A Graduate Course in Computer Networks

Hari Balakrishnan
M.I.T.

Contents

List of Figures

List of Tables

■ Preface

These lecture notes are about one-half of an intended complete set of notes for a graduate-level course on computer networks. They include background material (Chapters 1 and 2) that should be largely known to most systems students, but are provided here as a refresher. In my experience, students whose main research interests are in other areas will find these background chapters especially useful.

The latter part of these notes is still incomplete, and includes material on wireless network protocols, overlay networks, peer-to-peer networks, networking in data centers and cloud environments, network security, measurement techniques, and musings on network architecture. Unfortunately, they are not yet in a form that can be distributed.

Please email any comments, suggestions, and criticisms on these notes to Hari Balakrishnan, hari@csail.mit.edu.

December 2011

Chapter 1
Connecting Computers with Packet Switching

This chapter discusses different ways of interconnecting links (of the same kind) to build a simple computer network. To achieve this task, we will use a device called a *switch*, and discuss several different ways of switching to move data across a network. We focus on *packet switching*, discussing its main ideas and principles. We then describe the key ideas in the design of a simple routing scheme for small networks, called "LAN switching" (or "bridging").

We'll assume here that the reader is familiar with standard ways of communicating digital information (bits and link-layer frames) over a single link.

■ 1.1 Interconnections

The rather limited scope—in terms of physical distance, number of connected hosts, and amount of sustainable traffic—of single-link networks leads us to examine ways of interconnecting single-link communication media together to form larger *multi-hop* networks of computers. The fundamental problem is that the most obvious way to build a computer network—by connecting each pair of computers with a dedicated link—is both prohibitively expensive (because of the sheer number of links required, a number that grows quadratically with the network size) and technically challenging (because signals attenuate with distance, requiring ways to regenerate information across large distances). The solution to these problems is to develop ways to *share* links between different communicating nodes, and to regenerate the information being communicated as it travels across the network.

The key component used for such interconnections is a *switch*, which is a specialized computer, often built with dedicated hardware (to process and forward data at high speeds), running "routing software". A switch receives bits that are encapsulated in *data frames* arriving over its links, processes them, and forwards them over one (or sometimes multiple) other links. In the most common kind of network, these frames are called *packets*, as explained below. Links are physically or wirelessly connected to devices (end hosts and switches) at *network attachment points*, also known as *network interfaces*; network interfaces at switches are also sometimes called *switch ports*.

The fundamental functions performed by switches are to multiplex and demultiplex data frames belonging to different device-to-device information transfer sessions (or "conversations"), and to determine the link(s) along which to forward any given data frame. This task is essential because a given physical link will usually be shared by several concurrent sessions between different devices. We break these functions into three problems:

1. **Forwarding:** When a data frame arrives at a switch, the switch needs to process it, determine the correct outgoing link, and decide when to send the frame on that link.

2. **Routing:** Each switch somehow needs to determine the topology of the network, so that it can correctly construct the data structures required for proper forwarding. The process by which the switches in a network collaboratively compute the network topology, adapting to various kinds of failures, is called routing. It does not happen on each data frame, but occurs in the "background".

3. **Resource allocation:** Switches allocate their resources—access to the link and local memory—to the different communications that are in progress.

Over time, two radically different techniques have developed for doing this. The first, used by networks like the telephone network, is called *circuit switching*. The second, used by networks like the Internet, is called *packet switching*.

There are two crucial differences between the two methods, one philosophical and the other mechanistic. The mechanistic difference is the easier one to understand, so we'll talk about it first. In a circuit-switched network, the frames do not (need to) carry any special information that tells the switches how to forward information, while in packet-switched networks, they do. The philosophical difference is more substantive: a circuit-switched network provides the abstraction of a *dedicated link* of some bit rate to the communicating entities, whereas a packet switched network does not.[1] Of course, this dedicated link traverses multiple physical links and at least one switch, so the end points and switches must do some additional work to provide the illusion of a dedicated link. A packet-switched network, in contrast, provides no such illusion; the end points and switches must do some work to provide reliable and efficient communication service to the applications running on the end points.

The transmission of information in circuit-switched networks usually occurs in three phases:

1. The *setup phase*, in which some state is configured at each switch along a path from source to destination,

2. The *data transfer phase* when the communication of interest occurs, and

3. The *teardown phase* that cleans up the state in the switches after the data transfer ends.

[1]One can try to layer such an abstraction atop a packet-switched network, but we're talking about the inherent abstraction provided by the network here.

Because the frames themselves contain no information about where they should go, the setup phase needs to take care of this task, and also configure (reserve) any resources needed for the communication so that the illusion of a dedicated link is provided. The teardown phase is needed to release any reserved resources.

A common (but not the only) way to implement circuit switching is using *time-division multiplexing (TDM)*, also known as *isochronous transmission*. Here, the physical capacity of a link connected to a switch, C (in bits/s), is conceptually broken into some number N of virtual "channels," such that the ratio C/N bits/s is sufficient for each information transfer session (such as a telephone call between two parties). Call this ratio, R, the *rate* of each independent transfer session. Now, if we constrain each frame to be of some fixed size, s bits, then the switch can perform time multiplexing by allocating the link's capacity in time-slots of length s/C units each, and by associating the ith time-slice to the ith transfer (modulo N). It is easy to see that this approach provides each session with the required rate of R bits/s, because each session gets to send s bits over a time period of Ns/C seconds, and the ratio of the two is equal to $C/N = R$ bits/s.

Each data frame is therefore forwarded by simply using the time slot in which it arrives at the switch to decide which port it should be sent on. Thus, the state set up during the first phase has to associate one of these channels with the corresponding soon-to-follow data transfer by allocating the ith time-slice to the ith transfer. The end computers transmitting data send frames only at the specific time-slots that they have been told to do so by the setup phase.

Other ways of doing circuit switching include *wavelength division multiplexing (WDM)*, *frequency division multiplexing (FDM)*, and *code division multiplexing (CDM)*; the latter two (as well as TDM) are used in some cellular wireless networks. Various networking textbooks (e.g., Tanenbaum, Peterson and Davie, etc.) describe these schemes in some detail.

Circuit switching makes sense for a network where the workload is relatively uniform, with all information transfers using the same capacity, and where each transfer uses a *constant bit rate (CBR)* (or near-constant bit rate). The most compelling example of such a workload is telephony, and this is indeed how most telephone networks today are architected. (The other reason for this design choice is historical; circuit switching was invented long before packet switching.)

Circuit switching makes sense for a network where the workload is relatively uniform, with all information transfers using the same capacity, and where each transfer uses a *constant bit rate* (or near-constant bit rate). The most compelling example of such a workload is telephony, where each digitized voice call might operate at 64 kbits/s. Switching was first invented for the telephone network, well before devices were on the scene, so this design choice makes a great deal of sense. The classical telephone network as well as the cellular telephone network in most countries still operate in this way, though telephony over the Internet is becoming increasingly popular and some of the network infrastructure of the classical telephone networks is moving toward packet switching.

However, circuit-switching tends to waste link capacity if the workload has a *variable bit rate*, or if the frames arrive in bursts at a switch. Because a large number of computer applications induce burst data patterns, we should consider a different link sharing strategy for computer networks. Another drawback of circuit switching shows up when the $(N+1)^{\text{st}}$ communication arrives at a switch whose relevant link already has the maximum number

(N) of communications going over it. This communication must be denied access (or admission) to the system, because there is no capacity left for it. For applications that require a certain minimum bit rate, this approach might make sense, but even in that case a "busy tone" is the result. However, there are many applications that don't have a minimum bit rate requirement (file delivery is a prominent example); for this reason as well, a different sharing strategy is worth considering.

Packet switching doesn't have these drawbacks.

■ 1.2 Packet switching

An attractive way to overcome the inefficiencies of circuit switching is to permit any sender to transmit data at any time, but yet allow the link to be shared. Packet switching is a way to accomplish this task, and uses a tantalizingly simple idea: add to each frame of data a little bit of information that tells the switch how to forward the frame. This information is usually added inside a *header* immediately before the payload of the frame, and the resulting frame is called a *packet*.[2] In the most common form of packet switching, the header of each packet contains the *address* of the destination, which uniquely identifies the destination of data. The switches use this information to process and forward each packet. Packets usually also include the sender's address to help the receiver send messages back to the sender. A simple example of a packet header is shown in Figure **??**. In addition to the destination and source addresses, this header shows a checksum that can be used for error detection at the receiver.

The "purest" form of packet switching uses a *datagram* as the unit of framing, with the header containing the *address* of the destination. This address uniquely identifies the destination of data, which each switch uses to forward the datagram. The second form of packet switching is *source routing*, where the header contains a complete sequence of switches, or complete *route* that the datagram can take to reach the destination. Each switch now has a simple forwarding decision, provided the source of the datagram provides correct information. The third form of packet switching is actually a hybrid between circuit and packet switching, and uses an idea called *virtual circuits*. Because it uses a header, we classify it as a packet switching technique, although its use of a setup phase resembles circuit switching. We now look at each of these techniques in more detail.

Packet switches usually require *queues* to buffer packets that arrive in bursts. We will spend a fair amount of time discussing approaches to managing congestion and queues when we discuss network resource management schemes.

■ 1.2.1 Datagrams

Here, the sender transmits datagrams that include the address of the destination in the header; datagrams also usually include the sender's address to help the receiver send messages back to the sender. The job of the switch is to use the destination address as a key and perform a lookup on a data structure called a *routing table* (or *forwarding table*; the distinction between the two is sometimes important and will be apparent in later chapters). This lookup returns an output port to forward the packet on towards the intended destination.

[2]Sometimes, the term *datagram* is used instead of (or in addition to) the term "packet".

While forwarding is a relatively simple lookup in a data structure, the harder question is determining how the entries in the routing table are obtained. This occurs in a background process using a *routing protocol*, which is typically implemented in a distributed manner by the switches. There are several types of routing protocols possible. For now, it is enough to understand that the result of running a routing protocol is to obtain routes in the network to every destination.

Switches in datagram networks that implement the functions described in this section are often called *routers*, especially when the datagrams are IP packets.

■ 1.2.2 Source routing

Whereas switches implemented routing protocols to populate their routing tables in the "pure" datagram networks of the previous section, the equivalent function of determining which paths to use could also be performed by each sender. When this is done, the network can implement *source routing*, where the sender attaches an entire (and complete) sequence of switches (or more helpfully, per-switch next-hop ports) to each packet. Now, the task of each switch is rather simple; no table lookups are needed. However, it does require each sender to participate in a routing protocol to learn the topology of the network.

People tend not to build networks solely using source routing because of the above reason, but many networks (e.g., the Internet) allow source routing to co-exist with datagram routing.[3]

■ 1.2.3 Virtual circuits

Virtual circuit (VC) switching is an interesting hybrid between circuit and packet switching—it combines the setup phase of circuit switching with the explicit header of packet switching. The setup phase begins with the source sending a special *signaling* message addressed to a destination, which traverses a sequence of switches on its way to the destination. Each switch associates a local *tag* (or *label*), on a per-port basis, with this signaling message and sets this tag on the message before forwarding it to the next switch. When a switch receives a signaling message on one of its input ports, it first determines what output port will take the packet to its destination. It then associates the combination of input port and incoming tag to an entry in a local table, which maps this combination to an output port and outgoing tag (unique per-output).

Data transfer does not use the destination address in the packet header, but uses these tags instead. The forwarding task at each switch now consists of a tag lookup step, which yields and output port and replacement tag, and a tag swapping step which replaces the tag in the packet header.

The reason for the replacement tag is simply to avoid confusion; if global tags were used, then each source would have to be sure that any tag it chooses is not currently being used in the network.

There are many examples of network technologies that employ virtual circuit switching, including Frame Relay and Asynchronous Transfer Mode (ATM). These networks differ in the details of the tag formats and semantics (and these tags are known by different names; e.g., in ATM, a tag is a combination of a VPI or Virtual Path Identifier and VCI

[3]It turns out that source routing isn't deployed widely in the Internet today for security reasons.

or Virtual Circuit Identifier, which can be thought of as a single tag whose structure is hierarchical), and in the details of how these tags are computed. "Multi-Protocol Label Switching" (MPLS) is a link technology-independent approach that network switches can use to implement tag switching. The general principle in all these systems is as explained in this section.

Proponents of virtual circuit switching argue that it is advantageous over datagram routing because of several reasons, including:

- "It allows routes between source and destination to be "pinned" to a fixed route, which allows network operators to provision and engineer their network for various traffic patterns."
- "Tag lookups are more efficient than more-complex lookups based on various fields in the datagram header."
- "Because there is an explicit setup phase, applications that (think they) need resource reservation (e.g., link bandwidth, switch buffer space) can use this signaling to reserve resources."

The above claims are quoted, because they are all quite controversial, except perhaps the first one. Virtual circuit switching is arguably more complex than datagram routing, requiring a separate signaling protocol to set up switch state, and does not handle link or route failures as naturally. (This signaling is in addition to another protocol that allows the switches to discover the network topology, as in datagram networks.) Furthermore, the more complex forwarding table lookups required in IP datagrams are now efficient to implement even at high speeds, and the speed advantages of tag switching appear non-existent. The rationale and mechanism for resource reservation has been hotly debated in the community and will continue to be for some more years! (We will discuss these issues in later lectures.)

Virtual circuit technologies are common in the Internet infrastructure, and are often used to connect two IP routers in a so-called *transport network*.[4] Transport networks are the "link-layer" over which IP packets between two routers are communicated; examples include switched Ethernet-based local-area networks, 802.11-based wireless networks, etc. One successful example of a tag-switched network technology is MPLS (multi-protocol label switching), which is used in many internet service provider (ISP) networks.

■ 1.2.4 Why Packet Switching Works: Statistical Multiplexing

Packet switching does not provide the illusion of a dedicated link to any pair of communicating end points, but it has a few things going for it:

1. It doesn't waste the capacity of any link because each switch can send any packet available to it that needs to use that link.
2. It does not require any setup or teardown phases and so can be used even for small transfers without any overhead.

[4]Not to be confused with transport protocols, which are used by end points to communicate with each other.

3. It can provide variable data rates to different communications essentially on an "as needed" basis.

At the same time, because there is no reservation of resources, packets could arrive faster than can be sent over a link, and the switch must be able to handle such situations. Switches deal with transient bursts of traffic that arrive faster than a link's bit rate using *queues*. We will spend some time understanding what a queue does and how it absorbs bursts, but for now, let's assume that a switch has large queues and understand why packet switching actually works.

Packet switching supports end points sending data at variable rates. If a large number of end points conspired to send data in a synchronized way to exercise a link at the same time, then one would end up having to provision a link to handle the peak synchronized rate for packet switching to provide reasonable service to all the concurrent communications.

Fortunately, at least in a network with benign, or even greedy (but non-malicious) sending nodes, it is highly unlikely that all the senders will be perfectly synchronized. Even when senders send long bursts of traffic, as long as they alternate between "on" and "off" states and move between these states at random (the probability distributions for these could be complicated and involve "heavy tails" and high variances), the aggregate traffic of multiple senders tends to smooth out a bit.[5]

■ 1.2.5 Absorbing bursts with queues

Queues are a crucial component in any packet-switched network. The queues in a switch absorb bursts of data: when packets arrives for an outgoing link faster than the speed of that link, the queue for that link stores the arriving packets. If a packet arrives and the queue is full, then that packet is simply dropped (if the packet is really important, then the original sender can always infer that the packet was lost because it never got an acknowledgment for it from the receiver, and might decide to re-send it).

One might be tempted to provision large amounts of memory for packet queues because packet losses sound like a bad thing. In fact, queues are like seasoning in a meal—they need to be "just right" in quantity (size). Too small, and too many packets may be lost, but too large, and packets may be excessively delayed, causing it to take *longer* for the senders to know that packets are only getting stuck in a queue and not being delivered.

So how big must queues be? The answer is not that easy: one way to think of it is to ask what we might want the maximum packet delay to be, and use that to size the queue. A more nuanced answer is to analyze the dynamics of how senders react to packet losses and use that to size the queue. Answering this question is beyond the scope of this course, but is an important issue in network design. (The short answer is that we typically want a few tens to ≈ 100 milliseconds of a queue size—that is, we want the queueing delay of a packet to not exceed this quantity, so the buffer size in bytes should be this quantity multiplied by the rate of the link concerned.)

Thus, queues can prevent packet losses, but they cause packets to get delayed. These delays are therefore a "necessary evil". Moreover, queueing delays are *variable*—different

[5]It's worth noting that many large-scale *distributed denial-of-service attacks* try to take out web sites by saturating its link with a huge number of synchronized requests or garbage packets, each of which individually takes up only a tiny fraction of the link.

packets experience different delays, in general. As a result, analyzing the performance of a network is not a straightforward task. We will discuss performance measures next.

■ 1.3 Network Performance Metrics

Suppose you are asked to evaluate whether a network is working well or not. To do your job, it's clear you need to define some metrics that you can measure. As a user, if you're trying to deliver or download some data, a natural measure to use is the time it takes to finish delivering the data. If the data has a size of S bytes, and it takes T seconds to deliver the data, the *throughput* of the data transfer is $\frac{S}{T}$ bytes/second. The greater the throughput, the happier you will be with the network.

The throughput of a data transfer is clearly upper-bounded by the rate of the slowest link on the path between sender and receiver (assuming the network uses only one path to deliver data). When we discuss reliable data delivery, we will develop protocols that attempt to optimize the throughput of a large data transfer. Our ability to optimize throughput depends more fundamentally on two factors: the first factor is the *per-packet delay*, sometimes called the per-packet *latency* and the second factor is the *packet loss rate*.

The packet loss rate is easier to understand: it is simply equal to the number of packets dropped by the network along the path from sender to receiver divided by the total number of packets transmitted by the sender. So, if the sender sent S_t packets and the receiver got S_r packets, then the packet loss rate is equal to $1 - \frac{S_r}{S_t} = \frac{S_t - S_r}{S_t}$. One can equivalently think of this quantity in terms of the sending and receiving rates too: for simplicity, suppose there is one queue that drops packets between a sender and receiver. If the arrival rate of packets into the queue from the sender is A packets per second and the departure rate from the queue is D packets per second, then the packet loss rate is equal to $1 - \frac{D}{A}$.

The delay experienced by packets is actually the sum of four distinct sources: *propagation*, *transmission*, *processing*, and *queueing*, as explained below:

1. **Propagation delay.** This source of delay is due to the fundamental limit on the time it takes to send any signal over the medium. For a wire, it's the speed of light over that material (for typical fiber links, it's about two-thirds the speed of light in vacuum). For radio communication, it's the speed of light in vacuum (air), about 3×10^8 meters/second.

   The best way to think about the propagation delay for a link is that it is equal to the *time for the first bit of any transmission to reach the intended destination*. For a path comprising multiple links, just add up the individual propagation delays to get the propagation delay of the path.

2. **Processing delay.** Whenever a packet (or data frame) enters a switch, it needs to be processed before it is sent over the outgoing link. In a packet-switched network, this processing involves, at the very least, looking up the header of the packet in a table to determine the outgoing link. It may also involve modifications to the header of the packet. The total time taken for all such operations is called the processing delay of the switch.

3. **Transmission delay.** The transmission delay of a link is the time it takes for a packet

of size S bits to traverse the link. If the bit rate of the link is R bits/second, then the transmission delay is S/R seconds.

We should note that the processing delay adds to the other sources of delay in a network with *store-and-forward* switches, the most common kind of network switch today. In such a switch, each data frame (packet) is stored before any processing (such as a lookup) is done and the packet then sent. In contrast, some extremely low latency switch designs are *cut-through*: as a packet arrives, the destination field in the header is used for a table lookup, and the packet is sent on the outgoing link without any storage step. In this design, the switch *pipelines* the transmission of a packet on one link with the reception on another, and the processing at one switch is pipelined with the reception on a link, so the end-to-end per-packet delay is smaller than the sum of the individual sources of delay.

Unless mentioned explicitly, we will deal only with store-and-forward switches in this course.

4. **Queueing delay.** Queues are a fundamental data structure used in packet-switched networks to absorb bursts of data arriving for an outgoing link at speeds that are (transiently) faster than the link's bit rate. The time spent by a packet *waiting* in the queue is its queueing delay.

   Unlike the other components mentioned above, the queueing delay is usually variable. In many networks, it might also be the dominant source of delay, accounting for about 50% (or more) of the delay experienced by packets when the network is congested. In some networks, such as those with satellite links, the propagation delay could be the dominant source of delay.

■ 1.3.1 Little's Law

A common method used by engineers to analyze network performance, particularly delay and throughput (the rate at which packets are delivered), is *queueing theory*. In this course, we will use an important, widely applicable result from queueing theory, called *Little's law* (or Little's theorem).[6] It's used widely in the performance evaluation of systems ranging from communication networks to factory floors to manufacturing systems.

For any stable (i.e., where the queues aren't growing without bound) queueing system, Little's law relates the average arrival rate of items (e.g., packets), λ, the average delay experienced by an item in the queue, D, and the average number of items in the queue, N. The formula is simple and intuitive:

$$N = \lambda \times D \tag{1.1}$$

Example. Suppose packets arrive at an average rate of 1000 packets per second into a switch, and the rate of the outgoing link is larger than this number. (If the outgoing rate is smaller, then the queue will grow unbounded.) It doesn't matter how inter-packet arrivals are distributed; packets could arrive in weird bursts according to complicated

[6]This "queueing formula" was first proved in a general setting by John D.C. Little, who is now an Institute Professor at MIT (he also received his PhD from MIT in 1955). In addition to the result that bears his name, he is a pioneer in marketing science.

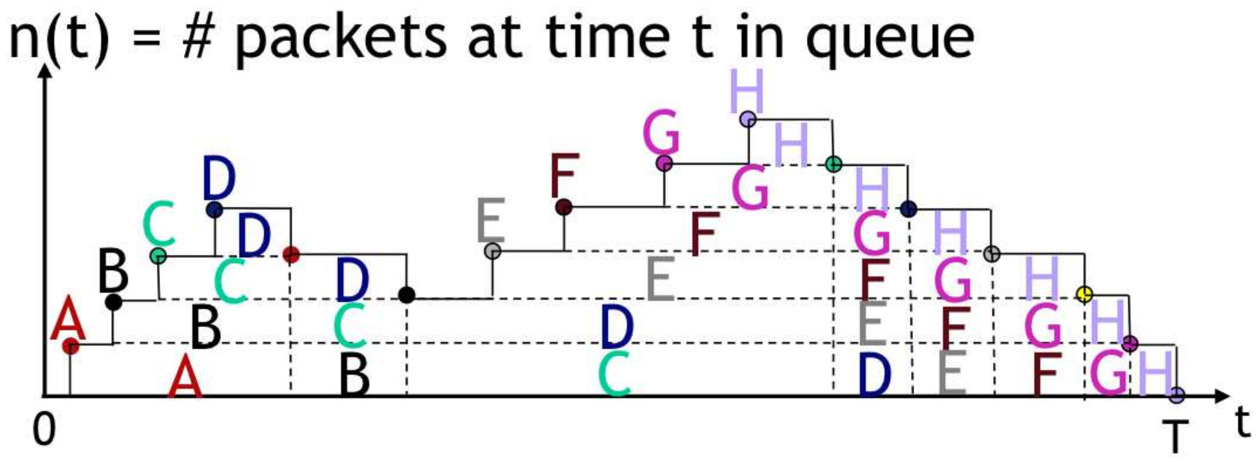


Figure 1-1: Packet arrivals into a queue, illustrating Little's law.

distributions. Now, suppose there are 50 packets in the queue on average. That is, if we sample the queue size at random points in time and take the average, the number is 50 packets.

Then, from Little's law, we can conclude that **the average queueing delay experienced by a packet is 50/1000 seconds = 50 milliseconds**.

Little's law is quite remarkable because it is independent of how items (packets) arrive or are serviced by the queue. Packets could arrive according to any distribution. They can be serviced in any order, not just first-in-first-out (FIFO). They can be of any size. In fact, about the only practical requirement is that the queueing system be stable. It's a useful result that can be used profitably in back-of-the-envelope calculations to assess the performance of real systems.

Why does this result hold? Proving the result in its full generality is beyond the scope of this course, but we can show it quite easily with a few simplifying assumptions using an essentially pictorial argument. The argument is instructive and sheds some light into the dynamics of packets in a queue.

Figure 1-1 shows $n(t)$, the number of packets in a queue, as a function of time t. Each time a packet enters the queue, $n(t)$ increases by 1. Each time the packet leaves, $n(t)$ decreases by 1. The result is the step-wise curve like the one shown in the picture.

For simplicity, we will assume that the queue size is 0 at time 0 and that there is some time $T >> 0$ at which the queue empties to 0. We will also assumes that the queue services jobs in FIFO order (note that the formula holds whether these assumptions are true or not).

Let P be the total number of packets forwarded by the switch in time T (obviously, in our special case when the queue fully empties, this number is the same as the number that entered the system).

Now, we need to define N, λ, and D. One can think of N as the *time average* of the number of packets in the queue; i.e.,

$$N = \sum_{t=0}^{T} n(t)/T.$$

The rate λ is simply equal to P/T, for the system processed P packets in time T.

D, the average delay, can be calculated with a little trick. Imagine taking the total area under the $n(t)$ curve and assigning it to packets as shown in Figure 1-1. That is, packets A, B, C, ... each are assigned the different rectangles shown. The height of each rectangle is 1 (i.e., one packet) and the length is the time until some packet leaves the system. Each packet's rectangle(s) last until the packet itself leaves the system.

Now, it should be clear that the time spent by any given packet is just the sum of the areas of the rectangles labeled by that packet.

Therefore, the average delay experienced by a packet, D, is simply the area under the $n(t)$ curve divided by the number of packets. That's because the total area under the curve, which is $\sum n(t)$, is the *total delay* experienced by all the packets.

Hence,

$$D = \sum_{t=0}^{T} n(t)/P.$$

From the above expressions, Little's law follows: $N = \lambda \times D$.

Little's law is useful in the analysis of networked systems because, depending on the context, one usually knows some two of the three quantities in Eq. (1.1), and is interested in the third. It is a statement about averages, and is remarkable in how little it assumes about the way in which packets arrive and are processed.

Let us now look at a concrete example of a switched network in more detail, using the widely deployed LAN (local-area network) switching technology as a case study.

■ 1.4 Case study: LAN switching (aka "bridging")

A single shared medium segment, like a single Ethernet, is limited by the number of stations it can support and by the amount of traffic that can be shared on it. To extend the reach of a single LAN segment requires some way of interconnecting many of them together. Perhaps the simplest way of extending LANs is via "LAN switching," a technique historically known as "bridging". Bridges (or LAN switches; we will use the two terms interchangably) are datagram switches that extend the reach of a single shared physical medium. They work by looking at data frames arriving on one segment, capturing them, and transmitting them on one or more other segments.

Another reason to study bridges is because they are a great example of self-configuring ("plug-and-play") networks.

Bridges are switches that are characterized by:

1. *Promiscuous store-and-forward* behavior with two or more ports. Each port has a unique identifier, or address, as does each network interface on computers attached to the LAN. "Promiscuous store-and-forward" means that any packet that arrives on a particular bridge port is replicated by the bridge and forwarded on all other ports to which LANs are currently attached.

   Each bridge port has at most one LAN attached to it (i.e., there could be bridges with no attached LANs), with each LAN in turn possibly containing other bridges. In such LANs implementing promiscuous store-and-forward behavior with no other mechanisms in place, the aggregate capacity does not exceed the capacity of the weakest

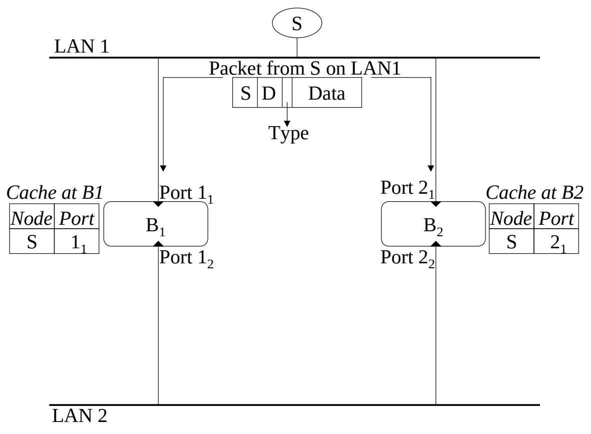


Figure 1-2: Learning bridges with loops. Such bridges suffer from packet duplication and proliferation.

segment, since any packet transmitted on any LAN appears on all others, including the slowest one.

2. *Learning*, wherein they learn which stations are on which LAN segments to forward packets more efficiently.

3. *Spanning tree construction*, by which bridge topologies with loops can form a loop-free topology and avoid packet duplication and implosion.

Bridges are *transparent* entities—they completely preserve the integrity of the packets they handle without changing them in any way. Of course, they may add a little to the delays experienced by packets and may occasionally (or frequently, under severe congestion) lose packets because they are, by nature, store-and-forward devices, but they are transparent to end-points in terms of the functionality they provide.

■ 1.4.1 Learning bridges

The basic idea of a learning bridge is that the bridge "learns", by building a *forwarding table*, which stations are downstream of which port. Then, when a packet destined to a given destination (MAC address) arrives, it knows which port to forward it on. How does it build this table? It learns by looking at the *source* address of packets it sees. When it associates the source address of a packet with a LAN segment, it adds that source-segment pair to the forwarding table.

If the table doesn't have an entry for some destination, the bridge simply floods the packet on all ports except the one the packet just arrived on. Thus, the forwarding table state maintained by a learning bridge is akin to a cache, and is used only as an optimization (albeit a very useful one). The consistency of this cache is not essential for correctness, because the absence of any information causes the packet to be flooded.

This strategy works well, except when there are *loops* in the network topology. In fact, in the absence of loops, not only does the above strategy handle routing in the static case, it also properly handles nodes that move in the topology from time to time (mobile nodes). If a node moves to another location in the switched network, the first time it sends data, the bridges along the (new) path to its destination cache the new point of attachment of the node that moved. Indeed, a variant of this method with a few optimizations is used in various wireless LAN access points to implement link-layer mobility. 802.11 ("WiFi") access points use this idea.

When there are loops in a switched LAN, significant problems arise. Consider the example shown in Figure 1.4.1, where bridges B_1 and B_2 have been configured to connect LAN1 and LAN2 together. (One reason for configuring such a network might be to add redundancy to the topology for reliability.) Consider a packet from source S transmitted on LAN1 for destination D. Since both B_1 and B_2 see this packet, they both pick it up and learn that node S is on LAN1, and add the appropriate information to their forwarding tables (caches) as shown in the picture. Then, if the bridges don't have any information in their tables for destination D, both bridges enqueue the packet for forwarding it on to LAN2. They compete for the channel (LAN2) according to the CSMA/CD protocol and one of them, say B_1, wins the contention race. When B_1 forwards the packet on to LAN2, the packet is seen by B_2.

B_2 has now seen a *duplicate* packet, but in addition, B_2 will believe that the source node S of this packet is on LAN2, when in fact it is on LAN1! The forwarding table has now been corrupted. However, B_2 would also enqueue the packet for transmission onto LAN1, because:

1. B_2 does not have an entry for D in its table, and
2. B_2 really has no way of telling that the packet is a duplicate short of carefully looking through every bit of each enqueued packet, a very inefficient process.

These problems are a direct consequence of one of the very reasons learning bridges are so attractive—their transparent behavior. Thus, the duplicate packet would continue to loop around forever (because bridges are transparent, there are no hop limit or time-to-live fields in the header).

But this looping isn't the worst part—packets in fact can *reproduce* over and over in some cases! To see how, add another bridge B_3 between the two LANs. Now, each time a bridge sends one packet on to a LAN, the two other bridges enqueue one packet *each* for the other LAN. It's not hard to see that this state of affairs will go on for ever and make the system unusable when bridges form loops.

There are several possible solutions to this problem, including: 1) avoiding loops by contruction, 2) detecting loops automatically and informing the network administrator to fix the problem when loops are found, or 3) making packet forwarding somehow work in the presence of loops. Clearly the last alternative is the most preferred one, if we can figure out how to do this.

The trick is to find a loop-free subset of the network topology. LAN switches use a *distributed spanning tree algorithm* for this task.

■ 1.4.2 The Solution: Spanning Trees

There are many distributed spanning tree algorithms, and in this course we'll encounter different ones in the context of unicast routing, wireless routing, multicast routing, and overlay networks. Bridges use a rooted spanning tree algorithm that generates the same tree as Dijkstra's shortest path trees. The idea is quite simple: bridges elect a root and form shortest paths to the root. The spanning tree induced by the union of these shortest paths is the final tree.

More specifically, the problem is as follows. For each bridge in the network, determine which of its ports should participate in forwarding data, and which should remain inactive, such that in the end each LAN has exactly one bridge directly connected to it on a path from the LAN to the root.

Viewing a network of LAN segments and LAN switches as a graph over which a spanning tree should be constructed is a little tricky. It turns out that the way to view this in order to satisfy the problem statement of the previous paragraph is to construct a graph by associating a node with each LAN segment and with each LAN switch. Edges in this graph emanate from each LAN switch, and connect to the LAN segment nodes they are connected to in the network topology, or to other LAN switches they are connected to. The goal is to find the subset of edges that form a tree, which span all the LAN segment nodes in the graph (notice that there may be LAN switch nodes that may be eliminated by this method; that is fine, because those LAN switches are in fact redundant).

The first challenge is to achieve this goal using a distributed, asynchronous algorithm, rather than using a centralized controller. The goal is for each bridge to independently discover which of its ports belong to the spanning tree, since it must forward packets along them alone. The result of the algorithm is a loop-free forwarding topology. The second challenging part is handling bridges that fail (e.g., due to manual removal or bugs), and arrive (e.g., new bridges and LAN segments that are dynamically attached to the network), without bringing the entire network to a halt.

Each bridge has a unique ID assigned by the network administrator (in practice, bridges have a vendor-specified unique ID, but administrators can set their own IDs to arrange for suitable trees to be built). Each port (network interface) on each bridge, as well as each network interface on end point computers has a unique vendor-specified ID.

Each bridge periodically, and asynchronous with the other bridges, sends *configuration messages* to all other bridges on the LAN. This message includes the following information:

```
[bridge_unique_ID] [bridge's_idea_of_root] [distance_to_root]
```

By consensus, the bridge with the smallest unique ID is picked as the root of the spanning tree. Each bridge sends in its configuration message the ID of the bridge that it thinks is the root. These messages are not propagated to the entire network, but only on each LAN segment; the destination address usually corresponds to a well-known link-layer address corresponding to "ALL-BRIDGES" and is received and processed by only the bridges on the LAN segment. Initially, each bridge advertises itself as the root, since that's the smallest ID it would have heard about thus far. The root's estimate of the distance to itself is 0.

At any point in time, a bridge hears of and builds up information corresponding to the smallest ID it has heard about and the distance to it. The distance usually corresponds to the number of other bridge ports that need to be traversed to reach the root. It stores the port on which this message arrived, the "root port" and advertises the new root on all its other ports with a metric equal to one plus the metric it has stored. Finally, it does not advertise any messages if it hears someone else advertising a better metric on the same

LAN. This last step is necessary to ensure that each LAN segment has exactly one bridge that configures itself to forward traffic for it. This bridge, called the *designated bridge* for the LAN, is the one that is closest to the root of the spanning tree. In the case of ties, the bridge with smallest ID performs this task.

It is not hard to see that this procedure converges to a rooted spanning tree if nothing changes for "a while." Each bridge only forwards packets on ports chosen as part of the spanning tree. To do this, it needs to know its root port and also which of its other ports are being used by other bridges as their preferred (or *designated*) root ports. Obtaining the former is trivial. Obtaining the latter is not hard either, since being a designated bridge for a LAN corresponds to this. When two bridges are directly connected to each other, each bridge views the other as a LAN segment.

The result of this procedure is a loop-free topology that is the same as a shortest-paths spanning tree rooted at the bridge with smallest ID.

Notice that the periodic announcements of configuration messages handle new bridges and LAN segments being attached to the network. The spanning tree topology reconfigures without bringing the entire network to a complete standstill in most cases.

This basic algorithm doesn't work when bridges fail. Failures are handled by timing information out in the absence of periodic updates. In this sense, bridges treat configuration announcements and their forwarding table entries as *soft state*. The absence of a configuration message from a designated bridge or the root triggers a spanning tree recalculation. Furthermore, an entry in the table that hasn't been used for a while may be timed out, resulting in packet flooding for that destination every once in a while. This approach trades of some bandwidth efficiency for robustness, and turns out to work well in practice in many networks.

The notion of "soft state" is an important idea that we will repeatedly see in this course, and is important to robust operation in a scalable manner. Together, periodic announcements to refresh soft state information inside the network enable *eventual consistency* to a loop-free spanning tree topology using the algorithm described above.

■ 1.4.3 Virtual LANs

As described thus far, switched LANs do not scale well to large networks. The reasons for this include the linear scaling behavior of the spanning tree algorithm, and the fact that all broadcast packets in a switched LAN must reach all nodes on all connected LAN segments. *Virtual LANs* improve the scaling properties of switched LANs by allowing a single switched LAN to be partitioned into several separate virtual ones. Each virtual LAN is assigned a "color," and packets are forwarded by a LAN switch on to another only if the color matches. Thus, the algorithms described in the previous section are implemented over each color of the LAN separately, and each port of a LAN switch is configured to some subset of all the colors in the entire system.

■ 1.5 Summary

Switches are specialized computers used to interconnect single-link communication media to form bigger networks. There are two main forms of switching—circuit switching and packet switching. Packet switching comes in various flavors, as do switched networks themselves. We studied some features of one of them, switched LANs.

This lecture illustrated two key ideas that we will encounter time and again:

- Soft state maintained by network elements.
- Distributed, asynchronous algorithms (spanning tree constuction in the case of LAN switches).

While LAN switches work well, they aren't enough to build a global network infrastructure. The primary reason for this is poor scalability. This approach may scale to a network with thousands of nodes (perhaps), but not to larger networks. The first scaling problem us caused by each LAN switch having to maintain per-destination information in its forwarding table. The second problem is due to the occasional flooding that's required.

