

19 TCP Reno and Congestion Management

This chapter addresses how TCP manages congestion, both for the connection's own benefit (to improve its throughput) and for the benefit of other connections as well (which may result in our connection *reducing* its own throughput). Early work on congestion culminated in 1990 with the flavor of TCP known as **TCP Reno**. The congestion-management mechanisms of TCP Reno remain the dominant approach on the Internet today, though alternative TCPs are an active area of research and we will consider a few of them in [22 Newer TCP Implementations](#).

The central TCP mechanism here is for a connection to adjust its window size. A smaller winsize means fewer packets are out in the Internet at any one time, and less traffic means less congestion. A larger winsize means better throughput, up to a point. All TCPs reduce winsize when congestion is apparent, and increase it when it is not. The trick is in figuring out when and by how much to make these winsize changes. Many of the improvements to TCP have come from mining more and more information from the stream of returning ACKs.

The Anternet

The Harvester Ant *Pogonomyrmex barbatus* uses a mechanism related to TCP Reno to “decide” how many ants should be out foraging at any one time [\[PDG12\]](#). The rate of ants leaving the nest to forage is closely tied to the rate of returning foragers; if foragers return quickly (meaning more food is available), the total number of foragers will increase (like the increasing winsize below). The ant algorithm is probabilistic, however, while most TCP algorithms are deterministic.

Recall Chiu and Jain's definition from [1.7 Congestion](#) that the “knee” of congestion occurs when the queue first starts to grow, and the “cliff” of congestion occurs when packets start being dropped. Congestion can be managed at either point, though dropped packets can be a significant waste of resources. Some newer TCP strategies attempt to take action at the congestion knee (starting with [22.6 TCP Vegas](#)), but TCP Reno is a cliff-based strategy: packets must be lost before the sender reduces the window size.

In [25 Quality of Service](#) we will consider some router-centric alternatives to TCP for Internet congestion management. However, for the most part these have not been widely adopted, and TCP is all that stands in the way of Internet congestive collapse.

The first question one might ask about TCP congestion management is just how did it get this job? A TCP sender is expected to monitor its transmission rate so as to *cooperate* with other senders to reduce overall congestion among the routers. While part of the goal of every TCP node is good, stable performance for its own connections, this emphasis on end-user cooperation introduces the prospect of “cheating”: a host might be tempted to maximize the throughput of its own connections at the expense of

others. Putting TCP nodes in charge of congestion among the core routers is to some degree like putting the foxes in charge of the henhouse. More accurately, such an arrangement has the potential to lead to the [Tragedy of the Commons](#). Multiple TCP senders share a common resource – the Internet backbone – and while the backbone is most efficient if every sender cooperates, each individual sender can improve its own situation by sending faster than allowed. Indeed, one of the arguments used by virtual-circuit routing adherents is that it provides support for the implementation of a wide range of congestion-management options under control of a central authority.

Nonetheless, TCP has been quite successful at distributed congestion management. In part this has been because system vendors do have an incentive to take the big-picture view, and in the past it has been quite difficult for individual users to replace their TCP stacks with rogue versions. Another factor contributing to TCP's success here is that most bad TCP behavior requires cooperation at the *server* end, and most server managers have an incentive to behave cooperatively. Servers generally want to distribute bandwidth fairly among their multiple clients, and – theoretically at least – a server's ISP could penalize misbehavior. So far, at least, the TCP approach has worked remarkably well.

19.1 Basics of TCP Congestion Management

TCP's congestion management is **window-based**; that is, TCP adjusts its window size to adapt to congestion. The window size can be thought of as the number of packets out there in the network; more precisely, it represents the number of packets and ACKs either in transit or enqueued. An alternative approach often used for real-time systems is **rate-based** congestion management, which runs into an unfortunate difficulty if the sending rate momentarily happens to exceed the available rate.

In the very earliest days of TCP, the window size for a TCP connection came from the `AdvertisedWindow` value suggested by the receiver, essentially representing how many packet buffers it could allocate. This value is often quite large, to accommodate large bandwidth \times delay products, and so is often reduced out of concern for congestion. When `winsize` is adjusted downwards for this reason, it is generally referred to as the **Congestion Window**, or `cwnd` (a variable name first appearing in Berkeley Unix). Strictly speaking, $\text{winsize} = \min(\text{cwnd}, \text{AdvertisedWindow})$. In newer TCP implementations, the variable `cwnd` may actually be used to mean the sender's estimate of the number of packets in flight; see the sidebar at [19.4 TCP Reno and Fast Recovery](#).

If TCP is sending over an idle network, the per-packet RTT will be $\text{RTT}_{\text{noLoad}}$, the travel time with no queuing delays. As we saw in [8.3.2 RTT Calculations](#), $(\text{RTT} - \text{RTT}_{\text{noLoad}})$ is the time each packet spends in the queue. The path bandwidth is $\text{winsize} / \text{RTT}$, and so the number of packets in queues is $\text{winsize} \times (\text{RTT} - \text{RTT}_{\text{noLoad}}) / \text{RTT}$. Usually all the queued packets are at the router at the head of the bottleneck link. Note that the sender

can calculate this number (assuming we can estimate RTT_{noLoad} ; the most common approach is to assume that the smallest RTT measured corresponds to RTT_{noLoad}).

TCP's self-clocking (*ie* that new transmissions are paced by returning ACKs) guarantees that, again assuming an otherwise idle network, the queue will build only at the bottleneck router. Self-clocking means that the rate of packet transmissions is equal to the available bandwidth of the bottleneck link. There are some spikes when a burst of packets is sent (*eg* when the sender increases its window size), but in the steady state self-clocking means that packets accumulate only at the bottleneck.

We will return to the case of the *non*-otherwise-idle network in the next chapter, in [20.2 Bottleneck Links with Competition](#).

The “optimum” window size for a TCP connection would be $bandwidth \times RTT_{noLoad}$. With this window size, the sender has exactly filled the transit capacity along the path to its destination, and has used none of the queue capacity.

Actually, TCP Reno does not do this.

Instead, TCP Reno does the following:

- guesses at a reasonable initial window size, using a form of polling
- slowly increases the window size if no losses occur, on the theory that maximum available throughput may not yet have been reached
- rapidly decreases the window size otherwise, on the theory that if losses occur then drastic action is needed

In practice, this usually leaves TCP's window size well above the theoretical “optimum”.

One interpretation of TCP's approach is that there is a time-varying “ceiling” on the number of packets the network can accept. Each sender tries to stay near but just below this level. Occasionally a sender will overshoot and a packet will be dropped somewhere, but this just teaches the sender a little more about where the network ceiling is. More formally, this ceiling represents the largest *cwnd* that does not lead to packet loss, *ie* the *cwnd* that at that particular moment completely fills but does not overflow the bottleneck queue. We have reached the ceiling when the queue is full.

In Chiu and Jain's terminology, the far side of the ceiling is the “cliff”, at which point packets are lost. TCP tries to stay above the “knee”, which is the point when the queue first begins to be persistently utilized, thus keeping the queue at least partially occupied; whenever it sends too much and falls off the “cliff”, it retreats.

The ceiling concept is often useful, but not necessarily as precise as it might sound. If we have reached the ceiling by *gradually* expanding the sliding-windows window size, then *winsize* will be as large as possible. But if the sender suddenly releases a burst of packets, the queue may fill and we will have reached a “temporary ceiling” without fully

utilizing the transit capacity. Another source of ceiling ambiguity is that the bottleneck link may be shared with other connections, in which case the ceiling represents our connection's particular share, which may fluctuate greatly with time. Finally, at the point when the ceiling is reached, the queue is *full* and so there are a considerable number of packets waiting in the queue; it is not possible for a sender to pull back instantaneously.

It is time to acknowledge the existence of different versions of TCP, each incorporating different congestion-management algorithms. The two we will start with are **TCP Tahoe** (1988) and **TCP Reno** (1990); the names Tahoe and Reno were originally the codenames of the Berkeley Unix distributions that included these respective TCP implementations. The ideas behind TCP Tahoe came from a 1988 paper by Jacobson and Karels [JK88]; TCP Reno then refined this a couple years later. TCP Reno is still in widespread use over twenty years later, and is still the undisputed TCP reference implementation, although some modest improvements (NewReno, SACK) have crept in.

A common theme to the development of improved implementations of TCP is for one end of the connection (usually the sender) to extract greater and greater amounts of information from the packet flow. For example, TCP Tahoe introduced the idea that duplicate ACKs likely mean a lost packet; TCP Reno introduced the idea that returning duplicate ACKs are associated with packets that have successfully been transmitted but follow a loss. TCP Vegas (22.6 TCP Vegas) introduced the fine-grained measurement of RTT, to detect when $RTT > RTT_{noLoad}$.

It is often helpful to think of a TCP sender as having breaks between successive windowfuls; that is, the sender sends *cwnd* packets, is briefly idle, and then sends another *cwnd* packets. The successive windowfuls of packets are often called **flights**. The existence of any separation between flights is, however, not guaranteed.

19.1.1 The Somewhat-Steady State

We will begin with the state in which TCP has established a reasonable guess for *cwnd*, comfortably below the Advertised Window Size, and which largely appears to be working. TCP then engages in some fine-tuning. This TCP “steady state” – steady here in the sense of regular oscillation – is usually referred to as the **congestion avoidance** phase, though all phases of the process are ultimately directed towards avoidance of congestion. The central strategy is that when a packet is lost, *cwnd* should decrease rapidly, but otherwise should increase “slowly”. This leads to slow oscillation of *cwnd*, which over time allows the average *cwnd* to adapt to long-term changes in the network capacity.

As TCP finishes each windowful of packets, it notes whether a loss occurred. The *cwnd*-adjustment rule introduced by TCP Tahoe and [JK88] is the following:

- if there were no losses in the previous windowful, $cwnd = cwnd + 1$
- if packets were lost, $cwnd = cwnd/2$

We are informally measuring $cwnd$ in units of full packets; strictly speaking, $cwnd$ is measured in bytes and is incremented by the maximum TCP segment size.

This strategy here is known as **Additive Increase, Multiplicative Decrease**, or AIMD; $cwnd = cwnd + 1$ is the additive increase and $cwnd = cwnd / 2$ is the multiplicative decrease. Typically, setting $cwnd = cwnd / 2$ is a medium-term goal; in fact, TCP Tahoe briefly sets $cwnd = 1$ in the immediate aftermath of an actual timeout. With no losses, TCP will send successive windowfuls of, say, 20, 21, 22, 23, 24, This amounts to conservative “probing” of the network and, in particular, of the queue at the bottleneck router. TCP tries larger $cwnd$ values because the absence of loss means the current $cwnd$ is below the “network ceiling”; that is, the queue at the bottleneck router is not yet overfull.

If a loss occurs (including multiple losses in a single windowful), TCP’s response is to cut the window size in half. (As we will see, TCP Tahoe actually handles this in a somewhat roundabout way.) Informally, the idea is that the sender needs to respond aggressively to congestion. More precisely, lost packets mean the queue of the bottleneck router has filled, and the sender needs to dial back to a level that will allow the queue to clear. If we assume that the transit capacity is roughly equal to the queue capacity (say each is equal to N), then we overflow the queue and drop packets when $cwnd = 2N$, and so $cwnd = cwnd / 2$ leaves us with $cwnd = N$, which just fills the transit capacity and leaves the queue empty. (When the sender sets $cwnd = N$, the actual number of packets in transit takes at least one RTT to fall from $2N$ to N .)

Of course, assuming any relationship between transit capacity and queue capacity is highly speculative. On a 5,000 km fiber-optic link with a bandwidth of 10 Gbps, the round-trip transit capacity would be about 60 MB, or 60,000 1 kB packets. Most routers probably do not have queues that large. Queue capacities in excess of the transit capacity are common, however. On the other hand, a competing model of a long-haul high-bandwidth TCP path is that the queue size should be a small fraction of the bandwidth \times delay product. We return to this in [19.7 TCP and Bottleneck Link Utilization](#) and [21.5.1 Bufferbloat](#).

Note that if TCP experiences a packet loss, and there is an actual timeout (as opposed to a packet loss detected by Fast Retransmit, [19.3 TCP Tahoe and Fast Retransmit](#)), then the sliding-window pipe has drained. No packets are in flight. No self-clocking can govern new transmissions. Sliding windows therefore needs to restart from scratch.

The congestion-avoidance algorithm leads to the classic “TCP sawtooth” graph, where the peaks are at the points where the slowly rising $cwnd$ crossed above the “network ceiling”. We emphasize that the TCP sawtooth is specific to TCP Reno and related TCP implementations that share Reno’s additive-increase/multiplicative-decrease mechanism.

During periods of no loss, TCP's $cwnd$ increases linearly; when a loss occurs, TCP sets $cwnd = cwnd/2$. This diagram is an idealization as when a loss occurs it takes the sender some time to discover it, perhaps as much as the Timeout interval.

The fluctuation shown here in the red ceiling curve is somewhat arbitrary. If there are only one or two other competing senders, the ceiling variation may be quite dramatic, but with many concurrent senders the variations may be smoothed out.

For some TCP sawtooth graphs created through actual simulation, see [31.2.1 Graph of \$cwnd\$ v time](#) and [31.4.1 Some TCP Reno \$cwnd\$ graphs](#).

19.1.1.1 A first look at fairness

The transit capacity of the path is more-or-less unvarying, as is the physical capacity of the queue at the bottleneck router. However, these capacities are also shared with other connections, which may come and go with time. This is why the ceiling does vary in real terms. If two other connections share a path with total capacity 60 packets, the “fairest” allocation might be for each connection to get about 20 packets as its share. If one of those other connections terminates, the two remaining ones might each rise to 30 packets. And if instead a fourth connection joins the mix, then after equilibrium is reached each connection might hope for a fair share of 15 packets.

Will this kind of “fair” allocation actually happen? Or might we end up with one connection getting 90% of the bandwidth while two others each get 5%?

Chiu and Jain [CJ89] showed that the additive-increase/multiplicative-decrease algorithm does indeed converge to roughly equal bandwidth sharing when two connections have a common bottleneck link, provided also that

- both connections have the same RTT
- during any given RTT, either both connections experience a packet loss, or neither connection does

To see this, let $cwnd1$ and $cwnd2$ be the connections' congestion-window sizes, and consider the quantity $cwnd1 - cwnd2$. For any RTT in which there is no loss, $cwnd1$ and $cwnd2$ both increment by 1, and so $cwnd1 - cwnd2$ stays the same. If there is a loss, then both are cut in half and so $cwnd1 - cwnd2$ is also cut in half. Thus, over time, the original value of $cwnd1 - cwnd2$ is repeatedly cut in half (during each RTT in which losses occur) until it dwindles to inconsequentiality, at which point $cwnd1 \approx cwnd2$.

Graphical and tabular versions of this same argument are in the next chapter, in [20.3 TCP Reno Fairness with Synchronized Losses](#).

The second bulleted hypothesis above we may call the **synchronized-loss hypothesis**. While it is very reasonable to suppose that the two connections will experience the same number of losses as a long-term *average*, it is a much stronger statement to suppose

that all loss events are shared by both connections. This behavior may not occur in real life and has been the subject of some debate; see [\[GV02\]](#). We return to this point in [31.3 Two TCP Senders Competing](#). Fortunately, equal-RTT fairness still holds if each connection is *equally likely* to experience a packet loss: both connections will have the same loss rate, and so, as we shall see in [21.2 TCP Reno loss rate versus cwnd](#), will have the same cwnd. However, convergence to fairness may take rather much longer. In [20.3 TCP Reno Fairness with Synchronized Losses](#) we also look at some alternative hypotheses for the unequal-RTT case.

19.2 Slow Start

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

The TCP Reno answer is known as **slow start**. If you are trying to guess a number in a fixed range, you are likely to use binary search. Not knowing the range for the “network ceiling”, a good strategy is to guess $cwnd=1$ (or $cwnd=2$) at first and keep doubling until you have gone too far. Then revert to the previous guess, which is known to have worked. At this point you are guaranteed to be within 50% of the true capacity.

The actual slow-start mechanism is to increment cwnd by 1 for each ACK received. This seems linear, but that is misleading: after we send a windowful of packets (cwnd many), we have received cwnd ACKs and so have incremented cwnd many times, and so have set cwnd to $(cwnd+cwnd) = 2 \times cwnd$. In other words, $cwnd=cwnd \times 2$ after each *windowful* is the same as $cwnd+=1$ after each *packet*.

Assuming packets travel together in windowfuls, all this means cwnd *doubles* each RTT during slow start; this is possibly the only place in the computer science literature where exponential growth is described as “slow”. It is indeed slower, however, than the alternative of sending an entire windowful at once.

Here is a diagram of slow start in action. This diagram makes the implicit assumption that the no-load RTT is large enough to hold well more than the 8 packets of the maximum window size shown.

For a different case, with a much smaller RTT, see [19.2.3 Slow-Start Multiple Drop Example](#).

Eventually the bottleneck queue gets full, and drops a packet. Let us suppose this is after N RTTs, so $cwnd=2^N$. Then during the previous RTT, $cwnd=2^{N-1}$ worked successfully, so we go back to that previous value by setting $cwnd = cwnd/2$.

19.2.1 TCP Reno Per-ACK Responses

During slow start, incrementing `cwnd` by one per ACK received is equivalent to doubling `cwnd` after each windowful. We can find a similar equivalence for the congestion-avoidance phase, above.

During congestion avoidance, `cwnd` is incremented by 1 after each windowful. To formulate this as a **per-ACK** increase, we spread this increment of 1 over the entire windowful, which of course has size `cwnd`. This amounts to the following upon each ACK received:

$$\text{cwnd} = \text{cwnd} + 1/\text{cwnd}$$

This is a slight approximation, because `cwnd` keeps changing, but it works well in practice. Because TCP actually measures `cwnd` in bytes, floating-point arithmetic is normally not required; see exercise 14.0. An exact equivalent to the per-windowful incrementing strategy is $\text{cwnd} = \text{cwnd} + 1/\text{cwnd}_0$, where cwnd_0 is the value of `cwnd` at the start of that particular windowful. Another, simpler, approach is to use $\text{cwnd} += 1/\text{cwnd}$, and to keep the fractional part recorded, but to use $\text{floor}(\text{cwnd})$ (the integer part of `cwnd`) when actually sending packets.

Most actual implementations keep track of `cwnd` in bytes, in which case using integer arithmetic is sufficient until `cwnd` becomes quite large.

If **delayed ACKs** are implemented ([18.8 TCP Delayed ACKs](#)), then in bulk transfers one arriving ACK actually acknowledges two packets. [RFC 3465](#) permits a TCP receiver to increment `cwnd` by $2/\text{cwnd}$ in that situation, which is the response consistent with incrementing `cwnd` by 1 upon receipt of enough ACKs to acknowledge an entire windowful.

19.2.2 Threshold Slow Start

Sometimes TCP uses slow start even when it knows the working network capacity. After a packet loss and timeout, TCP knows that a new `cwnd` of $\text{cwnd}_{\text{old}}/2$ should work.

If `cwnd` had been 100, TCP halves it to 50. The problem, however, is that after timeout there are no returning ACKs to self-clock the continuing transmission, and we do not want to dump 50 packets on the network all at once. So in restarting the flow TCP uses what might be called **threshold slow start**: it uses slow-start, but stops when `cwnd` reaches the target. Specifically, on packet loss we set the variable `ssthresh` to $\text{cwnd}/2$; this is our new target for `cwnd`. We set `cwnd` itself to 1, and switch to the slow-start mode ($\text{cwnd} += 1$ for each ACK). However, as soon as `cwnd` reaches `ssthresh`, we switch to the congestion-avoidance mode ($\text{cwnd} += 1/\text{cwnd}$ for each ACK). Note that the transition from threshold slow start to congestion avoidance is completely natural, and easy to implement.

TCP will use threshold slow-start whenever it is restarting from a pipe drain; that is, every time slow-start is needed after its very first use. (If a connection has simply been *idle*, non-threshold slow start is typically used when traffic starts up again.)

Threshold slow-start can be seen as an attempt at combining rapid window expansion with self-clocking.

By comparison, we might refer to the initial, non-threshold slow start as **unbounded slow start**. Note that unbounded slow start serves a fundamentally different purpose – initial probing to determine the network ceiling to within 50% – than threshold slow start.

Here is the TCP sawtooth diagram above, modified to show timeouts and slow start. The first two packet losses are displayed as “coarse timeouts”; the rest are displayed as if Fast Retransmit, below, were used.

RFC 2581 allows slow start to begin with $cwnd=2$.

19.2.3 Slow-Start Multiple Drop Example

Slow start has the potential to cause multiple dropped packets at the bottleneck link; packet losses continue for quite some time because the TCP sender is slow to discover them. The network topology is as follows, where the A-R link is infinitely fast and the R-B link has a bandwidth in the R→B direction of 1 packet/ms.

Assume that R has a queue capacity of 100, not including the packet it is currently forwarding to B, and that ACKs travel instantly from B back to A. In this and later examples we will continue to use the Data[N]/ACK[N] terminology of [8.2 Sliding Windows](#), beginning with N=1; TCP numbering is not done quite this way but the distinction is inconsequential.

When A uses slow-start here, the successive windowfuls will almost immediately begin to overlap. A will send one packet at T=0; it will be delivered at T=1. The ACK will travel instantly to A, at which point A will send two packets. From this point on, ACKs will arrive regularly at A at a rate of one per second. Here is a brief chart:

Time	A receives	A sends	R sends	R's queue
0		Data[1]	Data[1]	
1	ACK[1]	Data[2],Data[3]	Data[2]	Data[3]
2	ACK[2]	4,5	3	4,5
3	ACK[3]	6,7	4	5..7
4	ACK[4]	8,9	5	6..9
5	ACK[5]	10,11	6	7..11
..				
N	ACK[N]	2N,2N+1	N+1	N+2 .. 2N+1

At T=N, R's queue contains N packets. At T=100, R's queue is full. Data[200], sent at T=100, will be delivered and acknowledged at T=200, giving it an RTT of 100. At

$T=101$, R receives Data[202] and Data[203] and drops the latter one. Unfortunately, A's timeout interval must of course be greater than the RTT, and so A will not detect the loss until, at an absolute minimum, $T=200$. At that point, A has sent packets up through Data[401], and the 100 packets Data[203], Data[205], ..., Data[401] have all been lost. In other words, at the point when A *first* receives the news of one lost packet, in fact at least 100 packets have already been lost.

Fortunately, unbounded slow start generally occurs only once per connection.

19.2.4 Summary of TCP so far

So far we have the following features:

- Unbounded slow start at the beginning
- Congestion avoidance with AIMD once some semblance of a steady state is reached
- Threshold slow start after each loss
- Each threshold slow start transitioning naturally to congestion avoidance

Here is a table expressing the slow-start and congestion-avoidance phases in terms of manipulating `cwnd`.

phase	cwnd change, loss	cwnd change, no loss	
	per window	per window	per ACK
slow start	$\text{cwnd}/2$	$\text{cwnd} *= 2$	$\text{cwnd} += 1$
cong avoid	$\text{cwnd}/2$	$\text{cwnd} += 1$	$\text{cwnd} += 1/\text{cwnd}$

Viewing `cwnd`

Linux users can view the current values of `cwnd` and `ssthresh`, as well as a host of other TCP statistics, using the command `ss --tcp --info`.

The problem TCP often faces, in both slow-start and congestion-avoidance phases, is that when a packet is lost the sender will not detect this until much later (at least until the bottleneck router's current queue has been sent); by then, it may be too late to avoid further losses.

19.2.5 The initial value of `cwnd`

So far we have been assuming that slow start begins with `cwnd` equal to one packet, the value proposed in [JK88]. But 1988 was quite a while ago, and the initial `cwnd` has been creeping up. A decade later, RFC 2414 proposed setting the initial `cwnd` (used on startup and after a timeout) to between two and four packets, with a maximum size of 3×1460 (three packets if the maximum Ethernet packet size of 1500 bytes is used). For a while following, the most common value was two packets. Then RFC 6928, in 2013, proposed an "experimental" change to an initial value of up to ten packets, where each can be 1460 bytes. This ten-packet value is now in relatively common use.

This increase in the initial `cwnd` value has mostly to do with the steadily increasing capacity of the Internet, and the decreased likelihood that a single extra packet will make a material difference. Still, the basic behavior of slow start remains the same.

Non-Reno TCPs also implement something like slow start, though it may look slightly different. TCP BBR ([22.16 TCP BBR](#)), for example, has a STARTUP mode that serves the same function as TCP Reno slow start.

19.3 TCP Tahoe and Fast Retransmit

TCP Tahoe has one more important feature. Recall that TCP ACKs are cumulative; if packets 1 and 2 have been received and now Data[4] arrives, but not yet Data[3], all the receiver can (and must!) do is to send back another ACK[2]. Thus, from the sender's perspective, if we send packets 1,2,3,4,5,6 and get back ACK[1], ACK[2], ACK[2], ACK[2], ACK[2], we can infer two things:

- Data[3] got lost, which is why we are stuck on ACK[2]
- Data 4,5 and 6 probably *did* make it through, and triggered the three duplicate ACK[2]s (the three ACK[2]s following the first ACK[2]).

The **Fast Retransmit** strategy is to resend Data[N] when we have received three dupACKs for Data[N-1]; that is, four ACK[N-1]'s in all. Because this represents a packet loss, we also set `ssthresh = cwnd/2`, set `cwnd=1`, and begin the threshold-slow-start phase. The effect of this is typically to reduce the delay associated with the lost packet from that of a full timeout, typically $2 \times \text{RTT}$, to just a little over a single RTT. The lost packet is now discovered *before* the TCP pipeline has drained. However, at the end of the next RTT, when the ACK of the retransmitted packet will return, the TCP pipeline *will* have drained, hence the need for slow start.

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

Fast Retransmit waits for the *third* dupACK to allow for the possibility of moderate packet reordering. Suppose packets 1 through 6 are sent, but they arrive in the order 1,3,4,2,6,5, perhaps due to a router along the way with an architecture that is strongly parallelized. Then the ACKs that would be sent back would be as follows:

Received	Response
Data[1]	ACK[1]
Data[3]	ACK[1]
Data[4]	ACK[1]
Data[2]	ACK[4]
Data[6]	ACK[4]
Data[5]	ACK[6]

Waiting for the third dupACK is in most cases a successful compromise between responsiveness to lost packets and reasonable evidence that the data packet in question is actually lost.

However, a router that does more substantial delivery reordering would wreck havoc on connections using Fast Retransmit. In particular, consider the router R in the diagram below; when sending packets to B it might in principle wish to alternate on a packet-by-packet basis between the path via R1 and the path via R2. This would be a mistake; if the R1 and R2 paths had different propagation delays then this strategy would introduce major packet reordering. R should send all the packets belonging to any one TCP connection via a single path.

In the real world, routers generally go to considerable lengths to accommodate Fast Retransmit; in particular, use of multiple paths for a single TCP connection is almost universally frowned upon. Some actual data on packet reordering can be found in [\[VP97\]](#); the author suggests that a switch to retransmission on the second dupACK would be risky.

19.4 TCP Reno and Fast Recovery

Fast Retransmit requires a sender to set $cwnd=1$ because the pipe has drained and there are no arriving ACKs to pace transmission. Fast Recovery is a technique that often allows the sender to avoid draining the pipe, and to move from $cwnd$ to $cwnd/2$ in the space of a single RTT. TCP Reno is TCP Tahoe with the addition of Fast Recovery.

The idea is to use the arriving dupACKs to pace retransmission. We make the assumption that each arriving dupACK indicates that *some* data packet following the lost packet has been delivered successfully; it turns out not to matter which one. On discovery of the lost packet through Fast Retransmit, the goal is to set $cwnd=cwnd/2$; the next step is to figure out how many dupACKs we have to wait for before we can resume transmissions of new data.

Initially, at least, we assume that only one data packet is lost, though in the following section we will see that multiple losses can be handled via a slight modification of the Fast Recovery strategy.

During the recovery process, there is a problem with the direct use of sliding windows: the lost packet “pins” the lower end of the window until that packet is successfully retransmitted, so for the duration of the recovery period the window cannot slide forward. The official specification of fast recovery, originally in [RFC 2001](#) and now in [RFC 5681](#), describes retransmission in terms of **$cwnd$ inflation** and **deflation**. If C is the value of $cwnd$ at the time of the loss, then $cwnd$ is steadily inflated to $1.5 \times C$, allowing for the original window plus half a windowful more of new data. At the point the lost packet is

successfully retransmitted, the window is “deflated” to $cwnd = C/2$; at this point the lower end of the window slides forward by C , and the upper end of the window does not move. This all works out, but can be a bit hard to follow.

Instead we will use the concept of Estimated FlightSize, or **EFS**, which is the sender’s best guess at the number of outstanding packets. Under normal circumstances, EFS is the same as $cwnd$. The crucial Fast Recovery observation is that EFS should be decremented by 1 for each arriving dupACK, because the arrival of each dupACK means one less packet is in transit, and so transmission of new packets can resume when EFS is reduced to half of the original value of $cwnd$. Our EFS approach is equivalent to the inflationary approach at least when slow start is not involved.

Linux $cwnd$ is Estimated FlightSize

This chapter defines $cwnd$ to be the sender winsize strictly construed. As such, packet $N+cwnd$ cannot be sent until packet N is ACKed. However, the Linux kernel actually uses $cwnd$ as a synonym for Estimated FlightSize, which simplifies the Fast-Recovery code. This usage applies, in fact, to all TCP varieties, though it often makes little difference. See [tcp_cwnd_test\(\)](#).

We first outline the general case, and then look at a specific example. Let $cwnd = N$, and suppose packet 1 is lost (packet numbers here may be taken as relative). Until packet 1 is retransmitted, the sender can only send up through packet N (Data[N] can be sent only after ACK[0] has arrived at the sender). The receiver will send $N-1$ dupACK[0]s representing packets 2 through N .

At the point of the third dupACK, when the loss of Data[1] is discovered, the sender calculates as follows: EFS had been $cwnd = N$. Three dupACKs have arrived, representing three later packets no longer in flight, so EFS is now $N-3$. At this point the sender realizes a packet has been lost, which makes $EFS = N-4$ briefly, but that packet is then immediately retransmitted, which brings EFS back to $N-3$.

The sender expects at this point to receive $N-4$ more dupACKs, followed by one new ACK for the retransmission of the packet that was lost. This last ACK will be for the entire original windowful.

The new target for $cwnd$ is $N/2$ (for simplicity, we will assume N is even). So, we wait for $N/2 - 3$ more dupACKs to arrive, at which point EFS is $N-3-(N/2-3) = N/2$. *After* this point the sender will resume sending new packets; it will send one new packet for each of the $N/2-1$ subsequently arriving dupACKs (recall that there are $N-1$ dupACKs in all). These new transmissions will be Data[$N+1$] through Data[$N+(N/2-1)$].

After the last of the dupACKs will come the ACK corresponding to the retransmission of the lost packet; it will be ACK[N], acknowledging all of the original windowful. At this point, there are $N/2 - 1$ unacknowledged packets Data[$N+1$] through Data[$N+(N/2)-1$]. The sender now sends Data[$N+N/2$] and is thereby able to resume sliding windows

with $cwnd = N/2$: the sender has received $ACK[N]$ and has exactly one full windowful outstanding for the new value $N/2$ of $cwnd$. That is, we are right where we are supposed to be.

Here is a diagram illustrating Fast Recovery for $cwnd=10$. Data[10] is lost.

Data[9] elicits the initial $ACK[9]$, and the nine packets Data[11] through Data[19] each elicit a $dupACK[9]$. We denote the $dupACK[9]$ elicited by Data[N] by $dupACK[9]/N$; these are shown along the upper right. Unless SACK TCP (below) is used, the sender will have no way to determine N or to tell these $dupACK$ s apart. When $dupACK[9]/13$ (the third $dupACK$) arrives at the sender, the sender uses Fast Recovery to infer that Data[10] was lost and retransmits it. At this point $EFS = 7$: the sender has sent the original batch of 10 data packets, plus Data[19], and received one ACK and three $dupACK$ s, for a total of $10+1-1-3 = 7$. The sender has also inferred that Data[10] is lost ($EFS -= 1$) but then retransmitted it ($EFS += 1$). Six more $dupACK[9]$'s are on the way.

EFS is decremented for each subsequent $dupACK$ arrival; after we get two more $dupACK[9]$'s, EFS is 5. The next $dupACK[9]$ ($dupACK[9]/16$) reduces EFS to 4 and so allows us transmit Data[20] (which promptly bumps EFS back up to 5). We have

receive	send
$dupACK[9]/16$	Data[20]
$dupACK[9]/17$	Data[21]
$dupACK[9]/18$	Data[22]
$dupACK[9]/19$	Data[23]

We emphasize again that the TCP sender does not see the numbers 16 through 19 in the receive column above; it determines when to begin transmission by counting $dupACK[9]$ arrivals.

Working Backwards

Figuring out when a fast-recovery sender should resume transmissions of new data is error-prone. Perhaps the simplest approach is to work backwards from the retransmitted lost packet: it should trigger at the receiver an ACK for the entire original windowful. When Data[10] above was lost, the “stuck” window was Data[10]–Data[19]. The retransmitted Data[10] thus triggers $ACK[19]$; when $ACK[19]$ arrives, $cwnd$ should be $10/2 = 5$ so Data[24] should be sent. That in turn means the four packets Data[20] through Data[23] must have been sent earlier, via Fast Recovery. There are $10-1 = 9$ $dupACK$ s, so to send on the last four we must start with the sixth. The diagram above indeed shows new Fast Recovery transmissions beginning with the sixth $dupACK$.

The next thing to arrive at the sender side is the $ACK[19]$ elicited by the retransmitted Data[10]; at the point Data[10] arrives at the receiver, Data[11] through Data[19] have

already arrived and so the cumulative-ACK response is ACK[19]. The sender responds to ACK[19] with Data[24], and the transition to $cwnd=5$ is now complete.

During sliding windows without losses, a sender will send $cwnd$ packets per RTT. If a “coarse” timeout occurs, typically it is not discovered until after at least one complete RTT of link idleness; there are additional underutilized RTTs during the slow-start phase. It is worth examining the Fast Recovery sequence shown in the illustration from the perspective of underutilized bandwidth. The diagram shows three round-trip times, as seen from the sender side. During the first RTT, the ten packets Data[9]–Data[18] are sent. The second RTT begins with the sending of Data[19] and continues through sending Data[22], along with the retransmitted Data[10]. The third RTT begins with sending Data[23], and includes through Data[27]. In terms of recovery efficiency, the RTTs send 9, 5 and 5 packets respectively (we have counted Data[10] twice); this is remarkably close to the ideal of reducing $cwnd$ to 5 instantaneously.

Using the fast-recovery description in terms of inflation from [RFC 5681](#), inflation would begin at the point the sender resumed transmitting new packets, at which point $cwnd$ would be incremented for each dupACK. In the diagram above, at the instant labeled “send Data[24]” on the sender side, $cwnd$ would momentarily become 15, representing the window Data[10]..Data[24]. As soon as the sender realized that the lost packet Data[10] had been acknowledged, via ACK[19], $cwnd$ would immediately deflate to 5, representing the window Data[20]..Data[24]. For a diagram illustrating $cwnd$ inflation and deflation, see [32.2.1 Running the Script](#).

There is one more addition to Fast Recovery, described in [RFC 3042](#), and known as **Limited Transmit**. As described above, transmission of new data begins with the third dupACK, which is the point at which Fast Recovery is initiated. Limited Transmit means that one new data packet is also transmitted for each of the first two dupACKs, on a “just in case” basis. These two transmissions are “borrowed” against future forward motion of the send window; the value of $cwnd$ is not changed and the first “skipped” packet is not retransmitted. If there was no packet loss, and the dupACKs simply resulted from packet reordering, no harm is done. Otherwise, Fast Retransmit gets a slightly earlier start. For small window sizes this can make a significant difference.

19.5 TCP NewReno

TCP NewReno, described in [\[JH96\]](#) and [RFC 2582](#) (currently [RFC 6582](#)), is a modest tweak to Fast Recovery which greatly improves handling of the case when two or more packets are lost in a windowful. It is generally considered to be a part of contemporary TCP Reno. We again describe it in terms of Estimated FlightSize rather than in terms of $cwnd$ inflation and deflation.

If two data packets are lost and the first is retransmitted, the receiver will acknowledge data up to just before the second packet, and then continue sending dupACKs of this until the second lost packet is also retransmitted. These ACKs of data up to just before

the second packet are sometimes called **partial ACKs**, because retransmission of the first lost packet did not result in an ACK of all the outstanding data. The NewReno mechanism uses these partial ACKs as evidence to retransmit later lost packets, and also to keep pacing the Fast Recovery process.

In the diagram below, packets 1 and 4 are lost in a window 0..11 of size 12. Initially the sender will get dupACK[0]'s; the first 11 ACKs (dashed lines from right to left) are ACK[0] and 10 dupACK[0]'s. When packet 1 is successfully retransmitted on receipt of the third dupACK[0], the receiver's response will be ACK[3] (the heavy dashed line). This is the first partial ACK (a full ACK would have been ACK[12]). On receipt of any partial ACK during the Fast Recovery process, TCP NewReno assumes that the immediately following data packet was lost and retransmits it immediately; the sender does not wait for three dupACKs because if the following data packet had not been lost, no instances of the partial ACK would ever have been generated, even if packet reordering had occurred.

The TCP NewReno sender response here is, in effect, to treat each partial ACK as a dupACK[0], except that the sender *also* retransmits the data packet that – based upon receipt of the partial ACK – it is able to infer is lost. NewReno continues pacing Fast Recovery by whatever ACKs arrive, whether they are the original dupACKs or later partial ACKs or dupACKs.

When the receiver's first ACK[3] arrives at the sender, NewReno infers that Data[4] was lost and resends it; this is the second heavy data line. No dupACK[3]'s need arrive; as mentioned above, the sender can infer from the single ACK[3] that Data[4] is lost. The sender also responds as if another dupACK[0] had arrived, and sends Data[17].

The arrival of ACK[3] signals a reduction in the EFS by 2: one for the inference that Data[4] was lost, and one as if another dupACK[0] had arrived; the two transmissions in response (of Data[4] and Data[17]) bring EFS back to where it was. At the point when Data[16] is sent the actual (not estimated) flightsize is 5, not 6, because there is one less dupACK[0] due to the loss of Data[4]. However, once NewReno resends Data[4] and then sends Data[17], the actual flightsize is back up to 6.

There are four more dupACK[3]'s that arrive. NewReno keeps sending new data on receipt of each of these; these are Data[18] through Data[21].

The receiver's response to the retransmitted Data[4] is to send ACK[16]; this is the cumulative of all the data received up to that moment. At the point this ACK arrives back at the sender, it had just sent Data[21] in response to the fourth dupACK[3]; its response to ACK[16] is to send the next data packet, Data[22]. *The sender is now back to normal sliding windows*, with a cwnd of 6. Similarly, the Data[17] immediately following the retransmitted Data[4] elicits an ACK[17] (this is the first Data[N] to elicit an exactly matching ACK[N] since the losses began), and the corresponding response to the ACK[17] is to continue with Data[23].

As with the previous Fast Recovery example, we consider the number of packets sent per RTT; the diagram shows four RTTs as seen from the sender side.

RTT	First packet	Packets sent	count
first	Data[0]	Data[0]-Data[11]	12
second	Data[12]	Data[12]-Data[15], Data[1]	5
third	Data[16]	Data[16]-Data[20], Data[4]	6
fourth	Data[21]	Data[21]-Data[26]	6

Again, after the loss is detected we segue to the new cwnd of 6 with only a single missed packet (in the second RTT). NewReno is, however, only able to send one retransmitted packet per RTT.

Note that TCP Newreno, like TCPs Tahoe and Reno, is a **sender-side** innovation; the receiver does not have to do anything special. The next TCP flavor, SACK TCP, requires receiver-side modification.

19.6 Selective Acknowledgments (SACK)

A traditional TCP ACK is a cumulative acknowledgment of all data received up to that point. If Data[1002] is received but not Data[1001], then all the receiver can send is a duplicate ACK[1000]. This does indicate that *something* following Data[1001] made it through, but nothing more.

To provide greater specificity, TCP now provides a **Selective ACK** (SACK) option, implemented at the receiver. If this is available, the sender does not have to guess from dupACKs what has gotten through. The receiver can send an ACK that says:

- All packets up through 1000 have been received (the cumulative ACK)
- All packets up through 1050 have been received *except for* 1001, 1022, and 1035.

The second line is the SACK part. Almost all TCP implementations now support this.

Specifically, SACKs include the following information; the additional data beyond the cumulative ACK is included in a TCP Option field.

- The latest cumulative ACK
- The *three* most recent blocks of consecutive packets received

Thus, if we have lost 1001, 1022, 1035, and now 1051, and the highest received is 1060, the SACK might say:

- All packets up through 1000 have been received
- 1060-1052 have been received
- 1050-1036 have been received

- 1034–1023 have been received

From this the sender knows 1001s and 1022 were not received, but nothing about the packets in between. However, if the sender has been paying close attention to the previous SACKs received, it likely already knows that all packets 1002 through 1021 have been received.

The term **SACK TCP** is typically used to mean that the receiving side supports selective ACKs, and the sending side is a straightforward modification of TCP Reno to take advantage of them.

In practice, selective ACKs provide at best a modest performance improvement in many situations; TCP NewReno does rather well, in moderate-loss environments. The paper [\[FF96\]](#) compares Tahoe, Reno, NewReno and SACK TCP, in situations involving from one to four packet losses in a single RTT. While Classic Reno performed poorly with two packet losses in a single RTT and extremely poorly with three losses, the three-loss NewReno and SACK TCP scenarios played out remarkably similarly. Only when connections experienced four losses in a single RTT did SACK TCP's performance start to pull slightly ahead of that of NewReno.

19.7 TCP and Bottleneck Link Utilization

Consider a TCP Reno sender with no competing traffic. As *cwnd* saws up and down, what happens to throughput? Do those halvings of *cwnd* result in at least a dip in throughput? The answer depends to some extent on the size of the queue ahead of the bottleneck link, relative to the transit capacity of the path. As was discussed in [8.3.2 RTT Calculations](#), when *cwnd* is less than the transit capacity, the link is less than 100% utilized and the queue is empty. When *cwnd* is more than the transit capacity, the link is saturated (100% utilized) and the queue has about (*cwnd* – *transit_capacity*) packets in it.

The diagram below shows two TCP Reno teeth; in the first, the queue capacity exceeds the path transit capacity and in the second the queue capacity is a much smaller fraction of the total.

In the first diagram, the bottleneck link is always 100% utilized, even at the left edge of the teeth. In the second the interval between loss events (the left and right margins of the tooth) is divided into a **link-unsaturated** phase and a **queue-filling phase**. In the unsaturated phase, the bottleneck link utilization is less than 100% and the queue is empty; in the later phase, the link is saturated and the queue begins to fill.

Consider again the idealized network below, with an R-B bandwidth of 1 packet/ms.

We first consider the $\text{queue} \geq \text{transit}$ case. Assume that the total $\text{RTT}_{\text{noLoad}}$ delay is 100 ms, mostly due to propagation delay; this makes the $\text{bandwidth} \times \text{delay}$ product 100 packets. The question for consideration is to what extent TCP Reno, once slow-start is over, sometimes leaves the R-B link idle.

The R-B link will be saturated at all times provided A always keeps 100 packets in transit, that is, we always have $\text{cwnd} \geq 100$ (8.3.2 RTT Calculations). If $\text{cwnd}_{\min} = 100$, then $\text{cwnd}_{\max} = 2 \times \text{cwnd}_{\min} = 200$. For this to be the maximum, the queue capacity must be at least 99, so that the path can accommodate 199 packets without loss: 100 packets in transit plus 99 packets in the queue. In general, TCP Reno never leaves the bottleneck link idle as long as the queue capacity in front of that link is at least as large as the path round-trip transit capacity.

Now suppose instead that the queue size is 49, or about 50% of the transit capacity. Packet loss will occur when cwnd reaches 150, and so $\text{cwnd}_{\min} = 75$. Qualitatively this case is represented by the second diagram above, though the queue-to-network_ceiling proportion illustrated there is more like 1:8 than 1:3. There are now periods when the R-B link is idle. During RTT intervals when $\text{cwnd} = 75$, throughput will be 75% of the maximum and the R-B link will be idle 25% of the time.

However, cwnd will be 75 just for the first RTT following the loss. After 25 RTTs, cwnd will be back up to 100, and the link will be saturated. So we have 25 RTTs with an average cwnd of $87.5 = (75 + 100)/2$, meaning the link is 87.5% saturated, followed by 50 RTTs where the link is 100% saturated. The long-term average here is 95.8% utilization of the bottleneck link. This is not bad at all, given that using 10% of the link bandwidth on packet headers is almost universally considered reasonable. Furthermore, at the point when cwnd drops after a loss to $\text{cwnd}_{\min} = 75$, the queue must have been full. It may take one or two RTTs for the queue to drain; during this time, link utilization will be even higher.

If most or all of the time the bottleneck link is saturated, as in the first diagram, it may help to consider the average queue size. Let the queue capacity be C_{queue} and the transit capacity be C_{transit} , with $C_{\text{queue}} > C_{\text{transit}}$. Then cwnd will vary from a maximum of $C_{\text{queue}} + C_{\text{transit}}$ to a minimum of what works out to be $(C_{\text{queue}} - C_{\text{transit}})/2 + C_{\text{transit}}$. We would expect an average queue size about halfway between these, less the C_{transit} term: $3/4 \times C_{\text{queue}} - 1/4 \times C_{\text{transit}}$. If $C_{\text{queue}} = C_{\text{transit}}$, the expected average queue size should be about $C_{\text{queue}}/2$.

See exercises 12.0 and 12.5.

19.7.1 TCP Queue Sizes

From the perspective of link utilization, the previous section suggests that router queues be larger rather than smaller. A queue capacity at least as large as transit capacity seems like an excellent choice. To configure a router this way, we first make an educated guess

at the average RTT, and then multiply this by the output bandwidth to get the desired queue capacity. For an average RTT of 50 ms, a bandwidth of 1 Gbps leads to a queue capacity of about 6 MB, or 4000 packets of 1500 bytes each. If the numbers rise to 100 ms and 10 Gbps, queue capacity needs to be 125 MB.

Unfortunately, while large queues are helpful when the traffic consists exclusively of bulk TCP transfers, they introduce proportionately large queuing delays that can wreak havoc on real-time traffic. A bottleneck router with a queue size matching a flow's $\text{bandwidth} \times \text{delay}$ product will *double* the RTT for that flow, at points when the queue is full. Worse, if the goal is 100% TCP link utilization always, then the router queue must be sized for the highest-bandwidth flow with the longest RTT; shorter TCP connections will encounter a queue much larger than necessary. This problem of large queue capacity leading to excessive delay is known as **bufferbloat**; we will return to it at [21.5.1 Bufferbloat](#).

Because of the delay problems brought on by large queues, TCP connections must sometimes pass through bottleneck routers with small queues. In this case a tooth of a TCP Reno connection is divided into a large link-unsaturated phase and a small queue-filling phase.

The *need* for large buffers, if near-100% queue utilization is the goal, is to a large degree specific to the TCP Reno sawtooth. Some other TCP implementations (in particular TCP Vegas, [22.6 TCP Vegas](#)), do not overfill the queue. However, TCP Vegas does not compete well with TCP Reno, at least with traditional FIFO queuing ([20.1 A First Look At Queuing](#)) (but see [23.6.1 Fair Queuing and Bufferbloat](#)).

The worst case for TCP link utilization is if the queue size is close to zero. Using again a $\text{bandwidth} \times \text{delay}$ product 100 of packets, a zero-sized queue will mean that cwnd_{\max} will be 100 (or 101), and so cwnd_{\min} will be 50. Link utilization therefore ranges, over the lifetime of the tooth, from a low of $50/100 = 50\%$ to a high of 100%; the average utilization is **75%**. While this is not ideal, and while some non-Reno TCP variants have attempted to improve this figure, 75% link utilization is not all that bad, and can be compared with the 10% of the bandwidth consumed as packet headers (though that figure assumes 512 bytes of data per packet, which is low). (A literally zero-sized queue will not work at all well; one reason – though not the only one – is that TCP Reno sends a two-packet burst whenever cwnd is incremented.)

Traffic mix has a major influence on the appropriate queue size. For example, the analysis of the previous section assumed a single long-term TCP connection. The link-utilization situation improves with increasing numbers of TCP connections, at least if the losses are unsynchronized, because the halving of one connection's cwnd has a proportionately smaller impact on the total queue use. In [\[AKM04\]](#) it is shown that for a router with N TCP connections with unsynchronized losses, a queue size of $(\text{RTT}_{\text{average}} \times \text{bandwidth}) / \sqrt{N}$ is sufficient to keep the link almost always saturated. Larger values of N here are typically associated with “core” (backbone) routers. The

paper [\[EGMR05\]](#) proposes even smaller buffer capacities, on the order of the logarithm of the maximum window size. The argument makes two important assumptions, however: first, that we are willing to tolerate a link utilization somewhat less than 100% (though greater than 75%), and second, perhaps more importantly, that TCP is modified so as to spread out any packet bursts – even bursts of size two – over small intervals of time.

There are other problems created by too-small queues, even if we are willing to accept 75% link utilization. Internet traffic, not unlike city-bus traffic, tends to “bunch up”; queues serve as a way to keep these packet bunches from leading to unnecessary losses. For one example of unexpected traffic bunching, see [31.4.1.3 Transient queue peaks](#). Increased traffic randomization helps reduce the need for very large queues, but may increase the bunching effect. Internet “core” routers see more highly randomized traffic than end-user or “edge” routers; queues in the latter are often the most difficult to configure.

We will return to the issue of link utilization in [31.2.6 Single-sender Throughput Experiments](#) and (for two senders) [31.3.10.2 Higher bandwidth and link utilization](#), using the ns simulator to get experimental data. See also exercise 12.0.

Finally, the queue capacity does not necessarily have to remain static. We will return to this point in [21.5 Active Queue Management](#). Furthermore, many queue-size problems ultimately spring from the fact that all traffic is being dumped into a single FIFO queue; we will look at alternative queuing strategies in [23 Queuing and Scheduling](#). For a particular example related to bufferbloat, see [23.6.1 Fair Queuing and Bufferbloat](#).

19.8 Single Packet Losses

Again assuming no competition on the bottleneck link, the TCP Reno additive-increase policy has a simple consequence: at the end of each tooth, only a single packet will be lost.

To see this, let A be the sender, R be the bottleneck router, and B be the receiver:

Let T be the bandwidth delay at R, so that packets leaving R are spaced at least time T apart. A will therefore transmit packets T time units apart, except for those times when $cwnd$ has just been incremented and A sends a pair of packets back-to-back. Let us call the second packet of such a back-to-back pair the “extra” packet. To simplify the argument slightly, we will assume that the two packets of a pair arrive at R essentially simultaneously.

Only an extra packet can result in an increase in queue utilization; every other packet arrives after an interval T from the previous packet, giving R enough time to remove a packet from its queue.

A consequence of this is that $cwnd$ will reach the sum of the transit capacity and the queue capacity without R dropping a packet. (This is not necessarily the case if a $cwnd$ this large were sent as a single burst.)

Let C be this combined capacity, and assume $cwnd$ has reached C . When A executes its next $cwnd += 1$ additive increase, it will as usual send a pair of back-to-back packets. The second of this pair – the extra – is doomed; it will be dropped when it reaches the bottleneck router.

At this point there are $C = cwnd - 1$ packets outstanding, all spaced at time intervals of T . Sliding windows will continue normally until the ACK of the packet just before the lost packet arrives back at A . After this point, A will receive only dupACKs. A has received $C = cwnd - 1$ ACKs since the last increment to $cwnd$, but must receive $C + 1 = cwnd$ ACKs in order to increment $cwnd$ again. This will not happen, as no more new ACKs will arrive until the lost packet is transmitted.

Following this, $cwnd$ is reduced and the next sawtooth begins; the only packet that is lost is the “extra” packet of the previous flight.

See [31.2.3 Single Losses](#) for experimental confirmation, and exercise 15.0.

19.9 TCP Assumptions and Scalability

In the TCP design portrayed above, several embedded assumptions have been made. Perhaps the most important is that **every loss is treated as evidence of congestion**. As we shall see in the next chapter, this fails for high-bandwidth TCP (when rare random losses become significant); it also fails for TCP over wireless (either Wi-Fi or other), where lost packets are much more common than over Ethernet. See [21.6 The High-Bandwidth TCP Problem](#) and [21.7 The Lossy-Link TCP Problem](#).

The TCP $cwnd$ -increment strategy – to increment $cwnd$ by 1 for each RTT – has some assumptions of scale. This mechanism works well for cross-continent RTT's on the order of 100 ms, and for $cwnd$ in the low hundreds. But if $cwnd = 2000$, then it takes 100 RTTs – perhaps 20 seconds – for $cwnd$ to grow 10%; linear increase becomes *proportionally* quite slow. Also, if the RTT is very long, the $cwnd$ increase is slow. The absolute set-by-the-speed-of-light minimum RTT for geosynchronous-satellite Internet is 480 ms, and typical satellite-Internet RTTs are close to 1000 ms. Such long RTTs also lead to slow $cwnd$ growth; furthermore, as we shall see below, such long RTTs mean that these TCP connections compete poorly with other connections. See [21.8 The Satellite-Link TCP Problem](#).

Another implicit assumption is that if we have a lot of data to transfer, we will send all of it in one single connection rather than divide it among multiple connections. The web http protocol violates this routinely, though. With multiple short connections, $cwnd$ may never properly converge to the steady state for any of them; TCP Reno does not support

carrying over what has been learned about `cwnd` from one connection to the next. A related issue occurs when a connection alternates between relatively idle periods and full-on data transfer; most TCPs set `cwnd=1` and return to slow start when sending resumes after an idle period.

Finally, TCP's Fast Retransmit assumes that routers do not significantly reorder packets.

19.10 TCP Parameters

In TCP Reno's Additive Increase, Multiplicative Decrease strategy, the increase increment is 1.0 and the decrease factor is $1/2$. It is natural to ask if these values have some especial significance, or what are the consequences if they are changed.

Neither of these values plays much of a role in determining the average value of `cwnd`, at least in the short term; this is largely dictated by the path capacity, including the queue size of the bottleneck router. It seems clear that the exact value of the increase increment has no bearing on congestion; the per-RTT increase is too small to have a major effect here. The decrease factor of $1/2$ *may* play a role in responding promptly to incipient congestion, in that it reduces `cwnd` sharply at the first sign of lost packets. However, as we shall see in [22.6 TCP Vegas](#), TCP Vegas in its "normal" mode manages quite successfully with an Additive Decrease strategy, decrementing `cwnd` by 1 at the point it detects approaching congestion (to be sure, it detects this well before packet loss), and, by some measures, responds better to congestion than TCP Reno. In other words, not only is the exact value of the AIMD decrease factor not critical for congestion management, but multiplicative decrease itself is not mandatory.

There are two informal justifications in [\[JK88\]](#) for a decrease factor of $1/2$. The first is in slow start: if at the N th RTT it is found that `cwnd` = 2^N is too big, the sender falls back to `cwnd`/2 = 2^{N-1} , which is known to have worked without losses the previous RTT. However, a change here in the decrease policy might best be addressed with a concomitant change to slow start; alternatively, the reduction factor of $1/2$ might be left still to apply to "unbounded" slow start, while a new factor of β might apply to threshold slow start.

The second justification for the reduction factor of $1/2$ applies directly to the congestion avoidance phase; written in 1988, it is quite remarkable to the modern reader:

If the connection is steady-state running and a packet is dropped, it's probably because a new connection started up and took some of your bandwidth.... [I]t's probable that there are now exactly two conversations sharing the bandwidth. I.e., you should reduce your window by half because the bandwidth available to you has been reduced by half. [\[JK88\]](#), §D

Today, busy routers may have thousands of simultaneous connections. To be sure, Jacobson and Karels go on to state, "if there are more than two connections sharing the

bandwidth, halving your window is conservative – and being conservative at high traffic intensities is probably wise”. This advice remains apt today.

But while they do not play a large role in setting $cwnd$ or in avoiding “congestive collapse”, it turns out that these increase-increment and decrease-factor values of 1 and $1/2$ respectively play a *great* role in **fairness**: making sure competing connections get the bandwidth allocation they “should” get. We will return to this in [20.3 TCP Reno Fairness with Synchronized Losses](#), and also [21.4 AIMD Revisited](#).

19.11 Epilog

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

There are also broad changes in TCP usage patterns. Twenty years ago, the vast majority of all TCP traffic represented downloads from “major” servers. Today, over half of all Internet TCP traffic is peer-to-peer rather than server-to-client. The rise in online video streaming creates new demands for excellent TCP real-time performance.

In the next chapter we will examine the dynamic behavior of TCP Reno, focusing in particular on fairness between competing connections, and on other problems faced by TCP Reno senders. Then, in [22 Newer TCP Implementations](#), we will survey some attempts to address these problems.

19.12 Exercises

Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercises marked with a \diamond have solutions or hints at [34.14 Solutions for TCP Reno and Congestion Management](#).

1.0. Consider the following network, with each link other than the first having a bandwidth delay of 1 packet/second. Assume ACKs travel instantly from B to R (and thus to A). Assume there are no propagation delays, so the RTT_{noLoad} is 4; the $bandwidth \times RTT$ product is then 4 packets. If A uses sliding windows with a window size of 6, the queue at R1 will eventually have size 2.

Suppose A uses **threshold** slow start ([19.2.2 Threshold Slow Start](#)) with $ssthresh = 6$, and with $cwnd$ initially 1. Complete the table below until two rows after $cwnd = 6$; for these final two rows, $cwnd$ has reached $ssthresh$ and so A will send only one new packet for each ACK received. Assume the queue at R1 is large enough that no packets are dropped. How big will the queue at R1 grow?

T	A sends	R1 queues	R1 sends	B receives/ACKs	cwnd
0	1		1		1
1					
2					
3					
4	2,3	3	2	1	2
5			3		2
6					
7					
8	4,5	5	4	2	3

Note that if, instead of using slow start, A simply sends the initial windowful of 6 packets all at once, then the queue at R1 will initially hold $6-1 = 5$ packets.

2.0. Consider the following network from [19.2.3 Slow-Start Multiple Drop Example](#), with links labeled with bandwidths in packets/ms. Assume ACKs travel instantly from B to R (and thus to A).

A begins sending to B using unbounded slow start, beginning with Data[1] at $T=0$. Initially, $cwnd = 1$. Write out a table of packet transmissions and deliveries assuming R's maximum queue size is 4 (not counting the packet currently being forwarded). Stop with the arrival at A of the first dupACK triggered by the arrival at B of the packet that followed the first packet that was dropped by R. No retransmissions will occur by then.

T	A sends	R queues	R drops	R sends	B receives/ACKs
0	Data[1]			Data[1]	

3.0. Consider the network from exercise 2.0 above. A again begins sending to B using unbounded slow start, but this time R's queue size is 2, not counting the packet currently being forwarded. Make a table showing all packet transmissions by A, all packet drops by R, and other columns as are useful. Assume no retransmission mechanism is used at all (no timeouts, no fast retransmit), and that A sends new data only when it receives new ACKs (dupACKs, in other words, do not trigger new data transmissions). With these assumptions, new data transmissions will eventually cease; continue the table until all transmitted data packets are received by B.

4.0. Suppose a connection starts with $cwnd=1$ and increments $cwnd$ by 1 each RTT with no loss, and sets $cwnd$ to $cwnd/2$, rounding down, on each RTT with at least one loss. Lost packets are **not retransmitted**, and propagation delays dominate so each windowful is sent more or less together. Packets 5, 13, 14, 23 and 30 are lost. What is the window size each RTT, up until the first 40 packets are sent? What packets are sent each RTT? Hint: in the first RTT, Data[1] is sent. There is no loss, so in the second RTT $cwnd = 2$ and Data[2] and Data[3] are sent.

5.0. Suppose TCP Reno is used to transfer a large file over a path with bandwidth high enough that, during slow start, $cwnd$ can be treated as doubling each RTT as in [19.2 Slow Start](#). Assume the receiver places no limits on window size.

- (a). How many RTTs will it take for the window size to first reach $\sim 8,000$ packets (about 2^{13}), assuming unbounded slow start is used and there are no packet losses?
- (b). Approximately how many packets will have been sent and acknowledged by that point?
- (c). Now assume the bandwidth is 100 packets/ms and the RTT is 80 ms, making the bandwidth \times delay product 8,000 packets. What fraction of the total bandwidth will have been used by the connection up to the point where the window size reaches 8000? Hint: the total bandwidth is 8,000 packets per RTT.

6.0. (a) Repeat the diagram in [19.4 TCP Reno and Fast Recovery](#), done there with $cwnd=10$, for a window size of 8. Assume as before that the lost packet is Data[10]. There will be seven dupACK[9]'s, which it may be convenient to tag as dupACK[9]/11 through dupACK[9]/17. Be sure to indicate clearly when sending resumes.

- (b). Suppose you try to do this with a window size of 6. Is this window size big enough for Fast Recovery still to work? If so, at what dupACK[9]/N does new data transmission begin? If not, what goes wrong?

7.0. Suppose the window size is 100, and Data[1001] is lost. There will be 99 dupACK[1000]'s sent, which we may denote as dupACK[1000]/1002 through dupACK[1000]/1100. TCP Reno is used.

- (a). At which dupACK[1000]/N does the sender start sending new data?
- (b). When the retransmitted data[1001] arrives at the receiver, what ACK is sent in response?
- (c). When the acknowledgment in (b) arrives back at the sender, what data packet is sent?

Hint: express EFS in terms of dupACK[1000]/N, for $N \geq 1004$. The third dupACK is dupACK[1000]/1004; what is EFS at that point after retransmission of Data[1001]?

