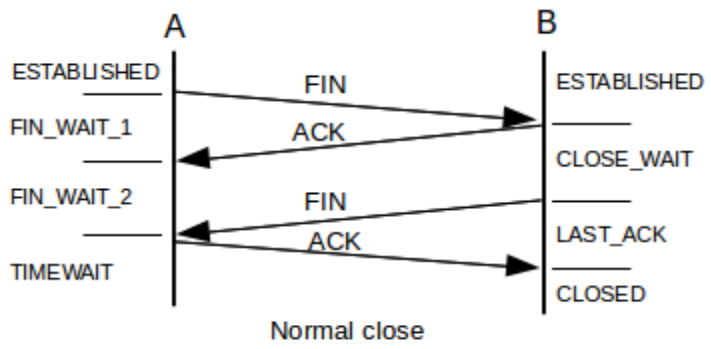
Either side may elect to close the connection (just as either party to a telephone call may elect to hang up). The first side to send a FIN takes the **Active CLOSE** path; the other side takes the **Passive CLOSE** path.

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.

A simultaneous close – having both sides simultaneously send each other FINs – is a little more likely, though still not very. Each side would move to state FIN_WAIT_1. Then, upon receiving each other's FIN packets, each side would move to CLOSING, and then to TIMEWAIT.

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. A TCP endpoint is **half-open** if it is in the ESTABLISHED state, but in the meantime the other side has rebooted. As soon as the ESTABLISHED side sends a packet, the other side will respond with RST and the connection will be fully closed.

The "normal" close is



In this scenario, A has moved from ESTABLISHED to FIN_WAIT_1 to FIN_WAIT_2 to TIMEWAIT (below) to CLOSED. B moves from ESTABLISHED to CLOSE_WAIT to LAST_ACK to CLOSED. All this essentially amounts to two separate two-way closure handshakes.

However, it is possible for B's ACK and FIN to be combined. In this case, A moves directly from FIN_WAIT_1 to TIMEWAIT. In order for this to happen, when A's FIN arrives at B, the socket-owning process at B must immediately wake up, recognize that A has closed its end, and immediately close its own end as well. This generates B's FIN; all this must happen before B's TCP layer sends the ACK of A's FIN. If the TCP layer adopts a policy of *immediately* sending ACKs upon receipt of any packet, this will never happen, as the FIN will arrive well before B's process can be scheduled to do anything. However, if B *delays* its ACKs slightly, then it is possible for B's ACK and FIN to be sent together.

Although this is not evident from the state diagram, 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*.

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.

12.7 TCP Old Duplicates

Conceptually, perhaps the most serious threat facing the integrity of TCP data is external old duplicates (*11.2 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) ignored.

As with TFTP, coming up with a possible scenario for 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. T/TCP (*12.11 TCP Faster Opening*) used 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 *12.10 Anomalous TCP scenarios*.

12.8 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 $2\times$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 (*11.2 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.

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, 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.

12.9 The Three-Way Handshake Revisited

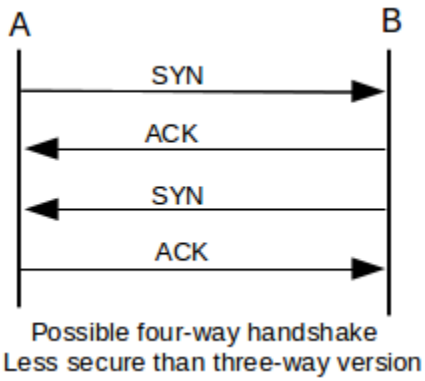
As stated earlier in *12.3 TCP Connection Establishment*, both sides choose an ISN; actual sequence numbers are the sum of the sender's ISN and the relative sequence number. There are two original reasons for this mechanism, and one later one (*12.9.1 ISNs and spoofing*). The original TCP specification, 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 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.



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.

12.9.1 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 "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.

12.10 Anomalous TCP scenarios

TCP, like any transport protocol, must address the transport issues in *11.2 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. In modern TCP implementations this is addressed by 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. This is normally easy to arrange. An old duplicate due to sequence-number wraparound can now be rejected as having an old timestamp.

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.

12.11 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 *12.9.1 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 Internet Draft draft-ietf-tcpm-fastopen-05.txt, 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.

12.12 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 strategy is to send an initial data packet with the IP 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.

12.13 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 $\times$ 100 ms bandwidth$\times$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 $\times$ 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".

12.14 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.

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.

12.15 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. It can usually be disabled on a per-connection basis. See also exercise 10.

12.16 TCP Flow Control

A TCP receiver may reduce the Window Size value of an open connection, thus informing the sender to switch to a smaller window size. This has nothing to do with congestion management but is instead related to **flow control**. This reduction appears in the ACKs sent back by the receiver. A given ACK is not supposed to reduce the window size by more than the size of the data it is acknowledging; in other words, the upper end of the window, lastACKed+winsize, is never supposed to get smaller. This means that no data sent by the sender will ever be retroactively invalidated by becoming beyond the upper edge of the window. A window might shrink from [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 window size to 0, this means that the sender may not send data. This would be done by the receiver to acknowledge that, yes, the data was received, but that 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 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.

12.17 TCP Timeout and Retransmission

For TCP to work well for both intra-server-room and trans-global connections, the timeout value 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

$$\text{EstRTT} = \alpha \times \text{EstRTT} + (1-\alpha) \times \text{SampleRTT}$$

for a fixed α, $0<\alpha<1$. Typical values of α might be $\alpha=1/2$ or $\alpha=7/8$. For α close to 1 this is very conservative in that EstRTT is slow to change. For α close to 0, EstRTT is very 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<β<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.

12.18 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.

12.19 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

12.20 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. That will be the focus of the next three chapters.

12.21 Exercises

1. 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?

2. 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.

3. 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. 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? Which node goes into TIMEWAIT state?

5. (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. Suppose that, after downloading a file, a user workstation is unplugged from the network. The workstation may or may not have first sent a FIN to start closing the connection.

(a). If the receiver has *not* sent the first FIN, what TCP states could this leave the server stuck in? Use the TCP state diagram.

(b). Now suppose the receiver *has* sent its FIN before being unplugged. What states could this leave the server stuck in?

7. 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. Suppose you are watching a video on a site like YouTube, where there is a progress bar showing how much of the video has been downloaded (and which hopefully stays comfortably head of the second progress bar showing your position in viewing the video). Occasionally, the download-progress bar jumps ahead by a modest but significant amount, instantaneously fast (much faster than the bandwidth alone could account for). At the TCP layer, what is going on to cause this jump? Hint: what does a TCP receiver perceive when a packet is lost and retransmitted?

9. 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 100 KB; the server should pause in sending when necessary to enforce this. An ordinary TCP connection will simply transfer data as fast as possible. What support, if any, do you need the receiving (client) application layer to provide, in order to enable this? What must the server-side application then do?

10. 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

11. 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.

13 TCP RENO AND CONGESTION MANAGEMENT

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. 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.

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, but TCP Reno is a cliff-based strategy: packets must be lost before the sender reduces the window size.

In *18 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).

If TCP is sending over an idle network, the per-packet RTT will be $\text{RTT}_{\text{noLoad}}$, the travel time with no queuing delays. As we saw in *6.3.2 RTT Calculations*, (RTT–$\text{RTT}_{\text{noLoad}}$) 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–$\text{RTT}_{\text{noLoad}}$) / 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 $\text{RTT}_{\text{noLoad}}$; the most common approach is to assume that the smallest RTT measured corresponds to $\text{RTT}_{\text{noLoad}}$).

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 × $\text{RTT}_{\text{noLoad}}$. 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 the 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 (*15.4 TCP Vegas*) introduced the fine-grained measurement of RTT, to detect when 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 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" 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 (which we expand on below) is that when a packet is lost, `cwnd` should decrease rapidly, but otherwise should increase "slowly". 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.

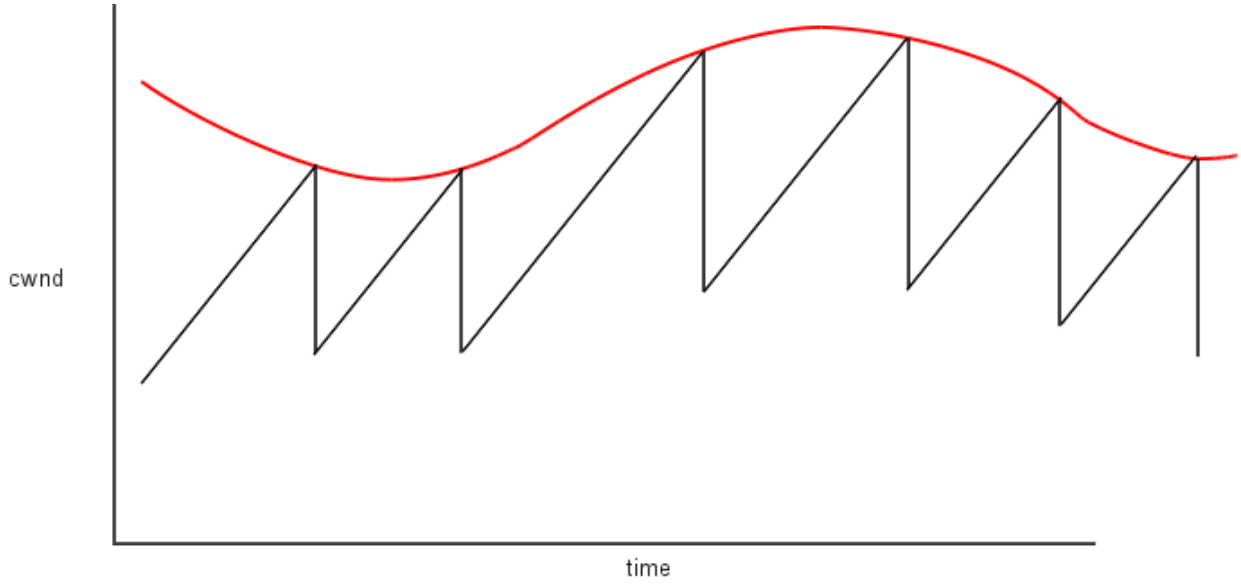
This strategy here is known as **Additive Increase, Multiplicative Decrease**, or AIMD; `cwnd` = `cwnd`+1 is the additive increase and `cwnd` = `cwnd`/2 is the multiplicative decrease. Typically, setting `cwnd`=`cwnd`/2 is a medium-term goal; in fact, TCP Tahoe briefly sets `cwnd`=1 in the immediate 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 1 Gbps, the round-trip transit capacity would be about 6 MB. That is much larger than the likely size of any router queue. A more contemporary model of a typical long-haul high-bandwidth TCP path might be that the queue size is a small fraction of the bandwidth×delay product; we return to this in *13.7 TCP and Bottleneck Link Utilization*.

Note that if TCP experiences a packet loss, and there is an actual timeout, 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.

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

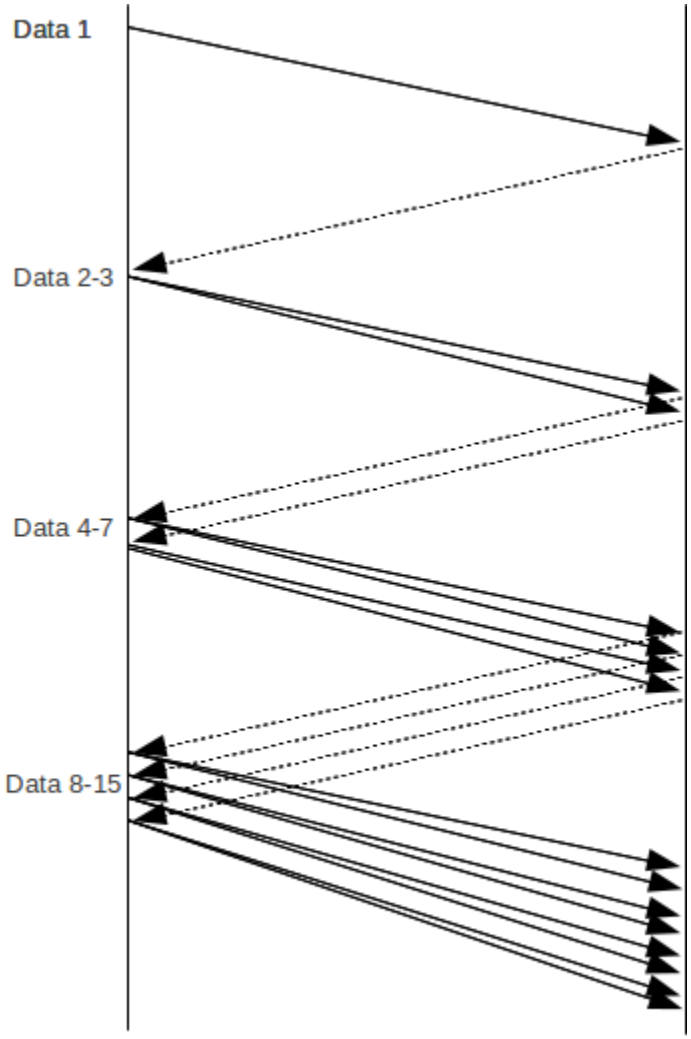
How do we make that initial guess as to the network capacity? What value of cwnd should we begin with? And even if we have a good target for cwnd, how do we avoid flooding the network sending an initial burst of packets?

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. An exact equivalent to the per-windowful incrementing strategy is `cwnd` = `cwnd` + 1/$\texttt{cwnd}_0$, where $\texttt{cwnd}_0$ 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 (*12.14 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 for each windowful.

13.2.2 Threshold Slow Start

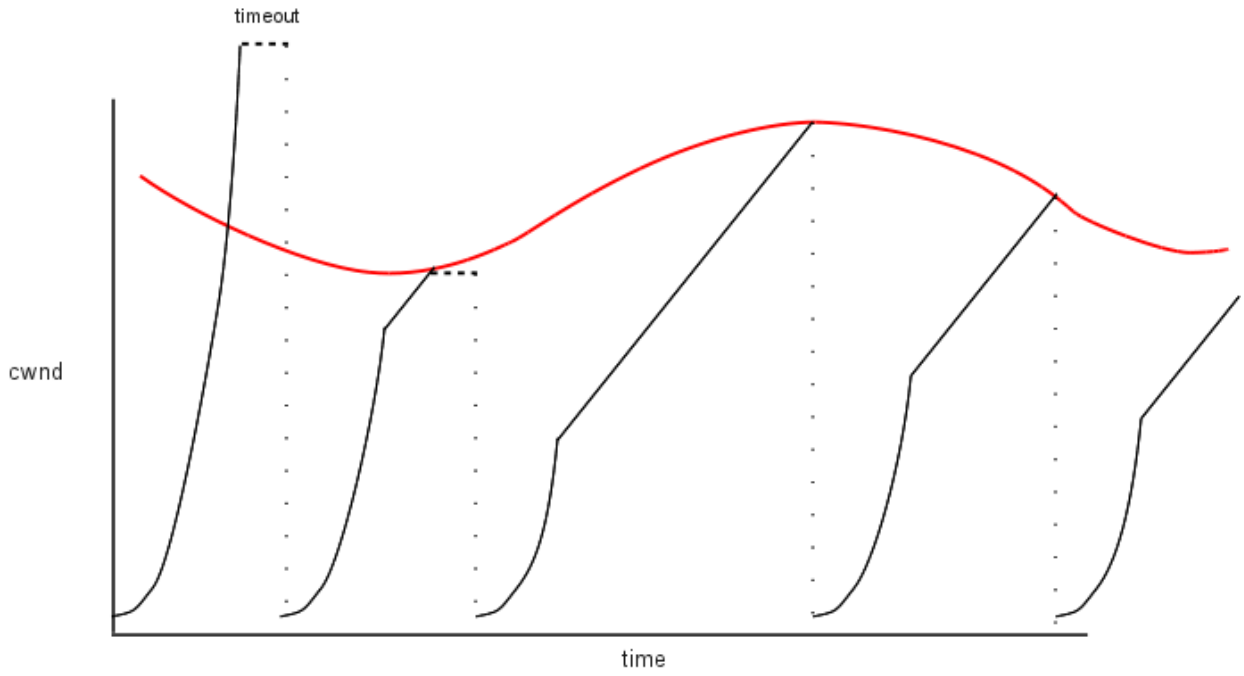
Sometimes TCP uses slow start even when it knows the working network capacity. After a packet loss and timeout, TCP knows that a new `cwnd` of $\texttt{cwnd}_{\text{old}}$/2 should work. If `cwnd` had been 100, TCP halves it to 50. The problem, however, is that after timeout there are no returning ACKs 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.

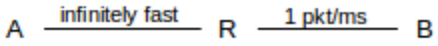


TCP Tahoe Sawtooth, red curve represents the network capacity
Slow Start is used after each packet loss until ssthresh is reached

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`

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×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.

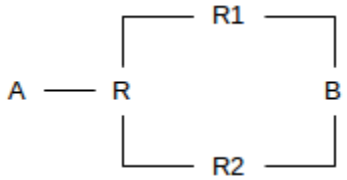
TCP Tahoe included all the features discussed so far: the `cwnd`+=1 and `cwnd`=`cwnd`/2 responses, slow start and Fast Retransmit.

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 set `cwnd`=`cwnd`/2, and then to figure out how many dupACKs we have to wait for. Initially, at least, we will assume that only one packet is lost. Let `cwnd` = N, and suppose we have sent packets 0 through N and packet 1 is lost (we send Data[N] only after ACK[0] has arrived at the sender). We will then get N-1 dupACK[0]s representing packets 2 through N.

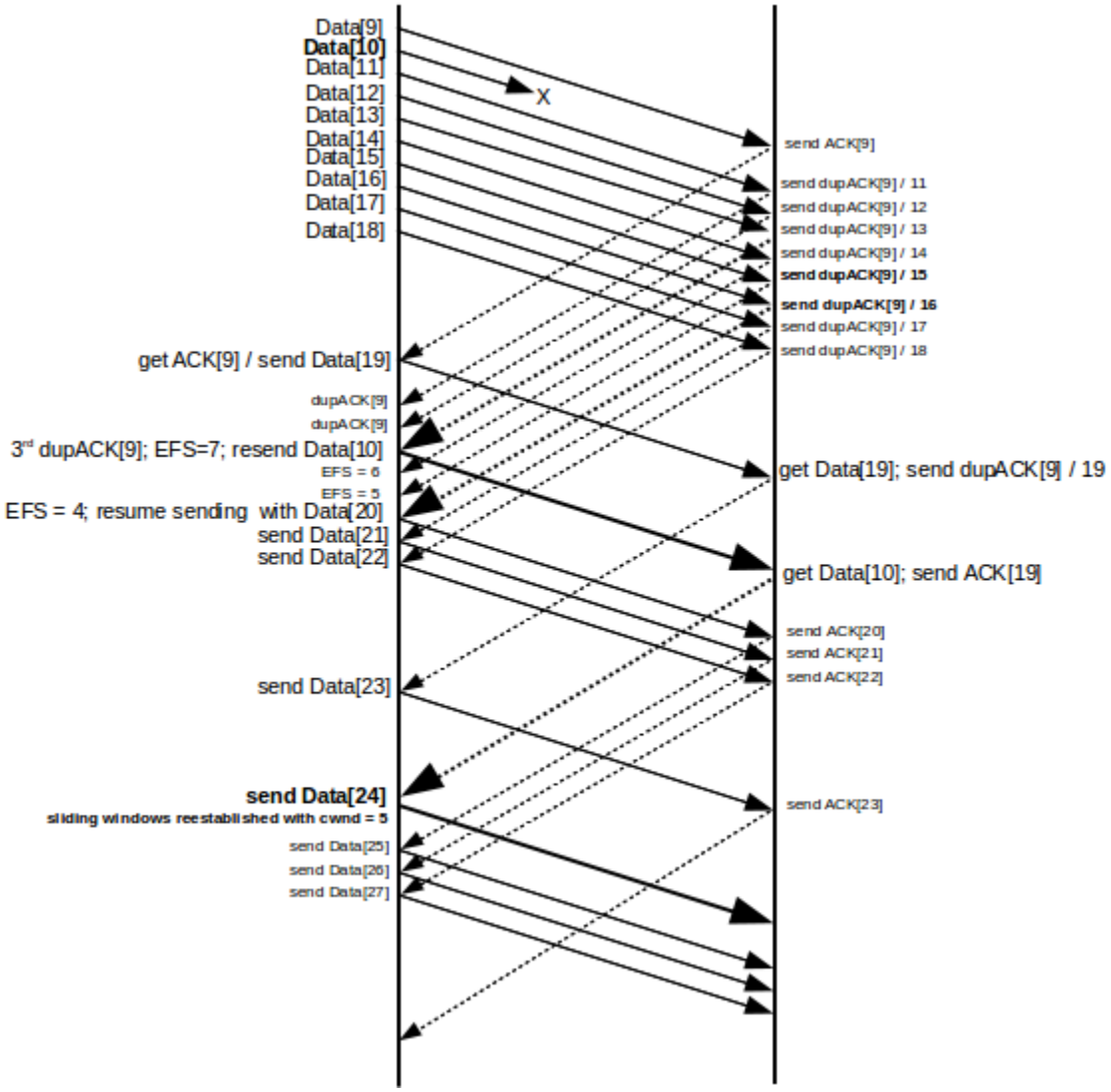
During the recovery process, we will ignore `cwnd` and instead use the concept of Estimated FlightSize, or **EFS**, which is the sender's best guess at the number of outstanding packets. Under normal circumstances, EFS is the same as `cwnd`, at least between packet departures and arrivals.

At the point of the third dupACK, the sender calculates as follows: EFS had been `cwnd` = N. However, one of the packets has been lost, making it N–1. Three dupACKs have arrived, representing three later packets no longer in flight, so EFS is now N–4. Fast Retransmit had the sender retransmit the packet that was inferred as lost, so EFS increments by 1, to N–3. The sender expects at this point to receive N–4 more dupACKs, plus 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. 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 subsequently arriving dupACKs.

After the last of the dupACKs will come the ACK corresponding to the retransmission of the lost packet; it will actually be a cumulative ACK for all the later received packets as well. At this point the sender declares `cwnd` = N/2, and resumes with sliding windows. As EFS was already N/2, and there are no lost packets outstanding, the sender has exactly one full windowful outstanding for the new value of `cwnd`. That is, we are right where we are supposed to be.