A second reason for the approach described in this lecture being unattractive for a global network is that the approach may not work well over heterogeneous link technologies, many of which don't resemble Ethernet.A method for interconnecting heterogeneous link technologies is needed. This interface is what IP, the Internet Protocol, provides.

IP solves the "internetworking problem" of connecting different networks together. In the coming lectures, we will investigate its design.

Chapter 2
Connecting Networks of Networks: The Internetworking Problem

These notes discuss the key ideas that go into the design of large networks. Using the Internet and its network protocol, IP, as an example, we will discuss some important design principles and the techniques that have enabled the Internet to scale to many millions of connected and heterogeneous networks. Many of these principles were articulated in Clark's retrospective paper on the design of the Internet protocols [Clark88].

We'll start by describing the "internetworking problem". The two sections following the description of the problem discuss ways of handling heterogeneity and important techniques to achieve scalability. These are two of the three important characteristics of an internetwork; the third is network autonomy, which is the topic for the next lecture. We then highlight some key features of IPv4 and IPv6, and in an appendix mention the key ideas in Cerf and Kahn's seminal paper on TCP.

■ 2.1 The Internetworking Problem: Many Different Networks

The previous chapter discussed the principles underlying packet switching, and developed a scheme for connecting local-area networks (LANs) together. The key idea was a device called a switch, which forwarded packets between different LANs.

Building a global data networking infrastructure requires much more than just connecting a set of Ethernet links with switches. Such a strawman has three main failings:

1. *They don't accommodate heterogeneity.*

2. *They don't scale well to large networks.*

3. *They assume a single administrative owner.*

Each point bears some discussion.

Heterogeneity. Although common, Ethernets are by no means the only network link technology in the world. Indeed, even before Ethernets, packets were being sent over other kinds of links including optical fiber, various kinds of radio, satellite communication, and telephone links. Ethernets are also usually ill-suited for long-haul links that have to traverse hundreds of miles. Advances in link technologies of various kinds have always been occurring, and will continue to occur, and our goal is to ensure that *any* new network link technology can readily be accommodated into the global data networking infrastructure.

Heterogeneity manifests itself in several ways, including:

- **Data rates & latency.** Heterogeneity in link data rates range from a small number of kbits/s or lower (low-speed packet radio) to many Gigabits/s, spanning many orders of magnitudes. Similarly, latencies can range from microseconds to several seconds.
- **Loss rates.** Networks differ widely in the loss rates and loss patterns of their links.
- **Addressing.** Each network might have its own way of addressing nodes; for example, Ethernets use a 6-byte identifier, telephones use a 10-digit number, etc.
- **Packet size.** The maximum packet sizes in general will vary between networks.
- **Routing protocol.** Packet routing could be handled differently by each constituent network, e.g., packet radio networks implement different routing protocols from wireline networks.

Scalability. The LAN switching solution described in the previous chapter requires switches to store per-node state in the network; in the absence of such state for a destination, switches revert to flooding the packet to all the links in the network. This approach does not scale to more than a moderate number of hosts and links. As we'll see, the key to designing ever larger networks is to figure out ways to manage state and bandwidth efficiently: for example, schemes that require per-node state or that end up sending messages through the entire network upon each change (e.g., link failure near the "edge" of the network) don't scale well.

Our scalability goal is to reduce the state maintained by the switches and the bandwidth consumed by control traffic (e.g., routing messages, periodic flooding of packets/messages, etc.) as much as we can.

Autonomy. For many years now, no single entity has run the Internet. Today, many thousand independent organizations operate different pieces of the global infrastructure. These include Internet Service Providers (ISPs) that offer service to customers, and the customer networks themselves. Both ISP and customer networks are quite complex in many cases. These autonomous organizations often have different objectives: for example, an ISP might favor traffic to or from its own customers over other kinds of traffic. The lack of a single objective, or even a set of common objectives, implies that we can't expect Internet routing to be as straightforward as finding shortest paths according to a given routing metric.

Independent networks need to cooperate to provide global connectivity, but at the same time they compete in many cases (e.g., ISPs in a region often compete for the same customers). Designing protocols that can accommodate this *competitive cooperation* is the primary goal of wide-area (interdomain) Internet routing.

We use the name *internetwork* to denote large networks that are made up of many different smaller networks. The best example of an internetwork is the Internet; because the Internet is so dominant, people rarely use the term "internetwork" any more.

The "internetworking problem" may be stated as follows:

> Architect a scalable network that interconnects different smaller networks built using many different link technologies, owned and operated by different indepenent entities, to enable data to be sent between hosts connected to the constituent networks.

The rest of this chapter discusses heterogeneity and scalability. The next chapter discusses autonomy and competitive cooperation in the context of interdomain routing.

■ 2.2 Handling Heterogeneity: A Universal Network Layer

Communication between different networks in an internetwork is facilitated by entities called *gateways.* Gateways interface between two networks with potentially differing characteristics and move packets between them. There are at least two ways of interfacing between different networks: *translation* and a *unified network layer*.

■ 2.2.1 Translation Gateways Considered Harmful

Translation-based gateways work by translating packet headers and protocols over one network to the other, trying to be as seamless as possible. Examples of this include the OSI X.25/X.75 protocol family, where X.75 is a translation-based "exterior" protocol between gateways.

There are some major problems with translation.

- **Feature deficiency.** As network technologies change and mature and new technologies arise, there are features (or bugs!) in them that aren't preserved when translated to a different network. As a result, some assumption made by a user or an application breaks, because a certain expected feature is rendered unusable since it wasn't supported by the translation.

- **Poor scalability.** Translation often works in relatively small networks and in internetworks where the number of different network types is small. However, it scales poorly because the amount of information needed by gateways to perform correct translations scales proportional to the square of the number of networks being interconnected. This approach is believed to be untenable for large internetworks.[1]

■ 2.2.2 Universality and Robustness Design Principles

The designers of the ARPANET (the precursor to today's global Internet) protocols recognized the drawbacks of translation early on and decided that the right approach was to standardize some key properties across all networks, and define a small number of features that all hosts and networks connected to the Internet must implement. In their seminal 1974 paper on IP and TCP, Cerf and Kahn describe the rationale behind this decision, pointing out the drawbacks of translation gateways mentioned above.

[1]As an aside, it's worth noting that the widespread deployment of NATs is making portions of the Internet look a lot like translation gateways! We will revisit this point later in this lecture when we discuss scaling issues, but the important difference is that NATs are used to bridge different *IP* networks and don't suffer from the translation-scalability problem, but prevent applications that embed IP addresses in them from working well (among other things).

Over time, we have been able to identify and articulate several important design principles that underlie the design of Internet protocols. We divide these into *universality* principles and *robustness* principles, and focus on only the most important ones (this list a subjective one).

Universality principles

U1. IP-over-everything. All networks and hosts must implement a standard network protocol called **IP**, the **Internet Protocol**. In particular, all hosts and networks, in order to be reachable from anywhere, must implement a standard well-defined addressing convention. This removes a scaling barrier faced by translation-based approaches and makes the gateways simple to implement.

The ability to support IP is necessary (and largely sufficient) for a network to be part of the Internet. The "sufficient" part also requires that the network's IP addresses be globally reachable, in which case the network is said to be part of the "global Internet".

U2. Best-effort service model. A consequence of the simple internal architecture of the gateways is that the service model is best-effort. All packets are treated (roughly) alike and no special effort is made inside the network to recover from lost data in most cases. This well-defined and simple service model is a major reason for the impressive scalability of the architecture. A key consequence of the best-effort design decision is that it is almost trivial for a link technology to support IP.

U3. The end-to-end principles. The end-to-end arguments, articulated by Saltzer et al., permeates many aspects of the Internet design philosophy. The principle as applied to the design of a network argues that very often only the end-points know exactly what they need in terms of certain functions, such as reliability. Implementing those functions at a lower layer (e.g., hop-by-hop) is wasteful because it does not eliminate the need for the higher layer to also provide the same function. That's because packet loss or corruption could happen anywhere in the system, including possibly on the end host itself.

Moreover, there are many kinds of end-to-end applications that have different reliability needs. Some don't care to get every byte reliably and in the order in which they were sent, while others do. Some care about reliability but not ordering. Some care about reliable delivery, but if some time bound has elapsed, then the missing data is of little value. This variety suggests that implementing such functions at a lower layer isn't a good idea because it isn't universally useful.

This argument does not preclude lower layers from implementing functions that improve reliability, or other functions that the end points anyway have to provide. In particular, if implementing a function at a lower layer improves performance significantly, then it's worth doing. For example, reliable data delivery over error-prone links benefits from link-layer recovery mechanisms. Note, however, that such lower layer mechanisms do not eliminate the need for higher layers to conduct their own end-to-end checks and recovery; they "only" improve performance.

The pure view of the Internet architecture is that the interior of the network is simple (or "dumb" or "stupid", as some have called it), with most of the complexity pushed to the ends. As we will see later, there are numerous forces (both technical and not) that question our ability to follow this idea religiously.

Robustness principles

R1. Soft-state inside the network. Most of the state maintained in the gateways is *soft-state*, which is easily refreshed when it is lots or corrupted. Routing table state is a great example of this; gateways use these tables to decide how to route each packet, but the periodic refreshes in the form of routing updates obviate the need for complex error handling software in the routers (IP switches are called "routers" because they implement routing protocols in addition to forwarding IP datagrams). In this way, gateways and routing protocols are designed to overcome failures as part of their *normal* operation, rather than requiring special machinery when failures occur.

R2. Fate sharing. When a host communicates with another, state corresponding to that communication is maintained in the system. The principle of *fate sharing*, as articulated by Dave Clark, says that the critical state for communication between two entities should "share fate" with the entities: i.e., it's alright for that state to be destroyed if one of the entities itself fails, but not otherwise. For example, if some routers on the path between two communication end points fails, that end-to-end communication should work as long as some other path exists (or if the failed router recovers in reasonable time before the end points time out and terminate their communication).

From time to time, people confuse the idea of fate sharing with the end-to-end principle. At other times, people restate the principle of fate sharing as "there should be no state in the network". It's important to avoid both these points of confusion. The end-to-end principle is a statement of where functions should be placed and doesn't say anything about where state should reside. It isn't possible to build a network with no state; at the very least, a packet-switched network needs routing state.

Perhaps the easiest way to understand what the fate sharing principle says is in contrast with the OSI X.25 approach toward reliable data delivery, where gateways maintained hard connection state, taking some responsibility for reliable packet delivery. In this model, end hosts shared fate with these intermediate nodes because the failure of the latter could disrupt the connection entirely.

R3. Conservative-transmission/liberal reception. "Be conservative in what you send; be liberal in what you accept."[2] This guideline for network protocol design significantly increases the robustness of the system. This principle is especially useful when different people, who may have never spoken to each other, code different parts of a large system. This idea is important even when writing code from a specification, because languages like English can be ambiguous, and specifications can change with time. A simple example of this principle is illustrated in what a TCP sender would do if it saw an acknowledgment that acknowledged a packet it had never sent. The worst behavior is to crash because the receiver isn't prepared to receive something it didn't expect. This behavior is bad—what's done is instead is to silently drop this

[2]This statement has been attributed to Jon Postel.

acknowledgment and see what else comes from the peer. Likewise, if a sender transmits a packet with a non-standard option, it should do so only if it has explicitly been negotiated.

This principle increases robustness in a world where entities are generally trustworthy, but the in the current Internet context, malice is common. As a result, it isn't clear that liberally accepting packets that might in fact carry an attack is a good idea. Hence, there is a tension between "firewalling" end hosts from accepting and processing packets liberally, and ensuring that an attacker doesn't take advantage of bugs to compromise machines.

■ 2.3 Scalability

The fundamental reason for the scalability of the Internet's network layer is its use of **topological addressing.** Unlike an Ethernet address that is location-independent and always identifies a host network interface independent of where it is connected in the network, an IP address of a connected network interface depends on its *location in the network topology*. This allows routes to IP addresses to be *aggregated* in the forwarding tables of the routers, and allows routes to be *summarized* and exchanged by the routers participating in the Internet's routing protocols. In the absence of aggressive aggregation, there would be no hope for scaling the system to huge numbers; indeed, this is challenging enough even with the techniques we use today.

Because an IP address signifies location in the network topology, the Internet's network layer can use a variant of classical *area routing* to scalably implement the IP forwarding path.

■ 2.3.1 The area routing idea

The simplest form of classical area routing divides a network layer address into two parts: a fixed-length "area" portion (in the most significant bits, say) and an "intra-area" address. Concatenating these two parts gives the actual network address. For example, one might design a 32-bit area routing scheme with 8 bits for the area and 24 bits for the intra-area address. In this model, forwarding is simple: If a router sees a packet not in its own area, it does a lookup on the "area" portion and forwards the packet on, and conversely. The state in the forwarding tables for routes not in the same area is at most equal to the number of areas in the network, typically much smaller than the total number of possible addresses.

One can extend this idea into a deeper hierarchy by recursively allocating areas at each level, and performing the appropriate recursion on the forwarding path. With this extension, in general, one can define "level 0" of the area routing hierarchy to be each individual router, "level 1" to be a group of routers that share a portion of the address prefix, "level 2" to be a group of level 1 routers, and so on. These levels are defined such that for any level i, there is a path between any two routers in the level $i - 1$ routers within level i that does not leave level i.

There are two reasons why this classical notion of area routing does not directly work in practice:

1. The natural determinant of an area is administrative, but independently adminis-

tered networks vary widely in size. As a result, it's usually hard to determine the right size of the "area" field at any level of the hierarchy. A fixed length for this simply does not work.

2. Managing and maintaining a carefully engineered explicit hierarchy tends to be hard in practice, and does not scale well from an administrative standpoint.

■ 2.3.2 Applying area routing to the Internet: Address classes

The second point above suggests that we should avoid a deep hierarchy that requires manual assignment and management. However, we can attempt to overcome the first problem above by allocating network addresses to areas according to the expected size of an area.

When version 4 of IP ("IPv4") was standardized with RFC 791 in 1981, addresses were standardized to be 32-bits long. At this time, addresses were divided into classes, and organizations could obtain a set of addresses belonging to a class. Depending on the class, the first several bits correspond to the "network" identifier and the others to the "host" identifier. *Class A* addresses start with a "0", use the next 7 bits of network id, and the last 24 bits of host id (e.g., MIT has a Class A network, with addresses in dotted-decimal notation of the form 18.*). *Class B* addresses start with "10" and use the next 14 bits for network id and the last 16 bits for host id. *Class C* addresses start with "110" and use the next 21 bits for network id, and the last 8 bits for host id.[3]

Thus, in the original design of IPv4, areas were allocated in three sizes: *Class A* networks had a large number of addresses, 2^{24}, *Class B* networks had 2^{16} addresses each, and *Class C* networks had 2^{8} addresses each.

The router forwarding path in IPv4 is a little more involved than for the exact-match required by LAN switches, but is still straightforward: a router determines for an address not in its area which class it belongs to, and performs a fixed-length lookup depending on the class.

■ 2.3.3 The problem with class-based addressing

The IETF realized that this two-level network-host hierarchy would soon eventually prove insufficient and in 1984 added a third hierarchical level corresponding to "subnets". Subnets can have any length and are specified by a 32-bit network "mask"; to check if an address belongs to a subnet, take the address and zero out all the bits corresponding to zeroes in the mask; the result should be equal to the subnet id for a valid address in that subnet. The only constraint on a valid mask is that the 1s must be contiguous from most significant bit, and all the 0s must be in the least significant bits.

With subnets, the class-based addressing approach served the Internet well for several years, but ran into scaling problems in the early 1990s as the number of connected networks continued growing dramatically. The problem was *address depletion*—available addresses started running out.

It's important to understand that the problem isn't that the entire space of 2^{32} addresses (about 4 billion) started running out, but that the class-based network address assignment started running out. This is the result of a fundamental inefficiency in the coarse-grained

[3]Class D addresses are used for IP multicast; they start with "1110" and use the next 28 bits for the group address. Addresses that start with "1111" are reserved for experiments.

allocation of addresses in the Class A and (often) Class B portions of the address space.[4] The main problem was that Class B addresses were getting exhausted; these were the most sought after because Class A addresses required great explanation to the IANA (Internet Assigned Numbers Authority) because of the large (2^{24}) host addresses, which Class C addresses with just 256 hosts per network were grossly inadequate. Because only $2^{14} = 16384$ Class B networks are possible, they were running out quickly.

Solution 1: CIDR

In a great piece of "just-in-time" engineering, the IETF stewarded the deployment of **CIDR** (pronounced "Cider" with a short "e"), or **Classless Inter-Domain Routing** recognizing that the division of addresses into classes was inefficient and that routing tables were exploding in size because more and more organizations were using non-contiguous Class C addresses.

CIDR optimizes the common case. The common case is that while 256 addresses are insufficient, most organizations require at most a few thousand. Instead of an entire Class B, a few Class C's will suffice. Furthermore, making these *contiguous* will reduce routing table sizes because routers aggregate routes based on IP prefixes in a classless manner.

With CIDR, each network gets a portion of the address space defined by two fields, A and m. A is a 32-bit number (often written in dotted decimal notation) signifying the address space and m is a number between 1 and 32. A could be thought of as the *prefix* and m the mask: if a network is assigned an address region denoted A/m, it means that it gets the 2^{32-m} addresses all sharing the first m bits of A. For example, the network "18.31/16" corresponds to the 2^{16} addresses in the range $[18.31.0.0, 18.31.255.255]$.

"Solution 2": NAT

NATs are an expedient solution to the address depletion problem. They allow networks to use *private IP addresses*, which different networks can reuse, and deploy globally reachable gateways that *translate* between external address/transport-port pairs and internal ones.

We will discuss NATs in more detail in a later lecture. RFC 1631 gives a quick and simple overview of NATs, though today's NATs are considerably more complicated.

Initiating connections from inside a private network via a NAT is easier (and has been supported from the first deployed NATs). The NAT maintains mappings between internal and external address/port pairs, and modifies the IP *and TCP* header fields on the connection in both directions. Because the TCP checksum covers a portion of the IP header (including the IP addresses), the TCP header needs to be modified.

NATs break several applications, notably ones that use IP addresses in their own payload or fields. Usually, these applications require an application-aware NAT (e.g., FTP).

More recent work has shown that it is possible to use NATs to obtain a form of "global" reachability, by using DNS as a way to name hosts uniquely inside private networks. The result of a DNS lookup in this case would be an IP address/transport-port pair, corresponding to the globally reachable NAT. The NAT would maintain an internal table translating this address/port pair to the appropriate internal address/port pair.

[4]For instance, MIT and Stanford each had entire Class A's to themselves. MIT still does!

Solution 3: IPv6

The long-term solution to the address depletion problem is a new version of IP, IPv6. We will discuss this in Section 2.5.2 after discussing the IPv4 packet forwarding path in a switch (i.e., what a switch must do to forward IP packets).

■ 2.3.4 CIDR lookups: Longest prefix match

The forwarding step with CIDR can no longer be based on determining the class of an address and doing a fixed-length match. Instead, a router needs to implement a *prefix match* to check if the address being looked-up falls in the range A/m for each entry in its forwarding table.

A simple prefix match works when the Internet topology is a tree and there's only one shortest path between any two networks in the Internet. However, the toplogy of the Internet is not a tree; many networks *multi-home* with multiple other networks for redundancy and traffic load balancing (redundancy is the most common reason today).

The consequence of having multiple possible paths is that a router needs to decide on its forwarding path which of potentially several matching prefixes to use for an address being looked-up. By definition, IP (CIDR) defines the correct route as the one corresponding to the *longest prefix* in the routing table that matches the sought destination address. As a result, each router must implement a *longest prefix match (LPM)* algorithm on its forwarding path.

Fast LPM

LPM is not a trivial operation to perform at high speeds of millions of packets (lookups) per second. For several years, the best implementations used an old algorithm based on *PATRICIA trees*, a trie (dictionary) data structure invented decades ago. This algorithm, while popular and useful for lower speeds, does not work in high-speed routers.

To understand the problem of high-speed lookups, let's study what an example high-speed Internet router has to do today. It needs to handle minimum-sized packets (e.g., 40 or 64 bytes depending on the type of link) at link speeds of between 10-20 Gbits/s (today) and 40 Gbits/s or more (soon), which gives the router a fleeting 32 ns (for 40-byte packets) or 51 ns (for 64-byte packets) to make a decision on what to do with the packet (divide those numbers by 4 to get the 40 Gbits/s latencies).

Furthermore, a high-speed Internet router today needs to be designed to handle *at least* 250,000 forwarding table entries, and probably close to 1M entries to accommodate future growth. The number of entries in the backbone routing tables has been growing with time as shown in Figure 2-1.

We can formulate the forwarding table problem solved in Internet routers as follows. Based on the link speed and minimum packet size, we can determine the number of lookups per second. We can then use the latency of the forwarding table memory to determine M, the maximum number of memory accesses allowed per lookup, such that the egress link will remain fully utilized. At the same time, we want all the routes to fit into whatever amount of memory (typically SRAM because it has lower latency than DRAM), S, we can afford for the router. The problem therefore is to maximize the number of routes that fit in S bytes, such that no lookup exceeds M memory accesses. This problem has

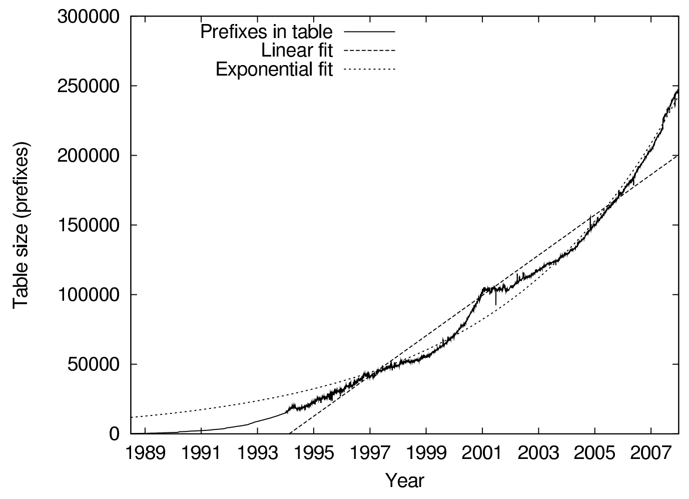


Figure 2-1: Growth in the size of the Internet routing tables as viewed by a router in the "middle" of the Internet. Observe the acceleration in 2000-2001 and the consequences of the "market correction" that occurred soon after! Figure courtesy of David G. Andersen.

been extensively studied by many researchers over the past few years and dozens of papers have been published on the topic. We will consider solutions to this problem when we talk about switch design.

We also note that how components of a router, such as the LPM engine, are engineered changes with time as technology evolves. What was once a good solution may not remain one in the future, and ideas considered impractical in the past may turn out to be good ones in the future.

■ 2.4 Weaknesses and Warts

The Internet is unquestionably a roaring success, going from a research project to world domination in about just three decades. It might therefore be obnoxious to ask what its weaknesses might be. All good architectures are defined as much by things they do well, as by things they don't do at all (or do quite badly). Clark's paper points out some of these. Twenty years later, one can produce a list of things that the architecture ought to do better, but doesn't.

1. *Design for untrusted and distrusting entities.* The original design was done in a world where one could assume trust between participating entities. That clearly doesn't hold any more, as one can tell from the numerous attacks and unsavory activities that occur daily (and make the newspapers from time to time). Spam, phising, worms, denial-of-service (DoS) attacks, etc. are quite common. These are often carried out with impunity because the architecture does not have a strong notion of accountability, whereby one can hold an entity responsible for any action (such as the transmission of a stream of packets).

2. *Support mobility as a first-class goal.* The majority of Internet devices in the near future may well be mobile. The architecture doesn't do a good job of handling this problem. The fundamental tension is between topological addressing, used for scalability, and changing addresses, caused by mobility. We will study solutions to this problem.

3. *Support host and site multi-homing as a first-class goal.* An increasing number of Internet hosts and sites (such as small companies, and even some homes) are multi-homed, having multiple links, sometimes from different ISPs. Current protocols don't do a great job of handling multiple interfaces and taking advantage of multiple paths.

4. *Treat data and content as first-class objects.* The Internet protocols were originally designed to support applications that were interested in connecting to particular remote computers (e.g., remote terminals). Today, much of what happens is *data-centric*: people care about receiving content from the web, and don't really care which specific computers that content comes from.

5. *Design practical "better-than-best-effort" services.* The pure model of the Internet is that it provides a best-effort service that essentially treats all packets the same. There are several reasons to consider schemes to prioritize certain flows, provide rate guarantees for certain flow aggregates, and delay guarantees as well.

6. *Improve availability for applications that need it.* Mission-critical applications require a higher degree of fault-tolerance and availability; how might we do that?

7. *Provide a better path to deploy improvements.* It's hard to imagine a "flag day" for the Internet any more. How might one go about developing an architecture that would allow new ideas to be tried out and deployed without disrupting existing activity?

8. *Rethink layering abstractions.* There are many examples where the current layer abstractions and interfaces are not a good match to the problem at hand. The best example is perhaps wireless networks, where a variety of new cross-layer techniques achieve dramatic throughput improvements. Layering is clearly a good idea in terms of achieving modularity, but the previously developed interfaces and abstractions were done assuming mostly wired networks. Is it time to rethink these abstractions?

9. *Managing large networks should be easier.* The architecture doesn't provide any intrinsic support for easy management, monotoring, and configuation.

10. *Fault diagnosis should be easier.* When something goes wrong, it takes wizardry and time to figure out what's happening.

■ 2.5 Summary

The Internet architecture is based on design principles that provide universality and improve robustness to failures. It achieves scaling by using an addressing structure that reflects network topology, enabling topological address information to be aggregated in routers.

There are, however, many things that the architecture doesn't do so well. Much of our study will focus on these problems, in addition to understanding the things that do work well.

■ Appendix: Some Details about IPv4 and IPv6

We now look at a few other details of the IP layer, dividing our discussion into IPv4 and IPv6 (the next version of IP from 4^5).

■ 2.5.1 IPv4

Fragmentation and Reassembly

Different networks do not always have the same MTU, or Maximum Transmission Unit. When an IP gateway receives a packet[6] that needs to be forwarded to a network with a smaller MTU than the size of the packet, it can take one of several actions.

1. Discard the packet. In fact, IP does this if the sender set a flag on the header telling gateways not to fragment the packet. After discarding it, the gateway sends an error message using the ICMP protocol to the sender. [What are some reasons for providing this feature in the Internet?]

2. Fragment the packet. The default behavior in IPv4 is to fragment the packet into MTU-sized fragments and forward each fragment to the destination. There's information in the IP header about the packet ID and offset that allows the receiver to reassemble fragments.

Time-to-live (TTL)

To prevent packets from looping forever, the IP header has a TTL field. It's usually decremented by 1 at each router and the packet is discarded when the TTL is zero.

Type-of-service (TOS)

The IP header includes an 8-bit type-of-service field that allows routers to treat packets differently. It's largely unused today, although we'll later see how differentiated services intends to use this field.

Protocol field

To demultiplex an incoming packet to the appropriate higher (usually transport) layer, the IP header contains an 8-bit protocol field.

Header checksum

To detect header corruption, IP uses a (weak) 16-bit header checksum.

[5] IPv5 didn't last long!

[6] Or *datagram.*

IP options

IP allows nodes to introduce options in its header. A variety of options are defined but are hardly used, even when they're useful. This is because it's expensive for high-speed routers (based on the way they're designed today) to process options on the fast path. Furthermore, most IP options are intended for end-points to process, but IPv4 mandates that all routers process all options, even when all they have to do is ignore them. This makes things slow—and is a severe disincentive against options. Engineering practice today is to avoid options in favor of IP-in-IP encapsulation or, in many cases, having the routers peek into transport headers (e.g., the TCP or UDP port numbers) where the fields are in well-known locations and require little parsing.

■ 2.5.2 IPv6

IPv6 is the next version of IP. Its development is motivated by several shortcomings and problems with IPv4. The main problem that IPv6 solves is the address space shortage problem.

Its design challenges, and how it achieves these goals include:

1. Incremental deployment. No more flag days are possible on an Internet with over tens of millions of nodes. Painstakingly engineered to coexist and seamlessly transition from IPv4.

2. Scale. It has been designed for a scale much larger than today's Internet. The main feature that enables this is a *huge* 128-bit address space.

3. Easy configuration. Attempts to reduce manual network configuration by an auto-configuration mechanism. Here, end node interfaces can pick an address automatically using a combination of provider prefix and local hardware MAC address.

4. Simplification. Gets rid of little used stuff in the IP header, such as fragment offsets, etc. Even though addresses are four times as large as IPv4, the base header size is only twice as large.

5. Improved option processing. Changes the way options are encoded to enable efficient forwarding. Options (e.g., security options) are placed in separate extension headers that routers can easily ignore.

6. Authentication & privacy. Network-level security is built-in for applications that desire it.

Addresses & Flows

IPv6 addresses are 128 bits long. The paper describes the details of how they are organized. An important point to note is that most wide-area communication is expected to use *provider-based* addresses, where contiguous address blocks are allocated to Internet service providers (ISPs) rather than directly to organizations. [What does this mean when an organization changes its ISP?]

IPv6 also has *local-use* addresses that don't have global scope, with link-local (e.g., same LAN) and site-local (within the organization) scope. These allow intra-organization communication even if disconnected from the global Internet.

To enable incremental deployment, IPv6 addresses can contain embedded IPv4 addresses. IPv6 also supports *anycast*, where routers will route a packet to the closest (defined in terms of the metric used by routers, such as hop count) of many interfaces that answers to that address. Anycast addresses aren't special in IPv6; they come from the space of unicast addresses.

Flows are an important useful addition to IPv6. You can think of a flow as a TCP connection or as a stream of UDP traffic between the same host-port pairs. IPv6 explicitly formalizes this notion and makes flow labels visible to routers and network-layer entities in the IP header. Flow labels are chosen randomly by end-hosts to enable good classification by network entities.

Fragmentation

IPv6 does away with network-layer fragmentation and reassembly. End-points are expected to perform path-MTU discovery. Locally, links are free to perform fragmentation and reassembly if they so desire. All IPv6 networks must handle an MTU of at least 1280 bytes.

■ Appendix: The standards process

An interesting aspect of the continuing development of the Internet infrastructure is the process by which standards are set and developments are made. By and large, this process has been a successful exercise in social engineering, in which people from many different organizations make progress toward consensus. The Internet Engineering Task Force (IETF), a voluntary organization, is the standards-setting body. It meets every 4 months, structured around short-lived working groups (usually less than 2 years from inception to termination). Internet standards are documented in RFCs ("Request for Comments" documents) published by the IETF (several different kinds of RFCs exist), and documents in draft form before they appear as RFCs are called "Internet Drafts." Check out http://www.ietf.org/ for more information.

Working groups conduct a large amount of work by email, and are open to all interested people. In general, voting on proposals are shunned, and the process favors "rough consensus." Running, working code is usually a plus, and can help decide direction. It used to be the case that at least two independent, interoperating implementations of a protocol were required for a standard to be in place, but this requirement seems to be generally ignored these days.

■ Appendix: Notes on Cerf & Kahn's original TCP [cerfkahn]

Note: The first section of this paper is required reading; the rest is optional and recommended, but not required. The notes below could help navigate the rest of the paper, which some might find hard to read because it's different from today's details.

Cerf & Kahn describe their solutions to the internetworking problems in their seminal 1974 paper. This early work has clearly had enormous impact, and this is one of the main reasons we study it despite the actual details of today's IP being very different. Some of the design principles have been preserved to date. Furthermore, this is a nice paper in that it presents a design with enough detail that it feels like one can sit in front of a computer and code up an implementation from the specifications described!

The following are the key highlights of their solution:

- Key idea: **gateways.** Reject translation in favor of gateways.
- IP over everything—beginnings of protocol hourglass model evident in first section of paper.
- As a corollary, addressing is common to entire internetwork. Internal characteristics of networks are of course different.
- Standard packet header format. But notice mixing of transport information in routing header. (And some fields like ttl are missing.)
- Gateways perform *fragmentation* if needed, *reassembly* is done at the end hosts.
- **TCP:** Process-level communication via the TCP, a reliable, in-order, byte-stream protocol. And in fact, they had the TCP do reassembly of segments (rather than IP that does it today at the receiver). TCP and IP were tightly inter-twined together; it took several years for the TCP/IP split to become visible.
- The spent a lot of time on figuring out how multiple processes between the same end hosts communicate. They considered two options:

  1. Use same IP packet for multiple TCP segments.
  2. Use different packets for different segments.

  As far as I can tell, they didn't consider the option of many TCP connections between the same hosts!
- 2 is simpler than 1 because it's easier to demultiplex and out-of-order packets are harder to handle in 1.
- Demux key is port number (typically one process per port).
- Byte-stream TCP, so use ES and EM flags. EM, ES, SEQ and CT adjusted by gateways. I.e., gateways look through and modify transport information. End-to-end checksums for error detection.
- Sliding window, cumulative ACK-based ARQ protocol (with timeouts) for reliability and flow control. Similar to CYCLADES and ARPANET protocols. (CYCLADES was probably the first system to use sliding windows.)
- Window-based flow control. Also achieve process-level flow control.
- Implementation details/recommendations for I/O handling, TCBs.

- First-cut connection setup and release scheme. (Heavily modified subsequently to what it is today.)
- Way off the mark on accounting issues! (The problem is a lot harder than they anticipated.)
- One-line summary of impact: A seminal paper that led to today's dominant data network architecture.

CHAPTER 3
Interdomain Internet Routing

Our goal is to explain how routing between different administrative domains works in the Internet. We discuss how Internet Service Providers (ISPs) exchange routing information, packets, and (above all) money between each other, and how the way in which they buy service from and sell service to each other and their customers influences routing. We discuss the salient features of the Border Gateway Protocol, Version 4 (BGP4, which we will refer to simply as BGP), the current interdomain routing protocol in the Internet. Finally, we discuss a few interesting failures and shortcomings of the routing system. These notes focus only on the essential elements of interdomain routing, often sacrificing detail for clarity and sweeping generality.

■ 3.1 Autonomous Systems

Network attachment points for Internet hosts are identified using IP addresses, which are 32-bit (in IPv4) numbers. The fundamental reason for the scalability of the Internet's network layer is its use of *topological addressing.* Unlike an Ethernet address that is location-independent and always identifies a host network interface independent of where it is connected in the network, an IP address of a connected network interface depends on its location in the network topology. Topological addressing allows routes to Internet destinations to be *aggregated* in the forwarding tables of the routers (as a simple example of aggregation, any IP address of the form 18.* in the Internet is in MIT's network, so external routers need only maintain a routing entry for 18.* to correctly forward packets to MIT's network), and allows routes to be *summarized* and exchanged by the routers participating in the Internet's routing protocols. In the absence of aggressive aggregation, there would be no hope for scaling the routing system to hundreds of millions of hosts connected over tens of millions of networks located in tens of thousands of ISPs and organizations.

Routers use *address prefixes* to identify a contiguous range of IP addresses in their routing messages. For example, an address prefix of the form 18.31.* identifies the IP address range 18.31.0.0 to 18.31.255.255, a range whose size is 2^{16}. Each routing message involves a router telling a neighboring router information of the form "To get to address prefix P, you can use the link on which you heard me tell you this,"[1] together with information about the route (the information depends on the routing protocol and could include the number of hops, cost of the route, other ISPs on the path, etc.).

An abstract, highly idealized view of the Internet is shown in Figure 3-1, where end-hosts hook up to routers, which hook up with other routers to form a nice connected graph of essentially "peer" routers that cooperate nicely using routing protocols that exchange "shortest-path" or similar information and provide global connectivity. The same view posits that the graph induced by the routers and their links has a large amount of redundancy and the Internet's routing algorithms are designed to rapidly detect faults and problems in the routing substrate and route around them. In addition to routing around failures, clever routing protocols performing load-sensitive routing dynamically shed load away from congested paths on to less-loaded paths.

Unfortunately, while simple, this abstraction is actually quite misleading as far as the wide-area Internet is concerned. The real story of the Internet routing infrastructure is that the Internet service is provided by a large number of commercial enterprises, generally in competition with each other. Cooperation, required for global connectivity, is generally at odds with the need to be a profitable commercial enterprise, which often occurs at the expense of one's competitors—the same people with whom one needs to cooperate. How this "competitive cooperation" is achieved in practice (although there's lots of room for improvement), and how we might improve things, provides an interesting study of how good technical research can be shaped and challenged by commercial realities.

Figure 3-2 depicts a more realistic view of what the Internet infrastructure looks like. Here, Internet Service Providers (ISPs) cooperate to provide global connectivity to their respective customer networks. An important point about ISPs is that they aren't all equal; they come in a variety of sizes and internal structure. Some are bigger and more "connected" than others, and others have global reachability in their routing tables. A simplified, but useful, model is that they come in three varieties, "small," "large," and "really huge", and these colloquial descriptions have names given to them. *Tier-3 ISPs* have a small number of usually localized (in geography) end-customers, *Tier-2 ISPs* generally have regional scope (e.g., state-wide, region-wide, or non-US country-wide), while *Tier-1 ISPs*, of which there are a handful (nine in early 2008), have global scope in the sense that their routing tables actually have explicit routes to all currently reachable Internet prefixes (i.e., they have *no default routes*). Figure shows this organization.

The previous paragraph used the term "route", which we define as a mapping from an IP prefix (a range of addresses) to a link, with the property that packets for any destination within that prefix may be sent along the corresponding link. A router adds such entries to its routing table after selecting the best way to reach each prefix from among route advertisements sent by its neighboring routers.

Routers exchange route advertisements with each other using a routing protocol. The current wide-area routing protocol in the Internet, which operates between routers at the boundary between ISPs, is *BGP* (Border Gateway Protocol, Version 4) [19, 20]. More precisely, the wide-area routing architecture is divided into *autonomous systems* (ASes) that exchange reachability information. An AS is owned and administered by a single com-

[1]It turns out that this simple description is not always true, notably in the iBGP scheme discussed later, but it's true most of the time.