8.0. Suppose the window size is 40, and Data[1001] is lost. Packet 1000 will be ACKed normally. Packets 1001–1040 will be sent, and 1002–1040 will each trigger a duplicate ACK[1000].

- (a). What actual data packets trigger the first three dupACKs? (The first ACK[1000] is triggered by Data[1000]; don't count this one as a duplicate.)
- (b). After the third dupACK[1000] has been received and the lost data[1001] has been retransmitted, how many packets/ACKs should the sender estimate as in flight?

When the retransmitted Data[1001] arrives at the receiver, ACK[1040] will be sent back.

- (c). What is the first Data[N] sent for which the response is ACK[N], for $N > 1000$?

(d). What is the first N for which $\text{Data}[N+20]$ is sent in response to $\text{ACK}[N]$ (this represents the point when the connection is back to normal sliding windows, with a window size of 20)?

9.0. Recall ([19.2 Slow Start](#)) that during slow start cwnd is incremented by 1 for each arriving ACK, resulting in the transmission of two new data packets. Suppose slow-start is modified so that, on each ACK, *three* new packets are sent rather than two; cwnd will now triple after each RTT, taking values 1, 3, 9, 27,

(a). For each arriving ACK, by how much must cwnd now be incremented?

(b). Suppose a path has mostly propagation delay. Progressively larger windowfuls are sent, with sizes successive powers of 3, until a cwnd is reached where a packet loss occurs. What window size can the sender be reasonably sure *does* work, based on earlier experience?

10.0. Suppose in the example of [19.5 TCP NewReno](#), $\text{Data}[4]$ had *not* been lost.

(a). When $\text{Data}[1]$ is received, what ACK would be sent in response?

(b). At what point in the diagram is the sender able to resume ordinary sliding windows with $\text{cwnd} = 6$?

11.0. Suppose in the example of [19.5 TCP NewReno](#), $\text{Data}[1]$ and $\text{Data}[2]$ had been lost, but not $\text{Data}[4]$.

(a). The third $\text{dupACK}[0]$ is sent in response to what $\text{Data}[N]$?

(b). When the retransmitted $\text{Data}[1]$ reaches the receiver, $\text{ACK}[1]$ is the response. When this $\text{ACK}[1]$ reaches the sender, which Data packets are sent in response?

12.0. Suppose two TCP connections have the same RTT and share a bottleneck link, on which there is no other traffic. The size of the bottleneck queue is negligible when compared to the $\text{bandwidth} \times \text{RTT}_{\text{noLoad}}$ product. Loss events occur at regular intervals, and are completely synchronized. Show that the two connections together will use 75% of the total bottleneck-link capacity, as in [19.7 TCP and Bottleneck Link Utilization](#) (there done for a single connection).

See also Exercise 16.0 of chapter [21 Further Dynamics of TCP](#).

13.0. In [19.7 TCP and Bottleneck Link Utilization](#) we showed that, if the bottleneck router queue capacity was 50% of a TCP Reno connection's transit capacity, and there was no other traffic, then the bottleneck-link utilization would be 95.8%.

(a). Suppose the queue capacity is $1/3$ of the transit capacity. Show the bottleneck link utilization is $11/12$, or 91.7%. Draw a diagram of the tooth, and find the relative lengths of the link-unsaturated and queue-filling phases. You may round off cwnd_{max} to $4/3$ the transit capacity (the value of cwnd just before the packet loss; the exact value of cwnd_{max} is higher by 1).

(b). \diamond Derive a formula for the link utilization in terms of the ratio $f < 1$ of queue capacity to transit capacity. Make the same simplifying assumption as in part (a).

14.0. In [19.2.1 TCP Reno Per-ACK Responses](#) we stated that the per-ACK response of a TCP sender was to increment `cwnd` as follows:

$$\text{cwnd} = \text{cwnd} + 1/\text{cwnd}$$

(a). What is the corresponding formulation if the window size is in fact measured in bytes rather than packets? Let `SMSS` denote the sender's maximum segment size, and let `bwnd = SMSS × cwnd` denote the congestion window as measured in bytes. Hint: solve this last equation for `cwnd` and plug the result in above.

(b). What is the appropriate formulation of $\text{cwnd} = \text{cwnd} + 1/\text{cwnd}$ if delayed ACKs are used ([18.8 TCP Delayed ACKs](#)) and we still want `cwnd` to be incremented by 1 for each windowful? Assume we are back to measuring `cwnd` in packets.

15.0. In [19.8 Single Packet Losses](#) we simplified the argument slightly by assuming that when A sent a pair of packets, they arrived at R “essentially simultaneously”.

Give a scenario in which it is not the “extra” packet (the second of the pair) that is lost, but the packet that follows it. Hint: see [31.3.4.1 Single-sender phase effects](#).

1.1 Layers

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

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

Fig. 1: Five-layer network model

This LAN physical/logical division gives us the Internet **five-layer model**. This is less a formal hierarchy than an *ad hoc* classification method. We will return to this below

in [1.15 IETF and OSI](#), where we will also introduce two more rather obscure layers that complete the **seven**-layer model.

1.2 Data Rate, Throughput and Bandwidth

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

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

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

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

1.3 Packets

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

Fig. 2: Packets with headers

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

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

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

Generally each layer adds its own header. Ethernet headers are typically 14 bytes, IP headers 20 bytes, and TCP headers 20 bytes. If a TCP connection sends 512 bytes of data per packet, then the headers amount to 10% of the total, a not-unreasonable overhead. For one common Voice-over-IP option, packets contain 160 bytes of data and 54 bytes of headers, making the header about 25% of the total. Compressing the 160 bytes of audio, however, may bring the data portion down to 20 bytes, meaning that the headers are now 73% of the total; see [25.11.4 RTP and VoIP](#).

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

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

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

be represented as a sequence of bytes in either *big-endian* or *little-endian* byte order ([16.1.5 Binary Data](#)). [RFC 1700](#) specifies that Internet protocols use big-endian byte order, therefore sometimes called network byte order.

1.4 Datagram Forwarding

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

In datagram forwarding this is achieved by providing each switch with a **forwarding table** of (destination,next_hop) pairs. When a packet arrives, the switch looks up the destination address (presumed globally unique) in its forwarding table and finds the **next_hop** information: the immediate-neighbor address to which – or interface by which – the packet should be forwarded in order to bring it one step closer to its final destination. The next_hop value in a forwarding table is a single entry; each switch is responsible for only one step in the packet’s path. However, if all is well, the network of switches will be able to deliver the packet, one hop at a time, to its ultimate destination.

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

Just how the forwarding table is built is a question for later; we will return to this for Ethernet switches in [2.4.1 Ethernet Learning Algorithm](#) and for IP routers in [13 Routing-Update Algorithms](#). For now, the forwarding tables may be thought of as created through initial configuration.

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

Fig. 3: Two switches, S1 and S2

Small numeric labels are interface numbers

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

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

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

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

In the network diagrammed above, all links are point-to-point, and so each interface corresponds to the unique immediate neighbor reached by that interface. We can thus replace the interface entries in the next_hop column with the name of the corresponding **neighbor**. For human readers, using neighbors in the next_hop column is usually much more readable. S1's table can now be written as follows (with consolidation of the entries for B, D and E):

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

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

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

The fundamental alternative to datagram forwarding is **virtual circuits**, [5.4 Virtual Circuits](#). In virtual-circuit networks, each router maintains state about each connection passing through it; different connections can be routed differently. If packet forwarding depends, for example, on per-connection information – *eg* both TCP port numbers – it is not datagram forwarding. (That said, it arguably still *is* datagram forwarding if web

traffic – to TCP port 80 – is forwarded differently than all other traffic, because that rule does not depend on the specific connection.)

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

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

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

S1	
destination	next_hop
A	0
C	1
default	2

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

1.5 Topology

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

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

Fig. 4: A network with more than one path from A to B

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

As we shall see, many LANs (in particular Ethernet) prefer “tree” networks with no redundancy, while IP has complex protocols in support of redundancy ([13 Routing-Update Algorithms](#)).

1.5.1 Traffic Engineering

In some cases the decision above between routes A—S1—S2—S4—B and A—S1—S3—S4—B might be of material significance – perhaps the S2-S4 link is slower than the others, or is more congested. We will use the term **traffic engineering** to refer to any intentional selection of one route over another, or any elevation of the priority of one class of traffic. The route selection can either be directly intentional, through configuration, or can be implicit in the selection or tuning of algorithms that then make these route-selection choices automatically. As an example of the latter, the algorithms of [13.1 Distance-Vector Routing-Update Algorithm](#) build forwarding tables on their own, but those tables are greatly influenced by the administrative assignment of link costs.

With pure datagram forwarding, used at either the LAN or the IP layer, the path taken by a packet is determined solely by its destination, and traffic engineering is limited to the choices made between alternative paths. We have already, however, suggested that datagram forwarding can be extended to take quality-of-service information into

account; this may be used to have voice traffic – with its relatively low bandwidth but intolerance for delay – take an entirely different path than bulk file transfers. Alternatively, the network manager may simply assign voice traffic a higher priority, so it does not have to wait in queues behind file-transfer traffic.

The quality-of-service information may be set by the end-user, in which case an ISP may wish to recognize it only for designated users, which in turn means that the ISP will implicitly use the traffic source when making routing decisions. Alternatively, the quality-of-service information may be set by the ISP itself, based on its best guess as to the application; this means that the ISP may be using packet size, port number ([1.12 Transport](#)) and other contents as part of the routing decision. For some explicit mechanisms supporting this kind of routing, see [13.6 Routing on Other Attributes](#).

At the LAN layer, traffic-engineering mechanisms are historically limited, though see [3.4 Software-Defined Networking](#). At the IP layer, more strategies are available; see [25 Quality of Service](#).

1.6 Routing Loops

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

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

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

In datagram routing, a switch is responsible only for the next hop to the ultimate destination; if a switch has a complete path in mind, there is no guarantee that the

next_hop switch or any other downstream switch will continue to forward along that path. Misunderstandings can potentially lead to routing loops. Consider this network:

Fig. 5: Network consisting of five nodes in a ring

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

1.7 Congestion

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

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

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

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

When to Upgrade?

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

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

Congestion *is* a sign of a problem in real-time networks, which we will consider in [25 Quality of Service](#). In these networks losses due to congestion must generally be kept to an absolute minimum; one way to achieve this is to limit the acceptance of new connections unless sufficient resources are available.

1.8 Packets Again

Perhaps the core justification for packets, Baran's concerns about node failure notwithstanding, is that the same link can carry, at different times, different packets representing traffic to different destinations and from different senders. Thus, packets are the key to supporting **shared transmission lines**; that is, they support the **multiplexing** of multiple communications channels over a single cable. The alternative of a separate physical line between every pair of machines grows prohibitively complex very quickly (though **virtual circuits** between every pair of machines in a datacenter are not uncommon; see [5.4 Virtual Circuits](#)).

From this shared-medium perspective, an important packet feature is the maximum packet size, as this represents the maximum time a sender can send before other senders get a chance. The alternative of unbounded packet sizes would lead to prolonged network unavailability for everyone else if someone downloaded a large file in a single 1 Gigabit packet. Another drawback to large packets is that, if the packet is corrupted, the entire packet must be retransmitted; see [7.3.1 Error Rates and Packet Size](#).

When a router or switch receives a packet, it (generally) reads in the entire packet before looking at the header to decide to what next node to forward it. This is known as **store-and-forward**, and introduces a **forwarding delay** equal to the time needed to read in the entire packet. For individual packets this forwarding delay is hard to avoid (though some higher-end switches do implement **cut-through** switching to begin forwarding a packet before it has fully arrived), but if one is sending a long train of packets then by keeping multiple packets *en route* at the same time one can essentially eliminate the significance of the forwarding delay; see [7.3 Packet Size](#).

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

- **Bandwidth delay**, *ie* sending 1000 Bytes at 20 Bytes/millisecond will take 50 ms. This is a per-link delay.
- **Propagation delay** due to the speed of light. For example, if you start sending a packet right now on a 5000-km cable across the US with a propagation speed of

200 m/μsec (= 200 km/ms, about 2/3 the speed of light in vacuum), the first bit will not arrive at the destination until 25 ms later. The bandwidth delay then determines how much after that the entire packet will take to arrive.

- **Store-and-forward delay**, equal to the sum of the bandwidth delays out of each router along the path
- **Queuing delay**, or waiting in line at busy routers. At bad moments this can exceed 1 sec, though that is rare. Generally it is less than 10 ms and often is less than 1 ms. Queuing delay is the only delay component amenable to reduction through careful engineering.

See [7.1 Packet Delay](#) for more details.

1.9 LANs and Ethernet

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

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

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

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

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

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

reduce throughput, in the larger view they should perhaps be thought of as a part of a remarkably inexpensive shared-access mediation protocol.

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

Password Sniffing

In the fall of 1994 at Loyola University I remotely changed the root password on several CS-department unix machines at the other end of campus, using telnet. I told no one. Within two hours, someone else logged into one of these machines, using the new password, from a host in Europe. Password sniffing was the likely culprit.

Two months later was the so-called “Christmas Day Attack” ([18.3.1 ISNs and spoofing](#)). One of the hosts used to launch this attack was Loyola’s hacked `apollo.it.luc.edu`. It is unclear the degree to which password sniffing played a role in that exploit.

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

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

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

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

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

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

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

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

To forward packets correctly, switches must know where all active destination addresses in the LAN are located; traditional Ethernet switches do this by a passive **learning** algorithm. (IP routers, by comparison, use "active" protocols, and some newer Ethernet switches take the approach of [3.4 Software-Defined Networking](#).) Typically a host physical address is entered into a switch's forwarding table when a packet from that host is first *received*; the switch notes the packet's arrival interface and *source* address and assumes that the same interface is to be used to deliver packets back to that sender. If a given destination address has not yet been seen, and thus is not in the forwarding table, Ethernet switches still have the backup delivery option of **flooding**: forwarding the packet to everyone by treating the destination address like the broadcast address, and allowing the host Ethernet cards to sort it out. Since this broadcast-like process is not generally used for more than one packet (after that, the switches will have learned the correct forwarding-table entries), the risks of excessive traffic and of eavesdropping are minimal.

The (host,interface) forwarding table is often easier to think of as (host,next_hop), where the next_hop node is whatever switch or host is at the immediate other end of the link

connecting to the given interface. In a fully switched network where each link connects only two interfaces, the two perspectives are equivalent.

1.10 IP – Internet Protocol

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

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

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

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

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

For example, the original IP address allocation for Loyola University Chicago was 147.126.0.0, a class B. In binary, 147 is **10010011**.

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

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

Nowadays the division into the network and host bits is dynamic, and can be made at different positions in the address at different levels of the network. For example, a small organization might receive a /27 address block (1/8 the size of a class-C /24) from its ISP, *eg* 200.1.130.96/27. The ISP routes to the organization based on this /27 prefix. At some higher level, however, routing might be based on the prefix 200.1.128/18; this might, for example, represent an address block assigned to the ISP (note that the first 18 bits of 200.1.130.x match 200.1.128; the first two bits of 128 and 130, taken as 8-bit quantities, are “10”). The network/host division point is *not* carried within the IP header; routers negotiate this division point when they negotiate the next_hop forwarding information. We will return to this in [9.5 The Classless IP Delivery Algorithm](#).

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

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

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

Each individual LAN technology has a **maximum packet size** it supports; for example, Ethernet has a maximum packet size of about 1500 bytes but the once-competing Token Ring had a maximum of 4 kB. Today the world has largely standardized on Ethernet and almost entirely standardized on Ethernet packet-size limits, but this was not the case when IP was introduced and there was real concern that two hosts on

separate large-packet networks might try to exchange packets too large for some small-packet intermediate network to carry.

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

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

Architecturally, this best-effort model represents what is known as **connectionless** networking: the IP layer does not maintain information about endpoint-to-endpoint connections, and simply forwards packets like a giant LAN. Responsibility for creating and maintaining connections is left for the next layer up, the TCP layer. Connectionless networking is *not* the only way to do things: the alternative could have been some form **connection-oriented** internetworking, in which routers *do* maintain state information about individual connections. Later, in [5.4 Virtual Circuits](#), we will examine how virtual-circuit networking can be used to implement a connection-oriented approach; virtual-circuit switching is the primary alternative to datagram switching.

Connectionless (IP-style) and connection-oriented networking each have advantages. Connectionless networking is conceptually more reliable: if routers do not hold connection state, then they cannot *lose* connection state. The path taken by the packets in some higher-level connection can easily be dynamically rerouted. Finally, connectionless networking makes it hard for providers to bill by the connection; once upon a time (in the era of dollar-a-minute phone calls) this was a source of mild astonishment to many new users. (This was not always a given; the paper [\[CK74\]](#) considers, among other things, the possibility of per-packet accounting.)

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

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

Although we will often assume, for simplicity, that routers have a fixed input queue size, the reality is often a little more complicated. See [21.5 Active Queue Management](#) and [23 Queuing and Scheduling](#).

1.10.1 IP Forwarding

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

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

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

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

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

Suppose A is the sending host, sending a packet to a destination host D. The IP header of the packet will contain D’s IP address in the “destination address” field (it will also

contain A's own address as the "source address"). The first step is for A to determine whether D is on the same LAN as itself or not; that is, whether D is **local**. This is done by looking at the network part of the destination address, which we will denote by D_{net} . If this net address is the same as A's (that is, if it is equal numerically to A_{net}), then A figures D is on the same LAN as itself, and can use direct LAN delivery. It looks up the appropriate physical address for D (probably with the **ARP** protocol, [10.2 Address Resolution Protocol: ARP](#)), attaches a LAN header to the packet in front of the IP header, and sends the packet straight to D via the LAN.

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

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

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

To make this concrete, consider the following diagram:

Fig. 6: Two LANs (200.0.0 and 200.0.1) joined by three routers R1,R2,R3

With Ethernet-style forwarding, R2 would have to maintain entries for each of A,B,C,D,E,F. With IP forwarding, R2 has just two entries to maintain in its forwarding table: 200.0.0/24 and 200.0.1/24. If A sends to D, at 200.0.1.37, it puts this address into the IP header, notes that 200.0.0 \neq 200.0.1, and thus concludes D is not a local delivery. A therefore sends the packet to its router R1, using LAN delivery. R1 looks up

the destination network 200.0.1 in its forwarding table and forwards the packet to R2, which in turn forwards it to R3. R3 now sees that it *is* connected directly to the destination network 200.0.1, and delivers the packet via the LAN to D, by looking up D's physical address.

In this diagram, IP addresses for the ends of the R1–R2 and R2–R3 links are not shown. They could be assigned global IP addresses, but they could also use “private” IP addresses. Assuming these links are point-to-point links, they might not actually need IP addresses at all; we return to this in [9.8 Unnumbered Interfaces](#).

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

The Invertebrate Internet

The backbone is not as essential as it once was. Once Upon A Time, all traffic between different providers passed through the backbone. The legacy backbone still exists, but today it is also common for traffic from large providers such as [Google](#) to take a backbone-free path; such providers connect (or “peer”) directly with large residential ISPs such as [Comcast](#). Google refers to this as their “Edge Network”; see [peering.google.com](#) and also [15.7.1 MED values and traffic engineering](#).

We will refer to the Internet **backbone** as those IP routers that specialize in large-scale routing on the commercial Internet, and which generally have forwarding-table entries covering all public IP addresses; note that this is essentially a business definition rather than a technical one. We can revise the table-size claim of the previous paragraph to state that, while there are many *private* IP networks, there are about 800,000 separate network prefixes (as of 2019) visible to the backbone. (In 2012, the year this book was started, there were about 400,000 prefixes.) A forwarding table of 800,000 entries is quite feasible; a table a hundred times larger is not, let alone a thousand times larger. For a graph of the growth in network prefixes / forwarding-table entries, see [15.5 BGP Table Size](#).

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

table size. The Internet backbone can be approximately described, in fact, as those routers that do *not* have a default entry.

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

Just how routers build their $\langle \text{destnet}, \text{next_hop} \rangle$ forwarding tables is a major topic itself, which we cover in [13 Routing-Update Algorithms](#). Unlike Ethernet, IP routers do *not* have a “flooding” delivery mechanism as a fallback, so the tables must be constructed in advance. (There is a limited form of IP broadcast, but it is basically intended for reaching the local LAN only, and does not help at all with delivery in the event that the destination network is unknown.)

Most forwarding-table-construction algorithms used on a set of routers under common management fall into either the **distance-vector** or the **link-state** category; these are described in [13 Routing-Update Algorithms](#). Routers *not* under common management – that is, neighboring routers belonging to different organizations – exchange information through the Border Gateway Protocol, BGP ([14 Large-Scale IP Routing](#)). BGP allows routing decisions to be based on a fusion of “technical” information (which sites are reachable at all, and through where) together with “policy” information representing legal or commercial agreements: which outside routers are “preferred”, whose traffic an ISP will carry even if it isn’t to one of the ISP’s customers, *etc.*

Most common residential “routers” involve **network address translation** in addition to packet forwarding. See [9.7 Network Address Translation](#).

1.10.2 The Future of IPv4

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

- [ARIN](#) – North America
- [RIPE](#) – Europe, the Middle East and parts of Asia
- [APNIC](#) – East Asia and the Pacific
- [AfrINIC](#) – most of Africa
- [LACNIC](#) – Central and South America

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

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

A few months after the IANA pool ran out in 2011, Microsoft purchased 666,624 IP addresses (2604 Class-C blocks) in a Nortel bankruptcy auction for \$7.5 million. Three years later, IP-address prices fell to half that, but, by 2019, had climbed to the \$20-and-up range. In that year Amazon bought a 4-million-address block (44.0.0.0/10) for \$108 million, or \$27/address. Prices remained in the \$20 range through 2020, but by the end of 2021 had climbed to \$50–\$55, with several /19 blocks sold in January 2022 in that range. The market for IPv4 address blocks continues to develop, but the only real solution is widespread adoption of IPv6 with its plentiful 128-bit addresses.

An IPv4 address price in the range of \$20–30 is unlikely to have much impact in residential Internet access, where annual connection fees are often \$600. Large organizations use NAT ([9.7 Network Address Translation](#)) extensively, leading to the need for only a small number of globally visible addresses. The IPv4 address shortage does not even seem to have affected wireless networking. It does, however, lead to inefficient routing tables, as sites that might once have had a single /17 address block – and thus a single backbone forwarding-table entry – might now be spread over more than a hundred /24 blocks and concomitant forwarding entries.

1.11 DNS

IP addresses are hard to remember (nearly impossible in IPv6). The **domain name system**, or DNS ([10.1 DNS](#)), comes to the rescue by creating a way to convert hierarchical text names to IP addresses. Thus, for example, one can type `www.luc.edu` instead of `147.126.1.230`. Virtually all Internet software uses the same basic library calls to convert DNS names to actual addresses.

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

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

1.12 Transport

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

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

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

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

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

A busy server may have thousands of connections to its port 80 (the web port) and hundreds of connections to port 25 (the email port). Web and email traffic are kept

separate by virtue of the different ports used. All those clients to the same port, though, are kept separate because each comes from a unique $\langle \text{host}, \text{port} \rangle$ pair. A TCP connection is determined by the $\langle \text{host}, \text{port} \rangle$ socket address at *each* end; traffic on different connections does not intermingle. That is, there may be multiple independent connections to $\langle \text{www.luc.edu}, 80 \rangle$. This is somewhat analogous to certain business telephone numbers of the “*operators are standing by*” type, which support multiple callers at the same time to the same toll-free number. Each call to that number is answered by a different operator (corresponding to a different cpu process), and different calls do not “overhear” each other.

TCP uses the **sliding-windows algorithm**, [8 Abstract Sliding Windows](#), to keep multiple packets *en route* at any one time. The **window size** represents the number of packets simultaneously in transit (TCP actually keeps track of the window size in bytes, but packets are easier to visualize). If the window size is 10 packets, for example, then at any one time 10 packets are in transit (perhaps 5 data packets and 5 returning acknowledgments). Assuming no packets are lost, then as each acknowledgment arrives the window “slides forward” by one packet. The data packet 10 packets ahead is then sent, to maintain a total of 10 packets on the wire. For example, consider the moment when the ten packets 20–29 are in transit. When ACK[20] is received, the number of packets outstanding drops to 9 (packets 21–29). To keep 10 packets in flight, Data[30] is sent. When ACK[21] is received, Data[31] is sent, and so on.

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

The ideal window size, at least from a throughput perspective, is such that it takes one round-trip time to send an entire window, so that the next ACK will always be arriving just as the sender has finished transmitting the window. Determining this ideal size, however, is difficult; for one thing, the ideal size varies with network load. As a result, TCP approximates the ideal size. The most common TCP strategy – that of so-called TCP Reno – is that the window size is slowly raised until packet loss occurs, which TCP takes as a sign that it has reached the limit of available network resources. At that point the window size is reduced to half its previous value, and the slow climb resumes. The effect is a “sawtooth” graph of window size with time, which oscillates (more or less) around the “optimal” window size. For an idealized sawtooth graph, see [19.1.1 The Somewhat-Steady State](#); for some “real” (simulation-created) sawtooth graphs see [31.4.1 Some TCP Reno cwnd graphs](#).

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

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

The basic alternative to TCP is known as **UDP**, for User Datagram Protocol. UDP, like TCP, provides port numbers to support delivery to multiple endpoints within the receiving host, in effect to a specific process on the host. As with TCP, a UDP socket consists of a ⟨host,port⟩ pair. UDP also includes, like TCP, a checksum over the data. However, UDP omits the other TCP features: there is no connection setup, no lost-packet detection, no automatic timeout/retransmission, and the application must manage its own packetization. This simplicity should not be seen as all negative: the absence of connection setup means data transmission can get started faster, and the absence of lost-packet detection means there is no head-of-line blocking. See [16 UDP Transport](#).

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

1.12.1 Transport Communications Patterns

The two “classic” traffic patterns for Internet communication are these:

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

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

To the above we might add **request/reply** operations, *eg* to query a database or to make DNS requests. TCP is widely used here as well, though most DNS traffic still uses UDP. There are periodic calls for a new protocol specifically addressing this pattern, though at this point the use of TCP is well established. If a *sequence* of request/reply operations is envisioned, a single TCP connection makes excellent sense, as the connection–setup overhead is minimal by comparison. See also [16.5 Remote Procedure Call \(RPC\)](#) and [18.15.2 SCTP](#).

This century has seen an explosion in **streaming video** ([25.3.2 Streaming Video](#)), in lengths from a few minutes to a few hours. Streaming radio stations might be left playing indefinitely. TCP generally works well here, assuming the receiver can get, say, a minute ahead, buffering the video that has been received but not yet viewed. That way, if there is a dip in throughput due to congestion, the receiver has time to recover. Buffering works a little less well for streaming radio, as the listener doesn’t want to get too far behind, though ten seconds is reasonable. Fortunately, audio bandwidth is smaller.

Another issue with streaming video is the bandwidth demand. Most streaming–video services attempt to estimate the available throughput, and then *adapt* to that throughput by changing the video resolution ([25.3 Real-time Traffic](#)).

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

If the video (or, for that matter, voice audio) is *interactive*, there is much less opportunity for stream buffering. If someone asks a simple question on an Internet telephone call, they generally want an answer more or less immediately; they do not expect to wait for the answer to make it through the other party’s stream buffer. 200 ms of buffering is noticeable. Here we enter the realm of genuine real-time traffic ([25.3 Real-time Traffic](#)). UDP is often used to avoid head-of-line blocking. Lower bandwidth helps; voice-grade communications traditionally need only 8 kB/sec, less if compression is used. On the other hand, there may be constraints on the *variation* in delivery time (known as *jitter*; see [25.11.3 RTP Control Protocol](#) for a specific numeric interpretation). Interactive video, with its much higher bandwidth requirements, is more difficult; fortunately, users seem to tolerate the common pauses and freezes.

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

- Internet telephony: there is no benefit in designating the party who place the call as the “client”
- Message passing in a CPU cluster, often using [16.5 Remote Procedure Call \(RPC\)](#)

- The routing-communication protocols of [13 Routing-Update Algorithms](#). When router A reports to router B we might call A the client, but over time, as A and B report to one another repeatedly, the peer-to-peer model makes more sense.
- So-called peer-to-peer file-sharing, where individuals exchange files with other individuals (and as opposed to “cloud-based” file-sharing in which the “cloud” is the server).

[RFC 5694](#) contains additional discussion of peer-to-peer patterns.

1.12.2 Content-Distribution Networks

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

CDN Mapping

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

Large web pages typically contain both *static* content and also individualized *dynamic* content. On a typical Facebook page, for example, the videos and javascript might be considered static, while the individual wall posts might be considered dynamic. The CDN may cache all or most of the static content at each of its edge servers, leaving the dynamic content to come from a centralized server. Alternatively, the dynamic content may be replicated at each CDN edge node as well, though this introduces some real-time coordination issues.

If dynamic content is *not* replicated, the CDN may include private high-speed links between its nodes, allowing for rapid low-congestion delivery to any node. Alternatively, CDN nodes may simply communicate using the public Internet. Finally, the CDN may (or may not) be configured to support fast *interactive* traffic between nodes, *eg* teleconferencing traffic, as is outlined in [25.6.1 A CDN Alternative to IntServ](#).

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

In principle, all that is needed to create a CDN is a multiplicity of datacenters, each with its own connection to the Internet; private links between datacenters are also common. In practice, many CDN providers also try to build direct connections with the ISPs that serve their customers; the Google Edge Network above does this. This can improve

performance and reduce traffic costs; we will return to this in [15.7.1 MED values and traffic engineering](#).

Finding the edge server that is closest to a given user is a tricky issue. There are three techniques in common use. In the first, the edge servers are all given different IP addresses, and DNS is configured to have users receive the IP address of the closest edge server, [10.1 DNS](#). In the second, each edge server has the *same* IP address, and *anycast* routing is used to route traffic from the user to the closest edge server, [15.8 BGP and Anycast](#). Finally, for HTTP applications a centralized server can look up the approximate location of the user, and then redirect the web page to the closest edge server.

1.13 Firewalls

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

A strategy known as **buffer overflow** ([28.2 Stack Buffer Overflow](#)) has been the basis for a great many total-compromise attacks. The idea is to identify a point in a server program where it fills a memory buffer with network-supplied data without careful length checking; almost any call to the C library function `gets(buf)` will suffice. The attacker then crafts an oversized input string which, when read by the server and stored in memory, overflows the buffer and overwrites subsequent portions of memory, typically containing the stack-frame pointers. The usual goal is to arrange things so that when the server reaches the end of the currently executing function, control is returned not to the calling function but instead to the attacker's own payload code located within the string.

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

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

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

The typical home router implements something called network–address translation ([9.7 Network Address Translation](#)), which, in addition to conserving IPv4 addresses, also provides firewall protection.

1.14 Some Useful Utilities

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

`ping`

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

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

See [10.4 Internet Control Message Protocol](#) for how it works. Sometimes `ping` fails because the necessary packets are blocked by a firewall.

`ifconfig`, `ipconfig`, `ip`

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

`nslookup`, `dig` and `host`

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

```
nslookup intronetworks.cs.luc.edu
```

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

```
dig intronetworks.cs.luc.edu
```

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

host intronetworks.cs.luc.edu

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

See [10.1.2 nslookup and dig](#).

traceroute

This lists the route from you to a remote host:

```
traceroute intronetworks.cs.luc.edu

 1  147.126.65.1 (147.126.65.1)  0.751 ms  0.753 ms  0.783 ms
 2  147.126.95.54 (147.126.95.54)  1.319 ms  1.286 ms  1.253 ms
 3  12.31.132.169 (12.31.132.169)  1.225 ms  1.231 ms  1.193 ms
 4  cr83.cgcil.ip.att.net (12.123.7.46)  4.983 ms cr84.cgcil.ip.att.net
    (12.123.7.170)  4.825 ms  4.812 ms
 5  cr83.cgcil.ip.att.net (12.123.7.46)  4.926 ms  4.904 ms  4.888 ms
 6  cr1.cgcil.ip.att.net (12.122.99.33)  5.043 ms cr2.cgcil.ip.att.net
    (12.122.132.109)  5.343 ms  5.317 ms
 7  gar13.cgcil.ip.att.net (12.122.132.121)  3.879 ms  18.347 ms
ggr4.cgcil.ip.att.net (12.122.133.33)  2.987 ms
 8  chi-b21-link.telia.net (213.248.87.253)  2.344 ms  2.305 ms  2.409 ms
 9  nyk-bb2-link.telia.net (80.91.248.197)  24.065 ms nyk-bb1-link.telia.net
    (213.155.136.70)  24.986 ms nyk-bb2-link.telia.net (62.115.137.58)  23.158 ms
10  nyk-b3-link.telia.net (62.115.112.255)  23.557 ms  23.548 ms nyk-b3-
    link.telia.net (80.91.248.178)  24.510 ms
11  netaccess-tic-133837-nyk-b3.c.telia.net (213.248.99.90)  23.957 ms  24.382
    ms  24.164 ms
12  0.e1-4.tbr1.mmu.nac.net (209.123.10.101)  24.922 ms  24.737 ms  24.754 ms
13  207.99.53.42 (207.99.53.42)  24.024 ms  24.249 ms  23.924 ms
```

The last router (and `intronetworks.cs.luc.edu` itself) don't respond to the traceroute packets, so the list is not quite complete. The Windows `tracert` utility is functionally equivalent. See [10.4.1 Traceroute and Time Exceeded](#) for further information.

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

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

route and netstat

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

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

netcat

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

See [16.1.4 netcat](#) and [17.7.1 netcat again](#).

WireShark

This is a convenient combination of packet capture and packet analysis, from wireshark.org. See [17.4 TCP and WireShark](#) and [12.4 Using IPv6 and IPv4 Together](#) for examples.

WireShark was originally named Etherreal. An earlier command-line-only packet-capture program is [tcpdump](#), though WireShark has greatly expanded support for packet-format decoding. Both WireShark and `tcpdump` support both live packet capture and reading from `.pcap` (packet capture) and `.pcapng` (next generation) files.

WireShark is the only non-command-line program listed here. It is sometimes desired to monitor packets on a remote system. If X-windows is involved (*eg* on Linux), this can be done by logging in from one's local system using `ssh -X`, which enables X-windows forwarding, and then starting `wireshark` (or perhaps `sudo wireshark`) from the command line. Other alternatives include `tcpdump` and **tshark**; the latter is part of the WireShark distribution and supports the same packet-decoding facilities as WireShark. Finally, there is [termshark](#), a frontend for `tshark` that offers a terminal-based interface reasonably similar to WireShark's graphical interface.

1.15 IETF and OSI

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

RFC standards sometimes allow modest flexibility. With this in mind, [RFC 2119](#) declares official understandings for the words MUST and SHOULD. A feature labeled with MUST is “an absolute requirement for the specification”, while the term SHOULD is used when

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

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

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

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

and RFC Editor [Jon Postel](#)’s Robustness Principle, here stated in its [RFC 761](#) / [RFC 793](#) form:

TCP implementations should follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others.

Postel’s aphorism is often shortened to “be liberal in what you accept, and conservative in what you send”. As such, it has come in for occasional criticism in recent years with regard to *cryptographic* protocols, for which lax enforcement can lead to security vulnerabilities. To be fair, however, Postel wrote this in an era when protocol specifications sometimes failed to fully spell out the rules for every possible situation, and too-strict implementations sometimes failed to interoperate. Just what *should* happen if a TCP packet arrives with the SYN bit set, for creating a new connection, and also the FIN bit, for terminating that connection? However, TCP specifications today are generally much more complete, and cryptographic protocols even moreso. One way to read Postel’s rule is that protocol implementations should be *as strict as necessary, but no stricter*.

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

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

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

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

Data presentation and session management are important concepts, but in many cases it has not proved necessary, or even particularly useful, to make them into true layers. The layer approach has been very helpful in organizing networking software into separate sections, but is hard to define precisely. One approach is to declare that a layer should communicate directly only with the layers immediately above and below it. The application passes its data to the Transport layer to receive the Transport header, which passes the data to the IP layer to receive the IP header, and on to the LAN layer. Each layer can be seen as an encapsulated software object, or module, by the layer above, and each layer in turn encapsulates the packet from its parent layer by adding a new header. This is mostly true for the Transport, IP and LAN layers, but there are irregularities: the transport-layer checksum, for example, needs information from the IP layer, and may in fact end up being calculated at the LAN layer ([16.1.3.2 UDP and IP addresses](#) and [17.5 TCP Offloading](#)).

Even allowing for these kinds of irregularities, however, it is hard to justify full-fledged layer status for the Session and Presentation actions. What often happens is that the Application layer manages its own Transport connections, and is responsible for reading and writing data directly from and to the Transport layer. The application then uses conventional libraries for Presentation actions such as encryption, compression and format translation, and for Session actions such as handling broken Transport connections and multiplexing streams of data over a single Transport connection. Version 2 of the HTTP protocol, for example, contains a subprotocol for managing multiple streams; this is generally regarded as part of the Application layer, if for no other reason than that it is not available to any other application.

An opposing view is that it is possible to view the SSL/TLS transport-encryption service, [29.5.2 TLS](#), as an example of a true Presentation layer. Applications generally read and write data directly to the SSL/TLS endpoint, which in turn *mostly* encapsulates (as a software module) the underlying TCP connection. The encapsulation is incomplete, though, in that SSL/TLS applications generally are responsible for creating their own Transport-layer (TCP) connections; see [29.5.3 A TLS Programming Example](#) and the

note at the end of [29.5.3.2 TLSserver](#). In the end, while the seven-layer model is reasonably popular, from a software-architecture standpoint those two extra layers aren't really in the same league as those of the five-layer model.

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

It seems clear that the primary reasons the OSI protocols failed in the marketplace were their ponderous bureaucracy for protocol management, their principle that protocols be completed before implementation began, and their insistence on rigid adherence to the specifications to the point of non-interoperability; indeed, Postel's aphorism above may have been intended as a response to this last point. In contrast, the IETF had (and still has) a "two working implementations" rule for a protocol to become a "Draft Standard". From [RFC 2026](#):

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

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

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

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

Fig. 7: A protocol graph

TCP and UDP/RTP are above IP, and ATM and Ethernet are below

Note the thin “wasp” waist at the IP layer: there are many protocols above and below, but only one protocol at the IP layer. This universality of the IP layer has been one of the things contributing to IP’s success.

1.16 Berkeley Unix

Though not officially tied to the IETF, the Berkeley Unix releases became *de facto* reference implementations for most of the TCP/IP protocols. 4.1BSD (BSD for Berkeley Software Distribution) was released in 1981, 4.2BSD in 1983, 4.3BSD in 1986, 4.3BSD–Tahoe in 1988, 4.3BSD–Reno in 1990, and 4.4BSD in 1994.

Descendants today include FreeBSD, OpenBSD and NetBSD. The TCP implementations TCP Tahoe and TCP Reno ([19 TCP Reno and Congestion Management](#)) took their names from the corresponding 4.3BSD releases.

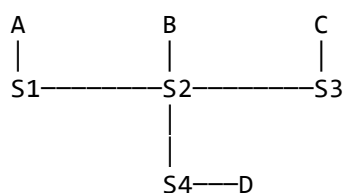
1.17 Epilog

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

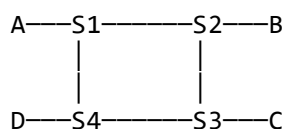
1.18 Exercises

Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercises marked with a ♦ have solutions or hints at [34.1 Solutions for An Overview of Networks](#).

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



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

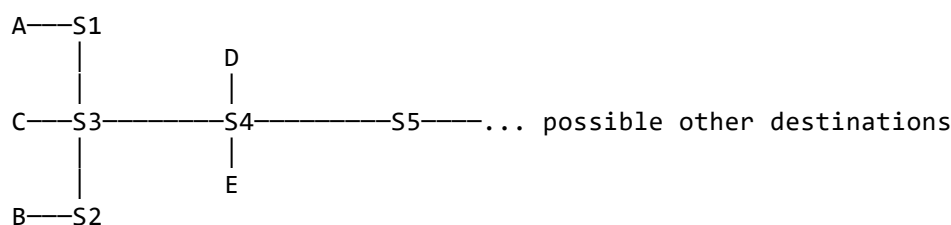


3.0. In the network of the previous exercise, suppose that destinations directly connected to an immediate neighbor are always reached via that neighbor; *eg* S1's forwarding table will always include $\langle B, S2 \rangle$ and $\langle D, S4 \rangle$. This leaves only routes to the diagonally opposite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

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

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

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



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



- Give the forwarding tables for S1 through S4 assuming packets to adjacent nodes are sent along the connecting link, and packets to diagonally opposite nodes are sent clockwise.
- Give the forwarding tables for S1 through S4 assuming the S1–S4 link is not used at all, not even for $S1 \leftrightarrow S4$ traffic.

7.0. Suppose we have switches S1 through S4; the forwarding-table destinations are the switches themselves. The tables for S2 and S3 are as below, where the next_hop value is specified in neighbor form:

S2: $\langle S1, S1 \rangle \langle S3, S3 \rangle \langle S4, S3 \rangle$
 S3: $\langle S1, S2 \rangle \langle S2, S2 \rangle \langle S4, S4 \rangle$

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

(a). The given tables are *consistent* with the network diagrammed in exercise 6.0. Are the tables also consistent with a network in which S1 and S4 are *not* directly connected? If so, give such a network; if not, explain why S1 and S4 must be connected.

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

S3: $\langle S1, S4 \rangle \langle S2, S2 \rangle \langle S4, S4 \rangle$

While the table for S4 is not given, you may assume that forwarding does work correctly. However, you should *not* assume that paths are the shortest possible. Hint: It follows from the S3 table above that the path from S3 to S1 starts $S3 \rightarrow S4$; how will this path continue? The next switch along the path cannot be S1, because of the hypothesis that S1 and S4 are not directly connected.

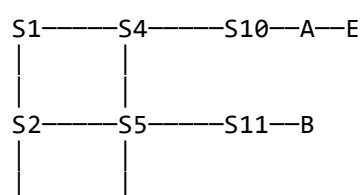
8.0. (a) Suppose a network is as follows, with the only path from A to C passing through B:

... —A—B—C— ...

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

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

9.0. Consider the following arrangement of switches:



S3——S6——S12—C—D—F

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

- (a). A
- (b). B
- (c). C
- (d). \diamond D
- (e). E
- (f). F

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

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

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

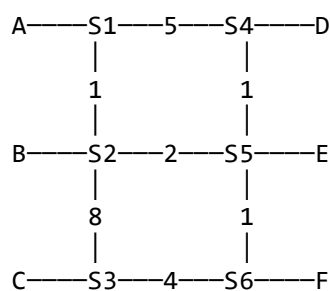
S3: (B,S6), (C,S2), (E,S6), (F,S6)

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

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

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

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



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

- (a). \diamond A→F
- (b). A→D
- (c). A→C

(d). Give the complete forwarding table for S2, where all routes are selected for lowest total cost.

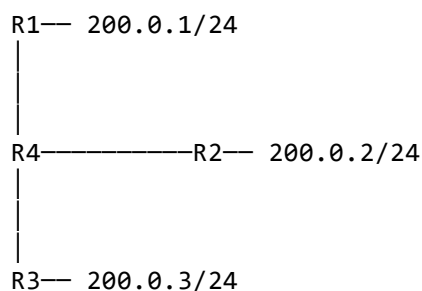
11.0. In exercise 7.0, the routes taken by packets A–D are reasonably direct, but the routes for E and F are rather circuitous.

(a). Assign weights to the seven links S1–S2, S2–S3, S1–S4, S2–S5, S3–S6, S4–S5 and S5–S6, as in exercise 10.0, so that destination E’s route in exercise 9.0 becomes the optimum (lowest total link weight) path.

(b). Assign weights to the seven links that make destination F’s route in exercise 9.0 optimal. (This will be a different set of weights from part (a).)

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

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



9 IP version 4

There are multiple LAN protocols below the IP layer and multiple transport protocols above, but IP itself stands alone. The Internet is the IP Internet. If you want to run your own LAN protocol somewhere, or if you want to run your own transport protocol, the Internet backbone will still work just fine for you. But if you want to change the IP layer, you will encounter difficulty. (Just talk to the IPv6 people, or the IP–multicasting or IP–reservations groups.)

In this chapter we discuss the original core IP protocol – known as version 4, or IPv4, and with a 32–bit address size. Most of the Internet today (2020) still uses IPv4, though IPv6 is making inroads. We will see how the IP layer enables efficient, scalable routing. In the following chapter we discuss some companion protocols: ICMP, ARP, DHCP and DNS.

Despite its ubiquity, IPv4 faces an unsettled future: the Internet has run out of new large blocks of IPv4 addresses ([1.10 IP – Internet Protocol](#)). There is therefore increasing pressure to convert to IPv6, with its 128-bit address size. Progress has been slow, however, and delaying tactics such as IPv4-address markets and NAT ([9.7 Network Address Translation](#)) – by which multiple hosts can share a single public IPv4 address – have allowed IPv4 to continue. Aside from the major change in address structure, there are relatively few differences in the routing models of IPv4 and IPv6. We will study IPv4 in this and the following chapters, and IPv6 in [11 IPv6](#); at points where the IPv4/IPv6 difference doesn't much matter we will simply write "IP".

IPv4 (and IPv6) is, in effect, a universal **routing and addressing** protocol. Routing and addressing are developed together; every node has an IP address and every router knows how to handle IP addresses. IP was originally seen as a way to *interconnect* multiple LANs, but it may make more sense now to view IP as a virtual LAN overlaying all the physical LANs.

A crucial aspect of IP is its **scalability**. As of 2019 the Internet had over 10^9 hosts; this estimate is probably low. However, at the same time the size of the largest forwarding tables was still under 10^6 ([15.5 BGP Table Size](#)). Ethernet, in comparison, scales poorly, as the forwarding tables need one entry for every active host.

How many IPv4 hosts are there?

Counting the size of the Internet is not easy. [The Internet Systems Consortium](#) used to run a [survey](#) based on DNS; it reached one billion **hosts** in 2012 and leveled off there. But not all devices get a DNS entry; behind NAT routers, few do. According to [InternetLiveStats.com](#), as of 2019 there were four billion Internet **users**, about 50% of the world's population, counting an "Internet user" as someone who could access the Internet in their own home. [Facebook](#) reported 2.4 billion monthly active users in June 2019.

Furthermore, IP, unlike Ethernet, offers excellent support for **multiple redundant links**. If the network below were an IP network, each node would communicate with each immediate neighbor via their shared direct link. If, on the other hand, this were an Ethernet network with the spanning-tree algorithm, then one of the four links would simply be disabled completely.

The IP network service model is to act like a giant LAN. That is, there are no acknowledgments; delivery is generally described as best-effort. This design choice is perhaps surprising, but it has also been quite fruitful.

If you want to provide a universal service for delivering any packet anywhere, what else do you need besides routing and addressing? Every network (LAN) needs to be able to carry any packet. The protocols spell out the use of octets (bytes), so the only possible

compatibility issue is that a packet is *too large* for a given network. IPv4 handles this by supporting **fragmentation**: a network may break a too-large packet up into units it can transport successfully. While IPv4 fragmentation is inefficient and clumsy, it does guarantee that any packet can potentially be delivered to any node. (Note, however, that IPv6 has given up on universal fragmentation; [11.5.4 IPv6 Fragment Header](#).)

9.1 The IPv4 Header

The IPv4 Header needs to contain the following information:

- destination and source addresses
- indication of ipv4 versus ipv6
- a Time To Live (TTL) value, to prevent infinite routing loops
- a field indicating what comes next in the packet (*eg* TCP v UDP)
- fields supporting fragmentation and reassembly.

The header is organized as a series of 32-bit words as follows:

The IPv4 header, and basics of IPv4 protocol operation, were originally defined in [RFC 791](#); some minor changes have since occurred. Most of these changes were documented in [RFC 1122](#), though the DS field was defined in [RFC 2474](#) and the ECN bits were first proposed in [RFC 2481](#).

The **Version** field is, for IPv4, the number 4: 0100. The **IHL** field represents the total IPv4 Header Length, in 32-bit words; an IPv4 header can thus be at most 15 words long. The base header takes up five words, so the IPv4 Options can consist of at most ten words. If one looks at IPv4 packets using a packet-capture tool that displays the packets in hex, the first byte will most often be 0x45.

The **Differentiated Services** (DS) field is used by the Differentiated Services suite to specify preferential handling for designated packets, *eg* those involved in VoIP or other real-time protocols. The **Explicit Congestion Notification** bits are there to allow routers experiencing congestion to mark packets, thus indicating to the sender that the transmission rate should be reduced. We will address these in [21.5.3 Explicit Congestion Notification \(ECN\)](#). These two fields together replace the old 8-bit **Type of Service** field.

The **Total Length** field is present because an IPv4 packet may be smaller than the minimum LAN packet size (see Exercise 1) or larger than the maximum (if the IPv4 packet has been fragmented over several LAN packets. The IPv4 packet length, in other words, cannot be inferred from the LAN-level packet size. Because the Total Length field is 16 bits, the maximum IPv4 packet size is 2^{16} bytes. This is probably much too large,

even if fragmentation were not something to be avoided (though see IPv6 “jumbograms” in [11.5.1 Hop-by-Hop Options Header](#)).

The second word of the header is devoted to fragmentation, discussed below at [9.4 Fragmentation](#).

The **Time-to-Live** (TTL) field is decremented by 1 at each router; if it reaches 0, the packet is discarded. A typical initial value is 64; it must be larger than the total number of hops in the path. In most cases, a value of 32 would work. The TTL field is there to prevent routing loops – always a serious problem should they occur – from consuming resources indefinitely. Later we will look at various IP routing-table update protocols and how they minimize the risk of routing loops; they do not, however, eliminate it. By comparison, Ethernet headers have no TTL field, but Ethernet also disallows cycles in the underlying topology.

The **Protocol** field contains a value to identify the contents of the packet body. A few of the more common values are

- 1: an ICMP packet, [10.4 Internet Control Message Protocol](#)
- 4: an encapsulated IPv4 packet, [9.9.1 IP-in-IP Encapsulation](#)
- 6: a TCP packet
- 17: a UDP packet
- 41: an encapsulated IPv6 packet, [12.6 IPv6 Connectivity via Tunneling](#)
- 50: an Encapsulating Security Payload, [29.6 IPsec](#)

A list of assigned protocol numbers is maintained by the [IANA](#).

The **Header Checksum** field is the “Internet checksum” applied to the header only, not the body. Its only purpose is to allow the discarding of packets with corrupted headers. When the TTL value is decremented the router must update the header checksum. This can be done “algebraically” by adding a 1 in the correct place to compensate, but it is not hard simply to re-sum the 8 halfwords of the average header. The header checksum must also be updated when an IPv4 packet header is rewritten by a NAT router.

The **Source** and **Destination Address** fields contain, of course, the IPv4 addresses. These would normally be updated only by NAT firewalls.

The source-address field is supposed to be the sender’s IPv4 address, but hardly any ISP checks that traffic they send out has a source address matching one of their customers, despite the call to do so in [RFC 2827](#). As a result, IP **spoofing** – the sending of IP packets with a faked source address – is straightforward. For some examples, see [18.3.1 ISNs and spoofing](#), and SYN flooding at [17.3 TCP Connection Establishment](#).

IP-address spoofing also facilitates an all-too-common IP-layer **denial-of-service** attack in which a server is flooded with a huge volume of traffic so as to reduce the bandwidth available to legitimate traffic to a trickle. This flooding traffic typically originates from a

large number of compromised machines. Without spoofing, even a lengthy list of sources can be blocked, but, with spoofing, this becomes quite difficult.

One IPv4 option is the **Record Route** option, in which routers are to insert their own IPv4 address into the IPv4 header option area. Unfortunately, with only ten words available, there is not enough space to record most longer routes (but see [10.4.1 Traceroute and Time Exceeded](#), below). The **Timestamp** option is related; intermediate routers are requested to mark packets with their address and a local timestamp (to save space, the option can request only timestamps). There is room for only four (address,timestamp) pairs, but addresses can be **prespecified**; that is, the sender can include up to four IPv4 addresses and only those routers will fill in a timestamp.

Another option, now deprecated as security risk, is to support **source routing**. The sender would insert into the IPv4 header option area a list of IPv4 addresses; the packet would be routed to pass through each of those IPv4 addresses in turn. With **strict** source routing, the IPv4 addresses had to represent adjacent neighbors; no router could be used if its IPv4 address were not on the list. With **loose** source routing, the listed addresses did not have to represent adjacent neighbors and ordinary IPv4 routing was used to get from one listed IPv4 address to the next. Both forms are essentially never used, again for security reasons: if a packet has been source-routed, it may have been routed outside of the at-least-somewhat trusted zone of the Internet backbone.

Finally, the IPv4 header was carefully laid out with memory alignment in mind. The 4-byte address fields are aligned on 4-byte boundaries, and the 2-byte fields are aligned on 2-byte boundaries. All this was once considered important enough that incoming packets were stored following two bytes of padding at the head of their containing buffer, so the IPv4 header, starting after the 14-byte Ethernet header, would be aligned on a 4-byte boundary. Today, however, the architectures for which this sort of alignment mattered have mostly faded away; alignment is a non-issue for [ARM](#) and Intel [x86](#) processors.

9.2 Interfaces

IP addresses (both IPv4 and IPv6) are, strictly speaking, assigned not to hosts or nodes, but to **interfaces**. In the most common case, where each node has a single LAN interface, this is a distinction without a difference. In a room full of workstations each with a single Ethernet interface `eth0` (or perhaps Ethernet adapter Local Area Connection), we might as well view the IP address assigned to the interface as assigned to the workstation itself.

Each of those workstations, however, likely also has a **loopback** interface (at least conceptually), providing a way to deliver IP packets to other processes on the same machine. On many systems, the name “localhost” resolves to the IPv4 loopback address 127.0.0.1 (the IPv6 address `::1` is also used); see [9.3 Special Addresses](#). Delivering

packets to the loopback interface is simply a form of interprocess communication; a functionally similar alternative is [named pipes](#).

Loopback delivery avoids the need to use the LAN at all, or even the need to *have* a LAN. For simple client/server testing, it is often convenient to have both client and server on the same machine, in which case the loopback interface is a convenient (and fast) standin for a “real” network interface. On unix-based machines the loopback interface represents a genuine logical interface, commonly named `lo`. On Windows systems the “interface” may not represent an actual operating-system entity, but this is of practical concern only to those interested in “sniffing” all loopback traffic; packets sent to the loopback address are still delivered as expected.

Workstations often have special other interfaces as well. Most recent versions of Microsoft Windows have a Teredo Tunneling pseudo-interface and an Automatic Tunneling pseudo-interface; these are both intended (when activated) to support IPv6 connectivity when the local ISP supports only IPv4. The Teredo protocol is documented in [RFC 4380](#).

When VPN connections are created, as in [5.1 Virtual Private Networks](#), each end of the logical connection typically terminates at a virtual interface (one of these is labeled `tun0` in the diagram of [5.1 Virtual Private Networks](#)). These virtual interfaces appear, to the systems involved, to be attached to a point-to-point link that leads to the other end.

When a computer hosts a virtual machine, there is almost always a virtual network to connect the host and virtual systems. The host will have a virtual interface to connect to the virtual network. The host may act as a NAT router for the virtual machine, “hiding” that virtual machine behind its own IP address, or it may act as an Ethernet switch, in which case the virtual machine will need an additional public IP address.

What’s My IP Address?

This simple-seeming question is in fact not very easy to answer, if by “my IP address” one means the IP address assigned to the interface that connects directly to the Internet. One strategy is to find the address of the default router, and then iterate through all interfaces (*eg* with the Java `NetworkInterface` class) to find an IP address with a matching network prefix; a Python3 example of this approach appears in [30.5.1 Multicast Programming](#). Unfortunately, finding the default router (to identify the primary interface) is hard to do in an OS-independent way, and even then this approach can fail if the Wi-Fi and Ethernet interfaces both are assigned IP addresses on the same network, but only one is actually connected.

Routers always have at least two interfaces on two separate IP networks. Generally this means a separate IP address for each interface, though some point-to-point interfaces can be used without being assigned any IP address ([9.8 Unnumbered Interfaces](#)).

9.2.1 Multihomed hosts

A non-router host with multiple non-loopback network interfaces is often said to be **multihomed**. Many laptops, for example, have both an Ethernet interface and a Wi-Fi interface. Both of these can be used simultaneously, with different IP addresses assigned to each. On residential networks the two interfaces will often be on the same IP network (*eg* the same bridged Wi-Fi/Ethernet LAN); at more security-conscious sites the Ethernet and Wi-Fi interfaces are often on quite different IP networks (though see [10.2.5 ARP and multihomed hosts](#)).

Multiple physical interfaces are not actually needed here; it is usually possible to assign multiple IP addresses to a *single* interface. Sometimes this is done to allow two IP networks (two distinct prefixes) to share a single physical LAN; in this case the interface would be assigned one IP address for each IP network. Other times a single interface is assigned multiple IP addresses on the same IP network; this is often done so that one physical machine can act as a server (*eg* a web server) for multiple distinct IP addresses corresponding to multiple distinct domain names.

Multihoming raises some issues with packets addressed to one interface, A, with IP address A_{IP} , but which arrive via another interface, B, with IP address B_{IP} . Strictly speaking, such arriving packets should be discarded unless the host is promoted to functioning as a router. In practice, however, the strict interpretation often causes problems; a typical user understanding is that the IP address A_{IP} should work to reach the host even if the physical connection is to interface B. A related issue is whether the host receiving such a packet addressed to A_{IP} on interface B is allowed to send its *reply* with source address A_{IP} , even though the reply must be sent via interface B.

[RFC 1122](#), §3.3.4, defines two alternatives here:

- The **Strong End-System** model: IP addresses – incoming and outbound – must match the physical interface.
- The **Weak End-System** model: A match is not required: interface B can accept packets addressed to A_{IP} , and send packets with source address A_{IP} .

Linux systems generally use the weak model by default. See also [10.2.5 ARP and multihomed hosts](#).

While it is important to be at least vaguely aware of the special cases that multihoming presents, we emphasize again that in most ordinary contexts each end-user workstation has one IP address that corresponds to a LAN connection.

9.3 Special Addresses

A few IPv4 addresses represent special cases.

While the standard IPv4 loopback address is 127.0.0.1, any IPv4 address beginning with 127 can serve as a loopback address. Logically they all represent the current host. Most hosts are configured to resolve the name “localhost” to 127.0.0.1. However, any loopback address – *eg* 127.255.37.59 – should work, *eg* with ping. For an example using 127.0.1.0, see [10.1 DNS](#).

Private addresses are IPv4 addresses intended only for site internal use, *eg* either behind a NAT firewall or intended to have no Internet connectivity at all. If a packet shows up at any non-private router (*eg* at an ISP router), with a private IPv4 address as either source or destination address, the packet should be dropped. Three standard private-address blocks have been defined:

- 10.0.0.0/8
- 172.16.0.0/12
- 192.168.0.0/16

The last block is the one from which addresses are most commonly allocated by DHCP servers ([10.3.1 NAT, DHCP and the Small Office](#)) built into NAT routers.

There are subtle issues with private addresses. First of all, when organizations merge, wholesale private-address renumbering is usually necessary. Second, suppose Alice uses 10.0.0.0/8 for her home network, where her laptop is 10.0.0.23. Suppose also that Alice connects via a VPN to work, and her server at work is also 10.0.0.23. Connection between laptop and server will then fail. It would also fail if the server were 10.0.0.24, as Alice’s laptop will think that should be on the local subnet, and it is not. There are other potential conflicts as well. Perhaps the main reason these problems are not worse than they are is that most home networks use 192.168.0.0/16, and most corporate networks use one of the other two private-address blocks.

There is an additional problem with mobile-phone networks. Most phones get a mobile-network IPv4 address from the carrier, and also a Wi-Fi IPv4 address from whatever Wi-Fi network the phone owner is currently connected to. If the carrier uses any of the above private-address blocks, there is a fair chance that, at some Wi-Fi-providing establishment, someone’s carrier-assigned mobile IPv4 address will conflict with their Wi-Fi address. The addresses may even be the same. Because of this, [RFC 6598](#) has established the following special address block for mobile-device carriers, known as a **shared-address** block:

- 100.64.0.0/10

It is exactly like a private-address block, except no one is to use it except mobile-device carriers.

Broadcast addresses are a special form of IPv4 address intended to be used in conjunction with LAN-layer broadcast. The most common forms are “broadcast to this

network”, consisting of all 1-bits, and “broadcast to network D”, consisting of D’s network-address bits followed by all 1-bits for the host bits. If you try to send a packet to the broadcast address of a remote network D, the odds are that some router involved will refuse to forward it, and the odds are even higher that, once the packet arrives at a router actually on network D, that router will refuse to broadcast it. Even addressing a broadcast to one’s own network will fail if the underlying LAN does not support LAN-level broadcast (*eg* ATM).

The highly influential early Unix implementation Berkeley 4.2 BSD used 0-bits for the broadcast bits, instead of 1’s. As a result, to this day host bits cannot be all 1-bits or all 0-bits in order to avoid confusion with the IPv4 broadcast address. One consequence of this is that a Class C network has 254 usable host addresses, not 256.

9.3.1 Multicast addresses

Finally, **IPv4 multicast addresses** remain as the last remnant of the Class A/B/C strategy: multicast addresses are Class D, with first byte beginning 1110 (meaning that the first byte is, in decimal, 224–239). Multicasting means delivering to a specified *set* of addresses, preferably by some mechanism more efficient than sending to each address individually. A reasonable goal of multicast would be that no more than one copy of the multicast packet traverses any given link.

Support for IPv4 multicast requires considerable participation by the backbone routers involved. For example, if hosts A, B and C each connect to different interfaces of router R1, and A wishes to send a multicast packet to B and C, then it is up to R1 to receive the packet, figure out that B and C are the intended recipients, and forward the packet *twice*, once for B’s interface and once for C’s. R1 must also keep track of what hosts have joined the **multicast group** and what hosts have left. Due to this degree of router participation, backbone router support for multicasting has not been entirely forthcoming. A discussion of IPv4 multicasting appears in [25 Quality of Service](#).