Here is a detailed diagram illustrating Fast Recovery. We have `cwnd`=10 initially, and 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.

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.

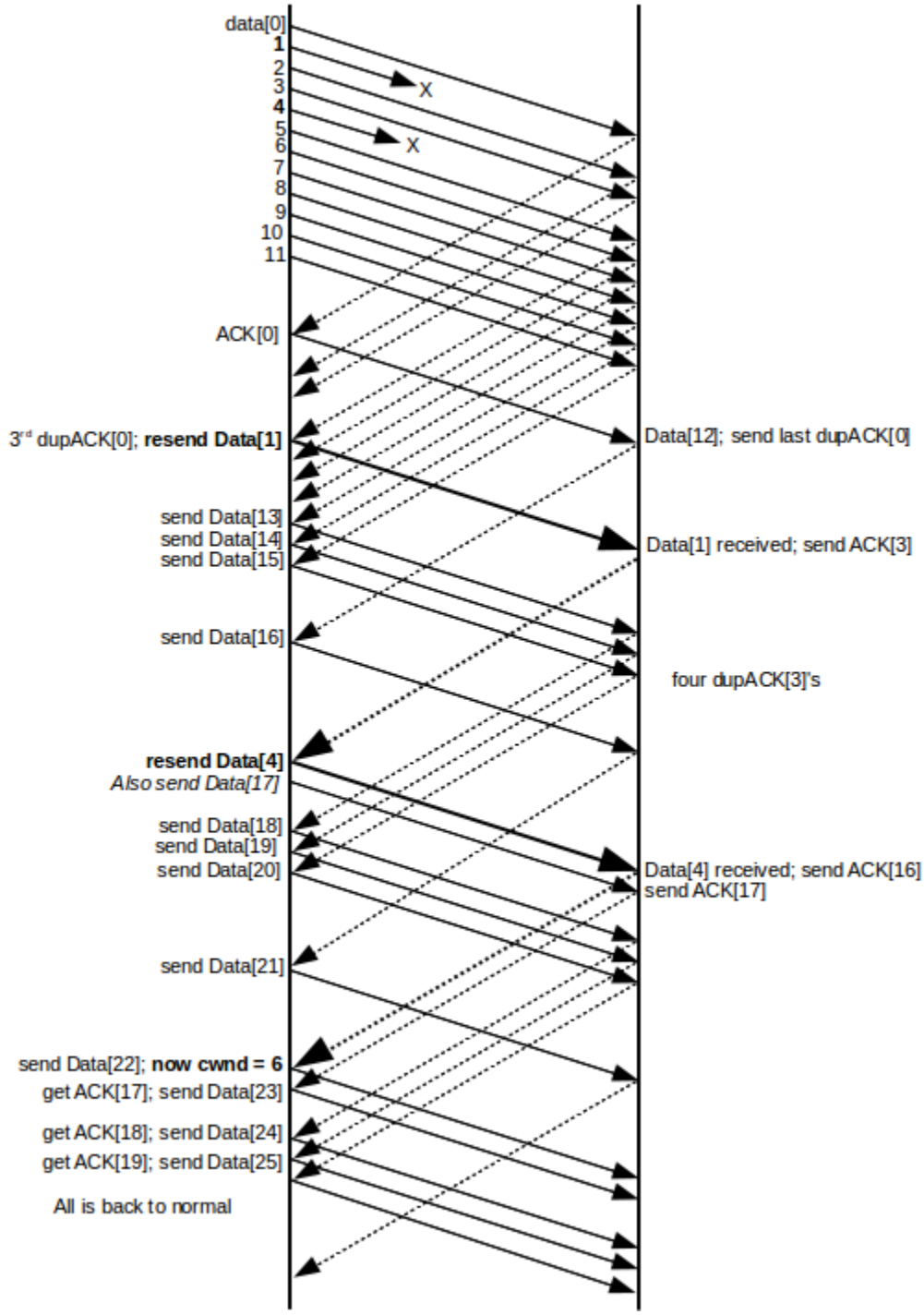
The reason we cannot use `cwnd` directly in the formulation of Fast Recovery is that, until the lost Data[10] is acknowledged, the window is frozen at [10..19]. The original **RFC 2001** description of Fast Recovery described retransmission in terms of `cwnd` "inflation" and "deflation"; inflation would begin at the point the sender resumed transmitting and `cwnd` would be incremented for each dupACK; in the example above, `cwnd` would finish at 15. When the ACK[20] elicited by the lost packet finally arrived, `cwnd` would immediately deflate to 5.

13.5 TCP NewReno

TCP NewReno, described in [JH96] and **RFC 2582**, introduced a modest tweak to Fast Recovery which greatly improves handling of the case when two or more packets are lost in a windowful. 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. NewReno 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 SACK TCP

SACK TCP is TCP with **Selective ACKs**, so the sender does not just guess from dupACKs what has gotten through. The receiver can send an ACK that says:

- All packets up through 1000 have been received
- All packets up through 1050 have been received except for 1001, 1022, and 1035.

Note that SACK TCP involves protocol modification at *both* ends of the connection. Most TCP implementations now support SACK TCP.

Specifically, SACK TCP includes the following information in its acknowledgments; the additional data in the ACKs is included as a TCP Option.

- 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 ACK 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

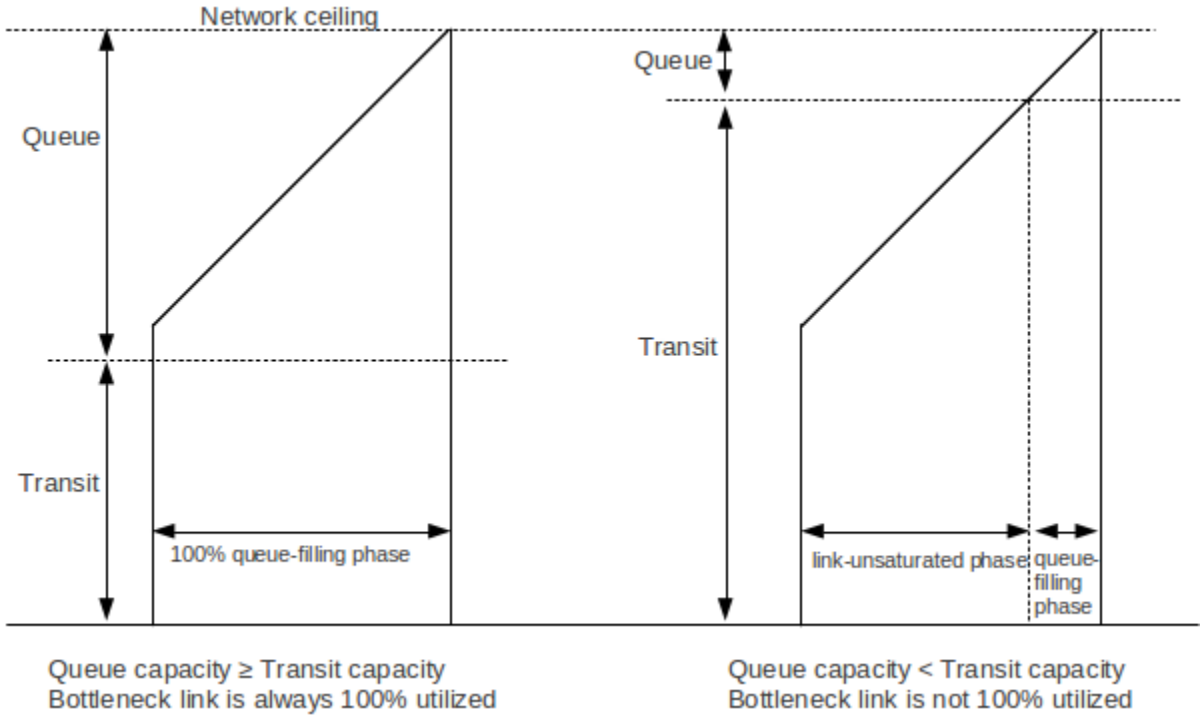
If the sender has been paying close attention to the previous SACKs received, it likely already knows that 1002 through 1021 have been received.

In practice, selective ACKs provide at best a modest performance improvement in most 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

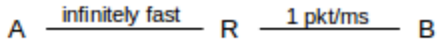
Consider a TCP Reno sender with no competing traffic. As `cwnd` saws up and down, what happens to throughput? Do those halvings of `cwnd` result in at least a dip in throughput? The answer depends to some extent on the size of the queue ahead of the bottleneck link, relative to the transit capacity of 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 $\geq$ transit case. Assume that the total RTT_{noLoad} delay is 20 ms, mostly due to propagation delay; this makes the bandwidth×delay product 20 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 20 packets in transit, that is, we always have `cwnd` $\geq$ 20 (*6.3.2 RTT Calculations*). If $\texttt{cwnd}_{min}$ = 20, then $\texttt{cwnd}_{max} = 2\times\texttt{cwnd}_{min}$ = 40. For this to be the maximum, the queue capacity must be at least 19, so that the path can accommodate 39 packets without loss: 20 packets in transit + 19 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 9. Packet loss will occur when `cwnd` reaches 30, and so $\texttt{cwnd}_{min}$ = 15. 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`=15, throughput will be 75% of the maximum and the R–B link will be idle 25% of the time.

However, `cwnd` will be 15 just for the first RTT following the loss. After 5 RTTs, `cwnd` will be back to 20, and the link will be saturated. So we have 5 RTTs with an average `cwnd` of 17.5, where the link is 17.5/20 = 87.5% saturated, followed by 10 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 $\texttt{cwnd}_{min}$=15, 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 $C_{queue}/2$.

The second case, with queue capacity less than transit capacity, is arguably the more common situation, and becoming more so as time goes on and bandwidths increase. This is almost always the case that applies to high-bandwidth×delay connections, where the queue size is typically much smaller than the bandwidth×delay product. Cisco routers, for example, generally are configured to use queue sizes considerably less than 100, regardless of bandwidths involved, while the bandwidth×delay product for a 50ms RTT with 1 Gbps bandwidth is ~6MB, or 4000 packets of 1500 bytes each. In this case the tooth is divided into a large link-unsaturated phase and a small queue-filling phase.

The worst case for TCP link utilization is if the queue size is close to zero. Using again a bandwidth×delay product 20 of packets, we can see that $\texttt{cwnd}_{max}$ will be 20 (or 21), and so $\texttt{cwnd}_{min}$ will be 10. Link utilization therefore ranges from a low of 10/20 = 50% to a high 100%, over 10 RTTs; the average utilization is **75%**. While this is not ideal, and while some TCP variants have attempted to improve this figure, 75% link utilization is not all that bad, and can again be compared with the 10% of the bandwidth consumed as packet headers. (Note that a literally zero-sized queue may not work at all well with slow start, because the sender will regularly send packets two at a time.)

We will return to this point 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.

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.

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 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 *15.4 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

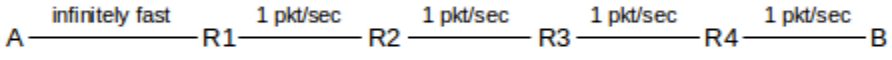
TCP Reno's core congestion algorithm is based on algorithms in Jacobson and Karel's 1988 paper [JK88], now twenty-five years old, although NewReno and SACK have been almost universally added to the standard "Reno" implementation.

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

1. 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 $\text{RTT}_{\text{noLoad}}$ 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 with `ssthresh` = 6, and with `cwnd` initially 1. Complete the table below until two rows after `cwnd` = 6; for these final two rows, 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. 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).

Write out all packet deliveries assuming R's queue size is 5, up until the first dupACK triggered by the arrival at B of a packet that followed a packet that was lost. Assume A starts by sending just Data[1].

3. Repeat the previous problem, except assume R's queue size is 2 and continue the table until all data packets are received. Assume no retransmission mechanism is used, and that A sends new data only when it receives new ACKs (dupACKs, in other words, do not trigger new data transmissions).

4. 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? 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. Suppose TCP Reno is used to transfer a large file over a path with a bandwidth of one packet per 10 µsec and an RTT of 80 ms. Assume the receiver places no limits on window size. Note the bandwidth×delay product is 8,000 packets.

(a). How many RTTs will it take for the window size to reach ~8,000 packets, assuming unbounded slow start is used and there are no packet losses?

(b). Approximately how many packets will be sent by that point?

(c). What fraction of the total bandwidth will have been used up to that point? Hint: the total bandwidth is 8,000 packets per RTT.

6. (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?

7. 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.

(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.

8. 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. 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. 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. 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. 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 $\times$ $\text{RTT}_{\text{noLoad}}$ 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 of the next chapter.

13. 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`$\times$`cwnd` denote the congestion window as measured in bytes.

(b). What is the appropriate formulation if delayed ACKs are used (*12.14 TCP Delayed ACKs*) and we still want `cwnd` to be incremented by 1 for each windowful?

14. 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*.

14 DYNAMICS OF TCP RENO

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.

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 *17.3 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 *18 Quality of Service*.

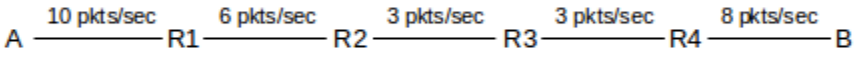
14.2 Bottleneck Links with Competition

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.

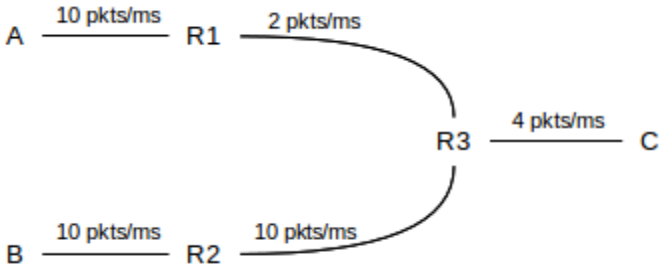


The bottleneck link for A→B traffic is at R2, and the queue will form at R2's outbound interface.

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:



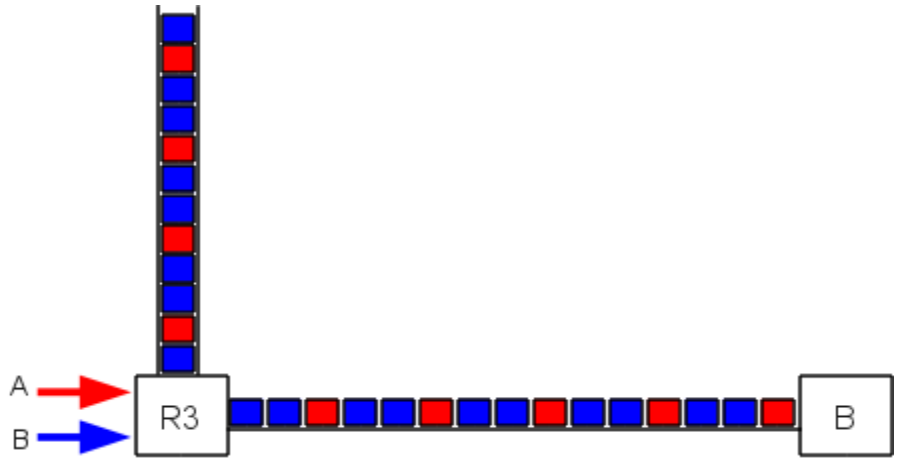
For a moment, assume R3 uses *priority* queuing, with the B⟶C path given priority over A⟶C. If B's flow to C is fixed at 3 packets/ms, then A's share of the R3–C link will be 1 packet/ms, and A's bottleneck will be at R3. However, if B's total flow rate drops to 1 packet/ms, then the R3–C link will have 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 $\alpha \leq 1$ 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

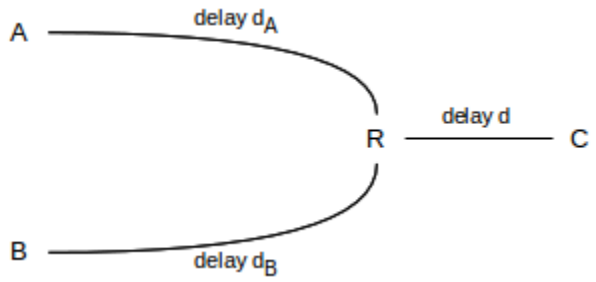
acknowledgments return, each sender will replenish the queue accordingly. However, it is not in A's long-term interest to settle for a queue utilization at R3 of four packets; A may want to take steps that will lead in this setting to a gradual increase of its queue share.

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 are the 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:

queue_usage = winsize – throughput × RTT_{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 RTT_{noLoad} for the A⟶C connection is $2(d_A+d)$ (the factor of 2 is to account for the round-trip) and the RTT_{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 $\times$ ($RTT - RTT_{noLoad}$).

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.

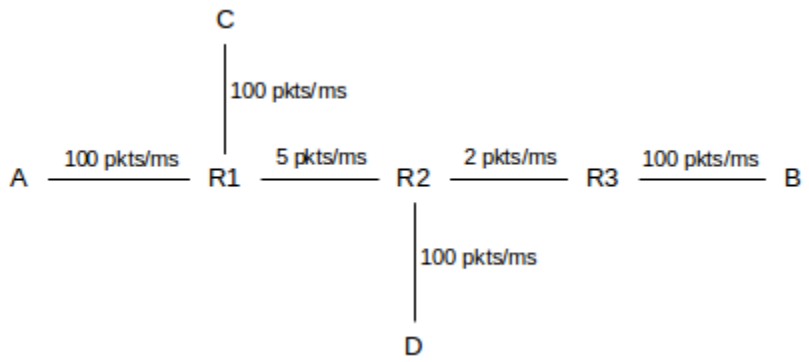
Finally, we can conclude that as w_A gets large and $\beta \to 0$, the limiting value for B's queue utilization Q_B at R will be the entire windowful w_B, up from its starting value (when w_A=0) of $w_B - 2(d_B+d)$. If d_B+d had been small relative to w_B, then Q_B's increase will be modest, and it may be appropriate to consider Q_B relatively constant.

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 $\alpha \to \alpha_{new}$ 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. 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.

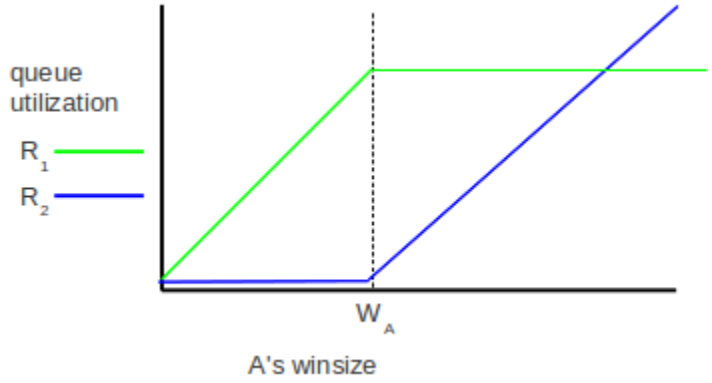
However, if A's winsize grows large enough that its packets consume 40% of R1's queue in the steady state, then this situation changes. At the point when A has 40% of R1's queue, by the Queue Competition Rule it will also have a 40% share of the R1–R2 link's bandwidth, that is, 40% × 5 = 2 packets/ms. 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.

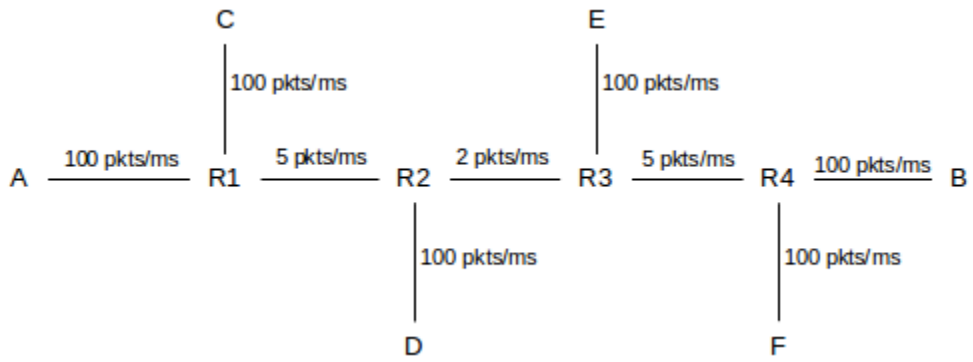
In the graph below, A's contribution to R1's queue is plotted in green and A's contribution to R2's queue is in blue. It may be instructive to compare this graph with the third graph in *6.3.3 Graphs at the Congestion Knee*, which illustrates a single connection with a single bottleneck.



In Exercise 5 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 links.

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.

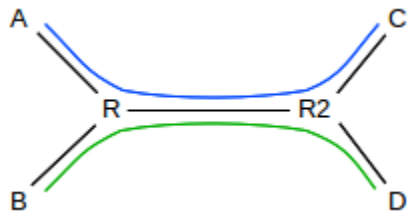
Two things can mar this analysis. First, packets may be reordered; P2 might arrive before P1. Second, P1 and P2 can arrive together at the bottleneck router and be sent consecutively, but then, later in the network, the two packets can arrive at a second router R2 with a (transient) queue large enough 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.