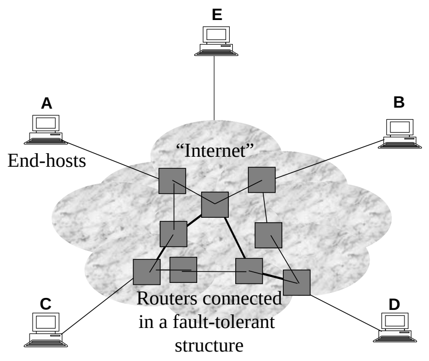


Figure 3-1: A rather misleading view of the Internet routing system.

mercial entity, and implements some set of policies in deciding how to route its packets to the rest of the Internet, and how to export its routes (its own, those of its customers, and other routes it may have learned from other ASes) to other ASes. Each AS is identified by a unique 16-bit number.

A different routing protocol operates within each AS. These routing protocols are called *Interior Gateway Protocols* (IGPs), and include protocols like the Routing Information Protocol (RIP) [10], Open Shortest Paths First (OSPF) [16], Intermediate System-Intermediate System (IS-IS) [17], and E-IGRP. In contrast, BGP is an interdomain routing protocol. Operationally, a key difference between BGP and IGPs is that the former is concerned with exchanging reachability information between ASes in a scalable manner while allowing each AS to implement autonomous *routing policies*, whereas the latter are typically concerned with optimizing a path metric. In general, IGPs don't scale as well as BGP does with respect to the number of participants involved.

The rest of this paper is in two parts: first, we will look at inter-AS relationships (transit and peering); then, we will study some salient features of BGP. We won't talk about IGPs like RIP and OSPF; to learn more about them, read a standard networking textbook (e.g., Peterson & Davie or Kurose & Ross).

■ 3.2 Inter-AS Relationships: Transit and Peering

The Internet is composed of many different types of ASes, from universities to corporations to regional ISPs to nation-wide ISPs. Smaller ASes (e.g., universities, corporations, etc.) typically purchase Internet connectivity from ISPs. Smaller regional ISPs, in turn, purchase connectivity from larger ISPs with large "backbone" networks.

Consider the picture shown in Figure 3-3. It shows an ISP, *X*, directly connected to a *provider* (from whom it buys Internet service) and a few *customers* (to whom it sells Internet service). In addition, the figure shows two other ISPs to whom it is directly connected, with whom *X* exchanges routing information via BGP.

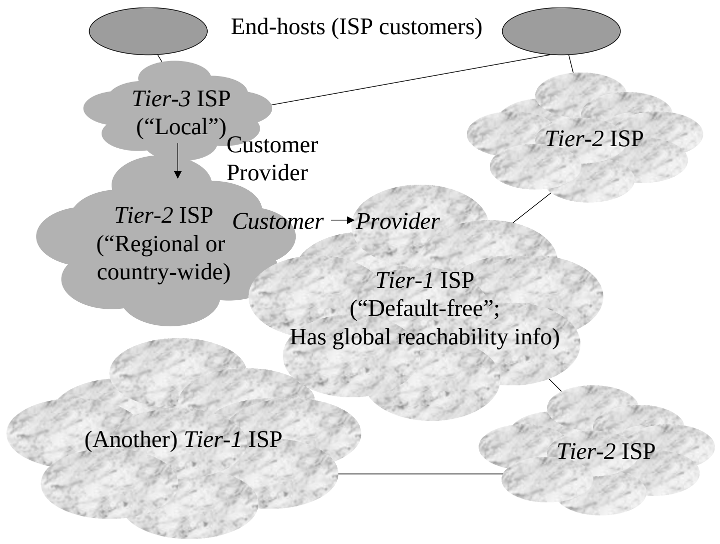

End-hosts (ISP customers)
Tier-3 ISP ("Local")
Customer
Provider
Tier-2 ISP ("Regional or country-wide)
Customer → *Provider*
Tier-2 ISP
Tier-1 ISP ("Default-free"; Has global reachability info)
(Another) *Tier-1* ISP
Tier-2 ISP

Figure 3-2: A more accurate picture of the wide-area Internet routing system, with various types of ISPs defined by their respective reach. *Tier-1* ISPs have "default-free" routing tables (i.e., they don't have any default routes), and have global reachability information. There are a handful of these today (nine in early 2008).

The different types of ASes lead to different types of business relationships between them, which in turn translate to different policies for exchanging and selecting routes. There are two prevalent forms of AS-AS interconnection. The first form is *provider-customer transit* (aka "transit"), wherein one ISP (the "provider" P in Figure 3-3) provides access to all (or most) destinations in its routing tables. Transit almost always is meaningful in an inter-AS relationship where financial settlement is involved; the provider charges its customers for Internet access, in return for forwarding packets on behalf of customers to destinations (and in the opposite direction in many cases). Another example of a transit relationship in Figure 3-3 is between X and its customers (the C_is).

The second prevalent form of inter-AS interconnection is called *peering*. Here, two ASes (typically ISPs) provide mutual access to a subset of each other's routing tables. The subset of interest here is their own transit customers (and the ISPs own internal addresses). Like transit, peering is a business deal, but it may not involve financial settlement. While paid peering is common in some parts of the world, in many cases they are reciprocal agreements. As long as the traffic ratio between the concerned ASs is not highly asymmetric (e.g., 4:1 is a commonly believed and quoted ratio), there's usually no financial settlement. Peering deals are almost always under non-disclosure and are confidential.

■ 3.2.1 Peering v. Transit

A key point to note about peering relationships is that they are often between business competitors. There are two main reasons for peering relationships:

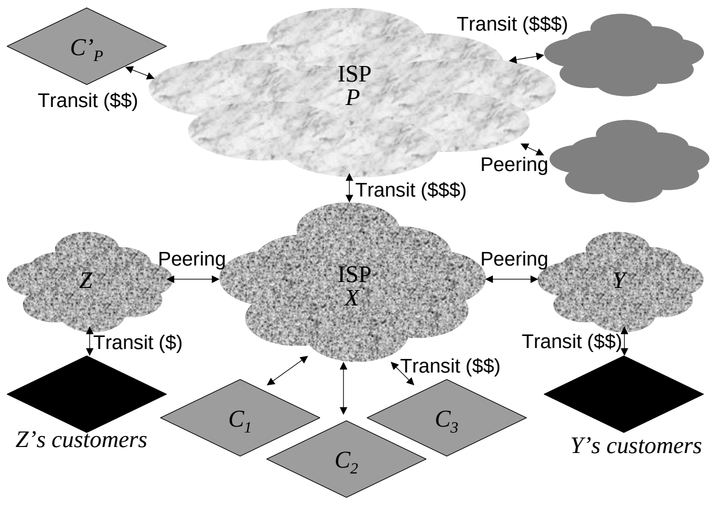


Figure 3-3: Common inter-AS relationships: transit and peering.

1. **Tier-1 peering:** An Internet that had only provider-customer transit relationships would require a single (large) Tier-1 at the root, because cycles in the directed graph of ASes don't make sense (a cycle would require that the money paid by each AS to its provider would flow from an AS back to itself). That single Tier-1 would in effect be a monopoly because it would control all Internet traffic (before the commercialization of the Internet, the NSFNET was in fact such a backbone). Commercial competition has led to the creation of a handful of large Tier-1 ISPs (nine today, up from five a few years ago). In effect, these ISPs act as a sort of cartel because an ISP has to be of a certain size and control a reasonable chunk of traffic to be able to enter into a peering relationship with any other Tier-1 provider. Peering between Tier-1 ISPs ensures that they have explicit default-free routes to all Internet destinations (prefixes).

2. **Saving money:** Peering isn't restricted to large Tier-1's, though. If a non-trivial fraction of the packets emanating from a an AS (regardless of its size) or its customers is destined for another AS or its customers, then each AS has an incentive to avoid paying transit costs to its respective providers. Of course, the best thing for each AS to do would be to wean away the other's customers, but that may be hard to achieve. The next best thing, which would be in their mutual interest, would be to avoid paying transit costs to *their* respective providers, but instead set up a transit-free link between each other to forward packets for their direct customers. In addition, this approach has the advantage that this more direct path would lead to better end-to-end performance (in terms of latency, packet loss rate, and throughput) for their customers.

Balancing these potential benefits are some forces against peering. Transit relationships generate revenue; peering relationships usually don't. Peering relationships typically need to be renegotiated often, and asymmetric traffic ratios require care to handle in a way that's mutually satisfactory. Above all, these relationships are often between competitors vying for the same customer base.

In the discussion so far, we have implicitly used an important property of current interdomain routing: *A route advertisement from B to A for a destination prefix is an agreement by B that it will forward packets sent via A destined for any destination in the prefix.* This (implicit) agreement implies that one way to think about Internet economics is to view ISPs as charging customers for entries in their routing tables. Of course, the data rate of the interconnection is also crucial, and is the major determinant of an ISP's pricing policy.

A word on pricing is in order. ISPs charge for Internet access in one of two ways. The first is a fixed price for a certain speed of access (fixed pricing is a common way to charge for home or small business access in the US). The second measures traffic and charges according to the amount of bandwidth used. A common approach is to measure usage in 5-minute windows over the duration of the pricing period (typically one month). These 5-minute averages are samples that form a distrubution. The ISP then calculates the 95th (or 90th) percentile of this distribution and charges an amount that depends on this value, using the terms set in the contract. ISPs also offer discounts for outages experienced; some ISPs now have delay guarantees (through their network) set up in the contract for certain kinds of traffic. It's unclear to what extent customer networks can and do go about verifying that providers comply with the all terms set in their contract.

The take-away lesson here is simple: providers make more money from a customer if the amount of traffic sent on behalf of the customer increases. This simple observation is the basis for a large number of complex routing policies that one sees in practice. In the rest of this section, we describe some common examples of routing policy.

■ 3.2.2 Exporting Routes to Make or Save Money

Each AS (ISP) needs to make decisions on which routes to export to its neighboring ISPs using BGP. The reason why export policies are important is that no ISP wants to act as transit for packets that it isn't somehow making money on. Because packets flow in the opposite direction to the (best) route advertisement for any destination, an AS should advertise routes to neighbors with care.

Transit customer routes. To an ISP, its customer routes are likely the most important, because the view it provides to its customers is the sense that *all* potential senders in the Internet can reach them. It is in the ISP's best interest to advertise routes to its transit customers to as many other connected ASes as possible. The more traffic that an ISP carries on behalf of a customer, the "fatter" the pipe that the customer would need, implying higher revenue for the ISP. Hence, if a destination were advertised from multiple neighbors, an ISP should prefer the advertisement made from a customer over all other choices (in particular, over peers and transit providers).

Transit provider routes. Does an ISP want to provide *transit* to the routes exported by its provider to it? Most likely not, because the ISP isn't making any money on providing such transit facilities. An example of this situation is shown in Figure 3-3, where C'_P is a customer of P, and P has exported a route to C'_P to X. It isn't in X's interest to advertise this route to everyone, e.g., to other ISPs with whom X has a peering relationship. An important exception to this, of course, is X's transit customers who are paying X for service—the service X provides its customers C_i's is that they can reach any location on the Internet via

X, so it makes sense for X to export as many routes to X as possible.

Peer routes. It usually makes sense for an ISP to export only selected routes from its routing tables to other peering ISPs. It obviously makes sense to export routes to all of ones transit customers. It also makes sense to export routes to addresses within an ISP. However, it does not make sense to export an ISP's transit provider routes to other peering ISPs, because that may cause a peering ISP to use the advertising ISP to reach a destination advertised by a transit provider. Doing so would expend ISP resources but not lead to revenue.

The same situation applies to routes learned from other peering relationships. Consider ISP Z in Figure 3-3, with its own transit customers. It doesn't make sense for X to advertise routes to Z's customers to another peering ISP (Y), because X doesn't make any money on Y using X to get packets to Z's customers!

These arguments show that most ISPs end up providing *selective transit*: typically, full transit capabilities for their own transit customers in both directions, some transit (between mutual customers) in a peering relationship, and transit only for one's transit customers (and ISP-internal addresses) to one's providers.

The discussion so far may make it sound like BGP is the only way in which to exchange reachability information between an ISP and its customers or between two ASes. That is not true in practice, though; a large fraction of end-customers (typically customers who don't provide large amounts of further transit and/or aren't ISPs) don't run BGP sessions with their providers. The reason is that BGP is complicated to configure, administer, and manage, and isn't particularly useful if the set of addresses in the customer's network is relatively invariant. These customers interact with their providers via *static routes*. These routes are usually manually configured. Of course, information about customer address blocks will in general be exchanged by a provider using BGP to other ASes (ISPs) to achieve global reachability to the customer premises.

For the several thousands of networks that run BGP, deciding which routes to export is an important policy decision. Such decisions are expressed using *route filters*, which are rules that decide which routes an AS's routers should filter to routers of neighboring ASes.

■ 3.2.3 Importing Routes to Make or Save Money

In addition to deciding how to filter routes while exporting them, when a router hears many possible routes to a destination network, it needs to decide which route to import into its forwarding tables. The problem boils down to *ranking* routes to each destination prefix.

Deciding which routes to import is a fairly involved process in BGP and requires a consideration of several attributes of the advertised routes. For the time being, we consider only one of the many things that an AS needs to consider, but it's the most important concern, viz., *who advertised the route*?

Typically, when a router (e.g., X in Figure 3-3) hears advertisements about its transit customers from other ASes (e.g., because the customer is multi-homed), it needs to ensure that packets to the customer do not traverse additional ASes unnecessarily. The main reason is that an AS doesn't want to spend money paying its providers for traffic destined to its direct customer, and wants to increase its value as perceived by the customer. This requirement usually means that customer routes are prioritized over routes to the same network advertised by providers or peers. Second, peer routes are likely more preferable to provider routes, because the purpose of peering was to exchange reachability information about mutual transit customers. These two observations imply that typically routes are imported in the following priority order:

customer $>$ **peer** $>$ **provider**

This rule (and many others like it) can be implemented in BGP using a special attribute that's locally maintained by routers in an AS, called the LOCAL PREF attribute. The first rule in route selection with BGP is to rank routes according to the LOCAL PREF attribute and pick the one with the highest value. It is only when this attribute is *not* set for a route that other attributes of a route even considered in the ranking procedure.

That said, in practice most routes are not selected using the LOCAL PREF attribute; other attributes like the length of the AS path tend to be quite common. We discuss these other route attributes and the details of the BGP route selection process, also called the *decision process*, when we talk about the main idea in BGP.

■ 3.2.4 Routing Policy = Ranking + Filtering

Network operators express a wide range of routing policies, but to first approximation, most of them can be boiled down to *ranking decisions* and *export filters*.

■ 3.3 BGP

We now turn to how reachability information is exchanged using BGP, and see how routing policies like the ones explained in the previous section can be expressed and realized. We start with a discussion of the main design goals in BGP and summarize the protocol. Most of the complexity in wide-area routing is not in the protocol, but in how BGP routers are configured to implement policy, and in how routes learned from other ASes are disseminated within an AS. The rest of the section discusses these issues.

■ 3.3.1 Design Goals

In the old NSFNET, the backbone routers exchanged routing information over a tree topology, using a routing protocol called EGP (Exterior Gateway Protocol). Because the backbone routing information was exchanged over a tree, the routing protocol was relatively simple. The evolution of the Internet from a singly administered backbone to its current commercial structure made the NSFNET EGP obsolete and required a more sophisticated protocol.

The design of BGP was motivated by three important needs:

1. **Scalability.** The division of the Internet into ASes under independent administration was done while the backbone of the then Internet was under the administration of the NSFNet. When the NSFNet was "turned off" in the early 1990s and the Internet routing infrastructure opened up to competition in the US, a number of ISPs providing different sizes sprang up. The growth in the number of networks (and hosts) has continued to date. To support this growth, routers must be able to handle an

increasing number of prefixes, BGP must ensure that the amount of advertisement traffic scales well with "churn" in the network (parts of the network going down and coming up), and BGP must converge to correct loop-free paths within a reasonable amount of time after any change. Achieving these goals is not easy.

2. **Policy.** The ability for each AS to implement and enforce various forms of routing policy was an important design goal. One of the consequences of this was the development of the BGP attribute structure for route announcements, allowing route filtering, and allowing each AS to rank its available routes arbitrarily.

3. **Cooperation under competitive circumstances.** BGP was designed in large part to handle the transition from the NSFNet to a situation where the "backbone" Internet infrastructure would no longer be run by a single administrative entity. This structure implies that the routing protocol should allow ASes to make purely local decisions on how to route packets, from among any set of choices. Moreover, BGP was designed to allow each AS to keep its ranking and filtering policies confidential from other ASes.

But what about security? Ensuring the authenticity and integrity of messages was understood to be a good goal, but there was a pressing need to get a reasonable routing system in place before the security story was fully worked out. Efforts to secure BGP, notably S-BGP [14], have been worked out and involve external registries and infrastructure to maintain mappings between prefixes and the ASes that own them, as well as the public keys for ASes. These approaches have not been deployed in the Internet for a variety of reasons, including the fact that existing routing registries tend to have a number of errors and omissions (so people can't trust them a great deal).

Misconfigurations and malice cause connectivity outages from time to time because routing to various destinations gets fouled up. The next section gives some examples of past routing problems that have made the news; those examples illustrate the adage that complex systems fail for complex reasons.

■ 3.3.2 Protocol Details

As protocols go, BGP is not an overly complicated protocol (as we'll see later, what makes its operation complicated is the variety and complexity of BGP router configurations). The basic operation of BGP—the protocol state machine, the format of routing messages, and the propagation of routing updates—are all defined in the protocol standard (RFC 4271, which obsoletes RFC 1771) [21]. BGP runs over TCP on a well-known port (179). To start participating in a *BGP session* with another router, a router sends an OPEN message after establishing a TCP connection to it on the BGP port. After the OPEN is completed, both routers exchange their tables of all active routes (of course, applying all applicable route filtering rules). Each router then integrates the information obtained from its neighbor into its routing table. The entire process may take many seconds to a few minutes to complete, especially on sessions that have a large number of active routes.

After this initialization, there are two main types of messages on the BGP session. First, BGP routers send route UPDATE messages sent on the session. These updates only send any routing entries that have changed since the last update (or transmission of all active routes). There are two kinds of updates: *announcements*, which are changes to existing routes or new routes, and *withdrawals*, which are messages that inform the receiver that the named routes no longer exist. A withdrawal usually happens when some previously announced route can no longer be used (e.g., because of a failure or a change in policy). Because BGP uses TCP, which provides reliable and in-order delivery, routes do not need to be periodically announced, unless they change.

But, in the absence of periodic routing updates, how does a router know whether the neighbor at the other end of a session is still functioning properly? One possible solution might be for BGP to run over a transport protocol that implements its own "is the peer alive" message protocol. Such messages are also called "keepalive" messages. TCP, however, does not implement a transport-layer "keepalive" (with good reason), so BGP uses its own. Each BGP session has a configurable keepalive timer, and the router guarantees that it will attempt to send at least one BGP message during that time. If there are no UPDATE messages, then the router sends the second type of message on the session: KEEPALIVE messages. The absence of a certain number BGP KEEPALIVE messages on a session causes the router to terminate that session. The number of missing messages depends on a configurable times called the *hold timer*; the specification recommends that the hold timer be at least as long as the keepalive timer duration negotiated on the session.

More details about the BGP state machine may be found in [2, 21].

Unlike many IGP's, BGP does not simply optimize any metrics like shortest-paths or delays. Because its goals are to provide reachability information and enable routing policies, its announcements do not simply announce some metric like hop-count. Rather, they have the following format:

$$IP\ prefix : Attributes$$

where for each announced IP prefix (in the "A/m" format), one or more attributes are also announced. There are a number of standardized attributes in BGP, and we'll look at some of them in more detail below.

We already talked about one BGP attribute, LOCAL PREF. This attribute isn't disseminated with route announcements, but is an important attribute used locally while selecting a route for a destination. When a route is advertised from a neighboring AS, the receiving BGP router consults its configuration and may set a LOCAL PREF for this route.

■ 3.3.3 eBGP and iBGP

There are two types of BGP sessions: *eBGP* sessions are between BGP-speaking routers in different ASes, while *iBGP* sessions are between BGP routers in the same AS. They serve different purposes, but use the same protocol.

eBGP is the "standard" mode in which BGP is used; after all, BGP was designed to exchange network routing information between different ASes in the Internet. eBGP sessions are shown in Figure 3-4, where the BGP routers implement route filtering rules and exchange a subset of their routes with routers in other ASes. These sessions generally operate over a one-hop IP path (i.e., over directly connected IP links).

In general, each AS will have more than one router that participates in eBGP sessions with neighboring ASes. During this process, each router will obtain information about some subset of all the prefixes that the entire AS knows about. Each such eBGP router

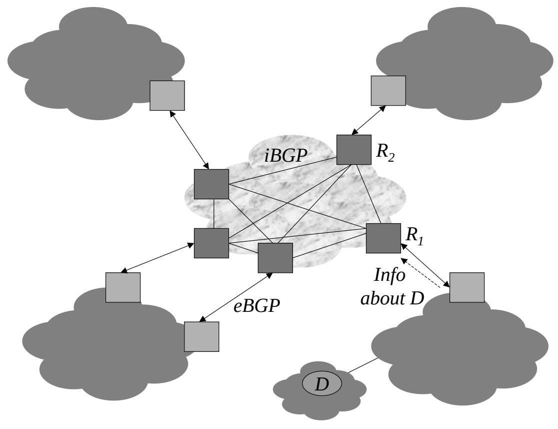


Figure 3-4: eBGP and iBGP.

must disseminate routes to the external prefix to all the other routers in the AS. This dissemination must be done with care to meet two important goals:

1. *Loop-free forwarding.* After the dissemination of eBGP learned routes, the resulting routes (and the subsequent forwarding paths of packets sent along those routes) picked by all routers should be free of deflections and forwarding loops [6, 9].

2. *Complete visibility.* One of the goals of BGP is to allow each AS to be treated as a single monolithic entity. This means that the several eBGP-speaking routes in the AS must exchange external route information so that they have a complete view of all external routes. For instance, consider Figure 3-4, and prefix D. Router R_2 needs to know how to forward packets destined for D, but R_2 hasn't heard a direct announcement on any of its eBGP sessions for D.[2] By "complete visibility", we mean the following: *for every external destination, each router picks the same route that it would have picked had it seen the best routes from each eBGP router in the AS.*

   The dissemination of externally learned routes to routers inside an AS is done over *internal BGP* (iBGP) sessions running in each AS.

An important question concerns the topology over which iBGP sessions should be run. One possibility is to use an arbitrary connected graph and "flood" updates of external routes to all BGP routers in an AS. Of cours, an approach based on flooding would require additional techniques to avoid routing loops. The original BGP specification solved this problem by simply setting up a *full mesh* of iBGP sessions (see Figure 3-5, where every eBGP router maintains an iBGP session with every other BGP router in the AS. Flooding updates is now straightforward; an eBGP router simply sends UPDATE messages to its iBGP neighbors. An iBGP router does not have to send any UPDATE messages because it does not have any eBGP sessions with a router in another AS.

[2]It turns out that each router inside doesn't (need to) know about all the external routes to a destination. Rather, the goal is for each router to be able to discover the best routes of the border routers in the AS for a destination.

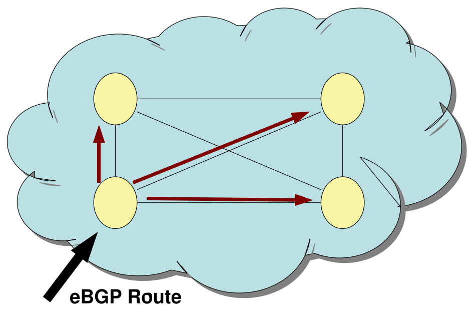


Figure 3-5: Small ASes establish a "full mesh" of iBGP sessions. Each circle represents a router within an AS. Only eBGP-learned routes are re-advertised over iBGP sessions.

It is important to note that *iBGP is not an IGP* like RIP or OSPF, and it cannot be used to set up routing state that allows packets to be forwarded correctly between internal nodes in an AS. Rather, iBGP sessions, running over TCP, provide a way by which routers inside an AS can use BGP to exchange information about external routes. In fact, iBGP sessions and messages are themselves routed between the BGP routers in the AS via whatever IGP is being used in the AS!

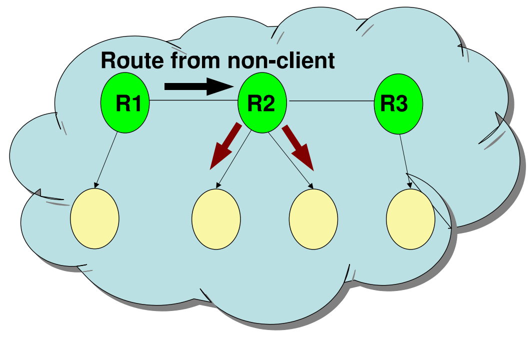
One might wonder why iBGP is needed, and why one can't simply use whatever IGP is being used in the AS to also send BGP updates. There are several reasons why introducing eBGP routes into an IGP is inconvenient, though it's possible to use that method. The first reason is that most IGPs don't scale as well as BGP does with respect to the number of routes being sent, and often rely on periodic routing announcements rather than incremental updates (i.e., their state machines are different). Second, IGPs usually don't implement the rich set of attributes present in BGP. To preserve all the information about routes gleaned from eBGP sessions, it is best to run BGP sessions inside an AS as well.

The requirement that the iBGP routers be connected via a complete mesh limits scalability: a network with e eBGP routers and i other interior routers requires $e(e-1)/2 + ei$ iBGP sessions in a full-mesh configuration. While this quadratic scaling is not a problem for a small AS with only a handful of routers, large backbone networks typically have several hundred (or more) routers, requiring tens of thousands of iBGP sessions. This quadratic scaling does not work well in those cases. The following subsection discusses how to improve the scalability of iBGP sessions.

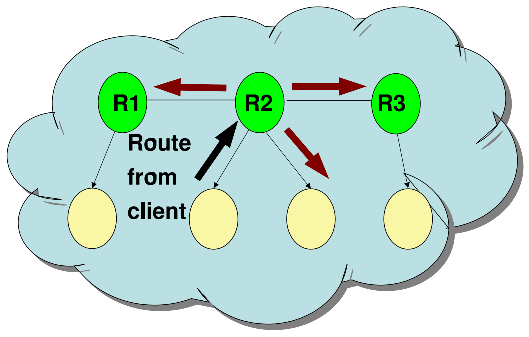
■ 3.3.4 iBGP Scalability

Two methods to improve iBGP scalability are currently popular. Both require the manual configuration of routers into some kind of hierarchy. The first method is to use *route reflectors* [1], while the second sets up *confederations* of BGP routers [23]. We briefly summarize the main ideas in route reflection here, and refer the interested reader to RFC 3065 [23] for a discussion of BGP confederations.

A route reflector is a BGP router that can be configured to have *client* BGP routers. A route reflector selects a single best route to each destination prefix and announces that route to all of its clients. An AS with a route reflector configuration follows the following rules in its route updates:



(a) Routes learned from non-clients are re-advertised to clients only.



(b) Routes learned from clients are re-advertised over all iBGP sessions.

Figure 3-6: Larger ASes commonly use route reflectors, which advertise some iBGP-learned routes, as described above. Directed edges between routers represent iBGP sessions from route reflectors to clients (e.g., router $R2$ is a route reflector with two clients). As in Figure 3-5, all routers re-advertise eBGP-learned routes over all iBGP sessions.

1. If a route reflector learns a route via eBGP or via iBGP from one of its clients, the it re-advertises that route over all of its sessions to its clients.

2. If a route reflector learns a route via iBGP from a router that is not one of its clients, then it re-advertises the route to its client routers, *but not over any other iBGP sessions*.

Having only one route reflector in an AS causes a different scaling problem, because it may have to support a large number of client sessions. More importantly, if there are multiple egress links from the AS to a destination prefix, a single route-reflector configuration may not use them all well, because all the clients would inherit the single choice made by the route reflector. To solve this problem, many networks deploy multiple route reflectors, organizing them hierarchically. Figure 3-6 shows an example route reflector hierarchy and how routes propagate from various iBGP sessions.

BGP route updates propagate differently depending on whether the update is propagating over an eBGP session or an iBGP session. An eBGP session is typically a *point-to-point* session: that is, the IP addresses of the routers on either end of the session are directly connected with one another and are typically on the same local area network. There are some exceptions to this practice (i.e., "multi-hop eBGP"), but directly connected eBGP sessions is normal operating procedure. In the case where an eBGP session is point-to-point, the next-hop attribute for the BGP route is guaranteed to be reachable, as is the other end of the point-to-point connection. A router will advertise a route over an eBGP session regardless of whether that route was originally learned via eBGP or iBGP.

On the other hand, an iBGP session may exist between two routers that are *not* directly connected, and it may be the case that the next-hop IP address for a route learned via iBGP is more than one IP-level hop away. In fact, as the next-hop IP address of the route is typically one of the border routers for the AS, this next hop may not even correspond to the router on the other end of the iBGP session, but may be several *iBGP* hops away. In iBGP, the routers thus rely on the AS's internal routing protocol (i.e., its IGP) to both (1) establish connectivity between the two endpoints of the BGP session and (2) establish

Route Attribute	Description
Next Hop	IP Address of the next-hop router along the path to the destination. On eBGP sessions, the next hop is set to the IP address of the border router. On iBGP sessions, the next hop is not modified.
AS path	Sequence of AS identifiers that the route advertisement has traversed.
Local Preference	This attribute is the first criteria used to select routes. It is not attached on routes learned via eBGP sessions, but typically assigned by the import policy of these sessions; preserved on iBGP sessions.
Multiple-Exit Discriminator (MED)	Used for comparing two or more routes from the same neighboring AS. That neighboring AS can set the MED values to indicate which router it prefers to receive traffic for that destination. *By default, not comparable among routes from different ASes.*

Table 3-1: Important BGP route attributes.

the route to the next-hop IP address named in the route attribute.

Configuring an iBGP topology to correctly achieve loop-free forwarding and complete visibility is non-trivial. Incorrect iBGP topology configuration can create many types of incorrect behavior, including persistent forwarding loops and oscillations [9]. Route reflection causes problems with correctness because not all route reflector topologies satisfy visibility (see [8] and references therein).

■ 3.3.5 Key BGP Attributes

We're now in a position to understand what the anatomy of a BGP route looks like and how route announcements and withdrawals allow a router to compute a forwarding table from all the routing information. This forwarding table typically has one chosen path in the form of the egress interface (port) on the router, corresponding to the next neighboring IP address, to send a packet destined for a prefix. Recall that each router implements the longest prefix match on each packet's destination IP address.

Exchanging Reachability: NEXT HOP Attribute

A BGP route announcement has a set of attributes associated with each announced prefix. One of them is the NEXT HOP attribute, which gives the IP address of the router to send the packet to. As the announcement propagates across an AS boundary, the NEXT HOP field is changed; typically, it gets changed to the IP address of the border router of the AS the announcement came from.

The above behavior is for eBGP speakers. For iBGP speakers, the first router that introduces the route into iBGP sets the NEXT HOP attribute to its so-called loopback address (the address that all other routers within the AS can use to reach the first router). All the other iBGP routers within the AS *preserve* this setting, and use the ASes IGP to route any packets destined for the route (in the reverse direction of the announcement) toward the NEXT HOP IP address. In general, packets destined for a prefix flow in the opposite direction to the route announcements for the prefix.

Length of AS Paths: ASPATH Attribute

Another attribute that changes as a route annoucement traverses different ASes is the ASPATH attribute, which is a *vector* that lists all the ASes (in reverse order) that this route announcement has been through. Upon crossing an AS boundary, the first router prepends the unique identifier of its own AS and propagates the announcement on (subject to its route filtering rules). This use of a "path vector"—a list of ASes per route—is the reason BGP is classified as a *path vector protocol*.

A path vector serves two purposes. The first is *loop avoidance*. Upon crossing an AS boundary, the router checks to see if its own AS identifier is already in the vector. If it is, then it discards the route announcement, since importing this route would simply cause a routing loop when packets are forwarded.

The second purpose of the path vector is to help pick a suitable path from among multiple choices. If no LOCAL PREF is present for a route, then the ASPATH length is used to decide on the route. Shorter ASPATH lengths are preferred to longer ones. However, it is important to remember that BGP isn't a strict shortest-ASPATH protocol (classical path vector protocols would pick shortest vectors), since it pays attention to routing policies. The LOCAL PREF attribute is always given priority over ASPATH. Many routes in practice, though, end up being picked according to shortest-ASPATH.

■ 3.3.6 Multi-Exit Discriminator (MED)

So far, we have seen the two most important BGP attributes: LOCAL PREF and ASPATH. There are other attributes like the multi-exit discriminator (MED) that are used in practice. The MED attribute is sometimes used to express route preferences between two ASes that are connected at multiple locations.[3]

When two ASes are linked at multiple locations, and one of them prefers a particular transit point over another for some (or all) prefixes, the LOCAL PREF attribute isn't useful. MEDs were invented to solve this problem.

It may be best to understand MED using an example. Consider Figure 3-7 which shows a provider-customer relationship where both the provider P and customer C have national footprints. Cross-country bandwidth is a much more expensive resource than local bandwidth, and the customer would like the provider to incur the cost of cross-country transit for the customer's packets. Suppose we want to route packets from the east coast (Boston) destined for D_{SF} to traverse P's network and not C's. We want to prevent P from transiting the packet to C in Boston, which would force C to use its own resources and defeat the purpose of having P as its Internet provider.

[3]The MED attribute is quite subtle, and tends to create about as many problems as it solves in practice, as multiple papers over the past few years have shown.

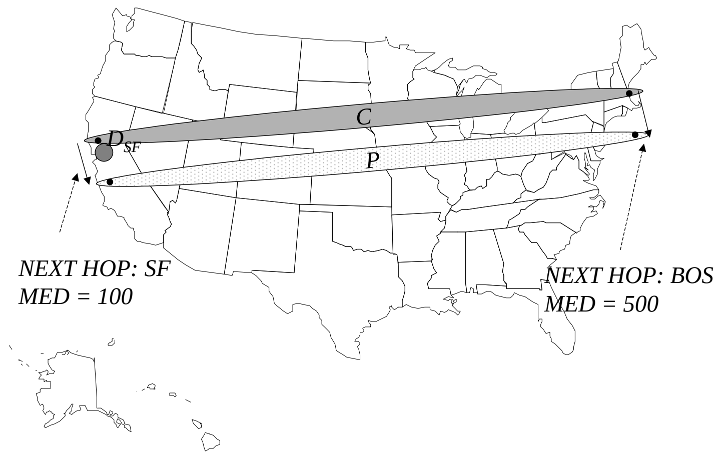


Figure 3-7: MED's **are useful in many situations, e.g., if** C **is a transit customer of** P**, to ensure that cross-country packets to** C **traverse** P**'s (rather than** C**'s wide-area network). However, if** C **and** P **are in a peering relationship,** MED **may (and often will) be ignored. In this example, the** MED **for** $D_S F$ **is set to 100 at the SF exchange point, and 500 in Boston, so** P **can do the right thing if it wants to.**

A MED attribute allows an AS, in this case C, to tell another (P) how to choose between multiple NEXT HOP's for a prefix D_{SF}. Each router will pick the smallest MED from among multiple choices coming from the same neighbor AS. No semantics are associated with how MED values are picked, but they must obviously be picked and announced consistently amongst the eBGP routers in an AS. In our example, a MED of 100 for the SF NEXT HOP for prefix D_{SF} and a MED of 500 for the BOS NEXT HOP for the same prefix accomplishes the desired goal.

An important point to realize about MED's is that they are usually ignored in AS-AS relationships that don't have some form of financial settlement (or explicit arrangement, in the absence of money). In particular, most peering arrangements ignore MED. This leads to a substantial amount of *asymmetric routing* in the wide-area Internet. For instance, if P and C were in a peering relationship in Figure 3-7, cross-country packets going from C to P would traverse P's wide-area network, while cross-country packets from P to C would traverse C's wide-area network. Both P and C would be in a hurry to get rid of the packet from their own network, a form of routing sometimes called *hot-potato routing*. In contrast, a financial arrangement would provide an incentive to honor MED's and allow "cold-potato routing" to be enforced.

The case of large content hosts peering with tier-1 ISPs is an excellent real-world example of cold-potato routing. For instance, an ISP might peer with a content-hosting provider to obtain direct access to the latter's customers (popular content-hosting sites), but does not want the hosting provider to free-load on its backbone. To meet this requirement, the ISP might insist that its MEDs be honored.

3.3.7 Putting It All Together: BGP Path Selection

We are now in a position to discuss the set of rules that BGP routers in an AS use to select a route from among multiple choices.

These rules are shown in Table 3-2, in priority order. These rules are actually slightly vendor-specific; for instance, the Router ID tie-break is not the default on Cisco routers, which select the "oldest" route in the hope that this route would be the most "stable."

Priority	Rule	Remarks
1	LOCAL PREF	Highest LOCAL PREF (§3.2.3). E.g., Prefer transit customer routes over peer and provider routes.
2	ASPATH	Shortest ASPATH length (§3.3.5) *Not* shortest number of Internet hops or delay.
3	MED	Lowest MED preferred (§**??**). May be ignored, esp. if no financial incentive involved.
4	eBGP > iBGP	Did AS learn route via eBGP (preferred) or iBGP?
5	IGP path	Lowest IGP path cost to next hop (egress router). If all else equal so far, pick shortest internal path.
6	Router ID	Smallest router ID (IP address). A random (but unchanging) choice; some implementations use a different tie-break such as the oldest route.

Table 3-2: How a BGP-speaking router selects routes. There used to be another step between steps 2 and 3 in this table, but it's not included in this table because it is now obsolete.

3.4 BGP in the Wild

From time to time, the fragility of the interdomain routing system manifests itself by disrupting connectivity or causing other anomalies. These problems are usually caused by misconfigurations, malice, or slow convergence. BGP is also increasingly used to allow customer networks to connect to multiple different providers for better load balance and fault-tolerance. Unfortunately, as we will see, BGP doesn't support this goal properly.