9.4 Fragmentation

If you are trying to interconnect two LANs (as IP does), what else might be needed besides Routing and Addressing? IPv4 (and IPv6) explicitly assumes all packets are composed on 8-bit bytes (something not universally true in the early days of IP; to this day the RFCs refer to “octets” to emphasize this requirement). IP also defines bit-order within a byte, and it is left to the networking hardware to translate properly. Neither byte size nor bit order, therefore, can interfere with packet forwarding.

There is one more feature IPv4 must provide, however, if the goal is universal connectivity: it must accommodate networks for which the maximum packet size, or **Maximum Transfer Unit**, MTU, is smaller than the packet that needs forwarding. Otherwise, if we were using IPv4 to join Token Ring (MTU = 4kB, at least originally) to Ethernet (MTU = 1500B), the token-ring packets might be too large to deliver to the

Ethernet side, or to traverse an Ethernet backbone *en route* to another Token Ring. (Token Ring, in its day, did commonly offer a configuration option to allow Ethernet interoperability.)

So, IPv4 must support fragmentation, and thus also reassembly. There are two potential strategies here: **per-link** fragmentation and reassembly, where the reassembly is done at the opposite end of the link (as in ATM), and **path** fragmentation and reassembly, where reassembly is done at the far end of the path. The latter approach is what is taken by IPv4, partly because intermediate routers are too busy to do reassembly (this is as true today as it was in 1981 when [RFC 791](#) was published), partly because there is no absolute guarantee that all fragments will go to the same next-hop router, and partly because IPv4 fragmentation has always been seen as the strategy of last resort.

An IPv4 sender is supposed to use a different value for the **IDENT** field for different packets, at least up until the field wraps around. When an IPv4 datagram is fragmented, the fragments keep the same IDENT field, so this field in effect indicates which fragments belong to the same packet.

After fragmentation, the **Fragment Offset** field marks the start position of the data portion of this fragment within the data portion of the original IPv4 packet. Note that the start position can be a number up to 2^{16} , the maximum IPv4 packet length, but the FragOffset field has only 13 bits. This is handled by requiring the data portions of fragments to have sizes a multiple of 8 (three bits), and left-shifting the FragOffset value by 3 bits before using it.

As an example, consider the following network, where MTUs are excluding the LAN header:

Suppose A addresses a packet of 1500 bytes to B, and sends it via the LAN to the first router R1. The packet contains 20 bytes of IPv4 header and 1480 of data.

R1 fragments the original packet into two packets of sizes $20+976 = 996$ and $20+504=524$. Having 980 bytes of payload in the first fragment would fit, but violates the rule that the sizes of the data portions be divisible by 8. The first fragment packet has FragOffset = 0; the second has FragOffset = 976.

R2 refragments the first fragment into three packets as follows:

- first: size = $20+376=396$, FragOffset = 0
- second: size = $20+376=396$, FragOffset = 376
- third: size = $20+224 = 244$ (note $376+376+224=976$), FragOffset = 752.

R2 refragments the second fragment into two:

- first: size = $20+376 = 396$, FragOffset = $976+0 = 976$
- second: size = $20+128 = 148$, FragOffset = $976+376=1352$

R3 then sends the fragments on to B, without reassembly.

Note that it would have been slightly more efficient to have fragmented into four fragments of sizes 376, 376, 376, and 352 in the beginning. Note also that the packet format is designed to handle fragments of different sizes easily. The algorithm is based on multiple fragmentation with reassembly only at the final destination.

Each fragment has its IPv4-header Total Length field set to the length of that fragment.

We have not yet discussed the three flag bits. The first bit is reserved, and must be 0. The second bit is the **Don't Fragment**, or DF, bit. If it is set to 1 by the sender then a router must *not* fragment the packet and must drop it instead; see [18.6 Path MTU Discovery](#) for an application of this. The third bit, the **More Fragments** bit, is set to 1 for all fragments *except* the final one (this bit is thus set to 0 if no fragmentation has occurred). The third bit tells the receiver where the fragments stop.

That first flag bit

[RFC 3514](#) proposes the use of the first flag bit here as a security flag: it would be set to 0 for legitimate traffic, while senders of malicious and hacking traffic were supposed to set the bit to 1. This would make detection and firewalling of malicious traffic much easier. The publication date of this RFC is April 1, or April Fool's Day; there is a long tradition of RFC humor released that day. Another example is [RFC 2324](#).

The receiver must take the arriving fragments and **reassemble** them into a whole packet. The fragments may not arrive in order – unlike in ATM networks – and may have unrelated packets interspersed. The reassembler must identify when different arriving packets are fragments of the same original, and must figure out how to reassemble the fragments in the correct order; both these problems were essentially trivial for ATM.

Fragments are considered to belong to the same packet if they have the same IDENT field and also the same source and destination addresses and same protocol.

As all fragment sizes are a multiple of 8 bytes, the receiver can keep track of whether all fragments have been received with a bitmap in which each bit represents one 8-byte fragment chunk. A 1 kB packet could have up to 128 such chunks; the bitmap would thus be 16 bytes.

If a fragment arrives that is part of a new (and fragmented) packet, a buffer is allocated. While the receiver cannot know the final size of the buffer, it can usually make a reasonable guess. Because of the FragOffset field, the fragment can then be stored in the buffer in the appropriate position. A new bitmap is also allocated, and a **reassembly timer** is started.

As subsequent fragments arrive, not necessarily in order, they too can be placed in the proper buffer in the proper position, and the appropriate bits in the bitmap are set to 1.

If the bitmap shows that all fragments have arrived, the packet is sent on up as a completed IPv4 packet. If, on the other hand, the reassembly timer expires, then all the pieces received so far are discarded.

TCP connections usually engage in **Path MTU Discovery**, and figure out the largest packet size they can send that will *not* entail fragmentation (18.6 **Path MTU Discovery**). But it is not unusual, for example, for UDP protocols to use fragmentation, especially over the short haul. In the Network File System (NFS) protocol, for example, UDP is used to carry 8 kB disk blocks. These are often sent as a single 8+ kB IPv4 packet, fragmented over Ethernet to five full packets and a fraction. Fragmentation works reasonably well here because most of the time the packets do not leave the Ethernet they started on. Note that this is an example of fragmentation done by the *sender*, not by an intermediate router.

Finally, any given IP link may provide its own link-layer fragmentation and reassembly; we saw in 5.5.1 **ATM Segmentation and Reassembly** that ATM does just this. Such link-layer mechanisms are, however, generally invisible to the IP layer.

9.5 The Classless IP Delivery Algorithm

Recall from Chapter 1 that any IPv4 address can be divided into a net portion IP_{net} and a host portion IP_{host} ; the division point was determined by whether the IPv4 address was a Class A, a Class B, or a Class C. We also indicated in Chapter 1 that the division point was not always so clear-cut; we now present the delivery algorithm, for both hosts and routers, that does *not* assume a globally predeclared division point of the input IPv4 address into net and host portions. We will, for the time being, punt on the question of forwarding-table lookup and assume there is a `lookup()` method available that, when given a destination address, returns the `next_hop` neighbor.

Instead of class-based divisions, we will assume that each of the IPv4 addresses assigned to a node's interfaces is configured with an associated length of the network prefix; following the slash notation of 1.10 **IP – Internet Protocol**, if B is an address and the prefix length is $k = k_B$ then the prefix itself is B/k . As usual, an ordinary host may have only one IP interface, while a router will always have multiple interfaces.

Let D be the given IPv4 destination address; we want to decide if D is **local** or **nonlocal**. The host or router involved may have multiple IP interfaces, but for each interface the length of the network portion of the address will be known. For each network address B/k assigned to one of the host's interfaces, we compare the first k bits of B and D ; that is, we ask if D **matches** B/k .

- If one of these comparisons yields a match, delivery is **local**; the host delivers the packet to its final destination via the LAN connected to the corresponding interface. This means looking up the LAN address of the destination, if applicable, and sending the packet to that destination via the interface.
- If there is no match, delivery is **nonlocal**, and the host passes D to the `lookup()` routine of the forwarding table and sends to the associated `next_hop` (which must represent a physically connected neighbor). It is now up to `lookup()` routine to make any necessary determinations as to how D might be split into D_{net} and D_{host} ; the split *cannot* be made outside of `lookup()`.

The forwarding table is, abstractly, a set of network addresses – now also with lengths – each of the form B/k , with an associated `next_hop` destination for each.

The `lookup()` routine will, in principle, compare D with each table entry B/k , looking for a match (that is, equality of the first $k = k_B$ bits). As with the local-delivery interfaces check above, the net/host division point (that is, k) will come from the table entry; it will not be inferred from D or from any other information borne by the packet. There is, in fact, no place in the IPv4 header to store a net/host division point, and furthermore different routers along the path may use different values of k with the same destination address D . Routers receive the prefix length $/k$ for a destination B/k as part of the process by which they receive $\langle \text{destination}, \text{next_hop} \rangle$ pairs; see [13 Routing-Update Algorithms](#).

In [14 Large-Scale IP Routing](#) we will see that in some cases multiple matches in the forwarding table may exist, *eg* $147.0.0.0/8$ and $147.126.0.0/16$. The **longest-match** rule will be introduced for such cases to pick the best match.

Here is a simple example for a router with immediate neighbors A–E:

destination	next_hop
10.3.0.0/16	A
10.4.1.0/24	B
10.4.2.0/24	C
10.4.3.0/24	D
10.3.37.0/24	E

The IPv4 addresses $10.3.67.101$ and $10.3.59.131$ both route to A. The addresses $10.4.1.101$, $10.4.2.157$ and $10.4.3.233$ route to B, C and D respectively. Finally, $10.3.37.103$ matches both A and E, but the E match is longer so the packet is routed that way.

The forwarding table may also contain a **default** entry for the `next_hop`, which it may return in cases when the destination D does not match any known network. We take the view here that returning such a default entry is a valid result of the routing-table `lookup()` operation, rather than a third option to the algorithm above; one approach is for the default entry to be the `next_hop` corresponding to the destination

0.0.0.0/0, which does indeed match everything (use of this would definitely require the above longest-match rule, though).

Default routes are hugely important in keeping leaf forwarding tables small. Even backbone routers sometimes expend considerable effort to keep the network address prefixes in their forwarding tables as short as possible, through consolidation.

At a site with a single ISP and with no Internet customers (that is, which is not itself an ISP for others), the top-level forwarding table usually has a single external route: its default route to its ISP. If a site has more than one ISP, however, the top-level forwarding table can expand in a hurry. For example, [Internet2](#) is a consortium of research sites with very-high-bandwidth internal interconnections, acting as a sort of “parallel Internet”. Before Internet2, Loyola’s top-level forwarding table had the usual single external default route. After Internet2, we in effect had a second ISP and had to divide traffic between the commercial ISP and the Internet2 ISP. The default route still pointed to the commercial ISP, but Loyola’s top-level forwarding table now had to have an entry for every individual Internet2 site, so that traffic to any of these sites would be forwarded via the Internet2 ISP. See exercise 5.0.

Routers may also be configured to allow passing quality-of-service information to the `lookup()` method, as mentioned in Chapter 1, to support different routing paths for different kinds of traffic (*eg* bulk file-transfer versus real-time).

For a modest exception to the local-delivery rule described here, see below in [9.8 Unnumbered Interfaces](#).

9.5.1 Efficient Forwarding-Table Lookup

Fast implementation of the `lookup()` operation above is tricky, especially in the presence of destination entries that may not result in unique matches, such as 10.0.0.0/8 and 10.11.0.0/16, which both match 10.11.12.13. Straightforward hashing, in particular, is out, as the prefix-length value k is not available to the call of `lookup()`.

The simplest approach is a **trie**, a form of tree in which the child nodes are labeled with bit strings; the concatenated node labels on a branch represent an address prefix. Node labels can represent single address bits or larger bit groups. A trie allows straightforward implementation of the longest-match rule by requiring that we descend in the trie until no further matches are possible.

As an example, we construct a trie for the following forwarding table:

destination	next_hop
1.10.0.0/16	A
1.10.104.0/24	B
1.10.105.0/24	C
1.11.0.0/16	D
1.12.116.0/24	E

destination	next_hop
1.12.117.0/24	F

As all the prefix lengths are multiples of 8 bits, we build the trie using bytes as node labels:

Heavy blue nodes represent matches; light nodes represent non-matches. There is one heavy node for each entry in the table above. To look up 1.10.105.213 in the trie, we view the address as a list (1,10,105,213). From the root, we traverse nodes 1 and 10, arriving at a heavy node representing the destination 1.10.0.0/16. However, the next element of the address list is 105, and so we continue down to child node 105, thus matching 1.10.105.0/24. There are no further child nodes, so this match is as long as possible.

A straightforward trie implementation for arbitrary prefix lengths requires single-bit node labels. However, the Luleå algorithm, [DBCP97], implements lookup with a trie of only three levels, representing address bits 0–15, 16–23 and 24–31. At the top of the trie, an array of size 2^{16} helps determine the first child node; similar supplemental data structures assist with the child-node lookups at subsequent levels.

Finally, high-performance switches often implement the `lookup()` operation using **content-addressable memory**. IP forwarding requires the ternary form of this memory, TCAM, described in the final paragraph of 3.2.1 **Switch Hardware**. If A/k is an IP address prefix of length k , then A goes in the TCAM memory register, and the corresponding mask register is used to indicate that only the first k bits matter.

When an IPv4 address is presented to the TCAM for lookup, there may now be multiple matches of differing lengths. To implement the longest-match rule, we first need to make sure that, in the TCAM sequence of registers, shorter prefixes come before longer ones. The TCAM encoder circuit then converts the matching register's position in the sequence to an address corresponding to the register, with which the rest of the routing information can be retrieved. This encoder must be designed so that it always prefers longer prefixes (higher TCAM sequence positions). A longest-match lookup can then be performed in a single memory-lookup cycle.

When backbone IPv4 routers experience acute difficulties due to growth of the forwarding table, it is often because the existing table has outgrown the space available in the TCAM hardware.

9.6 IPv4 Subnets

Subnets were the first step away from Class A/B/C routing: a large network (*eg* a class A or B) could be divided into smaller IPv4 networks called subnets. Consider, for example, a typical Class B network such as Loyola University's (originally 147.126.0.0/16); the

underlying assumption is that any packet can be delivered via the underlying LAN to any internal host. This would require a rather large LAN, and would require that a single physical LAN be used throughout the site. What if our site has more than one physical LAN? Or is really too big for one physical LAN? It did not take long for the IP world to run into this problem.

Subnets were first proposed in [RFC 917](#), and became official with [RFC 950](#).

Getting a separate IPv4 network prefix for each subnet is bad for routers: the backbone forwarding tables now must have an entry for every subnet instead of just for every site. What is needed is a way for a site to appear to the outside world as a single IP network, but for further IP-layer routing to be supported *inside* the site. This is what subnets accomplish.

Subnets introduce **hierarchical routing**: first we route to the primary network, then inside that site we route to the subnet, and finally the last hop delivers to the host.

Routing with subnets involves in effect moving the IP_{net} division line rightward. (Later, when we consider CIDR, we will see the complementary case of moving the division line to the left.) For now, observe that moving the line rightward within a site does not affect the outside world at all; outside routers are not even aware of site-internal subnetting.

In the following diagram, the outside world directs traffic addressed to 147.126.0.0/16 to the router R. Internally, however, the site is divided into subnets. The idea is that traffic from 147.126.1.0/24 to 147.126.2.0/24 is routed, not switched; the two LANs involved may not even be compatible (for example, the ovals might represent Token Ring while the lines represent Ethernet). Most of the subnets shown are of size /24, meaning that the third byte of the IPv4 address has become part of the network portion of the subnet's address; one /20 subnet is also shown. [RFC 950](#) would have disallowed the subnet with third byte 0, but having 0 for the subnet bits generally does work.

What we want is for the internal routing to be based on the extended network prefixes shown, while externally continuing to use only the single routing entry for 147.126.0.0/16.

To implement subnets, we divide the site's IPv4 network into some combination of physical LANs – the subnets –, and assign each a **subnet address**: an IPv4 network address which has the *site's* IPv4 network address as prefix. To put this more concretely, suppose the site's IPv4 network address is A, and consists of n network bits (so the site address may be written with the slash notation as A/n); in the diagram above, A/n = 147.126.0.0/16. A subnet address is an IPv4 network address B/k such that:

- The address B/k is within the site: the first n bits of B are the same as A/n's

- B/k extends A/n: $k \geq n$

An example B/k in the diagram above is 147.126.1.0/24. (There is a slight simplification here in that subnet addresses do not absolutely *have* to be prefixes; see below.)

We now have to figure out how packets will be routed to the correct subnet. For incoming packets we could set up some proprietary protocol at the entry router to handle this. However, the more complicated situation is all those existing internal hosts that, under the class A/B/C strategy, would still believe they can deliver via the LAN to any site host, when in fact they can now only do that for hosts on their own subnet. We need a more general solution.

We proceed as follows. For each subnet address B/k, we create a **subnet mask** for B consisting of k 1-bits followed by enough 0-bits to make a total of 32. We then make sure that every host and router in the site knows the subnet mask for every one of its *interfaces*. Hosts usually find their subnet mask the same way they find their IP address (by static configuration if necessary, but more likely via DHCP, below).

Hosts and routers now apply the IP delivery algorithm of the previous section, with the proviso that, if a subnet mask for an interface is present, then the subnet mask is used to determine the number of address bits rather than the Class A/B/C mechanism. That is, we determine whether a packet addressed to destination D is deliverable locally via an interface with subnet address B/k and corresponding mask M by comparing $D \& M$ with $B \& M$, where $\&$ represents bitwise AND; if the two match, the packet is local. This will generally involve a match of *more* bits than if we used the Class A/B/C strategy to determine the network portion of addresses D and B.

As stated previously, given an address D with no other context, we will *not* be able to determine the network/host division point in general (*eg* for outbound packets). However, that division point is not in fact what we need. All that *is* needed is a way to tell if a given destination host address D belongs to the current subnet, say B; that is, we need to compare the first k bits of D and B where k is the (known) length of B.

In the diagram above, the subnet mask for the /24 subnets would be 255.255.255.0; bitwise ANDing any IPv4 address with the mask is the same as extracting the first 24 bits of the IPv4 address, that is, the subnet portion. The mask for the /20 subnet would be 255.255.240.0 (240 in binary is 1111 0000).

In the diagram above none of the subnets overlaps or conflicts: the subnets 147.126.0.0/24 and 147.126.1.0/24 are disjoint. It takes a little more effort to realize that 147.126.16.0/20 does not overlap with the others, but note that an IPv4 address matches this network prefix only if the first four bits of the third byte are 0001, so the third byte itself ranges from decimal 32 to decimal 63 = binary 0001 1111.

Note also that if host A = 147.126.0.1 wishes to send to destination D = 147.126.1.1, and A is *not* subnet-aware, then delivery will fail: A will infer that the interface is a Class

B, and therefore compare the first two bytes of A and D, and, finding a match, will attempt direct LAN delivery. But direct delivery is now likely impossible, as the subnets are not joined by a switch. Only with the subnet mask will A realize that its network is 147.126.0.0/24 while D's is 147.126.1.0/24 and that these are not the same.

A *would* still be able to send packets to its own subnet. In fact A would still be able to send packets to the outside world: it would realize that the destination in that case does not match 147.126.0.0/16 and will thus forward to its router. Hosts on other subnets would be the only unreachable ones.

Properly, the subnet address is the entire prefix, *eg* 147.126.65.0/24. However, it is often convenient to identify the subnet address with just those bits that represent the extension of the site IPv4-network address; we might thus say casually that the subnet address here is 65.

The class-based IP-address strategy allowed any host anywhere on the Internet to properly separate any address into its net and host portions. With subnets, this division point is now allowed to vary; for example, the address 147.126.65.48 divides into 147.126 | 65.48 outside of Loyola, but into 147.126.65 | 48 inside. This means that the net-host division is no longer an absolute property of addresses, but rather something that depends on where the packet is on its journey.

Technically, we also need the requirement that given any two subnet addresses of different, disjoint subnets, neither is a proper prefix of the other. This guarantees that if A is an IP address and B is a subnet address with mask M (so $B = B \& M$), then $A \& M = B$ implies A does not match any other subnet. Regardless of the net/host division rules, we cannot possibly allow subnet 147.126.16.0/20 to represent one LAN while 147.126.16.0/24 represents another; the second subnet address block is a subset of the first. (We *can*, and sometimes do, allow the first LAN to correspond to everything in 147.126.16.0/20 that is not also in 147.126.16.0/24; this is the longest-match rule.)

The strategy above is actually a slight simplification of what the subnet mechanism actually allows: subnet address bits do not in fact have to be contiguous, and masks do not have to be a series of 1-bits followed by 0-bits. The mask can be *any* bit-mask; the subnet address bits are by definition those where there is a 1 in the mask bits. For example, we could at a Class-B site use the *fourth* byte as the subnet address, and the *third* byte as the host address. The subnet mask would then be 255.255.0.255. While this generality was once sometimes useful in dealing with “legacy” IPv4 addresses that could not easily be changed, life is simpler when the subnet bits precede the host bits.

9.6.1 Subnet Example

As an example of having different subnet masks on different interfaces, let us consider the division of a class-C network into subnets of size 70, 40, 25, and 20. The subnet addresses will of necessity have different lengths, as there is not room for four subnets each able to hold 70 hosts.

- A: size 70
- B: size 40
- C: size 25
- D: size 20

Because of the different subnet-address lengths, division of a local IPv4 address LA into net versus host on subnets cannot be done in isolation, without looking at the host bits. However, that division is not in fact what we need. All that is needed is a way to tell if the local address LA belongs to a given subnet, say B; that is, we need to compare the first n bits of LA and B, where n is the length of B's subnet mask. We do this by comparing $LA \& M$ to $B \& M$, where M is the mask corresponding to n . $LA \& M$ is not necessarily the same as LA_{net} , if LA actually belongs to one of the other subnets. However, if $LA \& M = B \& M$, then LA must belong subnet B, in which case $LA \& M$ is in fact LA_{net} .

We will assume that the site's IPv4 network address is 200.0.0.0/24. The first three bytes of each subnet address must match 200.0.0. Only some of the bits of the fourth byte will be part of the subnet address, so we will switch to binary for the last byte, and use both the $/n$ notation (for total number of subnet bits) and also add a vertical bar $|$ to mark the separation between subnet and host.

Example: 200.0.0.10 | 00 0000 / 26

Note that this means that the 0-bit following the 1-bit in the fourth byte is "significant" in that for a subnet to match, it must match this 0-bit exactly. The remaining six 0-bits are part of the host portion.

To allocate our four subnet addresses above, we start by figuring out just how many host bits we need in each subnet. Subnet sizes are always powers of 2, so we round up the subnets to the appropriate size. For subnet A, this means we need 7 host bits to accommodate $2^7 = 128$ hosts, and so we have a single bit in the fourth byte to devote to the subnet address. Similarly, for B we will need 6 host bits and will have 2 subnet bits, and for C and D we will need 5 host bits each and will have $8-5=3$ subnet bits.

We now start choosing non-overlapping subnet addresses. We have one bit in the fourth byte to choose for A's subnet; rather arbitrarily, let us choose this bit to be 1. This means that *every other subnet address* must have a 0 in the first bit position of the fourth byte, or we would have ambiguity.

Now for B's subnet address. We have two bits to work with, and the first bit must be 0. Let us choose the second bit to be 0 as well. If the fourth byte begins 00, the packet is part of subnet B, and the subnet addresses for C and D must therefore *not* begin 00.

Finally, we choose subnet addresses for C and D to be 010 and 011, respectively. We thus have

subnet	size	address bits in fourth byte	host bits in 4th byte	decimal range
A	128	1	7	128-255

subnet	size	address bits in fourth byte	host bits in 4th byte	decimal range
B	64	00	6	0-63
C	32	010	5	64-95
D	32	011	5	96-127

As desired, none of the subnet addresses in the third column is a prefix of any other subnet address.

The end result of all of this is that routing is now **hierarchical**: we route on the site IP address to get to a site, and then route on the subnet address within the site.

9.6.2 Links between subnets

Suppose the Loyola CS department subnet (147.126.65.0/24) and a department at some other site, we will say 147.100.100.0/24, install a private link. How does this affect routing?

Each department router would add an entry for the other subnet, routing along the private link. Traffic addressed to the other subnet would take the private link. All other traffic would go to the default router. Traffic from the remote department to 147.126.64.0/24 would take the long route, and Loyola traffic to 147.100.101.0/24 would take the long route.

Subnet anecdote

A long time ago I was responsible for two hosts, abel and borel. One day I was informed that machines in computer lab 1 at the other end of campus could not reach borel, though they could reach abel. Machines in lab 2, *adjacent* to lab 1, however, could reach both borel and abel just fine. What was the difference?

It turned out that borel had a bad (/16 instead of /24) subnet mask, and so it was attempting local delivery to the labs. This *should* have meant it could reach neither of the labs, as both labs were on a different subnet from my machines; I was still perplexed. After considerably more investigation, it turned out that between abel/borel and the lab building was a **bridge-router**: a hybrid device that properly routed subnet traffic at the IP layer, but which also forwarded Ethernet packets directly, the latter feature apparently for the purpose of backwards compatibility. Lab 2 was connected directly to the bridge-router and thus appeared to be on the same LAN as borel, despite the apparently different subnet; lab 1 was connected to its own router R1 which in turn connected to the bridge-router. Lab 1 was thus, at the LAN level, isolated from abel and borel.

Moral 1: Switching and routing are both great ideas, alone. But switching at one layer mixed with routing at another is not.

Moral 2: Test thoroughly! The reason the problem wasn't noticed earlier was that previously borel communicated only with other hosts on its own subnet and with hosts

outside the university entirely. Both of these worked with the bad subnet mask; it was different-subnet local hosts that were the problem.

How would nearby subnets at either endpoint decide whether to use the private link? Classical link-state or distance-vector theory (13 [Routing-Update Algorithms](#)) requires that they be able to compare the private-link route with the going-around-the-long-way route. But this requires a global picture of relative routing costs, which, as we shall see, almost certainly does not exist. The two departments are in different routing domains; if neighboring subnets at either end want to use the private link, then manual configuration is likely the only option.

9.6.3 Subnets versus Switching

A frequent network design question is whether to have many small subnets or to instead have just a few (or even only one) larger subnet. With multiple small subnets, IP routing would be used to interconnect them; the use of larger subnets would replace much of that routing with LAN-layer communication, likely Ethernet **switching**. Debates on this route-versus-switch question have gone back and forth in the networking community, with one aphorism summarizing a common view:

Switch when you can, route when you must

This aphorism reflects the idea that switching is faster, cheaper and easier to configure, and that subnet boundaries should be drawn only where “necessary”.

Ethernet switching equipment is indeed generally cheaper than routing equipment, for the same overall level of features and reliability. And traditional switching requires relatively little configuration, while to implement subnets not only must the subnets be created by hand but one must also set up and configure the routing-update protocols. However, the price difference between switching and routing is not always significant in the big picture, and the configuration involved is often straightforward.

Somewhere along the way, however, switching has acquired a reputation – often deserved – for being *faster* than routing. It is true that routers have more to do than switches: they must decrement TTL, update the header checksum, and attach a new LAN header. But these things are relatively minor: a larger reason many routers are slower than switches may simply be that they are inevitably *asked to serve as firewalls*. This means “deep inspection” of every packet, *eg* comparing every packet to each of a large number of firewall rules. The firewall may also be asked to keep track of connection state. All this drives down the forwarding rate, as measured in packets-per-second.

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

The rise of software-defined networking ([3.4 Software-Defined Networking](#)) has blurred the distinction between routing and switching. The term “Layer 3 switch” is sometimes used to describe routers that in effect do not support all the usual firewall bells and whistles. These are often SDN Ethernet switches ([3.4 Software-Defined Networking](#)) that are making forwarding decisions based on the contents of the IP header. Such streamlined switch/routers may also be able to do most of the hard work in specialized hardware, another source of speedup.

But SDN can do much more than IP-layer forwarding, by taking advantage of site-specific layout information. One application, of a switch hierarchy for traffic entering a datacenter, appears in [3.4.1 OpenFlow Switches](#). Other SDN applications include enabling Ethernet topologies with loops, offloading large-volume flows to alternative paths, and implementing policy-based routing as in [13.6 Routing on Other Attributes](#). Some SDN solutions involve site-specific programming, but others work more-or-less out of the box. Locations with switch-versus-route issues are likely to turn increasingly to SDN in the future.

9.7 Network Address Translation

What do you do if your ISP assigns to you a single IPv4 address and you have two computers? The solution is Network Address Translation, or **NAT**. NAT’s ability to “multiplex” an arbitrarily large number of individual hosts behind a single IPv4 address (or small number of addresses) makes it an important tool in the conservation of IPv4 addresses. It also, however, enables an important form of firewall-based security. It is documented in [RFC 3022](#), where this is called NAPT, or Network Address **Port** Translation. Another term in common use is **IP masquerading**.

The basic idea is that, instead of assigning each host at a site a publicly visible IPv4 address, just one such address is assigned to a special device known as a NAT router. A NAT router sold for residential or small-office use is commonly simply called a “router”, or (somewhat more precisely) a “residential gateway”. One side of the NAT router connects to the Internet; the other connects to the site’s internal network. Hosts on the internal network are assigned private IP addresses ([9.3 Special Addresses](#)), typically of the form or 192.168.x.y or 10.x.y.z. Connections to internal hosts that originate in the outside world are banned. When an internal machine wants to connect to the outside, the NAT router intercepts the connection, and forwards the connection’s packets after rewriting the source address to make it appear they came from the NAT router’s own IP address, shown below as 200.1.2.37.

The remote machine responds, sending its responses to the NAT router’s public IPv4 address. The NAT router remembers the connection, having stored the connection

information in a special forwarding table, and forwards the data to the correct internal host, rewriting the destination-address field of the incoming packets.

The NAT forwarding table also includes port numbers. That way, if two internal hosts attempt to connect to the same external host, the NAT router can tell which packets belong to which. For example, suppose internal hosts A and B each connect from port 3000 to port 80 on external hosts S and T, respectively. Here is what the NAT forwarding table might look like. No columns for the NAT router's own IPv4 addresses are needed; we shall let NR denote the router's external address.

remote host	remote port	outside source port	inside host	inside port
S	80	3000	A	3000
T	80	3000	B	3000

A packet to S from $\langle A, 3000 \rangle$ would be rewritten so that the source was $\langle NR, 3000 \rangle$. A packet from $\langle S, 80 \rangle$ addressed to $\langle NR, 3000 \rangle$ would be rewritten and forwarded to $\langle A, 3000 \rangle$. Similarly, a packet from $\langle T, 80 \rangle$ addressed to $\langle NR, 3000 \rangle$ would be rewritten and forwarded to $\langle B, 3000 \rangle$; the NAT table takes into account the source host and port as well as the destination.

Sometimes it is necessary for the NAT router to rewrite the internal-side port number as well; this happens if two internal hosts want to connect, each from the *same* port, to the same external host and port. For example, suppose B now opens a connection to $\langle S, 80 \rangle$, also from inside port 3000. This time the NAT router must remap the port number, because that is the only way to distinguish between packets from $\langle S, 80 \rangle$ back to A and to B. With B's second connection's internal port remapped from 3000 to 3001, the new table is

remote host	remote port	outside source port	inside host	inside port
S	80	3000	A	3000
T	80	3000	B	3000
S	80	3001	B	3000

The NAT router does not create TCP connections between itself and the external hosts; it simply forwards packets (with rewriting). The connection endpoints are still the external hosts S and T and the internal hosts A and B. However, NR might very well *monitor* the TCP connections to know when they have closed (by looking for FIN packets, [17.8.1 Closing a connection](#)), and so can be removed from the table. For UDP connections, NAT routers typically remove the forwarding entry after some period of inactivity; see [16 UDP Transport](#), exercise 16.0.

NAT still works for *some* traffic without port numbers, such as network pings, though the above table is then not quite the whole story. See [10.4 Internet Control Message Protocol](#).

Done properly, NAT improves the security of a site, by making it impossible for an external host to probe or to initiate a connection to any of the internal hosts. While this

firewall feature is of great importance, essentially the same effect can be achieved without address translation, and *with* public IPv4 addresses for all internal hosts, by having the router refuse to forward incoming packets that are not part of existing connections. The router still needs to maintain a table like the NAT table above, in order to recognize such packets. The address translation itself, in other words, is not the source of the firewall security. That said, it is hard for a NAT router to “fail open”; *ie* to fail in a way that lets outside connections in. It is much easier for a non-NAT firewall to fail open.

For the common residential form of NAT router, see [10.3.1 NAT, DHCP and the Small Office](#).

9.7.1 NAT Problems

NAT router’s refusal to allow inbound connections is a source of occasional frustration. We illustrate some of these frustrations here, using Voice-over-IP (VoIP) and the call-setup protocol SIP ([RFC 3261](#)). The basic strategy is that each phone is associated with a remote **phone server**. These phone servers, because they have to be able to accept incoming connections from anywhere, must not be behind NAT routers. The phones themselves, however, usually will be:

For phone1 to call phone2, phone1 first contacts Server1, which then contacts Server2. So far, all is well. The final step is for Server2 to contact phone2, which, however, cannot be done normally as NAT2 allows no inbound connections.

One common solution is for phone2 to maintain a persistent connection to Server2 (and ditto for phone1 and Server1). By having these persistent phone-to-server connections, we can arrange for the phone to ring on incoming calls.

As a second issue, somewhat particular to the SIP protocol, is that it is common for server and phone to prefer to use UDP port 5060 at *both* ends. For a single internal phone, it is likely that port 5060 will pass through without remapping, so the phone will appear to be connecting from the desired port 5060. However, if there are two phones inside (not shown above), one of them will appear to be connecting to the server *from* an alternative port. The solution here is to have the server tolerate such port remapping.

VoIP systems run into a much more serious problem with NAT, however. Once the call between phone1 and phone2 is set up, the servers would prefer to step out of the loop, and have the phones exchange voice packets directly. The SIP protocol was designed to handle this by having each phone report to its respective server the UDP socket (⟨IP address,port⟩ pair) it intends to use for the voice exchange; the servers then report these phone sockets to each other, and from there to the opposite phones. This socket information is rendered incorrect by NAT, however, certainly the IP address and quite likely the port as well. If only one of the phones is behind a NAT firewall, it can initiate

the voice connection to the other phone, but the other phone will see the voice packets arriving from a different socket than promised and will likely not recognize them as part of the call. If both phones are behind NAT firewalls, they may not be able to connect directly to one another at all. The common solution is for the VoIP server of a phone behind a NAT firewall to remain in the communications path, forwarding packets to its hidden partner. This works, but represents an unwanted server workload.

If a site wants to make it possible to allow external connections to hosts behind a NAT router or other firewall, one option is **tunneling**. This is the creation of a “virtual LAN link” that runs on top of a TCP connection between the end user and one of the site’s servers; the end user can thus appear to be on one of the organization’s internal LANs; see [5.1 Virtual Private Networks](#). Another option is to “open up” a specific port: in essence, a static NAT-table entry is made connecting a specific port on the NAT router to a specific internal host and port (usually the same port). For example, all UDP packets to port 5060 on the NAT router might be forwarded to port 5060 on internal host A, even in the absence of any prior packet exchange. Gamers creating peer-to-peer game connections must also usually engage in some port-opening configuration. The Port Control Protocol ([RFC 6887](#)) is sometimes used for this. See also [9.7.3 NAT Traversal](#), below.

NAT routers work very well when the communications model is of client-side TCP connections, originating from the inside and with public outside servers as destination. The NAT model works less well for peer-to-peer networking, as with the gamers above, where two computers, each behind a different NAT router, wish to establish a connection. Most NAT routers provide at least limited support for “opening” access to a given internal (host,port) socket, by creating a semi-permanent forwarding-table entry. See also [17.10 Exercises](#), exercise 3.0.

NAT routers also often have trouble with UDP protocols, due to the tendency for such protocols to have the public server reply from a *different* port than the one originally contacted. For example, if host A behind a NAT router attempts to use TFTP ([16.2 Trivial File Transport Protocol, TFTP](#)), and sends a packet to port 69 of public server C, then C is likely to reply from some *new* port, say 3000, and this reply is likely to be dropped by the NAT router as there will be no entry there yet for traffic from (C,3000).

9.7.2 Middleboxes

Firewalls and NAT routers are sometimes classed as **middleboxes**: network devices that block, throttle or modify traffic beyond what is necessary for basic forwarding.

Middleboxes play a very important role in network security, but they sometimes (as here with VoIP) break things. The word “middlebox” (versus “router” or “firewall”) usually has a perjorative connotation; middleboxes have, in some circles, acquired a rather negative reputation.

NAT routers' interference with VoIP, above, is a direct consequence of their function: NAT handles connections from inside to outside quite well, but the NAT mechanism offers no support for connections from one inside to another inside. Sometimes, however, middleboxes block traffic when there is no technical reason to do so, simply because correct behavior has not been widely implemented. As an example, the SCTP protocol, [18.15.2 SCTP](#), has seen very limited use despite some putative advantages over TCP, largely due to lack of NAT-router support. SCTP cannot be used by residential users because the middleboxes have not kept up.

A third category of middlebox-related problems is overzealous blocking in the name of security. SCTP runs into this problem as well, though not quite as universally: a few routers simply drop all SCTP packets because they represent an “unknown” – and therefore suspect – type of traffic. There is a place for this block-by-default approach. If a datacenter firewall blocks all inbound TCP traffic except to port 80 (the HTTP port), and if SCTP is not being used within the datacenter intentionally, it is hard to argue against blocking all inbound SCTP traffic. But if the frontline router for home or office users blocks all *outbound* SCTP traffic, then the users cannot use SCTP.

A consequence of overzealous blocking is that it becomes much harder to introduce new protocols. If a new protocol is blocked for even a small fraction of potential users, it is just not worth the effort. See also the discussion at [18.15.4 QUIC Revisited](#); the design of QUIC includes several elements to mitigate middlebox problems.

For another example of overzealous blocking by middleboxes, with the added element of spoofed TCP RST packets, see the sidebar at [21.5.3 Explicit Congestion Notification \(ECN\)](#).

9.7.3 NAT Traversal

If a server must be located behind a NAT router, the traditional way to make it visible to the outside internet is to “open up” one or more selected ports, using the Port Control Protocol, above. Surprisingly, it is often possible to arrange for a UDP connection between a client A and a server B, both behind different NAT firewalls, without any special NAT-router cooperation. TCP connections can, if desired, then be tunneled over the UDP connection. See [\[MEGK10\]](#) and the [pwnat](#) package.

We will first assume that the server B knows the public IP address A_{public} of A's NAT router, and knows that A wishes to communicate. B then begins sending a series of UDP packets to an agreed-upon port at A_{public} , using an agreed-upon source port; these packets might be sent at 10-second intervals. The pwnat package uses port 2222 for both endpoints; we will assume that here. These packets, of course, are dropped by A's NAT router.

A now sends a single UDP packet to B's public IP address, from port 2222 and to port 2222. When A's NAT router sees these packets, it will create a NAT-table entry as follows:

remote host	remote port	outside source port	inside host	inside port
B _{public}	2222	2222[?]	A	2222

Assuming that A's NAT router does not remap port 2222 as the outside source port, the existence of this NAT-table entry will allow B's packets to be delivered to A. A can then respond, and the bidirectional connection is established. Remapping of port 2222 by A's NAT router would be a serious problem, but is quite unlikely if no other host at A's end is using port 2222.

So far, we have assumed that server B knew about A's interest in advance. A similar trick, using ICMP (below at [10.4 Internet Control Message Protocol](#)) allows A to notify B of its interest and existence, so that B can begin sending its series of UDP packets. The idea here is for B to send periodic ICMP Echo Request (ping-request) packets to a fixed IP address IP_{blackhole} chosen because it is not in use. All these packets disappear somewhere, but they do create an ICMP opening in B's NAT router. The client A is assumed to know B's public IP address B_{public}, and begins to send ICMP Time Exceeded packets to B_{public} that are crafted to look like legitimate responses to B's Echo Request packets. ICMP Time Exceeded messages are acceptable regardless of their source IP address; their intended use is by intermediate routers reporting a problem. B's NAT router will now (hopefully) forward A's arriving Time Exceeded packets to B. The ICMP Time Exceeded packet has error-message space for A's public IP address A_{public}; when received, this triggers B's sending of UDP packets to (A_{public}, 2222) as above. A and B do have to standardize on a specific ping query identifier.

How NAT routers handle ICMP replies does vary from implementation to implementation, and some higher-end NAT devices do notice problems with B's forged ICMP Time Exceeded packets, either at A's end or at B's. However, for most consumer-grade NAT devices, this strategy works quite well.

9.8 Unnumbered Interfaces

We mentioned in [1.10 IP – Internet Protocol](#) and [9.2 Interfaces](#) that some devices allow the use of point-to-point IP links without assigning IP addresses to the interfaces at the ends of the link. Such IP interfaces are referred to as **unnumbered**; they generally make sense only on routers. It is a firm requirement that the node (*ie* router) at each endpoint of such a link has at least one other interface that *does* have an IP address; otherwise, the node in question would be anonymous, and could not participate in the router-to-router protocols of [13 Routing-Update Algorithms](#).

The diagram below shows a link L joining routers R1 and R2, which are connected to subnets 200.0.0.0/24 and 201.1.1.0/24 respectively. The endpoint interfaces of L, both labeled link0, are unnumbered.

The endpoints of L could always be assigned private IPv4 addresses (9.3 Special Addresses), such as 10.0.0.1 and 10.0.0.2. To do this we would need to create a subnet; because the host bits cannot be all 0's or all 1's, the minimum subnet size is four (eg 10.0.0.0/30). Furthermore, the routing protocols to be introduced in 13 Routing-Update Algorithms will distribute information about the subnet throughout the organization or "routing domain", meaning care must be taken to ensure that each link's subnet is unique. Use of unnumbered links avoids this.

If R1 were to *originate* a packet to be sent to (or forwarded via) R2, the standard strategy is for it to treat its link0 interface as if it shared the IP address of its Ethernet interface eth0, that is, 200.0.0.1; R2 would do likewise. This still leaves R1 and R2 violating the IP local-delivery rule of 9.5 The Classless IP Delivery Algorithm; R1 is expected to deliver packets via local delivery to 201.1.1.1 but has no interface that is assigned an IP address on the destination subnet 201.1.1.0/24. The necessary dispensation, however, is granted by RFC 1812. All that is necessary by way of configuration is that R1 be told R2 is a directly connected neighbor reachable via its link0 interface. On Linux systems this might be done with the `ip route` command on R1 as follows:

ip route

The Linux `ip route` command illustrated here was tested on a virtual point-to-point link created with `ssh` and `pppd`; the link interface name was in fact `ppp0`. While the command appeared to work as advertised, it was only possible to create the link if endpoint IP addresses were assigned at the time of creation; these were then removed with `ip route del` and then re-assigned with the command shown here.

```
ip route add 201.1.1.1 dev link0
```

Because L is a point-to-point link, there is no destination LAN address and thus no ARP query.

9.9 Mobile IP

In the original IPv4 model, there was a strong if implicit assumption that each IP host would stay put. One role of an IPv4 address is simply as a unique endpoint identifier, but another role is as a **locator**: some prefix of the address (eg the network part, in the class-A/B/C strategy, or the provider prefix) represents something about where the host is physically located. Thus, if a host moves far enough, it may need a new address.

When laptops are moved from site to site, it is common for them to receive a new IP address at each location, *eg* via DHCP as the laptop connects to the local Wi-Fi. But what if we wish to support devices like smartphones that may remain active and communicating while moving for thousands of miles? Changing IP addresses requires changing TCP connections; life (and application development) might be simpler if a device had a single, unchanging IP address.

One option, commonly used with smartphones connected to some so-called “3G” networks, is to treat the phone’s data network as a giant wireless LAN. The phone’s IP address need not change as it moves within this LAN, and it is up to the phone provider to figure out how to manage LAN-level routing, much as is done in [4.2.4.3 Roaming](#).

But **Mobile IP** is another option, documented in [RFC 5944](#). In this scheme, a mobile host has a permanent **home address** and, while roaming about, will also have a temporary **care-of address**, which changes from place to place. The care-of address might be, for example, an IP address assigned by a local Wi-Fi network, and which in the absence of Mobile IP would be *the* IP address for the mobile host. (This kind of care-of address is known as “co-located”; the care-of address can also be associated with some other device – known as a **foreign agent** – in the vicinity of the mobile host.) The goal of Mobile IP is to make sure that the mobile host is always reachable via its home address.

To maintain connectivity to the home address, a Mobile IP host needs to have a **home agent** back on the home network; the job of the home agent is to maintain an IP tunnel that always connects to the device’s current care-of address. Packets arriving at the home network addressed to the home address will be forwarded to the mobile device over this tunnel by the home agent. Similarly, if the mobile device wishes to send packets *from* its home address – that is, with the home address as IP source address – it can use the tunnel to forward the packet to the home agent.

The home agent may use proxy ARP ([10.2.1 ARP Finer Points](#)) to declare itself to be the appropriate destination on the home LAN for packets addressed to the home (IP) address; it is then straightforward for the home agent to forward the packets.

An **agent discovery** process is used for the mobile host to decide whether it is mobile or not; if it is, it then needs to notify its home agent of its current care-of address.

9.9.1 IP-in-IP Encapsulation

There are several forms of packet encapsulation that can be used for Mobile IP tunneling, but the default one is IP-in-IP encapsulation, defined in [RFC 2003](#). In this process, the entire original IP packet (with header addressed to the home address) is used as data for a new IP packet, with a new IP header (the “outer” header) addressed to the care-of address.

A value of 4 in the outer-IP-header Protocol field indicates that IPv4-in-IPv4 tunneling is being used, so the receiver knows to forward the packet on using the information in the inner header. The MTU of the tunnel will be the original MTU of the path to the care-of address, minus the size of the outer header. A very similar mechanism is used for IPv6-in-IPv4 encapsulation (that is, with IPv6 in the inner packet), except that the outer IPv4 Protocol field value is now 41. See [12.6 IPv6 Connectivity via Tunneling](#).

IP-in-IP encapsulation presents some difficulties for NAT routers. If two hosts A and B behind a NAT router send out encapsulated packets, the packets may differ only in the source IP address. The NAT router, upon receiving responses, doesn't know whether to forward them to A or to B. One partial solution is for the NAT router to support only one inside host sending encapsulated packets. If the NAT router knew that encapsulation was being used for Mobile IP, it might look at the home address in the inner header to determine the correct home agent to which to deliver the packet, but this is a big assumption. A fuller solution is outlined in [RFC 3519](#).

9.10 Epilog

At this point we have concluded the basic mechanics of IPv4. Still to come is a discussion of how IP routers build their forwarding tables. This turns out to be a complex topic, divided into routing within single organizations and ISPs – [13 Routing-Update Algorithms](#) – and routing between organizations – [14 Large-Scale IP Routing](#).

But before that, in the next chapter, we compare IPv4 with IPv6, now twenty years old but still seeing limited adoption. The biggest issue fixed by IPv6 is IPv4's lack of address space, but there are also several other less dramatic improvements.

9.11 Exercises

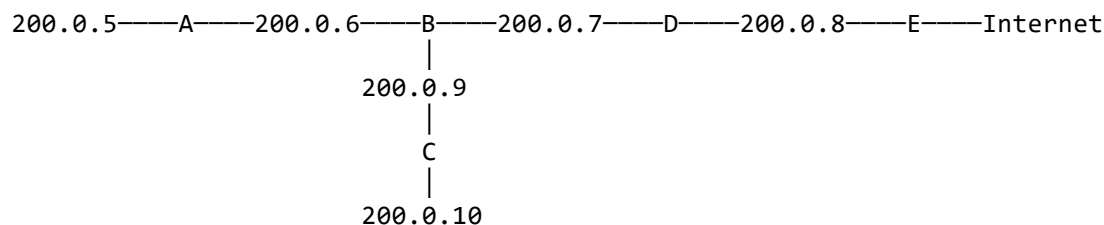
Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercises marked with a ♦ have solutions or hints at [34.9 Solutions for IPv4](#).

1.0. Suppose an Ethernet packet represents a TCP acknowledgment; that is, the packet contains the Ethernet header, the IPv4 header, a 20-byte TCP header, and a 4-byte CRC checksum. Is such a packet smaller than the 10-Mbit Ethernet minimum-packet size, and, if so, by how much?

2.0. How can a receiving host tell if an arriving IPv4 packet is unfragmented? Hint: such a packet will be both the “first fragment” and the “last fragment”; how are these two states marked in the IPv4 header?

3.0. How long will it take the IDENT field of the IPv4 header to wrap around, if the sender host A sends a stream of packets to host B as fast as possible? Assume the packet size is 1500 bytes and the bandwidth is 600 Mbps.

4.0. The following diagram has routers A, B, C, D and E; E is the “border router” connecting the site to the Internet. All router-to-router connections are via Ethernet-LAN /24 subnets with addresses of the form 200.0.x. Give forwarding tables for each of A, B, C and D. Each table should include each of the listed subnets and also a **default** entry that routes traffic toward router E. Directly connected subnets may be listed with a next_hop of “direct”.



5.0. (This exercise is an attempt at modeling Internet-2 routing.) Suppose sites $S_1 \dots S_n$ each have a single connection to the standard Internet, and each site S_i has a single IPv4 address block A_i . Each site's connection to the Internet is through a single router R_i ; each R_i 's default route points towards the standard Internet. The sites also maintain a separate, higher-speed network among themselves; each site has a single link to this separate network, also through R_i . Describe what the forwarding tables on each R_i will have to look like so that traffic from one S_i to another will always use the separate higher-speed network.

6.0. For each IPv4 network prefix given (with length), identify which of the subsequent IPv4 addresses are part of the same subnet.

- 10.0.130.0/23**: 10.0.130.23, 10.0.129.1, 10.0.131.12, 10.0.132.7
- 10.0.132.0/22**: 10.0.130.23, 10.0.135.1, 10.0.134.12, 10.0.136.7
- 10.0.64.0/18**: 10.0.65.13, 10.0.32.4, 10.0.127.3, 10.0.128.4
- 10.0.168.0/21**: 10.0.166.1, 10.0.170.3, 10.0.174.5, 10.0.177.7
- 10.0.0.64/26**: 10.0.0.125, 10.0.0.66, 10.0.0.130, 10.0.0.62

7.0. Convert the following subnet masks to /k notation, and vice-versa:

- 255.255.240.0
- 255.255.248.0
- 255.255.255.192
- /20
- /22
- /27

8.0. Suppose that the subnet bits below for the following five subnets A–E all come from the beginning of the fourth byte of the IPv4 address; that is, these are subnets of a /24 block.

- A: 00
- B: 01

- C: 110
- D: 111
- E: 1010

- (a). What are the sizes of each subnet, and the corresponding decimal ranges? Count the addresses with host bits all 0's or with host bits all 1's as part of the subnet.
- (b). How many IPv4 addresses in the class-C block do not belong to any of the subnets A, B, C, D and E?

10 IPv4 Companion Protocols

IP is the keystone of the Internet, but it occupies that position with a little help from its friends. DNS translates human-readable host names, such as `intronetworks.cs.luc.edu` to IP addresses. ARP translates IPv4 addresses to Ethernet addresses, for destinations on the same LAN. DHCP assigns IPv4 addresses. And ICMP enables the transmission of IPv4-related error and status messages. These four are the subject of this chapter.

The original DNS can be used with IPv6, with modest extensions, but for the other three, IPv6 has its own versions; see [11.6 Neighbor Discovery](#) (replacing ARP), [11.7 IPv6 Host Address Assignment](#) and [12.2 ICMPv6](#).

10.1 DNS

The **Domain Name System**, DNS, is an essential companion protocol to IPv4 (and IPv6); an overview can be found in [RFC 1034](#). It is DNS that permits users the luxury of not needing to remember numeric IP addresses. Instead of 162.216.18.28, a user can simply enter [intronetworks.cs.luc.edu](#), and DNS will take care of looking up the name and retrieving the corresponding address. DNS also makes it easy to move services from one server to another with a different IP address; as users will locate the service by **DNS name** and not by IP address, they do not need to be notified.

While DNS supports a wide variety of queries, for the moment we will focus on queries for IPv4 addresses, or so-called A records. The AAAA record type is used for IPv6 addresses, and, internally, the NS record type is used to identify the “name servers” that answer DNS queries.

While a workstation can use TCP/IP without DNS, users would have an almost impossible time finding anything, and so the core startup configuration of an Internet-connected workstation almost always includes the IP address of its DNS server (see [10.3 Dynamic Host Configuration Protocol \(DHCP\)](#) below for how startup configurations are often assigned).

It's Always DNS

That is a common, though exaggerated, sentiment among system administrators dealing with network problems. While DNS is on the whole remarkably reliable, its failures can be tricky to diagnose.

Most DNS traffic today is over UDP, although a TCP option exists. Due to the much larger response sizes, TCP is often necessary for DNSSEC ([29.7 DNSSEC](#)).

DNS is **distributed**, meaning that each domain is responsible for maintaining its own **DNS servers** to translate names to addresses. DNS, in fact, is a classic example of a highly distributed database where each node maintains a relatively small amount of data. That said, in days gone by it was common practice for each domain to maintain its own DNS server; today, domain registrars often provide DNS services for many of their domain customers.

DNS is **hierarchical** as well; for the DNS name `intronetworks.cs.luc.edu` the levels of the hierarchy are

- **edu**: the **top-level domain** (TLD) for educational institutions in the US
- **luc**: Loyola University Chicago
- **cs**: The Loyola Computer Science Department
- **intronetworks**: a hostname associated to a specific IP address

The hierarchy of DNS names (that is, the set of all names and suffixes of names) forms a tree, but it is not only leaf nodes that represent individual hosts. In the example above, domain names [luc.edu](#) and [cs.luc.edu](#) happen to be valid hostnames as well.

The DNS hierarchy is in a great many cases not very deep, particularly for DNS names assigned to commercial websites. Such domain names are often simply the company name (or a variant of it) followed by the top-level domain (often `.com`). Still, internally most organizations have many individually named behind-the-scenes servers with three-level (or more) domain names; sometimes some of these can be identified by viewing the source of the web page and searching it for domain names.

Top-level domains are assigned by [ICANN](#). The original top-level domains were seven three-letter domains – `.com`, `.net`, `.org`, `.int`, `.edu`, `.mil` and `.gov` – and the two-letter country-code domains (eg `.us`, `.ca`, `.mx`). Now there are hundreds of non-country top-level domains, such as `.aero`, `.biz`, `.info`, and, apparently, [.wtf](#). Domain names (and subdomain names) can also contain unicode characters, so as to support national alphabets. Some top-level domains are *generic*, meaning anyone can apply for a subdomain although there may be qualifying criteria. Other top-level domains are *sponsored*, meaning the sponsoring organization determines who can be assigned a subdomain, and so the qualifying criteria can be a little more arbitrary.

ICANN still must approve all new top-level domains. Applications are accepted only during specific intervals; the application fee for the 2012 interval was US\$185,000. The

actual leasing of domain names to companies and individuals is done by organizations known as **domain registrars** who work under contract with ICANN.

The full tree of all DNS names and prefixes is divided administratively into **zones**: a zone is an independently managed subtree, minus any sub-subtrees that have been placed – by delegation – into their own zone. Each zone has its own root DNS name that is a suffix of every DNS name in the zone. For example, the `luc.edu` zone contains most of Loyola's DNS names, but `cs.luc.edu` has been spun off into its own zone. A zone cannot be the disjoint union of two subtrees; that is, `cs.luc.edu` and `math.luc.edu` must be two distinct zones, unless both remain part of their parent zone.

A zone can define DNS names more than one level deep. For example, the `luc.edu` zone can define records for the `luc.edu` name itself, for names with one additional level such as `www.luc.edu`, and for names with two additional levels such as `www.cs.luc.edu`. That said, it is common for each zone to handle only one additional level, and to create subzones for deeper levels.

Each zone has its own **authoritative nameservers** for the zone, which are charged with maintaining the records – known as **resource records**, or RRs – for that zone. Each zone must have at least two nameservers, for redundancy. IPv4 addresses are stored as so-called **A records**, for Address. Information about how to find sub-zones is stored as **NS records**, for Name Server. Additional resource-record types are discussed at [10.1.3 Other DNS Records](#). Each DNS record type also includes a **time-to-live** field for caching (below).

An authoritative nameserver need not be part of the organization that manages the zone, and a single server can be the authoritative nameserver for multiple unrelated zones. For example, many domain registrars maintain single nameservers that handle DNS queries for all their domain customers who do not wish to set up their own nameservers.

The **root nameservers** handle the zone that is the root of the DNS tree; that is, that is represented by the DNS name that is the empty string. As of 2019, there are thirteen of them. The root nameservers contain only NS records, identifying the nameservers for all the immediate subzones. Each top-level domain is its own such subzone. The IP addresses of the root nameservers are widely distributed. Their DNS names (which are only of use if some DNS lookup mechanism is already present) are `a.root.servers.net` through `m.root-servers.net`. These names today correspond not to individual machines but to clusters of up to hundreds of servers.

10.1.1 DNS Resolvers

We can now put together a first draft of a DNS lookup algorithm. To find the IP address of [intronetworks.cs.luc.edu](#), a host first contacts a root nameserver (at a known address) to find the nameserver for the `edu` zone; this involves the retrieval of an NS record. The `edu` nameserver is then queried to find the nameserver for the `luc.edu` zone, which

in turn supplies the NS record giving the address of the `cs.luc.edu` zone. This last has an A record for the actual host. (This example is carried out in detail below.) The system (or application) that executes these DNS lookups is known as a **DNS resolver**. Confusingly, resolvers are also sometimes known as “nameservers” or, more precisely, **non**–authoritative nameservers.

To reduce overall DNS traffic, in particular to the root nameservers, it makes sense to cache intermediate (and final) results, so that in a later query for, say, uchicago.edu, the host can reuse the previously learned address of the edu nameserver.

For still greater DNS efficiency, we can provide one DNS resolver to handle requests for a large pool of users. The idea is that, if one user has looked up `youtube.com`, or `facebook.com`, those addresses are in hand locally for the next user. The benefit of this consolidation approach depends on the distribution of lookup requests, their lifetimes, and how likely it is that two users visit the same site. In recent years this caching benefit has been getting smaller, at least for “full” DNS names, as the Internet becomes more diverse, and as cache lifetimes have been shrinking. (The caching benefit for DNS “partial” names, such as `.edu` and `.com`, remains significant.)

Regardless of caching benefits, such pooled–use DNS resolvers are almost universally used. Almost all ISPs and most companies, for example, provide a resolver to handle the DNS needs of their customers and employees. We will refer to these as **site resolvers**. The IP addresses of these site resolvers are generally supplied via DHCP options ([10.3 Dynamic Host Configuration Protocol \(DHCP\)](#)); such resolvers are thus the default choice for DNS services.

DNS Policing

It is sometimes suggested that if a site is engaged in illegal activity or copyright infringement, such as thepiratebay.se, its domain name should be seized. The problem with this strategy is that it is straightforward for users to set up “nonstandard” nameservers (for example the Gnu Name System, [GNS](#)) that continue to list the banned site.

Sometimes, however, users elect to use a DNS resolver not provided by their ISP or company; there are a number of **public DNS servers** (that is, resolvers) available. Such resolvers generally serve much larger areas. Common choices include [OpenDNS](#), [Google DNS](#) (primary address 8.8.8.8), [Cloudflare](#) (primary address 1.1.1.1) and the Gnu Name System mentioned in the sidebar above, though there are many others. Searching for “public DNS server” turns up lists of them.

In theory, one advantage of using a public DNS server is that your local ISP can no longer track your DNS queries. However, some ISPs still do *record* customer DNS queries, and may even intercept and modify them. If this is a concern, DNS encryption is necessary. There are two primary proposals:

- DNS over TLS (DoT): [RFC 7858](#)
- DNS over HTTPS (DoH): [RFC 8484](#)

TLS is an encryption protocol ([29.5.2 TLS](#)). HTTPS – secure HTTP – uses TLS encryption, but the two are not the same. For one thing, eavesdroppers can still identify DoT traffic as DNS traffic (because it is sent to a special port), but DoH traffic is indistinguishable from HTTPS web traffic.

Use of any DNS server, whether via plain DNS or DoT or DoH, does mean that the DNS server now has access to all your DNS queries. Ultimately, the choice depends on how much you trust your site resolver versus your selected public resolver.

Both DoT and DoH often take some deliberate configuration to enable as the standard system resolver. However, as of 2020 Mozilla has started enabling DoH by default in their [Firefox](#) browser, that is, without operating-system support, though disabling DoH and reverting back to the system DNS resolver is straightforward.

IPv6

Google and Cloudflare provide public DNS for IPv6 users too, though the addresses are much less easy to remember. Google's is 2001:4860:4860::8888; Cloudflare's is 2606:4700:4700::1111.

In setting up DoH for Firefox, the [Mozilla foundation](#) created what it calls its [Trusted Recursive Resolver](#) program. Participating providers – initially [Cloudflare](#), and eventually others – had to agree to contractual requirements. Among these requirements are that personal browsing history not be collected, and that Query Name Minimization ([10.1.2.1 Query Name Minimization](#)) be supported. Ultimately, however, the choice of what DNS to trust belongs to the user.

Some public DNS servers provide additional services, such as automatically filtering out domain names associated with security risks, or content inappropriate for young users. Sometimes there is a fee for this service. A common drawback to filtering content at the DNS level is that, unless the DNS server provides an alternative address pointing to an explanation page, users who attempt to access blocked content may have absolutely no idea what went wrong. Because of this, filtering content at the browser level rather than the DNS level is sometimes preferred, though filtering at the browser level has its own drawbacks. In October 2021, Cloudflare (with public DNS 1.1.1.1, above) began offering two additional (and free) public DNS servers:

- 1.1.1.2, blocking known malware
- 1.1.1.3, blocking malware and adult content

See blog.cloudflare.com/introducing-1-1-1-1-for-families for further details.