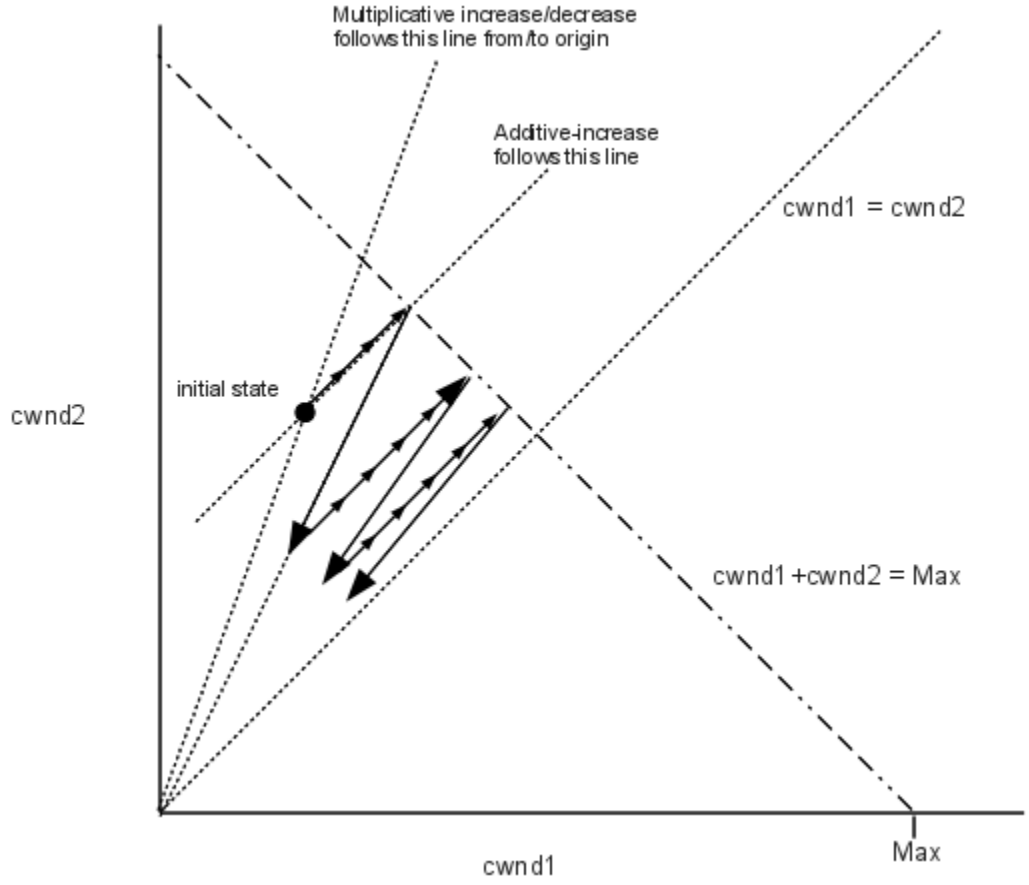
For the moment, consider again two competing TCP connections: Connection 1 (in blue) from A to C and Connection 2 (in green) from B to D, through the same bottleneck router R, *and with the same RTT*. The router R will use tail-drop queuing.



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 ($\texttt{cwnd}_1$ - $\texttt{cwnd}_2$) remained the same (both `cwnd`s incremented by 1), while in any RTT *with* a loss the expression ($\texttt{cwnd}_1$ - $\texttt{cwnd}_2$) 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 $\texttt{cwnd}_1$ on the x-axis and $\texttt{cwnd}_2$ on the y-axis. An additive increase of both (in equal amounts) moves the point (x,y) = ($\texttt{cwnd}_1$,$\texttt{cwnd}_2$) 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 $\texttt{cwnd}_1 + \texttt{cwnd}_2 > 24$, 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

Continued on next page

Table 1 – continued from previous page

T	A–C	B–D	
3	11	1	
4	12	2	
5	13	3	
6	14	4	
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.

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 $\texttt{cwnd}_1 + \texttt{cwnd}_2 > 24$.

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	
14	15	11	second loss

Continued on next page

Table 2 – continued from previous page

T	A–C	B–D	
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 $36 \leq T < 46$ 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 $36 \leq T < 46$ 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 *15.8 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 $\lambda > 1$. 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 $\texttt{cwnd}_1+\texttt{cwnd}_2 \geq C$, 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.

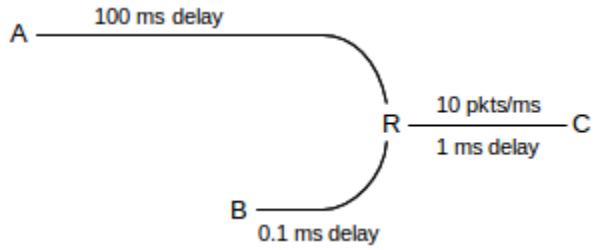
14.3.5 Loss Synchronization

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.

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.

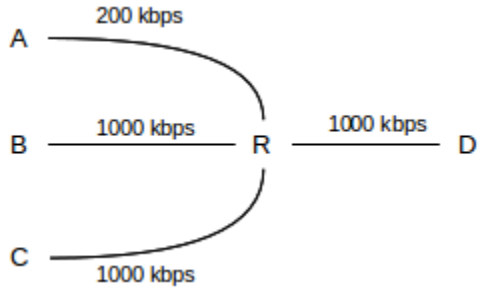
14.4 Notions of Fairness

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.

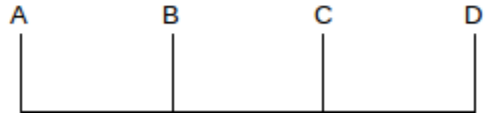
14.4.1 Max-Min Fairness

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:



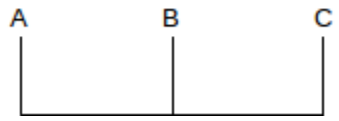
There are four connections: one from A to D covering all three links, and three single-link connections A–B, B–C and C–D. Each link has the same bandwidth. If bandwidth allocations are incrementally distributed among the four connections, then the first point at which any link bandwidth is maxed out 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 the assumption that C is constant represents a very stable network, the general case is not terribly different. 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=$\texttt{cwnd}_{min}$ and 2N=$\texttt{cwnd}_{max}$). If we let M = 3/2 N represent the average cwnd, $\texttt{cwnd}_{mean}$, 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=$\texttt{cwnd}_{mean}$ in terms of p; we get M^2 = 3/2 p^{-1} and thus

$$M = \texttt{cwnd}_{mean} = 1.225\ p^{-1/2}$$

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):

$$\text{bandwidth} = \texttt{cwnd}/\text{RTT} = 1.225/(\text{RTT} \times \sqrt{p})$$

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.

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). 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, 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 $\text{RTT}_2/\text{RTT}_1 = \lambda$ then $\texttt{cwnd}_1/\texttt{cwnd}_2 = \lambda$ and $\text{bandwidth}_1/\text{bandwidth}_2 = \lambda^2$, where $\texttt{cwnd}_1$ and $\texttt{cwnd}_2$ 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. The i^{th} connection's per-packet loss probability is $p_i = \text{losscount}_i/(\text{bandwidth}_i \times T) = (\text{losscount}_i \times \text{RTT}_i)/(\texttt{cwnd}_i \times T)$. But by the result of *14.5 TCP Reno loss rate versus cwnd*, we also have $\texttt{cwnd}_i = k/\sqrt{p_i}$, or $p_i = k^2/\texttt{cwnd}_i^2$. Equating, we get

$$p_i = k^2/\texttt{cwnd}_i^2 = (\text{losscount}_i \times \text{RTT}_i) / (\texttt{cwnd}_i \times T)$$

and so

$$\text{losscount}_i = k^2 T / (\texttt{cwnd}_i \times \text{RTT}_i)$$

Dividing and canceling, we get

$$\text{losscount}_1/\text{losscount}_2 = (\texttt{cwnd}_2/\texttt{cwnd}_1) \times (\text{RTT}_2/\text{RTT}_1)$$

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:

$$\gamma = (\texttt{cwnd}_2/\texttt{cwnd}_1) \times (\text{RTT}_2/\text{RTT}_1)$$

Therefore, as $\text{RTT}_2/\text{RTT}_1 = \lambda$, we must have $\texttt{cwnd}_2/\texttt{cwnd}_1 = \gamma/\lambda$ and thus

$$\text{bandwidth}_1/\text{bandwidth}_2 = (\texttt{cwnd}_1/\texttt{cwnd}_2) \times (\text{RTT}_2/\text{RTT}_1) = \lambda^2/\gamma.$$

Note that if $\gamma=\lambda$, that is, if the longer-RTT connection has fewer loss events in exact inverse proportion to the RTT, then $\text{bandwidth}_1/\text{bandwidth}_2 = \lambda = \text{RTT}_2/\text{RTT}_1$, and also $\texttt{cwnd}_1/\texttt{cwnd}_2 = 1$.

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 $\texttt{cwnd}_{\text{mean}}$ 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, *15.4 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.

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 *18.11.4 RTP and VoIP*.

We will return to RTP in *18.11 Real-time Transport Protocol (RTP)*.

14.7 AIMD Revisited

TCP Tahoe chose an increase increment of 1 on no losses, and a decrease factor of 1/2 otherwise.

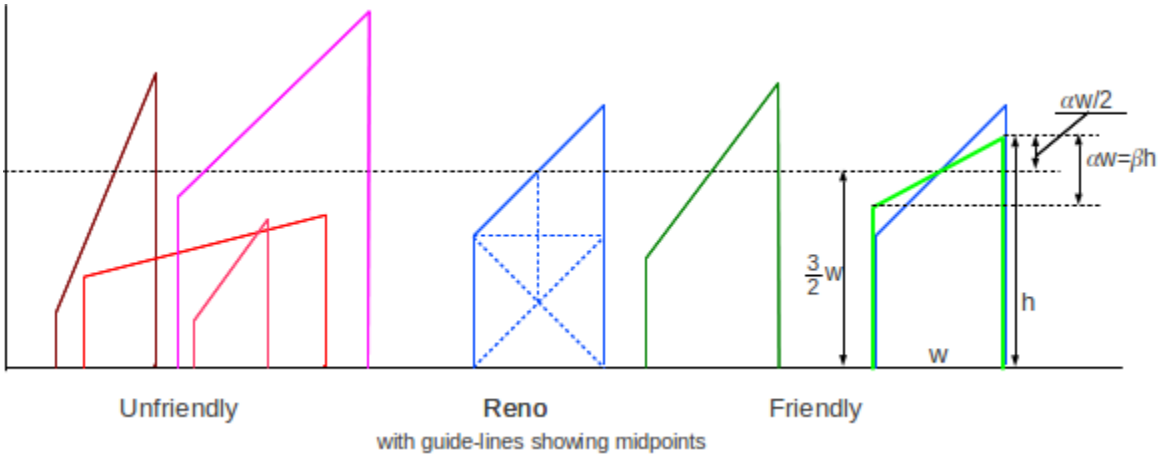
Another approach to TCP Friendliness is to retain TCP's additive-increase, multiplicative-decrease strategy, but to change the numbers. Suppose we denote by AIMD(α,β) the strategy of incrementing the window size by α after a window of no losses, and multiplying the window size by (1-β)<1 on loss (so 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 tooth height. If the teeth of two connections have the same area and the same 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.



The additional dashed lines within the central Reno tooth demonstrate the Reno $1\times1\times2$ dimensions, 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)×w + (α/2)×w. We already know h = (α/β)×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 means that routers take some active steps to manage their queues. Generally this means either **marking** packets or **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".

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 just over 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 *15.4 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; these are 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*.

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, 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).

Another advantage of the ECN mechanism, in addition to avoiding actual packet loss, is that the sender will discover the congestion within a single $\text{RTT}_{\text{noLoad}}$, rather than waiting for the existing queue to be transmitted. If a router's queue reaches the router's congestion-notification threshold, the very next packet forwarded (at the head of the queue) can have the ECN bit set. If a packet is dropped, however, the drop occurs upon arrival at the bottleneck router but is not discovered by the sender until every packet in the queue has been delivered and acknowledged, and the sender is in a position to notice the missing ACK. By then, other losses may have occurred.

Even with ECN, however, the marked packet must still reach its destination and be acknowledged before the sender finds out about the marking. 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 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; the slightly earlier notification about packet loss does not materially affect bandwidth allocation.

14.8.3 RED gateways

"Standard" routers drop packets only when the queue is full; senders have no overt indication before then that the cliff is looming. The idea behind **Random Early Detection** (RED) gateways, introduced in [FJ93], is that the router is allowed to drop an occasional packet much earlier, say when the queue is only 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, in a technical sense, its own "queuing discipline"; we address these further in *17 Queuing and Scheduling*. However, it is often more helpful to think of RED as a technique that an otherwise-FIFO router can use to improve the performance of TCP traffic through it.

Tuning RED has taken some thought. An earlier version 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.

One potential RED drawback is that the choice of the various parameters is decidedly *ad hoc*, and it is not clear how to set them so that TCP connections with both small and large bandwidth×delay products are handled appropriately. The probability p_{base} should, for example, be roughly 1/winsize, but winsize for TCP connections can vary by several orders of magnitude. Nonetheless, RED gateways have worked quite well in practice.

In *18.8 RED with In and Out* we will look at an application of RED to quality-of-service guarantees.

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 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	0.000002
1000	5555	8333	0.00000002
10,000	55555	83333	2×10^{-10}

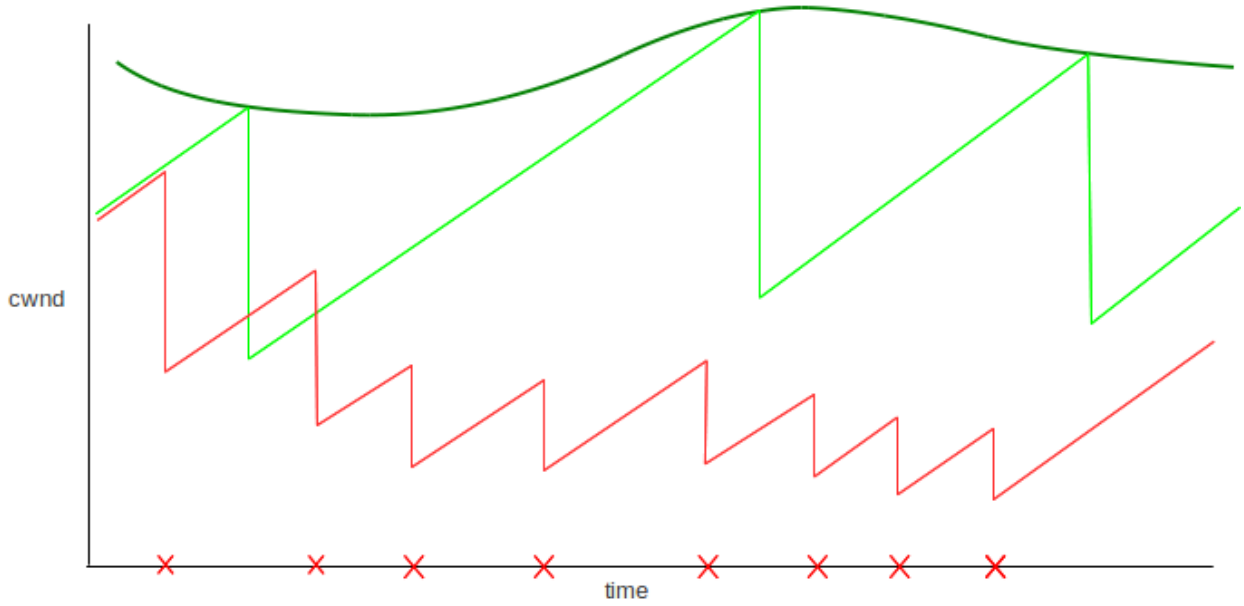
Note the very small loss probability needed to support 10 Gbps; this works out to a bit error rate of less than 2×10^{-14}. For fiber optic data links, alas, a physical bit error rate of 10^{-13} is often considered acceptable; there is thus no way to support the window size of the final row above.

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 (*3.5.2 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.

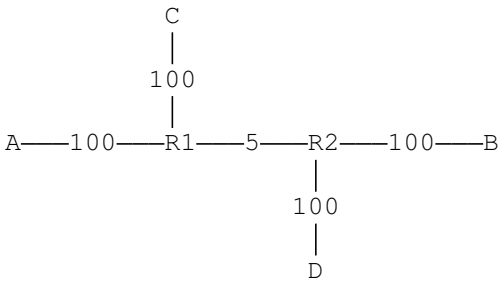
14.12 Epilog

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

1. 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.

2. In the previous exercise, give the average number of **data** packets in transit on each 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.

3. Consider the C–D path from the diagram of *14.2.4 Example 4: cross traffic and RTT variation*:

```
C——100——R1——5——R2——100——D
```

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. 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.

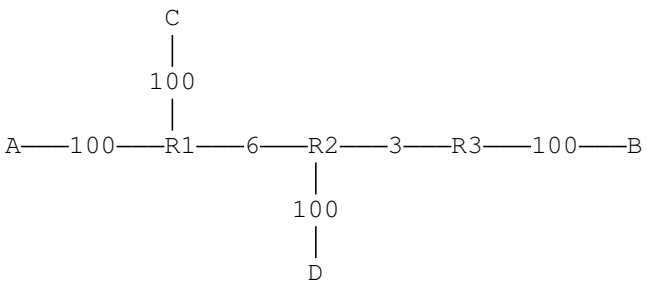


Diagram for exercises 4 and 5

5. 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 The R2–R3 bandwidth is 3 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 bandwidth 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 $\times$ 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 bandwidth 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. 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×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 $\texttt{cwnd}_1 + \texttt{cwnd}_2 > 24$.
(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. 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. Suppose two TCP flows compete. The first flow uses AIMD(α_1,β_1) and the second uses AIMD(α_2,β_2). The two connections compete fairly; 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*.

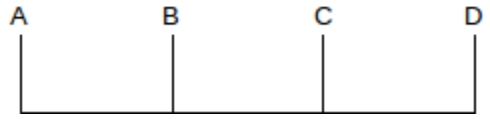
9. Suppose two 1K packets are sent as part of a packet-pair probe, and the minimum time measured between arrivals is 5 ms. What is the estimated bottleneck bandwidth?

10. 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*).

How might a sender distinguish between these three cases? Assume that the sender is willing to create special "probe" packet arrangements.

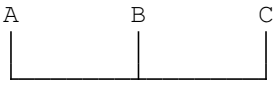
11. Consider again the three-link parking-lot network from *14.4.1 Max-Min Fairness*:



(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.

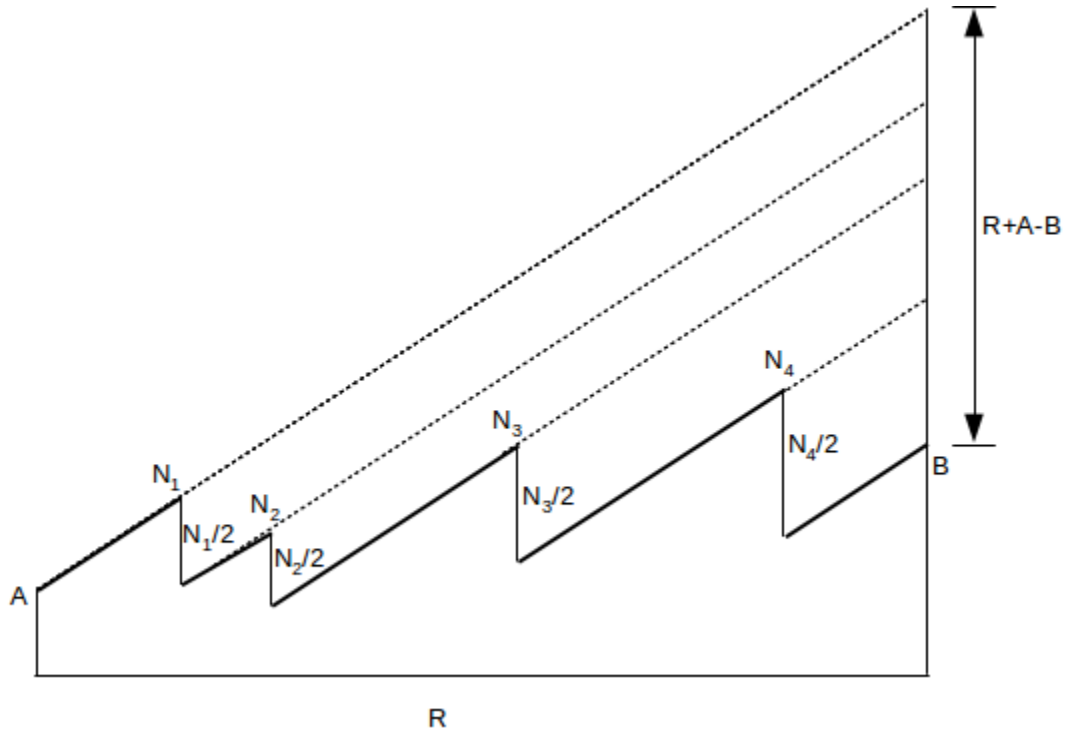
(b). Suppose we have a single end-to-end connection, and one B–C and C–D connection, but two A–B connections. Find the max-min-fair allocation.

12. Consider the two-link parking-lot network:

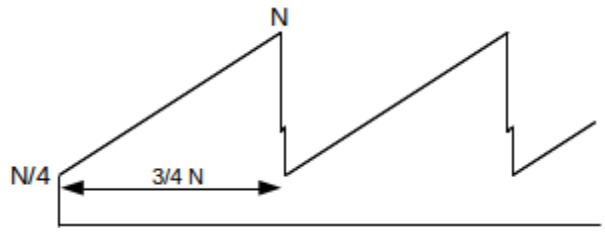


Suppose there are two A–C connections, one A–B connection and one A–C connection. Find the allocation that is proportionally fair.

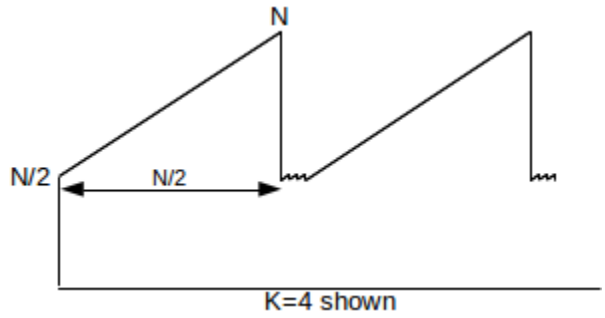
13. Suppose we use TCP Reno to send K packets over R RTT intervals. The transmission experiences n not-necessarily-uniform loss events; the TCP `cwnd` graph thus has n sawtooth peaks of heights N_1 through N_n. At the start of the graph, `cwnd` = A, and at the end of the graph, `cwnd` = B. Show that 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.



14. 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*, the ratio of the average `cwnd` to the square root of the loss rate p, `cwnd`/$\sqrt{}$p, is constant. Find the constant. Note that the loss rate here is p = 2/(number of packets sent in one tooth).



15. 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. 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.