3.4.1 Hijacking Routes by Mistake or for Profit

One set of problems stems from the lack of *origin authentication*, i.e., the lack of a reliable and secure way to tell which AS owns any given prefix. The result is that it's possible for any AS (or BGP-speaking node) to originate a route for any prefix and possibly cause traffic from other networks sent to any destination in the prefix to come to the AS. Here are two interesting examples of this behavior:

1. YouTube diverted to Pakistan: On February 24 2008, YouTube, the popular video sharing web site, became unreachable to most people on the Internet. To understand what happened, a few facts are useful:

1. Internet routers use the route corresponding to the *longest prefix match* (LPM) to send packets; i.e., if the destination IP address matches more than one prefix entry in the

routing table, use the entry with the longest match between the destination address and the prefix.

2. `www.youtube.com` resolves to multiple IP addresses, all in the same "/24"; i.e., the first 24 bits of their IP addresses are all the same.

3. The Pakistani government had ordered all its ISPs to block access to YouTube. In general, there are two ways to block traffic from an IP address; the first is to simply drop all packets to and from that address, while the second is to *divert* all traffic going *to* the address to a different location, where one might present the user with a web page that says something like "We're sorry, but your friendly government has decided that YouTube isn't good for your mental well-being." The latter presumably is a better customer experience because it tells users what's going on, so they aren't in the dark, don't make unnecessary calls to customer support, and don't send a whole lot of traffic by repeatedly making requests to the web site. (Such diversion is quite common; it's used in many public WiFi spots that require a sign-on, for example.)

Pakistan Telecom introduced a /24 routing table entry for the range of addresses to which `www.youtube.com` resolves. So far, so good. Unfortunately, because of a misconfiguration (presumably caused by human error by a stressed or careless engineer) rather than malice, routers from Pakistan Telecom leaked this /24 routing advertisement to one of its ISPs (PCCW in Hong Kong). Normally, this leak should not have caused a problem *if* PCCW knew (as it should have) the valid set of IP prefixes that Pakistan Telecom owned. Unfortunately, perhaps because of another error or oversight, PCCW didn't ignore this route, and in fact presumably prioritized this route over all the other routes it already knew to the relevant addresses (recall that the typical rule is for customer routes to be prioritized over peer and provider routes). At this stage, computers inside PCCW and its customer's networks would've been "blackholed" from YouTube, and all traffic destined there would've been sent to Pakistan Telecom.

The problem was much worse because essentially the entire Internet was unable to get to YouTube. That's because of the LPM method used to find the best route to a destination. Under normal circumtances, there is no /24 advertised on behalf of YouTube by its ISPs; those routes are contained in advertisements that cover a (much) wider range of IP addresses. So, when PCCW's neighbors saw PCCW advertising a more-specific route, they followed the rules and imported those routes into their routing tables, readvertising them to their respective neighbors, until the entire Internet (except, presumably, a few places such as YouTube's internal network itself) had this poisoned routing table entry for the IP addresses in question.

This discussion highlights a key theme about large systems: when they fail, the reasons are complicated. In this case, the following events all occurred:

1. The Pakistani government decided to censor a particular site because it was afraid that access would create unrest.

2. Pakistan Telecom decided to divert traffic using a /24 and leaked it in error.

3. PCCW, which ought to have ignored the leaked advertisement, didn't.

4. The original correct advertisements involved less-specific prefixes; in this case, had they been /24s as well, the problem may not have been as widespread.

5. Pakistan Telecom inadvertently created a massive traffic attack on itself (and on PCCW) because YouTube is a very popular site getting lots of requests. Presumably the amount of traffic made diagnosis difficult because packets from tools like `traceroute` might not have progressed beyond points of congestion, which might have been upstream of Pakistan Telecom.

   It appears that the first indication of what might have really happened came from an investigation of the logs of routing announcements that are available to various ISPs and also publicly. They showed that a new AS had started originating a route to a prefix that was previously always been originated by others.

   This observation suggests that a combination of public "warning systems" that maintain such information and flag anomalies might be useful (though there are maany legitimate reasons why routes often change origin ASes too). It also calls for the maintenance of a correct registry containing prefix to owner AS mappings; studies have shown that current registries unfortunately have a number of errors and omissions.

Far from being an isolated incident, such problems (black holes and hijacks) arise from time to time.[4] There are usually a few serious incidents of this kind every year, though selectively taking down a popular site tends to make the headlines more easily. There are also several smaller-scale incidents and anomalies that show up on a weekly basis.

2. Spam from hijacked prefixes: An interesting "application" of routing hijacks using principles similar to the one discussed above is in transmitting hard-to-trace email spam. On the Internet, it is easy for a source to spoof a source IP address and pretend to send packets from an IP address that it hasn't legitimately been assigned. Email, however, runs atop TCP, which uses a feedback channel for acknowledgments, and email is a bi-directional protocol. So, untraceable source spoofing is a bit trickier because the spoofer must also be able to *receive* packets at the spoofed IP address.

An ingenious solution to this problem is for the bad guy to convince one or more upstream ISPs (which might themselves be a bit sketchy or just look the other way because they're getting paid to ignore questionable behavior) to pay attention to BGP routing announcements. The bad guy temporarily hijacks a portion of the IP address space, typically an unassigned one (though it doesn't have to be), by sending announcements about that prefix. When that announcement propagates, routers in the Internet know how to reach the corresponding addresses. The bad guy initiates a large number of email connections, dumps a whole lot of spam, and then after 45 minutes or an hour simply withdraws the advertised route and disappears. Later, when one tries to trace a path to the offending source IP addresses of the spam, there is absolutely no trace! A recent study found that 10% of spam received at a "spam trap" (a domain with a number of fake email receiver addresses

[4]The "AS 7007" incident in 1997 was perhaps the first massive outage that partitioned most of Sprint's large network and customer base from the rest of the Internet. In that incident, a small ISP in Florida (AS number 7007) was the party that originated the misconfigured route leak, but other ISPs were also culpable in heeding those unrealistic route advertisements.

Finding	Time-frame
Serious routing pathology rate of 3.3%	Paxson 1995
10% of routes available less than 95% of the time	Labovitz et al. 1997
Less than 35% of routes available 99.99% of the time	Labovitz et al. 1997
40% of path outages take 30+ minutes to repair	Labovitz et al. 2000
5% of faults last more than 2 hours, 45 minutes	Chandra et al. 2001
Between 0.23% and 7.7% of "path-hours" experienced serious 30-minute problems in 16-node overlay	Andersen et al. 2001
Networks dual-homed to Tier-1 ISPs see many loss bursts on route change	Wang et al. 2006
50% of VoIP disruptions are highly correlated with BGP updates	Kushman 2006

Table 3-3: Internet path failure observations, as reported by several studies.

set up to receive spam to analyze its statistics) came from IP addresses that corresponded to such route hijacks [18].

Of course, if all BGP announcements were logged and carefully maintained, one can trace where messages sent from hijacked routes came from, but oftentimes these logs aren't maintained correctly at a fine-enough time granularity. In time, such logs will probably be maintained at multiple BGP vantage points in the Internet, and at that time those wishing to send untraceable garbage may have to invent some other way of achieving their goals.

■ 3.4.2 Convergence Problems

With BGP, faults take many seconds to detect and it may take several minutes for routes to con- verge to a consistent state afterwards. Upon the detection of a fault, a router sends a withdrawal message to its neighbors. To prevent routing advertisements from propagating through the entire network and causing routing table calculations for failures or routing changes that might only be transient, each router implements a route flap damping scheme by not paying attention to frequently changing advertisements from a router for a prefix. Damping is believed by many to improve scalability, but it also increases convergence time.

In practice, researchers have found that wide-area routes are often unavailable. The empirical observations summarized in Table 3-3 are worth noting.

■ 3.4.3 Multi-homing

BGP allows an AS to be multi-homed, supporting multiple links (and eBGP sessions) between two ASes, as well an AS connecting to multiple providers. In fact, there is no restriction on the AS topology that BGP itself imposes, though prevalent routing policies restrict the topologies one observes in practice. Multi-homing is used to tolerate faults and balance load. An example is shown in Figure 3-8, which shows the topology and address blocks of the concerned parties. This example uses *provider-based addressing* for the customer, which allows the routing state in the Internet backbones to scale better because transit providers can aggregate address blocks across several customers into one or a small number of route announcements to their respective providers.

Achieving scalable and efficient multi-homing with BGP is still an open research ques-

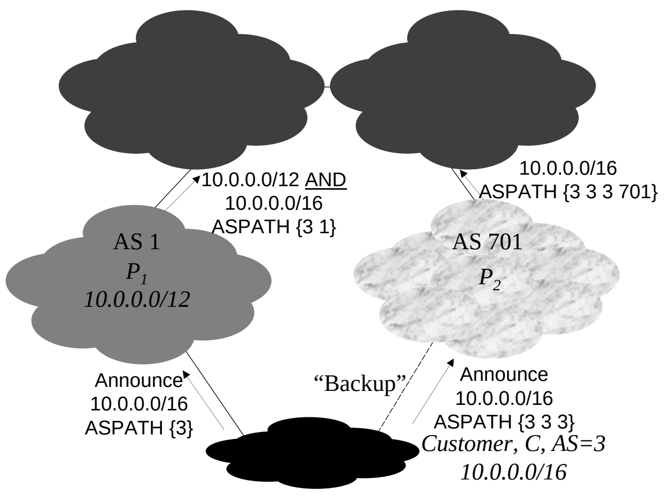


Figure 3-8: Customer *C* is multi-homed with providers P_1 and P_2 and uses provider-based addressing from P_1. *C* announces routes to itself on both P_1 and P_2, but to ensure that P_2 is only a backup, it might use a hack that pads the ASPATH attribute as shown above. However, notice that P_1 must announce (to its providers and peers) *explicit* routes on both its regular address block *and* on the customer block, for otherwise the path through P_2 would match based on longest prefix in the upstream ASes.

tion. As the number of multi-homed customer networks grows, the stress (in terms of routing churn, convergence time, and routing table state) on the interdomain routing system will increase. In addition, the interaction between LPM, failover and load balance goals, and hacks like the AS path padding trick are complex and often cause unintended consequences.

■ 3.5 Summary

The Internet routing system operates in an environment of "competitive cooperation", in which different independently operating networks must cooperate to provide connectivity while competing with each other. It must also support a range of routing policies, some of which we discussed in detail (transit and peering), and must scale well to handle a large and increasing number of constituent networks.

BGP, the interdomain routing protocol, is actually rather simple, but its operation in practice is extremely complex. Its complexity stems from configuration flexibility, which allows for a rich set of attributes to be exchanged in route announcements. There are a number of open and interesting research problems in the area of wide-area routing, relating to failover, scalability, configuration, correctness, load balance (traffic engineering), security, and policy specification. Despite much activity and impressive progress over the past few years, interdomain routing remains hard to understand, model, and make resilient.

■ Acknowledgments

These notes have evolved over the past few years. I thank Nick Feamster for a productive collaboration on various Internet routing problems that we had during his PhD research with me. Figures 3-5 and 3-6 are taken from Nick's dissertation. Thanks also to Jennifer Rexford and Mythili Vutukuru for several discussions on interdomain routing.

CHAPTER 4
Network Mobility

These are notes, *in outline form*, on network mobility.

■ 4.1 The Problem

The Internet architecture and the IP layer assume that the IP address of an host doesn't change over the duration of its connections. Unfortunately, because hosts move, host movement isn't dealt with properly. We start by developing a taxonomy for various kinds of movement, then highlight the main idea ("separating identity from location"), and then discuss how Mobile IP and HIP work.

■ 4.2 Taxonomy

In general, we can divide host movement into two categories: *nomadicity* and *mobility*. An example of nomadicity is a host operating at one network location, then being suspended and moved to a different location, and reconnecting to the Internet from there. Some network applications continue to work as if nothing happened, while others fail. By mobility, we refer to a more continuous movement in which a host moves while being connected to the network and we want network applications to continue working even as the network attachment points change.

Nomadicity. Applications that are stateless work easily across such movement. Ones that are maintain state may fail; for example, an SSH session because the underlying TCP socket state has a binding with the IP address. On the other hand, applications with plenty of state can be made to work. Take a web site like Amazon for example: if you're in the middle of a shopping session and stop, when you log in next to Amazon, it's possible to continue from where you left off before, and it's quite unlikely that it ever forgets what's in your shopping cart. The reason is that the site knows who you are because it uses a combination of login identifiers and web cookies, and that information doesn't change no matter where you connect to the site from or when.

Mobility. We distinguish between two kinds of mobility depending on the way in which the network might support it: *infrastructure mobility* and *ad hoc* mobility. Infrastructure mobility refers to a system that has fixed nodes that mobiles "attach" to as they move (the fixed nodes may be connected to the rest of the network using wired links or wireless backhaul, but their network addresses don't change). Ad hoc mobility has no such infrastructure. We won't deal with ad hoc mobility here.

■ 4.3 Key idea

The Amazon example mentioned above highlights a key concept in supporting mobility: because so many things change with movement, to work properly, we need something that doesn't change, which can then be used as a way to retrieve whatever state was previously kept. That is, we need an *identity*, an invariant name that does not change with movement. In a layered design, for a session at any given layer to work correctly across changes in the lower layers, the higher layer needs an invariant way to name and retrieve the state of the session. That invariant name (identity) must be independent of lower-layer parameters that might change, in particular, independent of network location.

With multi-homing, one has more than one network address, so a location-independent identity has the added advantage that it can more easily handle multi-homed hosts. Increasingly, many mobiles are multi-homed because they run multiple radio protocols and can connect to multiple wireless networks.

For the rest of this lecture, our problem will be to ensure that unicast network applications continue to work properly even as the network attachment points (IP addresses) of one or both of the participants changes.

■ 4.4 Possible solutions

There are at least three possible approaches to the problem posed above, distinguished by the layer in which they operate:

1. Session layer: TCP connections break when one of the IP addresses of the communicating peers changes. If that happens, use a session layer library that applications link against to hide this break from the application, and have the session layer restart TCP connections as needed. It's fairly straightforward to support one node moving; when both change IP addresses concurrently, more work is needed. This problem is the *double jump* problem. One approach is to have the session layer, when it finds itself unable to connect to the original host, do a DNS lookup, and have the hosts use dynamic DNS to update their DNS bindings whenever an IP address changes. This approach also requires each session layer to be able to identify itself to the other using some kind of identifier, perhaps created securely at the beginning of the session. That name (like the DNS name) must be invariant across movement.

2. Transport layer: One can migrate TCP connections by modifying TCP[1].

[1]A.C. Snoeren and H. Balakrishnan, "An end-to-end approach to host mobility," Proc. ACM MOBICOM, pages 155-166, Boston, MA, Aug. 2000.

3. Network layer: Mobile IP and HIP are proposals to solve the problem at the network layer and "hide" mobility even from TCP.

■ 4.5 Mobile IP

The invariant name (identity) is the home address, which never changes for a host at least as long as connections are active. The "locator" is whatever IP address the mobile host is currently at.

Mobile IP in its basic form uses triangle routing: packets go from a fixed host toward the home network, where they are intercepted by a home agent acting on behalf of the mobile host using a mechanism like proxy ARP. Then, they are *tunneled* using IP-in-IP to the mobile host's *foreign address*, which refers to the address of an entity that has agreed to deliver the packet to the mobile. Typically, this foreign agent entity either runs on the mobile host itself or is one IP hop awy from it and serves as its router.

Some questions to think about for Mobile IP:

1. Which messages need to be authenticated?
2. How does the mobile discover the foreign ("care-of") address?
3. How does the mobile host register a foreign (care-of) address with the home agent?
4. Does the foreign agent have any shared secret or trust association with the mobile?
5. Does the home agent have any shared secret or trust association with the mobile?
6. For route optimization to work, where the fixed host can directly use the mobile's current care-of address by being told what it is by the home agent, does there have to be any shared secret or trust between the home agent and the fixed host?
7. How do packets from the mobile host to the fixed host propagate when the foreign network deploys ingress source filtering, preventing source addresses that aren't assigned to the foreign network from being used?
8. Does Mobile IP deal with the address stealing attack?
9. Does Mobile IP deal with the address flooding attack?
10. Does Mobile IP in its basic form suffer from potential packet loss during mobility events not caused by congestion or packet corruption?
11. Does Mobile IP really need a foreign agent that runs on a router separate from the mobile host?
12. Does Mobile IP handle the double jump problem when both parties move concurrently?
13. Does Mobile IP handle multi-homed mobile hosts properly by allowing connections to use some or all interfaces at the same time?

■ 4.6 Host Identity Protocol (HIP)

Here, the identity is a unique end-point identifier, a public key that we'll call the EID. The locator is the current IP address. TCP connections bind to the EID, not IP address. DNS lookups return an EID. (DOA shares this part of its design with HIP, except for some details concerning the format of the EIDs).

EIDs are looked-up using some infrastructure and return IP addresses.

1. How does HIP propose dealing with the double jump problem?
2. How well does it handle multi-homing?
3. How does it deal with address stealing? With address flooding?
4. Unlike Mobile IP, it is able to achieve reasonably efficient paths that are close or identical to unicast paths between a fixed and mobile host without requiring any trust between commmunicating hosts. What property of HIP enables that?

Chapter 5
Coping with Best-Effort: Reliable Transport

This lecture discusses how end systems in the Internet cope with the best-effort properties of the Internet's network layer. We study the key reliability techniques in TCP, including TCP's cumulative acknowledgment feedback, and two forms of retransmission upon encountering a loss: *timer-driven* and *data-driven* retransmissions. The former relies on estimating the connection round-trip time (RTT) to set a timeout, whereas the latter relies on the successful receipt of later packets to initiate a packet recovery without waiting for a timeout. Examples of data-driven retransmissions include the *fast retransmission* mechanism and *selective acknowledgment (SACK)* mechanism. Because it is helpful to view the various techniques used in TCP through the lens of history, we structure our discussion in terms of various TCP variants.

We then discuss application-level framing (ALF) [5], an approach that helps achieve *selective reliability* by being more integrated with the application than TCP is with applications.

■ 5.1 The Problem: A Best-Effort Network

A best-effort network greatly simplifies the internal design of the network, but implies that there may be times when a packet sent from a sender does not reach the receiver, or does not reach the receiver in a timely manner, or is reordered behind packets sent later, or is duplicated. A transport protocol must deal with these problems:

1. *Losses*, which typically because of congestion, packet corruption due to noise or interference, routing anomalies, or link/path failures.

   Congestion losses are a consequence of using a packet-switched network for statistical multiplexing, because a burst of packets arriving into a queue may exhaust buffer space. It is usually a bad idea to maintain a really large queue buffer, because an excessively large buffer only causes packets to be delayed and doesn't make them move through the network any faster! We will discuss the factors governing queue buffer space during our discussions on router design and congestion control.

2. *Variable delays* for packet delivery.

3. Packet *reordering*, which typically arises because of multiple paths between endpoints, or pathologies in router implementations.

4. *Duplicate* packets, which occasionally arrive at a receiver because of bugs in network implementations or lower-layer data retransmissions.

Many applications, including file transfers, interactive terminal sessions, etc. require reliable data delivery, with packets arriving *in-order*, in the same order in which the sender sent them. These applications will benefit from a *reliable transport protocol*, which is implemented in a layer below the application (typically in the kernel) and provides an in-order, reliable, byte-stream abstraction to higher-layer applications via a well-defined "socket" interface. TCP provides such semantics.

However, this layered in-order TCP model is not always the best way to transport application data. First, many applications can process data that arrives out-of-order. For example, consider the delivery of video or audio streams on the Internet. The loss of a packet in a video or audio stream may not be important, because the receiver is capable of "hiding" this loss by averaging neighboring samples, with only a small degradation in quality to the end user. However, it might be the case that some video packets are important, because they contain information useful both to the current frame and to subsequent frames. These lost packets may need to be recovered. Hence, *selective reliability* is a useful goal.

Second, if data is provided to the application *only* in-order as with TCP, a missing packet causes a stall in data delivery until it is retransmitted by the sender, after which the receiver application has to process a number of packets. If the receiver application cares about interactive response, or if receiver processing before the user looks at the presented information is a bottleneck, a TCP-like approach is sub-optimal.

The above observations motivate a different (non-layered) approach to transport protocol design, called *ALF, application-level framing*.

This lecture discusses both approaches to transport protocol design. Our goal, as mentioned before, is to cope with the vagaries of a best-effort network.

■ 5.2 Transmission Control Protocol (TCP)

The TCP service model is that of an in-order, reliable, duplex, byte-stream abstraction. It doesn't treat datagrams as atomic units, but instead treats bytes as the fundamental unit of reliability. The TCP abstraction is for *unicast* transport, between two network attachment points (IP addresses, not end hosts). "Duplex" refers to the fact that the same connection handles reliable data delivery in both directions.

In general, reliable transmission protocols can use one or both of two kinds of techniques to achieve reliability in the face of packet loss. The first, called *Forward Error Correction* (FEC), is to use redundancy in the packet stream to overcome the effects of some losses. The second is called *Automatic Repeat reQuest* (ARQ), and uses packet retransmissions. The idea is for the sender to infer the receiver's state using acknowledgments (ACKs) it gets, and determine that packets are lost if an ACK hasn't arrived for a while, and retransmit if necessary.

In TCP, the receiver periodically[1] informs the sender about what data it has received via *cumulative ACKs*. For example, the sequence of bytes:

```
1:1000 1001:1700 2501:3000 3001:4000 4001:4576
```

received at the receiver will cause the ACKs

```
1001 1701 1701 1701 1701
```

to be sent by the receiver after each segment arrival. Each ACK acknowledges all the bytes received in-sequence thus far, and tells the sender what the next expected in-sequence byte is.

Each TCP ACK includes in it a receiver-advertised window that tells the sender how much space is available in its socket buffer at any point in time. This window is used for end-to-end *flow control*, and should not be confused with *congestion control,* which is how resource contention for "inside-the-network" router resources (bandwidth, buffer space) is dealt with. Flow control only deals with making sure that the sender doesn't overrun the receiver at any stage (it's a lot easier than congestion control, as you can see). We will study congestion control in more detail in later lectures.

TCP has two forms of retransmission upon encountering a loss: *timer-driven* retransmissions and *data-driven* retransmissions. The former relies on estimating the connection round-trip time (RTT) to set a timeout value; if an ACK isn't received within the timeout duration of sending a segment, that segment is retransmitted. On the other hand, the latter relies on the successful receipt of later packets to initiate a packet recovery without waiting for a timeout.

Early versions of TCP used only timer-driven retransmissions. The sender would estimate the average round-trip time of the connection, and the absence of an ACK within that timeout duration would cause the sender to retransmit the entire window of segments starting from the last cumulative ACK. Such retransmissions are called *go-back-N* retransmissions, and are quite wasteful because they end up re-sending an entire window (but they are simple to implement). Early TCPs did not have any mechanism to cope with congestion, either, and ended up using fixed-size sliding windows throughout a data transfer.

The evolution of TCP went roughly along the following lines (what follows isn't a timeline, but a sequence of important events):

1. In the mid-1980s, portions of the Internet suffered various congestion-collapse episodes. In response, Van Jacobson and Mike Karels developed a set of techniques to mitigate congestion [11]. Among these were better round-trip time (RTT) estimators, and the "slow start" and "congestion avoidance" congestion control schemes (which we will study in detail in a later lecture).

2. Karn and Partridge's solution to the "retransmission ambiguity" problem, in which a TCP sender could not tell the difference between an ACK for an original transmission and a retransmission [13].

[1]Every time it receives a segment; modern TCP receivers implement a *delayed ACK* policy where they should ACK only on every other received segment as long as data arrives in sequence, and must acknowledge at least once every 500 ms if there is any unacknowledged data. BSD-derived implementations use a 200 ms "heartbeat" timer for delayed ACKs.

3. The TCP timestamp option, which provide more accurate RTT estimation for TCP and eliminates the retransmission ambiguity problem.

4. Coarse-grained timers: After various attempts, researchers realized that developing highly accurate timers for reliable transport protocols was a pipe-dream, and that timeouts should be conservative.

5. TCP Tahoe, which combined the Jacobson/Karels techniques mentioned above with the first data-driven retransmission scheme for TCP, called *fast retransmission*. The idea is to use duplicate ACKs as a signal that a data segment has been lost.

6. TCP Reno, which extended fast retransmissions with *fast recovery*, correcting some shortcomings in Tahoe. Reno also implemented a "header prediction" scheme for more efficient high-speed implementation (this mechanism has no bearing on TCP's reliability functions).

7. TCP NewReno, developed by Janey Hoe, an improvement to the fast recovery scheme, allowing multiple data losses in large-enough windows to be recovered without a timeout in many cases.

8. TCP Selective Acknowledgments (SACK), standaradized in RFC 2018, which extends TCP's cumulative ACKs to include information about segments received out-of-order by a TCP receiver [15].

There have been other enhancements proposed to TCP over the past few years, such as TCP Vegas (a congestion control method), various optimizations for wireless networks, optimizations for small windows (e.g., RFC 3042), etc. We don't discuss them here.

The result of all these proposals is that current TCPs rarely timeout if window sizes are large enough (more than 6 or 7 segments), unless most of the window is lost, or most of the returning ACKs are lost, or retransmissions for lost segments are also lost. Moreover, current TCP retransmissions adhere to two important principles, both of which are important for sound congestion control (which we will study in a later lecture):

P1 Don't retransmit early; avoid spurious retransmissions. The events of the mid-1980s showed that congestion collapse was caused by the retransmission of segments that weren't lost, but just delayed and stuck in queues. TCP attempts to avoid spurious retransmissions.

P2 Conserve packets. TCP attempts to "conserve" packets in its data-driven retransmissions. It uses returning duplicate ACKs as a cue signifying that some segments have been received, and carefully determines what segments should be sent, if any, in response to these duplicate ACKs. The idea is to avoid burdening the network with excessive load during these loss periods.

The rest of this lecture discusses the techniques mentioned before, roughly in the order given there. We start with TCP timers.

■ 5.2.1 TCP timers

To perform retransmissions, the sender needs to know when packets are lost. If it doesn't receive an ACK within a certain amount of time, it assumes that the packet was lost and retransmits it. This is called a *timeout* or a *timeout-triggered* retransmission. The problem is to determine how long the timeout period should be.

What factors should the timeout depend on? Clearly, it should depend on the connection's round-trip time (RTT). To do this, the sender needs to estimate the RTT. It obtains samples by monitoring the time difference between sending a segment and receiving a positive ACK for it. It needs to do some averaging across all these samples to maintain a (running) average. There are many ways of doing this; TCP picks a simple approach called the Exponential Weighted Moving Average (EWMA), where $srtt = \alpha \times r + (1 - \alpha)srtt$. Here, r is the current sample and *srtt* the running estimate of the *smoothed* RTT. In practice, TCP implementations use $\alpha = 1/8$ (it turns out the precise value doesn't matter too much).

We now know how to get a running average of the RTT. How do we use this to set the retransmission timeout (RTO)? One option, an old one used in the original TCP specification (RFC 793), is to pick a multiple of the smoothed RTT and use it. For example, we might pick $RTO = \beta * srtt$, with β set to a constant like 2. Unfortunately, this simple choice doesn't work too well in preventing *spurious* retransmissions when slow start is used on certain network paths, leading to bad congestion effects, because the principle of "conservation of packets" will no longer hold true.

A nice and simple fix for this problem is to make the RTO a function of both the average and the standard deviation. In general, the tail probability of a spurious retransmission when the RTO is a few standard deviations away from the mean is rather small. So, TCP uses an RTO set according to: $RTO = srtt + 4 \times rttdev$, where *rttdev* is the mean linear deviation of the RTT from its mean. I.e., *rttdev* is calculated as $rttdev = \gamma \times dev + (1 - \gamma) \times rttdev$, where $dev = |r - srtt|$. In practice, TCP uses $\gamma = 1/4$.

These calculations aren't the end of the TCP-timer story. TCP also suffers from a significant *retransmission ambiguity* problem. When an ACK arrives for a segment the sender has retransmitted, how does the sender know whether the RTT to use is for the original transmission or for the retransmission? This question might seem like a trivial detail, but in fact is rather vexing because the RTT estimate can easily become meaningless and throughput can suffer. The solution to this problem is surprisingly simple—ignore samples that arrive when a retransmission is pending or in progress. I.e., don't consider samples for any segments that have been retransmitted [13].

The modern way of avoiding the retransmission ambiguity problem is to use the TCP *timestamp option* (RFC 1323), which most current (and good) TCP implementations adopt. Here, the sender uses 8 bytes (4 for seconds, 4 for microseconds) and stamps its current time on the segment. The receiver, in the cumulative ACK acknowledging the receipt of a segment, simply *echoes* the sender's stamped value. When the ACK arrives, the sender can do the calculation trivially by subtracting the echoed time in the ACK from the current time. Note that it now doesn't matter if this was a retransmission or an original transmission; the timestamp effectively serves as a "nonce" for the segment.

The other important issue is deciding what happens when a retransmission times out. Obviously, because TCP is a "fully reliable" end-host protocol, the sender must try again. But rather than try at the same frequency, it takes a leaf out of the contention resolution protocol literature (e.g., Ethernet CSMA) and performs *exponential backoffs* of the retransmission timer.

A final point to note about timeouts is that they are extremely *conservative* in practice. TCP retransmission timers are usually (but not always) coarse, with a granularity of 500 or 200 ms. This is a big reason why spurious retransmissions are rare in modern TCPs, but also why timeouts during downloads are highly perceptible by human users.

5.2.2 Fast retransmissions

Because timeouts are expensive (in terms of killing throughput for a connection, although they are a necessary evil from the standpoint of ensuring that senders back-off under extreme congestion), it makes sense to explore other retransmission strategies that are more responsive. Such strategies are also called *data-driven* (as opposed to *timer-driven*) retransmissions. Going back to the earlier example:

```
1:1000 1001:1700 2501:3000 3001:4000 4001:4576
```

with ACKs

```
      1001      1701      1701      1701      1701
```

It's clear that a sequence of repeated ACKs are a sign that something strange is happening, because in normal operation cumulative ACKs should monotonically increase. Repeated ACKs in the middle of a TCP transfer can occur for three reasons:

1. Window updates. When the receiver finds more space in its socket buffer, because the application has read some more bytes, it can send a window update even if no new data has arrived.

2. Segment loss.

3. Segment reordering. This could have happened, for example, if datagram 1701-2500 had been reordered because of different routes for different segments.

Repeated ACKs that aren't window updates are called *duplicate* ACKs or *dupacks*. TCP uses a simple heuristic to distinguish losses from reordering: if the sender sees an ACK for a segment more than three segments larger than a missing one, it assumes that the earlier (unacknowledged) segment has been lost. Unfortunately, cumulative ACKs don't tell the sender which segments have reached; they only tell the sender what the last in-sequence byte was. So, the sender simply *counts* the number of dupacks and infers that if it see three or more dupacks, that the corresponding segment was lost. Various empirical studies have shown that this heuristic works pretty well, at least on today's Internet, where reordering TCP segments is discouraged and there isn't much "parallel" ("dispersity") routing.

5.2.3 Fast Recovery

A TCP sender that performs a "fast retransmission" after receiving three duplicate ACKs must reduce its congestion window, because the loss was most likely due to congestion (we will study congestion control in depth in a later lecture). TCP Tahoe senders go into "slow start", setting the window size to 1 segment, and sending segments from the next new ACK. (If no new ACK arrives, then the sender times out.)

The problem with this approach is that the sender may send segments that are already received at the other end.

A different approach, called *fast recovery*, cuts the congestion window by one-half on a fast retransmission. As duplicate ACKs beyond the third one arrive for the window in which the loss occurred, the sender waits until half the window has been ACKed, and then sends one *new* segment per subsequent duplicate ACK. Fast recovery has the property that at the end of the recovery, assuming that not too many segments are lost in the window, the congestion window will be one-half what it was when the loss occurred, and that many segments will be in flight.

TCP Reno uses this combination of fast retransmission and fast recovery. This TCP variant, which was the dominant version in the 1990s, can recover from a small number of losses in a large-enough window without a timeout (the exact details depend on the pattern of losses; for details, see [7]).

In 1996, Hoe proposed an enhancement to TCP Reno, which subsequently became known as *NewReno*. The main idea here is for a sender to remain in fast recovery until all the losses in a window are recovered.

■ 5.2.4 Selective acknowledgment (SACK)

When the bandwidth-delay product of a connection is large, e.g., on a high-bandwidth, high-delay link like a satellite link, windows can get pretty large. When multiple segments are lost in a single window, TCP usually times out. This causes abysmal performance for such connections, which are colloquially called "LFNs" (for "Long Fat Networks" and pronounced "elephants," of course). Motivated in part by LFNs, selective ACKs (SACKs) were proposed as an embellishment to standard cumulative ACKs. SACKs were standardized a few years ago by RFC 2018, after years of debate.

Using the SACK option, a receiver can inform the sender of up to three maximally contiguous blocks of data it has received. For example, for the data sequence:

```
1:1000 1001:1700  2501:3000  3001:4000 4577:5062 6000:7019
```

received, a receiver would send the following ACKs and SACKs (in brackets):

```
      1001     1701        1701        1701      1701        1701
                     [2501-3000]  [2501-4000] [4577-5062; [6000-7019;
                                                2501-4000]  4577-5062;
                                                             2501-4000]
```

SACKs allow LFN connections to recover many losses in the same window with a much smaller number of timeouts. While SACKs are in general a Good Thing, they don't always prevent timeouts, including some situations that are common in the Internet today. One reason for this is that on many paths, the TCP window is rather small, and multiple losses in these situations don't give an opportunity for a TCP sender to use data-driven retransmissions.

5.2.5 Some other issues

There are a few other issues that are important for TCP reliabilty.

1. **Connection setup/teardown.** At the beginning of a connection, a 3-way handshake synchronizes the two sides. At the end, a teardown occurs.

2. **Segment size.** How should TCP pick its segment size for datagrams? Each side picks a "default" MSS (maximum segment size) and exchanges them in the SYN exchange at connection startup; the smaller of the two is picked.

   The recommended way of doing this, to avoid potential network fragmentation, is via *path MTU discovery.* Here, the sender sends a segment of some (large) size that's smaller than or equal to its interface MTU (when the IP and link-layer headers are added) with the "DON'T FRAGMENT (DF)" flag in the IP header. If the receiver gets this segment, then clearly every link en route could support this datagram size because there was no fragmentation. If not, and an ICMP error message was received by the sender from a router saying that the packet was too big and would have been fragmented, the sender tries a smaller segment size until one works.

3. **Low-bandwidth links.** Several TCP segments are small in size and are mostly comprised of headers; examples include telnet packets and TCP ACKs[2]. To work well over low-bandwidth links, TCP header compression can be done (RFC 1144). This takes advantage of the fact that most of the fields in a TCP header are either the same or are predictably different from the previous one. This allows a 40-byte TCP+IP header to be reduced to as little as 3-6 bytes.

In summary, TCP provides an in-order, reliable, duplex, byte-stream abstraction. It uses timer-driven and data-driven retransmissions; the latter provides better performance under many conditions, while the former ensures correctness.

5.3 Application Level Framing (ALF)

Perhaps the best way to understand the ALF idea is by example. Consider the problem of designing a protocol to stream video on the Internet. In most (but not all) video compression formats including the MPEG variants, there are two kinds of compressed frames—*reference frames* and *difference frames*. A reference frame is compressed by itself, whereas a difference frame is compressed in terms of its difference from previous frames (in practice, some frames may be compressed as a difference not just from a previous frame but also a succeeding frame). Because video scenes typically don't change dramatically from frame to frame, a difference frame usually compresses a lot better than a reference frame. However, if packets are lost in the stream and not recovered, a stream that uses only one reference frame followed by only difference frames suffers from the *propagation of errors* problem, because the subsequent frames after a lost packet end up cascading the errors associated with missing data.

[2]TCP does allow data to be piggybacked with ACKs and most data segments have valid ACK fields acknowledging data in the opposite direction of the duplex connection.

One approach to transporting streaming video is over TCP, but this has two problems. First, depending on the nature of the loss, recovery may take between one round-trip time and several seconds (if a long timeout occurs). Because not all lost packets need to be recovered, the receiver application might choose interactive presentation of whatever frames are available, over waiting for missing data that has only marginal value. However, some packets (e.g., in reference frames) may need to be *selectively* recovered. Second, when using TCP, a lost packet would cause all of the later packets in the stream to *wait* in the receiver's kernel buffers without being delivered to the application for processing. If receiver application processing is a bottleneck, a lost packet causes the application's interactive performance to degrade.

The above discussion suggests that what is needed is:

1. Out-of-order data delivery from the transport protocol to the receiver application, and

2. The ability for the receiver application to request the selective retransmission of specific data items.

One solution to this problem is to ask each application designer to design an application-specific transport protocol anew for each new application. This approach is fine if there were only one such application or if different applications had no functions to share, but not if there were common functions that one could provide in a way that different applications could benefit from. It seems quite plausible that a range of audio and video applications (whether streaming or for interactive conferencing) will all require roughly similar (but customizable) ways of dealing with packet loss, delay variations (jitter), and variable network conditions (data rates and latencies). Thus, it makes sense to architect a protocol that various applications can easily customize to their needs. ALF is a principle that helps us with this task.

To understand the problem better, let's consider whether it's possible to get what we need by hacking TCP to make the TCP receiver push out-of-order packets to the receiver application. The reason is that there is no shared vocabulary between the application and TCP to name data items! The application may have it's own naming method for data items (e.g., this packet is from frame 93 and has to be displayed at time t and corresponds to block number b within the frame), but the TCP knows nothing of these semantics. TCP's name for the corresponding packet would be a sequence number and length. Unfortunately, the receiver application has no idea what the TCP name for the data item corresponds to in the application's vocabulary.

What is required is a *common vocabulary* between the application and the transport to name data items. A convenient way to do this is to use the application's own name in the transport protocol, which means that an *application data unit* (what the application thinks of as independenly processible data items) and the *protocol data unit* (what the transport thinks of as indepedently transmittable and receivable data items) are one and the same thing. By making these two data units equivalent, when an out-of-order data unit arrives at the receiver transport protocol, it can deliver it to the receiver application and have it be understood because the delivered data is named in a manner that the application understands.