As mentioned earlier, each DNS record comes with a **time-to-live** (TTL) value, used by resolvers as an indication of how long they are supposed to keep that record in their caches. DNS TTL lifetimes can be up to several days; [RFC 1035](#) recommends a minimum TTL of “at least a day”. However, in recent years TTL values have been getting quite a bit smaller. Below, in [10.1.2 nslookup and dig](#), we retrieve the TTL values for `facebook.com` and `google.com` from their respective authoritative nameservers; in each case it is 300 seconds (5 minutes).

Authoritative nameservers also provide a TTL value for lookup *failures*. According to [RFC 2308](#), this is the TTL value specified in the SOA record. (Originally, the SOA TTL represented the default TTL for *successful* lookups.) Lookup-failure TTLs should usually be kept quite short; otherwise there is potential for large numbers of users to be locked out of a site. Consider, for example, the following scenario for updating a DNS record for host `foo.com`. Let us suppose that the lookup-failure TTL is one week:

Site <code>foo.com</code>	Site B (perhaps a large ISP)
Delete <code>foo.com</code> A record	
	Site B queries for <code>foo.com</code>
	Site B gets NXDOMAIN
Immediately reinstall <code>foo.com</code> A record	

At this point, Site B may be telling its users for a week that `foo.com` is unavailable, and site `foo.com` will be unable to fix it.

If I send a query to Loyola’s site resolver for `google.com`, it is almost certainly in the cache. If I send a query for the misspelling [googel.com](#), this may not be in the cache, but the `.com` top-level nameserver almost certainly *is* in the cache. From that nameserver my local resolver finds the nameserver for the `googel.com` zone, and from that finds the IP address of the `googel.com` host.

There are, as of 2019, around 1500 top-level domains. If, while still using Loyola’s site resolver, I send a query for a site in one of the more obscure top-level domains, there is a reasonable chance that the top-level domain will *not* be in the cache. A consequence of this aspect of caching is that popular top-level domains are likelier to result in faster lookups.

Applications almost always invoke DNS through library calls, such as Java’s `InetAddress.getByName()`. The library forwards the query to the system-designated resolver (though browsers sometimes offer other DNS options; see [29.7.4 DNS over HTTPS](#)). We will return to DNS library calls in [16.1.3.3 The Client](#) and [17.6.1 The TCP Client](#).

On unix-based systems, traditionally the IPv4 addresses of the local DNS resolvers were kept in a file `/etc/resolv.conf`. Typically this file was updated with the addresses of the current resolvers by DHCP ([10.3 Dynamic Host Configuration Protocol \(DHCP\)](#)), at the time the system received its IPv4 address. It is possible, though not common, to create

special entries in `/etc/resolv.conf` so that queries about different domains are sent to different resolvers, or so that single-level hostnames have a domain name appended to them before lookup. On Windows, similar functionality can be achieved through settings on the DNS tab within the Network Connections applet.

Recent systems often run a small “stub” resolver locally (eg Linux’s [dnsmasq](#)); such resolvers are sometimes also called DNS *forwarders*. The entry in `/etc/resolv.conf` is then an IPv4 address of `localhost` (sometimes 127.0.1.1 rather than 127.0.0.1). Such a stub resolver would, of course, still need access to the addresses of site or public resolvers; sometimes these addresses are provided by static configuration and sometimes by DHCP ([10.3 Dynamic Host Configuration Protocol \(DHCP\)](#)).

If a system running a stub resolver then runs internal virtual machines, it is usually possible to configure everything so that the virtual machines can be given an IP address of the host system as their DNS resolver. For example, often virtual machines are assigned IPv4 addresses on a private subnet and connect to the outside world using NAT ([9.7 Network Address Translation](#)). In such a setting, the virtual machines are given the IPv4 address of the host system interface that connects to the private subnet. It is then necessary to ensure that, on the host system, the local resolver accepts queries sent not only to the designated loopback address but also to the host system’s private-subnet address. (Generally, local resolvers do *not* accept requests arriving from externally visible addresses.)

When someone submits a query for a nonexistent DNS name, the resolver is supposed to return an error message, technically known as **NXDOMAIN** (Non eXistent Domain). Some resolvers, however, have been configured to return the IP address of a designated web server; this is particularly common for ISP-provided site resolvers. Sometimes the associated web page is meant to be helpful, and sometimes it presents an offer to buy the domain name from a registrar. Either way, additional advertising may be displayed. Of course, this is completely useless to users who are trying to contact the domain name in question via a protocol (ssh, smtp) other than http.

At the DNS protocol layer, a DNS lookup query can be either **recursive** or **non-recursive**. If A sends to B a recursive query to resolve a given DNS name, then B takes over the job until it is finally able to return an answer to A. If The query is non-recursive, on the other hand, then if B is not an authoritative nameserver for the DNS name in question it returns either a failure notice or an NS record for the sub-zone that is the next step on the path. Almost all DNS requests from hosts to their site or public resolvers are recursive.

A basic DNS response consists of an ANSWER section, an AUTHORITY section and, optionally, an ADDITIONAL section. Generally a response to a lookup of a hostname contains an ANSWER section consisting of a single A record, representing a single IPv4 address. If a site has multiple servers that are entirely equivalent, however, it is possible to give them all the same hostname by configuring the authoritative nameserver to return, for the hostname in question, multiple A records listing, in turn, each of the

server IPv4 addresses. This is sometimes known as **round-robin DNS**. It is a simple form of **load balancing**; see also [30.9.5 loadbalance31.py](#). Consecutive queries to the nameserver should return the list of A records in different orders; ideally the same should also happen with consecutive queries to a local resolver that has the hostname in its cache. It is also common for a single server, with a single IPv4 address, to be identified by multiple DNS names; see the next section.

The response AUTHORITY section contains the DNS names of the authoritative nameservers responsible for the original DNS name in question. These records are often NS records, which point to the zone from its parent, though SOA records – declaring a zone from within – are also seen. The ADDITIONAL section contains information the sender thinks is related; for example, this section often contains A records for the authoritative nameservers.

The [Tor Project](#) uses DNS-like names that end in “.onion”. While these are not true DNS names in that they are not managed by the DNS hierarchy, they do work as such for Tor users; see [RFC 7686](#). These names follow an unusual pattern: the next level of name is an 80-bit hash of the site’s RSA public key ([29.1 RSA](#)), converted to sixteen ASCII bytes. For example, 3g2upl4pq6kufc4m.onion is apparently the Tor address for the search engine [duckduckgo.com](#). Unlike DuckDuckGo, many sites try different RSA keys until they find one where at least some initial prefix of the hash looks more or less meaningful; for example, nytimes2tsqtnxek.onion. Facebook got very [lucky](#) in finding an RSA key whose corresponding Tor address is facebookcorewwwi.onion (though it is sometimes said that *fortune is infatuated with the wealthy*). This naming strategy is a form of **cryptographically generated addresses**; for another example see [11.6.4 Security and Neighbor Discovery](#). The advantage of this naming strategy is that you don’t need a certificate authority ([29.5.2.1 Certificate Authorities](#)) to verify a site’s RSA key; the site name does it for you.

10.1.2 nslookup and dig

Let us trace a non-recursive lookup of `intronetworks.cs.luc.edu`, using the [nslookup](#) tool. The nslookup tool is time-honored, but also not completely up-to-date, so we also include examples using the [dig](#) utility (supposedly an acronym for “domain Internet groper”). Lines we type in nslookup’s interactive mode begin below with the prompt “>”; the shell prompt is “#”. All dig commands are typed directly at the shell prompt.

The first step is to look up the IP address of the root nameserver `a.root-servers.net`. We can do this with a regular call to nslookup or dig, we can look this up in our nameserver’s configuration files, or we can search for it on the Internet. The address is 198.41.0.4.

We now send our nonrecursive query to this address. The presence of the single hyphen in the nslookup command line below means that we want to use 198.41.0.4 as the

nameserver rather than as the thing to be looked up; dig has places on the command line for both the nameserver (following the @) and the DNS name. For both commands, we use the norecurse option to send a nonrecursive query.

```
# nslookup -norecurse - 198.41.0.4
> intronetworks.cs.luc.edu
*** Can't find intronetworks.cs.luc.edu: No answer

# dig @198.41.0.4 intronetworks.cs.luc.edu +norecurse
```

These fail because by default nslookup and dig ask for an A record. What we want is an NS record: the name of the next zone down to ask. (We can tell the dig query failed to find an A record because there are zero records in the ANSWER section)

```
> set query=ns
> intronetworks.cs.luc.edu
edu      nameserver = a.edu-servers.net
...
a.edu-servers.net      internet address = 192.5.6.30

# dig @198.41.0.4 intronetworks.cs.luc.edu NS +norecurse
;; AUTHORITY SECTION:
edu.          172800  IN      NS      b.edu-servers.net.
;; ADDITIONAL SECTION:
b.edu-servers.net. 172800  IN      A      192.33.14.30
```

The full responses in each case are a list of all nameservers for the .edu zone; we list only the first response above. The IN in the two response records shown above indicates these are InterNet records.

Note that the full DNS name intronetworks.cs.luc.edu in the query here is *not* an exact match for the DNS name .edu in the resource record returned; the latter is a suffix of the former. This has privacy implications; the root nameserver didn't need to know we were searching for intronetworks.cs.luc.edu. We could have just asked for edu. We return to this in [10.1.2.1 Query Name Minimization](#).

We send the next NS query to a.edu-servers.net (which does appear in the full dig answer)

```
# nslookup -query=ns -norecurse - 192.5.6.30
> intronetworks.cs.luc.edu
...
Authoritative answers can be found from:
luc.edu nameserver = bcdns11.it.luc.edu.
bcdns11.it.luc.edu      internet address = 147.126.64.64

# dig @192.5.6.30 intronetworks.cs.luc.edu NS +norecurse
;; AUTHORITY SECTION:
luc.edu.          172800  IN      NS      bcdns11.it.luc.edu.
;; ADDITIONAL SECTION:
bcdns11.it.luc.edu. 172800  IN      A      147.126.1.217
```

(Again, we show only one of several luc.edu nameservers returned).

The next step is to ask the luc.edu nameserver for the cs.luc.edu nameserver.

```
# nslookup -query=ns - -norecurse 147.126.1.217
> cs.luc.edu
...
cs.luc.edu      nameserver = bcdns1s1.it.luc.edu.

# dig @147.126.1.217 intronetworks.cs.Luc.edu NS +norecurse
;; AUTHORITY SECTION:
cs.luc.edu.      300      IN      SOA      bcdns1s1.it.luc.edu.
postmaster.luc.edu. 589544360 1200 180 1209600 300
```

Desperately seeking Switzerland

To look up the .edu nameservers with the default resolver, we might use `dig edu NS`. This works for any top-level domain (eg us, de, uk, aero), *except India (.in) and Switzerland (.ch)*. The command `dig in NS` returns the root servers; that is, in is interpreted as the DNS abbreviation for InterNet. A working query is `dig in. IN NS` (note the period). For Switzerland the problem is that ch collides with the DNS abbreviation for the now-obsolete [ChaosNet](#); a working query is `dig ch IN NS`. For more, see [here](#).

The `nslookup` command returns the same nameserver as before; the `dig` command does also, but at least indicates it is returning an SOA rather than an NS record. The first data field of the SOA result – `bcdns1s1.it.luc.edu.` – is the primary nameserver for `cs.luc.edu`. All this is a somewhat roundabout way of saying that the same nameserver handles `cs.luc.edu` as handles `luc.edu`; that is, they are two zones that just happen to be handled by the same nameserver. Prior to 2019, `cs.luc.edu` was handled by a separate nameserver, but after a significant outage it was folded back to the `luc.edu` nameserver. If we drop the `intronetworks` label in the last query above, that is, we run `dig @147.126.1.217 cs.luc.edu NS +norecurse`, we now get an ANSWER section (instead of AUTHORITY), which declares that `bcdns1s1.it.luc.edu` is indeed the authoritative nameserver for `cs.luc.edu`.

In any event, we can now ask for the A record directly:

```
# nslookup -query=A - -norecurse 147.126.1.217
> intronetworks.cs.luc.edu
...
Name: intronetworks.cs.luc.edu
Address: 162.216.18.28

# dig @147.126.1.217 intronetworks.cs.Luc.edu A +norecurse
;; ANSWER SECTION:
intronetworks.cs.luc.edu. 600 IN A 162.216.18.28
```

This is the first time we get an ANSWER section (versus the AUTHORITY section)

Prior to 2019, the final result from `nslookup` was in fact this:

```

intronetworks.cs.luc.edu      canonical name = linode1.cs.luc.edu.
Name:  linode1.cs.luc.edu
Address: 162.216.18.28

```

Here we received a **canonical name**, or CNAME, record. The server that hosts `intronetworks.cs.luc.edu` also hosts several other websites, with different names; for example, introcs.cs.luc.edu. This is known as **virtual hosting**. Rather than provide separate A records for each website name, DNS was set up to provide a CNAME record for each website name pointing to a single physical server name `linode1.cs.luc.edu`. Only one A record is then needed, for this server. Post-2019, this CNAME strategy is no longer used. Note that both the CNAME and the corresponding A record was returned, for convenience. The pre-2019 answer returned by `dig` (above) made no mention of CNAMEs, because they are often of little user interest; `dig` will return CNAMEs however if asked explicitly.

Note that the IPv4 address here, 162.216.18.28, is unrelated to Loyola's own IPv4 address block 147.126.0.0/16. The server hosting `intronetworks.cs.luc.edu` is managed by an external provider; there is no connection between the DNS name hierarchy and the IP address hierarchy.

As another example of the use of `dig`, we can find the time-to-live values advertised by `facebook.com` and `google.com`:

```

dig facebook.com
;; ANSWER SECTION:
facebook.com.      78      IN      A       157.240.18.35
;; AUTHORITY SECTION:
facebook.com.      147771  IN      NS      b.ns.facebook.com.

dig google.com
;; ANSWER SECTION:
google.com.        103     IN      A       172.217.9.78
;; AUTHORITY SECTION:
google.com.        141861  IN      NS      ns3.google.com.

```

The TTLs are 78 and 103 seconds, respectively. But these are the TTLs coming from the local site resolver. To get the TTL values from `facebook.com` and `google.com` directly, we can do this:

```

dig @b.ns.facebook.com facebook.com
;; ANSWER SECTION:
facebook.com.      300     IN      A       157.240.18.35

dig @ns3.google.com google.com
;; ANSWER SECTION:
google.com.        300     IN      A       172.217.1.14

```

That is, both sites' authoritative nameservers advertise a TTL of 300 seconds (5 minutes). The TTL value of 78 received above means that our local resolver itself last asked about facebook.com $300 - 78 = 222$ seconds ago.

10.1.2.1 Query Name Minimization

In the example above in which we traced the lookup of DNS name `intronetworks.cs.luc.edu` starting from the root, we commented that the entire DNS name was sent to the root server. This was unnecessary, as the root nameservers only know how to reach the `.edu` nameservers; we could have sent an NS request just for `.edu`. There is a privacy issue here; the root nameservers don't need to know everyone's full queries.

[RFC 7816](#) proposes **query name minimization**, or QNAME minimization, as an alternative. The idea is to send the root server just a query for `.edu`, then the `.edu` nameserver just a query for `luc.edu`, and so on. After each NS query, one more DNS label is attached to the query name before proceeding to the next query to the next nameserver. This way, no nameserver learns more of the query than the absolute minimum.

There is a potential catch, though, as not every level of the DNS name corresponds to a different nameserver; to put it another way, not every '.' in a DNS name corresponds to a zone break. For example, it is now (2020) the case that the `luc.edu` nameserver is responsible for the formerly independent `cs.luc.edu` name hierarchy, and so there is no longer a need for a `cs.luc.edu` NS record. There happens still to be one, for legacy reasons, but if that were not the case, then an NS query sent to the `luc.edu` nameserver `cs.luc.edu` might return with the literally correct NODATA, or, worse, NXDOMAIN (the latter is not supposed to happen, but sometimes does).

The [RFC 7816](#) solution to this, when a negative answer is received, is to include one more DNS-name level and repeat the query. That is, if a lookup for `cs.luc.edu` failed, try the full name. That said, this and other issues mean that query name minimization has not quite seen widespread adoption; see this [APNIC blog post](#) for some actual measurements.

It is worth noting that the only privacy protection achieved here is from non-leaf DNS nameservers. Also, ones local DNS resolver still has full information about each query it is sent. In the presence of active caching, a local resolver would generally not need to query the root or the `.edu` nameservers at all.

10.1.2.2 Naked Domains

If we look up both `www.cs.luc.edu` and `cs.luc.edu`, we see they resolve to the same address. The use of `www` as a hostname for a domain's webserver is sometimes considered unnecessary and old-fashioned; many users and website administrators prefer the shorter, "naked" domain name, *eg* `cs.luc.edu`.

It might be tempting to create a CNAME record for the naked domain, `cs.luc.edu`, pointing to the full hostname `www.cs.luc.edu`. However, [RFC 1034](#) does not allow this:

If a CNAME RR is present at a node, no other data should be present; this ensures that the data for a canonical name and its aliases cannot be different.

There are, however, several other DNS data records for `cs.luc.edu`: an NS record (above), a SOA, or Start of Authority, record containing various administrative data such as the expiration time, and an MX record, discussed in the following section. All this makes `www.cs.luc.edu` and `cs.luc.edu` ineluctably quite different. [RFC 1034](#) adds, “this rule also insures that a cached CNAME can be used without checking with an authoritative server for other RR types.”

A better way to create a naked-domain record, at least from the perspective of DNS, is to give it its own A record. This does mean that, if the webserver address changes, there are now *two* DNS records that need to be updated, but this is manageable.

Recently ANAME records have been proposed to handle this issue; an ANAME is like a *limited* CNAME not subject to the [RFC 1034](#) restriction above. An ANAME record for a naked domain, pointing to another hostname rather than to address, is legal. See the Internet draft [draft-hunt-dnsop-aname](#). Some large CDNs ([1.12.2 Content-Distribution Networks](#)) already implement similar DNS tweaks internally. This does not require end-user awareness; the user requests an A record and the ANAME is resolved at the CDN side.

Finally, there is also an argument, at least when HTTP (web) traffic is involved, that the `www` *not* be deprecated, and that the naked domain should instead be *redirected*, at the HTTP layer, to the full hostname. This simplifies some issues; for example, you now have only one website, rather than two (though it does add an extra RTT). You no longer have to be concerned with the fact that HTTP cookies with and without the “`www`” are different. And some CDNs may not be able to handle website failover to another server if the naked domain is reached via an A record. But none of these are DNS issues.

10.1.3 Other DNS Records

Besides address lookups, DNS also supports a few other kinds of searches. The best known is probably **reverse DNS**, which takes an IP address and returns a name. This is slightly complicated by the fact that one IP address may be associated with multiple DNS names. What DNS does in this case is to return the canonical name, or CNAME; a given address can have only one CNAME.

Given an IPv4 address, say `147.126.1.230`, the idea is to reverse it and append to it the suffix `in-addr.arpa`.

`230.1.126.147.in-addr.arpa`

There is a DNS name hierarchy for names of this form, with zones and authoritative servers. If all this has been configured – which it often is not, especially for user workstations – a request for the PTR record corresponding to the above should return a DNS hostname. In the case above, the name luc.edu is returned (at least as of 2018).

PTR records are the only DNS records to have an entirely separate hierarchy; other DNS types fit into the “standard” hierarchy. For example, DNS also supports MX, or Mail eXchange, records, meant to map a domain name (which might not correspond to any hostname, and, if it does, is more likely to correspond to the name of a web server) to the hostname of a server that accepts email on behalf of the domain. In effect this allows an organization’s domain name, *eg* luc.edu, to represent both a web server and, at a different IP address, an email server. MX records can even represent a *set* of IP addresses that accept email.

DNS has from the beginning supported TXT records, for arbitrary text strings. The email Sender Policy Framework ([RFC 7208](https://tools.ietf.org/html/rfc7208)) was developed to make it harder for email senders to pretend to be a domain they are not; this involves inserting so-called SPF records as DNS TXT records.

For example, a DNS query for TXT records of google.com (not gmail.com!) might yield (2018)

```
google.com      text = "docusign=05958488-4752-4ef2-95eb-aa7ba8a3bd0e"
google.com      text = "v=spf1 include:_spf.google.com ~all"
```

The SPF system is interested in only the second record; the “v=spf1” specifies the SPF version. This second record tells us to look up _spf.google.com. That lookup returns

```
text = "v=spf1 include:_netblocks.google.com include:_netblocks2.google.com
include:_netblocks3.google.com ~all"
```

Lookup of _netblocks.google.com then returns

```
text = "v=spf1 ip4:64.233.160.0/19 ip4:66.102.0.0/20 ip4:66.249.80.0/20
ip4:72.14.192.0/18 ip4:74.125.0.0/16 ip4:108.177.8.0/21 ip4:173.194.0.0/16
ip4:209.85.128.0/17 ip4:216.58.192.0/19 ip4:216.239.32.0/19 ~all"
```

If a host connects to an email server, and declares that it is delivering mail from someone at google.com, then the host’s email list should occur in the list above, or in one of the other included lists. If it does not, there is a good chance the email represents spam.

Each DNS record (or “resource record”) has a name (*eg* cs.luc.edu) and a type (*eg* A or AAAA or NS or MX). Given a name and type, the set of matching resource records is known as the **RRset** for that name and type (technically there is also a “class”, but the class of all the DNS records we are interested in is IN, for Internet). When a nameserver responds to a DNS query, what is returned (in the ANSWER section) is always an entire

RRset: the RRset of all resource records matching the name and type contained in the original query.

In many cases, RRsets have a single member, because many hosts have a single IPv4 address. However, this is not universal. We saw above the example of a single DNS name having multiple A records when round-robin DNS is used. A single DNS name might also have separate A records for the host's public and private IPv4 addresses. TXT records, too, often contain multiple entries in a single RRset. In the TXT example above we saw that SPF data was stored in DNS TXT records, but there are other protocols besides SPF that also use TXT records; examples include [DMARC](#) and *google-site-verification* `<https://support.google.com/webmasters/answer/9008080?hl=en&visit_id=637364154322486031-1726883571&rd=1>`. Finally, perhaps most MX-record (Mail eXchange) RRsets have multiple entries, as organizations often prefer, for redundancy, to have more than one server that can receive email.

10.1.4 DNS Cache Poisoning

The classic DNS security failure, known as cache poisoning, occurs when an attacker has been able to convince a DNS resolver that the address of, say, `www.example.com` is something other than what it really is. A successful attack means the attacker can direct traffic meant for `www.example.com` to the attacker's own, malicious site.

The most basic cache-poisoning strategy is to send a stream of DNS reply packets to the resolver which declare that the IP address of `www.example.com` is the attacker's chosen IP address. The source IP address of these packets should be spoofed to be that of the `example.com` authoritative nameserver; such spoofing is relatively easy using UDP. Most of these reply packets will be ignored, but the hope is that one will arrive shortly after the resolver has sent a DNS request to the `example.com` authoritative nameserver, and interprets the spoofed reply packet as a legitimate reply.

To prevent this, DNS requests contain a 16-bit ID field; the DNS response must echo this back. The response must also come from the correct port. This leaves the attacker to guess 32 bits in all, but often the ID field (and even more often the port) can be guessed based on past history.

Another approach requires the attacker to wait for the target resolver to issue a legitimate request to the attacker's site, `attacker.com`. The attacker then piggybacks in the ADDITIONAL section of the reply message an A record for `example.com` pointing to the attacker's chosen bad IP address for this site. The hope is that the receiving resolver will place these A records from the ADDITIONAL section into its cache without verifying them further and without noticing they are completely unrelated. Once upon a time, such DNS resolver behavior was common.

Most newer DNS resolvers carefully validate the replies: the ID field must match, the source port must match, and any received DNS records in the ADDITIONAL section must match, at a minimum, the DNS zone of the request. Additionally, the request ID field and

source port should be chosen pseudorandomly in a secure fashion. For additional vulnerabilities, see [RFC 3833](#).

The central risk in cache poisoning is that a resolver can be tricked into supplying users with invalid DNS records. A closely related risk is that an attacker can achieve the same result by spoofing an authoritative nameserver. Both of these risks can be mitigated through the use of the DNS security extensions, known as **DNSSEC**. Because DNSSEC makes use of public-key signatures, we defer coverage to [29.7 DNSSEC](#).

10.1.5 DNS and CDNs

DNS is often pressed into service by CDNs ([1.12.2 Content-Distribution Networks](#)) to identify their closest “edge” server to a given user. Typically this involves the use of **geoDNS**, a slightly nonstandard variation of DNS. When a DNS query comes in to one of the CDN’s authoritative nameservers, that server

1. looks up the approximate location of the client ([14.4.4 IP Geolocation](#))
2. determines the closest edge server to that location
3. replies with the IP address of that closest edge server

This works reasonably well most of the time. However, the requesting client is essentially never the end user; rather, it is the DNS resolver being used by the user. Typically such resolvers are the site resolvers provided by the user’s ISP or organization, and are physically quite close to the user; in this case, the edge server identified above will be close to the user as well. However, when a user has chosen a (likely remote) public DNS resolver, as above, the IP address returned for the CDN edge server will be close to the DNS resolver but likely far from optimal for the end user.

One solution to this last problem is addressed by [RFC 7871](#), which allows DNS resolvers to include the IP address of the client in the request sent to the authoritative nameserver. For privacy reasons, usually only a prefix of the user’s IP address is included, perhaps /24. Even so, user’s privacy is at least partly compromised. For this reason, [RFC 7871](#) recommends that the feature be disabled by default, and only enabled after careful analysis of the tradeoffs.

A user who is concerned about the privacy issue can – in theory – configure their own DNS software to include this [RFC 7871](#) option with a zero-length prefix of the user’s IP address, which conveys no address information. The user’s resolver will then not change this to a longer prefix.

Use of this option also means that the DNS resolver receiving a user query about a given hostname can no longer simply return a cached answer from a previous lookup of the hostname. Instead, the resolver needs to cache separately each ⟨hostname,prefix⟩ pair it handles, where the prefix is the prefix of the user’s IP address forwarded to the

authoritative nameserver. This has the potential to increase the cache size by several orders of magnitude, which may thereby enable some cache-overflow attacks.

10.2 Address Resolution Protocol: ARP

If a host or router A finds that the destination IP address $D = D_{IP}$ matches the network address of one of its interfaces, it is to deliver the packet via the LAN (probably Ethernet). This means looking up the LAN address (MAC address) D_{LAN} corresponding to D_{IP} . How does it do this?

One approach would be via a special server, but the spirit of early IPv4 development was to avoid such servers, for both cost and reliability issues. Instead, the **Address Resolution Protocol** (ARP) is used. This is our first protocol that takes advantage of the existence of LAN-level broadcast; on LANs without physical broadcast (such as ATM), some other mechanism (usually involving a server) must be used.

The basic idea of ARP is that the host A sends out a broadcast ARP query or “who-has D_{IP} ?” request, which includes A’s own IPv4 and LAN addresses. All hosts on the LAN receive this message. The host for whom the message is intended, D, will recognize that it should reply, and will return an ARP reply or “is-at” message containing D_{LAN} . Because the original request contained A_{LAN} , D’s response can be sent directly to A, that is, unicast.

Additionally, all hosts maintain an **ARP cache**, consisting of $\langle IPv4, LAN \rangle$ address pairs for other hosts on the network. After the exchange above, A has $\langle D_{IP}, D_{LAN} \rangle$ in its table; anticipating that A will soon send it a packet to which it needs to respond, D also puts $\langle A_{IP}, A_{LAN} \rangle$ into its cache.

ARP-cache entries eventually expire. The timeout interval used to be on the order of 10 minutes, but Linux systems now use a much smaller timeout (~30 seconds observed in 2012). Somewhere along the line, and probably related to this shortened timeout interval, repeat ARP queries about a *timed-out* entry are first sent **unicast**, not broadcast, to the previous Ethernet address on record. This cuts down on the total amount of broadcast traffic; LAN broadcasts are, of course, still needed for new hosts. The ARP cache on a Linux system can be examined with the command `ip -s neigh`; the corresponding windows command is `arp -a`.

The above protocol is sufficient, but there is one further point. When A sends its broadcast “who-has D?” ARP query, all other hosts C check their own cache for an entry for A. If there *is* such an entry (that is, if A_{IP} is found there), then the value for A_{LAN} is updated with the value taken from the ARP message; if there is no pre-existing entry then no action is taken. This update process serves to avoid stale ARP-cache entries,

which can arise is if a host has had its Ethernet interface replaced. (USB Ethernet interfaces, in particular, can be replaced very quickly.)

ARP is quite an efficient mechanism for bridging the gap between IPv4 and LAN addresses. Nodes generally find out neighboring IPv4 addresses through higher-level protocols, and ARP then quickly fills in the missing LAN address. However, in some Software-Defined Networking ([3.4 Software-Defined Networking](#)) environments, the LAN switches and/or the LAN controller may have knowledge about IPv4/LAN address correspondences, potentially making ARP superfluous. The LAN (Ethernet) switching network might in principle even know exactly how to route *via the LAN* to a given IPv4 address, potentially even making LAN addresses unnecessary. At such a point, ARP may become an inconvenience. For an example of a situation in which it is necessary to work around ARP, see [30.9.5 loadbalance31.py](#).

10.2.1 ARP Finer Points

Most hosts today implement **self-ARP**, or **gratuitous ARP**, on startup (or wakeup): when station A starts up it sends out an ARP query *for itself*: “who-has A?”. Two things are gained from this: first, all stations that had A in their cache are now updated with A’s most current A_{LAN} address, in case there was a change, and second, if an answer is received, then presumably some other host on the network has the same IPv4 address as A.

Self-ARP is thus the traditional IPv4 mechanism for **duplicate address detection**. Unfortunately, it does not always work as well as might be hoped; often only a single self-ARP query is sent, and if a reply is received then frequently the only response is to log an error message; the host may even continue using the duplicate address! If the duplicate address was received via DHCP, below, then the host is supposed to notify its DHCP server of the error and request a different IPv4 address.

[RFC 5227](#) has defined an improved mechanism known as **Address Conflict Detection**, or ACD. A host using ACD sends out three ARP queries for its new IPv4 address, spaced over a few seconds and leaving the ARP field for the sender’s IPv4 address filled with zeroes. This last step means that any other host with that IPv4 address in its cache will ignore the packet, rather than update its cache. If the original host receives no replies, it then sends out two more ARP queries for its new address, this time with the ARP field for the sender’s IPv4 address filled in with the new address; this is the stage at which other hosts on the network will make any necessary cache updates. Finally, ACD requires that hosts that do detect a duplicate address must discontinue using it.

It is also possible for other stations to answer an ARP query on behalf of the actual destination D; this is called **proxy ARP**. An early common scenario for this was when host C on a LAN had a modem connected to a serial port. In theory a host D dialing in to this modem should be on a different subnet, but that requires allocation of a new subnet. Instead, many sites chose a simpler arrangement. A host that dialed in to C’s serial port

might be assigned IP address D_{IP} , from the same subnet as C. C would be configured to route packets to D; that is, packets arriving from the serial line would be forwarded to the LAN interface, and packets sent to C_{LAN} addressed to D_{IP} would be forwarded to D. But we also have to handle ARP, and as D is not actually on the LAN it will not receive broadcast ARP queries. Instead, C would be configured to answer on behalf of D, replying with (D_{IP}, C_{LAN}) . This generally worked quite well.

Proxy ARP is also used in Mobile IP, for the so-called “home agent” to intercept traffic addressed to the “home address” of a mobile device and then forward it (*eg* via tunneling) to that device. See [9.9 Mobile IP](#).

One delicate aspect of the ARP protocol is that stations are required to respond to a **broadcast** query. In the absence of proxies this theoretically should not create problems: there should be only one respondent. However, there were anecdotes from the Elder Days of networking when a broadcast ARP query would trigger an avalanche of responses. The protocol-design moral here is that determining who is to respond to a broadcast message should be done with great care. ([RFC 1122](#) section 3.2.2 addresses this same point in the context of responding to broadcast ICMP messages.)

ARP-query implementations also need to include a timeout and some queues, so that queries can be resent if lost and so that a burst of packets does not lead to a burst of queries. A naive ARP algorithm without these might be:

To send a packet to destination D_{IP} , see if D_{IP} is in the ARP cache. If it is, address the packet to D_{LAN} ; if not, send an ARP query for D

To see the problem with this approach, imagine that a 32 kB packet arrives at the IP layer, to be sent over Ethernet. It will be fragmented into 22 fragments (assuming an Ethernet MTU of 1500 bytes), all sent at once. The naive algorithm above will likely send an ARP query for *each* of these. What we need instead is something like the following:

To send a packet to destination D_{IP} :
 If D_{IP} is in the ARP cache, send to D_{LAN} and return
 If not, see if an ARP query for D_{IP} is pending.
 If it is, put the current packet in a queue for D.
 If there is no pending ARP query for D_{IP} , start one,
 again putting the current packet in the (new) queue for D

We also need:

If an ARP query for some C_{IP} times out, resend it (up to a point)
 If an ARP query for C_{IP} is answered, send off any packets in C’s queue

10.2.2 ARP Security

Suppose A wants to log in to secure server S, using a password. How can B (for Bad) impersonate S?

Here is an ARP-based strategy, sometimes known as **ARP Spoofing**. First, B makes sure the real S is down, either by waiting until scheduled downtime or by launching a denial-of-service attack against S.

When A tries to connect, it will begin with an ARP “who-has S?”. All B has to do is answer, “S is-at B”. There is a trivial way to do this: B simply needs to set its own IP address to that of S.

A will connect, and may be convinced to give its password to B. B now simply responds with something plausible like “backup in progress; try later”, and meanwhile use A’s credentials against the real S.

This works even if the communications channel A uses is encrypted! If A is using the SSH protocol ([29.5.1 SSH](#)), then A will get a message that the other side’s key has changed (B will present its own SSH key, not S’s). Unfortunately, many users (and even some IT departments) do not recognize this as a serious problem. Some organizations – especially schools and universities – use personal workstations with “frozen” configuration, so that the filesystem is reset to its original state on every reboot. Such systems may be resistant to viruses, but in these environments the user at A will always get a message to the effect that S’s credentials are not known.

10.2.3 ARP Failover

Suppose you have two front-line servers, A and B (B for Backup), and you want B to be able to step in if A freezes. There are a number of ways of achieving this, but one of the simplest is known as **ARP Failover**. First, we set $A_{IP} = B_{IP}$, but for the time being B does not use the network so this duplication is not a problem. Then, once B gets the message that A is down, it sends out an ARP query for A_{IP} , including B_{LAN} as the source LAN address. The gateway router, which previously would have had $\langle A_{IP}, A_{LAN} \rangle$ in its ARP cache, updates this to $\langle A_{IP}, B_{LAN} \rangle$, and packets that had formerly been sent to A will now go to B. As long as B is trafficking in stateless operations (*eg* html), B can pick up right where A left off.

10.2.4 Detecting Sniffers

Finally, there is an interesting use of ARP to detect Ethernet password sniffers (generally not quite the issue it once was, due to encryption and switching). To find out if a particular host A is in promiscuous mode, send an ARP “who-has A?” query. Address it not to the broadcast Ethernet address, though, but to some nonexistent Ethernet address.

If promiscuous mode is off, A’s network interface will ignore the packet. But if promiscuous mode is on, A’s network interface will pass the ARP request to A itself, which is likely then to answer it.

Alas, Linux kernels reject at the ARP–software level ARP queries to physical Ethernet addresses other than their own. However, they do respond to faked Ethernet multicast addresses, such as `ff:ff:ff:00:00:00` or `ff:ff:ff:ff:ff:fe`.

10.2.5 ARP and multihomed hosts

If host A has two interfaces `iface1` and `iface2` *on the same LAN*, with respective IP addresses A_1 and A_2 , then it is common for the two to be used interchangeably. Traffic addressed to A_1 may be received via `iface2` and vice-versa, and traffic *from* A_1 may be sent via `iface2`. In [9.2.1 Multihomed hosts](#) this is described as the **weak end–system** model; the idea is that we should think of the IP addresses A_1 and A_2 as bound to A rather than to their respective interfaces.

In support of this model, ARP can usually be configured (in fact this is often the default) so that ARP requests for either IP address and received by either interface may be answered with either physical address. Usually all requests are answered with the physical address of the preferred (*ie* faster) interface.

As an example, suppose A has an Ethernet interface `eth0` with IP address `10.0.0.2` and a faster Wi-Fi interface `wlan0` with IP address `10.0.0.3` (although Wi-Fi interfaces are *not* always faster). In this setting, an ARP request “who-has `10.0.0.2`” would be answered with `wlan0`’s physical address, and so all traffic to A, to either IP address, would arrive via `wlan0`. The `eth0` interface would go essentially unused. Similarly, though not due to ARP, traffic sent by A with source address `10.0.0.2` might depart via `wlan0`.

This situation is on Linux systems adjusted by changing `arp_ignore` and `arp_announce` in `/proc/sys/net/ipv4/conf/all`.

10.3 Dynamic Host Configuration Protocol (DHCP)

DHCP is the most common mechanism by which hosts are assigned their IPv4 addresses. DHCP started as a protocol known as Reverse ARP (RARP), which evolved into BOOTP and then into its present form. It is documented in [RFC 2131](#). Recall that ARP is based on the idea of someone broadcasting an ARP query for a host, containing the host’s IPv4 address, and the host answering it with its LAN address. DHCP involves a host, at startup, broadcasting a query containing its *own* LAN address, and having a server reply telling the host what IPv4 address is assigned to it, hence the “Reverse ARP” name.

The DHCP response message is also likely to carry, piggybacked onto it, several other essential startup options. Unlike the IPv4 address, these additional network parameters usually do not depend on the specific host that has sent the DHCP query; they are likely constant for the subnet or even the site. In all, a typical DHCP message includes the following:

- IPv4 address

- subnet mask
- default router
- DNS Server

These four items are a standard **minimal network configuration**; in practical terms, hosts cannot function properly without them. Most DHCP implementations support the piggybacking of the latter three above, and a wide variety of other configuration values, onto the server responses.

Default Routers and DHCP

If you lose your default router, you cannot communicate. Here is something that used to happen to me, courtesy of DHCP:

1. I am connected to the Internet via Ethernet, and my default router is via my Ethernet interface
2. I connect to my institution's wireless network.
3. Their DHCP server sends me a new default router on the wireless network. However, this default router will only allow access to a tiny private network, because I have neglected to complete the "Wi-Fi network registration" process.
4. I therefore disconnect from the wireless network, and my wireless-interface default router goes away. However, my system does not automatically revert to my Ethernet default-router entry; DHCP does not work that way. As a result, I will have no router at all until the next scheduled DHCP lease renegotiation, and must fix things manually.

The DHCP server has a range of IPv4 addresses to hand out, and maintains a database of which IPv4 address has been assigned to which LAN address. Reservations can either be permanent or dynamic; if the latter, hosts typically renew their DHCP reservation periodically (typically one to several times a day).

10.3.1 NAT, DHCP and the Small Office

If you have a large network, with multiple subnets, a certain amount of manual configuration is inevitable. What about, however, a home or small office, with a single line from an ISP? A combination of NAT ([9.7 Network Address Translation](#)) and DHCP has made **autoconfiguration** close to a reality.

The typical home/small-office "router" is in fact a NAT router ([9.7 Network Address Translation](#)) coupled with an Ethernet switch, and usually also coupled with a Wi-Fi access point and a DHCP server. In this section, we will use the term "NAT router" to refer to this whole package. One specially designated port, the **external** port, connects to the ISP's line, and uses DHCP as a client to obtain an IPv4 address for that port. The other, **internal**, ports are connected together by an Ethernet switch; these ports as a

group are connected to the external port using NAT translation. If wireless is supported, the wireless side is connected directly to the internal ports.

Isolated from the Internet, the internal ports can thus be assigned an arbitrary non-public IPv4 address block, *eg* 192.168.0.0/24. The NAT router typically contains a DHCP server, usually enabled by default, that will hand out IPv4 addresses to everything connecting from the internal side.

Generally this works seamlessly. However, if a second NAT router is also connected to the network (sometimes attempted to extend Wi-Fi range, in lieu of a commercial Wi-Fi repeater), one then has two operating DHCP servers on the same subnet. This often results in chaos, though is easily fixed by disabling one of the DHCP servers.

While omnipresent DHCP servers have made IPv4 autoconfiguration work “out of the box” in many cases, in the era in which IPv4 was designed the need for such servers would have been seen as a significant drawback in terms of expense and reliability. IPv6 has an autoconfiguration strategy ([11.7.2 Stateless Autoconfiguration \(SLAAC\)](#)) that does not require DHCP, though DHCPv6 may well end up displacing it.

10.3.2 DHCP and Routers

It is often desired, for larger sites, to have only one or two DHCP servers, but to have them support multiple subnets. Classical DHCP relies on broadcast, which isn’t forwarded by routers, and even if it were, the DHCP server would have no way of knowing on what subnet the host in question was actually located.

This is generally addressed by **DHCP Relay** (sometimes still known by the older name BOOTP Relay). The router (or, sometimes, some other node on the subnet) receives the DHCP broadcast message from a host, and notes the subnet address of the arrival interface. The router then relays the DHCP request, together with this subnet address, to the designated DHCP Server; this relayed message is sent directly (unicast), not broadcast. Because the subnet address is included, the DHCP server can figure out the correct IPv4 address to assign.

This feature has to be specially enabled on the router.

10.4 Internet Control Message Protocol

The Internet Control Message Protocol, or ICMP, is a protocol for sending IP-layer error and status messages; it is defined in [RFC 792](#). ICMP is, like IP, **host-to-host**, and so they are never delivered to a specific port, even if they are sent in response to an error related to something sent from that port. In other words, individual UDP and TCP connections do not receive ICMP messages, even when it would be helpful to get them.

ICMP messages are identified by an 8-bit **type** field, followed by an 8-bit subtype, or **code**. Here are the more common ICMP types, with subtypes listed in the description.

Type	Description
Echo Request	ping queries
Echo Reply	ping responses
Destination Unreachable	Destination network unreachable
	Destination host unreachable
	Destination port unreachable
	Fragmentation required but DF flag set
	Network administratively prohibited
Source Quench	Congestion control
Redirect Message	Redirect datagram for the network
	Redirect datagram for the host
	Redirect for TOS and network
	Redirect for TOS and host
Router Solicitation	Router discovery/selection/solicitation
Time Exceeded	TTL expired in transit
	Fragment reassembly time exceeded
Bad IP Header or Parameter	Pointer indicates the error
	Missing a required option
	Bad length
Timestamp Timestamp Reply	Like ping, but requesting a timestamp from the destination

The Echo and Timestamp formats are *queries*, sent by one host to another. Most of the others are all *error messages*, sent by a router to the sender of the offending packet. Error-message formats contain the IP header and next 8 bytes of the packet in question; the 8 bytes will contain the TCP or UDP port numbers. Redirect and Router Solicitation messages are informational, but follow the error-message format. Query formats contain a 16-bit *Query Identifier*, assigned by the query sender and echoed back by the query responder.

ping Packet Size

The author once had to diagnose a problem where pings were *almost* 100% successful, and yet file transfers failed immediately; this could have been the result of either a network fault or a file-transfer application fault. The problem turned out to be a failed network device with a very high bit-error rate: 1500-byte file-transfer packets were frequently corrupted, but ping packets, with a default size of 32–64 bytes, were mostly unaffected. If the bit-error rate is such that 1500-byte packets have a 50% success rate, 50-byte packets can be expected to have a 98% ($\approx 0.5^{1/30}$) success rate. Setting the ping packet size to a larger value made it immediately clear that the network, and not the file-transfer application, was at fault.

ICMP is perhaps best known for Echo Request/Reply, on which the ping tool ([1.14 Some Useful Utilities](#)) is based. Ping remains very useful for network troubleshooting: if you can ping a host, then the network is reachable, and any problems are higher up the protocol chain. Unfortunately, ping replies are often blocked by many firewalls, on the

theory that revealing even the existence of computers is a security risk. While this may sometimes be an appropriate decision, it does significantly impair the utility of ping.

Ping can be asked to include IP timestamps ([9.1 The IPv4 Header](#)) on Linux systems with the `-T` option, and on Windows with `-s`.

Source Quench was used to signal that congestion has been encountered. A router that drops a packet due to congestion experience was encouraged to send ICMP Source Quench to the originating host. Generally the TCP layer would handle these appropriately (by reducing the overall sending rate), but UDP applications never receive them. ICMP Source Quench did not quite work out as intended, and was formally deprecated by [RFC 6633](#). (Routers can inform TCP connections of impending congestion by using the ECN bits.)

The Destination Unreachable type has a large number of subtypes:

- **Network unreachable:** some router had no entry for forwarding the packet, and no default route
- **Host unreachable:** the packet reached a router that was on the same LAN as the host, but the host failed to respond to ARP queries
- **Port unreachable:** the packet was sent to a UDP port on a given host, but that port was not open. TCP, on the other hand, deals with this situation by replying to the connecting endpoint with a reset packet. Unfortunately, the UDP Port Unreachable message is sent to the host, not to the application on that host that sent the undeliverable packet, and so is close to useless as a practical way for applications to be informed when packets cannot be delivered.
- **Fragmentation required but DF flag set:** a packet arrived at a router and was too big to be forwarded without fragmentation. However, the Don't Fragment bit in the IPv4 header was set, forbidding fragmentation.
- **Administratively Prohibited:** this is sent by a router that knows it can reach the network in question, but has configured to drop the packet and send back Administratively Prohibited messages. A router can also be configured to **blackhole** messages: to drop the packet and send back nothing.

In [18.6 Path MTU Discovery](#) we will see how TCP uses the ICMP message **Fragmentation required but DF flag set** as part of **Path MTU Discovery**, the process of finding the largest packet that can be sent *to a specific destination* without fragmentation. The basic idea is that we set the DF bit on some of the packets we send; if we get back this message, that packet was too big.

Some sites and firewalls block ICMP packets in addition to Echo Request/Reply, and for some messages one can get away with this with relatively few consequences. However, blocking **Fragmentation required but DF flag set** has the potential to severely affect TCP connections, depending on how Path MTU Discovery is implemented, and thus is not recommended. If ICMP filtering is contemplated, it is best to base block/allow decisions

on the ICMP type, or even on the type and code. For example, most firewalls support rule sets of the form “allow ICMP destination-unreachable; block all other ICMP”.

The **Timestamp** option works something like Echo Request/Reply, but the receiver includes its own local timestamp for the arrival time, with millisecond accuracy. See also the IP Timestamp option, [9.1 The IPv4 Header](#), which appears to be more frequently used.

The type/code message format makes it easy to add new ICMP types. Over the years, a significant number of additional such types have been defined; a [complete list](#) is maintained by the IANA. Several of these later ICMP types were seldom used and eventually deprecated, many by [RFC 6918](#).

ICMP packets are usually forwarded correctly through NAT routers, though due to the absence of port numbers the router must do a little more work. [RFC 3022](#) and [RFC 5508](#) address this. For ICMP queries, like ping, the ICMP Query Identifier field can be used to recognize the returning response. ICMP error messages are a little trickier, because there is no direct connection between the inbound error message and any of the previous outbound non-ICMP packets that triggered the response. However, the headers of the packet that triggered the ICMP error message are embedded in the body of the ICMP message. The NAT router can look at those embedded headers to determine how to forward the ICMP message (the NAT router must also rewrite the addresses of those embedded headers).

10.4.1 Traceroute and Time Exceeded

The traceroute program uses ICMP Time Exceeded messages. A packet is sent to the destination (often UDP to an unused port), with the TTL set to 1. The first router the packet reaches decrements the TTL to 0, drops it, and returns an ICMP Time Exceeded message. The sender now knows the first router on the chain. The second packet is sent with TTL set to 2, and the second router on the path will be the one to return ICMP Time Exceeded. This continues until finally the remote host returns something, likely ICMP Port Unreachable.

For an example of traceroute output, see [1.14 Some Useful Utilities](#). In that example, the three traceroute probes for the Nth router are sometimes answered by two or even three different routers; this suggests routers configured to work in parallel rather than route changes.

Many routers no longer respond with ICMP Time Exceeded messages when they drop packets. For the distance value corresponding to such a router, traceroute reports ***.

Traceroute assumes the path does not change. This is not always the case, although in practice it is seldom an issue.

Route Efficiency

Once upon a time (~2001), traceroute showed that traffic from my home to the office, both in the Chicago area, went through the MAE-EAST Internet exchange point, outside of Washington DC. That inefficient route was later fixed. A situation like this is typically caused by two higher-level providers who did not negotiate sufficient Internet exchange points.

Traceroute to a nonexistent site works up to the point when the packet reaches the Internet “backbone”: the first router which does not have a default route. At that point the packet is not routed further (and an ICMP Destination Network Unreachable should be returned).

Traceroute also interacts somewhat oddly with routers using MPLS (see [25.12 Multi-Protocol Label Switching \(MPLS\)](#)). Such routers – most likely large-ISP internal routers – may continue to forward the ICMP Time Exceeded message on further towards its destination before returning it to the sender. As a result, the round-trip time measurements reported may be quite a bit larger than they should be.

10.4.2 Redirects

Most non-router hosts start up with an IPv4 forwarding table consisting of a single (default) router, discovered along with their IPv4 address through DHCP. ICMP Redirect messages help hosts learn of other useful routers. Here is a classic example:

A is configured so that its default router is R1. It addresses a packet to B, and sends it to R1. R1 receives the packet, and forwards it to R2. However, R1 also notices that R2 and A are on the same network, and so A could have sent the packet to R2 directly. So R1 sends an appropriate ICMP redirect message to A (“Redirect Datagram for the Network”), and A adds a route to B via R2 to its own forwarding table.

10.4.3 Router Solicitation

These ICMP messages are used by some router protocols to identify immediate neighbors. When we look at routing-update algorithms, [13 Routing-Update Algorithms](#), these are where the process starts.

10.5 Epilog

At this point we have concluded the basic mechanics of IPv4. Still to come is a discussion of how IP routers build their forwarding tables. This turns out to be a complex topic, divided into routing within single organizations and ISPs – [13 Routing-Update Algorithms](#) – and routing between organizations – [14 Large-Scale IP Routing](#).

But before that, in the next chapter, we compare IPv4 with IPv6, now twenty years old but still seeing limited adoption. The biggest issue fixed by IPv6 is IPv4's lack of address space, but there are also several other less dramatic improvements.

10.6 Exercises

Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises.

1.0. In [10.2 Address Resolution Protocol: ARP](#) it was stated that, in newer implementations, “repeat ARP queries about a timed out entry are first sent unicast”, in order to reduce broadcast traffic. Suppose host A uses ARP to retrieve B's LAN address (MAC address). A short time later, B *changes* its LAN address, either through a hardware swap or through software reconfiguration.

- (a). What will happen if A now sends a *unicast* repeat ARP query for B?
- (b). What will happen if A now sends a *broadcast* repeat ARP query for B?

2.0. Suppose A broadcasts an ARP query “who-has B?”, receives B's response, and proceeds to send B a regular IPv4 packet. If B now wishes to reply, why is it likely that A will already be present in B's ARP cache? Identify a circumstance under which this can fail.

3.0. Suppose A broadcasts an ARP request “who-has B”, but inadvertently lists the physical address of another machine C instead of its own (that is, A's ARP query has $IP_{src} = A$, but $LAN_{src} = C$). What will happen? Will A receive a reply? Will any other hosts on the LAN be able to send to A? What entries will be made in the ARP caches on A, B and C?

4.0. Suppose host A connects to the Internet via Wi-Fi. The default router is R_w . Host A now begins exchanging packets with a remote host B: A sends to B, B replies, *etc.* The exact form of the connection does not matter, except that TCP may not work.

(a). You now plug in A's Ethernet cable. The Ethernet port is assumed to be on a different subnet from the Wi-Fi (so that the strong and weak end-system models of [10.2.5 ARP and multihomed hosts](#) do not play a role here). Assume A automatically selects the new Ethernet connection as its default route, with router R_E . What happens to the original connection to A? Can packets still travel back and forth? Does the return address used for either direction change?

(b). You now *disconnect* A's Wi-Fi interface, leaving the Ethernet interface connected. What happens now to the connection to B? Hint: to what IP address are the packets from B being sent?

11 IPv6

What has been learned from experience with IPv4? First and foremost, more than 32 bits are needed for addresses; the primary motive in developing the new version of IP known as IPv6 was the specter of running out of IPv4 addresses (something which, at the highest level, has already happened; see the discussion at the end of [1.10 IP – Internet Protocol](#)). Another important issue is that IPv4 requires (or used to require) a modest amount of effort at configuration; IPv6 was supposed to improve this.

In this chapter we outline the basic format of IPv6 packets, including address format and address assignment. The following chapter continues with additional features of IPv6.

By 1990 the IPv4 address-space issue was well understood, and the IETF was actively interested in proposals to replace IPv4. A working group for the so-called “IP next generation”, or IPng, was created in 1993 to select the new version; [RFC 1550](#) was this group’s formal solicitation of proposals. In July 1994 the IPng directors voted to accept a modified version of the “Simple Internet Protocol Plus”, or SIPP ([RFC 1710](#)), as the basis for IPv6. The first IPv6 specifications, released in 1995, were [RFC 1883](#) (now [RFC 2460](#), with updates) for the basic protocol, and [RFC 1884](#) (now [RFC 4291](#), again with updates) for the addressing architecture.

SIPP addresses were originally 64 bits in length, but in the month leading up to adoption as the basis for IPv6 this was increased to 128. 64 bits would probably have been enough, but the problem is less the actual number than the simplicity with which addresses can be allocated; the more bits, the easier this becomes, as sites can be given relatively large address blocks without fear of waste. A secondary consideration in the 64-to-128 leap was the potential to accommodate now-obsolete CLNP addresses ([1.15 IETF and OSI](#)), which were up to 160 bits in length, but compressible.

IPv6 has to some extent returned to the idea of a fixed division between network and host portions; for most IPv6 addresses, the first 64 bits is the network prefix (including any subnet portion) and the remaining 64 bits represents the host portion. The rule as spelled out in [RFC 2460](#), in 1998, was that the 64/64 split would apply to all addresses except those beginning with the bits 000; those addresses were then held in reserve in the unlikely event that the 64/64 split ran into problems in the future. This was a change from 1995, when [RFC 1884](#) envisioned 48-bit host portions and 80-bit prefixes.

While the IETF occasionally revisits the issue, at the present time the 64/64 split seems here to stay; for discussion and justification, see [12.3.1 Subnets and /64](#) and [RFC 7421](#). The 64/64 split is not automatic, however; there is no default prefix length as there was in the Class A/B/C IPv4 scheme. Thus, it is misleading to think of IPv6 as a return to something like IPv4’s classful addressing scheme. Router advertisements must always include the prefix length, and, when assigning IPv6 addresses manually, the /64 prefix length must be specified explicitly; see [12.5.3 Manual address configuration](#).

High-level routing, however, can, as in IPv4, be done on prefixes of any length (usually that means lengths shorter than /64). Routing can also be done on different prefix lengths at different points of the network.

IPv6 is now twenty years old, and yet usage as of 2015 remains quite modest. However, the shortage in IPv4 addresses has begun to loom ominously; IPv6 adoption rates may rise quickly if IPv4 addresses begin to climb in price.

11.1 The IPv6 Header

The IPv6 **fixed header** is pictured below; at 40 bytes, it is twice the size of the IPv4 header. The fixed header is intended to support only what *every* packet needs: there is no support for fragmentation, no header checksum, and no option fields. However, the concept of **extension headers** has been introduced to support some of these as options; some IPv6 extension headers are described in [11.5 IPv6 Extension Headers](#). Whatever header comes next is identified by the Next Header field, much like the IPv4 Protocol field. Some other fixed-header fields have also been renamed from their IPv4 analogues: the IPv4 TTL is now the IPv6 Hop_Limit (still decremented by each router with the packet discarded when it reaches 0), and the IPv4 DS field has become the IPv6 Traffic Class.

The Flow Label is new. [RFC 2460](#) states that it

may be used by a source to label sequences of packets for which it requests special handling by the IPv6 routers, such as non-default quality of service or “real-time” service.

Senders not actually taking advantage of any quality-of-service options are supposed to set the Flow Label to zero.

When used, the Flow Label represents a sender-computed hash of the source and destination addresses, and perhaps the traffic class. Routers can use this field as a way to look up quickly any priority or reservation state for the packet. All packets belonging to the same flow should have the same Routing Extension header, [11.5.3 Routing Header](#). The Flow Label will in general *not* include any information about the source and destination *port* numbers, except that only some of the connections between a pair of hosts may make use of this field.

A **flow**, as the term is used here, is *one-way*; the return traffic belongs to a different flow. Historically, the term “flow” has also been used at various other scales: a single bidirectional TCP connection, multiple *related* TCP connections, or even all traffic from a particular subnet (*eg* the “computer-lab flow”).

11.2 IPv6 Addresses

IPv6 addresses are written in eight groups of four hex digits, with a–f preferred over A–F ([RFC 5952](#)). The groups are separated by colons, and have leading 0’s removed, *eg*

```
fedc:13:1654:310:fedc:bc37:61:3210
```

If an address contains a long run of 0’s – for example, if the IPv6 address had an embedded IPv4 address – then when writing the address the string “::” should be used to represent however many blocks of 0000 as are needed to create an address of the correct length; to avoid ambiguity this can be used only once. Also, embedded IPv4 addresses may continue to use the “.” separator:

```
::ffff:147.126.65.141
```

The above is an example of one standard IPv6 format for representing IPv4 addresses (see [12.4 Using IPv6 and IPv4 Together](#)). 48 bits are explicitly displayed; the :: means these are prefixed by 80 0–bits.

The IPv6 loopback address is ::1 (that is, 127 0–bits followed by a 1–bit).

Network address **prefixes** may be written with the “/” notation, as in IPv4:

```
12ab:0:0:cd30::/60
```

[RFC 3513](#) suggested that initial IPv6 unicast–address allocation be initially limited to addresses beginning with the bits 001, that is, the 2000::/3 block (20 in binary is 0010 0000).

Generally speaking, IPv6 addresses consist of a 64–bit network prefix (perhaps including subnet bits) followed by a 64–bit “interface identifier”. See [11.3 Network Prefixes](#) and [11.2.1 Interface identifiers](#).

IPv6 addresses all have an associated **scope**, defined in [RFC 4007](#). The scope of a unicast address is either **global**, meaning it is intended to be globally routable, or **link–local**, meaning that it will only work with directly connected neighbors ([11.2.2 Link–local addresses](#)). The loopback address is considered to have link–local scope. A few more scope levels are available for multicast addresses, *eg* “site–local” ([RFC 4291](#)). The scope of an IPv6 address is implicitly coded within the first 64 bits; addresses in the 2000::/3 block above, for example, have global scope.

Packets with local–scope addresses (*eg* link–local addresses) for either the destination or the source cannot be routed (the latter because a reply would be impossible).

Although addresses in the “unique local address” category of [11.3 Network Prefixes](#) officially have global scope, in a practical sense they still behave as if they had the now–officially–deprecated “site–local scope”.

11.2.1 Interface identifiers

As mentioned earlier, most IPv6 addresses can be divided into a 64-bit network prefix and a 64-bit “host” portion, the latter corresponding to the “host” bits of an IPv4 address. These host-portion bits are known officially as the **interface identifier**; the change in terminology reflects the understanding that all IP addresses attach to interfaces rather than to hosts.

The original plan for the interface identifier was to derive it in most cases from the LAN address, though the interface identifier can also be set administratively. Given a 48-bit Ethernet address, the interface identifier based on it was to be formed by inserting 0xffff between the first three bytes and the last three bytes, to get 64 bits in all. The seventh bit of the first byte (the Ethernet “universal/local” flag) was then set to 1. The result of this process is officially known as the **Modified EUI-64 Identifier**, where EUI stands for Extended Unique Identifier; details can be found in [RFC 4291](#). As an example, for a host with Ethernet address 00:a0:cc:24:b0:e4, the EUI-64 identifier would be 02a0:ccff:fe24:b0e4 (the leading 00 becomes 02 when the seventh bit is turned on). At the time the EUI-64 format was proposed, it was widely expected that Ethernet MAC addresses would eventually become 64 bits in length.

EUI-64 interface identifiers have long been recognized as a major privacy concern: no matter where a (portable) host connects to the Internet – home or work or airport or Internet cafe – such an interface identifier always remains the same, and thus serves as a permanent host fingerprint. As a result, EUI-64 identifiers are now discouraged for personal workstations and mobile devices. (Some fixed-location hosts continue to use EUI-64 interface identifiers, or, alternatively, administratively assigned interface identifiers.)

While these general issues are alone enough to warrant abandoning EUI-64 identifiers, there are, in fact, much more serious risks, such as the **IPvSeeYou** vulnerability of [\[RB22\]](#). Consider a budget home router combined with Wi-Fi access point that uses EUI-64 addresses; many such devices remain on the market, and many more are currently in use and offer no upgrade path. There is an excellent chance that all MAC addresses for this router – both Ethernet and Wi-Fi – are assigned sequentially (or they are related in a way determined by the OUI). Even if a user is using a privacy-protecting interface identifier when connecting the internet, the router’s MAC address can be exposed: an IPv6 traceroute to the user’s IPv6 address will reveal the router’s public-facing IPv6 address, as the last hop before the user’s own address. From this – via EUI-64 – the public-facing Ethernet MAC address is easily found. This router MAC address, in turn, determines the router’s Wi-Fi MAC address to within a handful of values. Finally, *maps exist of the physical GPS location of almost all Wi-Fi MAC addresses*, obtained via so-called “war-driving” (driving around scanning for Wi-Fi-access-point MAC addresses and recording the GPS coordinates of each); see, for example, [wigle.net](#). Thus, from the user’s non-EUI privacy-implementing IPv6 address, the user’s real home location is straightforward to determine.