17. 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 $\texttt{cwnd}_{\text{mean}} = C/\sqrt{p}$. 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)

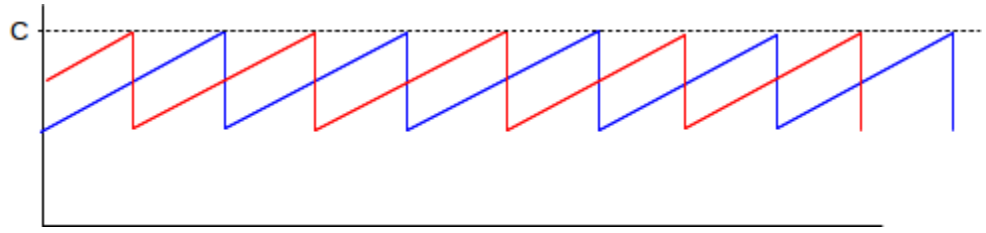
If a loss event marking the end of one tooth occurs at a specific value of `cwnd`, the next tooth begins at height c = `cwnd`/2. If N is the random value for the number of packets in this tooth, then by the previous exercise the tooth width in RTTs is $w = (c^2 + 2N)^{1/2} - c$; the next peak (that is, 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. 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 $\times$ $\text{RTT}_{\text{noLoad}}$ product. Loss events occur at regular intervals.

In Exercise 12 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?

(b). Find the bottleneck-link utilization. Hint: it should be at least 85%.

15 NEWER TCP IMPLEMENTATIONS

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 (*15.4 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.

On linux systems, the TCP congestion-control mechanism can be set by writing an appropriate string to `/proc/sys/net/ipv4/tcp_congestion_control`; the options on my system as of this writing are

- highspeed
- htcp
- hybla
- illinois
- vegas
- veno
- westwood
- bic
- cubic

We review several of these below. TCP Cubic is currently (2013) the default linux congestion-control implementation; TCP Bic was a precursor.

15.1 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.

15.2 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

15.3 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.

For each loss-free RTT, Highspeed TCP allows a `cwnd` increment by more than 1.0, at least once `cwnd` is large enough. If the `cwnd`-increment value is N = N(`cwnd`), this is equivalent to having N parallel TCP Reno connections. Here are the `cwnd`-increment values in terms of `cwnd`:

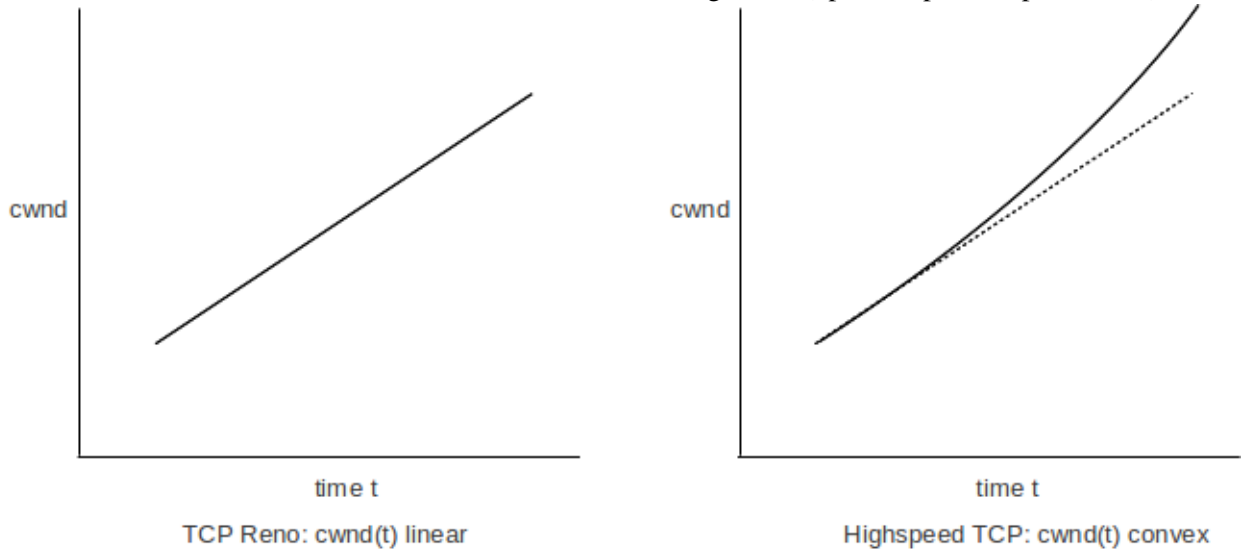
cwnd	N(cwnd)
1	1.0
10	1.0
100	1.4
1,000	3.6
10,000	9.2
100,000	23.0

The formula for N(`cwnd`) is largely empirical; an algebraic expression for it is

$$N(\texttt{cwnd}) = \max(1.0, 0.23 \times \texttt{cwnd}^{0.4})$$

The second term in the max() above begins to dominate when `cwnd` = 38 or so.

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 **convex** (lying above its tangent). This means that there is an increase in the *rate* of `cwnd` increase, as time goes on (up to the point of packet loss).



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`). In this it behaves very much like N separate TCP flows, or, more precisely, N separate TCP flows that have all their loss events completely synchronized.

15.4 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 concern. The goal of TCP Vegas is essentially to eliminate congestive losses, and to try to keep the bottleneck link 100% utilized at all times, thus improving on TCP Reno's typical sawtooth-average bottleneck-link utilization of 75% (*13.7 TCP and Bottleneck Link Utilization*).

TCP Vegas achieves this improvement by, like DECbit, recognizing TCP congestion at the *knee*, that is, at the point where the bottleneck link becomes 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 $\text{RTT}_{\text{noLoad}}$, 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 – $\text{RTT}_{\text{noLoad}}$). TCP Vegas uses this information to attempt to maintain at all times a small but positive number of packets in the bottleneck queue.

A TCP sender can easily measure available bandwidth; the simplest measurement is `cwnd`/RTT (measured in packets/sec). Let us denote this bandwidth estimate by BWE; for the time being we will accept BWE as accurate, though see *15.6.1 ACK Compression and Westwood+* below. TCP Vegas estimates $\text{RTT}_{\text{noLoad}}$ by the minimum RTT (RTT_{min}) encountered during the connection. The "ideal" `cwnd` that just saturates the bottleneck link is $\text{BWE}\times\text{RTT}_{\text{noLoad}}$. 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

$$\text{queue\_use} = \texttt{cwnd} - \text{BWE}\times\text{RTT}_{\text{noLoad}} = \texttt{cwnd} \times (1 - \text{RTT}_{\text{noLoad}}/\text{RTT}_{\text{actual}})$$

TCP Vegas then adjusts `cwnd` regularly to maintain the following:

$$\alpha \leq \text{queue\_use} \leq \beta$$

which is the same as

$$\text{BWE}\times\text{RTT}_{\text{noLoad}} + \alpha \leq \texttt{cwnd} \leq \text{BWE}\times\text{RTT}_{\text{noLoad}} + \beta$$

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 $\text{BWE}\times\text{RTT}_{\text{noLoad}} + \beta$. 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", *15.6.1 ACK Compression and Westwood+*. This is generally not a major problem with TCP Vegas, however.

15.4.1 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 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.

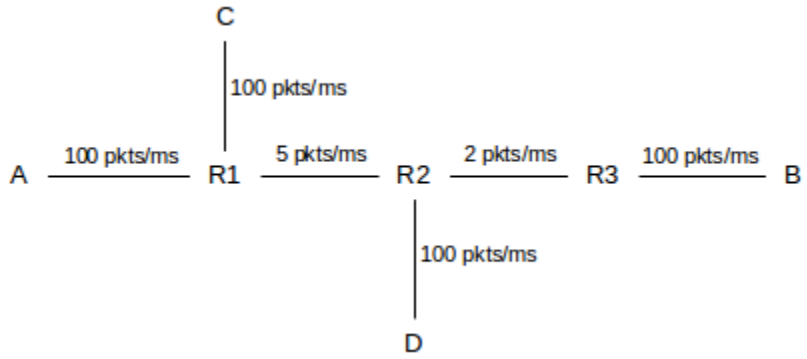
Even worse, Reno's aggressive queue filling will eventually force the TCP Vegas `cwnd` to decrease; see Exercise 4 below.

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, as is the case for most high-bandwidth paths, 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 actual 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.

Note that if the bottleneck router used Fair Queuing (to be introduced in *17.5 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.

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 $\text{RTT}_{\text{noLoad}}$ 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.

15.5 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 com-

petition 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 RTT_{noLoad} as RTT_{min}. At regular short **fixed** intervals (*eg* 20ms) `cwnd` is updated via the following weighted average:

$$\texttt{cwnd}_{new} = (1\text{-}\gamma)\times\texttt{cwnd} + \gamma\times((RTT_{noLoad}/RTT)\times\texttt{cwnd} + \alpha)$$

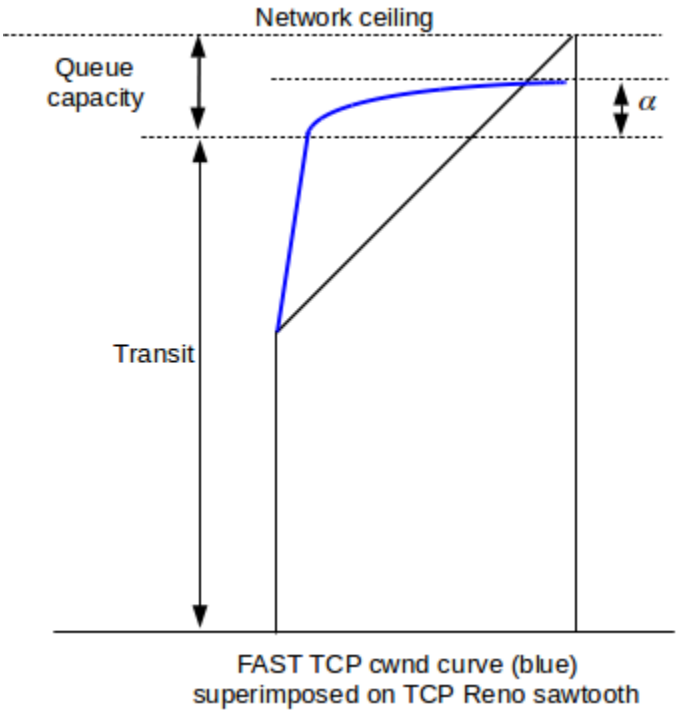
where γ is a constant between 0 and 1 determining how "volatile" the `cwnd` update is ($\gamma\simeq 1$ is the most volatile) and α is a fixed constant, which, as we will verify shortly, represents the number of packets the sender tries to keep in the bottleneck queue, as in TCP Vegas. Note that the `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 $\texttt{cwnd}_{new} = \texttt{cwnd} = ((RTT_{noLoad}/RTT)\times\texttt{cwnd} + \alpha)$, and from there we get $\texttt{cwnd}\times(RTT\text{–}RTT_{noLoad})/RTT = \alpha$. As we saw in *6.3.2 RTT Calculations*, `cwnd`/RTT is the throughput, and so α = throughput $\times$ ($RTT\text{–}RTT_{noLoad}$) 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 $\texttt{cwnd}_{new} = \texttt{cwnd} + \gamma\times\alpha$. 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.

However, FAST TCP can often offset this Reno-competition problem in other ways. 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$\times$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.



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.

15.6 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.

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:

$$\texttt{cwnd} = \max(\texttt{cwnd}/2, \text{BWE} \times \text{RTT}_{\text{noLoad}})$$

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. While BWE may be markedly reduced in the presence of competing traffic, $\text{BWE} \times \text{RTT}_{\text{noLoad}}$ represents how many packets can be in transit (rather than in queues) at the current bandwidth BWE. 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 $\texttt{cwnd}_{\text{min}}$ and $\texttt{cwnd}_{\text{max}} = 2 \times \texttt{cwnd}_{\text{min}}$.
- $\texttt{cwnd}_{\text{max}} \simeq$ 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 < $\texttt{cwnd}_{\text{min}}$, then Reno does a pretty good job keeping the bottleneck link saturated. However, if transit_capacity > $\texttt{cwnd}_{\text{min}}$, then when Reno drops to $\texttt{cwnd}_{\text{min}}$, the bottleneck link is not saturated until cwnd climbs to transit_capacity.

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 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 $\text{BWE} \times \text{RTT}_{\text{noLoad}}$ 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.

15.6.1 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.

15.7 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 $N_{queue} \geq \beta$, `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` = $(4/5) \times$ `cwnd`
if $N_{queue} \geq \beta$, the loss probably *is* due to congestion; set `cwnd` = $(1/2) \times$ `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.

15.8 TCP Hybla

TCP Hybla ([CF04]) has one very specific focus: to address the TCP satellite problem (*3.5.2 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 $\texttt{cwnd}_{min}$. 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:

$$\texttt{cwnd} = (t–t_0)/RTT + \texttt{cwnd}_{min}$$

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 $\sim\varrho^2$. 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.

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.

15.9 TCP Illinois

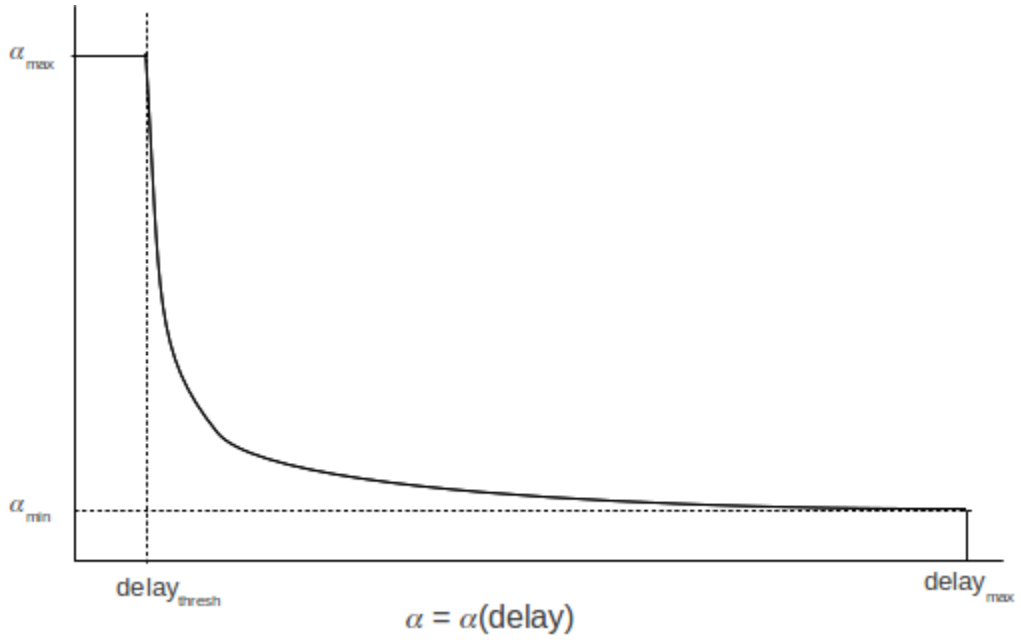
The general idea behind TCP Illinois, described in [LBS06], is to use the usual AIMD(α,β) strategy but to have $\alpha = \alpha$(RTT) be a decreasing function of the current RTT, rather than a constant. When the queue is empty and RTT is equal to $\text{RTT}_{\text{noLoad}}$, then α will be large, and `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 – $\text{RTT}_{\text{noLoad}}$. As with TCP Vegas, $\text{RTT}_{\text{noLoad}}$ 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 $\text{delay}_{\text{max}}$ 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 my 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 α(delay) as follows

$\alpha(\text{delay}) = \alpha_{\text{max}}$ if $\text{delay} \leq \text{delay}_{\text{thresh}}$
$\alpha(\text{delay}) = k_1/(\text{delay}+k_2)$ if $\text{delay}_{\text{thresh}} \leq \text{delay} \leq \text{delay}_{\text{max}}$

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 $\text{delay} \leq \text{delay}_{\text{max}}$. 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 α(delay) begins to fall – rather precipitously – to α_{min}. The value of α(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 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 $\texttt{cwnd}_{\text{min}}$ 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 $t \le t_L$ 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 α(t) after each RTT, or, equivalently, by α(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 $1-\beta$). 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

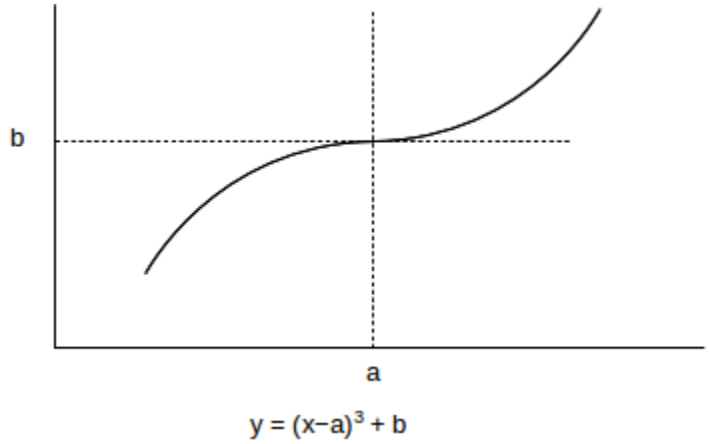
$$1 - \beta = \text{RTT}_{\text{min}}/\text{RTT}_{\text{max}}$$

The value $1-\beta$ 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 $\text{RTT}_{\text{noLoad}}$, 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, $1-\beta$ 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 $\texttt{cwnd}_{\text{new}}$ = 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 $1-\beta$ 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.

15.11 TCP CUBIC

TCP Cubic attempts, like Highspeed TCP, to solve the problem of efficient TCP transport when bandwidth×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 $\beta = 0.2$. 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

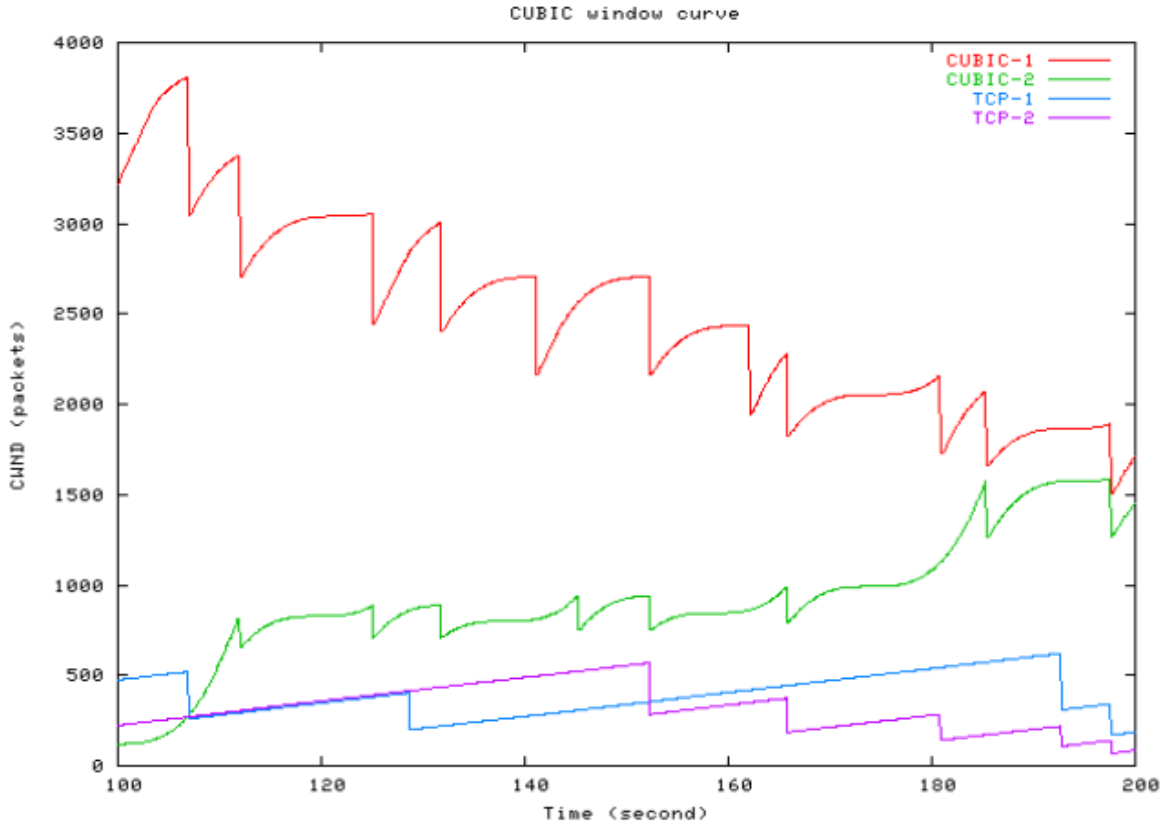
For smaller RTTs, the basic TCP Cubic strategy above runs the risk of being at a competitive disadvantage compared to TCP Reno. For this reason, TCP Cubic makes a **TCP-Friendly adjustment** in the window-size calculation: on each arriving ACK, `cwnd` is set to the maximum of W(t) and the window size 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 $(1\text{-}\beta)\times W_{max} + \alpha\times$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$\times$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).

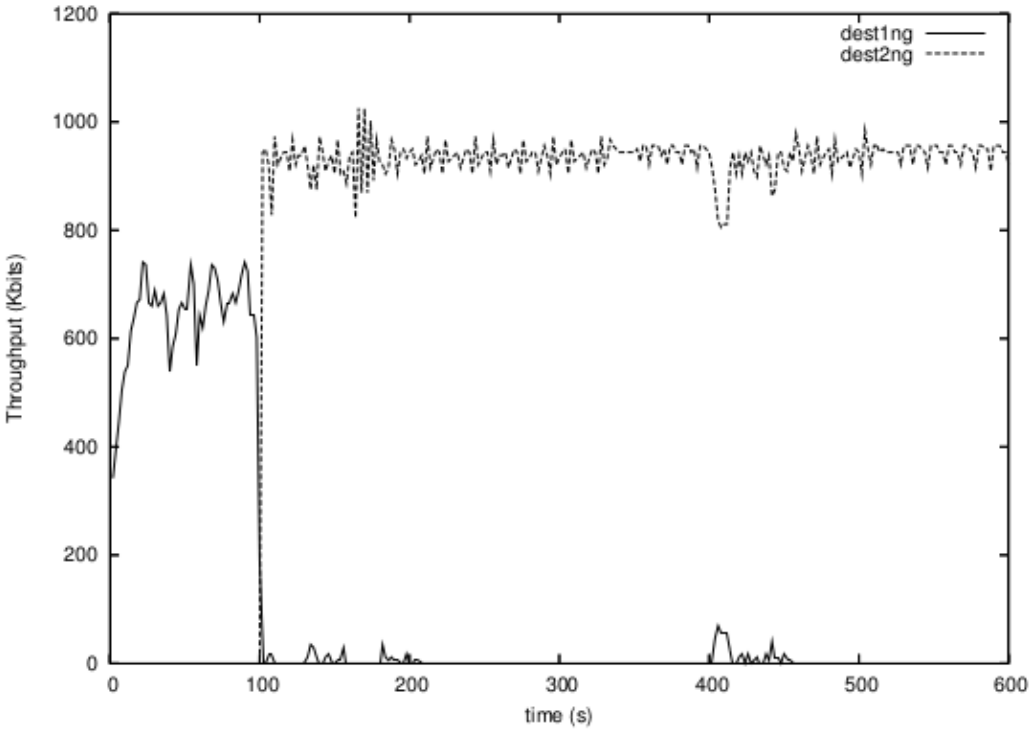


Fig. 14. Two Cubic TCP flows. 1Mbps link, flow 1 RTT 160ms, flow 2 RTT 16ms.