Selective reliability is obtained easily with ALF, because a missing application data unit that is deemed important by the receiver can be requested by the receiver application to the receiver transport library. In turn, the transport library sends a retransmission request to the sender library, and the required data is re-sent to the receiver either from the sender library's buffer (if it has one) or more likely via a callback to the sender application itself. Again, our common naming between transport and applications of the data items is instrumental in making this approach work.

It is hard to design an ALF mechanism in a strictly layered fashion where the ALF transport is in the kernel and the application is across a strict protection boundary. Instead, the best way to realize ALF is to design the transport as a *library* that sender and receiver applications link to, and use the library to send and receive data items.

Several ALF applications have been implemented in practice, including for video transport, image transport, shared whiteboards and collaborative tools, and multicast conferencing. In the context of Internet audio and video, the Real-time Transport Protocol (RTP) is a customizable protocol inspired by ALF that many applications use. RTP has an extensible header format that includes both a fixed set of fields as well as arbitrary application-defined extensions. To help with control functions, RTP applications use the Real-time Control Protocol (RTCP), which allows applications to connect to various URLs corresponding to streams, perform functions like pausing, rewind, etc., and provide feedback for adapting video quality and performing rate adaptation.

In summary, ALF is an elegant and useful principle that implementations can use to achieve out-of-order data delivery to applications.

Chapter 6
End-to-end Congestion Control

Congestion management is a fundamental problem in networking because the way to achieve cost-effective and scalable network designs is by *sharing* the network infrastructure. Managing shared resources in a careful way is therefore a critical problem. This lecture covers the principles and practice of (unicast) congestion control, including linear congestion control algorithms and the practically important details of TCP congestion control. Background readings for this lecture include Chiu and Jain's paper introducing linear controls [4] and Jacobson's TCP congestion control paper [11].

■ 6.1 The problem

The first question one should ask at this stage is: "What resources are being shared that need to be managed?" To answer this question, let's think about some of the properties of best-effort networks. Recall that the model at any router is for each packet to be processed and forwarded along one of several possible output links, at a rate determined by the rated bandwidth of that link. In the meantime, more packets, which in general could belong to any other flow,[1] could arrive at the router (recall that this is one of the key consequences of asynchronous multiplexing).

What happens to these packets? The router tries to accomodate them in its queues and process them, but if there isn't enough space in its queue, some of them may be dropped. Thus, packet queues build up when the link is busy and demand for the link outstrips the available link bandwidth, and packets are generally dropped when the queues get full. Figure 6-1 shows a simple network with connections between senders S_i and receivers R_i via a 100 Kbps bottleneck link.[2]

The above discussion makes it clear that there are two resources that flows contend for in a network:

1. **Link bandwidth.** The network has to decide how to apportion bandwidth between different flows. Network routers may also decide to prioritize certain types of pack-

[1]Or connection. A flow is just a generalization of a connection; the distinction between the two is irrelevant for the purposes of this discussion.

[2]We use Mbps for Megabits per second, Kbps for Kilobits per second, and bps for bits per second.

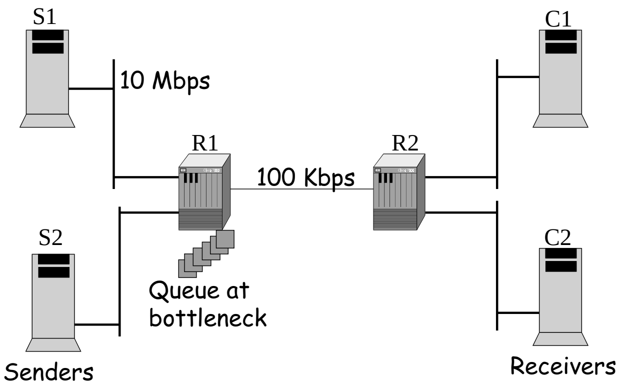


Figure 6-1: A simple network topology showing how sharing causes resource contention. Many connections between senders and receivers contend for the bottleneck 100 Kbps link.

ets (e.g., latency-sensitive audio or interactive `telnet` packets) over others (e.g., electronic mail).[3]

2. **Queue space.** When the router decides that a packet has to be dropped because it is running out of queue space (also known as buffer space), which packet should it drop? The arriving one? The earliest one? A random one? And when should it decide to drop packets: only when the queue is full, or sooner than that? Waiting too long to before dropping packets only serves to increase packet delays, and it may be advantageous to drop occasional packets even when the queue isn't full.

What happens if we don't manage network resources well? For one thing, the available bandwidth might end up being greatly under-utilized even when there is demand for it, causing economic heartburn. Most often, however, network designers end up provisioning for a certain amount of "expected" *offered load,* and then have to deal with overload, or *congestion.*

■ 6.1.1 Understanding congestion

A network link is said to be *congested* if contention for it causes queues to build up and for packets to start getting dropped. At such a time, demand for link bandwidth (and eventually queue space), outstrips what is available. For this reason, many network resource management problems are also called *congestion control* or *congestion management* problems, since the efficient management of congestion implies an efficient management of the above network resources.

[3]Note that there's a "type-of-service (TOS)" field, now also called the "differentiated services" field, in the IP header; routers sometimes look at this field to decide how to prioritize packets.

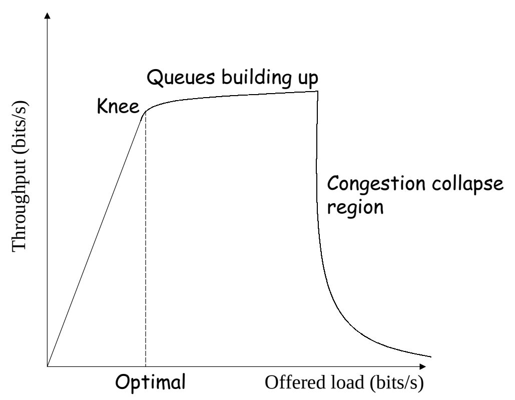


Figure 6-2: Schematic view of throughput vs. offered load, showing the optimal operating point and congestion collapse region.

Congestion occurs due to overload, and is exacerbated by network heterogeneity. Heterogeneity is a wonderful property of the Internet, which allows 14.4Kbps wireless links to co-exist with and connect to 10 Gigabit/s Ethernets. However, the transition between links of very different speeds implies that sources connected to high-bandwidth links can't simply blast their packets through, since the bottleneck available bandwidth to a destination can be quite a bit smaller. Sources should learn what rates are sustainable and adapt to available bandwidth; this should be done dynamically because there are few paths on the Internet whose conditions are unchanging.

There are many examples of how overload causes congestion. For example, one possible way to achieve high throughput in a network might be to have all the sources blast packets as fast as they can, so that bottleneck network links run at close to 100% utilization. While this seems reasonable, a little thought shows that this approach is self-defeating. All it really accomplishes are long packet queues and resultant end-to-end delays, as well as increased packet loss rates, which for a reliable end-to-end transport layer would cause a large number of retransmissions. This is therefore quite the wrong thing to do to obtain good network throughput. This example also demonstrates the fundamental tension between achieving high link utilizations on the one hand (by transmitting data rapidly), and low delays and loss rates on the other (which increase if data is sent too rapidly).

This notion of how increasing offered load to achieve high utilization is at odds with packet losses and delays is illustrated in Figure 6-2. This figure shows a schematic view of throughput as a function of the offered load on the network. Initially, at low levels of offered load, throughput is roughly proportional to offered load, because the network is under-utilized. Then, throughput plateaus at a value equal to the bottleneck link bandwidth because packets start getting queued.

After a while, when offered load is increased even further, the throughput shows a "cliff" effect, starting to decrease and eventually go to zero! Now, all the competing flows are sending data rapidly, but no user is getting any useful work done (low network throughput). The network has just suffered *congestion collapse*—a situation where observed throughput *decreases* in response to an increase in offered load. While seemingly impossible, this catastrophic result is quite real and occurs in a number of scenarios; for example, the mid-1980s did see such an event, with large portions of the then Internet coming to a standstill.[4]

Congestion collapse is an example of an *emergent property* of a complex system, one that shows up rather suddenly in certain situations when the network size and load exceeds a certain amount. Congestion control protocols attempt to reduce or eliminate the occurrence of such nasty situations. In particular, the goal for a good network resource management algorithm would be to operate near the left knee of the curve in Figure 6-2, by modulating the sources to provide the appropriate optimal offered load.

■ 6.2 Goals and constraints

At first sight, it might seem that network resource management problems are not that hard to solve. A particularly simple approach would be for a centralized network manager to arbitrate and decide on how link bandwidths and queue buffers are apportioned between different flows, and perform hard-edged admission control (we will see what this means when we talk about Internet quality of service; or see Chapter 4 of the 6.033 lecture notes for what this means) when new flows are started. Of course, this "solution" is impractical for networks larger than only a small number of nodes.

A second approach might be to throw some money at the problem and hope it will go away. In particular, we could decide to provision lots of queue space in our network routers, so the worst that happens upon overload is that packets get queued and are rarely discarded. Unfortunately, all this does is to increase the delays incurred by new packets and does not improve the throughput or efficiency of the system at all. This is a little bit like long lines at an amusement park, where people are waiting to go on a popular roller coaster ride; adding more and more space for people to wait in line doesn't improve system throughput at all! The real purpose of queues is primarily to absorb bursts of data that arrive from senders, without causing long delays. We need queues, but we should be careful how many packets we queue at each router; persistent queueing of packets is not a good thing.

What are the goals of network resource management in large, heterogeneous networks like the Internet? There are several desirable properties that any good solution must possess, which make the problem interesting and challenging:

- **Scalability.** The primary goal of Internet congestion control is to scale well. This scalability is along several different dimensions: the number of flows sharing a bottleneck link, the range of bandwidths over which the protocols must work (e.g., 8-9 orders of magnitude), and the range of delays over which the protocols must work (e.g., 3-4 orders of magnitude). It is crucial to remember that the goal is *not* to optimize congestion control performance at any given point along these axes, but to do well enough and work reasonably well *in the entire range*.

[4]The primary cause for this was a large number of spurious packet retransmissions together with the absence of any mechanism by which a TCP connection could figure out a good window size at any point in time.

- **Efficiency.** We would like our solutions to lead to high network utilization. What this translates to is high end-to-end application throughputs and low end-to-end delays for all the competing flows.

- **Fairness.** The next goal is fairness—we want fair allocations of resources to competing flows. After all, it is possible to achieve high efficiency by starving all flows but one, obviously an undesirable end result.

  It turns out that coming up with the right metric for fairness is tricky, and there are many possible definitions. One possible way of measuring fairness is via a *fairness index*, which is a function of the mean and variance of the throughputs achieved by different streams. If $x_1, x_2, \ldots, x_n$ are the throughputs of n competing flows, the fairness index f is defined as

  $$f = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2}.$$

  Clearly, $1/n \leq f \leq 1$, with a smaller value of f signifying a larger degree of unfairness in allocations.

  One common, but by no means the only, notion of fairness is *max-min fairness*. The allocation of a resource is *max-min fair* if, and only if:

  1. no user receives more than their request, ρ_i.
  2. no other allocations satisfying #1 has higher minimum allocation
  3. condition #2 recursively holds as we remove the minimal user and reduce the total resource accordingly.

  This condition reduces to $\mu_i = MIN(\mu_{fair}, \rho_i)$, where μ_i is the allocation for user i and ρ_i is user i's request. It assumes that all users have equal rights to the resource, although a weighted version can be analogously defined.

  We emphasize that in practice achieving a mathematically precise notion of fairness isn't as important as something that works "reasonably well." (This is a point on which reasonable people in the networking field disagree! It has also been subject to extensive and continuing debate.) We will use this notion of fairness when we talk about fair queueing a couple of lectures from now.

- **Distributed operation.** For even moderately sized networks, we can't assume the existence of a centralized oracle deciding what different flows should do.

- **Convergence.** Because of the inherently distributed nature of the system, each node (router or end-host) only has incomplete knowledge of the traffic in the network, with which it must act. An important goal of a practical algorithm is that it must not display large oscillatory behavior, and must try to converge towards high efficiency and fairness.

■ 6.3 End-to-end flow Control

A special case of the congestion problem is when the receiver is the bottleneck (e.g., because it has a slow processor or an application that's reading data off the transport stack too slowly). We devote this section to a brief discussion of this problem.

Consider an extremely simple network, a sender connected to a receiver over a fast 100 Mbps link via a fast switch. Suppose the sender is a fast 450 MHz PC, while the receiver is a slower 100 MHz machine. Assume that any sent data arrives in an 8-KByte receive buffer for the receiver to process. Clearly, if the sender sent data faster than the receiver could process, its buffer would overflow and packets will be lost. The receiver and sender need to negotiate how much data the sender is allowed to send at any point in time, and the sender must not violate this agreement. This agreement is called *flow control,* and is the result of negotiation between the end-to-end transport layers at the receiver and sender.

Most protocols implement flow control via windows. Here, every acknowledgment packet (ack) to the sender also tells the sender how much the sender can send at this point without overflowing receiver buffers. The sender can now be reasonably certain that as long as this agreement is not violated, no packets will be dropped because of overflow at the receiver.

Ensuring flow control is relatively easy—just have a simple agreement between sender and receiver on how much each window can be. However, this does not ensure that packets will not be lost anywhere else, because the network could easily be the bottleneck and become congested. For example, this can easily happen for the topology shown in Figure 6-1, especially when several flows are sharing the 100 Kbps link. To function well, the network needs a *congestion control* mechanism, which is significantly harder because neither the sender nor the receiver has precise knowledge about where the bottleneck might be or what the bottleneck bandwidth is.

Thus, flow control is an end-to-end protocol to avoid overwhelming the receiver application and is *different* from congestion control, which uses a combination of techniques to avoid overwhelming the routers and links in the network. Flow control is usually achieved via a periodic window-based negotiation between sender and receiver. It is also significantly easier to solve than congestion control, as we will soon discover. Networks need *both* to function well.

■ 6.4 End-to-end congestion control

End-to-end congestion control protocols address two fundamental questions:

1. How do senders react to congestion when it occurs, as signified by dropped packets?
2. How do senders determine the available capacity for a flow at any point in time?

When congestion occurs, all the flows that detect it must reduce their transmission rate. If they don't do so, then the network will remain in an unstable state, with queues continuing to build up. To move toward the optimal operating point of Figure 6-2, delays and loss rates must decrease, which can only happen if the congested flows react to congestion by decreasing their sending rates. A *decrease algorithm* determines how this is done by the end-to-end layer.

When packets are successfully being received and acknowledged by the receiver, the sender can deduce that the current rate is sustainable by the network. However, it does not know if the network can sustain still higher rates because available bandwidth varies with time as users start and depart, and so it needs to carefully probe for spare capacity to use. An *increase algorithm* determines how this is done by the end-to-end transport layer.

Before we can talk about the details of specific decrease and increase algorithms, we need to understand how the end-to-end transport layer at the sender modulates its transmission rate. We study a class of protocols called *window-based* protocols for doing this, since it is the most common way of doing it today (e.g., TCP uses this technique). Here, the sender maintains a window, also called the *congestion window,* or `cwnd`, that it uses to decide how many *unacknowledged* packets it can have in the network at any time. This window is different from the flow control window that we discussed before, which is negotiated by the sender and receiver to avoid overrunning the receiver's buffers. In contrast, `cwnd` is dynamically adjusted by the sender to try and track the state of congestion and available bandwidth in the network.

When the sender has packets to transmit, it sends out only as many as the *minimum* of these two windows permit, thereby ensuring that the number of outstanding, unacknowledged packets does not overrun the receiver nor (hopefully) overwhelm the network. In the rest of this discussion, we assume that the flow control window is larger than *cwnd*, or that the network rather than the receiver is the performance bottleneck. Because it takes roughly one round-trip time (`rtt`) to transmit a window's worth of packets, the equivalent transmission rate for such a window-based protocol is the ratio of `cwnd/rtt` packets per second.

A different approach from window-based congestion control is *rate-based* congestion control. Here, there's no window to modulate the transmission rate (although there usually is a flow control window to ensure that the receiver buffer does not overrun). Unlike in window-based protocols where incoming ACKs "clock" data packets (we discuss this idea in more detail when we talk about TCP), a rate-based protocol uses an explicit timer to decide when packets are sent on the network. In fact, it isn't enough to simply use a rate in such protocols; what's usually needed is both a burst size (number of packets or bytes) and a burst interval (over which the rate will be measured).[5]

As described above, both schemes react to individual indications of congestion. An alternate approach is to monitor *loss rate* p and set a rate r as a function of p. We will look at this idea in Section 6.5.

■ 6.4.1 Decrease and increase algorithms

We are now in a position to discuss the details of the decrease and increase algorithms. It is clear that there are an infinite number of ways to decrease and increase windows, and so we restrict ourselves to a simple, practical class of control algorithms called *linear* algorithms. Here, if w_i is the value of `cwnd` in the ith round-trip since the start of the connection,

$$w_{i+1} = aw_i + b$$

for real-numbered constants a and b. a and b are of course different for the decrease and increase cases, and ensure that w_{i+1} is smaller than or greater than w_i respectively.

A good way to view such controls is using *phase plots* (see the Chiu and Jain paper for an example). If there are n sources sharing the bottleneck, this plot has n axes; we consider

[5]One can also design protocols that combine both window- and rate-based ideas.

$n = 2$ for simplicity and clarity in the rest of our discussion. Each axis plots the window size (or rate) of a connection (w_i), normalized to the maximum window (rate) value. Thus, $w_1 + w_2 = 1$ is the "optimal utilization line" and $w_1 = w_2$ is the "equi-fairness line." Points below the $w_1 + w_2 = 1$ line are when the network is under-utilized. The goal is to get to the intersection of the two lines; if this happens then the system will have converged to an efficient and fair allocation.

Chiu and Jain show a neat result in this formalism: under a *synchronized feedback assumption* that the occurrence of congestion is signaled to all connections sharing the bottleneck, an *additive-increase, multiplicative-decrease* or AIMD strategy converges to efficiency and fairness. Furthermore, no other linear control does.

The intuition behind this follows from the phase-plot picture. Additive increase *improves fairness* and efficiency, while multiplicative decrease moves the system from overload to under-utilization *without altering fairness*. Thus, no matter where we start in the phase plot, AIMD pushes the system to $w_1 = w_2 = 0.5$. In contrast, additive-decrease *reduces fairness* in the decrease phase while multiplicative-increase, multiplicative-decrease (MIMD) does not ever improve fairness.

One can show all this algebraically too, but the pictorial arguments are a lot more intuitive and prettier!

Thus, with AIMD, upon congestion

$$w_{i+1} = aw_i, 0 < a < 1,$$

and while probing for excess capacity,

$$w_{i+1} = w_i + b, 0 < b \ll w_{max},$$

where w_{max} is the largest possible window size (equal to the product of end-to-end delay and bottleneck bandwidth). The intuition behind these rules is that when the network is congested, the queue lengths start to increase exponentially with time, and the decrease must be multiplicative to ensure that this dissipates quickly. The increase algorithm must move gingerly and also try to move toward fair allocations of bandwidth, which is achieved by an additive algorithm.

For example, TCP sets $a = 0.5$ and $b = 1$. That is, when the sender detects congestion, it reduces its window to half the current value, and when an entire round-trip goes by and all the packets were successfully received (as indicated by all the acks arriving safely), it increases the window by 1 packet, carefully probing for a little more bandwidth to use.

In general, this technique for increasing and decreasing windows is also called *additive-increase/multiplicative-decrease (AIMD)* congestion control. Pseudo-code for this algorithm using TCP's constants is shown below.

```
// On detecting congestion via packet loss,
cwnd := cwnd / 2;
// When the current window is fully acknowledged (roughly one round-trip),
cwnd := cwnd + 1;
```

6.4.2 Conservation of packets

Good window-based congestion control protocols follow the *conservation of packets* principle, which says that for a connection "in equilibrium," the packet flow must be conservative (this is a term from fluid mechanics). A flow is in equilibrium if it is running with a full window of packets in transit, equal to the product of available bandwidth and delay (in the absence of any persistent queueing). Of course, the end-to-end transport layer knows neither the available bandwidth nor the non-queueing delay with any great accuracy, which is why it needs to *estimate* them via its increase algorithm and round-trip calculations. Note that it need not explicitly calculate either parameter for its congestion control, and can rely instead on the observation that in the absence of congestion, an entire window is acknowledged by the receiver in one round-trip duration. Then, the end-to-end congestion control protocol can ensure that packet conservation isn't violated by ensuring that a lost packet is not prematurely retransmitted (this would violate conservation because a packet presumed to be lost would still be in transit at the time of retransmission). An excessive number of premature retransmissions does in fact lead to congestion collapse, and this was the cause of the mid-1980s disaster.

TCP uses a widely applicable principle called *self-clocking* to send packets. Here, the sender uses acks as an *implicit* clock to regulate new packet transmissions. Clearly, the receiver can generate acks no faster than data packets can reach it through the network, which provides an elegant self-clocking mechanism by which the sender can strobe new packets into the network.

Three possible reasons for packet conservation to *not* happen in a transport protocol:

1. Connection doesn't reach equilibrium.

2. Premature injection of new packets by sender.

3. Equilibrium can't be reached because of resource limits along path (or wild oscillations occur).

6.4.3 How TCP works: slow start, congestion avoidance and control

We now have sufficient background to use the above principles and apply them to a real protocol, TCP.

We start by observing that packet conservation applies to a connection in equilibrium, one which has reached its sweet-spot window size and is probing for more around it (and reacting to congestion using multiplicative decrease). But when a new connection starts, how does it know what this sweet spot is, and how does it get to it to get its ack-triggered "clock" going? TCP achieves this using a technique called *slow start.* Here, TCP sets its `cwnd` to 1 and sends one packet[6]. When an ack arrives for it, it increases its window size by 1 and now sends 2 packets. Now, when an ack arrives for *each* of these packets[7], *cwnd* increases by 1. That is, `cwnd` is now equal to 4. In the slow start phase, `cwnd` increases by 1 for each incoming ack. In this phase, TCP uses the ack clock to not only clock packets out, but also to open up its window.

[6]In reality, TCP maintains all its windows in terms of number of bytes, not packets. This only complicates the book-keeping and does not alter the principles or algorithms significantly.

[7]We assume that the receiver acknowledges every packet.

When do we stop this process? We would like to stop it when the window size reaches the equilibrium value (roughly the bandwidth-delay product), but of course we don't know what that is. So, we stop it at an estimate called the slow start threshold (`ssthresh`) or on packet loss (because that's a sign of congestion), after which we move into the linear increase phase described above. Of course, if we stop it due to congestion, `cwnd` multiplicatively decreases.

How fast does the window grow toward equilibrium in slow start? If the current `cwnd` is w_i, the sender transmits w_i packets within one `rtt`. If they all reach, w_i acks come back to the sender, *each of which* increases `cwnd` by 1. So, after all w_i packets are acknowledged (one `rtt` later), $w_{i+1} = 2w_i$. Thus, the window increase in slow start is *multiplicative,* increasing exponentially with time. Thus, slow start isn't really that slow, and reaches an equilibrium window size estimate W in $\log_2 W$ round-trips.

Once slow start completes, TCP uses the AIMD principle to adjust its `cwnd`. The only tricky part is in the increase pseudo-code, shown below.

```
        // When an ack arrives for a packet,\\
cwnd := cwnd + 1/cwnd;\\
```

The $1/cwnd$ increment follows from our wanting to increase the congestion window by 1 after one round-trip. Because TCP wants the increase to be smooth rather than abrupt, we approximate it by increasing by $1/cwnd$ *when each packet is acknowledged.* When `cwnd` acks arrive, the effective increase is approximately 1, after about one round-trip. And `cwnd` acks would indeed arrive in the absence of congestion. This phase of TCP is also called the *linear phase* or *congestion avoidance* phase.

This also serves to point out the dual purpose of a TCP ACK—on the one hand, it is used to detect losses and perform retransmissions, and on the other, it is used to clock packet transmissions and adjust `cwnd`. Thus, any variability or disruption in the ACK stream will often lead to degraded end-to-end performance.

The final twist to TCP's congestion control algorithm occurs when a large number of losses occur in any window. Taking this as a sign of persistent congestion, TCP takes a long timeout before trying to resume transmission, in the hope that this congestion would clear up by then[8]. At this point, it has obviously lost its self-clock, and so goes back into slow start to resume; it sets its slow start threshold, `ssthresh`, to one-half of `cwnd` at the time this persistent congestion occurred. Finally, for each failed retransmission, TCP performs *exponential backoff* for successive retransmissions by doubling the interval between attempts, similar to the *random exponential backoff* retry technique used in the Ethernet[9]. One can show that with a large user population, you need at least an exponential backoff on congestion (or in the Ethernet case, collision) to be safe from congestion collapse.

Pseudo-code for TCP achieving its self-clock via slow start is shown below.

```
        if (timeout) // connection in progress and many packets lost
                ssthresh := cwnd/2; // set ssthresh on timeout
        else {       // connection start-up
```

[8]It is actually untrue that the number of lost packets in a window is correlated with the degree of congestion in a drop-tail FIFO network. The real reason is that when a number of losses happen, TCP loses faith in its ack clock and decides to start over.

[9]But without the randomization that the Ethernet uses to avoid synchronized retries by multiple stations.

```
        ssthresh := MAX_WIN; // max window size, 64 KBytes or more
}
cwnd := 1;
```

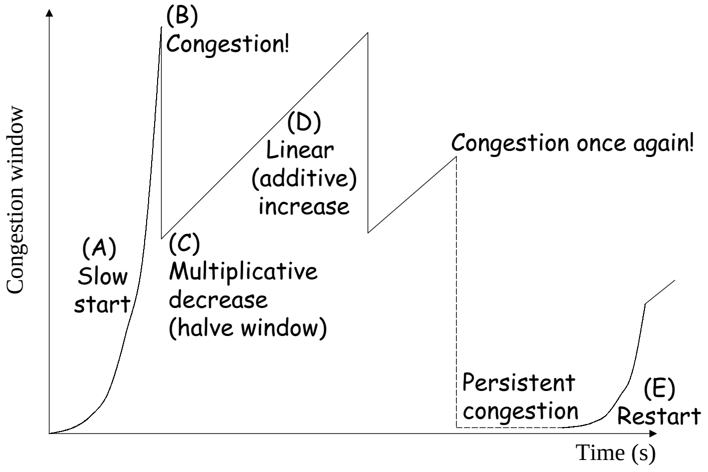


Figure 6-3: The time evolution of TCP's congestion window.

We are now in a position to sketch the evolution of the congestion window, `cwnd`, as a function of time for a TCP connection (Figure 6-3). Initially, it starts at 1 and in slow start doubles every `rtt`, (A) until congestion occurs (B). At this point, if the number of lost packets isn't large, the sender halves its transmission rate (C), and goes into the linear increase phase probing for more bandwidth (D). This sawtooth pattern continues, until several packets are lost in a window and TCP needs to restart its self-clocking machinery (E). The process now continues as if the connection just started, except now slow start terminates when `cwnd` reaches `ssthresh`.

We conclude this section by observing that the soundness of TCP's congestion management is to a large degree the main reason that the Internet hasn't crumbled in recent years despite the phenomenal growth in the number of users and traffic. Congestion management is still a hot research area, with additional challenges being posed by Web workloads (large numbers of short flows) and by real-time audio and video applications for which TCP is an inappropriate protocol.

■ 6.5 Throughput vs. loss-rate

A key result of the TCP AIMD mechanism is that the long-term TCP throughput λ is related to the loss rate p it observes. To first order, $\lambda \propto 1/\sqrt{p}$. This is sometimes called the "TCP-friendly equation" since flows that have a similar reaction to a static packet loss probability may compete well with TCP over long time scales.

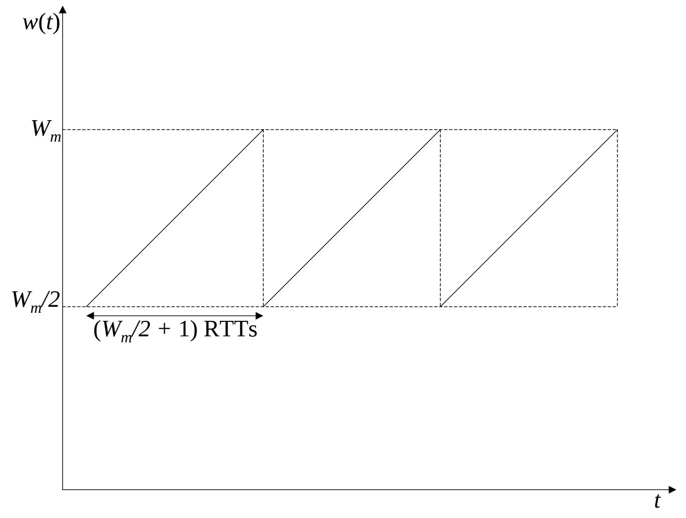


Figure 6-4: The steady-state behavior of the TCP AIMD congestion window.

This result can be used for an "equation-based" approach to congestion control, where the sender uses receiver feedback to monitor an average loss rate, and set the transmission rate according to some function of the loss rate. To compete well with TCP, it would use the TCP formula (more exact versions of which, incorporating timeouts and other TCP details, have been derived).

One way of showing that the TCP throughput in steady-state varies as the inverse-square-root of the loss rate uses the "steady-state model" of TCP. Figure 6-4 shows this. Here, we assume one TCP connection going through the AIMD phases, losing a packet when the window size reaches W_m. Upon this loss, it assumes congestion, and reduces its window to $W_m/2$. Then, in each RTT, it increases the window by 1 packet until the window reaches W_m, whereupon a window-halving occurs.

Consider the time period between two packet losses. During this time, the sender sends a total of $\frac{W_m}{2} + (\frac{W_m}{2} + 1) + (\frac{W_m}{2} + 2) + \ldots + (\frac{W_m}{2} + \frac{W_m}{2}) = \frac{3}{8}W_m^2 + O(W_m)$ packets. Thus, $p = \frac{8}{3W_m^2}$.

Now, what's the throughput, λ, of such a TCP connection? Its window size varies linearly between $\frac{W_m}{2}$ and W_m, so its average value is $\frac{3}{4}W_m$. Thus, $\lambda \approx \frac{3}{4}\frac{W_m}{RTT}$ packets/s.

These two expressions for p and λ imply that $\lambda \propto \frac{1}{\sqrt{p}}$.

This result can also be shown for a simplistic Bernoulli loss model. In this model, we assume that each packet can be lost independently with a probability p and that even if multiple packets are lost in the same RTT (window), only one window-halving occurs. (This assumption is correct for modern TCP's, which consider congestion "epochs" of 1 RTT, so multiple losses within an RTT are considered to be due to the same congestion event.)

In this model, suppose $W(t)$ is the current window size. A short time δt later, $W(t)$ can be one of two values, depending on whether a loss occurs (with probability p) or not (with probability $1-p$).

$$W(t+\delta t) = \begin{cases} W(t) + \frac{\delta t}{W_t \cdot RTT} & \text{w.p. } 1-p \\ \frac{W(t)}{2} & \text{w.p. } p \end{cases}$$

This assumes a "fluid" model of packet and ACK flow. The $\frac{1}{W}$ comes in because in additive-increase the window increases by 1 every RTT, and there are W ACKs in each RTT (assuming no delayed ACKs). What we want is the average value of the window, which is evolving stochastically according to the above equation. If the average is $\bar{w}$, then the *drift* in $\bar{w}$ with time should be 0. This drift is equal to $\frac{1-p}{\bar{w}} - p\frac{\bar{w}}{2}$. Setting this to 0 gives $\bar{w} \propto \frac{1}{\sqrt{p}}$.

6.6 Buffer sizing

An important problem in network design is figuring out how much packet buffering to put in the queues at each router. This is an important issue. Too little, and you don't accommodate bursts of packets very well. These bursts are commonplace in Internet traffic, in part because of the nature of applications, and in part because of the way in which TCP probes for spare capacity using its increase algorithms (slow start and additive-increase). On the other hand, too much buffering is both useless and detrimental to performance; the only thing you get is longer packet delays and larger delay variations. (This is akin to long lines at amusement parks—they don't improve the throughput of the system at all and only serve to frustrate people by causing really long waits.)

It turns out that for TCP AIMD sources, a buffer size equal to the product of the link bandwidth and round-trip delay achives nearly 100% link utilization.[10] To see why, consider a simple model of one TCP connection. Suppose the window size is W_m when it incurs a loss. Clearly, $W_m = P + Q$ where P is the "pipe size" or bandwidth-delay product and Q is the queue size. P is the amount of data that can be outstanding and "in the network pipe" that does not sit in any queue.

Now, when congestion is detected on packet loss and the window reduces to $\frac{W_m}{2}$ this round-trip sees a throughput of *at most* $\frac{W_m}{2 \times RTT}$. For 100% utilization, this should equal μ, the link bandwidth. The smallest value of $\frac{W_m}{2}$ that achieves this is $\mu \times RTT$, which means that Q should be set to $\mu \times RTT$ (i.e., P).

6.7 Other approaches

We now outline two other broad approaches to congestion control. The next two lectures deal with the first of these, router-based congestion control.

6.7.1 Router-based congestion control

Router-based solutions are implemented in the switches and routers in the network. The simplest form of router support for congestion control is displayed in FIFO (first-in-first-out) queues at routers, which forward packets in the same order they were received, and

[10]In practice, with enough connections and desynchronized access to the bottleneck, a smaller buffer can provide nearly 100% utilization.

drop packets on overload. In such a network, most of the burden for congestion control is pushed to the end-hosts, with the routers providing only the barest form of congestion feedback, via packet drops. The advantage of this minimalism is that the interior of the network (i.e., the routers) don't need to maintain any per-connection state and are extremely simple. The onus for dealing with congestion is pushed to the sender, which uses packet losses as a signal that congestion has occurred and takes appropriate action.

This is essentially the approach taken in most of the Internet today: most IP routers implement FIFO scheduling and *drop-tail* queue space management, dropping all packets from the "tail" of a full queue. This approach, of keeping the interior of the network largely stateless and simple, is in keeping with the fundamental design philosophy of the Internet, and is one of the reasons for the impressive scalability of the Internet with increasing numbers of users and flows.

In recent years, a number of queue management algorithms have been proposed for deploying in Internet routers, and more will be developed in the future. For example, routers needn't wait until their queues were full before dropping packets. In the next lecture, we will study some simple but effective ways in which routers can support and enhance end-to-end congestion control via good queue management (i.e., deciding when and which packets to drop).

In addition, a number of sophisticated scheduling algorithms, which segregate flows into different categories and arbitrate on which category gets to send the next packet, have also been developed in recent years. For instance, routers could segregate the packets belonging to each flow into separate queues and process them in a "fair" way, using a technique similar to round-robin scheduling (a variant of this, developed some years ago, is called *fair queueing*). This approach is of course significantly more complex than simple FIFO queueing, because it requires per-flow state and processing to handle packets. We will study fair queueing and some variants in a couple of lectures from now. Note, however, that even if these scheduling and queue management schemes were widely deployed in the network, the end-to-end transport layer would still have to eventually deal with congestion and decide how to modulate its transmissions to avoid congestion to achieve high throughput and network efficiency.

■ 6.7.2 Pricing

A radically different approach to network resource management would be to rely on pricing mechanisms and the associated economic incentives and disincentives to effect efficient and fair network use. Such approaches are based on the observation that bandwidth and queue space are simply economic commodities and that the economic principles of supply and demand ought to apply to them. Indeed, to some extent, telephone companies do control overload by variable-rate pricing (e.g., prices are cheaper during slack hours and more expensive during peak hours). For asynchronously multiplexed data networks, no one has really figured out how to do this and if it will work. In particular, while pricing is a promising mechanism to provision resources over sufficiently long time scales (e.g., weeks or months), relying exclusively on network service providers to deploy flexible pricing structures to keep up with the wide variation in traffic load over small time scales is unwise.

CHAPTER 7 Router-Assisted Congestion Control

In the previous lecture, we studied the principles of end-to-end congestion control and the practice of one of them (TCP). End-to-end congestion control using TCP is an extremely successful model for handling congestion problems, that approach does have some limitations. This lecture is the first of three that discusses router-assisted resource management. We will focus on explicit congestion notification (ECN) and active queue management schemes (specifically, RED). These notes do not discuss XCP, which is a generalization of ECN in which routers send rate information to sources.

■ 7.1 Motivation and Overview

The rationale for investigating router-assisted congestion control include the following:

1. Routers are the place where congestion occurs, so it would make sense to try and do smarter things at the routers with regard to congestion control. However, the trade-off here is making sure that what's running in the routers isn't overly complex or slow.

2. Purely end-to-end congestion control cannot enforce *isolation* between flows. If we want the rate of transmissions of flows to not affect each other, a purely end-to-end control strategy won't suffice in general. Depending on the economic model for network service, isolation may be a desired goal. We will discuss this issue in the context of scheduling and fair allocation schemes in the next lecture.

3. More generally, if various rate, delay, or delay jitter guarantees are desired, then router support is essential. We will discuss these issues in the context of quality-of-service (QoS).

4. Whereas the viability of end-to-end congestion control relies on cooperating sources, it isn't always a good assumption in today's Internet. Protecting the network against malicious traffic requires router support. We will address this issue when we discuss security later in the course.

What can routers do to overcome these limitations of end-to-end congestion control?

1. **Congestion signaling.** Routers can signal congestion when it occurs or is about to occur, to the sources in the network. This signaling can take on many forms: packet drops, explicit markings on packets, and explicit messages sent to sources. Some are more practical than others.

2. **Buffer management.** Routers have queues, primarily to absorb and accommodate bursts of packets. A router needs to decide *when* to signal congestion, and if it decides to, *which* packet to mark or drop to signal congestion. These two decisions are made by the buffer managememt algorithm at the router. A good buffer management scheme provides good incentives for end-to-end congestion control to be implemented.