[RFC 7217](#) proposes a privacy-improving alternative to EUI-64 identifiers: the interface identifier is a secure hash ([28.6 Secure Hashes](#)) of a so-called “Net_Iface” parameter, the 64-bit IPv6 address prefix, and a host-specific secret key (a couple other parameters are also thrown into the mix, but they need not concern us here). The “Net_Iface” parameter can be the interface’s MAC address, but can also be the interface’s “name”, *eg* eth0. Interface identifiers created this way change from connection point to connection point (because the prefix changes), do not reveal the Ethernet address, and are randomly scattered (because of the key, if nothing else) through the 2^{64} -sized interface-identifier space. The last feature makes probing for IPv6 addresses effectively impossible; see exercise 4.0.

Interface identifiers as in the previous paragraph do not change unless the prefix changes, which normally happens only if the host is moved to a new network.

In [11.7.2.1 SLAAC privacy](#) we will see that interface identifiers are often changed at regular intervals, for privacy reasons.

Finally, interface identifiers are often centrally assigned, using DHCPv6 ([11.7.3 DHCPv6](#)).

Remote probing for IPv6 addresses based on EUI-64 identifiers is much easier than for those based on RFC-7217 identifiers, as the former are not very random. If an attacker can guess the hardware vendor, and thus the first three bytes of the Ethernet address ([2.1.3 Ethernet Address Internal Structure](#)), there are only 2^{24} possibilities, down from 2^{64} . As the last three bytes are often assigned in serial order, considerable further narrowing of the search space may be possible. While it may amount to [security through obscurity](#), keeping internal global IPv6 addresses hidden is often of practical importance.

Additional discussion of host-scanning in IPv6 networks can be found in [RFC 7707](#) and [draft-ietf-opsec-ipv6-host-scanning-06](#).

11.2.2 Link-local addresses

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

The main applications of link-local addresses are as a “bootstrap” address for global-address autoconfiguration ([11.7.2 Stateless Autoconfiguration \(SLAAC\)](#)), and as an optional permanent address for routers. IPv6 routers often communicate with neighboring routers via their link-local addresses, with the understanding that these do not change when global addresses (or subnet configurations) change ([RFC 4861](#) §6.2.8). If EUI-64 interface identifiers are used then the link-local address does change whenever

the Ethernet hardware is replaced. However, if [RFC 7217](#) interface identifiers are used and that mechanism's "Net_Iface" parameter represents the interface name rather than its physical address, the link-local address can be constant for the life of the host. (When RFC 7217 is used to generate link-local addresses, the "prefix" hash parameter is the link-local prefix fe80::/64.)

A consequence of identifying routers to their neighbors by their link-local addresses is that it is often possible to configure routers so they do not even have global-scope addresses; for forwarding traffic and for exchanging routing-update messages, link-local addresses are sufficient. Similarly, many ordinary hosts forward packets to their default router using the latter's link-local address. We will return to router addressing in [12.6.2 Setting up a router](#) and [12.6.2.1 A second router](#).

For non-Ethernet-like interfaces, *eg* tunnel interfaces, there may be no natural candidate for the interface identifier, in which case a link-local address *may* be assigned manually, with the low-order 64 bits chosen to be unique for the link in question.

When sending to a link-local address, one must separately supply somewhere the link's "zone identifier", often by appending a string containing the interface name to the IPv6 address, *eg* fe80::f00d:cafe%eth0. See [12.5.1 ping6](#) and [12.5.2 TCP connections using link-local addresses](#) for examples of such use of link-local addresses.

IPv4 also has true link-local addresses, defined in [RFC 3927](#), though they are rarely used; such addresses are in the 169.254.0.0/16 block (not to be confused with the 192.168.0.0/16 private-address block). Other than these, IPv4 addresses always implicitly identify the link subnet by virtue of the network prefix.

Once the link-local address is created, it must pass the **duplicate-address detection** test before being used; see [11.7.1 Duplicate Address Detection](#).

11.2.3 Anycast addresses

IPv6 also introduced **anycast** addresses. An anycast address might be assigned to each of a set of routers (in addition to each router's own unicast addresses); a packet addressed to this anycast address would be delivered to only one member of this set. Note that this is quite different from multicast addresses; a packet addressed to the latter is delivered to *every* member of the set.

It is up to the local routing infrastructure to decide which member of the anycast group would receive the packet; normally it would be sent to the "closest" member. This allows hosts to send to any of a set of routers, rather than to their designated individual default router.

Anycast addresses are not marked as such, and a node sending to such an address need not be aware of its anycast status. Addresses are anycast simply because the routers involved have been configured to recognize them as such.

IPv4 anycast exists also, but in a more limited form ([15.8 BGP and Anycast](#)); generally routers are configured much more indirectly (*eg* through BGP).

11.3 Network Prefixes

We have been assuming that an IPv6 address, at least as seen by a host, is composed of a 64-bit network prefix and a 64-bit interface identifier. As of 2015 this remains a requirement; [RFC 4291](#) (IPv6 Addressing Architecture) states:

For all unicast addresses, except those that start with the binary value 000, Interface IDs are required to be 64 bits long....

This /64 requirement is occasionally revisited by the IETF, but is unlikely to change for mainstream IPv6 traffic. This firm 64/64 split is a departure from IPv4, where the host/subnet division point has depended, since the development of subnets, on local configuration.

Note that while the net/interface (net/host) division point is fixed, routers may still use CIDR ([14.1 Classless Internet Domain Routing: CIDR](#)) and may still base forwarding decisions on prefixes shorter than /64.

As of 2015, all allocations for globally routable IPv6 prefixes are part of the 2000::/3 block.

IPv6 also defines a variety of specialized network prefixes, including the link-local prefix and prefixes for anycast and multicast addresses. For example, as we saw earlier, the prefix ::ffff:0:0/96 identifies IPv6 addresses with embedded IPv4 addresses.

The most important class of 64-bit network prefixes, however, are those supplied by a provider or other address-numbering entity, and which represent the first half of globally routable IPv6 addresses. These are the prefixes that will be visible to the outside world.

IPv6 customers will typically be assigned a relatively large block of addresses, *eg* /48 or /56. The former allows $64 - 48 = 16$ bits for local “subnet” specification within a 64-bit network prefix; the latter allows 8 subnet bits. These subnet bits are – as in IPv4 – supplied through router configuration; see [12.3 IPv6 Subnets](#). The closest IPv6 analogue to the IPv4 subnet mask is that all network prefixes are supplied to hosts with an associated length, although that length will almost always be 64 bits.

Many sites will have only a single externally visible address block. However, some sites may be multihomed and thus have multiple independent address blocks.

Sites may also have private **unique local address** prefixes, corresponding to IPv4 private address blocks like 192.168.0.0/16 and 10.0.0.0/8. They are officially called Unique Local Unicast Addresses and are defined in [RFC 4193](#). These replace an earlier **site-local** address plan (and official site-local scope) formally deprecated in [RFC](#)

3879 (though unique-local addresses are sometimes still informally referred to as site-local).

The first 8 bits of a unique-local prefix are 1111 1101 (fd00::/8). The related prefix 1111 1100 (fc00::/8) is reserved for future use; the two together may be consolidated as fc00::/7. The last 16 bits of a 64-bit unique-local prefix represent the subnet ID, and are assigned either administratively or via autoconfiguration. The 40 bits in between, from bit 8 up to bit 48, represent the **Global ID**. A site is to set the Global ID to a pseudorandom value.

The resultant unique-local prefix is “almost certainly” globally unique (and is considered to have **global scope** in the sense of **11.2 IPv6 Addresses**), although it is not supposed to be routed off a site. Furthermore, a site would generally not admit any packets from the outside world addressed to a destination with the Global ID as prefix. One rationale for choosing unique Global IDs for each site is to accommodate potential later mergers of organizations without the need for renumbering; this has been a chronic problem for sites using private IPv4 address blocks. Another justification is to accommodate VPN connections from other sites. For example, if I use IPv4 block 10.0.0.0/8 at home, and connect using VPN to a site also using 10.0.0.0/8, it is possible that my printer will have the same IPv4 address as their application server.

11.4 IPv6 Multicast

IPv6 has moved away from LAN-layer *broadcast*, instead providing a wide range of LAN-layer *multicast* groups. (Note that LAN-layer multicast is often straightforward; it is general IP-layer multicast (**25.5 Global IP Multicast**) that is problematic.

See **2.1.2 Ethernet Multicast** for the Ethernet implementation.) This switch to multicast is intended to limit broadcast traffic in general, though many switches still propagate LAN multicast traffic everywhere, like broadcast.

An IPv6 multicast address is one beginning with the eight bits 1111 1111 (ff00::/8); numerous specific such addresses, and even classes of addresses, have been defined. For actual delivery, IPv6 multicast addresses correspond to LAN-layer (eg Ethernet) multicast addresses through a well-defined static correspondence; specifically, if x, y, z and w are the last four bytes of the IPv6 multicast address, in hex, then the corresponding Ethernet multicast address is 33:33:x:y:z:w (**RFC 2464**). A typical IPv6 host will need to join (that is, subscribe to) several Ethernet multicast groups.

The IPv6 multicast address with the broadest scope is **all-nodes**, with address ff02::1; the corresponding Ethernet multicast address is 33:33:00:00:00:01. This essentially corresponds to IPv4’s LAN broadcast, though the use of LAN multicast here means that non-IPv6 hosts should not see packets sent to this address. Another important IPv6 multicast address is ff02::2, the **all-routers** address. This is meant to be used to reach all routers, and routers only; ordinary hosts do not subscribe.

Generally speaking, IPv6 nodes on Ethernets send LAN-layer **Multicast Listener Discovery** (MLD) messages to multicast groups they wish to start using; these messages allow multicast-aware Ethernet switches to optimize forwarding so that only those hosts that have subscribed to the multicast group in question will receive the messages. Otherwise switches are supposed to treat multicast like broadcast; worse, some switches may simply fail to forward multicast packets to destinations that have not explicitly opted to join the group.

11.5 IPv6 Extension Headers

In IPv4, the IP header contained a Protocol field to identify the next header; usually UDP or TCP. All IPv4 options were contained in the IP header itself. IPv6 has replaced this with a scheme for allowing an arbitrary chain of supplemental IPv6 headers. The IPv6 Next Header field *can* indicate that the following header is UDP or TCP, but can also indicate one of several IPv6 options. These optional, or extension, headers include:

- Hop-by-Hop options header
- Destination options header
- Routing header
- Fragment header
- Authentication header
- Mobility header
- Encapsulated Security Payload header

These extension headers must be processed in order; the recommended order for inclusion is as above. Most of them are intended for processing only at the destination host; the hop-by-hop and routing headers are exceptions.

11.5.1 Hop-by-Hop Options Header

This consists of a set of (type,value) pairs which are intended to be processed by each router on the path. A tag in the type field indicates what a router should do if it does not understand the option: drop the packet, or continue processing the rest of the options. The only Hop-by-Hop options provided by [RFC 2460](#) were for padding, so as to set the alignment of later headers.

[RFC 2675](#) later defined a Hop-by-Hop option to support IPv6 **jumbograms**: datagrams larger than 65,535 bytes. The need for such large packets remains unclear, in light of [7.3 Packet Size](#). IPv6 jumbograms are not meant to be used if the underlying LAN does not have an MTU larger than 65,535 bytes; the LAN world is not currently moving in this direction.

Because Hop-by-Hop Options headers must be processed by each router encountered, they have the potential to overburden the Internet routing system. As a result, [RFC](#)

[6564](#) strongly discourages new Hop-by-Hop Option headers, unless examination at every hop is essential.

11.5.2 Destination Options Header

This is very similar to the Hop-by-Hop Options header. It again consists of a set of (type,value) pairs, and the original [RFC 2460](#) specification only defined options for padding. The Destination header is intended to be processed at the destination, before turning over the packet to the transport layer.

Since [RFC 2460](#), a few more Destination Options header types have been defined, though none is in common use. [RFC 2473](#) defined a Destination Options header to limit the nesting of tunnels, called the Tunnel Encapsulation Limit. [RFC 6275](#) defines a Destination Options header for use in Mobile IPv6. [RFC 6553](#), on the Routing Protocol for Low-Power and Lossy Networks, or RPL, has defined a Destination (and Hop-by-Hop) Options type for carrying RPL data.

A complete list of Option Types for Hop-by-Hop Option and Destination Option headers can be found at www.iana.org/assignments/ipv6-parameters; in accordance with [RFC 2780](#).

11.5.3 Routing Header

The original, or Type 0, Routing header contained a list of IPv6 addresses through which the packet should be routed. These did not have to be contiguous. If the list to be visited en route to destination D was $\langle R1, R2, \dots, Rn \rangle$, then this option header contained $\langle R2, R3, \dots, Rn, D \rangle$ with R1 as the initial destination address; R1 then would update this header to $\langle R1, R3, \dots, Rn, D \rangle$ (that is, the old destination R1 and the current next-router R2 were swapped), and would send the packet on to R2. This was to continue on until Rn addressed the packet to the final destination D. The header contained a Segments Left pointer indicating the next address to be processed, incremented at each Ri. When the packet arrived at D the Routing Header would contain the routing list $\langle R1, R3, \dots, Rn \rangle$. This is, in general principle, very much like IPv4 Loose Source routing. Note, however, that routers *between* the listed routers R1...Rn did not need to examine this header; they processed the packet based only on its current destination address.

This form of routing header was deprecated by [RFC 5095](#), due to concerns about a traffic-amplification attack. An attacker could send off a packet with a routing header containing an alternating list of just two routers $\langle R1, R2, R1, R2, \dots, R1, R2, D \rangle$; this would generate substantial traffic on the R1–R2 link. [RFC 6275](#) and [RFC 6554](#) define more limited routing headers. [RFC 6275](#) defines a quite limited routing header to be used for IPv6 mobility (and also defines the IPv6 Mobility header). The [RFC 6554](#) routing header used for RPL, mentioned above, has the same basic form as the Type 0 header described above, but its use is limited to specific low-power routing domains.

11.5.4 IPv6 Fragment Header

IPv6 supports limited IPv4-style fragmentation via the Fragment Header. This header contains a 13-bit Fragment Offset field, which contains – as in IPv4 – the 13 high-order bits of the actual 16-bit offset of the fragment. This header also contains a 32-bit Identification field; all fragments of the same packet must carry the same value in this field.

IPv6 fragmentation is done *only* by the original sender; routers along the way are not allowed to fragment or re-fragment a packet. Sender fragmentation would occur if, for example, the sender had an 8 kB IPv6 packet to send via UDP, and needed to fragment it to accommodate the 1500-byte Ethernet MTU.

If a packet needs to be fragmented, the sender first identifies the **unfragmentable part**, consisting of the IPv6 fixed header and any extension headers that must accompany each fragment (these would include Hop-by-Hop and Routing headers). These unfragmentable headers are then attached to each fragment.

IPv6 also requires that every link on the Internet have an MTU of at least 1280 bytes beyond the LAN header; link-layer fragmentation and reassembly can be used to meet this MTU requirement (which is what ATM links ([5.5 Asynchronous Transfer Mode: ATM](#)) carrying IP traffic do).

Generally speaking, fragmentation should be avoided at the application layer when possible. UDP-based applications that attempt to transmit filesystem-sized (usually 8 kB) blocks of data remain persistent users of fragmentation.

11.5.5 General Extension-Header Issues

In the IPv4 world, many middleboxes ([9.7.2 Middleboxes](#)) examine not just the destination address but also the TCP port numbers; firewalls, for example, do this routinely to block all traffic except to a designated list of ports. In the IPv6 world, a middlebox may have difficulty *finding* the TCP header, as it must traverse a possibly lengthy list of extension headers. Worse, some of these extension headers may be newer than the middlebox, and thus unrecognized. Some middleboxes would simply drop packets with unrecognized extension headers, making the introduction of new such headers problematic.

[RFC 6564](#) addresses this by requiring that all future extension headers use a common “type-length-value” format: the first byte indicates the extension-header’s type and the second byte indicates its length. This facilitates rapid traversal of the extension-header chain. A few older extension headers – for example the Encapsulating Security Payload header of [RFC 4303](#) – do not follow this rule; middleboxes must treat these as special cases.

[RFC 2460](#) states

With one exception [that is, Hop-by-Hop headers], extension headers are not examined or processed by any node along a packet's delivery path, until the packet reaches the node (or each of the set of nodes, in the case of multicast) identified in the Destination Address field of the IPv6 header.

Nonetheless, sometimes intermediate nodes do attempt to add extension headers. This can break Path MTU Discovery ([18.6 Path MTU Discovery](#)), as the sender no longer controls the total packet size.

[RFC 7045](#) attempts to promulgate some general rules for the real-world handling of extension headers. For example, it states that, while routers are allowed to drop packets with certain extension headers, they may not do this simply because those headers are unrecognized. Also, routers *may* ignore Hop-by-Hop Option headers, or else process packets with such headers via a slower queue.

11.6 Neighbor Discovery

IPv6 Neighbor Discovery, or **ND**, is a set of related protocols that replaces several IPv4 tools, most notably ARP, ICMP redirects and most non-address-assignment parts of DHCP. The messages exchanged in ND are part of the ICMPv6 framework, [12.2 ICMPv6](#). The original specification for ND is in [RFC 2461](#), later updated by [RFC 4861](#). ND provides the following services:

- Finding the local router(s) [[11.6.1 Router Discovery](#)]
- Finding the set of network address prefixes that can be reached via local delivery (IPv6 allows there to be more than one) [[11.6.2 Prefix Discovery](#)]
- Finding a local host's LAN address, given its IPv6 address [[11.6.3 Neighbor Solicitation](#)]
- Detecting duplicate IPv6 addresses [[11.7.1 Duplicate Address Detection](#)]
- Determining that some neighbors are now unreachable

11.6.1 Router Discovery

IPv6 routers periodically send **Router Advertisement** (RA) packets to the all-nodes multicast group. Ordinary hosts wanting to know what router to use can wait for one of these periodic multicasts, or can request an RA packet immediately by sending a **Router Solicitation** request to the all-routers multicast group. Router Advertisement packets serve to identify the routers; this process is sometimes called **Router Discovery**. In IPv4, by comparison, the address of the default router is usually piggybacked onto the DHCP response message ([10.3 Dynamic Host Configuration Protocol \(DHCP\)](#)).

These RA packets, in addition to identifying the routers, also contain a list of all network address prefixes in use on the LAN. This is "prefix discovery", described in the following section. To a first approximation on a simple network, prefix discovery supplies the

network portion of the IPv6 address; on IPv4 networks, DHCP usually supplies the entire IPv4 address.

RA packets may contain other important information about the LAN as well, such as an agreed-on MTU.

These IPv6 router messages represent a change from IPv4, in which routers need not send anything besides forwarded packets. To become an IPv4 router, a node need only have IPv4 forwarding enabled in its kernel; it is then up to DHCP (or the equivalent) to inform neighboring nodes of the router. IPv6 puts the responsibility for this notification on the router itself: for a node to become an IPv6 router, in addition to forwarding packets, it “MUST” ([RFC 4294](#)) also run software to support Router Advertisement. Despite this mandate, however, the RA mechanism does not play a role in the forwarding process itself; an IPv6 network can run without Router Advertisements if every node is, for example, manually configured to know where the routers are and to know which neighbors are on-link. (We emphasize that manual configuration like this scales very poorly.)

On Linux systems, the Router Advertisement agent is most often the [radvd](#) daemon. See [12.6 IPv6 Connectivity via Tunneling](#) below.

11.6.2 Prefix Discovery

Closely related to Router Discovery is the **Prefix Discovery** process by which hosts learn what IPv6 network-address prefixes, above, are valid on the network. It is also where hosts learn which prefixes are considered to be local to the host’s LAN, and thus reachable at the LAN layer instead of requiring router assistance for delivery. IPv6, in other words, does *not* limit determination of whether delivery is local to the IPv4 mechanism of having a node check a destination address against each of the network-address prefixes assigned to the node’s interfaces.

Even IPv4 allows two IPv4 network prefixes to share the same LAN (*eg* a private one 10.1.2.0/24 and a public one 147.126.65.0/24), but a consequence of IPv4 routing is that two such LAN-sharing subnets can only reach one another via a router on the LAN, even though they should in principle be able to communicate directly. IPv6 drops this restriction.

The Router Advertisement packets sent by the router should contain a complete list of valid network-address prefixes, as the **Prefix Information** option. In simple cases this list may contain a single globally routable 64-bit prefix corresponding to the LAN subnet. If a particular LAN is part of multiple (overlapping) physical subnets, the prefix list will contain an entry for each subnet; these 64-bit prefixes will themselves likely share a common site-wide prefix of length $N < 64$. For multihomed sites the prefix list may contain multiple unrelated prefixes corresponding to the different address blocks. Finally, site-specific “unique local” IPv6 address prefixes may also be included.

Each prefix will have an associated **lifetime**; nodes receiving a prefix from an RA packet are to use it only for the duration of this lifetime. On expiration (and likely much sooner) a node must obtain a newer RA packet with a newer prefix list. The rationale for inclusion of the prefix lifetime is ultimately to allow sites to easily **renumber**; that is, to change providers and switch to a new network–address prefix provided by a new router. Each prefix is also tagged with a bit indicating whether it can be used for autoconfiguration, as in [11.7.2 Stateless Autoconfiguration \(SLAAC\)](#) below.

Each prefix also comes with a flag indicating whether the prefix is **on-link**. If set, then every node receiving that prefix is supposed to be on the same LAN. Nodes assume that to reach a neighbor sharing the same on-link address prefix, Neighbor Solicitation is to be used to find the neighbor’s LAN address. If a neighbor shares an off-link prefix, a router must be used. The IPv4 equivalent of two nodes sharing the same on-link prefix is sharing the same subnet prefix. For an example of subnets with prefix-discovery information, see [12.3 IPv6 Subnets](#).

Routers advertise off-link prefixes only in special cases; this would mean that a node is part of a subnet but cannot reach other members of the subnet directly. This may apply in some wireless settings, *eg* MANETs ([4.2.8 MANETs](#)) where some nodes on the same subnet are out of range of one another. It may also apply when using IPv6 Mobility ([9.9 Mobile IP, RFC 3775](#)).

11.6.3 Neighbor Solicitation

Neighbor Solicitation messages are the IPv6 analogues of IPv4 ARP requests. These are essentially queries of the form “who has IPv6 address X?” While ARP requests were broadcast, IPv6 Neighbor Solicitation messages are sent to the **solicited-node multicast address**, which at the LAN layer usually represents a rather small multicast group. This address is ff02::0001:255.y.z.w, where y, z and w are the low-order three bytes of the IPv6 address the sender is trying to look up (note that we are using the notation here for an embedded IPv4 address, even though y, z and w are from an IPv6 address). Each IPv6 host on the LAN will need to subscribe to all the solicited-node multicast addresses corresponding to its own IPv6 addresses (normally this is not too many).

Neighbor Solicitation messages are repeated regularly, but followup verifications are initially sent to the unicast LAN address on file (this is common practice with ARP implementations, but is optional). Unlike with ARP, other hosts on the LAN are not expected to eavesdrop on the initial Neighbor Solicitation message. The target host’s response to a Neighbor Solicitation message is called **Neighbor Advertisement**; a host may also send these unsolicited if it believes its LAN address may have changed.

The analogue of Proxy ARP is still permitted, in that a node may send Neighbor Advertisements on behalf of another. The most likely reason for this is that the node receiving proxy services is a “mobile” host temporarily remote from the home LAN.

Neighbor Advertisements sent as proxies have a flag to indicate that, if the real target does speak up, the proxy advertisement should be ignored.

Once a node (host or router) has discovered a neighbor's LAN address through Neighbor Solicitation, it continues to monitor the neighbor's continued reachability.

Neighbor Solicitation also includes Neighbor Unreachability Detection. Each node (host or router) continues to monitor its known neighbors; reachability can be inferred either from ongoing IPv6 traffic exchanges or from Neighbor Advertisement responses. If a node detects that a neighboring host has become unreachable, the original node may retry the multicast Neighbor Solicitation process, in case the neighbor's LAN address has simply changed. If a node detects that a neighboring *router* has become unreachable, it attempts to find an alternative path.

Finally, IPv4 ICMP Redirect messages have also been moved in IPv6 to the Neighbor Discovery protocol. These allow a router to tell a host that another router is better positioned to handle traffic to a given destination.

11.6.4 Security and Neighbor Discovery

In the protocols outlined above, received ND messages are trusted; this can lead to problems with nodes pretending to be things they are not. Here are two examples:

- A host can pretend to be a router simply by sending out Router Advertisements; such a host can thus capture traffic from its neighbors, and even send it on – perhaps selectively – to the real router.
- A host can pretend to be another host, in the IPv6 analog of ARP spoofing ([10.2.2 ARP Security](#)). If host A sends out a Neighbor Solicitation for host B, nothing prevents host C from sending out a Neighbor Advertisement claiming to be B (after previously joining the appropriate multicast group).

These two attacks can have the goal either of eavesdropping or of denial of service; there are also purely denial-of-service attacks. For example, host C can answer host B's DAD queries (below at [11.7.1 Duplicate Address Detection](#)) by claiming that the IPv6 address in question is indeed in use, preventing B from ever acquiring an IPv6 address. A good summary of these and other attacks can be found in [RFC 3756](#).

These attacks, it is worth noting, can only be launched by nodes on the same LAN; they cannot be launched remotely. While this reduces the risk, though, it does not eliminate it. Sites that allow anyone to connect, such as Internet cafés, run the highest risk, but even in a setting in which all workstations are “locked down”, a node compromised by a virus may be able to disrupt the network.

[RFC 4861](#) suggested that, at sites concerned about these kinds of attacks, hosts might use the IPv6 Authentication Header or the Encapsulated Security Payload Header to supply digital signatures for ND packets (see [29.6 IPsec](#)). If a node is configured to

require such checks, then most ND-based attacks can be prevented. Unfortunately, [RFC 4861](#) offered no suggestions beyond static configuration, which scales poorly and also rather completely undermines the goal of autoconfiguration.

A more flexible alternative is Secure Neighbor Discovery, or **SEND**, specified in [RFC 3971](#). This uses public-key encryption ([29 Public-Key Encryption](#)) to validate ND messages; for the remainder of this section, some familiarity with the material at [29 Public-Key Encryption](#) may be necessary. Each message is digitally signed by the sender, using the sender's private key; the recipient can validate the message using the sender's corresponding public key. In principle this makes it impossible for one message sender to pretend to be another sender.

In practice, the problem is that public keys by themselves guarantee (if not compromised) only that the sender of a message is the same entity that previously sent messages using that key. In the second bulleted example above, in which C sends an ND message falsely claiming to be B, straightforward applications of public keys would prevent this *if* the original host A had previously heard from B, and trusted that sender to be the real B. But in general A would not know which of B or C was the real B. A cannot trust whichever host it heard from first, as it is indeed possible that C started its deception with A's very first query for B, beating B to the punch.

A common solution to this identity-guarantee problem is to create some form of "public-key infrastructure" such as **certificate authorities**, as in [29.5.2.1 Certificate Authorities](#). In this setting, every node is configured to trust messages signed by the certificate authority; that authority is then configured to vouch for the identities of other nodes whenever this is necessary for secure operation. SEND implements its own version of certificate authorities; these are known as **trust anchors**. These would be configured to guarantee the identities of all routers, and perhaps hosts. The details are somewhat simpler than the mechanism outlined in [29.5.2.1 Certificate Authorities](#), as the anchors and routers are under common authority. When trust anchors are used, each host needs to be configured with a list of their addresses.

SEND also supports a simpler public-key validation mechanism known as **cryptographically generated addresses**, or CGAs ([RFC 3972](#)). These are IPv6 interface identifiers that are secure hashes ([28.6 Secure Hashes](#)) of the host's public key (and a few other non-secret parameters). CGAs are an alternative to the interface-identifier mechanisms discussed in [11.2.1 Interface identifiers](#). DNS names in the .onion domain used by TOR also use CGAs.

The use of CGAs makes it impossible for host C to successfully claim to be host B: only B will have the public key that hashes to B's address *and* the matching private key. If C attempts to send to A a neighbor advertisement claiming to be B, then C can sign the message with its own private key, but the hash of the corresponding public key will not match the interface-identifier portion of B's address. Similarly, in the DAD scenario, if C

attempts to tell B that B's newly selected CGA address is already in use, then again C won't have a key matching that address, and B will ignore the report.

In general, CGI addresses allow recipients of a message to verify that the source address is the "owner" of the associated public key, without any need for a public-key infrastructure (29.3 [Trust and the Man in the Middle](#)). C *can* still pretend to be a router, using its own CGA address, because router addresses are not known by the requester beforehand. However, it is easier to protect routers using trust anchors as there are fewer of them.

SEND relies on the fact that finding two inputs hashing to the same 64-bit CGA is infeasible, as in general this would take about 2^{64} tries. An IPv4 analog would be impossible as the address host portion won't have enough bits to prevent finding hash collisions via brute force. For example, if the host portion of the address has ten bits, it would take C about 2^{10} tries (by tweaking the supplemental hash parameters) until it found a match for B's CGA.

SEND has seen very little use in the IPv6 world, partly because IPv6 itself has seen such slow adoption, but also because of the perception that the vulnerabilities SEND protects against are difficult to exploit.

RA-guard is a simpler mechanism to achieve ND security, but one that requires considerable support from the LAN layer. Outlined in [RFC 6105](#), it requires that each host connects directly to a switch; that is, there must be no shared-media Ethernet. The switches must also be fairly smart; it must be possible to configure them to know which ports connect to routers rather than hosts, and, in addition, it must be possible to configure them to block Router Advertisements from host ports that are *not* router ports. This is quite effective at preventing a host from pretending to be a router, and, while it assumes that the switches can do a significant amount of packet inspection, that is in fact a fairly common Ethernet switch feature. If Wi-Fi is involved, it does require that access points (which are a kind of switch) be able to block Router Advertisements; this isn't quite as commonly available. In determining which switch ports are connected to routers, [RFC 6105](#) suggests that there might be a brief initial learning period, during which all switch ports connecting to a device that *claims* to be a router are considered, permanently, to be router ports.

11.7 IPv6 Host Address Assignment

IPv6 provides two competing ways for hosts to obtain their full IP addresses. One is **DHCPv6**, based on IPv4's DHCP (10.3 [Dynamic Host Configuration Protocol \(DHCP\)](#)), in which the entire address is handed out by a DHCPv6 server. The other is **StateLess Address AutoConfiguration**, or SLAAC, in which the interface-identifier part of the address is generated locally, and the network prefix is obtained via prefix discovery. The original idea behind SLAAC was to support complete plug-and-play network setup: hosts on an isolated LAN could talk to one another out of the box, and if a router was

introduced connecting the LAN to the Internet, then hosts would be able to determine unique, routable addresses from information available from the router.

In the early days of IPv6 development, in fact, DHCPv6 may have been intended only for address assignments to routers and servers, with SLAAC meant for “ordinary” hosts. In that era, it was still common for IPv4 addresses to be assigned “statically”, via per-host configuration files. [RFC 4862](#) states that SLAAC is to be used when “a site is not particularly concerned with the exact addresses hosts use, so long as they are unique and properly routable.”

SLAAC and DHCPv6 evolved to some degree in parallel. While SLAAC solves the autoconfiguration problem quite neatly, at this point DHCPv6 solves it just as effectively, and provides for greater administrative control. For this reason, SLAAC may end up less widely deployed. On the other hand, SLAAC gives hosts greater control over their IPv6 addresses, and so may end up offering hosts a greater degree of privacy by allowing endpoint management of the use of private and temporary addresses (below).

When a host first begins the Neighbor Discovery process, it receives a Router Advertisement packet. In this packet are two special bits: the M (managed) bit and the O (other configuration) bit. The M bit is set to indicate that DHCPv6 is available on the network for address assignment. The O bit is set to indicate that DHCPv6 is able to provide additional configuration information (*eg* the name of the DNS server) to hosts that are using SLAAC to obtain their addresses. In addition, each individual prefix in the RA packet has an A bit, which when set indicates that the associated prefix may be used with SLAAC.

11.7.1 Duplicate Address Detection

Whenever an IPv6 host obtains a unicast address – a link-local address, an address created via SLAAC, an address received via DHCPv6 or a manually configured address – it goes through a **duplicate-address detection** (DAD) process. The host sends one or more Neighbor Solicitation messages (that is, like an ARP query), as in [11.6 Neighbor Discovery](#), asking if any other host has this address. If anyone answers, then the address is a duplicate. As with IPv4 ACD ([10.2.1 ARP Finer Points](#)), but *not* as with the original IPv4 self-ARP, the source-IP-address field of this NS message is set to a special “unspecified” value; this allows other hosts to recognize it as a DAD query.

Because this NS process may take some time, and because addresses are in fact almost always unique, [RFC 4429](#) defines an **optimistic DAD** mechanism. This allows limited use of an address before the DAD process completes; in the meantime, the address is marked as “optimistic”.

Outside the optimistic-DAD interval, a host is not allowed to use an IPv6 address if the DAD process has failed. [RFC 4862](#) in fact goes further: if a host with an established

address receives a DAD query for that address, indicating that some other host wants to use that address, then the original host should discontinue use of the address.

If the DAD process fails for an address based on an EUI-64 identifier, then some other node has the same Ethernet address and you have bigger problems than just finding a working IPv6 address. If the DAD process fails for an address constructed with the [RFC 7217](#) mechanism, [11.2.1 Interface identifiers](#), the host is able to generate a new interface identifier and try again. A counter for the number of DAD attempts is included in the hash that calculates the interface identifier; incrementing this counter results in an entirely new identifier.

While DAD works quite well on Ethernet-like networks with true LAN-layer multicast, it may be inefficient on, say, MANETs ([4.2.8 MANETs](#)), as distant hosts may receive the DAD Neighbor Solicitation message only after some delay, or even not at all. Work continues on the development of improvements to DAD for such networks.

11.7.2 Stateless Autoconfiguration (SLAAC)

To obtain an address via SLAAC, defined in [RFC 4862](#), the first step for a host is to generate its link-local address (above, [11.2.2 Link-local addresses](#)), appending the standard 64-bit link-local prefix fe80::/64 to its interface identifier ([11.2.1 Interface identifiers](#)). The latter is likely derived from the host's LAN address using either EUI-64 or the [RFC 7217](#) mechanism; the important point is that it is available without network involvement.

The host must then ensure that its newly configured link-local address is in fact unique; it uses DAD (above) to verify this. Assuming no duplicate is found, then at this point the host can talk to any other hosts on the same LAN, *eg* to figure out where the printers are.

The next step is to see if there is a router available. The host may send a Router Solicitation (RS) message to the all-routers multicast address. A router – if present – should answer with a Router Advertisement (RA) message that also contains a Prefix Information option; that is, a list of IPv6 network-address prefixes ([11.6.2 Prefix Discovery](#)).

As mentioned earlier, the RA message will mark with a flag those prefixes eligible for use with SLAAC; if no prefixes are so marked, then SLAAC should not be used. All prefixes will also be marked with a lifetime, indicating how long the host may continue to use the prefix. Once the prefix expires, the host must obtain a new one via a new RA message.

The host chooses an appropriate prefix, stores the prefix-lifetime information, and appends the prefix to the front of its interface identifier to create what should now be a routable address. The address so formed must now be verified through the DAD mechanism above.

In the era of EUI-64 interface identifiers, it would in principle have been possible for the receiver of a packet to extract the sender's LAN address from the interface-identifier portion of the sender's SLAAC-generated IPv6 address. This in turn would allow bypassing the Neighbor Solicitation process to look up the sender's LAN address. This was never actually permitted, however, even before the privacy options below, as there is no way to be certain that a received address was in fact generated via SLAAC. With [RFC 7217](#)-based interface identifiers, LAN-address extraction is no longer even potentially an option.

A host using SLAAC may receive multiple network prefixes, and thus generate for itself at least one address per prefix. [RFC 6724](#) defines a process for a host to determine, when it wishes to connect to destination address D, which of its own multiple addresses to use. For example, if D is a unique-local address, not globally visible, then the host will likely want to choose a source address that is also unique-local. [RFC 6724](#) also includes mechanisms to allow a host with a permanent public address (possibly corresponding to a DNS entry, but just as possibly formed directly from an interface identifier) to prefer alternative "temporary" or "privacy" addresses for outbound connections; see, for example, [11.7.2.1 SLAAC privacy](#). Finally, [RFC 6724](#) also defines the sorting order for multiple addresses representing the same destination; see [12.4 Using IPv6 and IPv4 Together](#).

At the end of the SLAAC process, the host knows its IPv6 address (or set of addresses) and its default router. In IPv4, these would have been learned through DHCP along with the identity of the host's DNS server; one concern with SLAAC is that it originally did not provide a way for a host to find its DNS server. One strategy is to fall back on DHCPv6 for this. However, [RFC 6106](#) now defines a process by which IPv6 routers can include DNS-server information in the RA packets they send to hosts as part of the SLAAC process; this completes the final step of autoconfiguration.

How to get DNS names for SLAAC-configured IPv6 hosts into the DNS servers is an entirely separate issue. One approach is simply not to give DNS names to such hosts. In the NAT-router model for IPv4 autoconfiguration, hosts on the inward side of the NAT router similarly do not have DNS names (although they are also not reachable directly, while SLAAC IPv6 hosts would be reachable). If DNS names are needed for hosts, then a site might choose DHCPv6 for address assignment instead of SLAAC. It is also possible to figure out the addresses SLAAC would use (by identifying the host-identifier bits) and then creating DNS entries for these hosts. Finally, hosts can also use **Dynamic DNS** ([RFC 2136](#)) to update their own DNS records.

11.7.2.1 SLAAC privacy

A portable host that always uses SLAAC as it moves from network to network and always bases its SLAAC addresses on the EUI-64 interface identifier (or on any other static interface identifier) will be easy to track: its interface identifier will never change. This is

one reason why the obfuscation mechanism of [RFC 7217](#) interface identifiers ([11.2.1 Interface identifiers](#)) includes the network *prefix* in the hash: connecting to a new network will then result in a new interface identifier.

Well before [RFC 7217](#), however, [RFC 4941](#) introduced a set of **privacy extensions** to SLAAC: optional mechanisms for the generation of alternative interface identifiers, based as with RFC 7217 on pseudorandom generation using the original LAN-address-based interface identifier as a “seed” value.

RFC 4941 goes further, however, in that it supports **regular changes** to the interface identifier, to increase the difficulty of tracking a host over time even if it does not change its network prefix. One first selects a 128-bit secure-hash function $F()$, *eg* MD5 ([28.6 Secure Hashes](#)). New temporary interface IDs (IIDs) can then be calculated as follows

$$(IID_{new}, seed_{new}) = F(seed_{old}, IID_{old})$$

where the left-hand pair represents the two 64-bit halves of the 128-bit return value of $F()$ and the arguments to $F()$ are concatenated together. (The seventh bit of IID_{new} must also be set to 0; *cf* [11.2.1 Interface identifiers](#) where this bit is set to 1.) This process is privacy-safe even if the initial IID is based on EUI-64.

The probability of two hosts accidentally choosing the same interface identifier in this manner is vanishingly small; the Neighbor Solicitation mechanism with DAD must, however, still be used to verify that the address is in fact unique within the host’s LAN.

The privacy addresses above are to be used only for connections initiated by the client; to the extent that the host accepts incoming connections and so needs a “fixed” IPv6 address, the address based on the original EUI-64/RFC-7217 interface identifier should still be available. As a result, the RFC 7217 mechanism is still important for privacy even if the RFC 4941 mechanism is fully operational.

RFC 4941 stated that privacy addresses were to be disabled by default, largely because of concerns about frequently changing IP addresses. These concerns have abated with experience and so privacy addresses are often now automatically enabled. Typical address lifetimes range from a few hours to 24 hours. Once an address has “expired” it generally remains available but *deprecated* for a few temporary-address cycles longer.

A consequence of privacy addresses (for either SLAAC or DHCPv6) is that one host will typically have **multiple active addresses** for any one network prefix, at any given time. [RFC 7934](#) suggests that a host might change its address, for privacy reasons, once per day, and that each address would have a lifetime of seven days. Add to that the use of separate addresses for virtual machines, and perhaps also for containerized applications, and [RFC 7934](#) suggests that up to 20 addresses might be needed. The number might be quite a bit higher; some proposals for privacy addresses suggest changing them much more often than once a day (though the address lifetimes might

also be reduced). It would not be entirely unreasonable, in fact, for a browser to use a separate IPv6 address for each separate website accessed. The use of too many addresses does add to the memory and traffic requirements of router Neighbor Discovery ([11.6 Neighbor Discovery](#)), however.

DHCPv6 also provides an option for temporary address assignments, again to improve privacy, but one of the potential advantages of SLAAC is that this process is entirely under the control of the end system.

Regularly (*eg* every few hours, or less) changing the host portion of an IPv6 address should make external tracking of a host more difficult, at least if tracking via web-browser cookies is also somehow prevented. However, for a residential “site” with only a handful of hosts, a considerable degree of tracking may be obtained simply by observing the common 64-bit prefix.

For a general discussion of privacy issues related to IPv6 addressing, see [RFC 7721](#).

11.7.3 DHCPv6

The job of a DHCPv6 server is to tell an inquiring host its network prefix(es) and also supply a 64-bit host-identifier, very similar to an IPv4 DHCPv4 server. Hosts begin the process by sending a DHCPv6 request to the All_DHCP_Relay_Agents_and_Servers multicast IPv6 address ff02::1:2 (versus the broadcast address for IPv4). As with DHCPv4, the job of a relay agent is to tag a DHCPv6 request with the correct current subnet, and then to forward it to the actual DHCPv6 server. This allows the DHCPv6 server to be on a different subnet from the requester. Note that the use of multicast does nothing to diminish the need for relay agents. In fact, the All_DHCP_Relay_Agents_and_Servers multicast address scope is limited to the current LAN; relay agents then forward to the actual DHCPv6 server using the *site*-scoped address All_DHCP_Servers.

Hosts using SLAAC to obtain their address can still use a special Information-Request form of DHCPv6 to obtain their DNS server and any other “static” DHCPv6 information.

Clients may ask for **temporary** addresses. These are identified as such in the “Identity Association” field of the DHCPv6 request. They are handled much like “permanent” address requests, except that the client may ask for a new temporary address only a short time later. When the client does so, a *different* temporary address will be returned; a repeated request for a permanent address, on the other hand, would usually return the same address as before. Temporary addresses are typically used to improve privacy, by making it more difficult to track users by IPv6 address.

When the DHCPv6 server returns a temporary address, it may of course keep a log of this address. When SLAAC is used, a log is still possible, as each new address must run through the Neighbor Discovery ([11.6 Neighbor Discovery](#)) process. However, SLAAC does place control of the cryptographic mechanisms for temporary-address creation in the hands of the end user, rather than in a centralized service. For example, the DHCPv6

temporary-address mechanism might have a flaw that would allow a remote observer to infer a relationship between different temporary addresses, though the secure-hash mechanism described below appears to be secure as long as the `secret_key` portion is not compromised.

A DHCPv6 response contains a list (perhaps of length 1) of IPv6 addresses. Each separate address has an expiration date. The client must send a new request before the expiration of any address it is actually using.

In DHCPv4, the host portion of addresses typically comes from “address pools” representing small ranges of integers such as 64–254; these values are generally allocated consecutively. A DHCPv6 server, on the other hand, should take advantage of the enormous range (2^{64}) of possible host portions by allocating values more sparsely, through the use of pseudorandomness. This is in part to make it very difficult for an outsider who knows one of a site’s host addresses to guess the addresses of other hosts, cf [11.2.1 Interface identifiers](#).

The Internet Draft [draft-ietf-dhc-stable-privacy-addresses](#) proposes the following mechanism by which a DHCPv6 server may generate the interface-identifier bits for the addresses it hands out; $F()$ is a secure-hash function and its arguments are concatenated together:

$$F(\text{prefix}, \text{client_DUID}, \text{IAID}, \text{DAD_counter}, \text{secret_key})$$

The `prefix`, `DAD_counter` and `secret_key` arguments are as in [11.7.2.1 SLAAC privacy](#). The `client_DUID` is the string by which the client identifies itself to the DHCPv6 server; it may be based on the Ethernet address though other options are possible. The `IAID`, or Identity Association identifier, is a client-provided name for this request; different names are used when requesting temporary versus permanent addresses.

Some older DHCPv6 servers may still allocate interface identifiers in serial order; such obsolete servers might make the SLAAC approach more attractive.

11.8 Epilog

IPv4 has run out of large address blocks, as of 2011. IPv6 has reached a mature level of development. Most common operating systems provide excellent IPv6 support.

Yet conversion has been slow. Many ISPs still provide limited (to nonexistent) support, and inexpensive IPv6 firewalls to replace the ubiquitous consumer-grade NAT routers are just beginning to appear. Time will tell how all this evolves. However, while IPv6 has now been around for twenty years, top-level IPv4 address blocks disappeared much more recently. It is quite possible that this will prove to be just the catalyst IPv6 needs.

11.9 Exercises

Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises.

1.0. Each IPv6 address is associated with a specific solicited-node multicast address.

(a). Explain why, on a typical Ethernet, if the original IPv6 host address was obtained via SLAAC then the LAN multicast group corresponding to the host's solicited-node multicast addresses is likely to be small, in many cases consisting of one host only. (Packet delivery to small LAN multicast groups can be much more efficient than delivery to large multicast groups.)

(b). What steps might a DHCPv6 server take to ensure that, for the IPv6 addresses it hands out, the LAN multicast groups corresponding to the host addresses' solicited-node multicast addresses will be small?

2.0. If an attacker sends a large number of probe packets via IPv4, you can block them by blocking the attacker's IP address. Now suppose the attacker uses IPv6 to launch the probes; for each probe, the attacker changes the low-order 64 bits of the address. Can these probes be blocked efficiently? If so, what do you have to block? Might you also be blocking other users?

3.0. Which of the following IPv6 addresses are part of the 2401:3c00::/22 block?

- (a). 2401:3cfe:2357:f00d::1
- (b). 2401:4c02:1248:beef::1
- (c). 2401:3b0e:0139:cafe::1
- (d). 2410:3d04:1247:face::1

4.0. An IPv6 fixed-header is 40 bytes. Taking this as the minimum packet size, how long will it take to send 10^{15} hosts (one quadrillion) probe packets to a target site (eg ping, [12.5.1 ping6](#), but not necessarily that), if the bandwidth is 1 Gbps?

12 IPv6 Additional Features

In this chapter we continue describing IPv6, with special attention paid to comparisons between IPv4 and IPv6, interoperability between IPv4 and IPv6, and operating IPv6 in an environment where it is not natively supported.

12.1 Globally Exposed Addresses

Perhaps the most striking difference between a contemporary IPv4 network and an IPv6 network is that on the former, most end-user workstation addresses are likely to be “hidden” behind a NAT router ([9.7 Network Address Translation](#)). On an IPv6 network, on the other hand, *every* host address is globally visible, though inbound access may be limited by firewalls. While IPv4 NAT is sometimes claimed to provide better security, its real benefit is that it enables sites to cope with the limited number of IPv4 addresses

available. (IPv6-only networks do often use a form of NAT to allow connectivity to IPv4-only servers.)

In addition to limiting the number of IPv4 addresses needed, legacy IPv4 NAT routers provide a measure of each of privacy, security and nuisance. Privacy in IPv6 can be handled, as in the previous chapter, through private or temporary addresses. Recall that in the IPv6 world such addresses are still globally visible; privacy here means that the address is changed regularly.

The degree of security provided via NAT is mostly due to the fact that all connections must be initiated from the inside; no packet from the outside is allowed through the NAT firewall unless it is a response to a packet sent from the inside. This feature, however, can also be implemented via a conventional firewall (IPv4 or IPv6), without address translation. Furthermore, given such a conventional firewall, it is then straightforward to modify it so as to support limited and regulated connections from the outside world as desired; an analogous modification of a NAT router is more difficult. (That said, a blanket ban on IPv6 connections initiated from the outside can prove as frustrating as IPv4 NAT.)

A second security benefit for hiding IPv4 addresses is that with IPv4 it is easy to map a /24 subnet by pinging or otherwise probing each of the 254 possible hosts; such mapping may reveal internal structure. In IPv6 such mapping is meant to be impractical as a /64 subnet has $2^{64} \approx 18$ quintillion hosts (though see the randomness note in [11.2.1 Interface identifiers](#)). If the low-order 64 bits of a host's IPv6 address are chosen with sufficient randomness, finding the host by probing is virtually impossible; see exercise 4.0 of the previous chapter.

As for nuisance, NAT has always broken protocols that involve negotiation of new connections (*eg* TFTP, FTP, or SIP, used by VoIP); IPv6 has the potential to make these much easier to manage. That said, a strict firewall rule blocking all inbound connections would eliminate this potential benefit.

12.2 ICMPv6

[RFC 4443](#) defines an updated version of the ICMP protocol for IPv6. As with the IPv4 version, messages are identified by 8-bit type and code (subtype) fields, making it reasonably easy to add new message formats. We have already seen the ICMP messages that make up Neighbor Discovery ([11.6 Neighbor Discovery](#)).

Unlike ICMPv4, ICMPv6 distinguishes between informational and error messages by the first bit of the type field. Unknown informational messages are simply dropped, while unknown error messages must be handed off, if possible, to the appropriate upper-layer process. For example, “[UDP] port unreachable” messages are to be delivered to the UDP sender of the undeliverable packet.

ICMPv6 includes an IPv6 version of Echo Request / Echo Reply, upon which the “ping6” command (12.5.1 ping6) is based; unlike with IPv4, arriving IPv6 echo-reply messages are delivered to the process that generated the corresponding echo request. The base ICMPv6 specification also includes formats for the error conditions below; this list is somewhat cleaner than the corresponding ICMPv4 list:

Destination Unreachable

In this case, one of the following numeric codes is returned:

1. **No route to destination**, returned when a router has no next_hop entry.
2. **Communication with destination administratively prohibited**, returned when a router *has* a next_hop entry, but declines to use it for policy reasons. Codes 5 and 6, below, are special cases of this situation; these more-specific codes are returned when appropriate.
3. **Beyond scope of source address**, returned when a router is, for example, asked to route a packet to a global address, but the return address is not, *eg* is unique-local. In IPv4, when a host with a private address attempts to connect to a global address, NAT is almost always involved.
4. **Address unreachable**, a catchall category for routing failure not covered by any other message. An example is if the packet was successfully routed to the last_hop router, but Neighbor Discovery failed to find a LAN address corresponding to the IPv6 address.
5. **Port unreachable**, returned when, as in ICMPv4, the destination host does not have the requested UDP port open.
6. **Source address failed ingress/egress policy**, see code 1.
7. **Reject route to destination**, see code 1.

Packet Too Big

This is like ICMPv4’s “Fragmentation Required but DontFragment flag set”; IPv6 however has no router-based fragmentation.

Time Exceeded

This is used for cases where the Hop Limit was exceeded, and also where *source*-based fragmentation was used and the fragment-reassembly timer expired.

Parameter Problem

This is used when there is a malformed entry in the IPv6 header, an unrecognized Next Header type, or an unrecognized IPv6 option.

_node information:

12.2.1 Node Information Messages

ICMPv6 also includes **Node Information (NI) Messages**, defined in [RFC 4620](#). One form of NI query allows a host to be asked directly for its name; this is accomplished in IPv4 via reverse-DNS lookups ([10.1.3 Other DNS Records](#)). Other NI queries allow a host to be asked for its other IPv6 addresses, or for its IPv4 addresses. Recipients of NI queries may be configured to refuse to answer.

12.3 IPv6 Subnets

In the IPv4 world, network managers sometimes struggle to divide up a limited address space into a pool of appropriately sized subnets. In IPv6, this is much simpler: all subnets are of size /64, following the guidelines set out in [11.3 Network Prefixes](#).

There is one common exception: [RFC 6164](#) permits the use of 127-bit prefixes at each end of a point-to-point link. The 128th bit is then 0 at one end and 1 at the other.

A site receiving from its provider an address prefix of size /56 can assign up to 256 /64 subnets. As with IPv4, the reasons for IPv6 subnetting are to join incompatible LANs, to press intervening routers into service as inter-subnet firewalls, or otherwise to separate traffic.

The diagram below shows a site with an external prefix of 2001::/62, two routers R1 and R2 with interfaces numbered as shown, and three internal LANS corresponding to three subnets 2001:0:0:1::/64, 2001:0:0:2::/64 and 2001:0:0:3::/64. The subnet 2001:0:0:0::/64 (2001::/64) is used to connect to the provider.

Interface 0 of R1 would be assigned an address from the /64 block 2001:0:0:0/64, perhaps 2001::2.

R1 will announce over its interface 1 – via router advertisements – that it has a *route* to ::/0, that is, it has the default route. It will also advertise via interface 1 the on-link prefix 2001:0:0:1::/64.

R2 will announce via interface 1 its routes to 2001:0:0:2::/64 and 2001:0:0:3::/64. It will also announce the default route on interfaces 2 and 3. On interface 2 it will advertise the on-link prefix 2001:0:0:2::/64, and on interface 3 the prefix 2001:0:0:3::/64. It could also, as a backup, advertise prefix 2001:0:0:1::/64 on its interface 1. On each subnet, only the subnet's on-link prefix is advertised.

12.3.1 Subnets and /64

Fixing the IPv6 division of prefix and host (interface) lengths at 64 bits for each is a compromise. While it does reduce the maximum number of subnets from 2^{128} to 2^{64} , in practice this is not a realistic concern, as 2^{64} is still an enormous number.

By leaving 64 bits for host identifiers, this 64/64 split leaves enough room for the privacy mechanisms of [11.7.2.1 SLAAC privacy](#) and [11.7.3 DHCPv6](#) to provide reasonable protection.

Much of the recent motivation for considering divisions other than 64/64 is grounded in concerns about ISP address-allocation policies. By declaring that users should each receive a /64 allocation, one hope is that users will in fact get enough for several subnets. Even a residential customer with only, say, two hosts and a router needs more than a single /64 address block, because the link from ISP to customer needs to be on its own subnet (it could use a 127-bit prefix, as above, but many customers would in fact have a need for multiple /64 subnets). By requiring /64 for a subnet, the hope is that users will all be allocated, for example, prefixes of at least /60 (16 subnets) or even /56 (256 subnets).

Even if that hope does not pan out, the 64/64 rule means that every user should *at least* get a /64 allocation.

On the other hand, if users *are* given only /64 blocks, and they want to use subnets, then they have to break the 64/64 rule locally. Perhaps they can create four subnets each with a prefix of length 66 bits, and each with only 62 bits for the host identifier. Wanting to do that in a standard way would dictate more flexibility in the prefix/host division.

But if the prefix/host division becomes completely arbitrary, there is nothing to stop ISPs from handing out prefixes with lengths of /80 (leaving 48 host bits) or even /120.

The general hope is that ISPs will not be so stingy with prefix lengths. But with IPv6 adoption still relatively modest, how this will all work out is not yet clear. In the IPv4 world, users use NAT ([9.7 Network Address Translation](#)) to create as many subnets as they desire. In the IPv6 world, NAT is generally considered to be a bad idea.

Finally, in theory it is possible to squeeze a site with two subnets onto a single /64 by converting the site's main router to a switch; all the customer's hosts now connect on an equal footing to the ISP. But this means making it much harder to use the router as a firewall, as described in [12.1 Globally Exposed Addresses](#). For most users, this is too risky.

12.4 Using IPv6 and IPv4 Together

In this section we will assume that IPv6 connectivity exists at a site; if it does not, see [12.6 IPv6 Connectivity via Tunneling](#).

If IPv6 coexists on a client machine with IPv4, in a so-called **dual-stack** configuration, which is used? If the client wants to connect using TCP to an IPv4-only website (or to some other network service), there is no choice. But what if the remote site also supports both IPv4 and IPv6?

The first step is the **DNS lookup**, triggered by the application's call to the appropriate address-lookup library procedure; in the Java stalk example of [16.1.3.3 The Client](#) we use `InetAddress.getByName()`. In the C language, address lookup is done with `getaddrinfo()` or (the now-deprecated) `gethostbyname()`. The DNS system on the client then contacts its DNS resolver and asks for the appropriate address record corresponding to the server name.

For IPv4 addresses, DNS maintains so-called “A” records, for “Address”. The IPv6 equivalent is the “AAAA” record, for “Address four times longer”. A dual-stack machine usually requests both. The Internet Draft [draft-vavrusa-dnsop-aaaa-for-free](#) proposes that, whenever a DNS server delivers an IPv4 A record, it also includes the corresponding AAAA record, much as IPv4 CNAME records are sent with piggybacked corresponding A records ([10.1.2 nslookup and dig](#)). The DNS requests are sent to the client's pre-configured DNS-resolver address (probably set via DHCP).

IPv6 and this book

This book is, as of April 2015, available via IPv6. Within the `cs.luc.edu` DNS zone are defined the following:

- [intronetworks](#): both A and AAAA records
- [intronetworks6](#): AAAA records only
- [intronetworks4](#): A records only

DNS itself can run over either IPv4 or IPv6. A DNS server (authoritative nameserver or just resolver) using only IPv4 can answer IPv6 AAAA-record queries, and a DNS server using only IPv6 can answer IPv4 A-record queries. Ideally each nameserver would eventually support both IPv4 and IPv6 for all queries, though it is common for hosts with newly enabled IPv6 connectivity to continue to use IPv4-only resolvers. See [RFC 4472](#) for a discussion of some operational issues.

Here is an example of DNS requests for A and AAAA records made with the `nslookup` utility from the command line. (In this example, the DNS resolver was contacted using IPv4.)

```
nslookup -query=A facebook.com
Name: facebook.com
Address: 173.252.120.6
nslookup -query=AAAA facebook.com
facebook.com has AAAA address 2a03:2880:2130:cf05:face:b00c:0:1
```

A few sites have IPv6-only DNS names. If the DNS query returns only an AAAA record, IPv6 must be used. One example in 2015 is [ipv6.google.com](#). In general, however, IPv6-only names such as this are recommended only for diagnostics and testing. The primary DNS names for IPv4/IPv6 sites should have both types of DNS records, as in the Facebook example above (and as for [google.com](#)).

Java `getByName()`

The Java `getByName()` call may *not* abide by system-wide [RFC 6742](#)-style preferences; the Java [Networking Properties documentation](#) (2015) states that “the default behavior is to prefer using IPv4 addresses over IPv6 ones”. This can be changed by setting the system property `java.net.preferIPv6Addresses` to `true`, using `System.setProperty()`.

If the client application uses a library call like Java’s `InetAddress.getByName()`, which returns a *single* IP address, the client will then attempt to connect to the address returned. If an IPv4 address is returned, the connection will use IPv4, and similarly with IPv6. If an IPv6 address is returned and IPv6 connectivity is not working, then the connection will fail.

For such an application, the DNS resolver library thus effectively makes the IPv4-or-IPv6 decision. [RFC 6724](#), which we encountered above in [11.7.2 Stateless Autoconfiguration \(SLAAC\)](#), provides a configuration mechanism, through a small table of IPv6 prefixes and **precedence** values such as the following.

prefix	precedence	
::1/128	50	IPv6 loopback
::/0	40	“default” match
2002::/16	30	6to4 address; see sidebar in 12.6 IPv6 Connectivity via Tunneling
::ffff:0:0/96	10	Matches embedded IPv4 addresses; see 11.3 Network Prefixes
fc00::/7	3	unique-local plus reserved; see 11.3 Network Prefixes

An address is assigned a precedence by looking it up in the table, using the longest-match rule ([14.1 Classless Internet Domain Routing: CIDR](#)); a list of addresses is then sorted in decreasing order of precedence. There is no entry above for link-local addresses, but by default they are ranked below global addresses. This can be changed by including the link-local prefix `fe80::/64` in the above table and ranking it higher than, say, `::/0`.

The default configuration is generally to prefer IPv6 if IPv6 is available; that is, if an interface has an IPv6 address that is (or should be) globally routable. Given the availability of both IPv6 and IPv4, a preference for IPv6 is implemented by assigning the prefix `::/0` – matching all IPv6 addresses – a higher precedence than that assigned to the IPv4-specific prefix `::ffff:0:0/96`. This is done in the table above.

Preferring IPv6 does not always work out well, however; many hosts have IPv6 connectivity through tunneling that may be slow, limited or outright down. The precedence table can be changed to prefer IPv4 over IPv6 by raising the precedence for the prefix `::ffff:0.0.0.0/96` to a value higher than that for `::/0`. Such system-wide configuration is usually done on Linux hosts by editing `/etc/gai.conf` and on

Windows via the `netsh` command; for example, `netsh interface ipv6 show prefixpolicies`.

We can see this systemwide IPv4/IPv6 preference in action using [OpenSSH](#) (see [29.5.1 SSH](#)), between two systems that each support both IPv4 and IPv6 (the remote system here is `intronetworks.cs.luc.edu`). With the IPv4–matching prefix precedence set high, connection is automatically via IPv4:

```
/etc/gai.conf: precedence ::ffff:0:0/96 100
ssh: Connecting to intronetworks.cs.luc.edu [162.216.18.28] ...
```

With the IPv4–prefix precedence set low, new connections use IPv6:

```
/etc/gai.conf: precedence ::ffff:0:0/96 10
ssh: Connecting to intronetworks.cs.luc.edu [2600:3c03::f03c:91ff:fe69:f438] ...
```

Applications can also use a DNS–resolver call that returns a *list* of all addresses matching a given hostname. (Often this list will have just two entries, for the IPv4 and IPv6 addresses, though round–robin DNS ([10.1 DNS](#)) can make the list much longer.) The C language `getaddrinfo()` call returns such a list, as does the Java `InetAddress.getAllByName()`. The [RFC 6724](#) preferences then determine the relative order of IPv4 and IPv6 entries in this list.