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.

15.12 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).

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 simply make sure the receiver has what it needs for optimum performance (*eg* SACK TCP) and run a few simple experiments to find what works best.

That leaves the question of what TCP to use on a server that is serving up large volumes of data, perhaps to a

range of disparate hosts and with a wide variety of competing-traffic scenarios. Experimentation works here too, but likely with a much larger number of trials. There is also the possibility that one eventually finds a solution that works well, only to discover that it succeeds at the expense of other, 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 (*17 Queuing and Scheduling*) that fundamentally changed the rules of the game.

15.13 Exercises

1. 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. 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. 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 $\texttt{cwnd}_V$. Show that the number of packets TCP Vegas calculates are in the queue, **queue_use**, is

queue_use = $\texttt{cwnd}_V \times Q/(Q+M)$

4. 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 $\texttt{cwnd}_V$ TCP Vegas packets and $\texttt{cwnd}_R$ TCP Reno packets in flight and $\texttt{cwnd}_V+\texttt{cwnd}_R>200$, N_{queue} is $\texttt{cwnd}_V/(\texttt{cwnd}_V+\texttt{cwnd}_R)$ multiplied by the total queue utilization $\texttt{cwnd}_V+\texttt{cwnd}_R-200$.

Continue the following table, where T is measured in RTTs, up through the next two RTTs where $\texttt{cwnd}_V$ 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, $\texttt{cwnd}_V$ is not decremented.) This can be done either with a spreadsheet or by simple algebra. Note that the TCP Reno $\texttt{cwnd}_R$ will always increment.

T	$\texttt{cwnd}_V$	$\texttt{cwnd}_R$	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

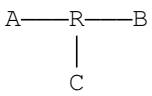
This exercise attempts to explain the *linear decrease* in the TCP Vegas graph in the diagram in *16.5 TCP Reno versus TCP Vegas*. Competition with TCP Reno means not only that $\texttt{cwnd}_V$ stops increasing, but in fact it decreases by 1 most RTTs.

5. 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 $\texttt{cwnd}_F$ and $\texttt{cwnd}_R$. You may use the fact that, as long as the queue is nonempty, $\text{RTT}/\text{RTT}_{\text{noLoad}}$ = ($\texttt{cwnd}_F$+|R2|)/200.

Find the value of $\texttt{cwnd}_F$ 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	$\texttt{cwnd}_F$	$\texttt{cwnd}_R$
0	100	100
1	105	101
2	108.47	102
3	110.77	103
4	112.20	104

6. 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 $\text{RTT}_{\text{noLoad}}$ 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. 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. 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 $\texttt{cwnd}_{\text{min}} = 0$ (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. 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 $\texttt{cwnd}_\text{I}$ and $\texttt{cwnd}_\text{R}$. The bottleneck queue capacity is 100.

Find the value of $\texttt{cwnd}_\text{I}$ at T=50, where T is the number of elapsed RTTs. At this point $\texttt{cwnd}_\text{R}$ is, of course, 150.

T	$\texttt{cwnd}_\text{I}$	$\texttt{cwnd}_\text{R}$
0	100	100
1	101	101
2	?	102

You may assume that the delay, RTT – $\text{RTT}_{\text{noLoad}}$, is proportional to queue_utilization = $\texttt{cwnd}_\text{I}$+|R3|–200α. Using this expression to represent delay, $\text{delay}_{\text{max}} = 100$ and so $\text{delay}_{\text{thresh}} = 1$. When calculating α(delay), assume $\alpha_{\text{max}} = 10$ and $\alpha_{\text{min}} = 0.1$.

10. 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. For each of the values of W_{max} below, find the *change* in `cwnd` over one 100 ms RTT at each of the following points:

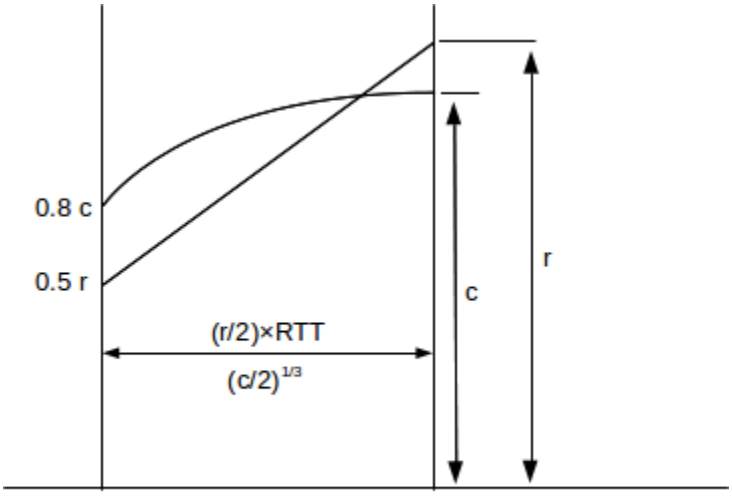
i. Immediately after the previous loss event, when t = 0.

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. 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)×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.

> 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*

16 NETWORK SIMULATIONS

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 ns-2, 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 almost universally used in the literature when comparing different TCP flavors for effective throughput (for example, the graphs excerpted in *15.11 TCP CUBIC* were created through simulation). 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 most 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 here the **ns-2** version, hosted at www.isi.edu/nsnam/ns. There is now also an ns-3 version, available at nsnam.org, though it is not backwards-compatible with ns-2 and the programming interface has changed considerably. While it is likely that ns-3 will be included in this book at some point in the future, the particular simulation examples presented here 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 and in some respects ns-2 is simpler to program and configure. The standard ns-2 tracefile format is also quite easy to work with.

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.

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 is also 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 types

```
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
```

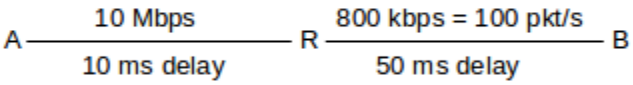
Object attributes are retrieved using `set` without a value; the following assigns variable `ack0` the current value of the `ack_` field of `tcp0`:

```
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$\times$RTT$_{\text{noLoad}}$ product is 0.1 packets/ms $\times$ 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_
```

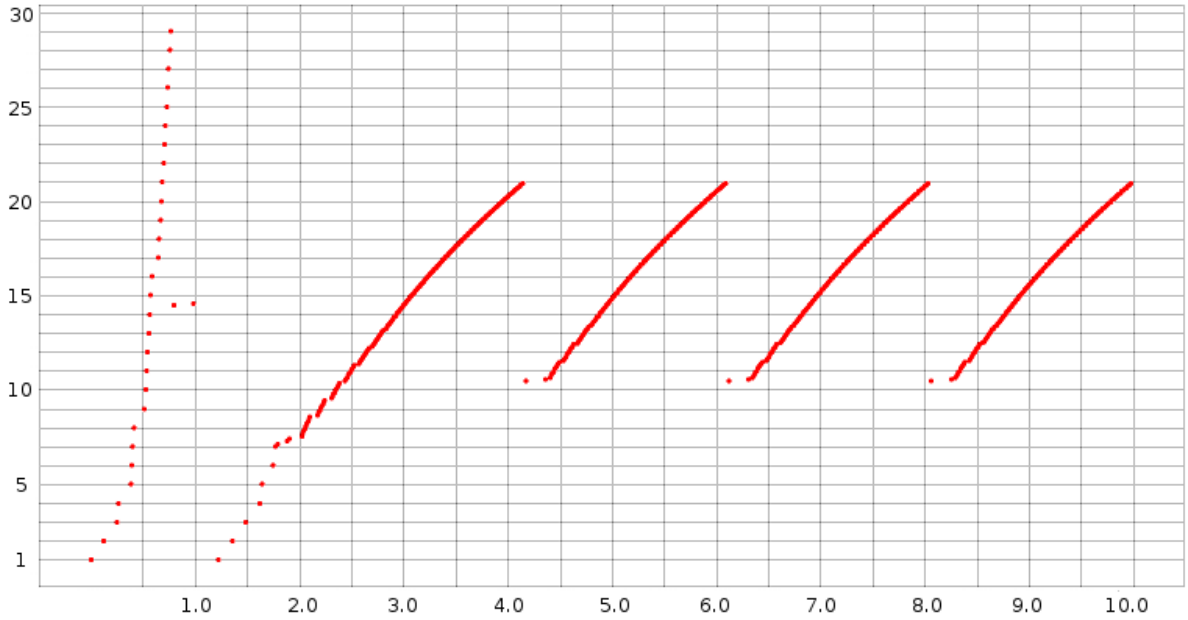
```
$tcp0 tracevar ack_
$tcp0 tracevar maxseq_

#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.

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 *15.3 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 drop, **+** 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 $\times$ 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 $\text{RTT}_{\text{noLoad}}$ 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

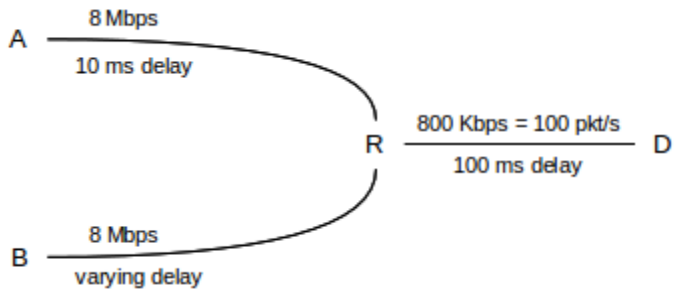
If we run the basic1.tcl simulation above until time 1000, there are 94915 packets acknowledged and 512 loss events. This yields a loss rate of p = 512/94915 = 0.00539, and so by the formula of *14.5 TCP Reno loss rate versus cwnd* we should expect the average `cwnd` to be about $1.225/\sqrt{p} \simeq 16.7$. 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 $\text{RTT}_{\text{noLoad}}$, 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 $k
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 γ=λ 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 $\text{RTT}_{\text{noLoad}}$ minus the bandwidth delay at R, we can also compute the phase as

$$\phi = \text{fractional\_part}(\text{RTT}_{\text{noLoad}} \div (\text{R's bandwidth delay})).$$

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.

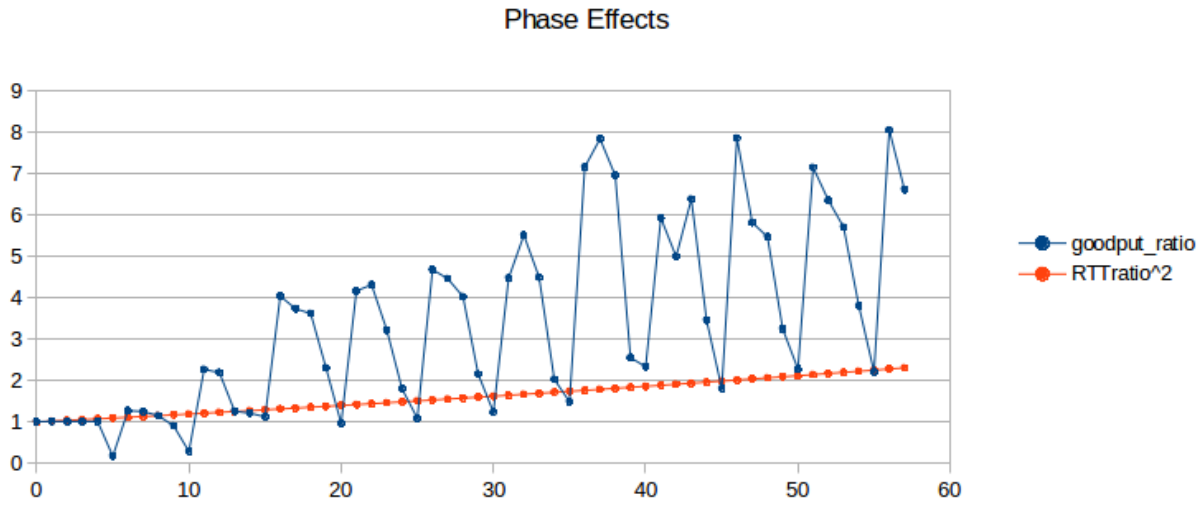
16.3.4.2 Two-sender phase effects

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.

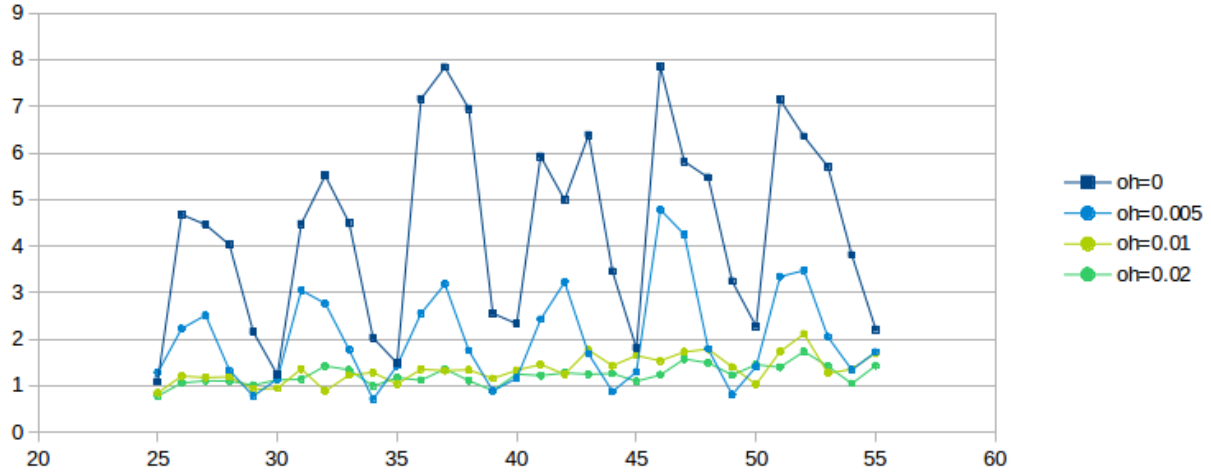
Techniques introduced in [FJ92] to break synchronization in ns-2 simulations were random "telnet" traffic – involving smaller packets sent according to a given random distribution – and the use of random-drop queues (not included in the standard ns-2 distribution). The second, of course, means we are 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.

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.

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 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.

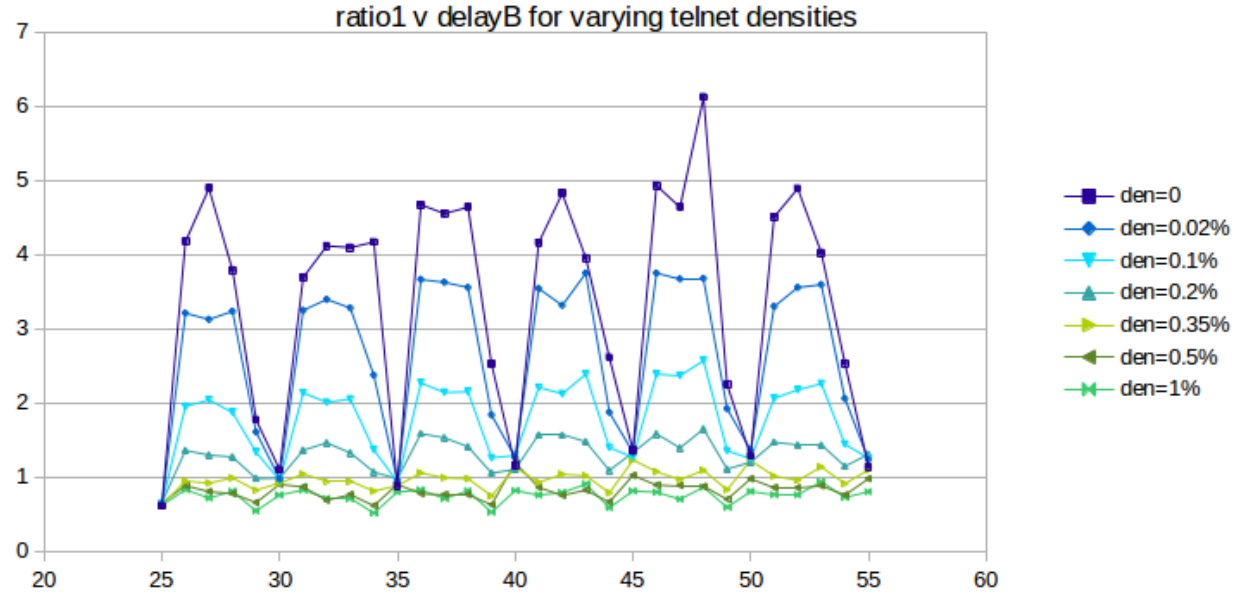
The bandwidth theoretically consumed by the telnet connection is simply `actualSize`/`$tninterval`; the actual bandwidth may be lower if the telnet packets are encountering congestion as noted above. It is convenient to define an attribute `tndensity` that denotes the fraction of the R–D link's bandwith 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

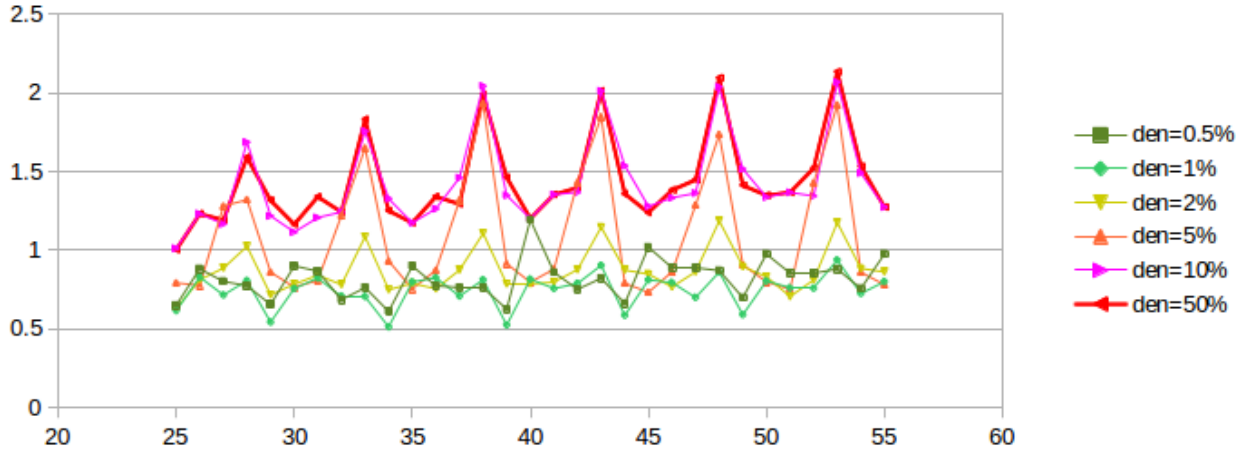
= 50 ms (converted for ns-2 to 0.05 sec).

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,

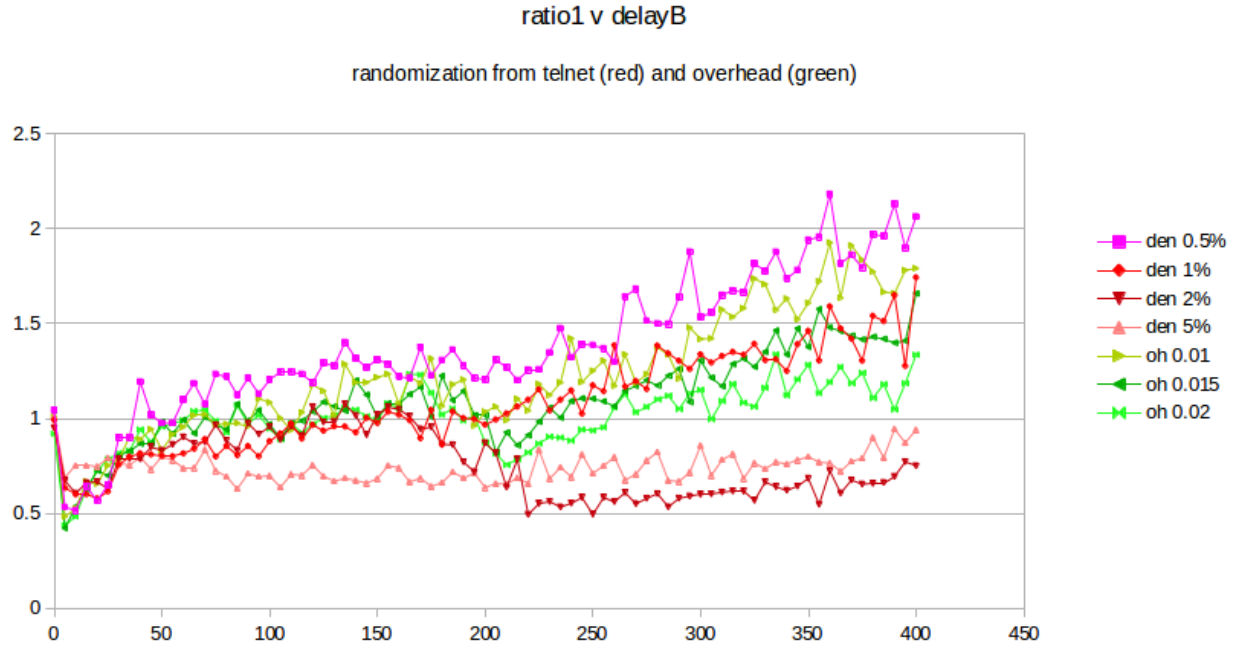
and the curves converge towards the heavier red curve at the top.

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.

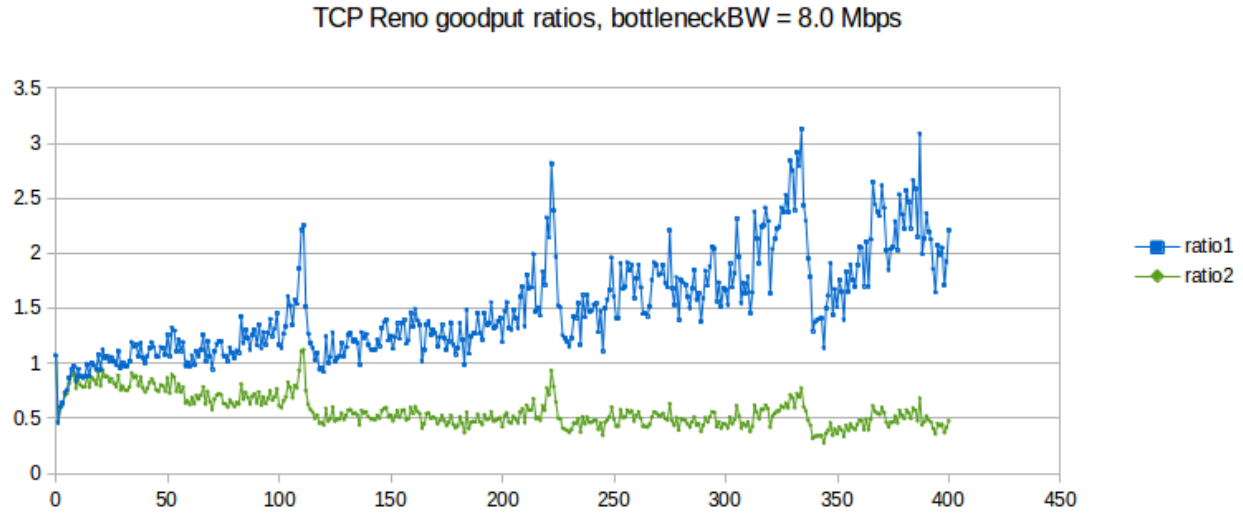
16.3.9 The RTT Problem

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 $\text{RTT}_{\text{noLoad}}$ 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)$\times$170/220 + (1/6)$\times$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 good-

put 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.

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.

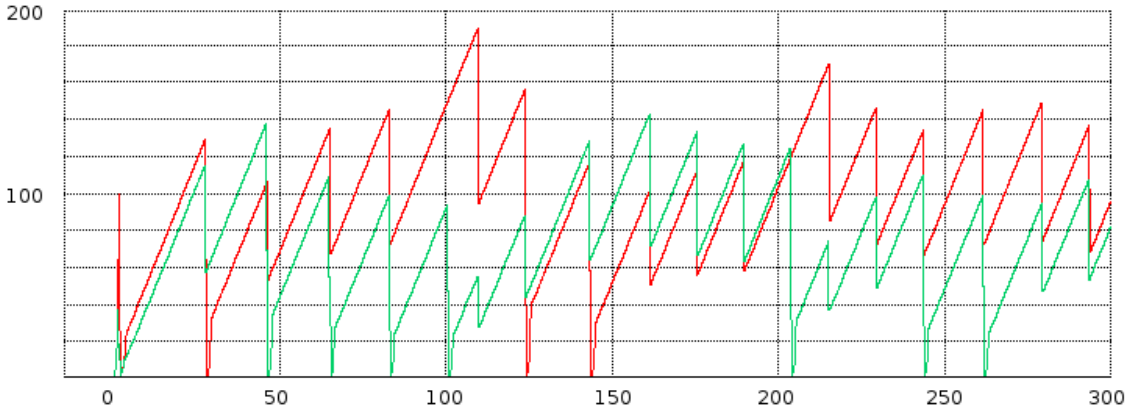
ns-2 sensitivity

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.

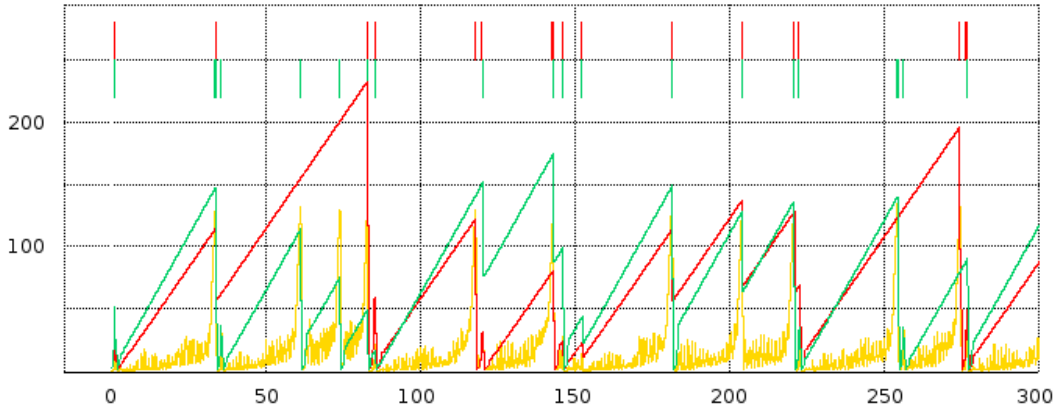
16.4.1.1 delayB = 0



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.

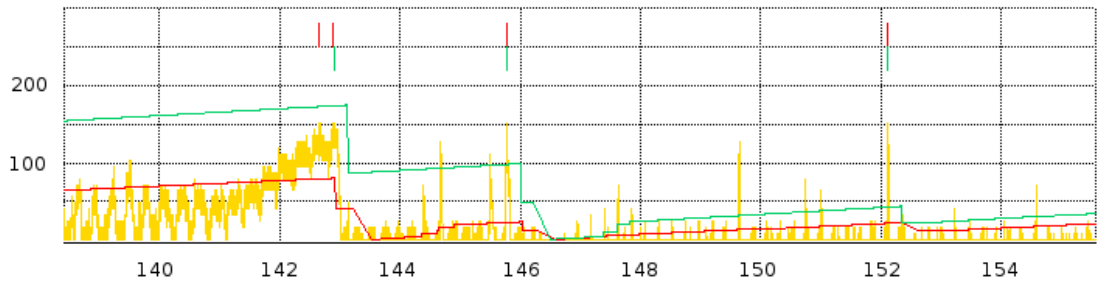
Overall, the red path has 11 teeth if we use a tooth-counting granularity of 3 seconds, and 8 teeth if the granularity is 5 seconds. We do not count anything that happens in the slow-start phase, generally before T=3.0, nor do we count the "tooth" at T=300 when the graph ends.

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 `cwnd`s 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.



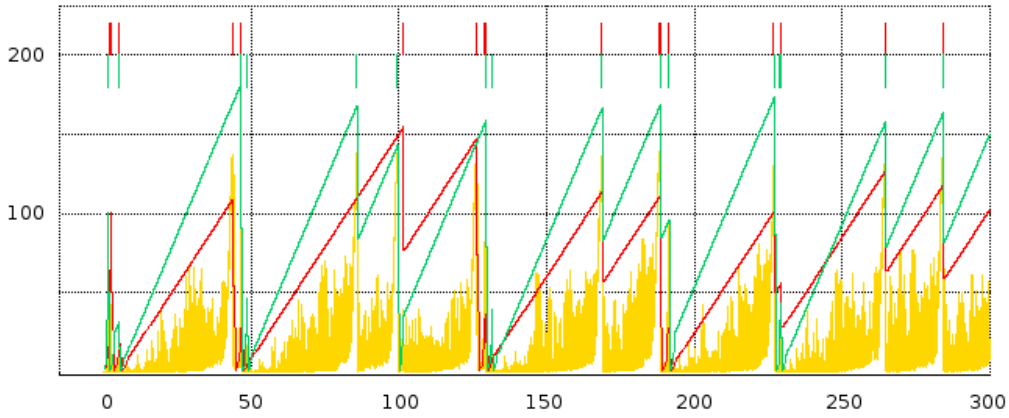
There are two very sharp, narrow peaks in the queue graph, just before T=146 and just after T=152, each causing packet drops for both flows. Neither peak, especially the latter one, appears to have much to do with the gradual queue-filling and loss event at T=143. Neither one of these **transient queue peaks** is associated with the sum of the `cwnd`s approaching the maximum of about 250; at the T≃146 peak the sum of the `cwnd`s 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 coincidentally timed so they 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.

16.4.1.4 delayB = 61

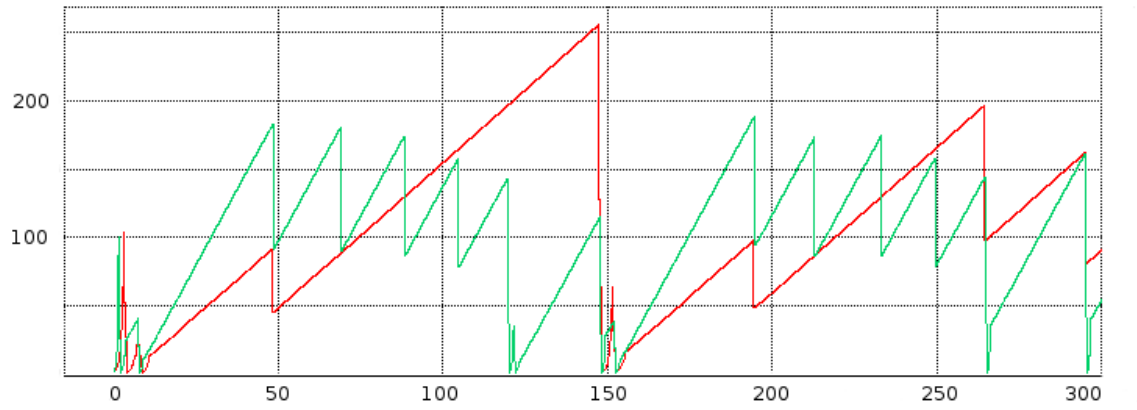


For this graph, note that the red and green loss events at T=130 are not quite aligned; the same happens at T=45 and T=100.

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.

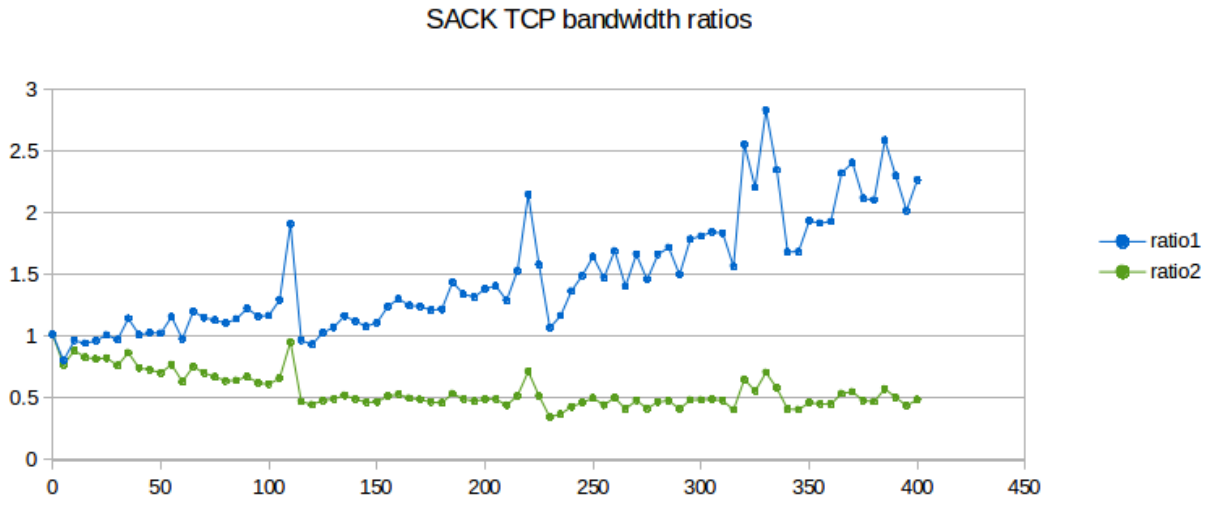
Clusters of multiple loss responses also vanish almost entirely. In these 3000-second runs, there are usually 1-2 cases where two loss responses occur within 1.0 seconds (over 4 RTTs for the A–D connection) of one another.

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
```