3. **Scheduling.** When packets from multiple connections share a router, which packet should be sent next? Scheduling deals with this decision.

One might think that if routers were to implement sophisticate resource management algorithms, then end-to-end congestion control isn't needed. This perception, however, is not correct. To understand why, we need to understand what the dangers are of not doing any form of congestion control. As discussed in L3, the problem is the potential for *congestion collapse.*

In general, congestion collapse might take several forms and might occur for several reasons. The following is a taxonomy developed by Floyd.

1. *Classical collapse:* This is the congestion collapse caused by redundant retransmissions, such as when the TCP retransmission timeout ends up re-sending packets that aren't actually lost but onlu in queues en route.

2. *Fragmentation-based collapse:* Here, (large) datagrams are broken up into smaller fragments and sent. The loss of even a single fragment will cause the higher layer to transmit the entire datagram. (Think of running NFS over UDP across a congested network with a block size of 8 KB and an MTU of 1500 bytes! This is not hypothetical.)

3. *Control traffic-triggered collapse:* Some protocols use control packets separate from the packets used to send application payload. Examples include multicast group membership packets, routing updates, etc. If not designed carefully, control packets can start overwhelming a network.

4. *Undelivered packets:* This form of inefficiency, caused by unresponsive or aggressive traffic, is the hardest form to tackle. Some applications also incrase their bandwidth use upon detecting loss, sometimes by increased FEC (FEC in itself isn't bad, it's the increased bandwidth use that is).

Consider the network shown in Figure 7-1. Here, there are two sources S_1 and S_2, and every router implements flow isolation using fair queueing. Despite this, end-to-end congestion adaptation is essential, for otherwise, the 1.5 Kbps middle link ends up sharing bandwidth across S_1 and S_2, but most of S_2's packets end up getting dropped on the

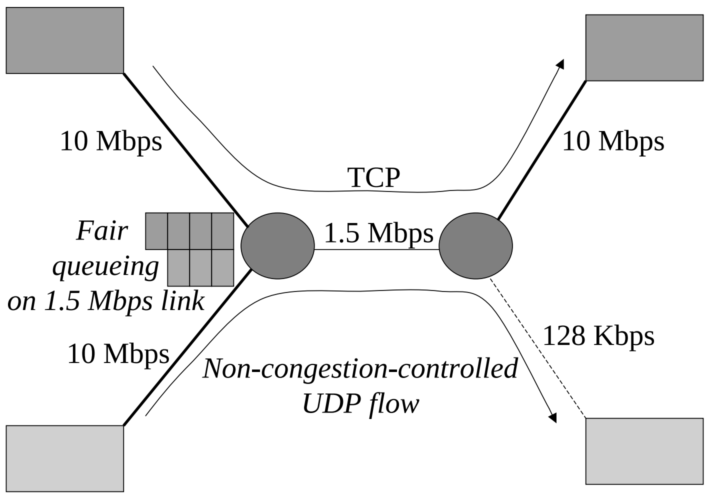

10 Mbps
TCP
10 Mbps
Fair queueing on 1.5 Mbps link
1.5 Mbps
10 Mbps
128 Kbps
Non-congestion-controlled UDP flow

Figure 7-1: End-to-end congestion control is essential even with ubiquitously deployed flow isolation mechanisms like fair queueing. This example is from Floyd and Fall, 1999.

downstream 128 Kbps link! The absence of end-to-end congestion control is wasteful indeed!

The rest of this lecture deals with congestion signaling strategies and buffer management. The next lecture concerns scheduling and fair capcity allocation strategies, specifically flow isolation using fair queueing.

■ 7.2 Congestion signaling

There are three ways in which routers can signal congestion to the sources. The first, and most robust for wired networks, is by dropping packets. On wired networks, packet losses occur mostly only because of congestion, and the right reaction is to reduce the speed of transmission (e.g., window reduction).

The second way is not to drop a packet when the need to signal congestion arises, but to *mark* it by setting some bits in the IP header. When the receiver receives this marked packet, it *echoes* the mark to the sender in its ACK. Upon receiving an ACK with a mark echoed on it, the sender does exactly the same thing it would've done on a packet loss-based congestion signal. An example of a one-bit marking scheme is ECN, Explicit Congestion Notification. A generalization of the approach is for the "bottleneck" switch to "mark" the packet with explicit rate information; when this information is relayed to the sender in ACKs, the sender can set its transmission rate according to the feedback it gets. XCP uses this idea.

The third approach is to send an *explicit message* to the sender of the packet when the router decides that the network is congested (or congestion is incipient). This message is sometimes called a *source quench* and was once believed to be a good idea. It isn't used much today; one of the big problems with it is that it forces more packets to be sent precisely when congestion is occurring in the network!

7.2.1 Packet drops

On wired networks, packet drops almost always signify network congestion. The receiver in a reliable transport protocol indicates the loss of packets to the sender, which can perform the appropriate congestion control tasks. For protocols that don't require reliability, some form of feedback is still important in order to perform congestion control.

Not all schemes drop packets only when the router's queue is full. One can gain a great deal by making the drops *early*, so that the *average* queue size is still kept relatively small, even though the total queue size available is substantially larger, to accommodate bursts of packets. We will study this in Section 7.3.

7.2.2 Marking packets

Various proposals have been made for doing this. Historically, the first proposal was the DECBIT scheme by Ramakrishnan and Jain, which combined an AIMD source with router support, where the routers would mark a bit in the packet header when the queue size (as estimated over a certain time interval, not the instantaneous queue size) exceeded some threshold.

The current standard way of signaling congestion using packet marking is called *Explicit Congestion Notification* (ECN). First worked out for TCP/IP by Floyd in 1994, it has recently been standardized. The implementation of this approach is quite straightforward:

1. The sender uses a special TCP option to tell the receiver it is capable of ECN in the connection setup (SYN) packet.

2. If the receiver is capable of ECN, it responds in its SYN ACK sent to the initiating peer.

3. All subsequent packets on the connection have a bit set in the IP header that tell the routers that this packet belongs to a connection that understands, and will react to, ECN.

4. Each router may use its own policy to implement the ECN mechanism. For example, one might use RED, and mark the packet (by setting another bit in the IP header) when the average queue size exceeds some threshold, rather than dropping it. Of course, this marking is done *only if* the packet is found to be ECN-capable (i.e., the packet belongs to an ECN-capable connection or flow); otherwise, it is simply dropped. Marking the bit only for ECN-capable traffic is important because it allows both ECN and non-ECN flows to co-exist and compete fairly with each other.

5. Upon receiving any packet with ECN set on it, the receiver *echoes* this information in its ACK (or equivalent feedback) message to the sender.

6. When the sender receives an ACK or feedback message with ECN echoed, it takes appropriate congestion control measures; e.g., by reducing its window. It also sets some information in the IP header that tells the receiver that the sender has in fact reacted to this echo.

7. To guard against possible ACK loss, the receiver sets ECN on *all* ACK messages until it receives a packet from the sender that indicates that in fact the sender has paid attention to this feedback. For reliable transport like TCP, this redundancy is sufficient because the receiver will eventually receive the packet from the sender that tells it that the sender has taken steps to deal with the information it received about congestion on the path.

There are several things to notice about the ECN design. First, an important goal is backward compatibility with existing TCPs, both in terms of the header formats, and in terms of how ECN and non-ECN flows compete for link capacity. Second, the ECN mechanism needs some attention to implementation details because bits in the IP header aren't freely available. Third, the ECN mechanism is susceptible to abuse by receivers, because the receiver can exchange information with the sender assuring it that it will echo ECN information, causing the sender to set the bits in the packet that tell the routers that the packet belongs to an ECN-capable connection. Then, when the receiver gets a packet with ECN set on it, it can silently *ignore* this and not echo it back to the sender! The end-result is that this malicious receiver can obtain a largely disproportionate amount of the bottleneck link's available bandwidth.

The solution proposed by Spring *et al.* to this problem is elegant and simple: use *nonces* on packets such that the receiver can *prove* to the sender that it did in fact receive a packet without the "congestion-experienced" bits when that packet arrived with this information set. In principle, this can be done by the source setting a random number in the packet header. When a router sees an ECN-capable connection and is about to set the "congestion-experienced" bits, it also *clears the nonce* to zero. The receiver, now, cannot lie that it received a packet without notification about congestion, since then it would have to echo the correct nonce!

The paper asserts that *one-bit nonces* suffice for this. It isn't hard to see why, because the probability of guessing a 1-bit random nonce correctly is 0.5. So, to be successful at lying k times falls off as $\frac{1}{2^k}$, which is rapid. Furthermore, upon discovering a lying receiver, the sender (if it wishes) can treat it punitively, by giving it much less than its fair share.

The final trick to completing the scheme is to make sure it works when not every packet is acknowledged. This is handled by making the nonces *cumulative* (i.e., the sum of all nonces sent so far, in $(0, 1)$ arithmetic). Furthermore, when the receiver cannot reconstruct the nonce (because it was cleared by a router on congestion), the sender needs to resynchronize its notion of the cumulative value with the receivers, and continue from there.

■ 7.3 Buffer management, active queue management, and RED

Buffer management schemes at routers decide *when to drop a packet* and *which packet to drop*. The simplest scheme is *drop-tail*, where packets are accommodated until the queue is full, after which all incoming packets are dropped until queue space is again available.

Drop-tail has several drawbacks: it doesn't signal congestion early enough, it causes packets to be dropped in bursts, and it doesn't keep average queue sizes small. This last drawback means that flows experience higher delays than might be reasonable.

It's worth noting that most of the Internet today still runs drop-tail gateways. The performance of drop-tail for TCP traffic under high degrees of statistical multiplexing isn't particularly well-understood.

7.4 RED: Random Early Detection

RED is an *active queue management* scheme, which explicitly tries to monitor and contain the average queue size to be small—small enough for delays to be small, but large enough for bottleneck link utilization to be high when bursty traffic causes a temporary lull in offered load (e.g., when a set of TCP sources cut down their sending speeds).

The key idea in RED is to use randomization to avoid synchronizations and biases that could occur in deterministic approaches, and to drop or mark packets earlier than when the queue is full.

The approach is as follows: If the average queue size, q_a is between min_{th} and max_{th} the packet is *marked* or *dropped* packet with some probability, p_a. If $q_a > max_{th}$, then the packet is (always) dropped. If $q_a < min_{th}$, then the packet is forwarded through.

Two key issues in RED are: (1) how to compute q_a and (2) how to compute p_a.

7.4.1 Computing q_a

q_a is computed using a standard EWMA filter:

$$q_a = (1 - w_q)q_a + w_q q$$

Here w_q is time constant of low pass filter. If w_q is too large, it implies a rapid response to *transient* congestion. If it is too low, it implies sluggish behavior. The paper shows a back-of-the-envelop for setting this: obtain an upper bound based on how much burstiness you want to allow, and obtain a lower bound based on how much memory you want to maintain. It isn't clear to me how well this works in practice.

The paper uses $w_q = 0.002$. Also, the recommendation is for $max_{th} \geq 2min_{th}$. The best place to find out how these parameters should be set is http://www.aciri.org/floyd/red.html

7.4.2 Computing p_a

Initially, compute $p_b = max_p \times \frac{q_a - min_{th}}{max_{th} - min_{th}}$. This is standard linear interpolation.

There are two ways in which p_a, the actual drop probability, can be derived from p_b.

1. Method #1: $p_a = p_b$. This leads to a geometric random variable (r.v.) for inter-arrivals between markings (or drops). Consider the r.v. X = number of packets between successive marked packets. Then $P(X = n) = (1 - p_b)^{n-1} p_b$. This is a geometric distribution.

   Unfortunately, this doesn't mark packets at fairly regular intervals. $E[X] = 1/p_b$, but because packets get marked in bunches, more synchronization than is good could occur. $Var[X] = 1/p_b^2 - 1/p_b$, which gets big for small p_b.

2. Method #2: Uniform r.v. for inter-arrivals between markings or drops. Let X be uniform in $\{1, 2, \ldots, 1/p_b - 1\}$. This has the nice property that we want, and is achieved

if we set $p_a = p_b/(1 - count \times p_b)$, where $count$ is the number of unmarked packets since last marked packet. $E[X] = 1/(2p_b)$.

The final point is about the choice of max_p. One argument is that it should never have to be large, since after $q_a > max_{th}$, all packets are dropped anyway. However, a small value of max_p (like 0.02) makes the behavior of p_b rather abrupt and discontinuous when $q_a \approx max_{th}$. A "gentle" variant of RED has been proposed that makes it less discontinuous, or indeed continuous.

Several variants of RED and alternatives to it have been proposed in the last few years. A search on Netbib (see the "Useful Links" page from the class Web page) or the Web will give you some pointers. Some examples include Gentle RED (where the transition from probabilistic to always drop is not so sudden), ARED, SRED, AVQ, BLUE, CHOKE, etc.

CHAPTER 8
Scheduling for Fairness: Fair Queueing and CSFQ

■ 8.1 Introduction

In the last lecture, we discussed two ways in which routers can help in congestion control: by signaling congestion, using either packet drops or marks (*e.g.,* ECN), and by cleverly managing its buffers (*e.g.,* using an active queue management scheme like RED). Today, we turn to the third way in which routers can assist in congestion control: *scheduling*.

Specifically, we will first study a particular form of scheduling, called *fair queueing* (FQ) that achieves (weighted) fair allocation of bandwidth between flows (or between whatever traffic aggregates one chooses to distinguish between.[1] While FQ achieves weighted-fair bandwidth allocations, it requires *per-flow* queueing and per-flow state in each router in the network. This is often prohibitively expensive. We will also study another scheme, called *core-stateless fair queueing* (CSFQ), which displays an interesting design where "edge" routers maintain per-flow state whereas "core" routers do not. They use their own non-per-flow state complimented with some information reaching them in packet headers (placed by the edge routers) to implement fair allocations. Strictly speaking, CSFQ is not a scheduling mechanism, but a way to achieve fair bandwidth allocation using differential packet dropping. But because it attempts to emulate fair queueing, it's best to discuss the scheme in conjuction with fair queueing.

The notes for this lecture are based on two papers:

1. A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm", *Internetworking: Research and Experience*, Vol. 1, No. 1, pp. 3–26, 1990. (Or read ACM SIGCOMM 1989 version.)

2. I. Stoica, S. Shenker, H. Zhang, "*Core*-Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Network," in *Proc. ACM Sigcomm*, September 1998, Vancouver, Canada.

[1]FQ is also known as *Packet Generalized Processor Sharing* (PGPS), a scheme developed at MIT at about the same time as FQ was developed at PARC.

■ 8.2 Scheduling

The fundmental problem solved by a scheduling algorithm running at a router is to answer the following question: "Which packet gets sent out next, and when?" The choice of which flow (or flow aggregate; we use the term "flow" to mean one or the other in this lecture, unless mentioned otherwise) is the next one to use an outbound link helps apportion bandwidth between flows. At the same time, a scheduling algorithm can also decide when to send any given packet (or flow's packet), a choice that can help guarantee (or bound) packet latencies through the router.

An important practical consideration, perhaps the most important consideration in the design of router scheduling algorithms concers *state management.* The key concern here is that schemes that maintain per-flow state are not scalable, at least in several router designs. This is in stark contrast to buffer management schemes like RED, where the state maintained by the router is independent of the number of flows. However, RED does not achieve fair bandwidth allocations amongst flows, whereas good scheduling algorithms can (precisely because they tend to directly control which flow's packet gets to use the link next).

It's also worth appreciating that scheduling for weighted-fair bandwidth allocation and scheduling to provide latency guarantees are actually orthogonal to each other and separable. For instance, one can imagine a hypothetical router that starves a flow for 6 days each week and gives extremely high bandwidth for one whole day, so the flow receives some pre-determined share of bandwidth, but does not receive any latency guarantees. Scheduling schemes have been developed that provide both bandwidth and latency guarantees to flows, although they tend to be rather complex and require care in how these requirements are specified.

Broadly speaking, all schedulers can be divided into two categories: *work-conserving* and *non-work-conserving*. The former, of which fair queueing is an example, *never* keep an outbound link idle if there is even one packet in the system waiting to use that link. In contrast, the latter might, for instance to provide delay bounds. Another example of the latter kind is a strict time-division multiplexing system like the one we saw in Problem 1 of Problem Set 1.

■ 8.2.1 Digression: What happens to a packet in a router?

To understand how scheduling fits into the grand scheme of things that happen in a router, it's worth understanding (for now, at a high level) all the different things that go on in a router. For concreteness, we look at an IP router. The following steps occur on the *data path*, which is the term given to the processing that occurs for each packet in the router. We only consider unicast packets for now (i.e., no multicast).

1. Validation. When a packet arrives, it is validated to check that the IP version number is correct and that the header checksum is correct.

2. Destination lookup. A longest-prefix-match (LPM) lookup is done on the forwarding table using the destination IP address as lookup key, to determine which output port to forward the packet on. Optionally, an IP source check may also be done to see if the port in which this packet is a valid, expected one. This is often used to help

protect against certain denial-of-service (DoS) attacks.

3. Classification. Many routers *classify* a packet using information in the IP header fields (*e.g.*, the "type-of-service" or TOS field, now also called the "differentiated services code point" or DSCP) and information in higher layers (*e.g.*, transport protocol type, TCP/UDP port number, etc.). This is also called "layer-4 classification." The result of a classification is typically a queue that the packet is enqueued on. In theory, one can classify packets so each connection (or flow) gets its own queue, specified by a unique combination of source and destination addresses, source and destination transport-layer ports, and TOS field, but this turns out to be prohibitively expensive because of the large number of dynamically varying queues the router needs to keep track of. In practice, routers are usually set up with a fixed number of queues, and a network operator can set up classification rules that assign packets to queues. A classified queue can be fine-grained, comprising packets from only a small number of transport-layer flows, or it could be coarse-grained comprising packets from many flows (*e.g.*, a large traffic aggregate of all packets from 18.0.0.0/8 to 128.32.0.0/16).

   We'll study classification techniques and router architectures in more detail in later lectures.

4. Buffer management. Once the router has decided which queue, it needs to append the packet to it. At this time, it should decide whether to accept this packet or drop it (or mark ECN), or in general to drop some other packet to accept this one. It might run a simple strategy like drop-tail, or it might run something more sophisticated like RED or one of its many variants.

5. Scheduling. Asynchronous to specific packet arrivals, and triggered only by queue occupancy, the router's link scheduler (conceptually one per link) decides which of several queues with packets for the link to allow next on the link. The simplest scheduling strategy is FIFO, first-in, first-out.

8.2.2 Back to fairness

In this lecture we will consider bandwidth scheduling algorithms that achieve *max-min fairness*. With max-min fairness, no other allocation of bandwidth has a larger minimum, and this property recursively holds. Recall from an earlier lecture what max-min fairness provides:

1. No flow receives more than its request, r_i.

2. No other allocations satisfying (1) has a higher minimum allocation.

3. Condition (2) recursively holds as we remove the minimal user and reduce the total resource accordingly.

This is by no means the only definition of fairness, but it's a useful one to shoot for. One consequence of max-min fairness is that all allocations converge to some value α such that all offered loads $r_i < \alpha$ are given a rate r_i, while all inputs $r_i > \alpha$ are given a rate equal to

α. Of course, the total rate available C (the output link's rate) is doled out amongst the n input flows such that

$$\sum_{i=1}^{n} \min(r_i, \alpha) = C. \tag{8.1}$$

Fair queueing (FQ) is a scheme that achieves allocations according to Equation 8.1. CSFQ is a distributed approximation to FQ that generally comes close, without the associated per-flow complexity in all routers.

■ 8.3 Fair queueing

Fair queueing is conceptually simple to understand: it is like round-robin scheduling amongst queues, except it takes special care to handle variable packet sizes. The primary practical concern about it is its requirement of per-flow state. The good thing, though, is that weighted variants of fair queueing add little additional complexity.

We emphasize that we're going to use the term "flow" without a precise definition of it, on purpose. Think of a "flow" as simply the result of the classification step from Section 8.2.1, where there are a set of queues all vying for the output link. We wish to apportion the bandwidth of this link in a weighted fashion amongst these queues ("flows").

Pure versions of fair queueing have proved to be hard to deploy in practice because the number of flows (and therefore queues) a router would have to have is in the tens of thousands and *a priori* unknown. However, bandwidth apportioning schemes among a fixed set of queues in a router are gaining in popularity.

■ 8.3.1 Bit-by-bit fair queueing

The problem with simple round-robin amongst all *backlogged* queues (i.e., queues with packets in them) is that variable packet sizes cause bandwidth shares to be uneven. Flows with small packets get penalized.

To understand how to fix this problem, let's consider an idealized bit-level model of the queues vying for the link. In this world, if we could forward one bit per flow at a time through the link, we would implement a round-robin scheme and bandwidth would be apportioned fairly. Unfortunately, in real life, packets are not *preemptible*, i.e., once you start sending a packet, you need to send the whole thing out; you cannot temporarily suspend in the middle of a packet and start sending a different one on the same link (this would make demultiplexing at the other end a pain, and would in fact require additional per-bit information!).

Sticking with the idealized bit-based model, let's define a few terms.

Definition: **Round**. One complete cycle through all the queues of sending one bit per flow.

Note that the time duration of a round depends on the number of backlogged queues.

Let $R(t)$ be the round number at time t. If the router can send μ bits per second, and there are N active flows, the number round number increases at rate

$$\frac{dR}{dt} = \frac{\mu}{N}. \tag{8.2}$$

The greater the number of active flows, the slower the rate of increase of the round number. However, the key point is that the *number of rounds* for a complete packet to finish being transmitted over the link is *independent* of the number of backlogged queues. This insight allows us to develop a scheduling strategy for fair queueing.

Suppose a packet arrives and is classified into queue α, and it is the i-th packet on the queue for flow α. Let the length of the packet be p_i^α bits.

In what round number S_i^α does the packet reach the head of the queue? ("S" for "start.") There are two cases to consider while answering this question.

1. If the queue is empty, then the packet reaches the head of the queue in the current round $R(t)$.

2. If the queue is already backlogged, then the packet reaches the head of the queue in the round right after the packet in front of it finishes, F_{i-1}^α. ("F" for "finish.") [2]

Combining the two cases, we have in general

$$S_i^\alpha = \max(R(t), F_{i-1}^\alpha). \tag{8.3}$$

Now, in what round does this packet finish? This depends *only* on the length of the packet, since in each round exactly one of its bits is sent out. Therefore

$$F_i^\alpha = S_i^\alpha + p_i^\alpha, \tag{8.4}$$

where p_i^α is the size of the i^{th} packet of flow α.

Using Equations 8.3 and 8.4, we can determine the finish round of every packet in every queue. We can do this as soon as the packet is enqueued, but it requires much less state to do it when each packet reaches the head of a queue.

We emphasize, again, that these formulae are independent of the number of backlogged queues. Using round numbers "normalizes" out the number of flows and is an important point.

This *bit-by-bit round robin* scheme (also called *generalized processor sharing* (GPS) is what the packet-level fair queueing scheme will emulate, as we describe next.

■ 8.3.2 Packet-level fair queueing

The packet-level emulation of bit-by-bit round robin is conceptually simple:

Send the packet which has the smallest finishing round number.

This scheme approximates bit-by-bit fair queueing by "catching up" to the what it *would have done* on a discrete packet-by-packet basis. The packet sent out next is the packet which had been starved the most while sending out the previous packet (from some other queue in general). The algorithm is illustrated in Figure 8-1.

This emulation is not the only possible one, but it is the most intuitive one. Some other ones don't work; for instance, what if we had sent out packets by the largest finishing round number? This won't work because a flow sending very large packets would always be chosen. A different emulation might choose the smallest starting round number.

[2]We are dropping a few tiny $+1$ and -1 terms here.

Thinking about whether this approach can be made to work is left as an exercise to the reader.

8.3.3 Some implementation details

Some implementation details are worth nailing down.

Buffer management

Each queue in the fair queueing system needs to manage its buffers. First off, preallocating a fixed amount of memory for each queue and drop whenever a queue becomes full is a bad idea, since we don't know how traffic is going to be distributed across queues.

A better idea is to have a global buffer pool, and the queues grow as necessary into the pool.

A common strategy for dropping packets in fair queueing is to drop from the longest queue or from the queue with largest finish number (they aren't always the same and they have different fairness properties). Presumably one can run RED on these queues if one likes, too, or a drop-from-front strategy.

State maintenance

Do we need to maintain the finishing round F_i^α for every packet? In other words, do we need to maintain *per-packet state*?.

No, since the finishing round can be calculated on the fly. The router maintains the finish round F_i^α for the packets on the front of the queue, and when a packet gets sent out, it uses the length of the packet after it to calculate the F_{i+1}^α of the new head of the queue.

The trick is figuring out what finish round number we should give to a packet that arrives into an empty queue. One approach that works well is to give it a start round number equal to the finish number (or start round number) of the packet currently in service.

Difficulties

In practice, fair queueing is challenging because classification may be slow; the more problematic issue is the need to maintain metadata pointers to memory not on the central chip of the router if one wants to implement "pure" flow-level fair queueing. A "core" router may have to maintain 50,000–100,000 flows, and have to process each packet in under 500ns (for 1 Gbps links) and 50ns (for 10 Gpbs links). It seems practical today to build routers with a large number of queues, provided there's some fixed number of them. A large, variable number of queues is particularly challenging and difficult to implement in high-speed routers.

8.3.4 Deficit Round Robin

Many different ways to implement fair queuing have been developed over the past several years, stemming from the implementation complexity of the scheme described above. One popular method is called *Deficit Round Robin* (DRR). The idea is simple: each queue has a *deficit counter* (aka *credit*) associated with it. The counter increments at a rate equal to the

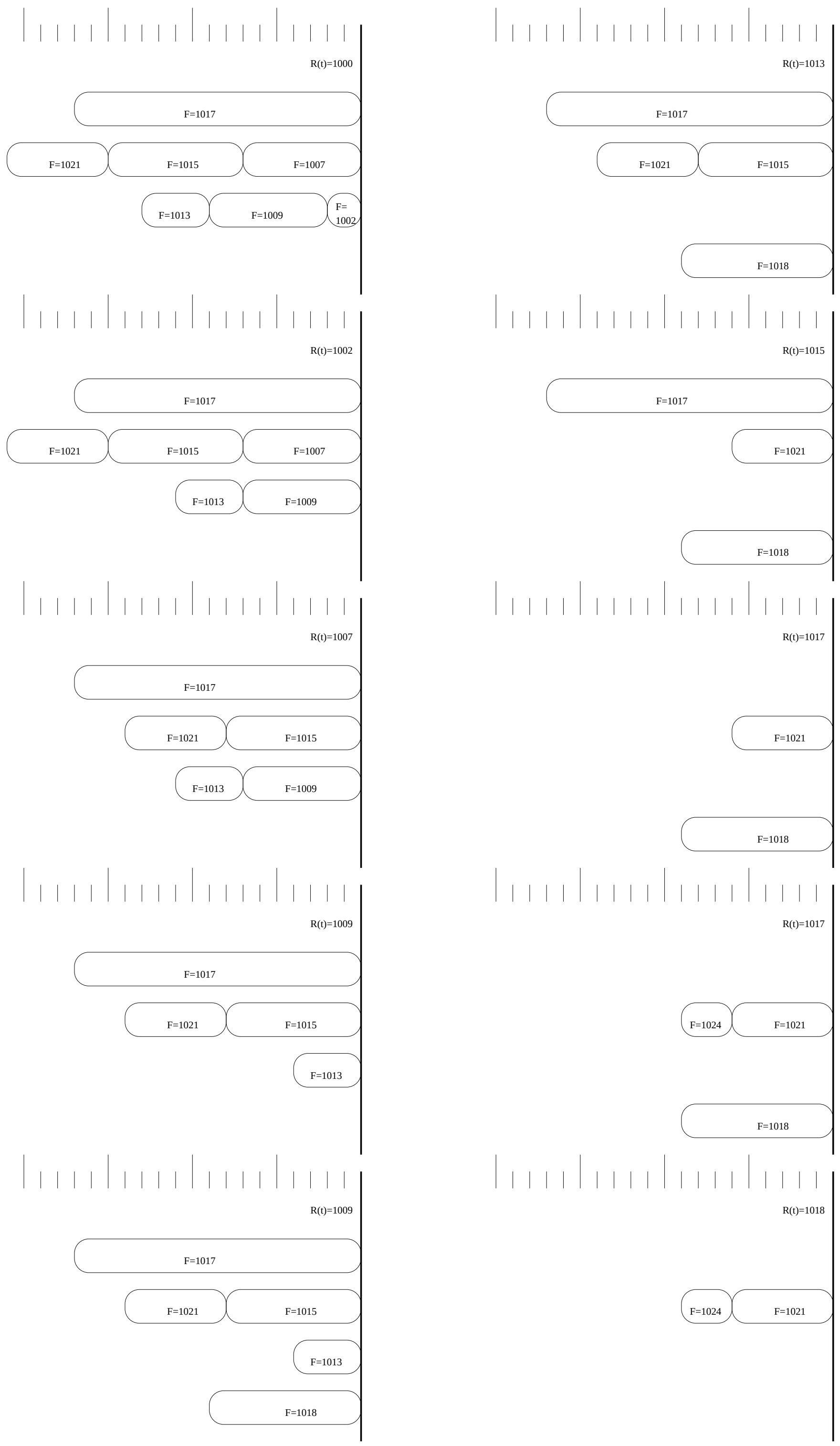


Figure 8-1: Fair queueing in action. (Thanks to abhi shelat for figure.)

fair rate of that queue (e.g., if the fair rate is X bits/s, it increments at that rate). Whenever the counter's value exceeds the size of the packet at the head of the queue, that packet may be sent out on the link by the scheduler. When a packet is sent out, the queue's counter decrements by the size of the packet.

This scheme, as described, is *not work-conserving*, because no packet may be sent out even when some queue has packets in it. That's because every queue may have a counter smaller than the size of the packet at its head. If the designer wants to make this scheme work-conserving, then she might modify the scheme to allow the counter to decrement in value to a small negative value (say, the negative of the maximum packet size). The DRR scheme outlined above also allows arbitrary "overdraft"—an empty queue may continually increment its counter even when there's no data in the queue, causing a large burst of access when packets eventually arrive. This approach, in general, may also cause long delays to some queues. A simple solution is to limit the extent to which the counter can grow when the corresponding queue is empty.

■ 8.4 CSFQ: Core-Stateless Fair Queueing

■ 8.4.1 Overview

In the first half of this lecture, we discussed a fair queuing scheme that divides the available bandwidth according to max-min fairness. The drawback of the FQ approach was that costly per-flow statistics needed to be maintained. In a typical backbone router, there may be 50,000-100,000 flows at a given time. At gigabit speeds, there may not be enough time or memory to manage per-flow meta-data.

We now turn to an approximation to FQ that refines it in a way that circumvents the problem of maintaining per-flow statistics. The *Core*-Stateless FQ scheme, CSFQ, distinguishes core routers, the higher-speed and busier routers at the "core" of an Internet AS backbone from edge routers. In typical deployment, edge routers might handle thousands of flows, while core routers might handle 50k-100k flows. CSFQ exploits this gap by delegating the management of per-flow statistics to the edge routers. Edge routers then share this information with core routers by labeling each packet that they forward. Core routers, in turn, can use the labels to allocate bandwidth fairly among all incoming flows. It is important to realize that in the case of CSFQ, edge routers run essentially the same algorithm as core routers (including probabilistically dropping incoming packets); however, edge routers have the added responsibility of maintaining per-flow state. In general, of course, edge and core routers in such an approach could run very different algorithms.

Here are the key points of CSFQ:

1. Dynamic Packet State: Edge routers label each packet with an estimate of the arrival rate for each flow. Per-flow statistics are maintained here.

2. Core routers use (1) estimated arrival rates provided on packet labels, and (2) an internal measure of fair-share, to compute the probability of dropping each incoming packet. Every packet that is accepted is processed and relabeled with new arrival rate information.

3. The estimation procedure for the "fair-share" value convergences rapidly to the op-

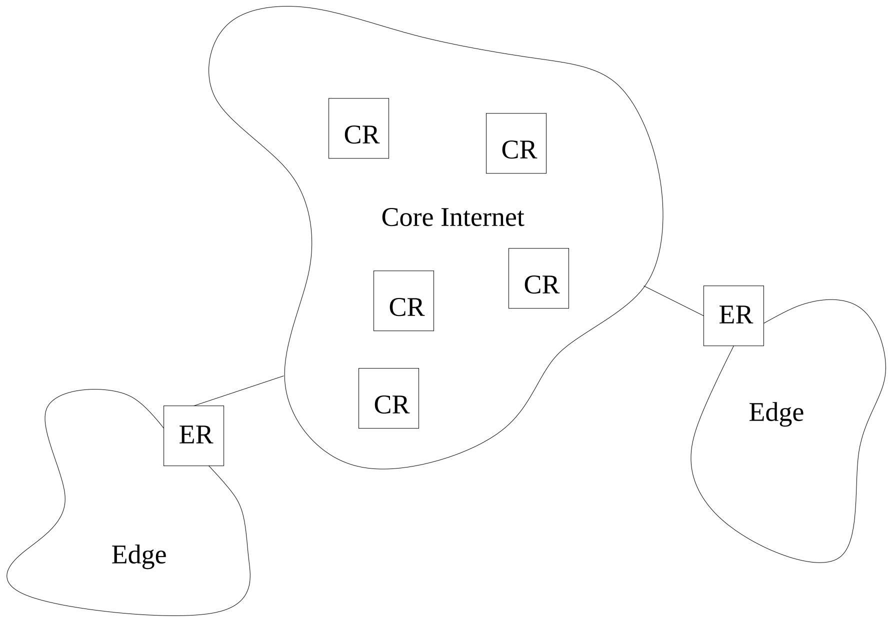


Figure 8-2: The core and the edges of an Internet backbone network. CR = core router; ER = edge router.

timal value. Cheaters cannot win too much extra bandwidth.

There are two goals for a CSFQ router:

1. Maintain max-min fairness for bandwidth allocation.

2. Avoid having to keep per-flow statistics in high-speed core routers.

Note that goal (2) prevents a core router from maintaining per-flow queues. Therefore, once a packet has been accepted by a core router, it sits in one of a small number of queues until it is eventually processed. Hence, the only action the core router can take in order to achieve (1) is to drop packets from greedy flows. Notably absent is the ability to schedule when a packet is to be sent. In order to avoid patterns of synchronization, packet dropping is done probabilistically using both information accumulated and added to the packet by an edge router, and a global parameter estimated by the core router.

8.4.2 Dynamic packet state

In order to achieve max-min fairness, a router needs to know the arrival rate, $r_i(t)$ for each flow. Under CSFQ, edge routers are assigned the task of estimating these rates. This estimation is computed by taking an exponentially weighted average of samples. In this paper, the weighting is slightly more complicated than previous exponentially weighted schemes that we have discussed. That is: if t_i^k, l_i^k are the arrival time and length of the kth packet in flow i, then $r_i(t)$ is computed according to the equation

$$r_i^{new} = (1 - e^{-T_i^k/K})\frac{l_i^k}{T_i^k} + e^{-T_i^k/K} r_i^{old}, \tag{8.5}$$

where $T_i^k = t_i^k - t_i^{k-1}$ and K is some constant. The T_i^k is the i^{th} interarrival time, $t_i^k - t_i^{k-1}$. Instead of using a fixed constant as their weighting factor, the authors argue that the $e^{-T_i^k/K}$ weight converges faster under a broader set of situations and isn't as sensitive to the specific packet-length distribution. As each incoming packet arrives, the new r_i is computed for this flow, a decision about whether to drop the packet is made, and if the router decides to forward the packet, the packet is *labeled* with r_i and forwarded. This idea of setting some information in the header of the packet is called *dynamic packet state* and is an architectural concept that generalizes beyond CSFQ.

8.4.3 Fair-share rate

Given a set of flows with arrival rates, $r_1(t), r_2(t), \ldots, r_n(t)$, each edge and core router must determine the fair-share bandwidth according to max-min by assigning each flow the minimum of their requested bandwidth and the fair-share bandwidth left: i.e.it needs to compute the fair-share rate according to Equation 8.1. Suppose there's some way of calculating α (this is easy for an edge to do, but we'll see in a bit how a core router might estimate it).

Given α, then enforcing a fair rate to incoming traffic isn't too hard. This can be done using random packet drops, according to the following equation.

$$P(\text{drop packet from stream } i) = \max(1 - \frac{\alpha}{r_i(t)}, 0) \tag{8.6}$$

If we have three flows that transmit at rates $r_1 = 8, r_2 = 6,$ and $r_3 = 2$ Mbps and these flows are sharing a 10 Mbps link, then $\alpha = 4$ Mbps. Then, if packets were dropped with probabilities $0.5, 0.33,$ and 0 respectively, a max-min-fair allocation can be achieved.

Hence, a router needs only two pieces of information in order achieve max-min fairness: α, which is one number per outgoing link at a router, and $r_i(t)$, the arrival rate for each flow. Fortunately, the edge routers calculate $r_i(t)$ using Equation 8.5 and provide it to the core routers using dynamic packet state, so what remains is the estimation of α. In addition to the architectural concept of dynamic packet state, a noteworthy intellectual contribution of this paper is how α may be estimated at the core routers.

The α fair-share parameter determines the rate at which the router accepts (and therefore processes) packets. So let us define $F(\alpha)$ as the core router's acceptance rate in bits/s. If max-min fairness according to Equation 8.1 is made to hold, then

$$F(\alpha) = \sum_i \min(r_i(t), \alpha) \tag{8.7}$$

That is, the aggregate acceptance rate for the router is the sum of the rates for each of the router's flows. If a flow is below its fair-share, then it will contribute $r_i(t)$, and otherwise, it will be capped at its fair-share limit.

The big picture of our strategy is as follows. Each router estimates $F(\alpha)$, the total (aggregate) rate at which packets are accepted into (forwarded by) the router. At the same time, each core router estimates the aggregate input rate (A) into each of its egress links; this is independent of the number of flows and can be done using an equation similar to Equation 8.5. Together, these two pieces of information, together with knowledge of whether the link (of capacity C) is "congested" or not, allow the router to *estimate* $\tilde{\alpha}$, an estimate for α. This estimation relies on the behavior of the function $F(\tilde{\alpha})$ as a function of $\tilde{\alpha}$.