If an application requests such a list of all addresses, probably the most common strategy is to try each address in turn, according to the system–provided order. In the example of the previous paragraph, OpenSSH does in fact request a list of addresses, using `getaddrinfo()`, but, according to its source code, tries them in order and so usually connects to the first address on the list, that is, to the one preferred by the [RFC 6724](#) rules. Alternatively, an application might implement user–specified configuration preferences to decide between IPv4 and IPv6, though user interest in this tends to be limited (except, perhaps, by readers of this book).

12.4.1 Happy Eyeballs

The “Happy Eyeballs” algorithm, [RFC 8305](#), offers a more nuanced strategy for deciding whether an application should connect using IPv4 or IPv6. Initially, the client might try the IPv6 address (that is, will send TCP SYN to the IPv6 address, [17.3 TCP Connection Establishment](#)). If that connection does not succeed within, say, 250 ms, the client would try the IPv4 address. 250 ms is barely enough time for the TCP handshake to succeed; it does not allow – and is not meant to allow – sufficient time for a retransmission. The client falls back to IPv4 well before the failure of IPv6 is certain.

IPv6 servers

As of 2015, the list of websites supporting IPv6 was modest, though the number has crept up since then. Some sites, such as `apple.com` and `microsoft.com`, require the

“www” prefix for IPv6 availability. Networking providers are more likely to be IPv6–available. Sprint.com gets an honorable mention for having the shortest IPv6 address I found: 2600::aaaa.

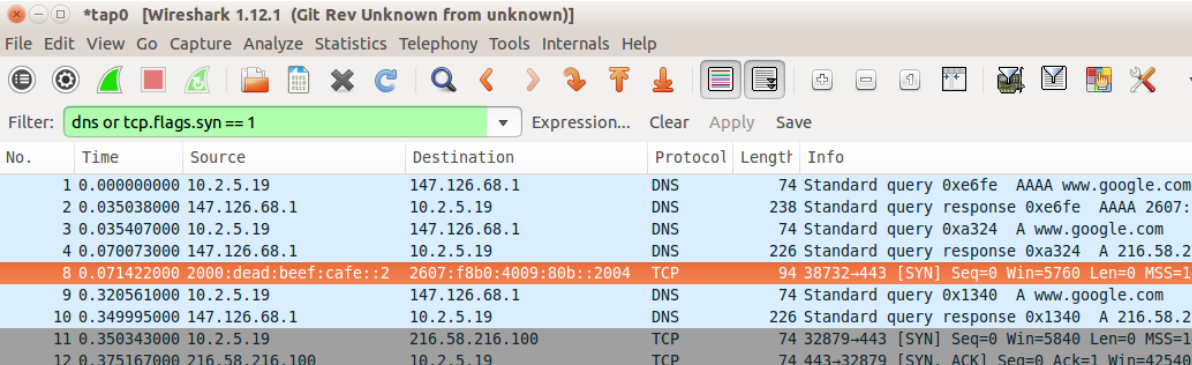
A Happy-Eyeballs client is also encouraged to **cache** the winning protocol, so for the next connection the client will attempt to use only the protocol that was successful before. The cache timeout is to be on the order of 10 minutes, so that if IPv6 connectivity failed and was restored then the client can resume using it with only moderate delay. Unfortunately, if the Happy Eyeballs mechanism is implemented at the *application* layer, which is often the case, then the scope of this cache may be limited to the particular application.

As IPv6 becomes more mainstream, Happy Eyeballs implementations are likely to evolve towards placing greater confidence in the IPv6 option. One simple change is to increase the time interval during which the client waits for an IPv6 response before giving up and trying IPv4.

We can test for the Happy Eyeballs mechanism by observing traffic with WireShark. As a first example, we imagine giving our client host a unique–local IPv6 address (in addition to its automatic link–local address); recall that unique–local addresses are not globally routable. If we now were to connect to, say, google.com, and monitor the traffic using WireShark, we would see a DNS AAAA query (IPv6) for “google.com” followed immediately by a DNS A query (IPv4). The subsequent TCP SYN, however, would be sent only to the IPv4 address: the client host would know that its IPv6 unique–local address is not routable, and it is not even tried.

Next let us change the IPv6 address for the client host to 2000:dead:beef:cafe::2, through manual configuration ([12.5.3 Manual address configuration](#)), and *without providing an actual IPv6 connection*. (We also manually specify a fake default router.) This address is part of the 2000::/3 block, and is *supposed* to be globally routable.

We now try two connections to google.com, TCP port 80. The first is via the Firefox browser.

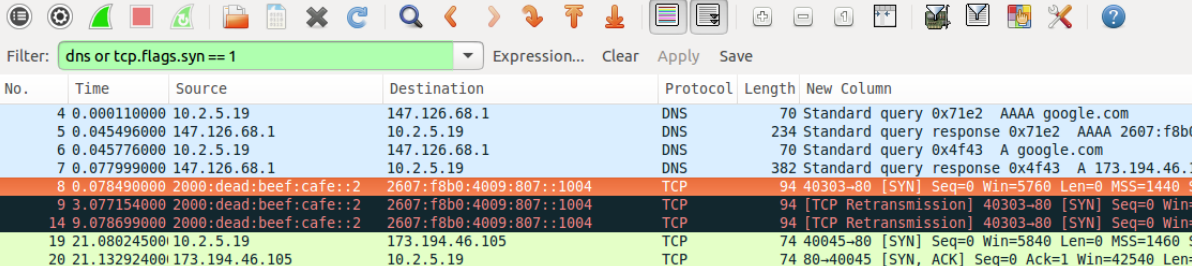


No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.2.5.19	147.126.68.1	DNS	74	Standard query 0xe6fe AAAA www.google.com
2	0.035038000	147.126.68.1	10.2.5.19	DNS	238	Standard query response 0xe6fe AAAA 2607::
3	0.035407000	10.2.5.19	147.126.68.1	DNS	74	Standard query 0xa324 A www.google.com
4	0.070073000	147.126.68.1	10.2.5.19	DNS	226	Standard query response 0xa324 A 216.58.21
8	0.071422000	2000:dead:beef:cafe::2	2607:f8b0:4009:80b::2004	TCP	94	38732->443 [SYN] Seq=0 Win=5760 Len=0 MSS=14
9	0.320561000	10.2.5.19	147.126.68.1	DNS	74	Standard query 0x1340 A www.google.com
10	0.349995000	147.126.68.1	10.2.5.19	DNS	226	Standard query response 0x1340 A 216.58.21
11	0.350343000	10.2.5.19	216.58.216.100	TCP	74	32879->443 [SYN] Seq=0 Win=5840 Len=0 MSS=14
12	0.375167000	216.58.216.100	10.2.5.19	TCP	74	443->32879 [SYN, ACK] Seq=0 Ack=1 Win=42540

We see two DNS queries, AAAA and A, in packets 1–4, followed by the first attempt (highlighted in orange) at T=0.071 to negotiate a TCP connection via IPv6 by sending a

TCP SYN packet (17.3 TCP Connection Establishment) to the google.com IPv6 address 2607:f8b0:4009:80b::200e. Only 250 ms later, at T=0.321, we see a second DNS A-query (IPv4), followed by an ultimately successful connection attempt using IPv4 starting at T=0.350. This particular version of Firefox, in other words, has implemented the Happy Eyeballs dual-stack mechanism.

Now we try the connection using the previously mentioned OpenSSH application, using -p 80 to connect to port 80. (This example was generated somewhat later; DNS now returns 2607:f8b0:4009:807::1004 as google.com's IPv6 address.)



Filter: dns or tcp.flags.syn == 1

No.	Time	Source	Destination	Protocol	Length	New Column
4	0.000110000	10.2.5.19	147.126.68.1	DNS	70	Standard query 0x71e2 AAAA google.com
5	0.045496000	147.126.68.1	10.2.5.19	DNS	234	Standard query response 0x71e2 AAAA 2607:f8b0:4009:80b::200e
6	0.045776000	10.2.5.19	147.126.68.1	DNS	70	Standard query 0x4f43 A google.com
7	0.077999000	147.126.68.1	10.2.5.19	DNS	382	Standard query response 0x4f43 A 173.194.46.105
8	0.078490000	2000::dead:beef:cafe::2	2607:f8b0:4009:807::1004	TCP	94	40303->80 [SYN] Seq=0 Win=5760 Len=0 MSS=1440 S
9	3.077154000	2000::dead:beef:cafe::2	2607:f8b0:4009:807::1004	TCP	94	[TCP Retransmission] 40303->80 [SYN] Seq=0 Win=
14	9.078699000	2000::dead:beef:cafe::2	2607:f8b0:4009:807::1004	TCP	94	[TCP Retransmission] 40303->80 [SYN] Seq=0 Win=
19	21.080245000	10.2.5.19	173.194.46.105	TCP	74	40045->80 [SYN] Seq=0 Win=5840 Len=0 MSS=1460 S
20	21.132924000	173.194.46.105	10.2.5.19	TCP	74	80->40045 [SYN, ACK] Seq=0 Ack=1 Win=42540 Len=

We see two DNS queries, AAAA and A, in packets numbered 4 and 6 (pale blue); these are made by the client from its IPv4 address 10.2.5.19. Half a millisecond after the A query returns (packet 7), the client sends a TCP SYN packet to google.com's IPv6 address; this packet is highlighted in orange. This SYN packet is retransmitted 3 seconds and then 9 seconds later (in black), to no avail. After 21 seconds, the client gives up on IPv6 and attempts to connect to google.com at its IPv4 address, 173.194.46.105; this connection (in green) is successful. The long delay shows that Happy Eyeballs was not implemented by OpenSSH, which its source code confirms.

(The host initiating the connections here was running Ubuntu 10.04 LTS, from 2010. The ultimately failing TCP connection gives up after three tries over only 21 seconds; newer systems make more tries and take much longer before they abandon a connection attempt.)

12.5 IPv6 Examples Without a Router

In this section we present a few IPv6 experiments that can be done without an IPv6 connection and without even an IPv6 router. Without a router, we cannot use SLAAC or DHCPv6. We will instead use link-local addresses, which require the specification of the interface along with the address, and manually configured unique-local (11.3 Network Prefixes) addresses. One practical problem with link-local addresses is that application documentation describing how to include a specification of the interface is sometimes sparse.

12.5.1 ping6

The IPv6 analogue of the familiar ping command, used to send ICMPv6 Echo Requests, is ping6 on Linux and Mac systems and ping -6 on Windows. The ping6 command

supports an option to specify the interface; *eg* `-I eth0`; as noted above, this is mandatory when sending to link-local addresses. Here are a few ping6 examples:

ping6 ::1: This pings the host's loopback address; it should always work.

ping6 -I eth0 ff02::1: This pings the all-nodes multicast group on interface eth0. Here are two of the answers received:

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

Answers were also received from a Windows machine and an Android phone. A VoIP phone – on the same subnet but supporting IPv4 only – remained mute, despite VoIP's difficulties with IPv4 NAT that would be avoided with IPv6. In lieu of the interface option `-I eth0`, the “zone-identifier” syntax **ping6 ff02::1%eth0** also usually works; see the following section.

ping6 -I eth0 fe80::2a0:ccff:fe24:b0e4: This pings the link-local address of the second Linux host answering the previous query; again, the **%eth0** syntax should also work. The destination interface identifier here uses the now-deprecated EUI-64 format; note the “ff:fe” in the middle. Also note the flipped seventh bit of the two bytes 02a0; the destination has Ethernet address 00:a0:cc:24:b0:e4.

12.5.2 TCP connections using link-local addresses

The next experiment is to create a TCP connection. Some commands, like ping6 above, may provide for a way of specifying the interface as a command-line option. Failing that, [RFC 4007](#) defines the concept of a **zone identifier** that is appended to the IPv6 address, separated from it by a “%” character, to specify the link involved. On Linux systems the zone identifier is most often the interface name, *eg* eth0 or ppp1. Numeric zone identifiers are also used, in which case it represents the number of the particular interface in some designated list and can be called the **zone index**. On Windows systems the zone index for an interface can often be inferred from the output of the `ipconfig` command, which should include it with each link-local address. The use of zone identifiers is often restricted to literal (numeric) IPv6 addresses, perhaps because there is little demand for symbolic link-local addresses.

The following link-local address with zone identifier creates an ssh connection to the second Linux host in the example of the preceding section:

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

That the ssh service is listening for IPv6 connections can be verified on that host by `netstat -a | grep -i tcp6`. That the ssh connection actually *used* IPv6 can be verified by, say, use of a network sniffer like WireShark (for which the filter expression `ipv6` or `ip.version == 6` is useful). If the connection fails, but ssh works

for IPv4 connections and shows as listening in the tcp6 list from the netstat command, a firewall-blocked port is a likely suspect.

12.5.3 Manual address configuration

The use of manually configured addresses is also possible, for either global or unique-local (*ie* not connected to the Internet) addresses. However, without a router there can be no Prefix Discovery, [11.6.2 Prefix Discovery](#), and this may create subtle differences.

The first step is to pick a suitable prefix; in the example below we use the unique-local prefix fd37:beef:cafe::/64 (though this particular prefix does *not* meet the randomness rules for unique-local prefixes). We could also use a globally routable prefix, but here we do not want to mislead any hosts about reachability.

Without a router as a source of Router Advertisements, we need some way to specify both the prefix and the prefix *length*; the latter can be thought of as corresponding to the IPv4 subnet mask. One might be forgiven for imagining that the default prefix length would be /64, given that this is the only prefix length generally allowed ([11.3 Network Prefixes](#)), but this is often not the case. In the commands below, the prefix length is included at the end as the /64. This usage is just slightly peculiar, in that in the IPv4 world the slash notation is most often used only with true prefixes, with all bits zero beyond the slash length. (The Linux `ip` command also uses the slash notation in the sense here, to specify an IPv4 subnet mask, *eg* 10.2.5.37/24. The `ifconfig` and Windows `netsh` commands specify the IPv4 subnet mask the traditional way, *eg* 255.255.255.0.)

Hosts will usually assume that a prefix configured this way with a length represents an **on-link** prefix, meaning that neighbors sharing the prefix are reachable directly via the LAN.

We can now assign the low-order 64 bits manually. On Linux this is done with:

- `host1: ip -6 address add fd37:beef:cafe::1/64 dev eth0`
- `host2: ip -6 address add fd37:beef:cafe::2/64 dev eth0`

Macintosh systems can be configured similarly except the name of the interface is probably `en0` rather than `eth0`. On Windows systems, a typical IPv6-address-configuration command is

```
netsh interface ipv6 add address "Local Area Connection" fd37:beef:cafe::1/64
```

Now on host1 the command

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

should create an ssh connection to host2, again assuming ssh on host2 is listening for IPv6 connections. Because the addresses here are not link-local, /etc/host entries may be created for them to simplify entry.

Assigning IPv6 addresses manually like this is *not* recommended, except for experiments.

On a LAN not connected to the Internet and therefore with no actual routing, it is nonetheless possible to start up a Router Advertisement agent ([11.6.1 Router Discovery](#)), such as **radvd**, with a manually configured /64 prefix. The RA agent will include this prefix in its advertisements, and reasonably modern hosts will then construct full addresses for themselves from this prefix using SLAAC. IPv6 can then be used within the LAN. If this is done, the RA agent should also be configured to announce only a meaningless route, such as ::/128, or else nodes may falsely believe the RA agent is providing full Internet connectivity.

12.6 IPv6 Connectivity via Tunneling

The best option for IPv6 connectivity is native support by one's ISP. In such a situation one's router should be sending out Router Advertisement messages, and from these all the hosts should discover how to reach the IPv6 Internet.

If native IPv6 support is not forthcoming, however, a short-term option is to connect to the IPv6 world using **packet tunneling** (less often, some other VPN mechanism is used). [RFC 4213](#) outlines the common **6in4** strategy of simply attaching an IPv4 header to the front of the IPv6 packet; it is very similar to the IPv4-in-IPv4 encapsulation of [9.9.1 IP-in-IP Encapsulation](#).

There are several available providers for this service; they can be found by searching for "IPv6 tunnel broker". Some tunnel brokers provide this service at no charge.

6in4, 6to4

6in4 tunneling should not be confused with 6to4 tunneling, which uses the same encapsulation as 6in4 but which constructs a site's IPv6 prefix by embedding its IPv4 address: a site with IPv4 address 129.3.5.7 gets IPv6 prefix 2002:8103:0507::/48 (129 decimal = 0x81). See [RFC 3056](#). There is also a 6over4, [RFC 2529](#).

The basic idea behind 6in4 tunneling is that the tunnel broker allocates you a /64 prefix out of its own address block, and agrees to create an IPv4 tunnel to you using 6in4 encapsulation. All your IPv6 traffic from the Internet is routed by the tunnel broker to you via this tunnel; similarly, IPv6 packets from your site reach the outside world using this same tunnel. The tunnel, in other words, is your link to an IPv6 router.

Generally speaking, the MTU of the tunnel must be at least 20 bytes less than the MTU of the physical interface, to allow space for the header. At the near end this requires a local configuration change; tunnel brokers often provide a way for users to set the MTU at the

far end. Practical MTU values vary from a mandatory IPv6 minimum of 1280 to the Ethernet maximum of $1500 - 20 = 1480$.

Setting up the tunnel does not involve creating a stateful connection. All that happens is that the tunnel client (*ie* your endpoint) and the broker record each other's IPv4 addresses, and agree to accept encapsulated IPv6 packets from one another provided these two endpoint addresses are used as source and destination. The tunnel at the client end is represented by an appropriate "virtual network interface", *eg* `sit0` or `gif0` or `IP6Tunnel`. Tunnel providers generally supply the basic commands necessary to get the tunnel interface configured and the MTU set.

Once the tunnel is created, the tunnel interface at the client end must be assigned an IPv6 address and then a (default) route. We will assume that the /64 prefix for the broker-to-client link is `2001:470:0:10::/64`, with the broker at `2001:470:0:10::1` and with the client to be assigned the address `2001:470:0:10::2`. The address and route are set up on the client with the following commands (Linux/Mac/Windows respectively; interface names may vary, and some commands assume the interface represents a point-to-point link):

```
ip addr add 2001:470:0:10::2/64 dev sit1
ip route add ::/0 dev sit1

ifconfig gif0 inet6 2001:470:0:10::2 2001:470:0:10::1 prefixlen 128
route -n add -inet6 default 2001:470:0:10::1

netsh interface ipv6 add address IP6Tunnel 2001:470:0:10::2
netsh interface ipv6 add route ::/0 IP6Tunnel 2001:470:0:10::1
```

At this point the tunnel client should have full IPv6 connectivity! To verify this, one can use `ping6`, or visit IPv6-only versions of websites (*eg* intronetworks6.cs.luc.edu), or visit IPv6-identifying sites such as IsMyIPv6Working.com. Alternatively, one can often install a browser plugin to at least make visible whether IPv6 is used. Finally, one can use `netcat` with the `-6` option to force IPv6 use, following the HTTP example in [17.7.1 netcat again](#).

There is one more potential issue. If the tunnel client is behind an IPv4 NAT router, that router must deliver arriving encapsulated 6in4 packets correctly. This can sometimes be a problem; encapsulated 6in4 packets are at some remove from the TCP and UDP traffic that the usual consumer-grade NAT router is primarily designed to handle. Careful study of the router forwarding settings may help, but sometimes the only fix is a newer router. A problem is particularly likely if two different inside clients attempt to set up tunnels simultaneously; see [9.9.1 IP-in-IP Encapsulation](#).

12.6.1 IPv6 firewalls

It is strongly recommended that an IPv6 host **block new inbound connections** over its IPv6 interface (*eg* the tunnel interface), much as an IPv4 NAT router would do. Exceptions

may be added as necessary for essential services (such as ICMPv6). Using the linux `ip6tables` firewall command, with IPv6-tunneled interface `sit1`, this might be done with the following:

```
ip6tables --append INPUT --in-interface sit1 --protocol icmpv6 --jump ACCEPT
ip6tables --append INPUT --in-interface sit1 --match conntrack --ctstate
ESTABLISHED,RELATED --jump ACCEPT
ip6tables --append INPUT --in-interface sit1 --jump DROP
```

At this point the firewall should be tested by attempting to access inside hosts from the outside. At a minimum, ping6 from the outside to any global IPv6 address of any inside host should fail if the ICMPv6 exception above is removed (and should succeed if the ICMPv6 exception is restored). This can be checked by using any of several websites that send pings on request; such sites can be found by searching for “online ipv6 ping”. There are also a few sites that will run a remote IPv6 TCP port scan; try searching for “online ipv6 port scan”. See also exercise 4.0.

12.6.2 Setting up a router

The next step, if desired, is to set up the tunnel endpoint as a router, so other hosts at the client site can also enjoy IPv6 connectivity. For this we need a second /64 prefix; we will assume this is `2001:470:0:20::/64` (note this is not an “adjacent” /64; the two /64 prefixes cannot be merged into a /63). Let `R` be the tunnel endpoint, with `eth0` its LAN interface, and let `A` be another host on the LAN.

We will use the linux `radvd` package as our Router Advertisement agent ([11.6.1 Router Discovery](#)). In the `radvd.conf` file, we need to say that we want the LAN prefix `2001:470:0:20::/64` advertised as on-link over interface `eth0`:

```
interface eth0 {
    ...
    prefix 2001:470:0:20::/64
    {
        AdvOnLink on;           # advertise this prefix as on-link
        AdvAutonomous on;      # allows SLAAC with this prefix
    };
};
```

If `radvd` is now started, other LAN hosts (eg `A`) will automatically get the prefix (and thus a full SLAAC address). `Radvd` will automatically share `R`’s default route (`::/0`), taking it not from the configuration file but from `R`’s routing table. (It may still be necessary to manually configure the IPv6 address of `R`’s `eth0` interface, eg as `2001:470:0:20::1`.)

On the author’s version of host `A`, the IPv6 route is now (with some irrelevant attributes not shown)

```
default via fe80::2a0:ccff:fe24:b0e4 dev eth0
```

That is, host A routes to R via the latter's **link-local** address, always guaranteed on-link, rather than via the subnet address.

If `radvd` or its equivalent is not available, the manual approach is to assign R and A each a /64 address:

```
On host R: ip -6 address add 2001:470:0:20::1/64 dev eth0
```

```
On host A: ip -6 address add 2001:470:0:20::2/64 dev eth0
```

Because of the “/64” here ([12.5.3 Manual address configuration](#)), R and A understand that they can reach each other via the LAN, and do so. Host A also needs to be told of the default route via R:

```
On host A: ip -
```

```
6 route add ::/0 via 2001:470:0:10::1 dev eth0
```

Here we use the subnet address of R, but we could have used R's link-local address as well.

It is likely that A's `eth0` will also need its MTU configured, so that it matches that of R's virtual tunnel interface (which, recall, should be at least 20 bytes less than the MTU of R's physical outbound interface).

12.6.2.1 A second router

Now let us add a second router R2, as in the diagram below. The R—R2 link is via a separate Ethernet LAN, not a point-to-point link. The LAN with A is, as above, subnet 2001:470:0:20::/64.

In this case, it is R2 that needs to run the Router Advertisement agent (*eg* `radvd`). If this were an IPv4 network, the interfaces `eth0` and `eth1` on the R—R2 link would need IPv4 addresses from some new subnet (though the use of private addresses is an option). We can't use unnumbered interfaces ([9.8 Unnumbered Interfaces](#)), because the R—R2 connection is not a point-to-point link.

But with IPv6, we can configure the R—R2 routing to use only link-local addresses. Let us assume for mnemonic convenience these are as follows:

```
R's eth0: fe80::ba5e:ba11
```

```
R2's eth1: fe80::dead:beef
```

R2's forwarding table will have a default route with `next_hop fe80::ba5e:ba11` (R). Similarly, R's forwarding table will have an entry for destination subnet 2001:470:0:20::/64 with `next_hop fe80::dead:beef` (R2). Neither `eth0` nor `eth1` needs any other IPv6 address.

R2's eth2 interface will likely need a global IPv6 address, *eg* 2001:470:0:20::1 again. Otherwise R2 may not be able to determine that its eth2 interface is in fact connected to the 2001:470:0:20::/64 subnet.

One advantage of not giving eth0 or eth1 global addresses is that it is then impossible for an outside attacker to reach these interfaces directly. It also saves on subnets, although one hopes with IPv6 those are not in short supply. All routers at a site are likely to need, for management purposes, an IP address reachable throughout the site, but this does not have to be globally visible.

12.7 IPv6-to-IPv4 Connectivity

What happens if you switch to IPv6 completely, perhaps because your ISP (or phone provider) has run out of IPv4 addresses? Some of the time – hopefully more and more of the time – you will only need to talk to IPv6 servers. For example, the DNS names `facebook.com` and `google.com` each correspond to an IPv4 address, but also to an IPv6 address (above). But what do you do if you want to reach an IPv4-only server? Such servers are expected to continue operating for a long time to come. It is necessary to have some sort of centralized IPv6-to-IPv4 **translator**.

An early strategy was NAT-PT ([RFC 2766](#)). The translator was assigned a /96 prefix. The IPv6 host would append to this prefix the 32-bit IPv4 address of the destination, and use the resulting address to contact the IPv4 destination. Packets sent to this address would be delivered via IPv6 to the translator, which would translate the IPv6 header into IPv4 and then send the translated packet on to the IPv4 destination. As in IPv4 NAT ([9.7 Network Address Translation](#)), the reverse translation will typically involve TCP port numbers to resolve ambiguities. This approach requires the IPv6 host to be aware of the translator, and is limited to TCP and UDP (because of the use of port numbers). Due to these and several other limitations, NAT-PT was formally deprecated in [RFC 4966](#).

Do you still have IPv4 service?

As of 2017, several phone providers have switched many of their customers to IPv6 while on their mobile-data networks. The change can be surprisingly inconspicuous. Connections to IPv4-only services still work just fine, courtesy of NAT64. About the only way to tell is to look up the phone's IP address.

The replacement protocol is **NAT64**, documented in [RFC 6146](#). This is also based on address translation, and, as such, cannot allow connections initiated from IPv4 hosts to IPv6 hosts. It is, however, transparent to both the IPv6 and IPv4 hosts involved, and is not restricted to TCP (though only TCP, UDP and ICMP are supported by [RFC 6146](#)). It uses a special DNS variant, DNS64 ([RFC 6147](#)), as a companion protocol.

To use NAT64, an IPv6 client sends out its ordinary DNS query to find the addresses of the destination server. The DNS resolver ([10.1 DNS](#)) receiving the request must use DNS64. If the destination has only an IPv4 address, then the DNS resolver will return to

the IPv6 client (as an AAAA record) a synthetic IPv6 address consisting of a prefix and the embedded IPv4 address of the server, much as in NAT-PT above (though multiple prefix-length options exist; see [RFC 6052](#)). The prefix belongs to the actual NAT64 translator; any packet addressed to an IPv6 address starting with the prefix will be delivered to the translator. There is no relationship between the NAT64 translator and the DNS64 resolver beyond the fact that the former's prefix is configured into the latter.

The IPv6 client now uses this synthetic IPv6 address to contact the IPv4 server. Its packets will be routed to the NAT64 translator itself, by virtue of the prefix, much as in NAT-PT. Upon receiving the first packet from the IPv6 client, the NAT64 translator will assign one of its IPv4 addresses to the new connection. As IPv4 addresses are in short supply, this pool of available IPv4 addresses may be small, so NAT64 allows one IPv4 address to be used by many IPv6 clients. To this end, the NAT64 translator will also (for TCP and UDP) establish a port mapping between the incoming IPv6 source port and a port number allocated by the NAT64 to ensure that traffic is uniquely reversible. As with IPv4 NAT, if two IPv6 clients try to contact the same IPv4 server using the same source ports, and are assigned the same NAT64 IPv4 address, then one of the clients will have its port number changed.

If an ICMP query is being sent, the Query Identifier is used in lieu of port numbers. To extend NAT64 to new protocols, an appropriate analog of port numbers must be identified, to allow demultiplexing of multiple connections sharing a single IPv4 address.

After the translation is set up, by creating appropriate table entries, the translated packet is sent on to the IPv4 server address that was embedded in the synthetic IPv6 address. The source address will be the assigned IPv4 address of the translator, and the source port will have been rewritten in accordance with the new port mapping. At this point packets can flow freely between the original IPv6 client and its IPv4 destination, with neither endpoint being aware of the translation (unless the IPv6 client carefully inspects the synthetic address it receives via DNS64). A timer within the NAT64 translator will delete the association between the IPv6 and IPv4 addresses if the connection is not used for a while.

As an example, suppose the IPv6 client has address 2000:1234::abba, and is trying to reach *intronetworks4.cs.Luc.edu* at TCP port 80. It contacts its DNS server, which finds no AAAA record but IPv4 address 162.216.18.28 (in hex, a2d8:121c). It takes the prefix for its NAT64 translator, which we will assume is 2000:cafe::, and returns the synthetic address 2000:cafe::a2d8:121c.

The IPv6 client now tries to connect to 2000:cafe::a2d8:121c, using source port 4000. The first packet arrives at the NAT64 translator, which assigns the connection the outbound IPv4 address of 200.0.0.1, and reassigns the source port on the IPv4 side to 4002. The new IPv4 packet is sent on to 162.216.18.28. The reply

from *intronetworks4.cs.Luc.edu* comes back, to $\langle 200.0.0.1, 4002 \rangle$. The NAT64 translator looks this up and finds that this corresponds to $\langle 2000:1234::abba, 4000 \rangle$, and forwards it back to the original IPv6 client.

12.7.1 IPv6-to-IPv6 Connectivity

While we are on the subject of connectivity, there is a significant lack of connectivity within the IPv6 world: two major ISPs do not connect to one another, neither directly nor indirectly. As of 2022, [Cogent Communications](#) and [Hurricane Electric](#) have no connectivity via IPv6, and apparently have not connected for some years. This has happened occasionally in the IPv4 world, but usually the ISPs involved come to an agreement quickly.

Each company maintains a **looking-glass** site (reachable via IPv4), from which one can launch IPv6 pings and traceroutes; these are [cogentco.com/en/looking-glass](#) and [lg.he.net](#). From Cogent, one can choose either an IPv4 or an IPv6 ping; the IPv6 ping to he.net fails. The same happens from Hurricane Electric to cogentco.com, though the selection of IPv4 or IPv6 is made after entering the destination.

Ultimately, this situation is due to a disagreement as to who should *pay* for the interconnection, or who should pay what share. In the language of [15.10 BGP Relationships](#), both ISPs are top-level backbone providers, or “peers”. Both are generally considered “tier-1”, although a common defining rule for tier-1 providers is that they directly connect to every other Tier 1 provider, and these two do not.

This situation, while definitely a problem, is not necessarily as calamitous as it may sound; IPv6 customers of each may be able to reach all major IPv6 services; they just cannot reach each other. While IPv6, with its general absence of NAT, supports direct connections between individual end-users, most connections are between end-users and servers, and most of these connections still work. Customers of *other* ISPs typically have full connectivity, including to Cogent and to HE.

12.8 Epilog

IPv4 and IPv6 are, functionally, rather similar. However, the widespread use of NAT in the IPv4 world makes IPv4 in practice appear rather different. IPv4 and IPv6 can, of course, coexist side-by-side, as two parallel and independent IP layers. But the demand for IPv4-to-IPv6 connectivity has led to multiple solutions.

12.9 Exercises

Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises.

1.0. Suppose someone tried to implement ping6 so that, if the address was a link-local address and no interface was specified, the ICMPv6 Echo Request was sent out all non-loopback interfaces. Could the end result be different than conventional ping6 with the correct interface supplied? If so, how likely is this?

2.0. Create an IPv6 ssh connection as in [12.5 IPv6 Examples Without a Router](#). Examine the connection's packets using WireShark or the equivalent. Does the TCP handshake ([17.3 TCP Connection Establishment](#)) look any different over IPv6?

3.0. Create an IPv6 ssh connection using manually configured addresses as in [12.5.3 Manual address configuration](#). Again use WireShark or the equivalent to monitor the connection. Is DAD ([11.7.1 Duplicate Address Detection](#)) used?

4.0. Suppose host A gets its IPv6 traffic through tunnel provider H, as in [12.6 IPv6 Connectivity via Tunneling](#). To improve security, A blocks all packets that are not part of connections it has initiated (which is common), and makes no exception for ICMPv6 traffic (which is not a good idea). H is correctly configured to know the MTU of the A-H link. For (a) and (b), this MTU is 1280, the minimum allowed for IPv6. Much of the Internet, however, allows larger MTU values.

A — H — Internet — B

(a). If A attempts to send a larger-than-1280-byte IPv6 packet to remote host B, will A be informed of the resultant failure? Why or why not?

(b). Suppose B attempts to send a larger-than-1280-byte IPv6 packet to A. Will B receive an ICMPv6 Packet Too Big message? Why or why not?

(c). Now suppose the MTU of the A-H link is raised to 1400 bytes. Outline a scenario in which A sends a packet of size greater than 1280 bytes to remote host B, the packet is too big to make it all the way to B, and yet A receives no notification of this.

13 Routing-Update Algorithms

How do IP routers build and maintain their forwarding tables?

Ethernet bridges always have the option of fallback-to-flooding for unknown destinations, so they can afford to build their forwarding tables “incrementally”, putting a host into the forwarding table only when that host is first seen as a *sender*. For IP, there is no fallback delivery mechanism: forwarding tables must be built *before* delivery can succeed. While manual table construction is possible, it is not practical.

In the literature it is common to refer to router-table construction as “routing algorithms”. We will avoid that term, however, to avoid confusion with the fundamental datagram-forwarding algorithm; instead, we will call these “routing-update algorithms”.

The two classes of algorithms we will consider here are **distance–vector** and **link–state**. In the distance–vector approach, often used at smaller sites, routers exchange information with their immediately neighboring routers; tables are built up this way through a sequence of such periodic exchanges. In the link–state approach, routers rapidly propagate information about the state of each link; all routers in the organization receive this link–state information and each one uses it to build and maintain a map of the entire network. The forwarding table is then constructed (sometimes on demand) from this map.

Both approaches assume that consistent information is available as to the **cost** of each link (*eg* that the two routers at opposite ends of each link know this cost, and agree on how the cost is determined). This requirement classifies these algorithms as **interior** routing–update algorithms: the routers involved are internal to a larger organization or other common administrative regime that has an established policy on how to assign link weights. The set of routers following a common policy is known as a **routing domain** or (from the BGP protocol) an **autonomous system**.

The simplest link–weight strategy is to give each link a cost of 1; link costs can also be based on bandwidth, propagation delay, financial cost, or administrative preference value. Careful assignment of link costs often plays a major role in herding traffic onto the faster or “better” links.

In the following chapter we will look at the Border Gateway Protocol, or BGP, in which no link–cost calculations are made. BGP is used to select routes that traverse other organizations, and financial rather than technical factors may therefore play the dominant role in making routing choices.

Generally, all these algorithms apply to IPv6 as well as IPv4, though specific protocols of course may need modification.

Finally, we should point out that from the early days of the Internet, routing was allowed to depend not just on the destination, but also on the “quality of service” (QoS) requested; thus, forwarding table entries are strictly speaking not $\langle \text{destination}, \text{next_hop} \rangle$ but rather $\langle \text{destination}, \text{QoS}, \text{next_hop} \rangle$. Originally, the Type of Service field in the IPv4 header ([9.1 The IPv4 Header](#)) could be used to specify QoS (often then called ToS). Packets could request low delay, high throughput or high reliability, and could be routed accordingly. In practice, the Type of Service field was rarely used, and was eventually taken over by the DS field and ECN bits. The first three bits of the Type of Service field, known as the precedence bits, remain available, however, and can still be used for QoS routing purposes (see the Class Selector PHB of [25.7 Differentiated Services](#) for examples of these bits). See also [RFC 2386](#).

In much of the following, we are going to ignore QoS information, and assume that routing decisions are based only on the destination. See, however, the first paragraph

of [13.5 Link-State Routing-Update Algorithm](#), and also [13.6 Routing on Other Attributes](#).

13.1 Distance-Vector Routing-Update Algorithm

Distance-vector is the simplest routing-update algorithm, used by the Routing Information Protocol, or RIP. Version 2 of the protocol is specified in [RFC 2453](#).

Routers identify their router neighbors (through some sort of neighbor-discovery mechanism), and add a third column to their forwarding tables representing the total **cost** for delivery to the corresponding destination. These costs are the “distance” of the algorithm name. Forwarding-table entries are now of the form $\langle \text{destination}, \text{next_hop}, \text{cost} \rangle$.

Costs are administratively assigned to each link, and the algorithm then calculates the total cost to a destination as the sum of the link costs along the path. The simplest case is to assign a cost of 1 to each link, in which case the total cost to a destination will be the number of links to that destination. This is known as the “hopcount” metric; it is also possible to assign link costs that reflect each link’s bandwidth, or delay, or whatever else the network administrators wish. Thoughtful cost assignments are a form of traffic engineering and sometimes play a large role in network performance.

At this point, each router then **reports** the $\langle \text{destination}, \text{cost} \rangle$ portion of its table to its neighboring routers at regular intervals; these table portions are the “vectors” of the algorithm name. It does not matter if neighbors exchange reports at the same time, or even at the same rate.

Each router also monitors its continued connectivity to each neighbor; if neighbor N becomes unreachable then its reachability cost is set to infinity.

In a real IP network, actual destinations would be *subnets* attached to routers; one router might be directly connected to several such destinations. In the following, however, we will identify all a router’s directly connected subnets with the router itself. That is, we will build forwarding tables to reach every *router*. While it is possible that one destination subnet might be reachable by two or more routers, thus breaking our identification of a router with its set of attached subnets, in practice this is of little concern. See exercise 6.0 for an example in which subnets are *not* identified with adjacent routers.

In [30.5 IP Routers With Simple Distance-Vector Implementation](#) we present a simplified working implementation of RIP using the Mininet network emulator.

13.1.1 Distance-Vector Update Rules

Let A be a router receiving a report $\langle D, c_D \rangle$ from neighbor N at cost c_N . Note that this means A can reach D *via* N with cost $c = c_D + c_N$. A updates its own table according to the following three rules:

1. **New destination:** D is a previously unknown destination. A adds $\langle D, N, c \rangle$ to its forwarding table.
2. **Lower cost:** D is a known destination with entry $\langle D, M, c_{old} \rangle$, but the new total cost c is less than c_{old} . A switches to the cheaper route, updating its entry for D to $\langle D, N, c \rangle$. It is possible that $M=N$, meaning that N is now reporting a cost decrease to D. (If $c = c_{old}$, A ignores the new report; see exercise 8.0.)
3. **Next_hop increase:** A has an existing entry $\langle D, N, c_{old} \rangle$, and the new total cost c is *greater* than c_{old} . Because this is a cost increase from the neighbor N that A is currently using to reach D, A must incorporate the increase in its table. A updates its entry for D to $\langle D, N, c \rangle$.

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

The third rule, however, introduces the possibility of instability, as a cost may also go up. It represents the **bad-news** case, in that neighbor N has learned that some link failure has driven up its own cost to reach D, and is now passing that “bad news” on to A, which routes to D *via* N.

The next_hop-increase case only passes bad news along; the very first cost increase must always come from a router discovering that a neighbor N is unreachable, and thus updating its cost to N to ∞ . Similarly, if router A learns of a next_hop increase to destination D from neighbor B, then we can follow the next_hops back until we reach a router C which is either the originator of the cost= ∞ report, or which has learned of an alternative route through one of the first two rules.

13.1.2 Example 1

For our first example, no links will break and thus only the first two rules above will be used. We will start out with the network below with empty forwarding tables; all link costs are 1.

After initial neighbor discovery, here are the forwarding tables. Each node has entries only for its directly connected neighbors:

A: $\langle B, B, 1 \rangle \langle C, C, 1 \rangle \langle D, D, 1 \rangle$
 B: $\langle A, A, 1 \rangle \langle C, C, 1 \rangle$
 C: $\langle A, A, 1 \rangle \langle B, B, 1 \rangle \langle E, E, 1 \rangle$
 D: $\langle A, A, 1 \rangle \langle E, E, 1 \rangle$
 E: $\langle C, C, 1 \rangle \langle D, D, 1 \rangle$

Now let D report to A; it sends records $\langle A,1 \rangle$ and $\langle E,1 \rangle$. A ignores D's $\langle A,1 \rangle$ record, but $\langle E,1 \rangle$ represents a new destination; A therefore adds $\langle E,D,2 \rangle$ to its table. Similarly, let A now report to D, sending $\langle B,1 \rangle \langle C,1 \rangle \langle D,1 \rangle \langle E,2 \rangle$ (the last is the record we just added). D ignores A's records $\langle D,1 \rangle$ and $\langle E,2 \rangle$ but A's records $\langle B,1 \rangle$ and $\langle C,1 \rangle$ cause D to create entries $\langle B,A,2 \rangle$ and $\langle C,A,2 \rangle$. A and D's tables are now, in fact, complete.

Now suppose C reports to B; this gives B an entry $\langle E,C,2 \rangle$. If C also reports to E, then E's table will have $\langle A,C,2 \rangle$ and $\langle B,C,2 \rangle$. The tables are now:

A: $\langle B,B,1 \rangle \langle C,C,1 \rangle \langle D,D,1 \rangle \langle E,D,2 \rangle$

B: $\langle A,A,1 \rangle \langle C,C,1 \rangle \langle E,C,2 \rangle$

C: $\langle A,A,1 \rangle \langle B,B,1 \rangle \langle E,E,1 \rangle$

D: $\langle A,A,1 \rangle \langle E,E,1 \rangle \langle B,A,2 \rangle \langle C,A,2 \rangle$

E: $\langle C,C,1 \rangle \langle D,D,1 \rangle \langle A,C,2 \rangle \langle B,C,2 \rangle$

We have two missing entries: B and C do not know how to reach D. If A reports to B and C, the tables will be complete; B and C will each reach D via A at cost 2. However, the following sequence of reports might also have occurred:

- E reports to C, causing C to add $\langle D,E,2 \rangle$
- C reports to B, causing B to add $\langle D,C,3 \rangle$

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

Note that A routes to E via D while E routes to A via C; this asymmetry was due to indeterminateness in the order of initial table exchanges.

If all link weights are 1, and if each pair of neighbors exchange tables once before any pair starts a second exchange, then the above process will discover the routes in order of length, i.e. the shortest paths will be the first to be discovered. This is not, however, a particularly important consideration.

13.1.3 Example 2

The next example illustrates link weights other than 1. The first route discovered between A and B is the direct route with cost 8; eventually we discover the longer A-C-D-B route with cost $2+1+3=6$.

The initial tables are these:

A: $\langle C,C,2 \rangle \langle B,B,8 \rangle$

B: $\langle A,A,8 \rangle \langle D,D,3 \rangle$

C: $\langle A,A,2 \rangle \langle D,D,1 \rangle$

D: $\langle B, B, 3 \rangle \langle C, C, 1 \rangle$

After A and C exchange, A has $\langle D, C, 3 \rangle$ and C has $\langle B, A, 10 \rangle$. After C and D exchange, C updates its $\langle B, A, 10 \rangle$ entry to $\langle B, D, 4 \rangle$ and D adds $\langle A, C, 3 \rangle$; D receives C's report of $\langle B, 10 \rangle$ but ignores it.

Now finally suppose B and D exchange. D ignores B's route to A, as it has a better one. B, however, gets D's report $\langle A, 3 \rangle$ and updates its entry for A to $\langle A, D, 6 \rangle$. D also reports $\langle C, 1 \rangle$ and so B creates an entry $\langle C, D, 4 \rangle$ for C. At this point the tables are as follows:

A: $\langle C, C, 2 \rangle \langle B, B, 8 \rangle \langle D, C, 3 \rangle$

B: $\langle A, D, 6 \rangle \langle D, D, 3 \rangle \langle C, D, 4 \rangle$

C: $\langle A, A, 2 \rangle \langle D, D, 1 \rangle \langle B, D, 4 \rangle$

D: $\langle B, B, 3 \rangle \langle C, C, 1 \rangle \langle A, C, 3 \rangle$

We have one more thing to fix before we are done: A has an inefficient route to B. This will be fixed when C reports $\langle B, 4 \rangle$ to A; A will replace its route to B with $\langle B, C, 6 \rangle$. If we look only at the A-B route, B discovers the lower-cost route to A when, first, C reports to D and, second, *after* that, D reports to B; a similar sequence leads to A's discovering the lower-cost route.

13.1.4 Example 3

Our third example will illustrate how the algorithm proceeds when a link **breaks**. We return to the first diagram, with all tables completed, and then suppose the D-E link breaks. This is the "bad-news" case: a link has broken, and is no longer available; this will bring the third rule into play.

We shall assume, as above, that A reaches E via D, but we will here assume – contrary to Example 1 – that C reaches D via A (see exercise 5.0 for the original case).

Initially, upon discovering the break, D and E update their tables to $\langle E, -, \infty \rangle$ and $\langle D, -, \infty \rangle$ respectively (whether or not they actually enter ∞ into their tables is implementation-dependent; we may consider this as equivalent to *removing* their entries for one another; the "-" as next_hop indicates there is no next_hop).

Eventually D and E will report the break to their respective neighbors A and C. A will apply the "bad-news" rule above and update its entry for E to $\langle E, -, \infty \rangle$. We have assumed that C, however, routes to D via A, and so it will ignore E's report.

We will suppose that the next steps are for C to report to E and to A. When C reports its route $\langle D, 2 \rangle$ to E, E will add the entry $\langle D, C, 3 \rangle$, and will again be able to reach D. When C reports to A, A will add the route $\langle E, C, 2 \rangle$. The final step will be when A next reports to D, and D will have $\langle E, A, 3 \rangle$. Connectivity is restored.

13.1.5 Example 4

The previous examples have had a “global” perspective in that we looked at the entire network. In the next example, we look at how one specific router, R, responds when it receives a distance–vector report from its neighbor S. Neither R nor S nor we have any idea of what the entire network looks like. Suppose R’s table is initially as follows, and the S–R link has cost 1:

destination	next_hop	cost
A	S	3
B	T	4
C	S	5
D	U	6

S now sends R the following report, containing only destinations and its costs:

destination	cost
A	2
B	3
C	5
D	4
E	2

R then updates its table as follows:

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

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

13.2 Distance–Vector Slow–Convergence Problem

There is a significant problem with distance–vector table updates in the presence of broken links. Not only can routing loops form, but the loops can persist indefinitely! As an example, suppose we have the following arrangement, with all links having cost 1:

Now suppose the D–A link breaks:

If A immediately reports to B that D is no longer reachable (cost = ∞), then all is well. However, it is possible that B reports to A first, telling A that it has a route to D, with cost 2, which B still believes it has.

This means A now installs the entry $\langle D, B, 3 \rangle$. At this point we have what we called in [1.6 Routing Loops](#) a linear routing loop: if a packet is addressed to D, A will forward it to B and B will forward it back to A.

Worse, this loop will be with us a while. At some point A will report $\langle D, 3 \rangle$ to B, at which point B will update its entry to $\langle D, A, 4 \rangle$. Then B will report $\langle D, 4 \rangle$ to A, and A's entry will be $\langle D, B, 5 \rangle$, etc. This process is known as **slow convergence to infinity**. If A and B each report to the other once a minute, it will take 2,000 years for the costs to overflow an ordinary 32-bit integer.

13.2.1 Slow-Convergence Fixes

The simplest fix to this problem is to use a small value for infinity. Most flavors of the RIP protocol use $\text{infinity} = 16$, with updates every 30 seconds. The drawback to so small an infinity is that no path through the network can be longer than this; this makes paths with weighted link costs difficult. Cisco IGRP uses a variable value for infinity up to a maximum of 256; the default infinity is 100.

There are several well-known other fixes:

13.2.1.1 Split Horizon

Under split horizon, if A uses N as its next_hop for destination D, then A simply does not report to N that it can reach D; that is, in preparing its report to N it first deletes all entries that have N as next_hop. In the example above, split horizon would mean B would never report to A about the reachability of D because A is B's next_hop to D.

Split horizon prevents all linear routing loops. However, there are other topologies where it cannot prevent loops. One is the following:

Suppose the A-D link breaks, and A updates to $\langle D, -, \infty \rangle$. A then reports $\langle D, \infty \rangle$ to B, which updates its table to $\langle D, -, \infty \rangle$. But then, before A can also report $\langle D, \infty \rangle$ to C, C reports $\langle D, 2 \rangle$ to B. B then updates to $\langle D, C, 3 \rangle$, and reports $\langle D, 3 \rangle$ back to A; neither this nor the previous report violates split-horizon. Now A's entry is $\langle D, B, 4 \rangle$. Eventually A will report to C, at which point C's entry becomes $\langle D, A, 5 \rangle$, and the numbers keep increasing as the reports circulate counterclockwise. The actual routing proceeds in the other direction, clockwise.

Split horizon often also includes **poison reverse**: if A uses N as its next_hop to D, then A in fact reports $\langle D, \infty \rangle$ to N, which is a more definitive statement that A cannot reach D by

itself. However, coming up with a scenario where poison reverse actually affects the outcome is not trivial.

13.2.1.2 Triggered Updates

In the original example, if A was first to report to B then the loop resolved immediately; the loop occurred if B was first to report to A. Nominally each outcome has probability 50%. Triggered updates means that any router should report immediately to its neighbors whenever it detects any change for the worse. If A reports first to B in the first example, the problem goes away. Similarly, in the second example, if A reports to both B and C before B or C report to one another, the problem goes away. There remains, however, a small window where B could send its report to A just as A has discovered the problem, before A can report to B.

13.2.1.3 Hold Down

Hold down is sort of a receiver-side version of triggered updates: the receiver does not use new alternative routes for a period of time (perhaps two router-update cycles) following discovery of unreachability. This gives time for bad news to arrive. In the first example, it would mean that when A received B's report $\langle D, 2 \rangle$, it would set this aside. It would then report $\langle D, \infty \rangle$ to B as usual, at which point B would now report $\langle D, \infty \rangle$ back to A, at which point B's earlier report $\langle D, 2 \rangle$ would be discarded. A significant drawback of hold down is that legitimate new routes are also delayed by the hold-down period.

These mechanisms for preventing slow convergence are, in the real world, quite effective. The Routing Information Protocol (RIP, [RFC 2453](#)) implements all but hold-down, and has been widely adopted at smaller installations.

However, the potential for routing loops and the limited value for infinity led to the development of alternatives. One alternative is the link-state strategy, [13.5 Link-State Routing-Update Algorithm](#). Another alternative is Cisco's Enhanced Interior Gateway Routing Protocol, or EIGRP, [13.4.4 EIGRP](#). While part of the distance-vector family, EIGRP is provably loop-free, though to achieve this it must sometimes suspend forwarding to some destinations while tables are in flux.

13.3 Observations on Minimizing Route Cost

Does distance-vector routing actually achieve minimum costs? For that matter, does each packet incur the cost its sender expects? Suppose node A has a forwarding entry $\langle D, B, c \rangle$, meaning that A forwards packets to destination D via next_hop B, and expects the total cost to be c. If A sends a packet to D, and we follow it on the actual path it takes, must the total link cost be c? If so, we will say that the network has **accurate costs**.

The answer to the accurate-costs question, as it turns out, is *yes* for the distance-vector algorithm, if we follow the rules carefully, and the network is stable (meaning that no

routing reports are changing, or, more concretely, that every update report now circulating is based on the current network state); a proof is below. However, if there is a routing loop, the answer is of course no: the actual cost is now infinite. The answer would also be no if A's neighbor B has just switched to using a longer route to D than it last reported to A.

It turns out, however, that we seek the shortest route not because we are particularly trying to save money on transit costs; a route 50% longer would generally work just fine. (AT&T, back when they were the Phone Company, once ran a series of print advertisements claiming longer routes as a *feature*: if the direct path was congested, they could still complete your call by routing you the long way 'round.) However, we *are* guaranteed that if all routers seek the shortest route – and if the network is stable – then all paths are loop-free, because in this case the network will have accurate costs.

Here is a simple example illustrating the importance of global cost-minimization in preventing loops. Suppose we have a network like this one:

Now suppose that A and B use distance-vector but are allowed to choose the shortest route *to within 10%*. A would get a report from C that D could be reached with cost 1, for a total cost of 21. The forwarding entry via C would be $\langle D, C, 21 \rangle$. Similarly, A would get a report from B that D could be reached with cost 21, for a total cost of 22: $\langle D, B, 22 \rangle$. Similarly, B has choices $\langle D, C, 21 \rangle$ and $\langle D, A, 22 \rangle$.

If A and B both choose the minimal route, no loop forms. But if A and B both use the 10%-overage rule, they would be allowed to choose the other route: A could choose $\langle D, B, 22 \rangle$ and B could choose $\langle D, A, 22 \rangle$. If this happened, we would have a routing loop: A would forward packets for D to B, and B would forward them right back to A.

As we apply distance-vector routing, each router independently builds its tables. A router might have some notion of the path its packets would take to their destination; for example, in the case above A might believe that with forwarding entry $\langle D, B, 22 \rangle$ its packets would take the path A-B-C-D (though in distance-vector routing, routers do not particularly worry about the big picture). Consider again the accurate-cost question above. This *fails* in the 10%-overage example, because the actual path is now infinite.

We now prove that, in distance-vector routing, the network will have accurate costs, provided

- each router selects what it believes to be the shortest path to the final destination, and
- the network is stable, meaning that further dissemination of any reports would not result in changes

To see this, suppose the actual route taken by some packet from source to destination, as determined by application of the distributed distance–vector algorithm, is longer than the cost calculated by the source. Choose an example of such a path with the **fewest number of links**, among all such paths in the network. Let S be the source, D the destination, and k the number of links in the actual path P . Let S 's forwarding entry for D be $\langle D, N, c \rangle$, where N is S 's next_hop neighbor.

To have obtained this route through the distance–vector algorithm, S must have received report $\langle D, c_D \rangle$ from N , where we also have the cost of the S – N link as c_N and $c = c_D + c_N$. If we follow a packet from N to D , it must take the same path P with the first link deleted; this sub–path has length $k-1$ and so, by our hypothesis that k was the length of the shortest path with non–accurate costs, the cost from N to D is c_D . But this means that the cost along path P , from S to D via N , must be $c_D + c_N = c$, contradicting our selection of P as a path longer than its advertised cost.

There is one final observation to make about route costs: any cost–minimization can occur only within a single routing domain, where full information about all links is available. If a path traverses multiple routing domains, each separate routing domain may calculate the optimum path traversing that domain. But these “local minimums” do not necessarily add up to a globally minimal path length, particularly when one domain calculates the minimum cost from one of its routers only to the other *domain* rather than to a router within that other domain. Here is a simple example. Routers BR1 and BR2 are the **border routers** connecting the domain LD to the left of the vertical dotted line with domain RD to the right. From A to B, LD will choose the shortest path to RD (not to B, because LD is not likely to have information about links within RD). This is the path of length 3 through BR2. But this leads to a total path length of $3+8=11$ from A to B; the global minimum path length, however, is $4+1=5$, through BR1.

In this example, domains LD and RD join at two points. For a route across two domains joined at only a single point, the domain–local shortest paths do add up to the globally shortest path.

13.4 Loop–Free Distance Vector Algorithms

It is possible for routing–update algorithms based on the distance–vector idea to eliminate routing loops – and thus the slow–convergence problem – entirely. We present brief descriptions of two such algorithms.

13.4.1 DSDV

DSDV, or **Destination–Sequenced Distance Vector**, was proposed in [PB94]. It avoids routing loops by the introduction of **sequence numbers**: each router will always prefer

routes with the most recent sequence number, and bad-news information will always have a lower sequence number than the next cycle of corrected information.

DSDV was originally proposed for MANETs ([4.2.8 MANETs](#)) and has some additional features for traffic minimization that, for simplicity, we ignore here. It is perhaps best suited for wired networks and for small, relatively stable MANETs.

DSDV forwarding tables contain entries for every other reachable node in the system. One successor of DSDV, Ad Hoc On-Demand Distance Vector routing or AODV, [13.4.2 AODV](#), allows forwarding tables to contain only those destinations in active use; a mechanism is provided for discovery of routes to newly active destinations.

Under DSDV, each forwarding table entry contains, in addition to the destination, cost and next_hop, the current sequence number for that destination. When neighboring nodes exchange their distance-vector reachability reports, the reports include these per-destination sequence numbers.

When a router R receives a report from neighbor N for destination D, and the report contains a sequence number larger than the sequence number for D currently in R's forwarding table, then R always updates to use the new information. The three cost-minimization rules of [13.1.1 Distance-Vector Update Rules](#) above are used only when the incoming and existing sequence numbers are equal.

Each time a router R sends a report to its neighbors, it includes a new value for its *own* sequence number, which it always increments by 2. This number is then entered into each neighbor's forwarding-table entry for R, and is then propagated throughout the network via continuing report exchanges. Any sequence number originating this way will be even, and whenever another node's forwarding-table sequence number for R is even, then its cost for R will be finite.

Infinite-cost reports are generated in the usual way when former neighbors discover they can no longer reach one another; however, in this case each node increments the sequence number for its former neighbor by 1, thus generating an odd value. Any forwarding-table entry with infinite cost will thus always have an odd sequence number. If A and B are neighbors, and A's current sequence number is s , and the A-B link breaks, then B will start reporting A at cost ∞ with sequence number $s+1$ while A will start reporting its own new sequence number $s+2$. Any other node now receiving a report originating with B (with sequence number $s+1$) will mark A as having cost ∞ , but will obtain a valid route to A upon receiving a report originating from A with new (and larger) sequence number $s+2$.

The triggered-update mechanism is used: if a node receives a report with some destinations newly marked with infinite cost, it will in turn forward this information immediately to its other neighbors, and so on. This is, however, not essential; "bad" and "good" reports are distinguished by sequence number, not by relative arrival time.

It is now straightforward to verify that the slow-convergence problem is solved. After a link break, if there is some alternative path from router R to destination D, then R will eventually receive D's latest even sequence number, which will be greater than any sequence number associated with any report listing D as unreachable. If, on the other hand, the break partitioned the network and there is no longer any path to D from R, then the highest sequence number circulating in R's half of the original network will be odd and the associated table entries will all list D at cost ∞ . One way or another, the network will quickly settle down to a state where every destination's reachability is accurately described.

In fact, a stronger statement is true: not even transient routing loops are created. We outline a proof. First, whenever router R has next_hop N for a destination D, then N's sequence number for D must be greater than or equal to R's, as R must have obtained its current route to D from one of N's reports. A consequence is that all routers participating in a loop for destination D must have the same (even) sequence number s for D throughout. This means that the loop would have been created if only the reports with sequence number s were circulating. As we noted in [13.1.1 Distance-Vector Update Rules](#), any application of the next_hop-increase rule must trace back to a broken link, and thus must involve an odd sequence number. Thus, the loop must have formed from the sequence-number- s reports by the application of the first two rules only. But this violates the claim in Exercise 10.0.

There is one drawback to DSDV: nodes may sometimes briefly switch to routes that are longer than optimum (though still correct). This is because a router is required to use the route with the newest sequence number, even if that route is longer than the existing route. If A and B are two neighbors of router R, and B is closer to destination D but slower to report, then every time D's sequence number is incremented R will receive A's longer route first, and switch to using it, and B's shorter route shortly thereafter.

DSDV implementations usually address this by having each router R keep track of the time interval between the *first* arrival at R of a new route to a destination D with a given sequence number, and the arrival of the *best* route with that sequence number. During this interval following the arrival of the first report with a new sequence number, R will use the new route, but will refrain from including the route in the reports it sends to its neighbors, anticipating that a better route will soon arrive.

This works best when the hopcount cost metric is being used, because in this case the best route is likely to arrive first (as the news had to travel the fewest hops), and at the very least will arrive soon after the first route. However, if the network's cost metric is unrelated to the hop count, then the time interval between first-route and best-route arrivals can involve multiple update cycles, and can be substantial.

13.4.2 AODV

AODV, or Ad-hoc On-demand Distance Vector routing, is another routing mechanism often proposed for MANETs, though it is suitable for some wired networks as well. Unlike DSDV, above, AODV messages circulate only if a link breaks, or when a node is looking for a route to some other node; this second case is the rationale for the “on-demand” in the name. For larger MANETs, this may result in a significant reduction in routing-management traffic. AODV is described in [\[PR99\]](#) and [RFC 3561](#).

The “ad hoc” in the name was intended to suggest that the protocol is well-suited for mobile nodes forming an ad hoc network ([4.2.4 Access Points](#)). It is, but the protocol is also works well with infrastructure (those with access points) Wi-Fi networks.

AODV has three kinds of messages: *RouteRequest* or RREQ, for nodes that are looking for a path to a destination, *RouteReply* or RREP, as the response, and *RouteError* or RERR for the reporting of broken links.

AODV performs reasonably well for MANETs in which the nodes are highly mobile, though it does assume all routing nodes are trustworthy.

AODV is loop-free, due to the way it uses sequence numbers. However, it does not always find the *shortest* route right away, and may in fact not find the shortest route for an arbitrarily long interval.

Each AODV node maintains a node *sequence number* and also a *broadcast counter*. Every routing message contains a sequence number for the destination, and every routing record kept by a node includes a field for the destination’s sequence number. Copies of a node’s sequence number held by other nodes may not be the most current; however, nodes always discard routes with an older (smaller) sequence number as soon as they hear about a route with a newer sequence number.

AODV nodes also keep track of other nodes that are directly reachable; in the diagram below we will assume these are the nodes connected by a line.

If node A wishes to find a route to node F, as in the diagram below, the first step is for A to increment its sequence number and send out a *RouteRequest*. This message contains the addresses of A and F, A’s just-incremented sequence number, the highest sequence number of any previous route to F that is known to A (if any), a hopcount field set initially to 1, and A’s broadcast counter. The end result should be a route from A to F, entered at each node along the path, and also a return route from F back to A.