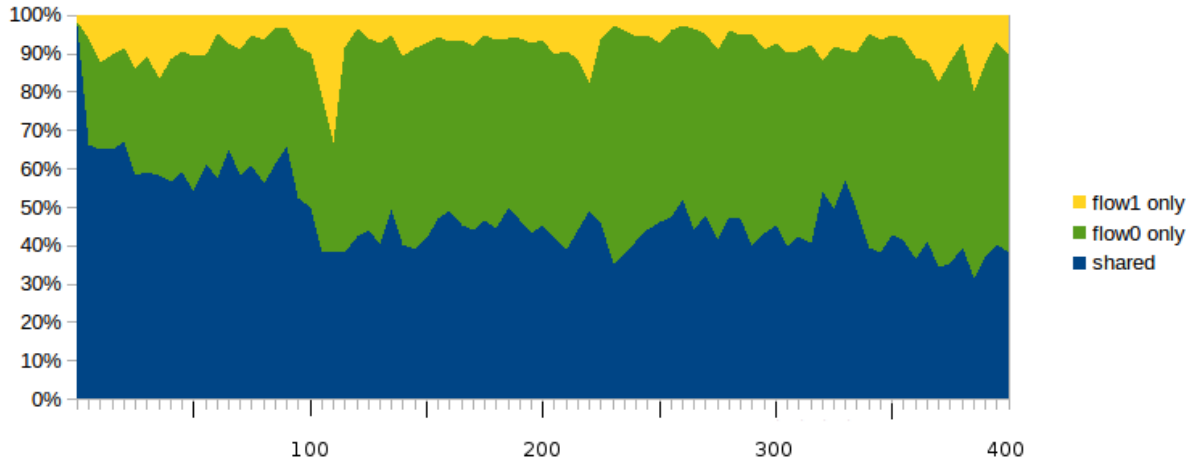
```
class flowstats:                # python object to hold per-flow data
   def __init__(self):
       self.toothcount = 0
       self.prevcwnd = 0
       self.CTOcount = 0                # Coarse TimeOut count

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.

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` ≤ 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.

16.4.4 Clusters of Packet Drops

During a multiple-packet-loss event involving two flows, just how many packets are lost from each flow? Are these losses usually divided evenly between the flows? The synchronized-loss hypothesis simply requires that each flow have at least one loss; it says nothing about how many.

As discussed in *16.4.2.1 Counting teeth in Python*, we can use the tracefile to count clusters of packet drops, given a suitable granularity value. We can also easily count the number of packet drops, for each flow, in each cluster. Each packet loss is indicated by an ns-2 event-trace line beginning with "d". In our simple arrangement, all losses are at the router R and all losses are of data (rather than ACK) packets.

At each drop cluster (loss event), each flow loses zero or more packets. For each pair of integers (X,Y) we will count the number of drop clusters for which flow 0 experienced X losses and flow 1 experienced Y losses. In the simplest model of synchronized losses, we might expect the majority of drop clusters to correspond to the pair (1,1).

Whenever we identify a new drop cluster, we mark its start time (`dcstart` below); one cluster then consists

of all drop events from that time until one granularity interval of length `DC_GRANULARITY` has elapsed, and the next loss after the end of that interval marks the start of the next drop cluster. The granularity interval must be larger than the coarse-grained timeout interval and smaller than the tooth-to-tooth interval. Because the timeout interval is often 1.0 sec, it is convenient to set `DC_GRANULARITY` to 1.5 or 2.0; in that range the clusters we get are not very sensitive to the exact value.

We will represent a drop cluster as a Python tuple (pair) of integers `(d0,d1)` where `d0` represents the number of packet losses for flow 0 and `d1` the losses for flow 1. As we start each new cluster, we add the old one to a Python dictionary `clusterdict`, mapping each cluster to the count of how many times it has occurred.

The code looks something like this, where `dropcluster` is initialized to the pair `(0,0)` and `addtocluster(c,f)` adds to cluster `c` a drop for flow `f`. (The portion of the script here fails to include the final cluster in `clusterdict`.)

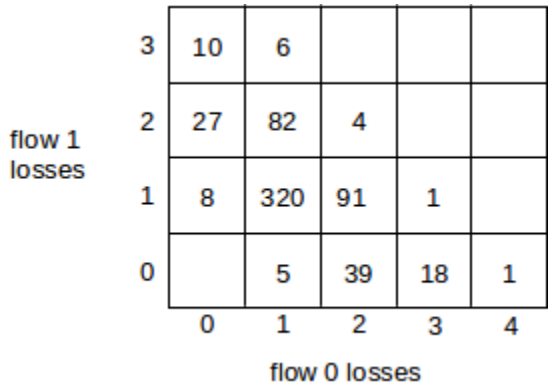
```
if nstrace.istrace():           # counting regular trace lines for DROP CLUSTERS
       (event, time, sendnode, dnode, proto, dummy, dummy, flow, dummy, dummy, seqno, pktic
       if (event=='d'):                                          # ignore all others
               if (time > dcstart + DC_GRANULARITY):             # save old cluster, start ne
                       dcstart = time
                       if (dropcluster != (0,0)):                # dropcluster==(0,0) means
                               inc_cluster(dropcluster, clusterdict)   # add dropcluster to
                       dropcluster = addtocluster((0,0), flow)         # start new cluster
               else:                               # add to dropcluster
                       dropcluster = addtocluster(dropcluster, flow)
```

At the end of the tracefile, we can print out `clusterdict`; if (f0,f1) had an occurrence count of N then there were N drop events where flow 0 had f0 losses and flow 1 had f1. We can also count the number of loss events that involved only flow 0 by adding up the occurrence counts of all drop clusters of the form (x,0), that is, with 0 as the number of flow-1 losses in the cluster; similarly for flow 1.

A full Python script for this is at teeth.py.

If we do this for our earlier *16.3 Two TCP Senders Competing* scenario, with `bottleneckBW` = 0.8 and with a runtime of 3000 seconds, we find out that **for equal paths**, the two connections have completely synchronized loss events (that is, no flow-0-only or flow-1-only losses) for `overhead` values of 0, 0.0012, 0.003, 0.0055 and 0.008 sec (recall the bottleneck-link packet send time is 0.01 sec). For all these overhead values but 0.008, each loss event does indeed consist of exactly one loss for flow 0 and one loss for flow 1. For `overhead` = 0.008, the final clusterdict is {(1,2):1, (1,1):674, (2,1):1}, meaning there were 674 loss events with exactly one loss from each flow, one loss event with one loss from flow 0 and two from flow 1, and one loss event with two losses from flow 0 and one from flow 1.

As `overhead` increases further, however, the final clusterdicts become more varied; below is a diagram representing the dictionary for `overhead` = 0.02.



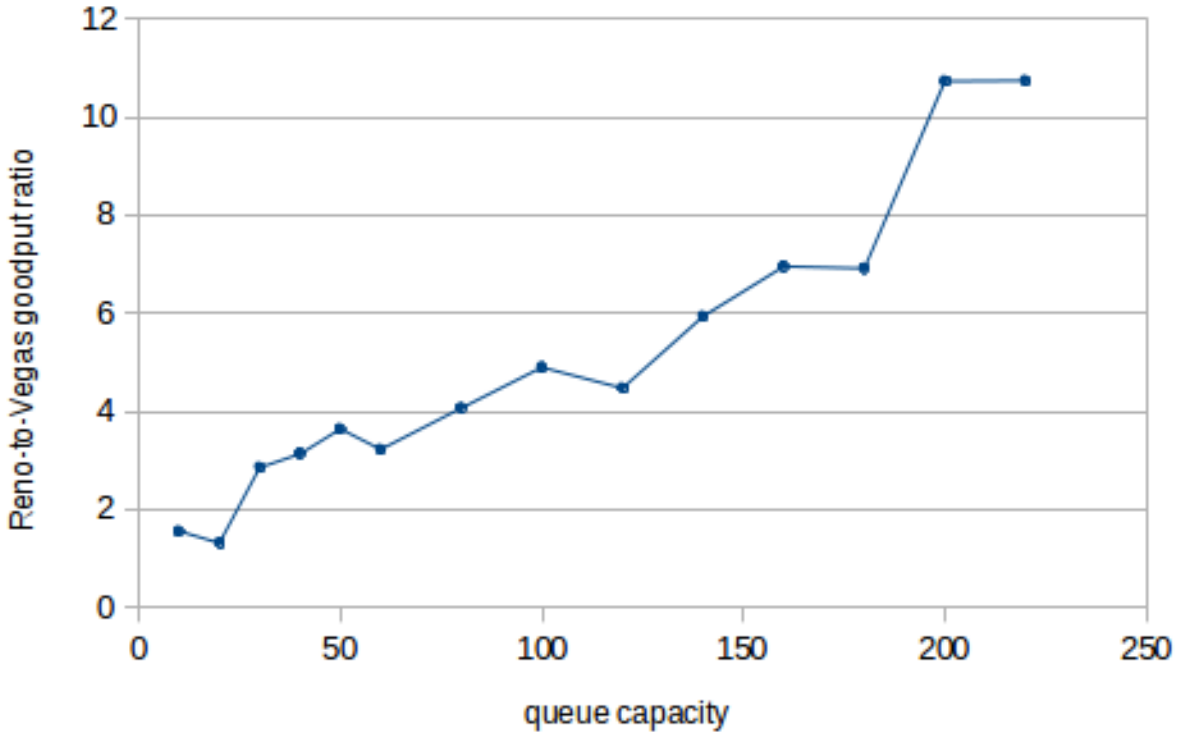
The numbers are the occurrence counts for each dropcluster pair; for example, the entry of 91 in row 1 column 2 means that there were 91 times flow 0 had two losses and flow 1 had a single loss.

16.5 TCP Reno versus TCP Vegas

In *15.4 TCP Vegas* we described an alternative to TCP Reno known as TCP Vegas; in *15.4.1 TCP Vegas versus TCP Reno* we argued that when the queue capacity 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 above with `bottleneckBW` = 8.0 Mbps, the transit capacity for the A–D path 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.

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
```