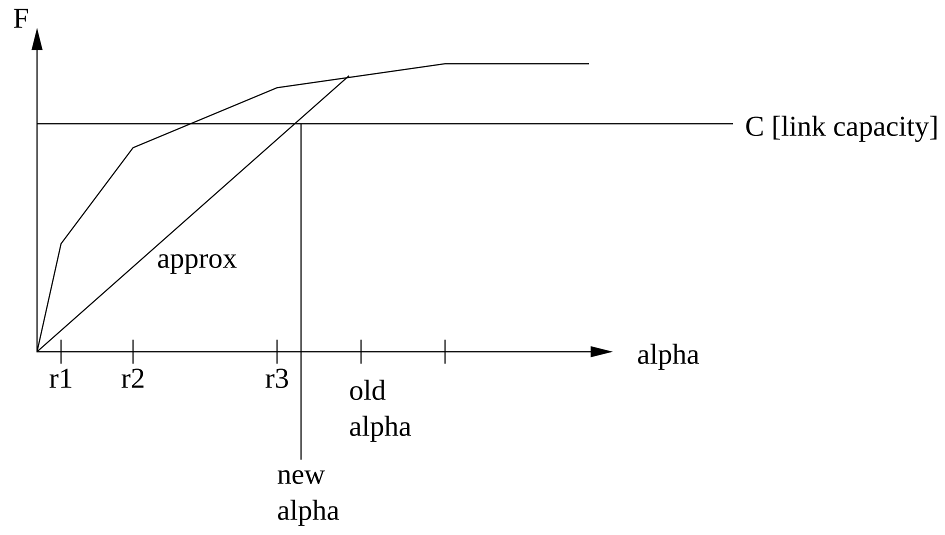


Figure 8-3: $F(\tilde{\alpha})$ **vs** $\tilde{\alpha}$**.**

So consider the graph shown in Figure 8-3 of $F(\cdot)$ as a function of $\tilde{\alpha}$, and assume that the flow arrival rates are in non-decreasing order such that $r_1 \leq r_2 \leq \ldots \leq r_n$.

When $\tilde{\alpha}$ is less than r_1, the aggregate acceptance rate is $n \times \tilde{\alpha}$ and the slope of the curve is n. However, as soon as $\alpha > r_1$, then the first stream wil be capped at r_1, and the aggregate max-min acceptance rate will now be $(n-1)\tilde{\alpha} + r_1$. That is, the function now has slope $n-1$, so it is flatter. Continuing thus, it is easy to see that $F(\tilde{\alpha})$ is continuous, piecewise linear, with n pieces, with the slope of each piecewise linear portion (as $\tilde{\alpha}$ increases) smaller than the previous one (ending in a line parallel to the horizontal axis when $\tilde{\alpha}$ exceeds the largest input rate, r_n). It is therefore a *concave* function.

Hence, in order to determine $\tilde{\alpha}$ so as to use exactly the capacity in the outbound link, the core router needs to solve the equation $F(\tilde{\alpha}) = C$. Whereas an edge router can store n pieces of information that encode the curve $F(\cdot)$ (this information is simply the arriving per-flow rates), a core router cannot. Instead, the core router simply approximates $F(\cdot)$ with a straight line. If the current acceptance rate (forwarding rate) is $\tilde{F}$ and we already have an estimate $\tilde{\alpha}$ for α, then we can approximate

$$F(\alpha) \approx \frac{\tilde{F}}{\tilde{\alpha}}\alpha. \tag{8.8}$$

During the processing of packets, the core router simply:

1. Observes the system over a given time period.

2. If the system is "uncongested" then the acceptance parameter, α is increased to $\max_i(r_i)$. With this value, the probability of dropping a packet is 0 for all flows.

3. If the system is "congested" then α is too large, and we need to re-estimate it. The new α is calculated by setting to the solution to the equation $F(\alpha) = C$, where $F(\alpha)$ is

approximated using Equation 8.8.

Because $F()$ is concave, the core router will eventually converge on the correct α value. The only remaining issue is how a router decides if one of its links is "congested" or not. A router performs this assessment by looking at the estimate of the aggregate arrival rate A and seeing if it is smaller than C for a period of K ms ("uncongested") or larger ("congested"). If neither is true, then α is left unchanged.

In summary, CSFQ is a hybrid scheme that asymptotically converges to max-min fairness. It incorporates two neat ideas: a decomposition of edge and core routers with dynamic packet state transferring information to the core routers, and a simple technique to estimate the fair rate in a core router. The technique generalizes to weighted fair queueing too, although the case of different weights in each router in a CSFQ island is ill-defined and not supported. The paper reports performance results under various conditions, although most of the evaluated conditions don't dramatically vary the number of flows or cause bandwidth oscillations.

Of course, CSFQ suffers from some drawbacks too. All routers in an island need to be simultaneously upgraded; dynamic packet state needs to be implemented somehow, and the proposal is to use the IP fragmentation header fields; and the consequences of buggy or malicious routers needs to be explored.

CHAPTER 9
Principles Underlying Internet QoS

This lecture discusses the main principles underlying Internet QoS. We don't focus much on specific schemes (IntServ and DiffServ), leaving that for the sections at the end.

The material for these notes is drawn partially from:

1. S. Shenker [She95], Fundamental Design Issues for the Future Internet , IEEE Journal on Selected Areas in Communications, Vol. 13, No. 7, September 1995, pp. 1176-1188.

2. D. Clark, S. Shenker , and L. Zhang, Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanisms, in Proc. SIGCOMM '92, Baltimore, MD, August 1992.

■ 9.1 Motivation: Why QoS?

In the previous three lectures, we studied how the Internet manages two network resources: link bandwidth and queue buffer space. A key feature of how the Internet manages these resources is that it relies heavily on end-system cooperation. The lecture on fair queueing showed how network switches could isolate flows from each other using separate queues and non-FIFO link scheduling.

Armed with link scheduling and queue management methods, we can think about suitable *architectures* for improving on the Internet's best-effort service model. That's what QoS is all about, and research on this topic includes both architecture and specific scheduling algorithms. This lecture will focus on the key architectural approaches, rather than on algorithms (variants of fair queueing suffice for our discussion).

QoS research was motivated by the perceived need for the network to provide more than the best-effort service. The arguments for why the network should provide something better than best-effort are usually made along the following lines:

1. The best-effort model serves all applications, but is not optimal for many applications. In particular, applications that require a minimum rate (e.g., some forms of video), or delay bounds (e.g., telephony), or bounded variation in delays (e.g., live

conferencing, perhaps streaming media) could all benefit from being isolated from other less-demanding applications.

2. Revenue opportunities of ISPs: If ISPs were able to treat certain customers or certain application flows better than others, they might make more money. This argument has become a critical determinant of whether QoS is deployed in any network or not: only schemes that provide clear economic incentives and revenue opportunities are likely to be deployed by network providers.

3. Traffic overload protection: Denial-of-service attacks, flash crowds, and certain bandwidth-hungry applications (e.g., P2P file sharing applications today) all consume large amounts of bandwidth and prevent other traffic from receiving their "share". ISPs are interested in isolating classes of traffic from each other, usually by setting rate controls on traffic classes and/or providing guaranteed bandwidth to certain other classes. QoS mechanisms turn out to help in meeting traffic management and overload protection goals.

9.1.1 What kind of QoS might we want?

From the standpoint of the application provider, the following goals might make sense:

1. Guaranteed bandwidth for certain application flows.

2. Guaranteed delay bounds for certain application flows.

3. Minimizing jitter (the variation in delay).

4. Low or close-to-zero packet loss.

From the standpoint of the network provider, the following goals might make sense:

1. Traffic classes for different customer flows, such as "Gold", "Silver", "Bronze", etc., all priced differently. For example, a provider might sell a customer content provider company "Gold" service at a certain rate (and perhaps delay) for all its port 80 traffic as long as the rate of traffic does not exceed some bound specified in the service level agreement (SLA).

2. SLAs that specify minimum rate guarantees through the ISP network for customer traffic, relative rate shares between traffic classes, maximum rates through a network for traffic classes (for protection), delay priorities, etc.

3. Under some conditions, the network provider might want to control who gets to use the network. If the provider does this, he is said to implement *admission control*. Some QoS architectures fundamentally rely on admission control, but others don't.[1] We will look at this issue later in this lecture.

Although the goal of improving application performance using QoS is laudable, that alone turns out to be poor motivation for anyone to deploy QoS. The only chance QoS has

[1]Recall that the circuit-switched telephone network implements admission control.

of being deployed in practice is if it is strongly tied to an improved revenue stream for ISPs.

It took network researchers close to a decade to fully realize this point. The initial approach to network QoS, called *Integrated Services (IntServ)* did not make significant practical impact because it did not have a clear plan for how ISPs would make money with it. The natural question is why an ISP should deploy complex machinery for a content provider to make money from end clients. The lack of a good answer to this question meant that this architecture did not have the right deployment incentives, even though the IntServ model had well-defined end-to-end service semantics for applications.

In contrast, the *Differentiated Services (DiffServ)* architecture was motivated by ISP interests, and defines *per-hop behaviors* that don't really translate into anything well-defined for applications. However, it allows ISPs to provide service differentiation between customers and their traffic classes.

The [CF98] paper covers the Diff Serv topic, while the [CSZ92] paper covers the IntServ topic. Note that the methodology for implementing these two service models in the internet today is a bit different from the descriptions in the paper. The papers provide an overview of the key issues.

Sections 9.4 and 9.5 discuss the details of DiffServ and IntServ. Before that, we discuss a few issues in QoS service models.

9.2 Debating the future service model: Best-effort v. reservations

This section is a quick summary of Shenker's paper [She95].

9.2.1 Motivation

- The Internet today offers a single class of "best-effort" service. No assurance of when or if packets will be delivered, no admission control.

- But the Internet isn't just ftp, email, and telnet any more.

- The above applications are *elastic* applications. Q: what does this term mean? (A: they can derive additional "utility" from every marginal increase in bandwidth starting from (close to) 0.)

- Starting to see more and more real-time streaming applications such as voice and video.

- Claim: these apps are not as elastic or adaptive. Delay variations and losses pose severe problems. And they don't *typically* react to congestion.

  This is a questionable assumption, but seems to be true to some extent for real-time Internet telephony.

- **Question: Should we change the underlying Internet architecture to better accomodate these applications?**

- I.e., should the fundamental best-effort Internet architecture be changed to accomodate multiple service classes? And should the architecture be changed to perform *admission control*?
- Goal of paper: To provide a concrete framework in which this debate can rage on.
- At the highest level, the "right" answers depend on the nature of future network applications, the cost of bandwidth, the cost of more complex mechanisms, etc. Hard to quantify! In particular, if all applications can become rate-adaptive, the conclusions of this paper will be questionable.
- Basic idea: multiple service classes and scheduling schemes in the network routers to allocate bandwidth.

9.2.2 What should network architects optimize?

- Throughput? Utilization? Packet drops? Delays?
- Shenker's claim: No! Really want to optimize "user satisfaction," which would in an efficient system be tied to revenue. More precisely, if s_i is the network service (in terms of throughput, drops, etc.) delivered to user i, there is a *utility function* $U_i(s_i)$ which measures the performance of the application and the degree of user satisfaction. The goal of network design is to maximize $V = \sum_i U(s_i)$.
- But what about other optimization criteria? For example, a service provider might choose to optimize revenue.
- Anyway, this formulation makes it apparent that it's worth rethinking the basic service model the architecture provides. Simple examples show that pure FIFO without favor toward particular flows doesn't always optimize V.
- But should we add multiple service classes, or just add more bandwidth?
- Note: while the goal is to optimize V, it won't necessarily optimize each of the $U_i's$. I.e., the nature of this optimization is to take bandwidth away from the elastic applications and satisfy the more sensitive, inelastic applications.

9.2.3 How is service for a flow chosen?

- Option #1: Implicitly supplied
  - Here, network automatically chooses service class based on packet/flow.
  - Advantage: No changes to end-hosts or applications.
  - Disadvantage: New applications cause problems, since routers don't know what to do about them.
  - In general, this approach embeds application information in the network layer, which has deficiencies.
  - Also suffers from stability problems, because of a changing service model, changing unknown to applications.

  - Uses multi-field classification capability to assign service to packet flow.

- Option #2: Explicitly requested
  - Here, applications explicitly ask for particular levels of service.
  - Problem: incentives. Why will some applications volunteer to ask for lower levels of service?
  - Possible solution: pricing.
  - Social implications of usage-based pricing. This will discourage the browsing mentality of users (which information providers find attractive).
  - Key point: even in a single best-effort class, the notion of incentives is important. Addresses the issue of "whether to send data."
  - Usage-based pricing at a higher granularity than an individual user might be more appropriate.
  - In the explicit model, the network service model is known to the application.

- Link-sharing as an implicit model. Works well when dealing with aggregates of flows.

9.2.4 Do we need admission control?

- Some services might need explicit resource reservation, so the network may need to turn away flows, which if admitted, will violate its current quantitative commitments.

- One approach to answering this question: If there are values of population size n and n' such that $V(n) > V(n')$ for $n < n'$, then $V()$ has a local maximum, and it may be better to turn flows away.

- Question amounts to monotonicity of $V = \sum_i U(s_i)$.

- What is the shape of $U(s_i)$ for various classes of applications?
  1. Elastic applications: concave. For this, $V()$ is monotonically increasing with n.
  2. Hard real-time apps: step function. Clearly admission control is important here (e.g., telephone networks).
  3. Delay-adaptive real-time apps: convex first then concave after inflection point.
  4. Rate-adaptive real-time apps: earlier inflection point than above case.

- If U is convex in some region, then $V()$ has a local maximum. Such a U argues for admission control.

- Q: What does this argument miss? I think it does not adequately capture the dissatisfaction of denied flows! Also, it appears to tacitly assume that in general it is better to turn a flow away that terminate an existing flow. This may not be a good assumption in some cases.

- Overprovisioning as an alternative: how much you have to overprovision depends on variance of bandwidth use. Furthermore, while backbone bandwidths will continue to improve dramatically with time, it's unclear that the same trend is true for last-mile access links.
- Need to overprovision even if you have admission control!
- Shenker concludes that admission control is more cost-effective than the amount of overprovisioning you otherwise have to do.
- Bottom line: Paper sets framework for this debate.

9.3 Principles

In my opinion, there are five key principles underlying Internet QoS models. They are:

1. Explicit v. implicit signaling ends up in radically different architectures. The former leads to end-to-end guarantees or assurances; the latter is only per-hop or perhaps per-ISP assurances. The former, however, isn't tied well with today's economic realities, whereas the latter is. To date, no one has successfully figured out how to marry the nice end-to-end properties of explicit signaling schemes with the simpler models and economic/deployment advantages of implicit approaches.

2. Isolation principle: To isolate flows, use some variant of weighted fair queueing. If the traffic feeding any single fair-queueing queue is modulated by a *linear bounded arrival process* (LBAP), then such a scheduling discipline can bound worst-case delays *independent* of what other traffic there is in the network. An LBAP has two parameters, (b, r), and modulates traffic from a source such that within any period of time t, the number of bits from the source (which could be an aggregate of many flows) does not exceed $b + r \cdot t$. That is, b specifies the largest burst size, and r the rate. Then, as long as the network of weighted fair queueing switches assures a rate of at least r to the LBAP-modulated source, the worst case delay is bounded by b/r + a few correction terms that depend on the largest packet size (these are not too important in practice). If the LBAP source gets a higher rate, g, through the network, then the corresponding delay is bounded by b/g plus correction terms.

3. Jitter sharing principle: Flows in a fair queueing network tend to have poor jitter, and in general have lower jitter in a FIFO network. The reason is that fair queueing spreads bursts out in time. Hence, the designer of a network who wishes both guarantees on delays and low jitter has a complex problem: fair queueing can give good bounds on delays, but has lousy jitter properties; FIFO has good jitter properties, but poor delay bounds.

   Researchers have spent years developing highly complex scheduling schemes to have both good delay bounds and good jitter bounds, but these are generally of limited use in practice.

4. Admission control: A good way to think about the need for admission control is using a utility function framework; in that framework, admission control boils down to whether the utility function has a convex portion.

5. Fancy QoS v. proper provisioning: In the end, if a network has persistent overload, the only correct approach is to provision the network better. No amount of fancy QoS can increase capacity; all it can do is to shift resources around between flows differently from straight best-effort FIFO, so it's inevitable that some flows are going to be unhappy. It's true that delays can be apportioned differently than in FIFO with a fancier scheduling scheme, but that's not a cure for a persistently overloaded network.

The following two sections weren't covered in Fall 2005.

■ 9.4 DiffServ Details

This section is based on notes of Balakrishnan's lecture scribed by Matt Lepinski and Peter Portante, who took 6.829 in Fall 2001.

The goal of DiffServ is to allow ISPs to make more money by offering service classes which are better than best effort. The idea behind Diff Serv is similar to airline ticket pricing, where airlines charge more to first and business class passengers who in return receive preferential treatment over those who pay the standard fare. Unlike IntServ, which attempts to provide applications with end-to-end service guarantees, with DiffServ, ISPs provide customers only with certain *per hop behaviors* and make no claims about end-to-end performance.

However, it is reasonable for customers to think that preferential treatment by their local ISP will result in better overall performance because internet bottlenecks frequently occur at the edges. Additionally, DiffServ has the following advantages over Int Serv:

- DiffServ is easier to implement than IntServ because per hop behaviors can be provided without cooperation from upstream or downstream ISPs.

- Since DiffServ guarantees are relative (i.e. Service Class 1 receives better service than Service Class 2) DiffServ can provide this differentiation for any number of flows. This means that no admission control is needed.

- DiffServ Service Level Agreements (SLA) fit nicely into the current ISP business model.

■ 9.4.1 DiffServ Implementation

DiffServ policy is implemented within a Differentiated Services Region. A DS Region is a group of routers which all use the same Diff Serv policy (typically all the routers belonging to a single ISP). Each DS Region has a set of traffic classes (6 is common in practice) and a policy for associating a traffic aggregate with a traffic class. A DiffServ Router contains the following pieces:

1. A Classifier which looks at each incoming packet and determines which traffic class it belongs in.

2. Multiple Queues, one for each traffic class.

3. A scheduler (using some scheduling algorithm like weighted fair queueing) that decides when each queue gets to send packets.

4. A Meter which determines the the rate of a particular traffic aggregate. This is important because most SLAs specify that a traffic aggregate gets a certain class of service provided that the traffic aggregate doesn't exceed some load. If the rate for a particular traffic aggregate is too high (it is out of profile) then the excess packets either need to be dropped or the traffic needs to be shaped. (Shaping traffic involves holding excess packets in a buffer and letting them trickle out at the rate specified in the SLA).

5. A Dropper/Shaper to deal with traffic which the meter determines to be out of profile

Unfortunately, correctly classifying and metering each incoming packet takes valuable cycles and so it is generally not feasible to do this work at every router within an ISP. Typically, edge routers have extra cycles available for these kind of tasks but high-speed core routers do not. The solution to this problem is to use dynamic packet state (similar to the mechanism used in the Core Stateless Fair Queueing paper). Edge routers do classification and metering and then have a *Marker* which sticks the result of the classification into the IP header (the portion of the IP header which is used is referred to as the DiffServ Code Point (DSCP)). The entire process of classifying, metering, marking and shaping/dropping packets in the edge router is known as *traffic conditioning*.

The picture at the edge looks like this:

```
                           +-------+
                           |       |-------------------+
                    +----->| Meter |                   |
                    |      |       |--+                |
                    |      +-------+  |                |
                    |                 V                V
              +------------+      +--------+      +---------+
              |            |      |        |      | Shaper/ |
packets =====>| Classifier |=====>| Marker |=====>| Dropper |=====>
              |            |      |        |      |         |
              +------------+      +--------+      +---------+
```

9.4.2 Dealing with Excess Traffic

As noted earlier, the Meter in the edge router measures the rate of a traffic aggregate and determines if traffic is out of profile. One solution is to simply drop every packet that is out of profile at the edge router. However, traffic is very bursty and since not all sources are continually sending at their maximum rate, it is possible that the system is able to handle the excess packets. Therefore, dropping at the edge router is often undesirable. The solution then is to let all of the packets through at the edge, but to mark those packets which are out of profile so that they can be dropped later if the network does not have the capacity to handle the extra traffic.

We then want some mechanism for a bottleneck core router to give preferential treatment to those packets which are marked as being in profile. The easiest solution would be to maintain separate queues for in profile and out of profile packets. This is a bad idea because often a particular flow contains both in profile and out of profile packets and so maintaining separate queues would lead to a large amount of reordering. The solution is to run RED on each queue in the router but to have RED treat each queue as two virtual queues. This algorithm is known as RIO (RED IN/OUT).

RIO maintains a variable q_{ave-in} which is the average number of in profile packets in the queue. RIO then uses this variable along with the RED parameters MIN_{in} and MAX_{in} to determine when to drop in profile packets. Additionally, RIO maintains a variable q_{tot} which is the average size of the queue (including both in profile and out of profile packets). RIO uses this variable along with the RED parameters MIN_{out} and MAX_{out} ($MIN_{out} < MIN_{in}$ and $MAX_{out} < MAX_{in}$) to determine when to drop out of profile packets.

■ 9.4.3 In Practice

In practice, DiffServ implementations use six traffic classes. These classes include one Expedited Forwarding (EF) class, Four Assured Forwarding Classes and one Best Effort class. Standard implementations have the following properties:

- No reordering of packets within a traffic class.
- No standard for bandwidth allocation between classes.
- For each class, there are three drop precedences.
- Expedited Forwarding has the property that whenever a packet is in this queue it gets sent out immediately. Note: This requires that edge routers drop all out of profile EF packets (since core routers never drop EF packets, you can't just mark EF packets out of profile and send them on).

■ 9.4.4 Paper contributions

The authors of [CF98] provide two notable contributions. The first is the movement of packet classification to the edge routers using DSCP fields in the IP header. The second is a set of policies on how to deal with excess traffic, in particular the RIO scheme.

■ 9.5 IntServ Details

This section is based on notes of Balakrishnan's lecture scribed by Matt Lepinski and Peter Portante, who took 6.829 in Fall 2001.

In contrast to DiffServ, IntServ takes an end-to-end approach to the task of providing a QoS beyond Best Effort to application flows through absolute or statistical guarantees. Note that most of the complexity of IntServ results from handling multicast. Additional difficulties arise attempting to perform effective admission control in the face of "controlled load".

9.5.1 End-to-End QoS classes

There are three end-to-end QoS classes:

- Best Effort Service

  As the default class of service provided by the internet, this service makes no guarantees for delivery rate, capacity or even successful delivery of data transmitted through the network.

- Guaranteed Service

  ([RFC 2212] provides the in-depth description of this service)

  This class of service provides firm (mathematically provable) bounds on end-to-end datagram queueing delays. This class of service makes it possible to provide a service that guarantees both delay and bandwidth. [RFC 2212]

- Controlled Load Service

  ([RFC 2211] provides the in-depth description of this service)

  This class of service provides the client data flow with a quality of service closely approximating the QoS that same flow would receive from an unloaded network element, but uses capacity (admission) control to assure that this service is received even when the network element is overloaded. [RFC 2211]

9.5.2 End-to-End QoS Mechanisms

We have covered aspects of the internet's existing end-to-end Best Effort service in previous lectures. Here we describe how Guaranteed and Controlled Load services achieve their provided service levels.

These last two services share three aspects of their design. They contain a signaling protocol used to request service, an admission control scheme to help ensure proper service, and scheduling algorithms in routers which provide the basis for the service.

- Signaling protocol

  A signaling protocol is used to request one of the last two classes of service. The signaling mechanism provides the traffic characteristics (or specification) and a reservation for the resulting traffic specification provided.

  The Resource ReSerVation Protocol (RSVP, [RFC 2205]) is a signalling protocol which takes a traffic specification (TSpec) as input, and returns a PATH message confirming the resource reservation as output. Receivers initiate the reservation request which allows the protocol to accommodate both unicast and multicast traffic flows.

  A TSpec consists of the following information:

  - Maximum Transmission Rate
  - Average Transmission Rate
  - Maximum Burst
  - Minimum Packet Size

  - Maximum Packet Size

  As a TSpec flows through the network, each router provisions resources based on a flow's TSpec, adjusting the TSpec based on what it decides it can handle before forwarding it on to the next router.

  Once the TSpec arrives at the sender, a PATH message is returned to the receiver on the reverse path confirming the reservation. For Guaranteed Service, the PATH message also contains an RSpec, which indicates the level of service reserved for this flow.

  Since each router along the path implicitly keeps a record of a flow's reservation, to avoid keeping track of too much state, routers throw out this information after a certain period of time. This means the state of reservation has to be periodically refreshed.

  There are two aspects to signaling which complicate the design, support for multicast and route changes (since neither service appears to pin routes down).

  Below is a summary of the characteristics of the RSVP signaling protocol taken directly from [RFC 2205]:

  - Makes resource reservations for both unicast and many-to- many multicast applications, adapting dynamically to changing group membership as well as to changing routes.
  - Is simplex, i.e., it makes reservations for unidirectional data flows.
  - Is receiver-oriented, i.e., the receiver of a data flow initiates and maintains the resource reservation used for that flow.
  - Maintains ”soft” state in routers and hosts, providing graceful support for dynamic membership changes and automatic adaptation to routing changes.
  - Is not a routing protocol but depends upon present and future routing protocols.
  - Transports and maintains traffic control and policy control parameters that are opaque to RSVP.
  - Provides several reservation models or "styles" (defined below) to fit a variety of applications.
  - Provides transparent operation through routers that do not support it.
  - Supports both IPv4 and IPv6.

  The use of RSVP under Guaranteed and Controlled Load services is specified in [RFC 2210, 2211 and 2212].

- Admission control

  In order to provide a given service level it is necessary to only admit flows up to the point where the service could no longer be provided, e.g. the telephone system. Guaranteed and Controlled Load services use admission control in slightly different ways.

- Scheduling algorithms

  Guaranteed and Controlled Load services require scheduling algorithms in network routers in order to implement their QoS. Variants on Weighted Fair Queueing are used with Guaranteed Service and FIFO variants are used under Controlled Load.

9.5.3 Guaranteed Service

Guaranteed Service provides an assured level of bandwidth to a conforming flow, with a bounded delay and no queueing losses. This is accomplished by combining parameters from individual network elements (routers, subnets, etc.) to produce the guarantee of service.

A conforming flow adheres to a token bucket behavior. Given this description of a flow, a network element computes various parameters describing how the element will handle the flow's data. By combining these individual parameters, it is possible to compute the maximum delay a piece of data will experience on the flow's path (assuming no network element failures or routing changes during the life of the flow). [RFC 2212]

A flow can be described as a token bucket flow with parameters (r, b), where r is the average rate and b is the maximum burst size, if for any time interval, T, the number of bytes sent during T is $\leq (r * T + b)$.

Signaling

The RSVP protocol is modified to add an RSpec along with the TSpec when a receiver initiates its request. A RSpec consists of simply a rate term, R, and a slack term, S, where the slack term is the difference between the requested rate and the reserved rate.

The returned PATH message then contains a modified RSpec describing the exact level of service provided to the requesting flow.

Admission Control

Admission control is straightforward for this service. When a new flow requests service, it is only added if the flow does not cause the combined set of flows in service to exceed the network's capacity.

$$\sum_i alloc + req \leq \text{available G.S. bandwidth}$$

It should be noted that under bursty traffic, the network might end up underutilized. This is a potential drawback to this service.

Scheduling

The routers participating in a guaranteed service offering use a variant on Weighted Fair Queueing for scheduling flows.

Since the flow's traffic conforms to a token bucket behavior, then if WQ: $R_i >= r_i$, there is a maximum bound on end-to-end delay:

$$\text{Max queue } delay < \frac{b}{r_i}$$

Summary

In summary, there are two costs to guaranteed service:

- Potential under-utilization of the network caused by rejecting flows.
- Higher per-flow jitter caused by WFQ spreading out bursts.

9.5.4 Controlled Load

This service class tries to remove the costs associated with Guaranteed Service. It relaxes the admission control policy to address network under-utilization issues, and uses FIFO queueing in the routers to address high jitter.

The end-to-end behavior provided to an application by a series of network elements providing controlled-load service tightly approximates the behavior visible to applications receiving best-effort service *under unloaded conditions* from the same series of network elements. [RFC 2211]

Signaling

Uses the receiver initiated RSVP protocol, but does not rely on RSpec like data as in the Guaranteed Service. Clients provide a TSpec estimating the data traffic they generate, just like they would for Guaranteed Service. A PATH message is returned, but if the clients traffic should exceed its previously specified TSpec, the client's flow may encounter delayed or lost packets.

Admission control

Unlike Guaranteed Service, where admission control is based on whether or not a flow, considering its TSpec, can be strictly accommodated by the network, here network elements are allowed to admit flows with some level of over-commitment. A network element could admit a flow even though the sum of all existing flows' requested TSpecs is greater than the available network capacity, if it notices that flows in general are falling short of their TSpec.

So a network element use the following algorithm to determine if a flow is admitted:

$$\sum_i TSpec_{avg} < \text{avail C.L. bandwidth}$$

The network element sums all known flows' average TSpec, which is based on individual observation, instead of taking it from the RSVP setup message. This would allow network elements to accommodate fluctuations in flow traffic rates while maximizing the number of allowed flows.

Scheduling

In order to alleviate the effects on bursty traffic by WFQ scheduling, routers use FIFO scheduling variants. A FIFO will receive a burst and transmit a burst without spreading out the packets as WFQ thus reducing jitter.

CHAPTER 10
Design of Big, Fast Routers

This lecture discusses why high-speed Internet routers are complex and briefly surveys how they are designed. The main ideas to understand are: (1) the design constraints, such as speed, size, and power consumption, (2) the components in a modern (early 2000s) router (such as ports, line cards, packet buffers, forwarding engine, switch arbiter, switch fabric, link scheduler, etc.), and (3) the need for efficient algorithms for lookups, packet queueing, and switch arbitration. We will focus on the general ideas rather than on how a specific router is implemented; these notes are best read together with specific case studies such as BBN's multi-gigabit router (which is a relatively old design) and the somewhat more recent Stanford TinyTera system.

■ 10.1 Introduction

In the first few lectures of this course, we looked at the architecture of packet-switched networks and saw that the *switch* is a fundamental building block in such networks. In this lecture, we will look at the problem of designing an packet switch for IP packet processing that can support links of high speeds.[1] Currently (early 2000s), high-speed Internet routers that process IP packets have links whose speeds are in the range of 1 Gigabit/s to about 40 Gigabits/s per link. A typical high-end router might have between 500 and 1000 *ports*, distributed across perhaps 64 or 128 *line cards*. The ports usually correspond to distinct "next hops".[2]

The conceptual model of a router is quite straightforward: a router has a *data path* and a *control path*. The job of the data path is to move packets that arrive on *input* or *ingress* ports to *output* or *egress* ports; from the outputs, these packets are sent on downstream to the next switch on the path or to an end-host. The job of the control path is to implement algorithms, which are typically distributed, that help the data path figure out the right port to send the packet out on. In high-speed switches, the data path typically requires a hardware implementation, while the control path is almost always done in software.

[1]A switch in the context of Internet datagram delivery is generally called a *router*, because it also runs routing protocol software to build forwarding tables. We will use the terms "switch" and "router" interchangeably in this lecture.

[2]These numbers change every year or two.

Routers are a great example of special-purpose computers built for a specific set of tasks. Routers are big business: according to some market research firms, worldwide revenue from enterprise and service provider router sales in 2003-2004 was about US $5.5 billion (source: Infonetics Research Report, Q2 2004).

Most of this lecture will be concerned with the data path of a router, focusing on how routers can implement the data path efficiently. There are three parts to this lecture. We will start by looking at an abstract model of an Internet router, describing the key components of a "bare-bones" (but functional) router, and giving a broad sense of how routers have evolved over the past several years. Then, we will figure out how to make our bare-bones IP router fast, a process that involves understanding hardware characteristics and trends, some clever algorithms, and will give you a sense for why even simple IP routers are complex. Finally, we will discuss "feature creep"—IP routers today implement much more than our bare-bones router, leading to an daunting degree of complexity.

The earliest routers (until the mid-to-late 1980s) were built by having no packet buffer memory on line cards and by attaching each line card to a shared bus backplane. Packet buffer memory was also attached to the backplane, and all forwarding decisions were made by a central processor also connnected to the backplane. As packets arrived, they were stored in memory by moving the bits over the shared bus, after first passing the packet (or the header) to the processor to determine the egress link for the packet.

This design, while conceptually simple, has a scaling bottleneck: the shared bus. Various caching-based optimizations were developed to alleviate this bottleneck (e.g., caching parts of the forwarding table).

Modern high-speed routers don't use a shared bus: they use a *crossbar*, which is a hardware component that permits more parallelism than a shared bus. Conceptually, a crossbar has N input ports and N output ports (in general, every input port is also an output port). The crossbar has a *scheduler* that typically works in units of time-slots. In each time-slot, the crossbar can move some number of bits between ports across its fabric, subject to a *crossbar constraint*. This constraint states that at any given time, if input i is connected to output j, then no other output can be simultaneously connected to output j, and no other input can be simultaneously connected to input i. In other words, one can think of the crossbar as forming a bipartite graph with the input ports being one partition and the output ports being the other, and an edge from i to j signifies that input i has a packet destined for output j. Then, at any given time, the crossbar scheduler can pick some *matching* along which bits can be moved concurrently across the crossbar.

■ 10.2 High-Speed Routers are Complex

The fundamental problem in the design of high-speed routers is that the designer can't simply sit back and let Moore's law take care of processing packets at the desired speeds. They must solve a number of problems efficiently, and of these many problems, we survey two: fast IP lookups and fast crossbar scheduling.

■ 10.2.1 Fast Lookups

A router needs to implement a *prefix match* to check if the address being looked-up falls in the range A/m for each entry in its forwarding table. A simple prefix match works when the Internet topology is a tree and there's only one shortest path between any two networks in the Internet. The toplogy of the Internet is not a tree, however: many networks *multi-home* with multiple other networks for redundancy and traffic load balancing (redundancy is the most common reason today).

The consequence of having multiple possible paths is that a router needs to decide on its forwarding path which of potentially several matching prefixes to use for an address being looked-up. By definition, IP (CIDR) defines the correct lookup response as the *longest prefix* that matches the sought address. As a result, each router must implement a *longest prefix match* (LPM) algorithm on its forwarding path.

LPM is not a trivial operation to perform at high speeds of millions of packets (lookups) per second. For several years, the best implementations used an old algorithm based on *PATRICIA trees*, a trie (dictionary) data structure invented decades ago [**?**]. This algorithm, while popular and useful at lower speeds, does not work in high-speed routers.

To understand the problem of high-speed lookups, let's study what an example high-speed Internet router has to do today. It needs to handle minimum-sized packets (e.g., 40 or 64 bytes depending on the type of link) at link speeds of between 10 Gbits/s (today) and 40 Gbits/s (soon), which gives the router a fleeting 32 ns (for 40-byte packets) or 51 ns (or 64-byte packets) to make a decision on what to do with the packet! (Divide those numbers by 4 to get the 40 Gbits/s latencies.)

Furthermore, a high-speed Internet router today needs to be designed to handle on the order of 250,000 forwarding table entries, and maybe more (today's Internet backbones appear to have around 120,000 routes, and this number has been growing; people expect a commercial router to have a lifetime between five and ten years in the field).

30–50 nanoseconds per lookup is difficult to achieve without being clever. One can't really store the forwarding tables in DRAM since DRAM latencies are on the order of 50ns, and unless one has a prohibitively large amount of memory, doing an LPM in one lookup is impossible.

We are left with SRAM, which has latencies (on the order of 5 ns) that are workable for our needs. Furthermore, over the past couple of years, SRAM densities have approached DRAM densities, allowing router designers to use SRAM for largish forwarding tables.

We can formulate the forwarding table problem solved in Internet routers as follows. Based on the link speed and minimum packet size, we can determine the number of lookups per second. We can then use the latency of the forwarding table memory to determine M, the maximum number of memory accesses allowed per lookup, such that the egress link will remain fully utilized. At the same time, we want all the routes to fit into whatever amount of memory (typically SRAM), S, we can afford for the router. The problem therefore is to maximize the number of routes that fit in S bytes, such that no lookup exceeds M memory accesses.

This problem has been extensively studied by many researchers in the past few years and dozens of papers have been published on the topic. The paper by Degermark *et al.* in the optional readings describes one such scheme (now called the "Lulea" scheme), which uses aggressive ompression to fit as many routes as possible into S bytes. Although their paper was originally motivated by software routers, these are the same optimizations that make sense in fast hardware implementations of the IP forwarding path.

The Lulea scheme observes that a forwarding table viewed as a *prefix tree* that is complete (each node with either 0 or 2 children) can be compressed well. The scheme works in three stages; first, it matches on 16 bits in the IP address, and recursively applies the same idea to the next 8 and the last 8 bits. See the paper for details.

10.2.2 High Throughput in a Crossbar-based Switch Fabric

The key problem is crossbar scheduling: finding a matching in the bipartite graph efficiently.

This requires a number of optimizations, and many approaches have been proposed in the literature (and several designs have been realized in practice).

The first important optimization is *virtual output queueing*. If there is only one queue per input storing all packets regardless of their destination output port, then *head-of-line* blocking ensues. In the presence of such blocking, the maximum throughput of the switch does not exceed $2 - \sqrt{2}$. All crossbar switches now implement virtual output queueing, where each input has (at least) N separate queues, one for each output. (They may have additional queues for each output if they implement QoS.)