The *RouteRequest* is sent initially to A’s direct neighbors, B and C in the diagram above, using UDP. We will assume for the moment that the *RouteRequest* reaches all the way to F before a *RouteReply* is generated. This is always the case if the “destination only” flag is set, though if not then it is possible for an intermediate node to generate the *RouteReply*.

A node that receives a *RouteRequest* must flood it (“broadcast” it) out all its interfaces to all its directly reachable neighbors, after incrementing the hopcount field. B therefore sends A’s message to C and D, and C sends it to B and E. For this example, we will assume that C is a bit slow sending the message to E.

Each node receiving a *RouteRequest* must hang on to it for a short interval (typically 3 seconds). During this period, if it sees a duplicate of the *RouteRequest*, identified by having the same source and the same broadcast counter, it discards it. This discard rule ensures that *RouteRequest* messages do not circulate endlessly around loops; it may be compared to the reliable-flooding algorithm in [13.5 Link-State Routing-Update Algorithm](#).

A node receiving a new *RouteRequest* also records (or updates) a routing-table entry for reaching the source of the *RouteRequest*. Unless there was a pre-existing newer route (that is, with larger sequence number), the entry is marked with the sequence number contained in the message, and with next_hop the neighbor from which the *RouteRequest* was received. This process ensures that, as part of each node’s processing of a *RouteRequest* message, it installs a return route back to the originator.

We will suppose that the following happen in the order indicated:

- B forwards the *RouteRequest* to D*
- D forwards the *RouteRequest* to E and G
- C forwards the *RouteRequest* to E
- E forwards the *RouteRequest* to F

Because E receives D’s copy of the *RouteRequest* first, it ignores C’s copy. This will mean that, at least initially, the return path will be longer than necessary. Variants of AODV (such as HWMP below) sometimes allow E to accept C’s message on the grounds that C has a shorter path back to A. This does mean that initial *RouteRequest* messages farther on in the network now have incorrect hopcount values, though these will be corrected by later *RouteRequest* messages.

After the above messages have been received, each node has a path back to A as indicated by the blue arrows below:

F now increments its own sequence number and creates a *RouteReply* message; F then sends it to A by following the highlighted (unicast) arrows above, $F \rightarrow E \rightarrow D \rightarrow B \rightarrow A$. As each node on the path processes the message, it creates (or updates) its route to the final destination, F; the return route to A had been created earlier when the node processed the corresponding *RouteRequest*.

At this point, A and F can communicate bidirectionally. (Each *RouteRequest* is acknowledged to ensure bidirectionality of each individual link.)

This $F \rightarrow E \rightarrow D \rightarrow B \rightarrow A$ is longer than necessary; a shorter path is $F \rightarrow E \rightarrow C \rightarrow A$. The shorter path will be adopted if, at some future point, E learns that $E \rightarrow C \rightarrow A$ is a better path, though there is no mechanism to seek out this route.

If the “destination only” flag were not set, any intermediate node reached by the *RouteRequest* flooding could have answered with a route to F, if it had one. Such a node would generate the *RouteReply* on its own, without involving F. The sequence number of the intermediate node’s route to F must be greater than the sequence number in the *RouteRequest* message.

If two neighboring nodes can no longer reach one another, each sends out a *RouteError* message, to invalidate the route. Nodes keep track of what routes pass through them, for just this purpose. One node’s message will reach the source and the other’s the destination, at which point the route is invalidated.

In larger networks, it is standard for the originator of a *RouteRequest* to set the IPv4 header TTL value (or the IPv6 Hop_Limit) to a smallish value ([RFC 3561](#) recommends an initial value of 1) to limit the scope of the *RequestRoute* messages. If no answer is received, the originator tries again, with a slightly larger TTL value. In a large network, this reduces the volume of *RouteRequest* messages that have gone too far and therefore cannot be of use in finding a route.

AODV cannot form even short-term loops. To show this, we start with the observation that whenever a $\langle \text{destination}, \text{next_hop} \rangle$ forwarding entry installed at a node, due either to a *RouteRequest* or to a *RouteReply*, the next_hop is always the node from which the *RouteRequest* or *RouteReply* was received, and therefore the destination sequence number cannot get smaller as we move from the original node to its next_hop. That is, as we follow any route to a destination, the destination sequence numbers are nondecreasing. It immediately follows that, for a routing loop, the destination sequence number is constant along the loop. This means that each node on the route must have heard of the route via the same *RouteRequest* or *RouteReply* message, as forwarded.

The second observation, completing the argument, is that the hopcount field must strictly decrease as we travel along the route to the destination; the processing rules for *RouteRequests* and *RouteReplies* mean that each node installs a hopcount of one more than that of the neighboring node from which the route was received. This is impossible for a route that returns to the same node.

13.4.3 HWMP

The Hybrid Wireless Mesh Protocol is based on AODV, and has been chosen for the IEEE 802.11s Wi-Fi mesh networking standard ([4.2.4.4 Mesh Networks](#)). In the discussion here, we will assume HWMP is being used in a Wi-Fi network, though the protocol applies to any type of network. A set of nodes is designated as the routing (or forwarding) nodes; ordinary Wi-Fi stations may or may not be included here.

HWMP replaces the hopcount metric used in AODV with an “airtime link metric” which decreases as the link throughput increases and as the link error rate decreases. This encourages the use of higher-quality wireless links.

HWMP has two route-generating modes: an **on-demand mode** very similar to AODV, and a **proactive mode** used when there is at least one identified “root” node that connects to the Internet. In this case, the route-generating protocol determines a loop-free subset of the relevant routing links (that is, a spanning tree) by which each routing node can reach the root (or one of the roots). This tree-building process does not attempt to find best paths between pairs of non-root nodes, though such nodes can use the on-demand mode as necessary.

In the first, on-demand, mode, HWMP implements a change to classic AODV in that if a node receives a *RouteRequest* message and then later receives a second *RouteRequest* message with the same sequence number but a lower-cost route, then the second route replaces the first.

In the proactive mode, the designated root node – typically the node with wired Internet access – periodically sends out specially marked *RouteRequest* messages. These are sent to the broadcast address, rather than to any specific destination, but otherwise propagate in the usual way. Routing nodes receiving two copies from two different neighbors pick the one with the shortest path. Once this process stabilizes, each routing node knows the best path to the root (or to a root); the fact that each routing node chooses the best path from among all *RouteRequest* messages received ensures eventual route optimality. Routing nodes that have traffic to send can at any time generate a *RouteReply*, which will immediately set up a reverse route from the root to the node in question. Finally, reversing each link to the root allows the root to send broadcast messages.

HWMP has yet another mode: the root nodes can send out *RootAnnounce* (RANN) messages. These let other routing nodes know what the root is, but are not meant to result in the creation of routes to the root.

13.4.4 EIGRP

EIGRP, or the **Enhanced Interior Gateway Routing Protocol**, is a once-proprietary Cisco distance-vector protocol that was released as an Internet Draft in February 2013. As with DSDV, it eliminates the risk of routing loops, even ephemeral ones. It is based on the “distributed update algorithm” (DUAL) of [IG93]. EIGRP is an actual protocol; we present here only the general algorithm. Our discussion follows [CH99].

Each router R keeps a list of neighbor routers N_R , as with any distance-vector algorithm. Each R also maintains a data structure known (somewhat misleadingly) as its **topology table**. It contains, for each destination D and each N in N_R , an indication of whether N has reported the ability to reach D and, if so, the reported cost $c(D,N)$. The router also keeps,

for each N in N_R , the cost c_N of the link from R to N . Finally, the forwarding-table entry for any destination can be marked “passive”, meaning safe to use, or “active”, meaning updates are in process and the route is temporarily unavailable.

Initially, we expect that for each router R and each destination D , R 's next_hop to D in its forwarding table is the neighbor N for which the following total cost is a minimum:

$$c(D,N) + c_N$$

Now suppose R receives a distance-vector report from neighbor N_1 that it can reach D with cost $c(D,N_1)$. This is processed in the usual distance-vector way, unless it represents an increased cost and N_1 is R 's next_hop to D ; this is the third case in [13.1.1 Distance-Vector Update Rules](#). In this case, let C be R 's current cost to D , and let us say that neighbor N of R is a **feasible** next_hop (feasible successor in Cisco's terminology) if N 's cost to D (that is, $c(D,N)$) is strictly less than C . R then updates its route to D to use the feasible neighbor N for which $c(D,N) + c_N$ is a minimum. Note that this may not in fact be the shortest path; it is possible that there is another neighbor M for which $c(D,M) + c_M$ is smaller, but $c(D,M) \geq C$. However, because N 's path to D is loop-free, and because $c(D,N) < C$, this new path through N must also be loop-free; this is sometimes summarized by the statement “one cannot create a loop by adopting a shorter route”.

If no neighbor N of R is feasible – which would be the case in the D — A — B example of [13.2 Distance-Vector Slow-Convergence Problem](#), then R invokes the “DUAL” algorithm. This is sometimes called a “diffusion” algorithm as it invokes a diffusion-like spread of table changes proceeding away from R .

Let C in this case denote the new cost from R to D as based on N_1 's report. R marks destination D as “active” (which suppresses forwarding to D) and sends a special query to each of its neighbors, in the form of a distance-vector report indicating that its cost to D has now increased to C . The algorithm terminates when all R 's neighbors reply back with their own distance-vector reports; at that point R marks its entry for D as “passive” again.

Some neighbors may be able to process R 's report without further diffusion to other nodes, remain “passive”, and reply back to R immediately. However, other neighbors may, like R , now become “active” and continue the DUAL algorithm. In the process, R may receive other queries that elicit its distance-vector report; as long as R is “active” it will report its cost to D as C . We omit the argument that this process – and thus the network – must eventually converge.

13.5 Link-State Routing-Update Algorithm

Link-state routing is an alternative to distance-vector. It is often – though certainly not always – considered to be the routing-update algorithm class of choice for networks that are “sufficiently large”, such as those of ISPs. There are two specific link-state protocols: the IETF's Open Shortest Path First (**OSPF**, [RFC 2328](#)), and OSI's Intermediate Systems to Intermediate Systems (**IS-IS**, documented unofficially in [RFC 1142](#)).

In distance-vector routing, each node knows a bare minimum of network topology: it knows nothing about links beyond those to its immediate neighbors. In the link-state approach, each node keeps a *maximum* amount of network information: a full map of all nodes and all links. Routes are then computed locally from this map, using the shortest-path-first algorithm. The existence of this map allows, in theory, the calculation of different routes for different quality-of-service requirements. The map also allows calculation of a new route as soon as news of the failure of the existing route arrives; distance-vector protocols on the other hand must wait for news of a new route after an existing route fails.

Link-state protocols distribute network map information through a modified form of broadcast of the status of each individual link. Whenever either side of a link notices the link has died (or if a node notices that a new link has become available), it sends out **link-state packets** (LSPs) that “flood” the network. This broadcast process is called **reliable flooding**. In general, broadcast mechanisms are not compatible with networks that have topological looping (that is, redundant paths); broadcast packets may circulate around the loop endlessly. Link-state protocols must be carefully designed to ensure that both every router sees every LSP, and also that no LSPs circulate repeatedly. (The acronym LSP is used by IS-IS; the preferred acronym used by OSPF is LSA, where A is for advertisement.) LSPs are sent immediately upon link-state changes, like triggered updates in distance-vector protocols except there is no “race” between “bad news” and “good news”.

It is possible for ephemeral routing loops to exist; for example, if one router has received a LSP but another has not, they may have an inconsistent view of the network and thus route to one another. However, as soon as the LSP has reached all routers involved, the loop should vanish. There are no “race conditions”, as with distance-vector routing, that can lead to persistent routing loops.

The link-state flooding algorithm avoids the usual problems of broadcast in the presence of loops by having each node keep a database of all LSP messages. The originator of each LSP includes its identity, information about the link that has changed status, and also a **sequence number**. Other routers need only keep in their databases the LSP packet with the largest sequence number; older LSPs can be discarded. When a router receives a LSP, it first checks its database to see if that LSP is old, or is current but has been received before; in these cases, no further action is taken. If, however, an LSP arrives with a sequence number not seen before, then in typical broadcast fashion the LSP is retransmitted over all links except the arrival interface.

As an example, consider the following arrangement of routers:

Suppose the A-E link status changes. A sends LSPs to C and B. Both these will forward the LSPs to D; suppose B's arrives first. Then D will forward the LSP to C; the LSP

traveling $C \rightarrow D$ and the LSP traveling $D \rightarrow C$ might even cross on the wire. D will ignore the second LSP copy that it receives from C and C will ignore the second copy it receives from D.

It is important that LSP sequence numbers not wrap around. (Protocols that *do* allow a numeric field to wrap around usually have a clear-cut idea of the “active range” that can be used to conclude that the numbering has wrapped rather than restarted; this is harder to do in the link-state context.) OSPF uses **lollipop sequence-numbering** here: sequence numbers begin at -2^{31} and increment to $2^{31}-1$. At this point they wrap around back to 0. Thus, as long as a sequence number is less than zero, it is guaranteed unique; at the same time, routing will not cease if more than 2^{31} updates are needed. Other link-state implementations use 64-bit sequence numbers.

Actual link-state implementations often give link-state records a maximum lifetime; entries *must* be periodically renewed.

13.5.1 Shortest-Path-First Algorithm

The next step is to compute routes from the network map, using the shortest-path-first (SPF) algorithm. This algorithm computes shortest paths from a given node, A in the example here, to all other nodes. Below is our example network; we are interested in the shortest paths from A to B, C and D.

Before starting the algorithm, we note the shortest path from A to D is A-B-C-D, which has cost $3+4+2=9$.

The algorithm builds the set **R** of all shortest-path routes iteratively. Initially, **R** contains only the 0-length route to the start node; one new destination and route is added to **R** at each stage of the iteration. At each stage we have a **current** node, representing the node most recently added to **R**. The initial **current** node is our starting node, in this case, A.

We will also maintain a set **T**, for tentative, of routes to other destinations. This is also initialized to empty.

At each stage, we find all nodes which are immediate neighbors of the **current** node and which do not already have routes in the set **R**. For each such node N, we calculate the cost of the route from the start node to N that goes through the **current** node. We see if this is our first route to N, or if the route improves on any route to N already in **T**; if so, we add or update the route in **T** accordingly. Doing this, the routes will be discovered in order of increasing (or nondecreasing) cost.

At the end of this process, we choose the shortest path in **T**, and move the route and destination node to **R**. The destination node of this shortest path becomes the

next **current** node. Ties can be resolved arbitrarily, but note that, as with distance-vector routing, we must choose the minimum or else the accurate-costs property will fail.

We repeat this process until all nodes have routes in the set **R**.

For the example above, we start with **current** = A and **R** = {⟨A,A,0⟩}. The set **T** will be {⟨B,B,3⟩, ⟨C,C,10⟩, ⟨D,D,11⟩}. The lowest-cost entry is ⟨B,B,3⟩, so we move that to **R** and continue with **current** = B. No path through C or D can possibly have lower cost.

For the next stage, the neighbors of B without routes in **R** are C and D; the routes from A to these through B are ⟨C,B,7⟩ and ⟨D,B,12⟩. The former is an improvement on the existing **T** entry ⟨C,C,10⟩ and so replaces it; the latter is not an improvement over ⟨D,D,11⟩. **T** is now {⟨C,B,7⟩, ⟨D,D,11⟩}. The lowest-cost route in **T** is that to C, so we move this node and route to **R** and set C to be **current**.

Again, ⟨C,B,7⟩ must be the shortest path to C. If any lower-cost path to C existed, then we would be selecting that shorter path – or a prefix of it – at this point, instead of the ⟨C,B,7⟩ path; see the proof below.

For the next stage, D is the only non-**R** neighbor; the path from A to D via C has entry ⟨D,B,9⟩, an improvement over the existing ⟨D,D,11⟩ in **T**. The only entry in **T** is now ⟨D,B,9⟩; this has the lowest cost and thus we move it to **R**.

We now have routes in **R** to all nodes, and are done.

Here is another example, again with links labeled with costs:

We start with **current** = A. At the end of the first stage, ⟨B,B,3⟩ is moved into **R**, **T** is {⟨D,D,8⟩}, and **current** is B. The second stage adds ⟨C,B,5⟩ to **T**, and then moves this to **R**; **current** then becomes C. The third stage introduces the route (from A) ⟨D,B,6⟩; this is an improvement over ⟨D,D,8⟩ and so replaces it in **T**; at the end of the stage this route to D is moved to **R**.

In both the examples above, the **current** nodes progressed along a path, A→B→C→D. This is not generally the case; here is a similar example but with different lengths in which **current** jumps from B to D:

As in the previous example, at the end of the first stage ⟨B,B,3⟩ is moved into **R**, with **T** = {⟨D,D,4⟩}, and B becomes **current**. The second stage adds ⟨C,B,6⟩ to **T**. However, the shortest path in **T** is now ⟨D,D,4⟩, and so it is D that becomes the next **current**. The final stage replaces ⟨C,B,6⟩ in **T** with ⟨C,D,5⟩. At that point this route is added to **R** and the algorithm is completed.

Proof that SPF paths are shortest: suppose, by contradiction, that, for some node, a shorter path exists than the one generated by SPF. Let A be the start node, and let U be the *first* node generated for which the SPF path is not shortest. Let T be the Tentative set and let R be the set of completed routes at the point when we choose U as **current**, and let d be the cost of the new route to U . Let $P = \langle A, \dots, X, Y, \dots, U \rangle$ be the shorter path to U , with cost $c < d$, where Y is the first node along the path *not* to have a route in R (it is possible $Y=U$).

At some strictly earlier stage in the algorithm, we must have added a route to node X , as the route to X is in R . In the following stage, we would have included the prefix $\langle A, \dots, X, Y \rangle$ of P in Tentative. This path to Y has cost $\leq c$. This route must still be in T at the point we chose U as **current**, as there is no route to Y in R , but this means we should instead have chosen Y as **current**, contradicting the choice of U .

A link-state source node S computes the entire path to a destination D (in fact it computes the path to every destination). But as far as the actual path that a packet sent by S will take to D , S has direct control only as far as the first hop N . While the accurate-cost rule we considered in distance-vector routing will still hold, the actual path taken by the packet may differ from the path computed at the source, in the presence of alternative paths of the same length. For example, S may calculate a path $S-N-A-D$, and yet a packet may take path $S-N-B-D$, so long as the $N-A-D$ and $N-B-D$ paths have the same length.

Link-state routing allows calculation of routes on demand (results are then cached), or larger-scale calculation. Link-state also allows routes calculated with quality-of-service taken into account, via straightforward extension of the algorithm above.

Because the starting node is fixed, the shortest-path-first algorithm can be classified as a **single-source** approach. If the goal is to compute the shortest paths between all pairs of nodes in a network, the [Floyd-Warshall algorithm](#) is an alternative, with slightly better performance in networks with large numbers of links.

13.6 Routing on Other Attributes

There is sometimes a desire to route on packet attributes other than the destination, or the destination and QoS bits. For example, we might want to route packets based in part on the packet source, or on the TCP port number. This kind of routing is decidedly nonstandard, though it is often available, and often an important component of traffic engineering.

This option is often known as **policy-based routing**, because packets are routed according to attributes specified by local administrative policy. (This term should not be

confused with BGP *routing policy* ([1.5 Border Gateway Protocol \(BGP\)](#)), which means something quite different.)

Policy-based routing is not used frequently, but one routing decision of this type can have far-reaching effects. If an ISP wishes to route customer voice traffic differently from customer data traffic, for example, it need only apply policy-based routing to classify traffic at the point of entry, and send the voice traffic to its own router. After that, ordinary routers on the voice path and on the separate data path can continue the forwarding without using policy-based methods.

Sometimes policy-based routing is used to *mark* packets for special processing; this might mean different routing further downstream or it might mean being sent along the same path as the other traffic but with preferential treatment. For two packet-marking strategies, see [25.7 Differentiated Services](#) and [25.12 Multi-Protocol Label Switching \(MPLS\)](#).

On Linux systems policy-based routing is part of the Linux Advanced Routing facility, often grouped with some advanced queuing features known as Traffic Control; the combination is referred to as [LARTC](#).

As a simple example of what can be done, suppose a site has two links L1 and L2 to the Internet, with L1 the default route to the Internet. Perhaps L1 is faster and L2 serves more as a backup; perhaps L2 has been added to increase outbound capacity. A site may wish to route some outbound traffic via L2 for any of the following reasons:

- the traffic may involve protocols deemed lower in priority (*eg* email)
- the traffic may be real-time traffic that can benefit from reduced competition on L2
- the traffic may come from lower-priority senders; *eg* some customers within the site may be relegated to using L2 because they are paying less
- a few large-volume [elephant flows](#) may be offloaded from L1 to L2

In the first two cases, routing might be based on the destination port numbers; in the third, it might be based on the source IP address. In the fourth case, a site's classification of its elephant flows may have accumulated over time.

Note that nothing can be done in the *inbound* direction unless L1 and L2 lead to the same ISP, and even there any special routing would be at the discretion of that ISP.

The trick with LARTC is to be compatible with existing routing-update protocols; this would be a problem if the kernel forwarding table simply added columns for other packet attributes that neighboring non-LARTC routers knew nothing about. Instead, the forwarding table is split up into multiple (dest, next_hop) (or (dest, QoS, next_hop)) tables. One of these tables is the **main** table, and is the table that is updated by routing-update protocols interacting with neighbors. Before a packet is forwarded, administratively supplied rules are consulted to determine which table to apply; these

rules *are* allowed to consult other packet attributes. The collection of tables and rules is known as the **routing policy database**.

As a simple example, in the situation above the main table would have an entry (default, L1) (more precisely, it would have the IP address of the far end of the L1 link instead of L1 itself). There would also be another table, perhaps named **slow**, with a single entry (default, L2). If a rule is created to have a packet routed using the “slow” table, then that packet will be forwarded via L2. Here is one such Linux rule, which causes all traffic from host 10.0.0.17 to be routed using table “slow”:

```
ip rule add from 10.0.0.17 table slow
```

Now suppose we want to route traffic to port 25 (the SMTP port) via L2. This is harder; Linux provides no support here for routing based on port numbers. However, we can instead use the [iptables](#) mechanism to “mark” all packets destined for port 25, and then create a routing-policy rule to have such marked traffic use the slow table. The mark is known as the forwarding mark, or fwmark; its value is 0 by default. The fwmark is not actually part of the packet; it is associated with the packet only while the latter remains within the kernel.

```
iptables --table mangle --append PREROUTING \\  
        --protocol tcp --dport 25 --jump MARK --set-mark 1  
  
ip rule add fwmark 1 table slow
```

Consult the applicable man pages for further details.

The iptables mechanism can also be used to set the appropriate QoS bits – the IPv4 DS bits ([9.1 The IPv4 Header](#)) or the IPv6 Traffic Class bits ([11.1 The IPv6 Header](#)) – so that a single standard IP forwarding table can be used, though support for the IPv4 QoS bits is limited.

Linux actually maintains a numbered set of up to 255 tables; the **main** table is number 254. These associations between numbers and table names are stored in /etc/iproute2/rt_tables. If we want a table named “slow” then we have to give it a number in this file, perhaps with an entry

```
100  slow
```

(Alternatively, it is possible to replace the name “slow” in the earlier commands with the number.)

13.6.1 Reverse-Path Filtering

There is one other issue we must deal with to build a working example from the fragments above. When a packet arrives at a router, a feature called **reverse-path filtering** will prevent forwarding the packet if the router does not have a forwarding entry

that would forward the *source* address back out the interface by which the packet just arrived; that is, if the router knows it won't be able to route reply packets properly, it won't forward the original packet. This is very commonly implemented, and is meant to prevent spoofing. If an ISP, for example, has IP address block 200.1.0.0/16, then reverse-path filtering will prevent miscreant ISP customers from sending out packets with "spoofed" source addresses not in this block. If every ISP implemented this feature, certain malicious attacks such as SYN flooding ([17.3 TCP Connection Establishment](#)) would be much harder.

But for the router above with the two paths L1 and L2, reverse-path filtering must be disabled. Suppose this router is R, and let D be the destination that the "slow" path is being set up to reach. R will have a "normal" (perhaps default) route to reach D via L1. The point of policy-based routing is so packets to D will be routed using the other, "slow" path, over L2. But with reverse-path filtering in place the router will detect that *return* packets from D, *arriving* via L2, have a source address that does not get forwarded *back* over L2, because the return packets don't match the routing-policy rule. So, unless reverse-path-filtering is disabled, R will drop these return packets. Packets *to* D will be delivered, but all D's replies will be lost.

On Linux systems, the files in the directory `/proc/sys/net/ipv4/conf` determine the status of reverse-path filtering. This directory contains a subdirectory for each interface, plus `default`. Each subdirectory contains, among other things, a file `rp_filter`; if this file contains a 0, reverse-path filtering is off, while a value of 1 enables it. A value of 2 enables "loose reverse-path filtering", in that a packet is forwarded if there is *some* interface – not necessarily the arrival interface – by which the router can reach the source address.

13.7 ECMP

Equal-Cost MultiPath routing, or ECMP, is a technique for combining two (or more) routes to a destination into a single unit, so that traffic to that destination is distributed (not necessarily equally) among the routes. ECMP is supported by EIGRP ([13.4.4 EIGRP](#)) and the link-state implementations OSPF and IS-IS ([13.5 Link-State Routing-Update Algorithm](#)). At the Ethernet level, ECMP is supported in spirit (if not in name) by TRILL and SPB ([3.3 TRILL and SPB](#)). It is also supported by BGP ([15 Border Gateway Protocol \(BGP\)](#)) for inter-AS routing.

A simpler alternative to ECMP is **channel bonding**, also known as **link aggregation**, and often based on the [IEEE 802.3ad](#) standard. In channel bonding, two parallel Ethernet links are treated as a single unit. In many cases it is simpler and cheaper to bond two or three 1 Gbps Ethernet links than to upgrade everything to support 10 Gbps. Channel bonding applies, however, in limited circumstances; for example, the two channels must both be Ethernet, and must represent a single link.

In the absence of channel bonding, equal-cost does not necessarily mean equal-propagation-delay. Even for two short parallel links, queuing delays on one link may mean that packet delivery order is not preserved. As TCP usually interprets out-of-order packet delivery as evidence of packet loss ([19.3 TCP Tahoe and Fast Retransmit](#)), this can lead to large numbers of spurious retransmissions. For this reason, ECMP is almost always configured to send all the packets of any one TCP connection over just one of the links (as determined by a hash function); some channel-bonding implementations do the same, in fact. A consequence is that ECMP configured this way must see a large number of parallel TCP connections in order to utilize all participating paths reasonably equally. In special cases, however, it may be practical to configure ECMP to alternate between the paths on a per-packet basis, using round-robin transmission; this approach has the potential to achieve much better load-balancing between the paths.

In terms of routing-update protocols, ECMP can be viewed as allowing two (or more) next_hop values, each with the same cost, to be associated with the same destination.

See [30.9.4 multitrunk.py](#) for an example of the use of software-defined networking to have multiple TCP connections take different paths to the same destination, in a way similar to the ECMP approach.

13.8 Epilog

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

At the very largest scales, these requirements are impractical. The next chapter is devoted to this issue of very-large-scale IP routing, *eg* on the global Internet.

13.9 Exercises

Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercises marked with a \diamond have solutions or hints at [34.10 Solutions for Routing-Update Algorithms](#).

1.0. Suppose the network is as follows, where distance-vector routing update is used. Each link has cost 1, and each router initially has entries in its forwarding table only for its immediate neighbors (so A's table contains $\langle B, B, 1 \rangle$, $\langle D, D, 1 \rangle$ and B's table contains $\langle A, A, 1 \rangle$, $\langle C, C, 1 \rangle$).

(a). Suppose each router creates a report from its initial configuration and sends that to each of its neighbors. What will each router's forwarding table be after this set of exchanges? The exchanges, in other words, are all conducted simultaneously; each

router first sends out its own report and then processes the reports arriving from its two neighbors.

(b). What destinations, with corresponding cost values, will be added to each router's table after the simultaneous-and-parallel exchange process of part (a) is repeated a second time? Ignore the next_hop values.

Hint: for each router, this step will add one additional destination entry. This new destination entry can be determined without going through the table exchanges in detail.

2.0. Now suppose the configuration of routers has the link weights shown below. Each router again initially has entries in its forwarding table only for its immediate neighbors.

(a). As in the previous exercise, give each router's forwarding table after each router exchanges with its immediate neighbors simultaneously and in parallel.

(b). How many iterations of such parallel exchanges will it take before C learns to reach F via B; that is, before it creates the entry (F,B,11)? Count the answer to part (a) as the first iteration.

3.0. ♦ A router R has the following distance-vector table:

destination	cost	next hop
A	2	R1
B	3	R2
C	4	R1
D	5	R3

R now receives the following report from R1; the cost of the R-R1 link is 1.

destination	cost
A	1
B	2
C	4
D	3

Give R's updated table after it processes R1's report. For each entry that changes, give a brief explanation

4.0. A router R has the following distance-vector table:

destination	cost	next hop
A	5	R1
B	6	R1
C	7	R2
D	8	R2
E	9	R3

R now receives the following report from R1; the cost of the R-R1 link is 1.

destination	cost
A	4
B	7
C	7
D	6
E	8
F	8

Give R's updated table after it processes R1's report. For each entry that changes, give a brief explanation, in the style of [13.1.5 Example 4](#).

5.0. At the start of Example 3 ([13.1.4 Example 3](#)), we changed C's routing table so that it reached D via A instead of via E: C's entry $\langle D, E, 2 \rangle$ was changed to $\langle D, A, 2 \rangle$. This meant that C had a valid route to D at the start.

How might the scenario of Example 3 play out if C's table had not been altered? Give a sequence of reports that leads to correct routing between D and E.

6.0. In the following exercise, A–D are routers and the attached subnets N1–N6, which are the ultimate destinations, are shown explicitly. In the case of N1 through N4, the links *are* the subnets. Routers still exchange distance–vector reports with neighboring routers, as usual. In the tables requested below, if a router has a direct connection to a subnet, you may report the next_hop as “direct”, *eg*, from A's table, $\langle N1, \text{direct}, 0 \rangle$

- Give the initial tables for A through D, before any distance–vector exchanges.
- Give the tables after each router A–D exchanges with its immediate neighbors simultaneously and in parallel.
- At the end of (b), what subnets are *not* known by what routers?

7.0. Suppose A, B, C, D and E are connected as follows. Each link has cost 1, and so each forwarding table is uniquely determined; B's table is $\langle A, A, 1 \rangle$, $\langle C, C, 1 \rangle$, $\langle D, A, 2 \rangle$, $\langle E, C, 2 \rangle$. Distance–vector routing update is used.

Now suppose the D–E link fails, and so D updates its entry for E to $\langle E, -, \infty \rangle$.

- Give A's table after D reports $\langle E, \infty \rangle$ to A
- Give B's table after A reports to B
- Give A's table after B reports to A; note that B has an entry $\langle E, C, 2 \rangle$
- Give D's table after A reports to D.

8.0. In the network below, A receives alternating reports about destination D from neighbors B and C. Suppose A uses a modified form of Rule 2 of [13.1.1 Distance–Vector Update Rules](#), in which it updates its forwarding table whenever new cost c is less than *or equal to* c_{old} .

Explain why A's forwarding entry for destination D never stabilizes.

9.0. Consider the network in [13.2.1.1 Split Horizon](#), using distance-vector routing updates. B and C's table entries for destination D are shown. All link costs are 1.

Suppose the D-A link breaks and then these update reports occur:

- A reports $\langle D, \infty \rangle$ to B (as before)
- C reports $\langle D, 2 \rangle$ to B (as before)
- A now reports $\langle D, \infty \rangle$ to C (instead of B reporting $\langle D, 3 \rangle$ to A)

(a). Give A, B and C's forwarding-table records for destination D, including the cost, after these three reports.

(b). What additional reports (a pair should suffice) will lead to the formation of the routing loop?

(c). What (single) additional report will eliminate the possibility of the routing loop?

10.0. Suppose the network of [13.2 Distance-Vector Slow-Convergence Problem](#) is changed to the following. Distance-vector update is used; again, the D-A link breaks.

(a). Explain why B's report back to A, after A reports $\langle D, -, \infty \rangle$, is now valid.

(b). Explain why hold down ([13.2.1.3 Hold Down](#)) will delay the use of the new route A-B-E-D.

11.0. Suppose the routers are A, B, C, D, E and F, and all link costs are 1. The distance-vector forwarding tables for A and F are below. Give the network with the fewest links that is consistent with these tables. Hint: any destination reached at cost 1 is directly connected; if X reaches Y via Z at cost 2, then Z and Y must be directly connected.

A's table

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

F's table

dest	cost	next_hop
A	3	E

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

12.0. (a) Suppose routers A and B somehow end up with respective forwarding-table entries $\langle D, B, n \rangle$ and $\langle D, A, m \rangle$, thus creating a routing loop. Explain why the loop may be removed more quickly if A and B both use **poison reverse** with split horizon ([13.2.1.1 Split Horizon](#)), versus if A and B use split horizon only.

(b). Suppose the network looks like the following. The A-B link is extremely slow.

Suppose A and B send reports to each other advertising their routes to D, and immediately afterwards the C-D link breaks and C reports to A and B that D is unreachable. *After* those unreachability reports from C are processed, A and B's reports sent to each other before the break finally arrive. Explain why the network is now in the state described in part (a).

13.0. Suppose the distance-vector algorithm is run on a network and no links break (so by the last paragraph of [13.1.1 Distance-Vector Update Rules](#) the next_hop-increase rule is never applied).

(a). Prove that whenever A is B's next_hop to destination D, then A's cost to D is strictly less than B's. Hint: assume that if this claim is true, then it remains true after any application of the rules in [13.1.1 Distance-Vector Update Rules](#). If the lower-cost rule is applied to B after receiving a report from A, resulting in a change to B's cost to D, then one needs to show A's cost is less than B's, and also B's new cost is less than that of any neighbor C that uses B as its next_hop to D.

(b). Use (a) to prove that no routing loops ever form.

14.0. It was mentioned in [13.5 Link-State Routing-Update Algorithm](#) that link-state routing might give rise to an ephemeral routing loop. Give a concrete scenario illustrating creation (and then dissolution) of such a loop.

Hint: consider the following network.

C's next_hop to A is B, due to the relative link weights (not shown) of the A-B1 and A-C1 links. Then B gets some news. (There are also several other possible scenarios.)

15.0. Use the Shortest-Path-First algorithm to find the shortest path from A to E in the network below. Show the sets **R** and **T**, and the node **current**, after each step.

16.0. Suppose you take a laptop, plug it into an Ethernet LAN, *and* connect to the same LAN via Wi-Fi. From laptop to LAN there are now two routes. Which route will be preferred? How can you tell which way traffic is flowing? How can you configure your OS to prefer one path or another? (See also [10.2.5 ARP and multihomed hosts](#), [9 IP version 4](#) exercise 4.0, and [17 TCP Transport Basics](#) exercise 5.0.)

14 Large-Scale IP Routing

In the previous chapter we considered two classes of routing-update algorithms: distance-vector and link-state. Each of these approaches requires that participating routers have agreed first to a common protocol, and then to a common understanding of how link costs are to be assigned. We will address this in the following chapter on BGP, [15 Border Gateway Protocol \(BGP\)](#); a basic problem is that if one site prefers the hop-count approach, assigning every link a cost of 1, while another site prefers to assign link costs in proportion to their bandwidth, then *meaningful* path cost comparisons between the two sites simply cannot be done.

Before we can get to BGP, however, we need to revisit the basic IP forwarding idea. From the beginning we envisioned an IP address as being divisible into network and host portions; the job of most routers was to examine only the network prefix, and make the next_hop forwarding decision based on that. The net/host division for IPv4 addresses was originally based on the Class A/B/C mechanism; the net/host division for most IPv6 addresses was 64 bits for each. In [9.5 The Classless IP Delivery Algorithm](#) we saw how to route on IPv4 network prefixes of arbitrary length; the same idea works for IPv6.

In this chapter we are going to expand on this by allowing different routers at different positions in the routing universe to make their forwarding decisions based on different network-prefix lengths, all for the same destination address. That is, a backbone router might forward to a given IPv4 address D based only on the first 12 bits of D; a more regional router might base its decision on the first 18 bits, and a site router might forward to the final subnet based on the first 24 bits. This allows the creation of **routing hierarchies** with multiple levels, which has the potential to greatly increase routing scalability and reduce the size of the forwarding tables. The actual mechanics of forwarding by any one router will still be as in [9.5 The Classless IP Delivery Algorithm](#).

The term **routing domain** is used to refer to a set of routers under common administration, using a common link-cost assignment; another term for this is **autonomous system**. In the previous chapter, all routing took place within one routing domain; now we will envision the Internet as a whole consisting of a patchwork of independent routing domains. While use of a common routing-update protocol within the routing domain is not an absolute requirement – for example, some subnets may

internally use distance-vector while the site's "backbone" routers use link-state – we can assume that all routers have a uniform view of the site's topology and cost metrics.

"Large-scale" IP routing is fundamentally about the coordination of routing between multiple independent routing domains. Even in the earliest Internet there were multiple routing domains, if for no other reason than that how to measure link costs was (and still is) too unsettled to set in stone. However, another component of large-scale routing is support for hierarchical routing, above the level of subnets; we turn to this next.

14.1 Classless Internet Domain Routing: CIDR

CIDR is the mechanism for supporting hierarchical routing in the Internet backbone. Subnetting moves the network/host division line further rightwards; CIDR allows moving it to the left as well. With subnetting, the revised division line is visible only within the organization that owns the IP network-address block; subnetting is not visible outside. CIDR allows aggregation of IP address blocks in a way that *is* visible to the Internet backbone.

When CIDR was introduced to IPv4 in 1993, the following were some of the justifications for it, all relating to the increasing size of the backbone IP forwarding tables, and expressed in terms of the then-current Class A/B/C mechanism:

- The Internet is running out of Class B addresses (this happened in the mid-1990's)
- There are too many Class C's (the most numerous) for backbone forwarding tables to be efficient
- Eventually IANA (the Internet Assigned Numbers Authority) will run out of IP addresses (this happened in 2011)

Assigning non-CIDRed multiple Class C's in lieu of a single Class B would have helped with the first point in the list above, but made the second point worse.

Ironically, the current (2013) very tight market for IPv4 address blocks is likely to lead to larger and larger backbone IPv4 forwarding tables, as sites are forced to use multiple small address blocks instead of one large block.

By the year 2000, CIDR had essentially eliminated the Class A/B/C mechanism from the backbone Internet, and had more-or-less completely changed how backbone routing worked. You purchased an address block from a provider or some other IP address allocator, and it could be whatever size you needed, from /32 to /15.

What CIDR enabled is IP routing based on an address prefix of any length; the Class A/B/C mechanism of course used fixed prefix lengths of 8, 16 and 24 bits. Furthermore, CIDR allows different routers, at different levels of the backbone, to route on prefixes of different lengths. If organization P were allocated a /10 block, for example, then P

could *suballocate* into /20 blocks. At the top level, routing to P would likely be based on the first 10 bits, while routing within P would be based on the first 20 bits.

IPv6 never had address classes, and so arguably CIDR was supported natively from the beginning. Routing to an address 2400:1234:5678:abcd:: could be done based on the /32 prefix 2400:1234::, or the /48 prefix 2400:1234:5678::, or the /56 prefix 2400:1234:5678:ab::, or any other length.

CIDR was formally introduced to IPv4 by [RFC 1518](#) and [RFC 1519](#). For a while there were strategies in place to support compatibility with non-CIDR-aware routers; these are now obsolete. In particular, it is no longer appropriate for large-scale IPv4 routers to fall back on the Class A/B/C mechanism in the absence of CIDR information; if the latter is missing, the routing should fail.

One way to look at the basic strategy of CIDR is as a mechanism to consolidate multiple network blocks going to the same destination into a single entry. Suppose a router has four IPv4 class C's all to the same destination:

```
200.7.0.0/24 → foo
200.7.1.0/24 → foo
200.7.2.0/24 → foo
200.7.3.0/24 → foo
```

The router can replace all these with the single entry

```
200.7.0.0/22 → foo
```

It does not matter here if foo represents a single ultimate destination or if it represents four sites that just happen to be routed to the same next_hop.

It is worth looking closely at the arithmetic to see why the single entry uses /22. This means that the first 22 bits must match 200.7.0.0; this is all of the first and second bytes and the first six bits of the third byte. Let us look at the third byte of the network addresses above in binary:

```
200.7.000000 00.0/24 → foo
200.7.000000 01.0/24 → foo
200.7.000000 10.0/24 → foo
200.7.000000 11.0/24 → foo
```

The /24 means that the network addresses stop at the end of the third byte. The four entries above cover every possible combination of the last two bits of the third byte; for an address to match one of the entries above it suffices to begin 200.7 and then to have 0-bits as the first *six bits* of the third byte. This is another way of saying the address must match 200.7.0.0/22.

Most implementations actually use a bitmask, *eg* 255.255.252.0, rather than the number 22. Note 252 is, in binary, 1111 1100, with 6 leading 1-bits, so 255.255.252.0 has $8+8+6=22$ 1-bits followed by 10 0-bits.

The IP delivery algorithm of [9.5 The Classless IP Delivery Algorithm](#) still works with CIDR, with the understanding that the router's forwarding table can now have a network-prefix length associated with *any* entry. Given a destination D, we search the forwarding table for network-prefix destinations B/k until we find a match; that is, equality of the first k bits. In terms of masks, given a destination D and a list of table entries $\langle \text{prefix}, \text{mask} \rangle = \langle B[i], M[i] \rangle$, we search for i such that $(D \& M[i]) = B[i]$.

But what about the possibility of multiple matches? For subnets, avoiding this was the responsibility of the subnetting site, but responsibility for avoiding this with CIDR is much too distributed to be declared illegal by IETF mandate. Instead, CIDR introduced the **longest-match rule**: if destination D matches both B_1/k_1 and B_2/k_2 , with $k_1 < k_2$, then the longer match B_2/k_2 match is to be used. (Note that if D matches two distinct entries B_1/k_1 and B_2/k_2 then either $k_1 < k_2$ or $k_2 < k_1$).

14.2 Hierarchical Routing

Strictly speaking, CIDR is simply a mechanism for routing to IP address blocks of any prefix length; that is, for setting the network/host division point to an arbitrary place within the 32-bit IP address.

However, by making this network/host division point *variable*, CIDR introduced support for routing on *different* prefix lengths at different places in the backbone routing infrastructure. For example, top-level routers might route on /8 or /9 prefixes, while intermediate routers might route based on prefixes of length 14. This feature of routing on fewer bits at one point in the Internet and more bits at another point is exactly what is meant by **hierarchical routing**.

We earlier saw hierarchical routing in the context of subnets: traffic might first be routed to a class-B site 147.126.0.0/16, and then, within that site, to subnets such as 147.126.1.0/24, 147.126.2.0/24, *etc.* But with CIDR the hierarchy can be much more flexible: the top level of the hierarchy can be much larger than the “customer” level, lower levels need not be administratively controlled by the higher levels (as is the case with subnets), and more than two levels can be used.

CIDR is an address-block-allocation *mechanism*; it does not directly speak to the kinds of *policy* we might wish to implement with it. Here are four possible applications; the latter two involve hierarchical routing:

- Application 1 (legacy): CIDR allows the allocation of multiple blocks of Class C, or fragments of a Class A, to a single customer, so as to require only a single forwarding-table entry for that customer

- Application 2 (legacy): CIDR allows opportunistic **aggregation** of routes: a router that sees the four 200.7.x.0/24 routes above in its table may consolidate them into a single entry.
- Application 3 (current): CIDR allows huge provider blocks, with suballocation by the provider. This is known as **provider-based** routing.
- Application 4 (hypothetical): CIDR allows huge regional blocks, with suballocation within the region, somewhat like the original scheme for US phone numbers with area codes. This is known as **geographical** routing.

Each of these has the potential to achieve a considerable reduction in the size of the backbone forwarding tables, which is arguably the most important goal here. Each involves using CIDR to support the creation of arbitrary-sized address blocks and then *routing to them as a single unit*. For example, the Internet backbone might be much happier if all its routers simply had to maintain a single entry $\langle 200.0.0.0/8, R1 \rangle$, versus 256 entries $\langle 200.x.0.0/16, R1 \rangle$ for every value of x . (As we will see below, this is still useful even if a few of the x 's have a different next_hop.) Secondary CIDR goals include bringing some order to IP address allocation and (for the last two items in the list above) enabling a routing hierarchy that mirrors the actual flow of most traffic.

Hierarchical routing does introduce one new wrinkle: the routes chosen may no longer be globally optimal, at least if we also apply the routing-update algorithms hierarchically. Suppose, for example, at the top level forwarding is based on the first eight bits of the address, and all traffic to 200.0.0.0/8 is routed to router R1. At the second level, R1 then routes traffic (hierarchically) to 200.20.0.0/16 via R2. A packet sent to 200.20.1.2 by an independent router R3 might therefore pass through R1, *even if there were a lower-cost path $R3 \rightarrow R4 \rightarrow R2$ that bypassed R1*. The top-level forwarding entry $\langle 200.0.0.0/8, R1 \rangle$, in other words, may represent a simplification of the real situation. Prohibiting “back-door” routes like $R3 \rightarrow R4 \rightarrow R2$ is impractical (and would not be helpful either); customers are independent entities.

This non-optimal routing issue cannot happen if all routers agree upon one of the shortest-path mechanisms of [13 Routing-Update Algorithms](#); in that case R3 would learn of the lower-cost $R3 \rightarrow R4 \rightarrow R2$ path. But then the potential hierarchical benefits of decreasing the size of forwarding tables would be lost. More seriously, complete global agreement of all routers on one common update protocol is simply not practical; in fact, one of the goals of hierarchical routing is to provide a workable alternative. We will return to this below in [14.4.3 Hierarchical Routing via Providers](#).

14.3 Legacy Routing

Back in the days of NSFNet, the Internet backbone was a single routing domain. While most customers did not connect directly to the backbone, the intervening providers tended to be relatively compact, geographically – that is, *regional* – and often had a single primary routing-exchange point with the backbone. IP addresses were allocated to

subscribers directly by the IANA, and the backbone forwarding tables contained entries for every site, even the Class C's.

Because the NSFNet backbone and the regional providers did not necessarily share link-cost information, routes were even at this early point not necessarily globally optimal; compromises and approximations were made. However, in the NSFNet model routers generally did find a reasonable approximation to the shortest path to each site referenced by the backbone tables. While the legacy backbone routing domain was not all-encompassing, if there *were* differences between two routes, at least the backbone portions – the longest components – would be identical.

14.4 Provider-Based Routing

In provider-based routing, large CIDR blocks are allocated to large-scale providers. The different providers each know how to route to one another. While some subscribers will still have legacy IP-address blocks assigned to them directly, many others will obtain their IP addresses from within their providers' blocks. For the latter group, traffic from the outside is routed first to the provider via the provider's address block, and then, *within* the provider's routing domain, to the subscriber. We may even have a hierarchy of providers, so packets would be routed first to the large-scale provider, and eventually to the local provider. There may no longer be a central backbone; instead, multiple providers may each build parallel transcontinental networks.

Here is a simpler example, in which providers have *unique* paths to one another. Suppose we have providers P0, P1 and P2, with customers as follows:

- P0: customers A,B,C
- P1: customers D,E
- P2: customers F,G

We will also assume that each provider has an IP address block as follows:

- P0: 200.0.0.0/8
- P1: 201.0.0.0/8
- P2: 202.0.0.0/8

Let us now allocate addresses to the customers:

A: 200.0.0.0/16
B: 200.1.0.0/16
C: 200.2.16.0/20 (16 = 0001 0000)
D: 201.0.0.0/16
E: 201.1.0.0/16
F: 202.0.0.0/16
G: 202.1.0.0/16

The routing model is that packets are first routed to the appropriate provider, and then to the customer. While this model may not in general guarantee the shortest end-to-end path, it does in this case because each provider has a single point of interconnection to the others. Here is the network diagram:

With this diagram, P0's forwarding table looks something like this:

P0	
destination	next_hop
200.0.0.0/16	A
200.1.0.0/16	B
200.2.16.0/20	C
201.0.0.0/8	P1
202.0.0.0/8	P2

That is, P0's table consists of

- one entry for each of P0's own customers
- one entry for each other provider

If we had 1,000,000 customers divided equally among 100 providers, then each provider's table would have only 10,099 entries: 10,000 for its own customers and 99 for the other providers. Without CIDR, each provider's forwarding table would have 1,000,000 entries.

CIDR enables hierarchical routing by allowing the routing decision to be made on different prefix lengths in different contexts. For example, when a packet is sent from D to A, P1 looks at the first 8 bits while P0 looks at the first 16 bits. Within customer A, routing might be made based on the first 24 bits.

Even if we have some additional "secondary" links, that is, additional links that do not create alternative paths between providers, the routing remains *relatively* straightforward. Shown here are the private customer-to-customer links B-D and E-F; these are likely used only by the customers they connect. Two customers, A and E, are **multihomed**; that is, they have connections to alternative providers: A-P1 and E-P2. (The term "multihomed" is often applied to any host with multiple network interfaces on different LANs, which includes any router; here we mean more specifically that there are multiple network interfaces connecting to different providers.)

Typically, though, while A and E may use their alternative-provider links all they want for *outbound* traffic, their respective *inbound* traffic would still go through their primary providers P0 and P1 respectively.

14.4.1 Internet Exchange Points

The long links joining providers in these diagrams are somewhat misleading; providers do not always like sharing long links and the attendant problems of sharing responsibility for failures. Instead, providers often connect to one another at **Internet eXchange Points** or IXPs; the link P0————P1 might actually be P0——IXP——P1, where P0 owns the left-hand link and P1 the right-hand. IXPs can either be third-party sites open to all providers, or private exchange points. The term “Metropolitan Area Exchange”, or MAE, appears in the names of the IXPs MAE-East, originally near Washington DC, and MAE-West, originally in San Jose, California; each of these is now actually a *set* of IXPs. MAE in this context is now a trademark.

14.4.2 CIDR and Staying Out of Jail

Suppose we want to change providers. One way we can do this is to accept a new IP-address block from the new provider, and change all our IP addresses. The paper *Renumbering: Threat or Menace* [LKCT96] was frequently cited – at least in the early days of CIDR – as an intimation that such renumbering was inevitably a Bad Thing. In principle, therefore, we would like to allow at least the option of *keeping* our IP address allocation while changing providers.

An address-allocation standard that did not allow changing of providers might even be a violation of the US Sherman Antitrust Act; see *American Society of Mechanical Engineers v Hydrolevel Corporation*, 456 US 556 (1982). The IETF thus had the added incentive of wanting to stay out of jail, when writing the CIDR standard so as to allow portability between providers (actually, antitrust violations usually involve civil penalties).

The CIDR **longest-match** rule turns out to be exactly what we (and the IETF) need. Suppose, in the diagrams above, that customer C wants to move from P0 to P1, and does not want to renumber. What routing changes need to be made? One solution is for P0 to add a route (200.2.16.0/20, P1) that routes all of C’s traffic to P1; P1 will then forward that traffic on to C. P1’s table will be as follows, and P1 will use the longest-match rule to distinguish traffic for its new customer C from traffic bound for P0.

P1	
destination	next_hop
200.0.0.0/8	P0
202.0.0.0/8	P2
201.0.0.0/16	D
201.1.0.0/16	E
200.2.16.0/20	C

This does work, but all C's inbound traffic except for that originating in P1 will now be routed through C's ex-provider P0, which as an *ex-provider* may not be on the best of terms with C. Also, the routing is inefficient: C's traffic from P2 is routed $P2 \rightarrow P0 \rightarrow P1$ instead of the more direct $P2 \rightarrow P1$.

A better solution is for *all* providers other than P1 to add the route (200.2.16.0/20, P1). While traffic to 200.0.0.0/8 otherwise goes to P0, this particular sub-block is instead routed by each provider to P1. The important case here is P2, as a stand-in for all other providers and their routers: P2 routes 200.0.0.0/8 traffic to P0 *except* for the block 200.2.16.0/20, which goes to P1.

Having every other provider in the world need to add an entry for C has the potential to cost some money, and, one way or another, C will be the one to pay. But at least there is a choice: C can consent to renumbering (which is not difficult if they have been diligent in using DHCP and perhaps NAT too), or they can pay to keep their old address block.

As for the second diagram above, with the various private links (shown as dashed lines), it is likely that the longest-match rule is *not* needed for these links to work. A's "private" link to P1 might only mean that

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

P2, in other words, is still free to route to A via P0. P1 may not *advertise* its route to A to anyone else.

14.4.3 Hierarchical Routing via Providers

With provider-based routing, the route taken may no longer be end-to-end optimal; we have replaced the problem of finding an optimal route from A to B with the two problems of finding an optimal route from A to B's provider P, and then from P's entry point to B. This strategy mirrors the two-stage hierarchical routing process of first routing on the address bits that identify the provider, and then routing on the address bits including the subscriber portion.

This two-stage strategy may not yield the same result as finding the globally optimal route. The result *will* be the same if B's customers can only be reached through P's single entry-point router RP, which models the situation that P and its customers look like a single site. However, either or both of the following can disrupt this model:

- There may be multiple entry-point routers into provider P's network, *eg* RP_1 , RP_2 and RP_3 , with different costs from A.
- P's customer B may have an alternative connection to the outside world via a different provider, as in the second diagram in [14.4 Provider-Based Routing](#).

Consider the following example representing the first situation (the more important one in practice), in which providers P1 and P2 have three interconnection points IX1, IX2, IX3 (from Internet eXchange, [14.4.1 Internet Exchange Points](#)). Links are labeled with costs; we assume that P1's costs happen to be comparable with P2's costs.

The globally shortest path between A and B is via the R2-IX2-S2 crossover, with total length $5+1+0+4=10$. However, traffic from A to B will be routed by P1 to its closest crossover to P2, namely the R3-IX3-S3 link. The total path is $3+0+1+8+4=16$. Traffic from B to A will be routed by P2 via the R1-IX1-S1 crossover, for a length of $3+0+1+7+5=16$. This routing strategy is sometimes called **hot-potato** routing; each provider tries to get rid of any traffic (the potatoes) as quickly as possible, by routing to the closest exit point.

Not only are the paths taken *inefficient*, but the A→B and B→A paths are now **asymmetric**. This can be a problem if forward and reverse timings are critical, or if one of P1 or P2 has significantly more bandwidth or less congestion than the other. In practice, however, route asymmetry is seldom important.

As for the route inefficiency itself, this also is not necessarily a significant problem; the primary reason routing-update algorithms focus on the *shortest* path is to guarantee that all computed paths are loop-free. As long as each half of a path is loop-free, and the halves do not intersect except at their common midpoint, these paths too will be loop-free.

The BGP “MED” value ([15.6.3 MULTI_EXIT_DISC](#)) offers an optional mechanism for P1 to agree that A→B traffic should take the r1-s1 crossover. This might be desired if P1's network were “better” and customer A was willing to pay extra to keep its traffic within P1's network as long as possible.

14.4.4 IP Geolocation

In principle, provider-based addressing may mean that consecutive IP addresses are scattered all over a continent. In practice, providers (even many mobile providers) do not do this; any given small address block – perhaps /24 – is used in a limited geographical area. Different blocks are used in different areas. A consequence of this is that it is possible in principle to determine, from a given IP address, the corresponding approximate geographical location; this is known as **IP geolocation**. Even satellite-Internet users can be geolocated, although sometimes only to within a couple hundred miles. Several companies have created detailed geolocation maps, identifying many locations roughly down to the [zip code](#), and typically available as an online service.

IP geolocation was originally developed so that advertisers could serve up regionally appropriate advertisements. It is, however, now used for a variety of purposes including identification of the closest CDN edge server ([1.12.2 Content-Distribution Networks](#)),

network security, compliance with national regulations, higher-level user tracking, and restricting the streaming of copyrighted content.

14.5 Geographical Routing

The classical alternative to provider-based routing is geographical routing; the archetypal model for this is the telephone area code system. A call from anywhere in the US to Loyola University's main switchboard, 773-274-3000, would traditionally be routed first to the 773 area code in Chicago. From there the call would be routed to the north-side 274 exchange, and from there to subscriber 3000. A similar strategy *can* be used for IP routing.

Geographical addressing has some advantages. Figuring out a good route to a destination is usually straightforward, and close to optimal in terms of the path physical distance. Changing providers never involves renumbering (though moving may). And approximate IP address geolocation (determining a host's location from its IP address) is automatic.

Geographical routing has some minor technical problems. First, routing may be inefficient between immediate neighbors A and B that happen to be split by a boundary for larger geographical areas; the path might go from A to the center of A's region to the center of B's region and then to B. Another problem is that some larger sites (*eg* large corporations) are themselves geographically distributed; if efficiency is the goal, each office of such a site would need a separate IP address block appropriate for its physical location.

But the real issue with geographical routing is apparently the business question of who carries the traffic. The provider-based model has a very natural answer to this: every link is owned by a specific provider. For geographical IP routing, my local provider might know at once from the prefix that a packet of mine is to be delivered from Chicago to San Francisco, but who will carry it there? My provider might have to enter into different traffic contracts for multiple different regions. If different local providers make different arrangements for long-haul packet delivery, the routing efficiency (at least in terms of table size) of geographical routing is likely lost. Finally, there is no natural answer for who should own those long inter-region links. It may be useful to recall that the present area-code system was created when the US telephone system was an AT&T monopoly, and the question of who carried traffic did not exist.

That said, the top five Regional Internet Registries represent geographical regions (usually continents), and provider-based addressing is *below* that level. That is, the IANA handed out address blocks to the geographical RIRs, and the RIRs then allocated address blocks to providers.

At the intercontinental level, geography does matter: some physical link paths are genuinely more expensive than other (shorter) paths. It is much easier to string

terrestrial cable than undersea cable. However, within a continent physical distance does not always matter as much as might be supposed. Furthermore, a large geographically spread-out provider can always divide up its address blocks by region, allowing internal geographical routing to the correct region.

Here is a diagram of IP address allocation as of 2006: <http://xkcd.com/195>.

14.6 Epilog

CIDR was a deceptively simple idea. At first glance it is a straightforward extension of the subnet concept, moving the net/host division point to the left as well as to the right. But it has ushered in true hierarchical routing, most often provider-based. While CIDR was originally offered as a solution to some early crises in IPv4 address-space allocation, it has been adopted into the core of IPv6 routing as well.

14.7 Exercises

Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises. Exercises marked with a \diamond have solutions or hints at [34.11 Solutions for Large-Scale IP Routing](#).

1.0. \diamond Consider the following IPv4 forwarding table that uses CIDR.

destination	next_hop
200.0.0.0/8	A
200.64.0.0/10	B
200.64.0.0/12	C
200.64.0.0/16	D

For each of the following IP addresses, indicate to what destination it is forwarded. 64 is 0x40, or 0100 0000 in binary.

- (i) 200.63.1.1
- (ii) 200.80.1.1
- (iii) 200.72.1.1
- (iv) 200.64.1.1

2.0. Consider the following IPv4 forwarding table that uses CIDR. IP address bytes are in **hexadecimal** here, so each hex digit corresponds to four address bits. This makes prefixes such as /12 and /20 align with hex-digit boundaries. As a reminder of the hexadecimal numbering, “:” is used as the separator rather than “.”

destination	next_hop
81:30:0:0/12	A
81:3c:0:0/16	B
81:3c:50:0/20	C
81:40:0:0/12	D

destination	next_hop
81:44:0:0/14	E

real hex IPv4 addresses

The hexadecimal format used here for IPv4 addresses is not allowed in the real world, but it turns out there *is* a valid hex format with reasonably common support, namely 0xac.0xd9.0x04.0xce. This was never standardized beyond being something BSD Unix did, but as of 2021 it is recognized (with `http://` in front of it) by the Chrome and Firefox browsers. For more examples of archaic IPv4 address formats, see [this blog post](#).

For each of the following IP addresses, give the next_hop for each entry in the table above that it matches. If there are multiple matches, use the longest-match rule to identify where the packet would be forwarded.

- (i) 81:3b:15:49
- (ii) 81:3c:56:14
- (iii) 81:3c:85:2e
- (iv) 81:4a:35:29
- (v) 81:47:21:97
- (vi) 81:43:01:c0

3.0. Consider the following IPv4 forwarding table, using CIDR. As in exercise 1, IP address bytes are in **hexadecimal**, and “.” is used as the separator as a reminder.

destination	next_hop
00:0:0:0/2	A
40:0:0:0/2	B
80:0:0:0/2	C
c0:0:0:0/2	D

- (a). To what next_hop would each of the following be routed? 63:b1:82:15, 9e:00:15:01, de:ad:be:ef
- (b). Explain why every IP address is routed somewhere, even though there is no default entry. Hint: convert the first bytes to binary.

4.0. Give an IPv4 forwarding table – using CIDR – that will route all Class A addresses (first bit 0) to next_hop A, all Class B addresses (first two bits 10) to next_hop B, and all Class C addresses (first three bits 110) to next_hop C.

5.0. Suppose an IPv4 router using CIDR has the following entries. Address bytes are in decimal except for the third byte, which is in **binary**.

destination	next_hop
37.149.0000 0000.0/18	A
37.149.0100 0000.0/18	A
37.149.1000 0000.0/18	A
37.149.1100 0000.0/18	B

If the next_hop for the last entry were also A, we could consolidate these four into a single entry $37.149.0.0/16 \rightarrow A$. But with the final next_hop as B, how could these four be consolidated into *two* entries? You will need to assume the longest-match rule.

6.0. Suppose P, Q and R are ISPs with respective CIDR address blocks (with bytes in decimal) $51.0.0.0/8$, $52.0.0.0/8$ and $53.0.0.0/8$. P then has customers A and B, to which it assigns address blocks as follows:

A: $51.10.0.0/16$

B: $51.23.0.0/16$

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

C: $52.14.0.0/16$

D: $52.15.0.0/16$

(a). ♦ Give forwarding tables for P, Q and R assuming they connect to each other and to each of their own customers.