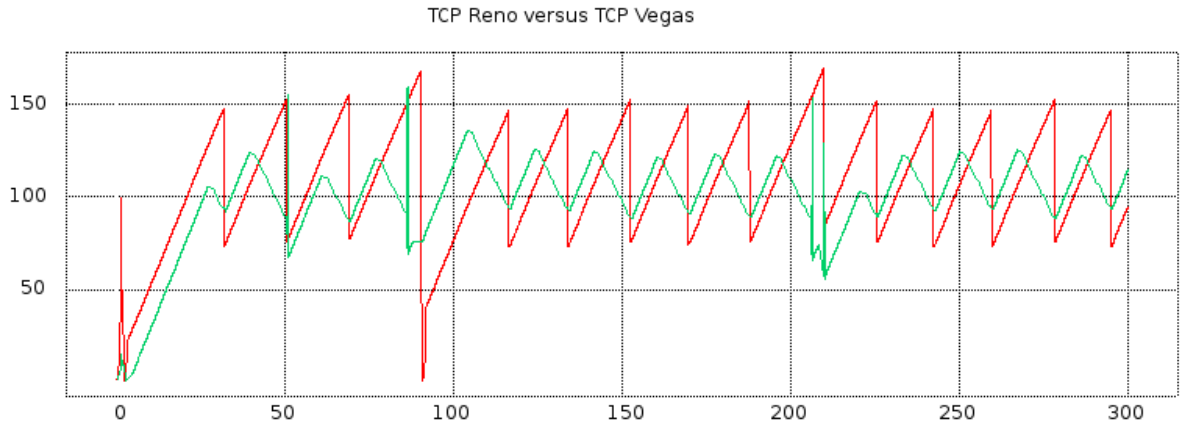
In prior simulations we have also made the following setting, in order to make the total TCP packetsize including headers be 1000:

```
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 frame of *16.6 Wireless Simulation*, is it plausible that what is rendered for the CBR packets represents just the first bit of the packet?

(d). What might be a more accurate animated representation of wireless packets?

17 QUEUING AND SCHEDULING

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 *18 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 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 gateways*) 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.

17.1 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.

17.2 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.

17.3 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.

17.4 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 *18 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 *17.9 Token Bucket Filters* below for an example.

17.5 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. 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.

17.5.1 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, *17.5.4 The GPS Model*, does provide an algorithm that has direct, natural support for weighted fair queuing.

17.5.2 Different Packet Sizes

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.

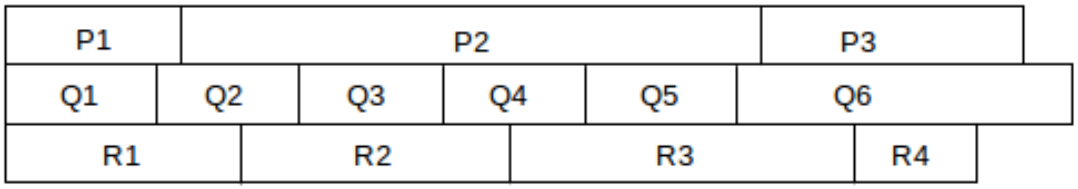
We present two mechanisms for handling different-sized packets; the two are ultimately equivalent. The first – *17.5.3 Bit-by-bit Round Robin* – is a straightforward extension of the round-robin idea, and the second – *17.5.4 The GPS Model* – uses a "fluid" model of simultaneous packet transmission. Both mechanisms share the idea of a "virtual clock" that runs at a rate 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 – *17.5.5 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.

17.5.2.1 A first virtual-finish example

As an example, suppose there are two subqueues, Q_1 and Q_2. Suppose further that a stream of 1001-byte packets P_1, P_2, P_3, ... arrives at Q_1, and a stream of 400-byte packets R_1, R_2, R_3, ... arrives for Q_2; 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 Q_1 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_2 the finishing times would be R_1 at 400, R_2 at 800, R_3 at 1200, etc. So the order of transmission of all the packets together will be as follows:

Packet	virtual finish	actual finish
R_1	400	400
R_2	800	800
P_1	1001	1801
R_3	1200	2201
R_4	1600	2601
R_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.

The next step is to extend this strategy to the case where subqueues can sometimes be inactive, using the model of bit-by-bit round robin.

17.5.3 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, 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 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 $R_{prev} + M/K$.

17.5.3.1 BBRR example

As an example, suppose the fair queuing router has three input subqueues Q_1, Q_2 and Q_3, initially empty. The following packets arrive at the wall-clock times shown.

Packet	Queue	Size	Arrival time, t
P_{11}	Q_1	1000	0
P_{12}	Q_1	1000	0
P_{21}	Q_2	600	800
P_{22}	Q_2	400	800
P_{23}	Q_2	400	800
P_{31}	Q_3	200	1200
P_{32}	Q_3	200	2100

At t=0, we have R(t)=0 and we assign finishing R-values F_{11}=1000 to P_{11} and $F_{12} = F_{11}$+1000 = 2000 to P_{12}. Transmission of P_{11} begins.

When the three Q_2 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 P_{21}, P_{22} and P_{23} of F_{21} = 800+600 = 1400, F_{22} = 1400+400 = 1800, and F_{23} = 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_{11} is completed; R(t) is 800 + 200/2 = 900. The smallest finishing R-value on the books is F_{21}, at 1400, so P_{21} is the second packet transmitted. P_{21}'s real finishing time will be t = 1000+600 = 1600.

At t=1200, P_{31} arrives; transmission of P_{21} is still in progress. R(t) is 800 + 400/2 = 1000; we calculate F_{31} = 1000 + 200 = 1200. Note this is less than the finishing R-value for P_{21}, which is currently transmitting, but P_{21} is not preempted. At this point (t=1200, R(t)=1000), the R(t) rate falls to 1/3.

At t=1600, P_{21} has finished transmission. We have R(t) = 1000 + 400/3 = 1133. The next smallest finishing R-value is F_{31} = 1200 so transmission of P_{31} commences.

At t=1800, P_{31} finishes. We have R(1800) = R(1200) + 600/3 = 1000 + 200 = 1200 (3 subqueues have been busy since t=1200). Q_3 is now empty, so the R(t) rate rises from 1/3 to 1/2. The next smallest finishing R-value is F_{22}=1800, so transmission of P_{22} begins. It will finish at t=2200.

At t=2100, we have R(t) = R(1800) + 300/2 = 1200 + 150 = 1350. P_{32} arrives on Q_3; it is assigned a finishing time of F_{32} = 1350 + 200 = 1550. Again, transmission of P_{22} is not preempted even though $F_{32} < F_{22}$. The R(t) rate again falls to 1/3.

At t=2200, P_{22} finishes. R(t) = 1350 + 100/3 = 1383. The next smallest finishing R-value is F_{32}=1550, so transmission of P_{32} begins.

At t=2400, transmission of P_{32} ends. R(t) is now 1350 + 300/3 = 1450. The next smallest finishing R-value is F_{12} = 2000, so transmission of P_{12} begins. The R(t) rate rises to 1/2, as Q_3 is again empty.

At t=3400, transmission of P_{12} ends. R(t) is 1450 + 1000/2 = 1950. The only remaining unsent packet is P_{23}, with F_{23}=2200. We send it.

At t=3800, transmission of P_{23} 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_{11}	0	1000	0	900
P_{21}	1000	1400	900	1133
P_{31}	1600	1200*	1133	1200
P_{22}	1800	1800	1200	1383
P_{32}	2200	1550*	1383	1450
P_{12}	2400	2000	1450	1950
P_{23}	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, P_{31} was not yet available when it was time to send P_{21}.

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.

17.5.4 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, $0 \leq i < N$, 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 $1/\alpha \geq 1$ 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 $VC(t) = N \times R(t)$; 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 *17.8.1.1 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 $0 \leq T \leq 10$, 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.

17.5.4.1 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.

17.5.4.2 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 *17.5.4.7 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.

17.5.4.3 GPS Example 1

As a first example we return to the scenario of *17.5.2.1 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 R_1, R_2, R_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
R_1	800	400
R_2	1600	800
P_1	2002	1801
R_3	2400	2201
R_4	3200	2601
R_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.

17.5.4.4 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 $A_i/\alpha = F_i$.

17.5.4.5 GPS Example 3

The next example illustrates a smaller but later-arriving packet, in this case P_4, that finishes ahead of P_3 under GPS but not under WFQ. P_3 can be said to *leapfrog* P_4 and P_5 under WFQ.

Suppose packets P_1 through P_5 arrive at R 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.

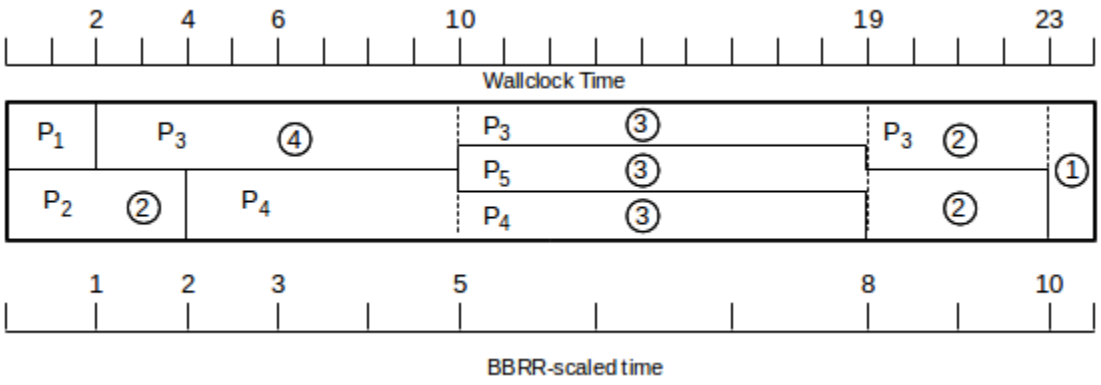
subqueue 1	subqueue 2	subqueue3
P_1: T=0, L=1	P_2: T=0, L=2	P_5: T=10, L=5
P_3: T=2, L=10	P_4: T=4, L=6	

Under WFQ, we send P_1 and then P_2; P_2 is second because its finishing time is later. When P_2 finishes the wallclock time is T=3. At this point, P_3 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 P_2 finishes under GPS at T=4 and P_4 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 P_4 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. P_5 arrives then; we calculate its BBRR finishing time as 5+5=10 and its GPS finishing time as $15+5/\alpha = 30$.

Because P_4 has the earlier virtual-clock finishing time, WFQ sends it next after P_3, followed by P_5.

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.

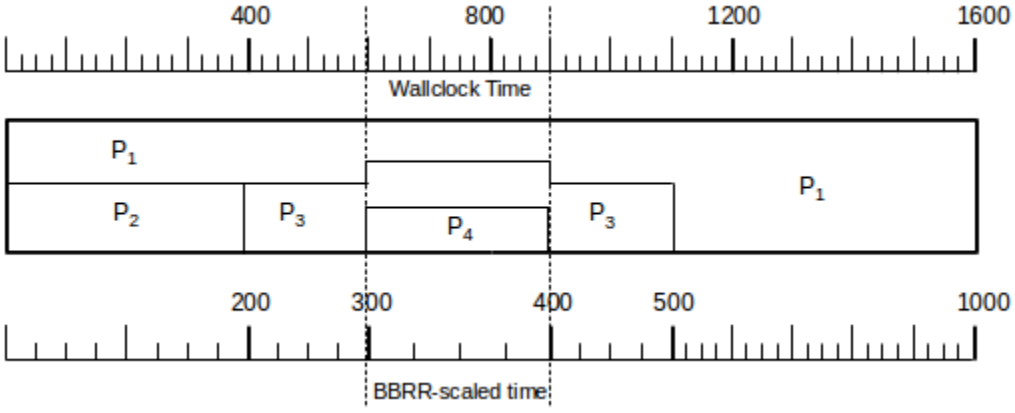
Note that, while the transmission order under WFQ is P_1, P_2, P_3, P_4, P_5, the actual finishing order under GPS is P_1, P_2, P_4, P_5, P_3. That is, P_3 managed to leapfrog P_4 and P_5 under WFQ by the simple expedient of being the only packet available for transmission at T=3.

17.5.4.6 GPS Example 4

As a second example of leapfrogging, suppose we have the following arrivals; in this scenario, the smaller but later-arriving P_4 finishes ahead of P_1 and P_3 under GPS, but not under WFQ.

subqueue 1	subqueue 2	subqueue3
P_1: T=0, L=1000	P_2: T=0, L=200	P_4: T=600, L=100
	P_3: 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 P_2, P_4, P_3, P_1.



Under WFQ, the transmission order is P_2, P_3, P_1, P_4, because when P_3 finishes at T=500, P_4 has not yet arrived.

17.5.4.7 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 *17.13.3 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

$$F_{WFQ} \leq F_{GPS} + L_{max}/r$$

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. 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 (*17.5.4.2 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 $F_k \leq G_k$.

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 *17.5.4.5 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

$$G_k - T_{start} \geq F_k - F_m$$
$$G_k - (F_m - L_m/r) \geq F_k - F_m$$
$$F_k \leq G_k + L_m/r \leq G_k + L_{max}/r$$

The last line represents the desired conclusion.

17.5.5 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 queue a **quantum**. At each round, a queue 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 per-class queue has a chance to send, it will be able to send at least one packet). Then, we again service the queues in round-robin fashion, but each time we take as many packets from the queue 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 *17.5.4.7 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 senders 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 queue.

The next time that queue 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 queue 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 an input queue 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.

17.5.6 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 is 1024). The buckets are then used as the fair queuing input queues.

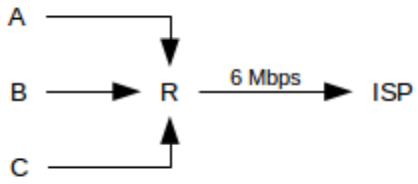
If two connections hash to the same bucket, each will get only half the bandwidth to which it is entitled. Therefore the hash function has some additional parameters that allow it to be *perturbed* at regular intervals; after perturbation, socketpairs that formerly hashed to the same bucket likely now will not.

Stochastic fair queuing is available in the linux kernel.

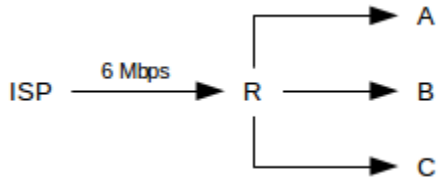
17.6 Applications of Fair Queuing

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 *17.10 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.

17.7 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.

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 with real links, to construct composite queuing structures in which packets are *not* stored only at leaf nodes, the resultant **internal-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 with real links, then either packets simply pile up at the root, or else the interior links become the bottleneck. For a case where the internal-storage construction does work, see the end of *17.12 Hierarchical Token Bucket*.

17.7.1 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.

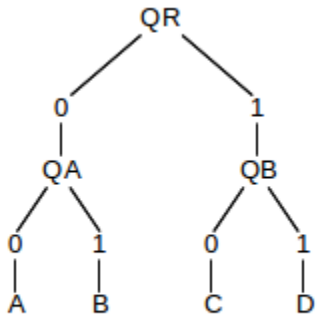
Many homogeneous queuing-discipline hierarchies support this generic mechanism. Priority queuing, as in the example below, works, because when a parent node needs to dequeue a packet it finds the highest-priority nonempty child subqueue, dequeues a packet from it, and sends it up the tree. In this case 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 *17.8.1 A Hierarchical Weighted Fair Queuing Algorithm*.

17.7.2 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.

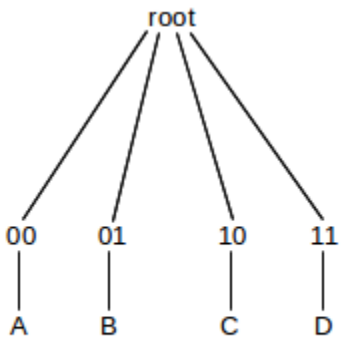


Generic Hierarchical Queuing

We now need a **classifier**: something that can 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:

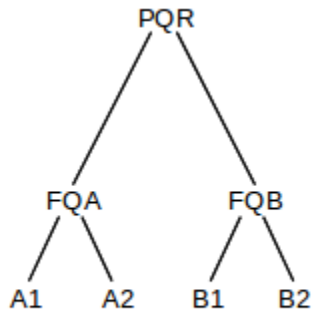
- at the root node, we first see if packets are available in the left (high-priority) subqueue, QA. If so, we dequeue the packet from there; if not, we go on to the right subqueue QB.
- at either QA or QB, we first see if packets are available in the left input, labeled 0; if so, we dequeue from there. 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.

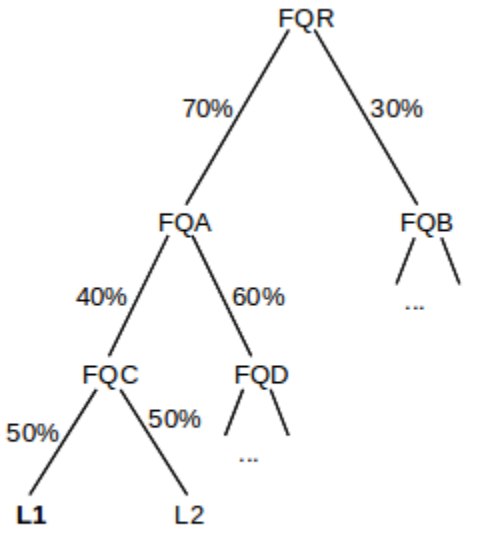
17.8 Hierarchical Weighted Fair Queuing

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 $0<k\leq r$. 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 $70\% \times 40\% \times 50\% = 14\%$ of the total bandwidth.

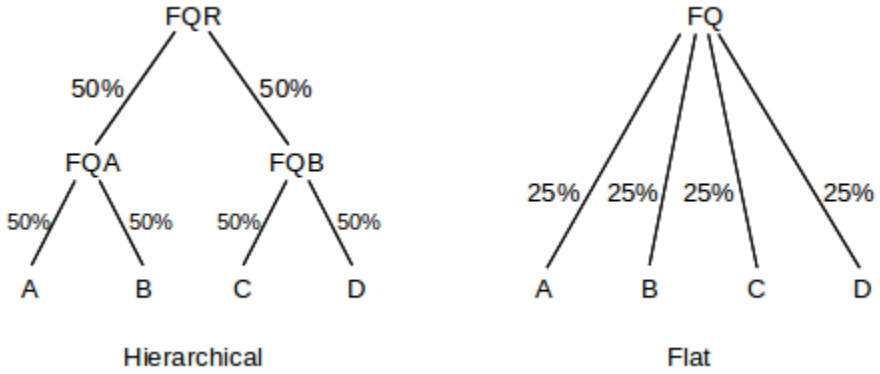


Hierarchical Weighted Fair Queuing
L1 is guaranteed 70%×40%×50% = 14%

If, however, node FQD becomes inactive, then FQA will assign 100% to FQC, in which case L1 will receive $70\% \times 100\% \times 50\% = 35\%$ 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.

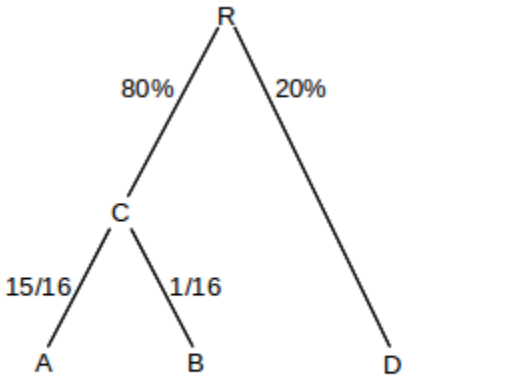
Hierarchical fair queuing – unlike priority queuing – does *not* collapse; the hierarchical queuing discipline is not equivalent to any one-level queuing discipline. Here is an example illustrating this. We form a tree of three fair queuing nodes, labeled FQR, FQA and FQB in the diagram. Each grants 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 *17.8.1.2 A Hierarchical-WFQ Example*.

17.8.1 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 *17.5.4 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.

17.8.1.1 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.

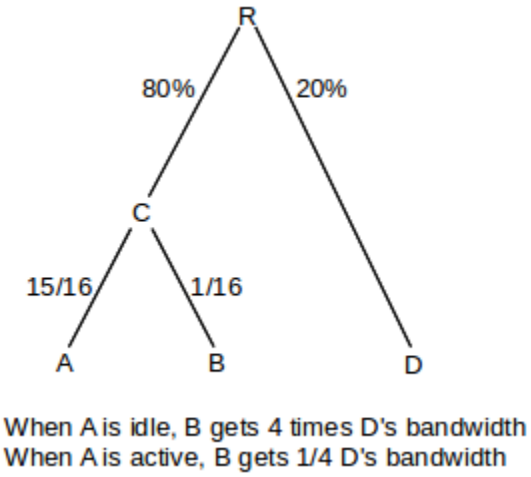
The active-subqueue notification is also exactly what is necessary for the WFQ node to maintain its virtual clock. If A is the set of active subqueues, and the ith subqueue has bandwidth share α_i, then the clock is to run at rate equal to the reciprocal of the sum of the α_i for i in A. This rate needs 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 *17.5.4 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].

17.8.1.2 A Hierarchical-WFQ Example

Let us consider again the example at the end of *17.8 Hierarchical Weighted Fair Queuing*:



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 *17.5.4.4 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 $8 \times 16 = 128$.

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 $9 \times 16 = 144$ 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.

17.9 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 *17.4 Queuing Disciplines* does not provide for marking packets, but this is a straightforward extension.

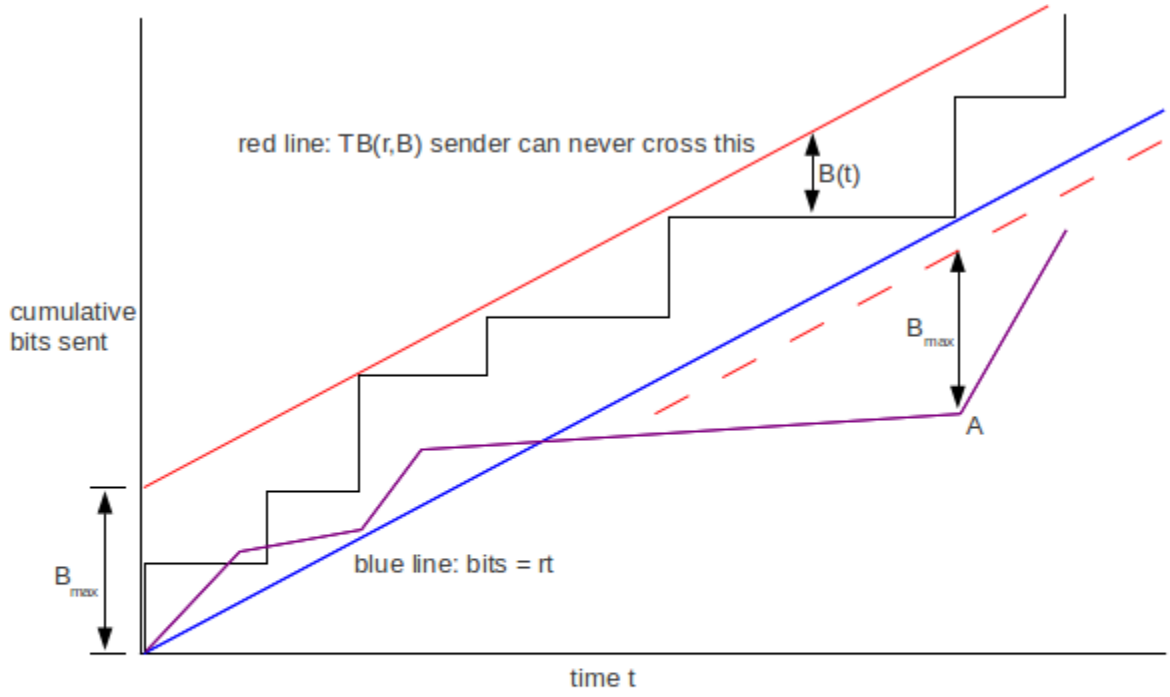
17.9.1 Token Bucket Definition

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$\geq$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 *17.11 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 (*3.8 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.

17.9.2 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

17.9.3 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 the packet arriving at T=2.5 had instead arrived at T=2, we would have 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.

17.9.4 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:

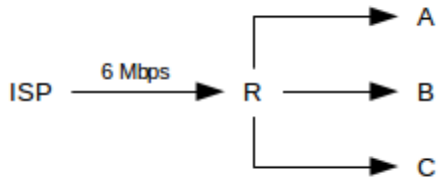
1. t $\geq$ tat–τ: the packet is compliant, and we update tat to max(t,tat) + T
2. t < tat–τ: the packet is too early and is noncompliant; tat is unchanged.

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.

17.10 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 *17.6 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.

17.10.1 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 *17.6 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.

17.11 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 *17.9.1 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:

bits(t) <= 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.

17.11.1 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.

17.11.2 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.

17.11.3 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.

17.12 Hierarchical Token Bucket

Token-bucket filters can also be used to form a hierarchy, as in *17.7.1 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.

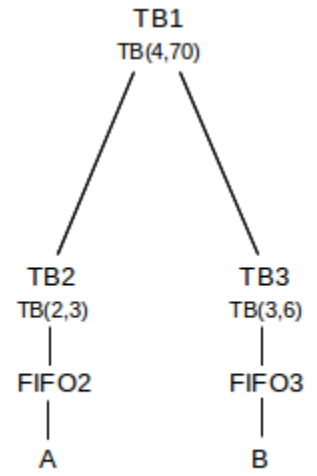
To make token-bucket filters classful, we will assume that each node may have multiple input subqueues, but treats these as if they were consolidated into a single FIFO subqueue. That is, the node releases packets to its parent in the order they were released to the node by its children.