Perhaps the earliest crossbar scheduler was the Wavefront arbiter, later implemented in the BBN switch. The idea here is to produce a maximal matching by proceeding in a "wavefront" across the matrix of input-output demands. The rows of the matrix correspond to the inputs, and the columns to the outputs. Each element in the matrix is 0 if there are no packets waiting on that pair, and non-zero otherwise. The wavefront arbiter produces a matching by starting with the (0,0) entry, and moving diagonally across the matrix. It can match all previously unmatched non-zero elements at the same time, because along any (minor) diagonal, there can be no contention for the same input or output. Hence, by doing $2N - 1$ such sweeps across the matrix, it produces a maximal matching. To improve fairness, the order of the sweeps can be randomized.

Because the Wavefront might be too slow for some systems, other schemes were also developed. An early switch scheduling scheme was *Parallel Iterative Matching* (PIM), pioneered in DEC's AutoNet switch (Anderson, Owicki, et al.). PIM is a randomized matching scheme that operates in three phases: request, grant, and accept.

In the request phase, each input port sends "requests" to all outputs for which it has packets. In the grant phase, an output picks an input *at random*, and sends a grant to it. At this stage, no output is committed to more than one input, but an input may receive a grant from more than one output. Hence, in the final "accept" phase, an input picks one of the granting outputs *at random*.

It is easy to see that this three-phase approach produces a matching, but the matching is not maximal: i.e., there may an input and output that are each unmatched, and which have packets for each other. To produce a more complete matching, PIM runs the above three-phase procedure a number of times, eliminating at each stage all previously matched nodes. They show that doing that about $\log N$ times results in maximal matchings most of the time.

The problem with PIM is that it can lead to unfairness over short time scales. McKeown's iSLIP algorithm overcomes this problem by removing the randomness. Each input maintains a round-robin list of outputs, and each output maintains a round-robin list of inputs. The request phase is the same as in PIM. In the grant phase, rather than pick at random, each output picks the first input in the round-robin sequence from the previously matched input for that output, and updates this state only if this input is picked in the accept phase. Similarly, in the accept phase, an input picks the first output in round-robin sequence following the previously matched output for that input.

Simulations show that this scheme has good fairness properties. It is implemented in many commercial routers.

For several years, it was not clear whether these simple schemes achieve 100% throughput in a switch. Several complex scheduling schemes were shown to achieve 100% throughput, but they were impossible to implement. Dai and Prabhakar showed a breakthrough result a few years ago, proving that *any maximal matching* scheme running in a crossbar switch with $2\times$ speedup can achieve 100% throughput. (The speedup of a switch is the relative speed of the internal links of the crossbar to the input and output link speeds.)

■ 10.3 Death by a Thousand Cuts: Feature Creep in Modern Routers

The data path often does more than simply forward packets in the order in which they came in. The following is a sampling of the "check-box" data-path functions that today's high-speed routers tend to have to support:

1. Packet classification, involving processing higher-layer protocol fields.
2. Various counters and statistics for measurement and debugging.
3. IPSec (security functions), especially at "edge" routers. More sophisticated security services such as Virtual Private Networks (VPNs). Firewalls.
4. Quality-of-service, QoS (I): Rate guarantees and traffic class isolation.
5. Packet "shaping" and "policing" functions.
6. IPv6.
7. Denial-of-service remediation schemes (the simplest of which is access control based on ingress filtering of packets that don't have a valid source address for the incoming interface that the packet came in on) and traceback (figuring out the Internet path along which a given packet or stream of packets came).
8. IP multicast.
9. Packet-drop policies ("active queue management") such as RED.
10. QoS: Differentiated Services (differentiating traffic classes using suitable scheduling).
11. QoS: Integrated Services (end-to-end delay/rate guarantees).

We won't discuss these features in detail here, nor will we pass judgement on which of these is actually a feature users care about and which are simply "check box" items. Some of these features have been covered in previous lectures in the course, either because they are intellectually interesting or because some users care about them.

Appendix A: A "cheat sheet" summarizing the main ideas in the BBN 50 Gb/s router

10.3.1 Why bother?

- Trends
  1. Faster and faster link bandwidths.
  2. Larger and larger network size (hosts, routers, users).
  3. Users want more and more (15% increase in per-user demand in 1 year!).
  4. Not just because of Web: for example, Internet size has been growing at 80% per year since at least 1984!
  5. Conclusion: Unlike other areas in computer science that can rely on Moore's law to achieve progress, many aspects of networking cannot!
- *Conventional "wisdom":* IP routers are inherently slow. They cannot possibly forward packets fast enough, where "fast" refers to multi-gigabit (or higher) rates.
- This paper's motivation, at least in part, was to refute this belief.

10.3.2 How does it work?

- Routers do two things: participate in routing protocols with their neighbors, and forward packets. We are concerned primarily with the latter, since it's the data path.
- At first sight, forwarding is straightforward:
  1. Router gets packet.
  2. Looks at packet header for destination.
  3. Looks up routing table for output interface.
  4. Modifies header (ttl, IP header checksum).
  5. Passes packet to output interface.
- But there's a lot of stuff to take care of: RFC 1812 is 175 pages long!
- **Challenge.** How to do all this *fast.* In particular, in the common case.
- **Architecture.** Consists of:
  1. Multiple line cards.
  2. Multiple forwarding engines.
  3. High-speed switch fabric and processor.
  4. Network processor.
- Salient features of architecture:
  1. Separate forwarding engine from line card. From input line card, send only header to engine. Engine returns modified header to input line card, after which it gets vectored to output line card.

2. Each forwarding engine has its own routing table (cache). Notice that only a subset of the complete routing table, which gives prefix-to-output-interface mapping is needed. This is therefore a *forwarding table* rather than complete routing table.
3. Use a switched backplane rather than a shared bus. Custom-designed for IP rather than an ATM switch. The big advantage of this is parallelism.
4. QoS processing in router (on output line card).
5. An abstract link-layer to handle different link technologies.

- **Forwarding engines:**
  - Key design principle: *optimize the common case.*
  - Not ASIC but based on Alpha 21164 at 415 MHz.
  - 8 KB I and D caches: forwarding code.
  - 96 KB L2 cache (Scache): cache of routing table. Only relevant table entries. 12000 routes (95% hit rate).
  - 16 MB L3 cache (Bcache): complete forwarding table in 2 8 MB chunks. [Why is it banked like this?]
  - Why not an ASIC or embedded processor? Want software. And chip has a very high clock speed and large Icache and Scache.
  - Optimize common case in custom assembler. 85 instructions to forward a packet.
  - Note: Does not check IP header checksum. Q: is this a problematic drawback?
  - Not-so-common cases deferred to network processor. This includes route cache misses, multicast, IP options, packets requiring ICMP (uses templates), packets needing fragmentation.
  - Q: Do we need a route cache in the future? A: depends on what algorithm you choose to forward.
- **Switching fabric:** (Draw pic. of switch)
  - Advantage over bus? Parallelism.
  - Disadvantage? Multicast is hard. Need scheduling machinery. (They call this "allocation").
  - Input-queued switch. Q: what's the problem with standard input-queued switch? A: h-o-l blocking.
  - How to fix this? Keep track of per-output port traffic. Each input port bids for different outputs. In each epoch, match an input to output port.
  - Allocator arbitrates amongst bids to do this. Maximize switch throughput (rather than bounded latency).
  - Scheduling is pipelined (4 stages): get bids, allocate schedule, notify pairings, ship bits.

  - Q: What graph problem does this correspond to? A: bipartite matching.
  - Flow control by destination ports. Prevents an input from hogging an output.

- **Allocation algorithm:**
  - Heavily studied problem. Approaches include Parallel Iterative Matching (from DEC), wavefront scheduling, etc.
  - Use *randomization* to alleviate unfairness issues.
  - Essentially a greedy algorithm that progresses along diagonal of allocation matrix.
  - Prioritize forwarding engine inputs over line cards (by skewing shuffling of ports).

- **Line cards:**
  - Input: simple, except for ATM (small cells) and multicast (copies have to be made).
  - Output: QoS processor for scheduling and queue management. A VLIW FPGA.

- **Network processor:**
  - Alpha processor running NetBSD. (Another nice example in this architecture of using off-the-shelf components.)
  - Handles "not-so-common" cases of packet processing.
  - Handles routing protocols (gated).
  - Builds forwarding tables and populates table for each forwarding engine using a subset of each entry.
  - Using 2 banks in forwarding engines reduces disruptions and flapping effects; allows network processor to upload a forwarding table while forwarding is still in progress.

10.3.3 Fast lookups

- Two approaches from Sigcomm 97 papers (current state-of-the-art)
  1. Make better use of caching via compression (Degermark et al.).
  2. Constant-time lookups via clever data structures (Waldvogel et al.).

- Problem: Best-matching prefix.

- Basic idea: use hash tables for each prefix length. Then, apply binary search techniques to get to the longest matching prefix to find output interface.

10.3.4 Performance

- Peak performance of about 50 Gigabits/s, assuming 15 simultaneous transfers take place through the switch in a single transfer cycle (called an *epoch*, equal to 16 clock ticks). (Notice that it includes the line card to forwarding engine transfers too!)
- Note: this is only an estimate, not much deployment experience/measurements reported in paper.

CHAPTER 11
Wireless Network Architectures and Channel Access Protocols

This lecture introduces the problems and challenges in the design of wireless networks and discusses the area of channel access protocols (aka "medium access", or MAC protocols).

■ 11.1 Wireless Networks are Different

Communicating over radio is an old idea, and people have been doing it for a long time. Data networks have been around for about four decades. Combining the two is attractive because wireless networks offer important potential benefits over their wired counterparts: they enable mobile communications, they don't require as much investment in and upkeep of a wired infrastructure, and they enable broadcasts in a natural way.

For many years now people have been building data networks that operate over radio channels. One lesson that has been learned is that radios are different from wired links and affect the design of networks in important ways: radios make the design of data networks both interesting and challenging.

Realizing the potential benefits of wireless networks in practical, working systems is difficult for one deceptively simple reason: *radios are not wires*. To understand the implications of this statement, consider resource sharing in a wired network. Placing a "network layer" over the "link" and "physical" layers works well in wired networks because the network designer has to worry about sharing only at the network layer (and higher): links themselves (and therefore the link and physical layers) are shielded from one another and concurrent data transmissions on two or more wired links have no influence on each other unless those transmissions interfere at a switch (i.e., at the network layer).

In contrast, wireless transmissions are over a shared medium with very little shielding, especially when omni-directional antennas are used. The problem is that two radio transmitters that are near each other will interact adversely with each other, especially if their intended receivers can hear both transmissions. Such "link-layer" and "physical-layer" interactions don't usually occur with wired links (except for shared media such as Ethernet, but as we'll see in a bit, coping with the problem in Ethernet is much easier because packet collisions can be detected reliably in an Ethernet).

One approach to designing a robust radio-based wireless network is to make each communicating pair look like a robust point-to-point link. If that can be achieved, then the radio channel between a pair of nodes starts to look like a "link". The underlying principle here is to ignore all other transmissions and focus on just one pair, and attempt to achieve the Shannon capacity for that communicating pair:

$$C \leq B \cdot \log_2(1 + \frac{S}{N}), \tag{11.1}$$

where B is the bandwidth of the communication channel in Hz, S is the signal level at the intended receiver, and N is the background noise. The signal level, S, itself attenuates with distance, so if the transmitter sends data at a power level P, then S usually drops off with distance d as d^{-n}, where n is a small number (2 in free space, and perhaps 4 indoors). In the formula above, the log term is the maximum number of bits that can be sent; practical systems attempt to achieve that information-carrying capacity using various modulation and coding schemes.

The idea of making each pair of radio transmissions look like a "link" is a reasonable way to manage the complexity of the system, but does not eliminate the channel-sharing issues that must be solved. In particular, the goal of many wireless network designers is to maximize the aggregate data delivery capacity of the network, while allocating that capacity in a reasonably fair way (e.g., to prevent gross unfairness, avoid starvation, etc.). Achieving this goal is hard for the following reasons:

- While engineering a single communicating pair to come close to the Shannon capacity is possible, engineering S/N on a *network-wide* basis is very hard. In fact, the answer to the question, "What is the maximum capacity of a wireless network (as opposed to "wireless link")?" is still not known.

- Radio channels vary in quality with time and depend on location: various studies have shown that channel variations occur across multiple time scales, from a few bit-times to much longer. The result is that channel bit-error rates vary with time and space, and errors often occur in bursts. Coping with these variations to maximize capacity is not easy.

- A graph is not the best abstract model of a wireless network. Because radios are inherently broadcast media, packet reception is often probabilistic. The number of nodes that will successfully receive a packet is hard to predict, and determining a graph among the nodes to model likely collisions is also very hard. As a result, the traditional approach of solving the routing problem by finding paths in a graph (topology) is not the best approach to maximizing capacity.

In addition, designing good routing protocols, coping with mobility, achieving good TCP performance, and conserving energy on battery-operated devices are all problems that are non-trivial in wireless networks.

■ 11.2 Architectures to Increase Spatial Reuse

There are two different kinds of wireless network architectures: *cellular* and *ad hoc* (or "mesh"). Cellular architectures achieve high capacity by partitioning the network into *cells* and by suitably *provisioning* resources within and between neighboring cells. For example, neighboring cells might be provisioned to use different frequencies, to reduce the likelihood that concurrent transmissions in two nearby cells will interact adversely. Usually, cellular networks require only one wireless hop before packets from a wireless device reach an access point or base station connected to a wireline network infrastructure. Cellular wireless networks are the most common form today.

A very different wireless network architecture might be required when the network has to be deployed quickly in an area where pre-planning and provisioning is not easy or is impossible, or if the network is to be deployed in remote areas. Such *ad hoc* wireless networks usually involve multiple wireless hops between communicating entities, some of which might even be access points connecting devices to a wireline infrastructure. Ad hoc wireless networks, originally proposed for mobile devices in remote areas (e.g., for applications such as disaster relief or the military), are now gaining in popularity in two places: in wireless sensor networks, and in wireless "mesh" networks to bring inexpensive Internet access to users. In both cases, a key feature of these networks is the use of multiple wireless hops and absence of careful provisioning or planning in their deployment.

Interestingly, WiFi deployments in many locations today are set up as one-hop "cellular" systems, but because they are often run by independent parties (e.g., people in different homes or office buildings), they tend to exhibit the same "anarchic" and unplanned characteristics of ad hoc wireless networks. Understanding how to achieve high capacity in these networks is, in many ways, similar to the same goals in multi-hop wireless ad hoc networks.

Cellular wireless networks achieve spatial reuse using two techniques: resource provisioning (e.g., frequency allocation) and power control. The idea is that access points and wireless devices transmit in pre-configured frequencies (or using pre-allocated codes or time-slots) and at one of a set of pre-defined power levels, so as to control how much they will interact with other transmissions in nearby cells.

■ 11.3 Wireless Channel Access Protocols

Wireless channel access (aka MAC, for media access) protocols attempt to share the wireless channel amongst multiple contending transmitter-receiver pairs in the same neighborhood. The goal of these schemes is to maximize capacity within the "local neighborhood" by attempting to make every transmission count. Because concurrent transmissions in the same neighborhood might collide and cause packet corruption, the goal is to manage which nodes are allowed to send at the same time. These protocols don't take a network-wide view of the sharing problem, focusing instead on just "local" radio regions. Such protocols are also called *multiple access* protocols because they arbitrate transmissions amongst multiple concurrent users.

A popular MAC protocol design, CSMA, uses the *carrier sense* mechanism. Before transmitting a packet, the sender *listens* on the channel to determine if any other transmission is in progress. If it is, then the sender *defers*, waiting until the channel becomes idle. There is a rich literature in CSMA protocols: schemes differ based on how persistently they try when the channel is idle (e.g., a node may send with probability p when the channel is idle), and in how nodes detect collisions. The latter problem is easy on a wired Ethernet because a sender can detect a collision reliably on an Ethernet.

In contrast, a radio transmitter cannot detect collisions reliably, because the receiver and sender don't "share fate" with respect to successful packet delivery. For example, the receiver may be near a source of noise or near other interfering transmitters that cause the packet to be corrupted because the signal strength, which attenuates with distance, is not high enough (cf. Eq. 11.1). The approach of just listening after a radio transmission to look for evidence of collision will not work without additional machinery (e.g., explicit ACKs or NACKs from the intended receiver). Hence, CSMA protocols for wireless channels cannot use collision detection mechanisms as in Ethernet, but require other *collision avoidance* mechanisms to reduce the likelihood of repeated collisions.

Wireless MAC protocols must handle the following problems:

1. Use suitable collision avoidance schemes to reduce the number of wasted transmissions.

2. Provide reasonable fairness among contending nodes.

3. Cope with *hidden terminals*: a hidden terminal situation occurs when two nodes, A and B, cannot hear each other (when they each sense carrier when the other transmits, not much energy is detected), but a node C hears both of them. The problem is that both A and B may end up transmitting at the same time, and if one (or both) of those transmissions is destined for C, C may not be able to receive the packet successfully.[1]

4. Take advantage of *exposed terminals*: Consider four nodes, A, B, C, and D, where A and B are near each other, C can receive packets from A but not B (even when B transmits at the same time as A), and D can receive packets from B but not A (even when A transmits at the same time as B). With carrier sense in place, A and B will detect each others' transmissions, and only one of them would transmit at a time, even though it might have been possible for both of them to concurrently send data to C and D respectively.

No MAC protocol successfully solves all these problems in all situations today. Most practical MAC protocols favor reducing collisions over maximizing every bit of available capacity. The rest of this lecture discusses the ideas behind three kinds of MAC protocols: CSMA with collision avoidance (CSMA/CA), reservation-based protocols, and time-division multiple access (TDMA) protocols.

[1]In some cases, when both nodes are sending to C, one of them may be able to "capture" the channel and have its data received successfully. We don't worry about this capture effect further in this lecture, although it is important in some networks in practice.

■ 11.4 Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA)

The best current example of CSMA/CA is in the 802.11 (WiFi) family. The general idea behind such protocols is as follows:

Each node maintains a *contention window*, CW, and before transmitting data, picks a random "slot" in the range [0, CW]. Each slot is a fixed (small) amount of time, and all transmissions must start at slot boundaries. (Years ago, the Aloha system showed that a slotted distributed multiple access system has higher capacity than an unslotted one, because it forced all collisions to occur at slot boundaries rather than spread them out so they can happen anywhere through the entire packet transmission).

In 802.11, when a node has data to transmit, it picks a random number in [0, CW]. The node *listens* by sensing the carrier. On each idle slot, the node decrements a countdown timer by 1. When that timer reaches 0, and the channel is sensed as "idle", the node sends data.

During the countdown process, if the node senses the carrier as "busy" (usually done by comparing the energy with the "background noise" level gathered when the channel was believed to be "idle"), then the node *defers* transmission. *At this time, it "holds" its countdown timer, until the carrier is sensed as "idle" once again.* We will return to discussing this "holding" step in a bit.

But how does a node know when there are a large number of collisions occuring? One approach is for the protocol to include a *link-layer* ACK on each frame. This is the approach used in 802.11 for unicast transmissions. The absence of an ACK, which is sent "synchronously" and with a very short time delay from the receiver, signals a packet loss, and the sender attributes that loss to a collision. In response, the sender backs-off exponentially by doubling CW.

A flowchart explaining the essential elements of the 802.11 CSMA MAC protocol is shown in Figure 11-1.

A different approach, used in some wireless LANs before the 802.11 standard emerged, is not to use link-layer ACKs, but for a sender to infer the likelihood of a collision if it finds the channel "busy" each time it wants to transmit. In response, the sender backs-off its CW exponentially. The benefit of this approach is that it does not require ACKs, but the problem is that it turns out to be quite unfair. Nodes that successfully transmit data end up picking small CW values, while nodes that detect a busy channel back-off. 802.11 avoids this problem by *holding* the countdown timer when a sending node detects a busy channel, and by using a reactive, rather than proactive, method to back-off its CW. Thus, while the 802.11 approach may end up with more lost data, it has better fairness properties. Because the absence of a link-layer ACK causes the sender to retransmit the corresponding data at the link layer, these losses can often be shielded from higher layers.

As discussed thus far, CSMA/CA does not handle hidden or exposed terminals satisfactorily. In practice, however, carrier sense mechanisms are based on a set of heuristics that often cause a "busy" carrier to be sensed even when the other transmitting node's (or nodes') transmission is not properly decodable. That is, the "carrier sense range" is often larger than the "largest reception range" (we use these terms in quotes because they are not fixed quantities, varying in both time and space). It should be easy to see that in a very

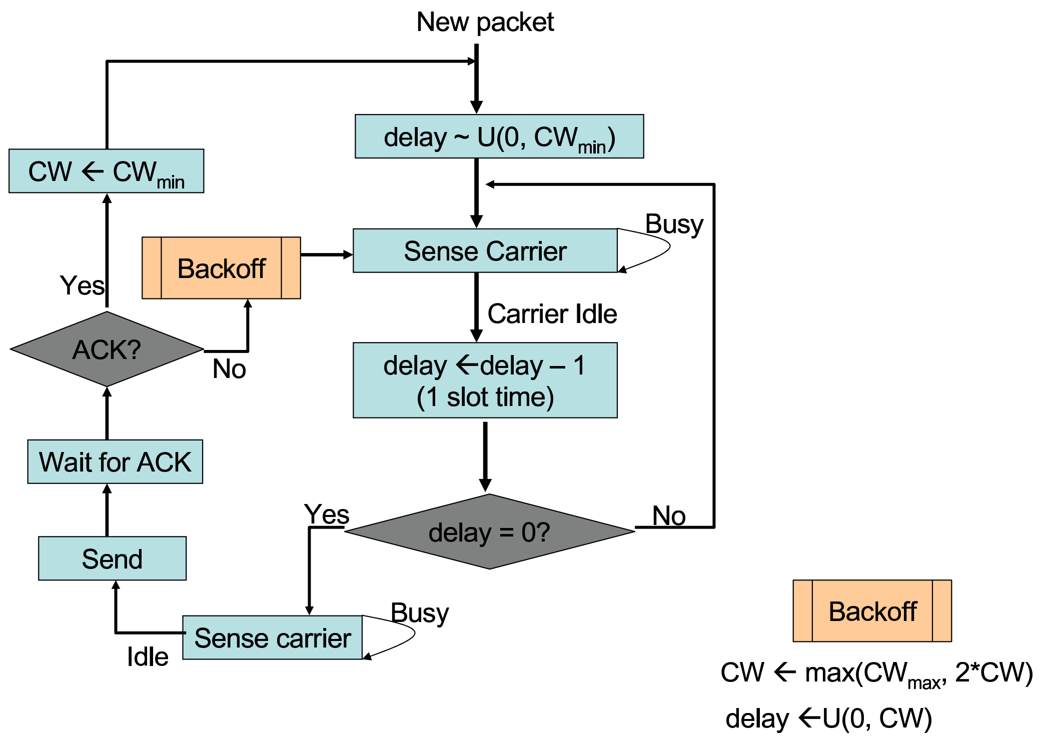


Figure 11-1: 802.11 CSMA MAC protocol.

ideal (and impractical) world of circular or spherical "ranges", a "carrier sense range" that is two or more times bigger than the "reception range" will not have any hidden terminals.

■ 11.5 Reservation-Based Protocols

A different approach to orchestrating wireless data transmissions is to use explicit reservations, as described in the MACAW paper [3]. MACAW is based on the previously proposed MACA scheme [12], which advocated that each transmission be preceded by a handshake between the sender and receiver to "reserve" the channel for a period of time. In one instantiation of such a protocol, before sending data, a node sends an RTS ("request to send") message, as long as it has not heard any CTS message in the recent past. If the intended receiver hears this message and has not heard any other RTS, then it responds with a CTS. The original RTS message includes the amount of time that the sender wishes to reserve the channel, as does the CTS (subtracting out the RTS time). A node that does not get a response to its RTS sets an exponential backoff timer, and retries the RTS after a random period of time chosen from the backoff interval.

This "RTS-CTS-Data" approach reduces the number of collisions because (1) a sender can send only if it has not heard any recent CTS (so it's own transmission can't interfere with another node's reception), and (2) a sender can send only if the receiver has not heard any RTS (so there's no other sender in the receiver's vicinity). Of course, in practice, RF data reception may not be symmetric, and other factors could conspire to corrupt packet delivery, but the approach does have merit. Moreover, it does not require any carrier sensing support.

When link-layer ACKs are used, the protocol as described thus far does not suffice, because ACKs are as important as the data packets themselves (the absence of an ACK causes a retransmission, so if packets are big enough, lost ACKs are quite wasteful). The solution to this problem is to enhance the protocol by having the sender not send an RTS *either* when it has heard another CTS *or* a RTS in the recent past, and similarly, for a receiver to not send a CTS if it has heard either an RTS or a CTS recently. Thus, all communication channels between pairs of nodes are treated as if they are bidirectional, which makes sense because we want to avoid collisions of both data and ACKs.

It is easy to see that this protocol handles hidden terminals, and because it defines all communication to be bidirectional, exposed terminal-based transmission opportunities are effectively eliminated from consideration.

The 802.11 committee standardized both CSMA/CA and RTS/CTS for 802.11 devices. In practice, both are supported, but network operators typically turn of RTS/CTS. I believe there are three reasons why:

1. RTS/CTS has very high overhead, particularly for small packets and in *rate-diverse* networks. 802.11 devices support a variety of modulations and associated transmission rates, adaptively picking a suitable scheme. RTS/CTS has to be sent at the lowest supported rate, because the goal is to avoid collisions regardless of transmission rate (because RTS/CTS bits must be decoded, a low transmission rate implies a higher probability that all nodes that are likely collide will hear the RTS or CTS). Because the data transfer itself may happen at a much higher rate than the RTS/CTS exchange, the overhead is prohibitively expensive.

2. Most current deployments of 802.11 are based on a cellular infrastructure, and are not *ad hoc*. Neighboring cells are usually configured to operate on different channels (frequencies) by explicit provisioning, so hidden terminal problems on the downlink (to the wireless LAN clients) are rare. On the uplink, hidden terminals could occur, but often these packets are small (e.g., TCP ACKs) and the RTS/CTS overhead is then significant.

3. In practice, many commercial WiFi cards can sense carrier as "busy" even when they can't decode the bits, reducing the need for explicit reservations.

■ 11.6 Time-Division Multiple Access (TDMA)

An entirely different approach to sharing wireless channels is to allocate access by time. This approach is used in some cellular telephone networks, where the base station determines a transmission time-schedule for clients. A form of TDMA is used in Bluetooth, where nodes form a subnetwork with a master and one or more slaves. The master divides time into odd and even slots, with the convention that in each odd slot, the master gets to send data to some slave device. The even slot that follows an odd one is reserved for the device that received data in the previous slot from the master. This approach is called "time division duplex" (TDD).

In general, at high loads, time division makes sense; otherwise, slots are wasted. Avoiding this waste in TDMA usually makes protocols more complex. At the same time, CSMA-based approaches don't perform too well when there is heavy, persistent load from a large number of nodes. Much work has been done in the community on MAC protocols, including on hybrid CSMA/TDMA protocols.

■ 11.7 Scaling Concerns

In general, designers of wireless networks have to consider three scaling issues that might limit their size and growth:

1. *Aggregate impact of far-away nodes.* The issue here is whether the aggregate impact of transmissions from a set of nodes that are each "far" from a particular transmitter-receiver pair can, in aggregate, cause the aggregate interference or noise to reach the point where the particular transmission may not be possible at a high enough rate. Specifically, as explained in the previous lecture, the maximum transmission rate is given by the Shannon capacity, $B \cdot (1 + \frac{S}{N})$, and the question is whether the aggregate N caused by other concurrent transmissions can cause N to become too high.

   First, observe that MAC protocols don't handle this problem, because they are concerned only with avoiding collisions in the "local" neighborhood. This question is relevant for transmitting nodes that don't necessarily detect each other's transmissions when sensing their carrier.

2. *The price of cooperation.* As nodes are added to a network, how much additional data-delivery capacity does the network gain? The question is whether the nodes in the network increasingly use their capacity to *forward* other nodes' data, leaving little for their own data.

3. *Routing protocol scalability.* As the network grows in size, how does the routing protocol scale in terms of the amount of state per node, and the rate of routing traffic required to maintain a consistent routing topology (especially if nodes move), and the ability to select good paths.

We will talk only about the first issue here.

The following model is due to Tim Shepard.[2] Consider a radio network in which the impact of any concurrent transmission on any other reception may be modeled as noise (spread spectrum systems with orthogonal codes are examples of such systems). Suppose nodes are laid out in a two-dimensional space at constant density, ρ. Suppose also that each node is interested in directly communicating with one of its closest neighbors (i.e., if a node wishes to send packets to a destination further away, a routing protocol would have to arrange for that multi-hop wireless data delivery).

It is easy to see that the distance to a nearest neighbor is proportional to $\frac{1}{\sqrt{\rho}}$. Call this quantity R_0. If each transmitter sends data at a power level P, and if the attenuation at

[2]T. Shepard, "A channel access scheme for large dense packet radio networks," Proc. ACM SIGCOMM 1996.

distance r is proportional to r^{-2} (true for free space; in other environments, it falls off faster than that, e.g., r^{-4}), then the signal strength at the receiver is propoportional to r^{-2}.

Now consider the total noise at this receiver. To calculate this quantity, we will estimate the contribution from all nodes at distancee r from the receiver, and then integrate this contribution for all values of r from R_0 (the smallest value of r) to infinity. The number of nodes in the annulus at distance r and width dr is equal to $2\pi r\rho dr$. The contribution of these nodes to the noise at a given receiver is equal to $\frac{2\pi r\rho dr}{r^2}$. Integrating that from R_0 to infinity, we find that the aggregate noise is infinite!

That is bad news. Fortunately, we don't live in an infinite world. If we assume that there are M nodes in all, we find that at node density ρ, the maximum distance of a node, R_{max}, is given by $\pi R_{max}^2\rho = M$. Solving for R_{max} and integrating the aggregate noise from R_0 to R_{max}, we find that the total signal-to-noise ratio falls off as $\frac{1}{\log M}$.

This result is good news for large-scale wireless networks, because it says that the SNR from lots of far-away concurrent transmitters falls off pretty slowly (the noise from them grows logarithmically in the number of nodes).

CHAPTER 12
Wireless Opportunistic Routing

- Radios aren't wires.
- Spatial diversity:
  - Probabilistic reception.
  - Independent reception.
  - Broadcast.
- Many ways to take advantage of spatial diversity:
  - Signal level: MIMO (multiple antennas in a single card).
  - Fragment level: Multi-radio diversity (pool together fragments of packets from the receptions at multiple access points to recover correct packets).
  - Packet level: ExOR and MORE in mesh networks.
  - Symbol level. Very recent work and perhaps the best way.
- Opportunistic routing: Traditional routing protocols pre-compute a route (either in the background or on demand). But in opportunistic routing, you decide a suitable forwarder for a packet *after* packet reception.
- Ideally, the best receiver (i.e., receiver closest to the destination) forwards the packet. But the challenge is: how to efficiently coordinate amongst the forwarders to avoid forwarding duplicates.
- Coordination between forwarders.
  - Send packets in batches to spread coordination overhead over many pakcets.
  - Link-state flooding of link success probabilities.
  - Source/upstream node figures out the list of forwarders and their priorities (based on who is closer to the destination).
- ExOR: Sequential coordination

  - Source sorts nodes in ascending order of ETX to destination.
  - Places this list in every packet header.
  - Nodes are scheduled one after the other in this order.
  - Every node hears the batch summaries of all the nodes that transmitted before itself, and sends only the packets not transmitted by the earlier nodes.

- Weaknesses of ExOR
  - Requires a global schedule.
  - Only one transmission per flow at a time.
  - Transmits the last 10% of packets using traditional routing.
  - ExOR batches do not interact very well with TCP.
  - How to do bit-rate adaptation with opportunistic routing? (It is an open problem.)

- MORE: applies the idea of "network coding" to opportunistic routing. Inspired by ExOR, but differs in the following manner:
  - Does not use a global schedule.
  - Does not impose the condition that only one node transmit at a time. Scope for spatial reuse.
  - Recovers all packets with opportunistic routing.

- Main idea of MORE: Random network coding.
  - Source transmits random independent linear combinations of the packets it has.
  - Every forwarder also transmits random independent linear combinations of the packets it receives (which are also linear combinations of the original packets).
  - The destination can solve a bunch of linear equations to recover the original packets once it has received a sufficient number of indepedent linear combinations.

- Coordination in MORE
  - Nodes no longer need to coordinate about who should send what packet.
  - Only need to worry about how much to send.
  - Two things to keep in mind when deciding how many linear combinations to send: (a) a node must send till all its downstream nodes have a good probability of receiving the packet, (b) nodes closer to the destination are preferable and hence should forward packets with higher priority.
  - After computing how many packets each node should send using above the criteria, upstream nodes dole out per-node credits to forwarders, which determine how many packets a node sends.

- Weaknesses of MORE

- Overhead of computation
- What about TCP?
- Bit-rate adaptation?

CHAPTER 13 Wireless Sensor Networks

- Sensor networks
  - Embedded computation + communication + sensing.
  - Many applications and deployments.
- Basic model and constaints.
  - Energy: want long lifetimes.
  - Computation
  - Communication: RF is lowspeed (usually).
  - Repeat the cycle of (sleep, wakeup, sense, think/compute, maybe send)
- General approach
  - Energy efficient MAC
  - Tree computation and send data along tree
- Pushing computation into the network. E.g., TAG (TinyDB). Basic idea: aggregate data along path before transmission to reduce communication costs. Computation consumes less energy than computation.
- Another problem: software updates.
  - Reliable flooding—how?
  - On the Internet or a mesh, scalability is the concern.
  - Here, energy is the concern. Same as scalability, must try to reduce number of messages sent.
- Trickle. Key idea: random suppression.
  - Pick a time t in $(0, \tau)$ to transmit at.
  - If k other nodes have transmitted same info, supress.

  - By default: send only metadata (summary/version number). If you know that someone has stale data, send code update also.
  - If you know someone has newer data, send your metadata. It acts as a request for update.

- In the absence of losses, requires a constant number of messages irrespective of number of nodes.
- In the presense of loss rate p, number of messages scales as $log_{\frac{1}{p}} n$.
- In the absence of synchronization between nodes, number of messages scales as $O(\sqrt{n})$. This problem can be fixed by having a listen period before transmission.
- Scaling limit: eventually $log_{\frac{1}{p}} n$ takes over.

References

[1] T. Bates, R. Chandra, and E. Chen. *BGP Route Reflection - An Alternative to Full Mesh IBGP*. Internet Engineering Task Force, Apr. 2000. RFC 2796. (Cited on page 46.)

[2] I. V. Beijnum. *BGP*. O'Reilly and Associates, Sept. 2002. (Cited on page 44.)

[3] V. Bharghavan, A. Demers, S. Shenker, and L. Zhang. MACAW: A Media-Access Protocol for Packet Radio. In *Proc. ACM SIGCOMM*, London, England, Aug. 1994. (Cited on page 134.)

[4] D.-M. Chiu and R. Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems*, 17:1–14, 1989. (Cited on page 71.)

[5] D. Clark and D. Tennenhouse. Architectural Consideration for a New Generation of Protocols. In *Proc. ACM SIGCOMM*, pages 200–208, Philadelphia, PA, Sept. 1990. (Cited on page 61.)

[6] R. Dube. A comparison of scaling techniques for BGP. *ACM Computer Communications Review*, 29(3):44–46, July 1999. (Cited on page 45.)

[7] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno, and Sack TCP. *ACM Computer Communications Review*, 26(3):5–21, July 1996. (Cited on page 67.)

[8] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *Proc. 2nd Symposium on Networked Systems Design and Implementation*, Boston, MA, May 2005. (Cited on page 48.)

[9] T. Griffin and G. Wilfong. On the correctness of IBGP configuration. In *Proc. ACM SIGCOMM*, Pittsburgh, PA, Aug. 2002. (Cited on pages 45 and 48.)

[10] C. Hedrick. *Routing Information Protocol*. Internet Engineering Task Force, June 1988. RFC 1058. (Cited on page 37.)

[11] V. Jacobson. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM*, pages 314–329, Vancouver, British Columbia, Canada, Sept. 1998. (Cited on pages 63 and 71.)

[12] P. Karn. MACA – A New Channel Access Method for Packet Radio. In *Proc. 9th ARRL Computer Networking Conference*, 1990. (Cited on page 134.)

[13] P. Karn and C. Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *ACM Transactions on Computer Systems*, 9(4):364–373, Nov. 1991. (Cited on pages 63 and 65.)

[14] S. Kent, C. Lynn, and K. Seo. Secure border gateway protocol (S-BGP). *IEEE JSAC*, 18(4):582–592, Apr. 2000. (Cited on page 43.)

[15] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *TCP Selective Acknowledgment Options*. Internet Engineering Task Force, 1996. RFC 2018. (Cited on page 64.)

[16] J. Moy. *OSPF Version 2*, Mar. 1994. RFC 1583. (Cited on page 37.)

[17] D. Oran. *OSI IS-IS intra-domain routing protocol*. Internet Engineering Task Force, Feb. 1990. RFC 1142. (Cited on page 37.)

[18] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *Proc. ACM SIGCOMM*, Pisa, Italy, Aug. 2006. (Cited on page 54.)

[19] Y. Rekhter and T. Li. *A Border Gateway Protocol 4 (BGP-4)*. Internet Engineering Task Force, Mar. 1995. RFC 1771. (Cited on page 36.)

[20] Y. Rekhter, T. Li, and S. Hares. *A Border Gateway Protocol 4 (BGP-4)*. Internet Engineering Task Force, Nov. 2003. Internet Draft draft-ietf-idr-bgp4-23.txt; work in progress. (Cited on page 36.)

[21] Y. Rekhter, T. Li, and S. Hares. *A Border Gateway Protocol 4 (BGP-4)*. Internet Engineering Task Force, Jan. 2006. RFC 4271. (Cited on pages 43 and 44.)

[22] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. ACM Mobicom*, pages 155–166, Boston, MA, Aug. 2000. (Not cited.)

[23] P. Traina, D. McPherson, and J. Scudder. *Autonomous System Confederations for BGP*. Internet Engineering Task Force, Feb. 2001. RFC 3065. (Cited on page 46.)