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 *17.7.1 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 *17.7 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 $2\times(4/5)$ and $3\times(4/5)$ 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.

17.13 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.

17.13.1 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.

17.13.2 Linux htb

The linux **htb** queuing discipline 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 avoided some technical timing issues that CBQ users had to watch out for. For efficiency, htb uses the quantum

algorithm for fair queuing; as noted in *17.5.5 The Quantum Algorithm*, this means less precise control over packet delay.

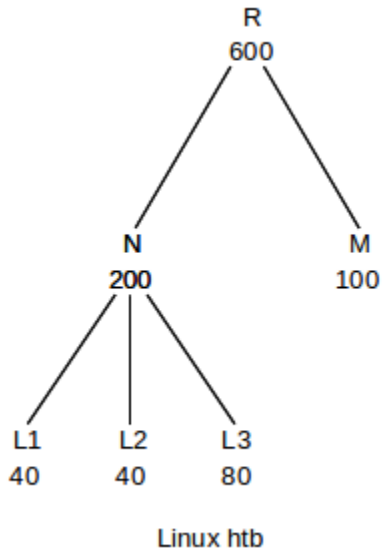
It is common to arrange for htb to apply token-bucket shaping only at the leaf nodes, and to configure the interior nodes do fair queuing only.

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.

The most important attribute of each node is its guaranteed rate. If inner nodes are not doing token-bucket shaping then the 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. If the parent-node rate is at least as large as the sum of the child-node rates, the parent node will not be a bottleneck.

After L1, L2 and L3 have each emptied their buckets, they will not exhaust N's rate. Similarly, after N and M have emptied their buckets they will use only half of R's rate. Packets are allowed to "borrow" – without payback – from their parent's rates; such packets 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 *17.12 Hierarchical Token Bucket*.

17.13.3 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 $f_i \leq r_i$. 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 *17.11.2 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 *17.5.4.7 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.

17.14 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 *18 Quality of Service* we will encounter fair queuing and token-bucket specifications repeatedly.

17.15 Exercises

1. Suppose a router uses fair queuing with three input classes, and uses the quantum algorithm of *17.5.2 Different Packet Sizes*. 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

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 *17.5.5 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 *17.9.3 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 $t_1 \le t \le t_2$ the amount of traffic is at most $B + r \times (t_2 - t_1)$. Hint: during the interval $t_1 \le t \le t_2$ 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 *17.7.1 Generic Hierarchical Queuing*, collapses to a single FIFO queue.

18. Show that the token-bucket leaf-storage hierarchy of *17.12 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?

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 $\alpha \leq 10\%$), 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%

18 QUALITY OF SERVICE

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, at least some senders may wish to arrange in advance for a certain minimum level of network services. 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 *17 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**, not to be confused with the specific IETF protocol suite of that name (*18.4 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 *17 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 *18.3.3 UDP and Real-Time Traffic*.

18.1 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.

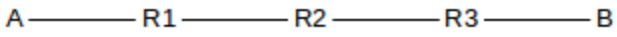
Without taking sides on the net-neutrality debate, there are good reasons for arguing that additional charges for specific services from backbone routers are more appropriate than additional charges to avoid rate throttling.

18.2 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 is often still claimed that the Internet backbone has excess capacity, and therefore is seldom congested; delays, therefore, are more likely to be encountered more locally.

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.

18.3 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 *18.3.2 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.

18.3.1 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	rec'd	(rec'd – expected)
1	0	50	50	0
2	20	70	55	-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 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.

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.

18.3.2 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 (*10.6.5.3 MULTI_EXIT_DISC*) 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 *18.7 Differentiated Services*) that is reserved for N's traffic.

18.3.3 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.

18.4 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 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.

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.

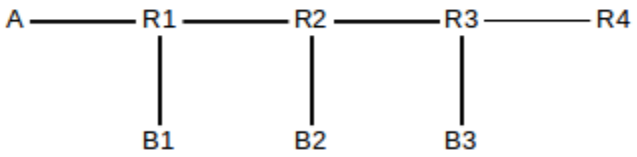
IntServ has mostly not been supported by the backbone ISP industry. Partly this is because of practical difficulties in figuring out how to charge for reservations; if the charge is zero then everyone might make reservations for every connection. Another issue is that a busy router might have to maintain 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 *18.9 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.

18.5 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 (*18.11 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.

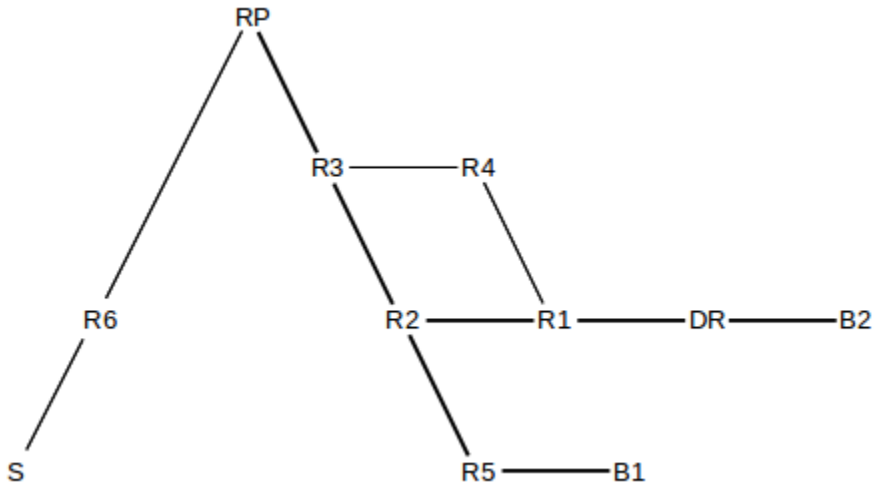
The multicast protocol most suited for these purposes is known as PIM-SM, defined in **RFC 2362**. PIM stands for **Protocol-Independent Multicast**, where "protocol-independent" means that it is not tied to a specific routing protocol such as distance-vector or link-state. SM here stands for **Sparse 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.

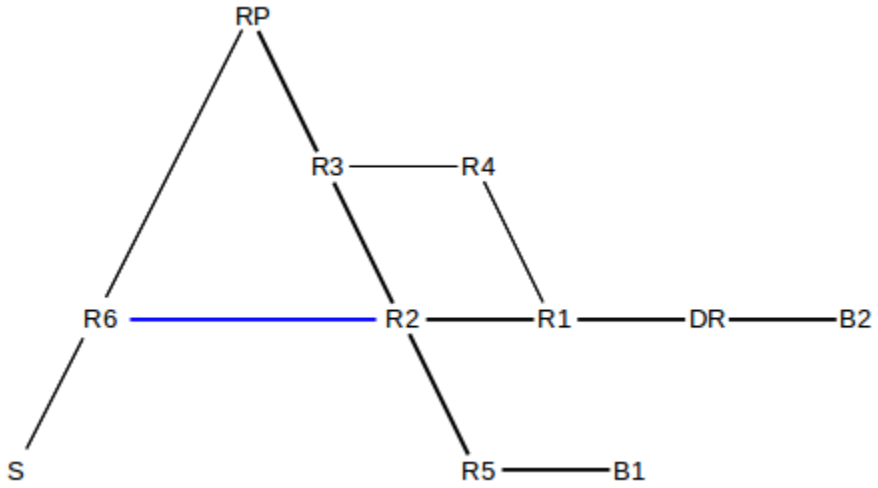
Now B2 wishes to join, and its designated router DR sends its join request along the path DR→R1→R2→R3→RP. R2 updates its entry for G to reflect that packets addressed to G are to be forwarded to the set {R5,R1}. In fact, R2 does not need to forward the join packet to R3 at all.

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 Provider-Based Hierarchical Routing*, 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.

18.6 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.

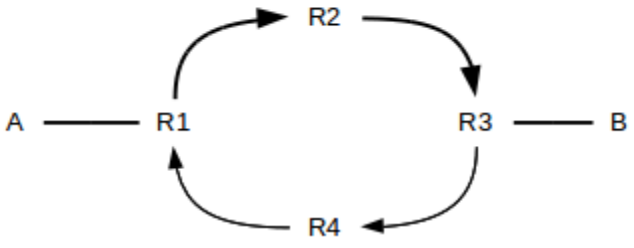
Like multicast, RSVP generally requires participation of intermediate routers.

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 filterspecs 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.

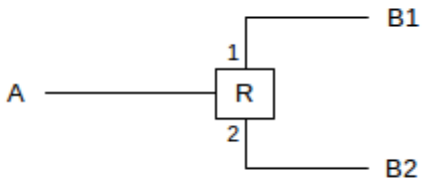
The PATH packet, in other words, tells the receiver what the Tspec is, and prepares the routers along the way for future reservations.

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 *18.2 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** (*18.11.1 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.

18.7 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 (*18.7.1 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 (*18.7.2 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.

18.7.1 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.

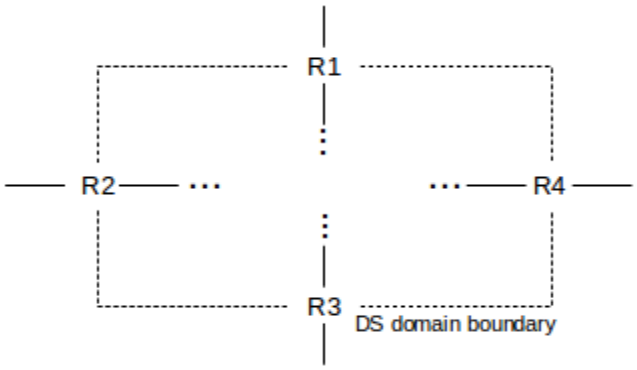
Note that this does *not* mean that EF traffic is given strict priority over all other traffic (though implementation of EF-traffic processing via priority queuing is a reasonable strategy); however, the sending interface must provide service to the EF queue at intervals of no more than L/R; the EF rate 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×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.

18.7.2 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.

18.8 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 gateways*. 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.

18.9 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.

18.10 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 (*18.7.2 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 (*18.8 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.

18.11 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.

18.11.1 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.

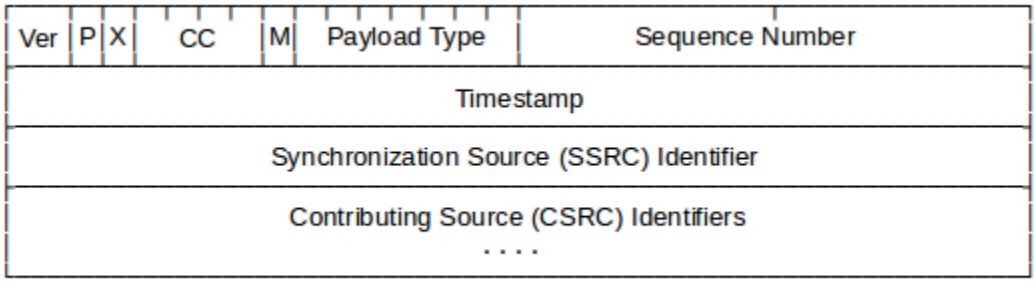
One thing mixers do *not* do is mix audio and video together; separate audio and video streams should be carried by separate RTP sessions. RTP packets carry timestamps to allow a receiver to synchronize audio and video playback, so mixing is not necessary. But more importantly, mixing of audio and video 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 pseudo-randomly (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.

18.11.2 RTP Packet Format

The basic RTP header has the following format, from **RFC 3550**:



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.

18.11.3 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 $\alpha = 15/16$:

$$J_i = \alpha \times J_{i-1} + (1-\alpha) \times Dev_i$$

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.

18.11.4 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 *18.3 Real-time Traffic*, some voice codecs – such as Opus and Speex – do support rate-adaptiveness.

18.12 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** (virtual circuits) 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.

In effect, virtual-circuit tags are added to each packet, on entrance to the routing domain, allowing packets to be routed along predefined virtual circuits. The virtual circuits 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.

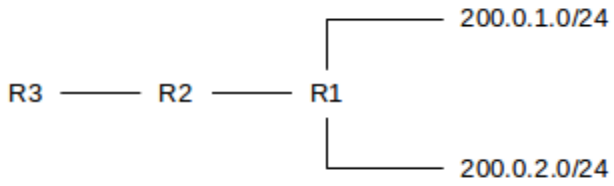
To implement MPLS, we start with a set of participating routers, called **label-switching routers** or LSRs. (The LSRs can comprise an entire ISP, or just a subset.) Edge routers partition (or *classify*) traffic into large flow classes; one distinct flow (which might, for example, correspond to all 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 (*3.8 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:

$\textbf{interface}_{\textbf{in}}$	$\textbf{label}_{\textbf{in}}$	$\textbf{interface}_{\textbf{out}}$	$\textbf{label}_{\textbf{out}}$
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 $\text{label}_{\text{out}}$, $\text{interface}_{\text{out}}$ 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 (*18.7 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.

18.13 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.

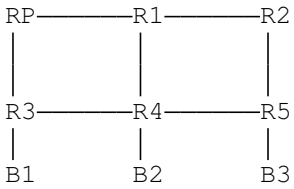
Still, after twenty years it is not unreasonable to ask whether integrated networks are in fact the correct approach. One school of thought says that real-time applications (such as VoIP) are only just beginning to come into the mainstream, and integrated networks are sure to follow, or else that video 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.

18.14 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 *18.7.1 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 *18.7.1 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.

BIBLIOGRAPHY

Note that RFCs are not included here.

INDICES AND TABLES

- *genindex*
- *search*

BIBLIOGRAPHY

[AP99] Mark Allman and Vern Paxson, "On Estimating End-to-End Network Path Properties", Proceedings of ACM SIGCOMM 1999, August 1999.

[PB62] Paul Baran, "On Distributed Computing Networks", Rand Corporation P-2626, 1962.

[JB05] John Bickett, "Bit-rate Selection in Wireless Networks", MS Thesis, Massachusetts Institute of Technology, 2005.

[BP95] Lawrence Brakmo and Larry Peterson, "TCP Vegas: End to End Congestion Avoidance on a Global Internet", IEEE Journal on Selected Areas in Communications, vol 13, no 8, 1995.

[BZ97] Jon Bennett and Hui Zhang, Hierarchical Packet Fair Queueing Algorithms, IEEE/ACM Transactions on Networking, vol 5, October 1997.

[CF04] Carlo Caini and Rosario Firrincieli, "TCP Hybla: a TCP enhancement for heterogeneous networks", International Journal of Satellite Communications and Networking, vol 22, pp 547-566, 2004.

[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.

[CJ89] Dah-Ming Chiu and Raj Jain, "Analysis of the Increase/Decrease Algorithms for Congestion Avoidance in Computer Networks", Journal of Computer Networks vol 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.

[DS78] Yogen Dalal and Carl Sunshine, "Connection Management in Transport Protocols", Computer Networks 2, 1978.

[DKS89] Alan Demers, Srinivasan Keshav and Scott Shenker, "Analysis and Simulation of a Fair Queueing Algorithm", ACM SIGCOMM Proceedings on Communications Architetures and Protocols, 1989.

[FF96] Kevin Fall and Sally Floyd, "Simulation-based Comparisons of Tahoe, Reno and SACK TCP", ACM SIGCOMM Computer Communication Review, July 1996.

[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, vol 3, pp 115-156, 1992.

[FJ93] Sally Floyd and Van Jacobson, "Random Early Detection Gateways for Congestion Avoidance", IEEE/ACM Transactions on Networking, vol 1, August 1993.

[FJ95] Sally Floyd and Van Jacobson, "Link-sharing and Resource Management Models for Packet Networks", IEEE/ACM Transactions on Networking, vol 3, June 1995.

[FP01] Sally Floyd and Vern Paxson, "Difficulties in Simulating the Internet", IEEE/ACM Transactions on Networking, vol 9, August 2001.

[FL03] Cheng Fu and Soung Liew, "TCP Veno: TCP Enhancement for Transmission over Wireless Access Networks", IEEE Journal on Selected Areas in Communications, vol 21 number 2, February 2003.

[LG01] Lixin Gao, "On Inferring Autonomous System Relationships in the Internet", IEEE/ACM Transactions on Networking, vol 9, December 2001.

[GR01] Lixin Gao and Jennifer Rexford, "Stable Internet Routing without Global Coordination", IEEE/ACM Transactions on Networking, vol 9, December 2001.

[JG93] Jose J Garcia-Lunes-Aceves, "Loop-Free Routing Using Diffusing Computations", IEEE/ACM Transactions on Networking, vol 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, vol 34 number 2, April 2004.

[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, vol 42 number 5, July 2008.

[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.

[CH99] Christian Huitema, *Routing in the Internet*, 2nd 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, vol 18(4), 1988.

[JWL04] Cheng Jin, David Wei and Steven Low, "FAST TCP: Motivation, Architecture, Algorithms, Performance", IEEE Infocom, 2004.

[SK88] Srinivasan Keshav, "REAL: A Network Simulator" (Technical Report), University of California at Berkeley, 1988

[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 methodolgies 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.

[MB76] Robert Metcalfe and David Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", Communications of the ACM, vol 19 number 7, 1976.

[MW00] Jeonghoon Mo and Jean Walrand, "Fair End-to-End Window-Based Congestion Control", IEEE/ACM Transactions on Networking, vol 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

[OKM96] Teunis Ott, JHB Kemperman and Matt Mathis, "The stationary behavior of ideal TCP congestion avoidance", 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, vol 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, vol 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.

[PB94] Charles Perkins and Pravin Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers", ACM SIGCOMM Computer Communications Review, vol 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.

[RJ90] Kadangode Ramakrishnan and Raj Jain, "A Binary Feedback Scheme for Congestion Avoidance in Computer Networks", ACM Transactions on Computer Systems, vol 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.

[SRC84] Jerome Saltzer, David Reed and David Clark, "End-to-End Arguments in System Design", ACM Transactions on Computer Systems, vol 2 number 4, November 1984.

[SM90] Nachum Shacham and Paul McKenney, "Packet recovery in high-speed networks using coding and buffer management", IEEE INFOCOM '90 Proceedings, June 1990.

[SV96] M Shreedhar and George Varghese, "Efficient Fair Queuing Using Deficit Round Robin", IEEE/ACM Transactions on Networking, vol 4 number 3, June 1996.

[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

[VGE00] Kannan Varadhan, Ramesh Govindan and Deborah Estrin, "Persistent Route Oscillations in Inter-domain Routing", Computer Networks, vol 32, January, 2000.

[WJLH06] David Wei, Cheng Jin, Steven Low and Sanjay Hegde, "FAST TCP: Motivation, Architecture, Algorithms, Performance", ACM Transactions on Networking, December 2006.

[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

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W