(b). Now suppose A switches from provider P to provider Q, and takes its address block with it. Give the changes to the forwarding tables for P, Q and R. Do not assume P is willing to forward traffic from R destined for its ex-customer A. the longest-match rule may be needed to resolve conflicts.

7.0. Let P, Q and R be the ISPs of exercise 6.0. This time, suppose customer C switches from provider Q to provider R. R will now have a new entry $52.14.0.0/16 \rightarrow C$. Give the changes to the forwarding tables of P and Q. Again, do not assume provider Q will forward traffic from P destined for C.

8.0. Suppose P, Q and R are ISPs as in exercise 6.0. This time, P and R do not connect directly; they route traffic to one another via Q. In addition, customer B is multihomed and has a secondary connection to provider R; customer D is also multihomed and has a secondary connection to provider P. R and P use these secondary connections to send to B and D respectively; however, these secondary connections are used only within R and P respectively. Give forwarding tables for P, Q and R.

9.0. Suppose that Internet routing in the US used geographical routing, and the first 12 bits of every IP address represent a geographical area similar in size to a telephone area code. Megacorp gets the prefix $12.34.0.0/16$, based geographically in Chicago, and allocates subnets from this prefix to its offices in all 50 states. Megacorp routes all its internal traffic over its own network.

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

(b). Now suppose each office has its own link to a local ISP for outbound traffic, but still uses its $12.34.0.0/16$ IP addresses. Now what is the route of traffic between the San Diego office and its neighbor?

(c). Suppose Megacorp gives up and gets a separate geographical prefix for each office, eg $12.35.1.0/24$ for San Diego and $12.37.3.0/24$ for Boston. Traffic into and out

of Megacorp will now take geographically reasonable paths. However, Megacorp now wants to be sure that interoffice traffic stays on its internal network. How must Megacorp configure its internal IP forwarding tables to ensure this?

17 TCP Transport Basics

The standard transport protocols riding above the IP layer are **TCP** and **UDP**. As we saw in [16 UDP Transport](#), UDP provides simple datagram delivery to remote sockets, that is, to (host,port) pairs. TCP provides a much richer functionality for sending data to (connected) sockets. In this chapter we cover the basic TCP protocol; in the following chapter we cover some subtle issues related to potential data loss, some TCP implementation details, and then some protocols that serve as alternatives to TCP.

TCP is quite different in several dimensions from UDP. TCP is **stream-oriented**, meaning that the application can write data in very small or very large amounts and the TCP layer will take care of appropriate packetization (and also that TCP transmits a stream of bytes, not messages or records; cf [18.15.2 SCTP](#)). TCP is **connection-oriented**, meaning that a connection must be established before the beginning of any data transfer. TCP is **reliable**, in that TCP uses sequence numbers to ensure the correct order of delivery and a timeout/retransmission mechanism to make sure no data is lost short of massive network failure. Finally, TCP automatically uses the **sliding windows** algorithm to achieve throughput relatively close to the maximum available.

These features mean that TCP is very well suited for the transfer of large files. The two endpoints open a connection, the file data is written by one end into the connection and read by the other end, and the features above ensure that the file will be received correctly. TCP also works quite well for interactive applications where each side is sending and receiving streams of small packets. Examples of this include ssh or telnet, where packets are exchanged on each keystroke, and database connections that may carry many queries per second. TCP even works *reasonably* well for **request/reply** protocols, where one side sends a single message, the other side responds, and the connection is closed. The drawback here, however, is the overhead of setting up a new connection for each request; a better application-protocol design might be to allow multiple request/reply pairs over a single TCP connection.

Note that the connection-orientation and reliability of TCP represent abstract features built on top of the IP layer, which supports neither of them.

The connection-oriented nature of TCP warrants further explanation. With UDP, if a server opens a socket (the OS object, with corresponding socket address), then any client on the Internet can send to that socket, via its socket address. Any UDP application, therefore, must be prepared to check the source address of each packet that arrives. With TCP, all data arriving at a *connected* socket must come from the other endpoint of the connection. When a server S initially opens a socket s, that socket is “unconnected”; it is said to be in the LISTEN state. While it still has a socket address consisting of its host

and port, a LISTENing socket will never receive data directly. If a client C somewhere on the Internet wishes to send data to *s*, it must first establish a connection, which will be defined by the **socketpair** consisting of the socket addresses (that is, the (IP_addr,port) pairs) at both C and S. As part of this connection process, a new *connected* child socket *s_C* will be created; it is *s_C* that will receive any data sent from C. Usually, the server will also create a new thread or process to handle communication with *s_C*. Typically the server will have multiple connected children of the original socket *s*, and, for each one, a process attached to it.

If C1 and C2 both connect to *s*, two connected sockets at S will be created, *s₁* and *s₂*, and likely two separate processes. When a packet arrives at S addressed to the socket address of *s*, the *source* socket address will also be examined to determine whether the data is part of the C1–S or the C2–S connection, and thus whether a read on *s₁* or on *s₂*, respectively, will see the data.

If S is acting as an ssh server, the LISTENing socket listens on port 22, and the connected child sockets correspond to the separate user login connections; the process on each child socket represents the login process of that user, and may run for hours or days.

In Chapter 1 we likened TCP sockets to telephone connections, with the server like one high-volume phone number 800-BUY-NOWW. The unconnected socket corresponds to the number everyone dials; the connected sockets correspond to the actual calls. (This analogy breaks down, however, if one looks closely at the way such multi-operator phone lines are actually configured: each typically *does* have its own number.)

TCP was originally defined in [RFC 793](#), with important updates in [RFC 1122](#), dated October 1989. Since then there have been many miscellaneous updates; all of these have now been incorporated into a single specification [RFC 9293](#).

17.1 The End-to-End Principle

The End-to-End Principle is spelled out in [\[SRC84\]](#); it states in effect that transport issues are the responsibility of the endpoints in question and thus should not be delegated to the core network. This idea has been very influential in TCP design.

Two issues falling under this category are data corruption and congestion. For the first, even though essentially all links on the Internet have link-layer checksums to protect against data corruption, TCP still adds its own checksum (in part because of a history of data errors introduced *within* routers). For the latter, TCP is today essentially the *only* layer that addresses congestion management.

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

The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end

points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)

This does not mean that the backbone Internet should not concern itself with congestion; it means that backbone congestion-management mechanisms should not completely replace end-to-end congestion management.

17.2 TCP Header

Below is a diagram of the TCP header. As with UDP, source and destination ports are 16 bits. The 4-bit Data Offset field specifies the number of 32-bit words in the header; if no options are present its value is 5.

As with UDP, the checksum covers the TCP header, the TCP data and an IP “pseudo header” that includes the source and destination IP addresses. The checksum must be updated by a NAT router that modifies any header values. (Although the IP and TCP layers are theoretically separate, and [RFC 793](#) in some places appears to suggest that TCP can be run over a non-IP internetwork layer, [RFC 793](#) also explicitly defines 4-byte addresses for the pseudo header. [RFC 2460](#) officially redefined the pseudo header to allow IPv6 addresses.)

The **sequence** and **acknowledgment** numbers are for numbering the data, at the byte level. This allows TCP to send 1024-byte blocks of data, incrementing the sequence number by 1024 between successive packets, or to send 1-byte telnet packets, incrementing the sequence number by 1 each time. There is no distinction between DATA and ACK packets; all packets carrying data from A to B also carry the most current acknowledgment of data sent from B to A. Many TCP applications are largely unidirectional, in which case the sender would include essentially the same acknowledgment number in each packet while the receiver would include essentially the same sequence number.

It is traditional to refer to the data portion of TCP packets as **segments**.

TCP History

The clear-cut division between the IP and TCP headers did not spring forth fully formed. See [\[CK74\]](#) for a discussion of a proto-TCP in which the sequence number (but not the acknowledgment number) appeared in the equivalent of the IP header (perhaps so it could be used for fragment reassembly).

The value of the sequence number, in *relative* terms, is the position of the first byte of the packet in the data stream, or the position of what would be the first byte in the case

that no data was sent. The value of the acknowledgment number, again in relative terms, represents the byte position for the next byte expected. Thus, if a packet contains 1024 bytes of data and the first byte is number 1, then that would be the sequence number. The data bytes would be positions 1–1024, and the ACK returned would have acknowledgment number 1025.

The sequence and acknowledgment numbers, as sent, represent these relative values *plus* an **Initial Sequence Number**, or ISN, that is fixed for the lifetime of the connection. Each direction of a connection has its own ISN; see below.

TCP acknowledgments are **cumulative**: when an endpoint sends a packet with an acknowledgment number of N, it is acknowledging receipt of all data bytes numbered less than N. Standard TCP provides no mechanism for acknowledging receipt of packets 1, 2, 3 and 5; the highest cumulative acknowledgment that could be sent in that situation would be to acknowledge packet 3.

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

- **SYN**: for SYNchronize; marks packets that are part of the new-connection handshake
- **ACK**: indicates that the header Acknowledgment field is valid; that is, all but the first packet
- **FIN**: for FINish; marks packets involved in the connection closing
- **PSH**: for PuSH; marks “non-full” packets that should be delivered promptly at the far end
- **RST**: for ReSeT; indicates various error conditions
- **URG**: for URGeNT; part of a now-seldom-used mechanism for high-priority data
- **CWR** and **ECE**: part of the Explicit Congestion Notification mechanism, [21.5.3 Explicit Congestion Notification \(ECN\)](#)

17.3 TCP Connection Establishment

TCP connections are established via an exchange known as the **three-way handshake**. If A is the client and B is the LISTENing server, then the handshake proceeds as follows:

- A sends B a packet with the SYN bit set (a SYN packet)
- B responds with a SYN packet of its own; the ACK bit is now also set
- A responds to B’s SYN with its own ACK

Normally, the three-way handshake is triggered by an application’s request to connect; data can be sent only after the handshake completes. This means a one-RTT delay before any data can be sent. The original TCP standard **RFC 793** does allow data to be

sent with the first SYN packet, as part of the handshake, but such data cannot be released to the remote-endpoint application until the handshake completes. Most traditional TCP programming interfaces offer no support for this early-data option.

There are recurrent calls for TCP to support data transmission within the handshake itself, so as to achieve request/reply turnaround comparable to that with RPC ([16.5 Remote Procedure Call \(RPC\)](#)). We return to this in [18.5 TCP Faster Opening](#).

The three-way handshake is vulnerable to an attack known as **SYN flooding**. The attacker sends a large number of SYN packets to a server B. For each arriving SYN, B must allocate resources to keep track of what appears to be a legitimate connection request; with enough requests, B's resources may face exhaustion. SYN flooding is easiest if the SYN packets are simply spoofed, with forged, untraceable source-IP addresses; see spoofing at [9.1 The IPv4 Header](#), and [18.3.1 ISNs and spoofing](#) below. SYN-flood attacks can also take the form of a large number of real connection attempts from a large number of real clients – often compromised and pressed into service by some earlier attack – but this requires considerably more resources on the part of the attacker. See [18.15.2 SCTP](#) for an alternative handshake protocol (unfortunately not available to TCP) intended to mitigate SYN-flood attacks, at least from spoofed SYNs.

To **close** the connection, a superficially similar exchange involving FIN packets may occur:

- A sends B a packet with the FIN bit set (a FIN packet), announcing that it has finished sending data
- B sends A an ACK of the FIN
- B may continue to send additional data to A
- When B is also ready to cease sending, it sends its own FIN to A
- A sends B an ACK of the FIN; this is the final packet in the exchange

Here's the ladder diagram for this:

The FIN handshake is really more like two separate two-way FIN/ACK handshakes. We will return to TCP connection closing in [17.8.1 Closing a connection](#).

Now let us look at the full exchange of packets in a representative connection, in which A sends strings “abc”, “defg”, and “foobar” ([RFC 3092](#)). B replies with “hello”, and which point A sends “goodbye” and closes the connection. In the following table, *relative* sequence numbers are used, which is to say that sequence numbers begin with 0 on each side. The SEQ numbers in **bold** on the A side correspond to the ACK numbers in **bold** on the B side; they both count data flowing from A to B.

	A sends	B sends
1	SYN, seq=0	

	A sends	B sends
2		SYN+ACK, seq=0, ack=1 (expecting)
3	ACK, seq=1 , ack=1 (ACK of SYN)	
4	“abc”, seq=1 , ack=1	
5		ACK, seq=1, ack=4
6	“defg”, seq=4 , ack=1	
7		seq=1, ack=8
8	“foobar”, seq=8 , ack=1	
9		seq=1, ack=14 , “hello”
10	seq=14 , ack=6, “goodbye”	
11,12	seq=21 , ack=6, FIN	seq=6, ack=21 ;; ACK of “goodbye”, crossing packets
13		seq=6, ack=22 ;; ACK of FIN
14		seq=6, ack=22 , FIN
15	seq=22 , ack=7 ;; ACK of FIN	

(We will see below that this table is slightly idealized, in that real sequence numbers do *not* start at 0.)

Here is the ladder diagram corresponding to this connection:

In terms of the sequence and acknowledgment numbers, SYN's count as 1 byte, as do FIN's. Thus, the SYN counts as sequence number 0, and the first byte of data (the “a” of “abc”) counts as sequence number 1. Similarly, the ack=21 sent by the B side is the acknowledgment of “goodbye”, while the ack=22 is the acknowledgment of A's subsequent FIN.

Whenever B sends ACK=n, A follows by sending more data with SEQ=n.

TCP does *not* in fact transport relative sequence numbers, that is, sequence numbers as transmitted do not begin at 0. Instead, each side chooses its **Initial Sequence Number**, or **ISN**, and sends that in its initial SYN. The third ACK of the three-way handshake is an acknowledgment that the server side's SYN response was received correctly. All further sequence numbers sent are the ISN chosen by that side plus the relative sequence number (that is, the sequence number as if numbering did begin at 0). If A chose ISN_A=1000, we would add 1000 to all the bold entries above: A would send SYN(seq=1000), B would reply with ISN_B and ack=1001, and the last two lines would involve ack=1022 and seq=1022 respectively. Similarly, if B chose ISN_B=7000, then we would add 7000 to all the **seq** values in the “B sends” column and all the **ack** values in the “A sends” column. The table above up to the point B sends “goodbye”, with actual sequence numbers instead of relative sequence numbers, is below:

	A, ISN=1000	B, ISN=7000
1	SYN, seq=1000	

	A, ISN=1000	B, ISN=7000
2		SYN+ACK, seq=7000, ack=1001
3	ACK, seq=1001 , ack=7001	
4	“abc”, seq=1001 , ack=7001	
5		ACK, seq=7001, ack=1004
6	“defg”, seq=1004 , ack=7001	
7		seq=7001, ack=1008
8	“foobar”, seq=1008 , ack=7001	
9		seq=7001, ack=1014 , “hello”
10	seq=1014 , ack=7006, “goodbye”	

If B had not been LISTENing at the port to which A sent its SYN, its response would have been **RST** (“reset”), meaning in this context “connection refused”. Similarly, if A sent data to B before the SYN packet, the response would have been RST.

Finally, a RST can be sent by either side at any time to abort the connection. Sometimes routers along the path send “spoofed” RSTs to tear down TCP connections they are configured to regard as undesired; see [9.7.2 Middleboxes](#) and [RFC 3360](#). Worse, sometimes external attackers are able to tear down a TCP connection with a spoofed RST; this requires brute-force guessing the endpoint port numbers and the RST sender’s current SEQ value ([RFC 793](#) does not in general require the RST packet’s ACK value to match, but see exercise 9.0). In the days of 4 kB window sizes, guessing a valid SEQ value was a one-in-a-million chance, but window sizes have steadily increased ([21.6 The High-Bandwidth TCP Problem](#)); a 4 MB window size makes SEQ guessing quite feasible. See also [RFC 4953](#) and the RST-validation fix proposed in [RFC 5961](#) §3.2.

If A sends a series of small packets to B, then B has the option of assembling them into a full-sized I/O buffer before releasing them to the receiving application. However, if A sets the **PSH** bit on each packet, then B should release each packet immediately to the receiving application. In Berkeley Unix and most (if not all) BSD-derived socket-library implementations, there is in fact no way to set the PSH bit; it is set automatically for each write. (But even this is not *guaranteed* as the sender may leave the bit off or consolidate several PuSHed writes into one packet; this makes using the PSH bit as a record separator difficult. In a series of runs of the program written to generate the WireShark packet trace, below, most of the time the strings “abc”, “defg”, *etc* were PuSHed separately but occasionally they were consolidated into one packet.)

As for the **URG** bit, imagine a telnet (or ssh) connection, in which A has sent a large amount of data to B, which is momentarily stalled processing it. The application at A wishes to abort that processing by sending the interrupt character CNTL-C. Under normal conditions, the application at B would have to finish processing all the pending data before getting to the CNTL-C; however, the use of the URG bit can enable immediate asynchronous delivery of the CNTL-C. The bit is set, and the TCP header’s Urgent Pointer field points to the CNTL-C in the current packet, far ahead in the normal data stream. The receiving application then skips ahead in its processing of the arriving

data stream until it reaches the urgent data. For this to work, the receiving application process must have signed up to receive an asynchronous signal when urgent data arrives.

The urgent data does appear as part of the ordinary TCP data stream, and it is up to the protocol to determine the start of the data that is to be considered urgent, and what to do with the unread, buffered data sent ahead of the urgent data. For the CNTL-C example in the telnet protocol ([RFC 854](#)), the urgent data might consist of the telnet “Interrupt Process” byte, preceded by the “Interpret as Command” escape byte, and the earlier data is simply discarded.

Officially, the Urgent Pointer value, when the **URG** bit is set, contains the offset from the start of the current packet data to the *end* of the urgent data; it is meant to tell the receiver “you should read up to this point as soon as you can”. The original intent was for the urgent pointer to mark the last byte of the urgent data, but §3.1 of [RFC 793](#) got this wrong and declared that it pointed to the first byte *following* the urgent data. This was corrected in [RFC 1122](#), but most implementations to this day abide by the “incorrect” interpretation. [RFC 6093](#) discusses this and proposes, first, that the near-universal “incorrect” interpretation be accepted as standard, and, second, that developers avoid the use of the TCP urgent-data feature.

17.4 TCP and WireShark

Below is a screenshot of the [WireShark](#) program displaying a tcpdump capture intended to represent the TCP exchange above. Both hosts involved in the packet exchange were Linux systems. Side A uses socket address (10.0.0.3,45815) and side B (the server) uses (10.0.0.1,54321).

WireShark is displaying *relative* TCP sequence numbers. The first three packets correspond to the three-way handshake, and packet 4 is the first data packet. Every data packet has the flags [PSH, ACK] displayed. The data in the packet can be inferred from the WireShark Len field, as each of the data strings sent has a different length.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	10.0.0.3	10.0.0.1	TCP	45815 > 54321 [SYN] Seq=0 Win=5840 Len=0 MSS=1460 TSV=1019572186 TSER=
2	0.000517	10.0.0.1	10.0.0.3	TCP	54321 > 45815 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 TSV=3681
3	0.000578	10.0.0.3	10.0.0.1	TCP	45815 > 54321 [ACK] Seq=1 Ack=1 Win=5888 Len=0 TSV=1019572187 TSER=36
4	0.015863	10.0.0.3	10.0.0.1	TCP	45815 > 54321 [PSH, ACK] Seq=1 Ack=1 Win=5888 Len=3 TSV=1019572190 TS
5	0.016139	10.0.0.1	10.0.0.3	TCP	54321 > 45815 [ACK] Seq=1 Ack=4 Win=5792 Len=0 TSV=36819608 TSER=1019
6	0.116268	10.0.0.3	10.0.0.1	TCP	45815 > 54321 [PSH, ACK] Seq=4 Ack=1 Win=5888 Len=4 TSV=1019572215 TS
7	0.116730	10.0.0.1	10.0.0.3	TCP	54321 > 45815 [ACK] Seq=1 Ack=8 Win=5792 Len=0 TSV=36819618 TSER=1019
8	0.217328	10.0.0.3	10.0.0.1	TCP	45815 > 54321 [PSH, ACK] Seq=8 Ack=1 Win=5888 Len=6 TSV=1019572241 TS
9	0.217539	10.0.0.1	10.0.0.3	TCP	54321 > 45815 [ACK] Seq=1 Ack=14 Win=5792 Len=0 TSV=36819628 TSER=101
10	0.218665	10.0.0.1	10.0.0.3	TCP	54321 > 45815 [PSH, ACK] Seq=1 Ack=14 Win=5792 Len=5 TSV=36819628 TSE
11	0.218729	10.0.0.3	10.0.0.1	TCP	45815 > 54321 [ACK] Seq=14 Ack=6 Win=5888 Len=0 TSV=1019572241 TSER=3
12	0.319319	10.0.0.3	10.0.0.1	TCP	45815 > 54321 [PSH, ACK] Seq=14 Ack=6 Win=5888 Len=7 TSV=1019572266 T
13	0.329759	10.0.0.3	10.0.0.1	TCP	45815 > 54321 [FIN, ACK] Seq=21 Ack=6 Win=5888 Len=0 TSV=1019572269 T
14	0.355151	10.0.0.1	10.0.0.3	TCP	54321 > 45815 [ACK] Seq=6 Ack=22 Win=5792 Len=0 TSV=36819642 TSER=101
15	0.370970	10.0.0.1	10.0.0.3	TCP	54321 > 45815 [FIN, ACK] Seq=6 Ack=22 Win=5792 Len=0 TSV=36819643 TSE
16	0.371009	10.0.0.3	10.0.0.1	TCP	45815 > 54321 [ACK] Seq=22 Ack=7 Win=5888 Len=0 TSV=1019572279 TSER=3

Frame 12 (73 bytes on wire, 73 bytes captured)

Ethernet II, Src: Usi_e1:f9:b2 (00:24:7e:e1:f9:b2), Dst: 3com_b0:e5:f3 (00:60:08:b0:e5:f3)

Internet Protocol, Src: 10.0.0.3 (10.0.0.3), Dst: 10.0.0.1 (10.0.0.1)

Transmission Control Protocol, Src Port: 45815 (45815), Dst Port: 54321 (54321), Seq: 14, Ack: 6, Len: 7

Data (7 bytes)

Data: 676F6F64627965
[Length: 7]

```

0000 00 60 08 b0 e5 f3 00 24 7e e1 f9 b2 08 00 45 00  . . . . . $ ~ . . . . . E.
0010 00 3b a2 e7 40 00 40 06 84 32 0a 00 00 03 0a 00  . ; . @ . @ . 2 . . . . .
0020 00 01 b2 f7 d4 31 20 d9 cd 69 51 75 d4 68 80 18  . . . . . 1 . . i Qu . h . .
0030 00 5c 14 31 00 00 01 01 08 0a 3c c5 70 2a 02 31  . \ . 1 . . . . . < . p * . 1
0040 d2 ac 67 6f 6f 64 62 79 65                       . . goodbye e

```

Frame (frame), 73 bytes Packets: 16 Displayed: 16 Marked: 0 Profile: Default

The packets are numbered the same as in the table above up through packet 8, containing the string “foobar”. At that point the table shows B replying by a combined ACK plus the string “hello”; in fact, TCP sent the ACK alone and then the string “hello”; these are WireShark packets 9 and 10 (note packet 10 has Len=5). WireShark packet 11 is then a standalone ACK from A to B, acknowledging the “hello”. WireShark packet 12 (the packet highlighted) then corresponds to table packet 10, and contains “goodbye” (Len=7); this string can be seen at the right side of the bottom pane.

The table view shows A’s FIN (packet 11) crossing with B’s ACK of “goodbye” (packet 12). In the WireShark view, A’s FIN is packet 13, and is sent about 0.01 seconds after “goodbye”; B then ACKs them both with packet 14. That is, the table-view packet 12 does not exist in the WireShark view.

Packets 11, 13, 14 and 15 in the table and 13, 14, 15 and 16 in the WireShark screen dump correspond to the connection closing. The program that generated the exchange at B’s side had to include a “sleep” delay of 40 ms between detecting the closed connection (that is, reading A’s FIN) and closing its own connection (and sending its own FIN); otherwise the ACK of A’s FIN traveled in the same packet with B’s FIN.

The ISN for A in this example was 551144795 and B’s ISN was 1366676578. The actual pcap packet-capture file is at [demo_tcp_connection.pcap](#). This pcap file was generated by a TCP connection between two physical machines; for an alternative approach to observing TCP behavior see [30.2.2 Mininet WireShark Demos](#).

17.5 TCP Offloading

In the Wireshark example above, the hardware involved used **TCP checksum offloading**, or TCO, to have the network-interface card do the actual checksum calculations; this permits a modest amount of parallelism. As a result, the checksums for outbound packets are wrong in the capture file. WireShark has an option to disable the reporting of this. Despite the name, TCO can handle UDP packets as well.

Most Ethernet (and some Wi-Fi) cards have the ability to calculate the Internet checksum ([7.4 Error Detection](#)) over a certain range of bytes, and store the result (after taking the complement) at a designated offset. However, cards cannot, as a rule, handle the UDP and TCP “pseudo headers”. So what happens is the host system calculates the pseudo-header checksum and stores it in the normal checksum field; the LAN card then includes this pseudo-header checksum value in its own checksum calculation, and the correct result is obtained.

It is also possible, with many newer network-interface cards, to offload the TCP *segmentation* process to the LAN hardware; that is, the kernel sends a very large TCP buffer – perhaps 64 KB – to the LAN hardware, along with a TCP header, and the LAN hardware divides the buffer into multiple TCP packets of at most 1500 bytes each. This is most useful when the application is writing data continuously and is known as **TCP segmentation offloading**, or TSO. The use of TSO requires TCO, but not vice-versa.

TSO can be divided into large *send* offloading, LSO, for outbound traffic, as above, and large receive offloading, LRO, for inbound. For inbound offloading, the network card accumulates multiple inbound packets that are part of the same TCP connection, and consolidates them in proper sequence to one much larger packet. This means that the network card, upon receiving one packet, must wait to see if there will be more. This wait is very short, however, at most a few milliseconds. Specifically, all consolidated incoming packets must have the same TCP Timestamp value ([18.4 Anomalous TCP scenarios](#)).

TSO is of particular importance at very high bandwidths. At 10 Gbps, a system can send or receive close to a million packets per second, and offloading some of the packet processing to the network card can be essential to maintaining high throughput. TSO allows a host system to behave as if it were reading or writing very large packets, and yet the actual packet size on the wire remains at the standard 1500 bytes.

On Linux systems, the status of TCO and TSO can be checked using the command `ethtool --show-offload interface`. TSO can be disabled with `ethtool -offload interface tso off`.

17.6 TCP simplex-talk

Here is a Java version of the simplex-talk server for TCP. As with the UDP version, we start by setting up the socket, here a `ServerSocket` called `ss`. This socket remains in

the LISTEN state throughout. The main `while` loop then begins with the call `ss.accept()` at the start; this call blocks until an incoming connection is established, at which point it returns the connected child socket `s`. The `accept()` call models the TCP protocol behavior of waiting for three-way handshakes initiated by remote hosts and, for each, setting up a new connection.

Connections will be accepted from *all* IP addresses of the server host, *eg* the “normal” IP address, the loopback address 127.0.0.1 and, if the server is multihomed, any additional IP addresses. Unlike the UDP case ([16.1.3.2 UDP and IP addresses](#)), [RFC 1122](#) requires (§4.2.3.7) that server response packets always be sent from the same server IP address that the client first used to contact the server. (See [18 TCP Issues and Alternatives](#), exercise 5.0 for an example of non-compliance.)

A server application can process these connected children either serially or in parallel. The stalk version here can handle both situations, either one connection at a time (`THREADING = false`), or by creating a new thread for each connection (`THREADING = true`). Either way, the connected child socket is turned over to `line_talker()`, either as a synchronous procedure call or as a new thread. Data is then read from the socket’s associated `InputStream` using the ordinary `read()` call, versus the `receive()` used to read UDP packets. The main loop within `line_talker()` does not terminate until the client closes the connection (or there is an error).

In the serial, non-threading mode, if a second client connection is made while the first is still active, then data can be sent on the second connection but it sits in limbo until the first connection closes, at which point control returns to the `ss.accept()` call, the second connection is processed, and the second connection’s data suddenly appears.

In the threading mode, the main loop spends almost all its time waiting in `ss.accept()`; when this returns a child connection we immediately spawn a new thread to handle it, allowing the parent process to go back to `ss.accept()`. This allows the program to accept multiple concurrent client connections, like the UDP version.

The code here serves as a very basic example of the creation of Java threads. The inner class `Talker` has a `run()` method, needed to implement the `Runnable` interface. To start a new thread, we create a new `Talker` instance; the `start()` call then begins `Talker.run()`, which runs for as long as the client keeps the connection open. The file here is [tcp_stalks.java](#).

```
/* THREADED simplex-talk TCP server */
/* can handle multiple CONCURRENT client connections */
/* newline is to be included at client side */

import java.net.*;
import java.io.*;

public class tstalks {
```



```

static public int destport = 5431;
static public int bufsize = 512;
static public boolean THREADING = true;

static public void main(String args[]) {
    ServerSocket ss;
    Socket s;
    try {
        ss = new ServerSocket(destport);
    } catch (IOException ioe) {
        System.err.println("can't create server socket");
        return;
    }
    System.err.println("server starting on port " + ss.getLocalPort());

    while(true) { // accept loop
        try {
            s = ss.accept();
        } catch (IOException ioe) {
            System.err.println("Can't accept");
            break;
        }

        if (THREADING) {
            Talker talk = new Talker(s);
            (new Thread(talk)).start();
        } else {
            line_talker(s);
        }
    } // accept loop
} // end of main

public static void line_talker(Socket s) {
    int port = s.getPort();
    InputStream istr;
    try { istr = s.getInputStream(); }
    catch (IOException ioe) {
        System.err.println("cannot get input stream");           // most
likely cause: s was closed
        return;
    }
    System.err.println("New connection from <" +
        s.getInetAddress().getHostAddress() + "," + s.getPort() + ">");
    byte[] buf = new byte[bufsize];
    int len;

    while (true) {           // while not done reading the socket
        try {
            len = istr.read(buf, 0, bufsize);
        }
        catch (SocketTimeoutException ste) {
            System.out.println("socket timeout");
            continue;
        }
        catch (IOException ioe) {
            System.err.println("bad read");
            break;           // probably a socket ABORT; treat as a close
        }
    }
}

```

```

        if (len == -1) break;           // other end closed gracefully
        String str = new String(buf, 0, len);
        System.out.print("" + port + ": " + str); // str should contain
newline
    } //while reading from s

    try {istr.close();}
    catch (IOException ioe) {System.err.println("bad stream
close");return;}
    try {s.close();}
    catch (IOException ioe) {System.err.println("bad socket
close");return;}
    System.err.println("socket to port " + port + " closed");
} // line_talker

static class Talker implements Runnable {
    private Socket _s;

    public Talker (Socket s) {
        _s = s;
    }

    public void run() {
        line_talker(_s);
    } // run
} // class Talker
}

```

17.6.1 The TCP Client

Here is the corresponding client [tcp_stalkc.java](#). As with the UDP version, the default host to connect to is localhost. We first call `InetAddress.getByName()` to perform the DNS lookup. Part of the construction of the `Socket` object is the connection to the desired dest and destport. Within the main while loop, we use an ordinary `write()` call to write strings to the socket's associated `OutputStream`.

```

// TCP simplex-talk CLIENT in java

import java.net.*;
import java.io.*;

public class stalkc {

    static public BufferedReader bin;
    static public int destport = 5431;

    static public void main(String args[]) {
        String desthost = "localhost";
        if (args.length >= 1) desthost = args[0];
        bin = new BufferedReader(new InputStreamReader(System.in));

        InetAddress dest;
        System.err.print("Looking up address of " + desthost + "...");
        try {

```

```

        dest = InetAddress.getByName(desthost);
    }
    catch (UnknownHostException uhe) {
        System.err.println("unknown host: " + desthost);
        return;
    }
    System.err.println(" got it!");

    System.err.println("connecting to port " + destport);
    Socket s;
    try {
        s = new Socket(dest, destport);
    }
    catch(IOException ioe) {
        System.err.println("cannot connect to <" + desthost + ", " +
destport + ">");
        return;
    }

    OutputStream sout;
    try {
        sout = s.getOutputStream();
    }
    catch (IOException ioe) {
        System.err.println("I/O failure!");
        return;
    }

    //=====

    while (true) {
        String buf;
        try {
            buf = bin.readLine();
        }
        catch (IOException ioe) {
            System.err.println("readLine() failed");
            return;
        }
        if (buf == null) break;    // user typed EOF character

        buf = buf + "\n";        // protocol requires sender includes \n
        byte[] bbuf = buf.getBytes();

        try {
            sout.write(bbuf);
        }
        catch (IOException ioe) {
            System.err.println("write() failed");
            return;
        }
    } // while
}

```

A Python3 version of the stalk client is available at [tcp_stalkc.py](https://github.com/0x00sec/tcp_stalkc.py).

Here are some things to try with `THREADING=false` in the server:

- start up two clients while the server is running. Type some message lines into both. Then exit the first client.
- start up the client before the server.
- start up the server, and then the client. Kill the server and then type some message lines to the client. What happens to the client? (It may take a couple message lines.)
- start the server, then the client. Kill the server and restart it. Now what happens to the client?

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

See also exercise 13.0.

17.7 TCP and `bind()`

The server version calls the `ServerSocket()` constructor, to create a socket in the `LISTEN` state; the local port must be specified here. The client version just calls `Socket()`, and the socket is then bound to a port by the operating system. It is also possible to create a client socket that is bound to a programmer-specified port; one application of this is to enable internal firewalls to identify the traffic class by source port. More commonly, client sockets are assigned *ephemeral* ports by the system. The Linux ephemeral port range can be found in `/proc/sys/net/ipv4/ip_local_port_range`; as of 2022 it is 32768 to 60999.

In C, sockets are created with `socket()`, bound to a port with `bind()`, and placed in the `LISTEN` state with `listen()` or else connected to a server with `connect()`. For client sockets, the call to `bind()` may be performed implicitly by `connect()`.

If two sockets on host A are connected via TCP to two *different* servers, B1 and B2, then it is possible that the operating system will assign the *same* local port to both sockets. On systems with an exceptional number of persistent outbound connections, such port reuse may be essential, as it is otherwise possible to run out of local ports. That said, port reuse is not an option if `bind()` is called explicitly, as at the time of the call to `bind()` the operating system does not yet know if the socket is to be used for `LISTENing`, for which a unique local port is necessary. To help deal with this, Linux has the socket option `IP_BIND_ADDRESS_NO_PORT`, which causes `bind()` to bind an IP address to the socket but defers port binding to a later `connect()`. Apple OS X provides the `connectx()` system call, which introduces similar functionality. See also [this Cloudflare blog post](#).

Ephemeral ports were, originally, assigned more-or-less in sequence, skipping over values in use and ultimately wrapping around. This makes it easy, however, for

adversaries to predict source-port numbers, so [RFC 6056](#) proposed making a first try at a new local-port number as follows:

```
try0 = next_ephemeral + hash(local_addr, remote_addr, remote_port,
secret_key)
```

If that is not available, subsequent tries incremented this value successively, with appropriate wrapping to stay within the designated ephemeral-port range. (See [18.3.1 ISNs and spoofing](#) for a related technique with ISN generation.)

This had the advantage that any *specific* remote socket would see the local ports incremented successively, which makes port reuse unlikely until the entire ephemeral-port range has been cycled through. However, a different remote socket would get another, unrelated, port-number sequence, making it very difficult for attackers to guess another connection's port.

Unfortunately, this elegant scheme introduced an unexpected problem: it enabled **fingerprinting** of the system that uses it, which lasted for the lifetime of the `secret_key`. This was done through the creation of many “probe” connections, and observing the behavior of the `local_port` value; see [\[KK22\]](#) for details. The fix is to add a fast-changing timestamp to the hash arguments above, and to increment a failed port try by a random value between 1 and 7, rather than always by 1.

17.7.1 netcat again

As with UDP ([16.1.4 netcat](#)), we can use the netcat utility to act as either end of the TCP simplex-talk connection. As the client we can use

```
netcat localhost 5431
```

while as the server we can use

```
netcat -l -k 5431
```

Here (but not with UDP) the `-k` option causes the server to accept multiple connections in sequence. The connections are handled one at a time, as is the case in the stalk server above with `THREADING=false`.

We can also use netcat to download web pages, using the [HTTP](#) protocol. The command below sends an HTTP GET request (version 1.1; [RFC 2616](#) and updates) to retrieve part of the website for this book; it has been broken over two lines for convenience.

```
echo -e 'GET /index.html HTTP/1.1\r\nHOST: intronetworks.cs.luc.edu\r\n'|
netcat intronetworks.cs.luc.edu 80
```

The `\r\n` represents the officially mandatory carriage–return/newline line–ending sequence, though `\n` will often work. The `index.html` identifies the file being requested; as `index.html` is the default it is often omitted, though the preceding `/` is still required. The webserver may support other websites as well via virtual hosting ([10.1.2 nslookup and dig](#)); the `HOST:` specification identifies to the server the specific site we are looking for. Version 2 of HTTP is described in [RFC 7540](#); its primary format is binary. (For production command–line retrieval of web pages, [cURL](#) and [wget](#) are standard choices.)

17.8 TCP state diagram

A formal definition of TCP involves the **state diagram**, with conditions for transferring from one state to another, and responses to all packets from each state. The state diagram originally appeared in [RFC 793](#); the following interpretation of the state diagram came from http://commons.wikimedia.org/wiki/File:Tcp_state_diagram_fixed.svg and was authored by Wikipedia users Sergiodc2, Marty Pauley, and DnetSvg. The blue arrows indicate the sequence of state transitions typically followed by the server; the brown arrows represent the client. Arrows are labeled with **event / action**; that is, we move from `LISTEN` to `SYN_RECV` upon receipt of a `SYN` packet; the action is to respond with `SYN+ACK`.

In general, this finite–state–machine approach to protocol specification has proven very effective, and is now used for most protocols. It makes it very clear to the implementer how the system should respond to each packet arrival. It is also a useful model for the implementation itself. Finally, we also observe that the TCP layer within an operating system cannot easily be modeled as anything *other* than a state machine; it must respond immediately to packet and program events, without indefinite waiting, as the operating system must go on to other things.

It is visually impractical to list every possible transition within the state diagram, full details are usually left to the accompanying text. For example, although this does not appear in the state diagram above, the per–state response rules of TCP require that in the `ESTABLISHED` state, if the receiver sends an `ACK` outside the current sliding window, then the correct response is to reply with one’s own current `ACK`. This includes the case where the receiver *acknowledges data not yet sent*.

The `ESTABLISHED` state and the states below it are sometimes called the **synchronized** states, as in these states both sides have confirmed one another’s ISN values.

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

Although it essentially never occurs in practice, it is possible for each side to send the other a SYN, requesting a connection, **simultaneously** (that is, the SYNs cross on the wire). The telephony analogue occurs when each party dials the other simultaneously. On traditional land-lines, each party then gets a busy signal. On cell phones, your mileage may vary. With TCP, a single connection is created. With OSI TP4, two connections are created. The OSI approach is not possible in TCP, as a connection is determined only by the socketpair involved; if there is only one socketpair then there can be only one connection.

It is possible to view connection states under either Linux or Windows with `netstat -a`. Most states are ephemeral, exceptions being LISTEN, ESTABLISHED, TIMEWAIT, and CLOSE_WAIT. One sometimes sees large numbers of connections in CLOSE_WAIT, meaning that the remote endpoint has closed the connection and sent its FIN, but the process at your end has not executed `close()` on its socket. Often this represents a programming error; alternatively, the process at the local end is still working on something. Given a local port number `p` in state CLOSE_WAIT on a Linux system, the (privileged) command `lsof -i :p` will identify the process using port `p`.

The reader who is implementing TCP is encouraged to consult [RFC 793](#) and updates. For the rest of us, below are a few general observations about closing connections.

17.8.1 Closing a connection

The “normal” TCP close sequence is as follows:

A’s FIN is, in effect, a promise to B not to *send* any more. However, A must still be prepared to receive data from B, hence the optional data shown in the diagram. A good example of this occurs when A is sending a stream of data to B to be sorted; A sends FIN to indicate that it is done sending, and only then does B sort the data and begin sending it back to A. This can be generated with the command, on A, `cat thefile | ssh B sort`. That said, the presence of the optional B-to-A data above following A’s FIN is relatively less common.

In the diagram above, A sends a FIN to B and receives an ACK, and then, later, B sends a FIN to A and receives an ACK. This essentially amounts to two separate two-way closure handshakes.

Either side may elect to close the connection, just as either party to a telephone call may elect to hang up. The first side to send a FIN – A in the diagram above – takes the **Active CLOSE** path; the other side takes the **Passive CLOSE** path. In the diagram, active-closer A moves from state ESTABLISHED to FIN_WAIT_1 to FIN_WAIT_2 (upon receipt of B’s ACK of

A's FIN), and then to TIMEWAIT and finally to CLOSED. Passive-closer B moves from ESTABLISHED to CLOSE_WAIT to LAST_ACK to CLOSED.

A **simultaneous close** – having both sides send each other FINs before receiving the other side's FIN – is a little more likely than a simultaneous open, earlier above, though still not very. Each side would send its FIN and move to state FIN_WAIT_1. Then, upon receiving each other's FIN packets, each side would send its final ACK and move to CLOSING. See exercises 5.0 and 6.0.

A TCP endpoint is **half-closed** if it has sent its FIN (thus promising not to send any more data) and is waiting for the other side's FIN; this corresponds to A in the diagram above in states FIN_WAIT_1 and FIN_WAIT_2. With the BSD socket library, an application can half-close its connection with the appropriate call to `shutdown()`.

Unrelatedly, A TCP endpoint is **half-open** if it is in the ESTABLISHED state, but during a lull in the exchange of packets the other side has rebooted; this has nothing to do with the close protocol. As soon as the ESTABLISHED side sends a packet, the rebooted side will respond with RST and the connection will be fully closed.

In the absence of the optional data from B to A after A sends its FIN, the closing sequence reduces to the left-hand diagram below:

If B is ready to close immediately, it is possible for B's ACK and FIN to be combined, as in the right-hand diagram above, in which case the resultant diagram superficially resembles the connection-opening three-way handshake. In this case, A moves directly from FIN_WAIT_1 to TIMEWAIT, following the state-diagram link labeled "FIN + ACK-of-FIN". In theory this is rare, as the ACK of A's FIN is generated by the kernel but B's FIN cannot be sent until B's process is scheduled to run on the CPU. If the TCP layer adopts a policy of *immediately* sending ACKs upon receipt of any packet, this will never happen, as the ACK will be sent well before B's process can be scheduled to do anything. However, if B *delays* its ACKs slightly (and if it has no more data to send), then it is possible – and in fact not uncommon – for B's ACK and FIN to be sent together. Delayed ACKs, are, as we shall see below, a common strategy ([18.8 TCP Delayed ACKs](#)). To create the scenario of [17.4 TCP and WireShark](#), it was necessary to introduce an artificial delay to prevent the simultaneous transmission of B's ACK and FIN.

17.8.2 Calling `close()`

Most network programming interfaces provide a `close()` method for ending a connection, based on the close operation for files. However, it usually closes bidirectionally and so models the TCP closure protocol rather imperfectly.

As we have seen in the previous section, the TCP close sequence is followed more naturally if the active-closing endpoint calls `shutdown()` – promising not to send more,

but allowing for continued receiving – before the final `close()`. Here is what *should* happen at the application layer if endpoint A of a TCP connection wishes to initiate the closing of its connection with endpoint B:

- A's application calls `shutdown()`, thereby promising not to send any more data. A's FIN is sent to B. A's application is expected to continue reading, however.
- The connection is now half-closed. On receipt of A's FIN, B's TCP layer knows this. If B's application attempts to read more data, it will receive an end-of-file indication (this is typically a `read()` or `recv()` operation that returns immediately with 0 bytes received).
- B's application is now done reading data, but it may or may not have more data to send. When B's application is done sending, it calls `close()`, at which point B's FIN is sent to A. Because the connection is already half-closed, B's `close()` is really a second half-close, ending further transmission by B.
- A's application keeps reading until it too receives an end-of-file indication, corresponding to B's FIN.
- The connection is now fully closed. No data has been lost.

It is sometimes the case that it is evident to A from the application protocol that B will not send more data. In such cases, A might simply call `close()` instead of `shutdown()`. This is risky, however, unless the protocol is crystal clear: if A calls `close()` and B later does send a little more data after all, or if B has already sent some data but A has not actually read it, A's TCP layer may send RST to B to indicate that not all B's data was received properly. [RFC 1122](#) puts it this way:

If such a host issues a CLOSE call while received data is still pending in TCP, or if new data is received after CLOSE is called, its TCP SHOULD send a RST to show that data was lost.

If A's RST arrives at B before all of A's sent data has been processed by B's application, it is entirely possible that data sent by A will be lost, that is, will never be read by B.

In the BSD socket library, A can set the `SO_LINGER` option, which causes A's `close()` to block until A's data has been delivered to B (or until the `SO_LINGER` timeout, provided by the user when setting this option, has expired). However, `SO_LINGER` has no bearing on the issue above; post-close data from B to A will still cause A to send a RST.

In the simplex-talk program at [17.6 TCP simplex-talk](#), the client does not call `shutdown()` (it implicitly calls `close()` when it exits). When the client is done, the server calls `s.close()`. However, the fact that there is no data at all sent from the server to the client prevents the problem discussed above.

It is sometimes the case that A is expected to send a large amount of data to B and then exit:

```
byte[] bbuf = byte[1000000];
```

```
...
sout.write(bbuf);           // Java OutputStream attached to the socket s
s.close()
```

In this case, the `close()` call is supposed to result in A sending all the data before actually terminating the connection. [RFC 793](#) puts it this way:

Closing connections is intended to be a graceful operation in the sense that outstanding SENDs will be transmitted (and retransmitted), as flow control permits, until all have been serviced.

The Linux interpretation of this is given in the `socket(7)` man page:

When the socket is closed as part of `exit(2)`, it always lingers in the background.

The linger *time* is not specified. If there is an explicit `close(2)` before `exit(2)`, the `SO_LINGER` status above determines TCP's behavior.

Alternatively, A can send the data and then attempt to read from the socket. A will receive an end-of-file indication (typically 0 bytes read) as soon as the other endpoint B closes. If B waits to close until it has read all the data, this end-of-file indication will mean it is safe for A to call `s.close()`. However, B might equally well call `shutdown()` immediately on startup, as it does not intend to write any data, in which case A's received end-of-file is **not** an indication it is safe to close.

See also exercises 13.0 and 15.0.

17.9 Epilog

At this point we have covered the basic mechanics of TCP. The next chapter discusses, among other things, some of the subtle issues TCP must deal with in order to maintain reliability.

17.10 Exercises

Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises.

1.0. Experiment with the TCP version of simplex-talk. How does the server respond differently with threading enabled and without, if two simultaneous attempts to connect are made, from two different client instances?

2.0. Trace the states visited if nodes A and B attempt to create a TCP connection by *simultaneously* sending each other SYN packets, that then cross in the network. Draw the ladder diagram, and label the states on each side. Hint: there should be two pairs of crossing packets. A SYN+ACK counts, in the state diagram, as an ACK.

3.0. Suppose nodes A and B are each behind their own NAT firewall ([9.7 Network Address Translation](#)).

A — NAT_A — Internet — NAT_B — B

A and B attempt to connect to one another, using TCP; A uses source port 2000 and B uses 3000. A sends to the public IPv4 address of NAT_B, port 3000, and B sends to NAT_A, port 2000. Assume that neither NAT_A nor NAT_B changes the port numbers in outgoing packets, at least for the packets involved in this connection attempt.

(a). Suppose A sends a SYN packet to (NAT_B, 3000). It will be rejected at NAT_B, as the connection was not initiated by B. However, a short while later, B sends a SYN packet to (NAT_A, 2000). Explain why this second SYN packet *is* delivered to A.

(b). Now suppose A and B attempt to connect simultaneously, each sending a SYN to the other. Show that the connection succeeds.

4.0. When two nodes A and B simultaneously attempt to connect to one another using the OSI TP4 protocol, two bidirectional network connections are created (rather than one, as with TCP).

(a). Explain why this semantics is impossible with the existing TCP header. Hint: if a packet from $\langle A, \text{port1} \rangle$ arrives at $\langle B, \text{port2} \rangle$, how would the receiver tell to which of the two possible connections it belongs?

(b). Propose an additional field in the TCP header that would allow implementation of the TP4 semantics.

5.0. Simultaneous connection initiations are rare, but simultaneous connection termination is relatively common. How do two TCP nodes negotiate the simultaneous sending of FIN packets to one another? Draw the ladder diagram, and label the states on each side. Which node goes into TIMEWAIT state? Hint: there should be two pairs of crossing packets.

6.0. The state diagram at [17.8 TCP state diagram](#) shows a dashed path from FIN_WAIT_1 to TIMEWAIT on receipt of FIN+ACK. All FIN packets contain a valid ACK field, but that is not what is meant here. Under what circumstances is this direct arc from FIN_WAIT_1 to TIMEWAIT taken? Explain why this arc can never be used during simultaneous close. Hint: consider the ladder diagram of a “normal” close.

7.0. (a) Suppose you see multiple connections on your workstation in state FIN_WAIT_1. What is likely going on? Whose fault is it?

(b). What might be going on if you see connections languishing in state FIN_WAIT_2?

8.0. Suppose that, after downloading a file, the client host is unplugged from the network, so it can send no further packets. The server’s connection is still in the

ESTABLISHED state. In each case below, use the TCP state diagram to list all states that are reachable by the server.

- (a). Before being unplugged, the client was in state ESTABLISHED; *ie* it had *not* sent the first FIN.
- (b). Before being unplugged the client had sent its FIN, and moved to FIN_WAIT_1.

Eventually, the server connection here would in fact transition to CLOSED due to repeated timeouts. For this exercise, though, assume only transitions explicitly shown in the state diagram are allowed.

9.0. In [17.3 TCP Connection Establishment](#) we noted that RST packets had to have a valid SEQ value, but that “[RFC 793](#) does not require the RST packet’s ACK value to match”. There is an exception for RST packets arriving at state SYN-SENT: “the RST is acceptable if the ACK field acknowledges the SYN”. Explain the reasoning behind this exception.

10.0. Suppose A and B create a TCP connection with $ISN_A=20,000$ and $ISN_B=5,000$. A sends three 1000-byte packets (Data1, Data2 and Data3 below), and B ACKs each. Then B sends a 1000-byte packet DataB to A and terminates the connection with a FIN. In the table below, fill in the SEQ and ACK fields for each packet shown.

A sends	B sends
SYN, $ISN_A=20,000$	
	SYN, $ISN_B=5,000$, ACK=_____
ACK, SEQ=_____, ACK=_____	
Data1, SEQ=_____, ACK=_____	
	ACK, SEQ=_____, ACK=_____
Data2, SEQ=_____, ACK=_____	
	ACK, SEQ=_____, ACK=_____
Data3, SEQ=_____, ACK=_____	
	ACK, SEQ=_____, ACK=_____
	DataB, SEQ=_____, ACK=_____
ACK, SEQ=_____, ACK=_____	
	FIN, SEQ=_____, ACK=_____

11.0. Suppose you are downloading a large file, and there is a progress bar showing how much of the file has been downloaded. For definiteness, assume the progress bar moves 1 mm for each megabyte received by the application, and the throughput averages 0.5 MB per second (so the progress bar normally advances at a rate of 0.5 mm/sec).

You see the progress bar stop advancing for an interval of time. Suddenly, it jumps forward 5 mm, and then resumes its steady 0.5 mm/sec advance.

- (a). Explain the jump in terms of a lost packet and subsequent timeout and retransmission.

(b). Give an approximate value for the connection winsize, assuming only one packet was lost.

12.0. Suppose you are creating software for a streaming-video site. You want to limit the video read-ahead – the gap between how much has been downloaded and how much the viewer has actually watched – to approximately 1 MB; the server should pause in sending when necessary to enforce this. On the other hand, you do want the receiver to be able to read ahead by up to this much. You should assume that the TCP connection throughput will be higher than the actual video-data-consumption rate.

(a). Suppose the TCP window size happens to be exactly 1 MB. If the receiver simply reads each video frame from the TCP connection, displays it, and then pauses briefly before reading the next frame in accordance with the frame rate, explain how the flow-control mechanism of [18.10 TCP Flow Control](#) will achieve the desired effect.

(b). Applications, however, cannot control their TCP window size. Suppose the receiver application reads 1 MB ahead of what is being displayed, and then stops reading, until more can be displayed. Again, explain how the TCP flow-control mechanism will soon cause the sender to stop sending. (Specifically, the sender would never send more than $\text{winsize} + 1\text{MB}$ ahead of what was necessary.)

(It is also possible to implement sender pauses via an application-layer protocol.)

13.0. Modify the simplex-talk server of [17.6 TCP simplex-talk](#) so that `line_talker()` breaks out of the while loop as soon as it has printed the first string received (or simply remove the while loop). Once out of the while loop, the existing code calls `s.close()`.

(a). Start up the modified server, and connect to it with a client. Send a single message line, and use `netstat` to examine the TCP states of the client and server. What are these states?

(b). Send two message lines to the server. What are the TCP states of the client and server?

(c). Send three message lines to the server. Is there an error message at the client?

(d). Send two message lines to the server, while monitoring packets with [WireShark](#). The WireShark filter expression `tcp.port == 5431` may be useful for eliminating irrelevant traffic. What FIN packets do you see? Do you see a RST packet?

14.0. Outline a scenario in which TCP endpoint A sends data to B and then calls `close()` on its socket, and after the connection terminates B has not received all the data, even though the network has not failed. In the style of [17.6.1 The TCP Client](#), A's code might look like this:

```
s = new Socket(dest, destport);
sout = s.getOutputStream();
sout.write(large_buffer)
s.close()
```

18 TCP Issues and Alternatives

In this chapter we cover some issues relating to TCP reliability, some technical issues relating to TCP efficiency, and finally some outright alternatives to TCP.

18.1 TCP Old Duplicates

Conceptually, perhaps the most serious threat facing the integrity of TCP data is external old duplicates (16.3 Fundamental Transport Issues), that is, very late packets from a previous instance of the connection. Suppose a TCP connection is opened between A and B. One packet from A to B is duplicated and unduly delayed, with sequence number N. The connection is closed, and then another instance is reopened, that is, a connection is created using the same ports. At some point in the second connection, when an arriving packet with seq=N would be acceptable at B, the old duplicate shows up. Later, of course, B is likely to receive a seq=N packet from the new instance of the connection, but that packet will be seen by B as a duplicate (even though the data does not match), and (we will assume) be ignored.

For TCP, it is the actual sequence numbers, rather than the relative sequence numbers, that would have to match up. The diagram above ignores that.

As with TFTP, coming up with a possible scenario accounting for the generation of such a late packet is not easy. Nonetheless, many of the design details of TCP represent attempts to minimize this risk.

Solutions to the old-duplicates problem generally involve setting an upper bound on the lifetime of any packet, the MSL, as we shall see in the next section. T/TCP (18.5 TCP Faster Opening) introduced a connection-count field for this.

TCP is also vulnerable to sequence-number wraparound: arrival of an old duplicates from the *same* instance of the connection. However, if we take the MSL to be 60 seconds, sequence-number wrap requires sending 2^{32} bytes in 60 seconds, which requires a data-transfer rate in excess of 500 Mbps. TCP offers a fix for this (Protection Against Wrapped Segments, or PAWS), but it was introduced relatively late; we return to this in 18.4 Anomalous TCP scenarios.

18.2 TIMEWAIT

The TIMEWAIT state is entered by whichever side initiates the connection close; in the event of a simultaneous close, both sides enter TIMEWAIT. It is to last for a time $2 \times \text{MSL}$, where MSL = Maximum Segment Lifetime is an agreed-upon value for the maximum lifetime on the Internet of an IP packet. Traditionally MSL was taken to be 60 seconds,

but more modern implementations often assume 30 seconds (for a TIMEWAIT period of 60 seconds).

One function of TIMEWAIT is to solve the external-old-duplicates problem. TIMEWAIT requires that between closing and reopening a connection, a long enough interval must pass that any packets from the first instance will disappear. After the expiration of the TIMEWAIT interval, an old duplicate cannot arrive.

A second function of TIMEWAIT is to address the lost-final-ACK problem (16.3 Fundamental Transport Issues). If host A sends its final ACK to host B and this is lost, then B will eventually retransmit *its* final packet, which will be its FIN. As long as A remains in state TIMEWAIT, it can appropriately reply to a retransmitted FIN from B with a duplicate final ACK. As with TFTP, it is possible (though unlikely) for the final ACK to be lost as well as all the retransmitted final FINs sent during the TIMEWAIT period; should this happen, one side thinks the connection closed normally while the other side thinks it did not. See exercise 4.0.

TIMEWAIT only blocks reconnections for which both sides reuse the same port they used before. If A connects to B and closes the connection, A is free to connect again to B using a different port at A's end.

Conceptually, a host may have many old connections to the same port simultaneously in TIMEWAIT; the host must thus maintain for each of its ports a list of all the remote $\langle \text{IP_address}, \text{port} \rangle$ sockets currently in TIMEWAIT for that port. If a host is connecting as a client, this list likely will amount to a list of recently used ports; no port is likely to have been used twice within the TIMEWAIT interval. If a host is a server, however, accepting connections on a standardized port, and happens to be the side that initiates the active close and thus later goes into TIMEWAIT, then its TIMEWAIT list for that port can grow quite long.

Generally, busy servers prefer to be free from these bookkeeping requirements of TIMEWAIT, so many protocols are designed so that it is the client that initiates the active close. In the original HTTP protocol, version 1.0, the server sent back the data stream requested by the http GET message (17.7.1 netcat again), and indicated the end of this stream by closing the connection. In HTTP 1.1 this was fixed so that the client initiated the close; this required a new mechanism by which the server could indicate "I am done sending this file". HTTP 1.1 also used this new mechanism to allow the server to send back multiple files over one connection.

In an environment in which many short-lived connections are made from host A to the same port on server B, port exhaustion – having all ports tied up in TIMEWAIT – is a theoretical possibility. If A makes 1000 connections per second, then after 60 seconds it has gone through 60,000 available ports, and there are essentially none left. While this rate is high, early Berkeley-Unix TCP implementations often made only about 4,000

ports available to clients; with a 120-second TIMEWAIT interval, port exhaustion would occur with only 33 connections per second.

If you use `ssh` to connect to a server and then issue the `netstat -a` command on your own host (or, more conveniently, `netstat -a |grep -i tcp`), you should see your connection in ESTABLISHED state. If you close your connection and check again, your connection should be in TIMEWAIT.

18.3 The Three-Way Handshake Revisited

As stated earlier in [17.3 TCP Connection Establishment](#), both sides choose an ISN; actual sequence numbers are the sum of the sender's ISN and the relative sequence number. There are two original reasons for this mechanism, and one later one ([18.3.1 ISNs and spoofing](#)). The original TCP specification, as clarified in [RFC 1122](#), called for the ISN to be determined by a special **clock**, incremented by 1 every 4 microseconds.

The most basic reason for using ISNs is to detect duplicate SYNs. Suppose A initiates a connection to B by sending a SYN packet. B replies with SYN+ACK, but this is lost. A then times out and retransmits its SYN. B now receives A's second SYN while in state SYN_RECEIVED. Does this represent an entirely new request (perhaps A has suddenly restarted), or is it a duplicate? If A uses the clock-driven ISN strategy, B can tell (*almost* certainly) whether A's second SYN is new or a duplicate: only in the latter case will the ISN values in the two SYNs match.

While there is no danger to data integrity if A sends a SYN, restarts, and sends the SYN again as part of a reopening the same connection, the arrival of a second SYN with a new ISN means that the original connection cannot proceed, because that ISN is now wrong. The receiver of the duplicate SYN should drop any connection state it has recorded so far, and restart processing the second SYN from scratch.

The clock-driven ISN also originally added a second layer of protection against external old duplicates. Suppose that A opens a connection to B, and chooses a clock-based ISN N_1 . A then transfers M bytes of data, closed the connection, and reopens it with ISN N_2 . If $N_1 + M < N_2$, then the old-duplicates problem *cannot occur*: all of the absolute sequence numbers used in the first instance of the connection are less than or equal to $N_1 + M$, and all of the absolute sequence numbers used in the second instance will be greater than N_2 .

Early Berkeley–Unix implementations of the socket library often allowed a second connection meeting the above ISN requirement to be reopened *before* TIMEWAIT would have expired; this potentially addressed the problem of port exhaustion. We might call this **TIMEWAIT connection reuse**. Of course, if the first instance of the connection transferred data faster than the ISN clock rate, that is at more than 250,000 bytes/sec, then $N_1 + M$ would be greater than N_2 , and TIMEWAIT would have to be enforced. But in

the era in which TCP was first developed, sustained transfers exceeding 250,000 bytes/sec were not as common. Alternatively, the connection in TIMEWAIT might allow *incoming* connections that reuse the same ports, because in this case the host in TIMEWAIT can *choose* its own ISN to be greater than the final absolute sequence number of the previous instance of the connection. This second alternative is allowed by :rfc:1192:, §4.2.2.13.

The three-way handshake was extensively analyzed by Dalal and Sunshine in [DS78]. The authors noted that with a two-way handshake, the second side receives no confirmation that its ISN was correctly received. The authors also observed that a four-way handshake – in which the ACK of ISN_A is sent separately from ISN_B , as in the diagram below – could fail if one side restarted.

For this failure to occur, assume that after sending the SYN in line 1, with ISN_{A1} , A restarts. The ACK in line 2 is either ignored or not received. B now sends its SYN in line 3, but A interprets this as a new connection request; it will respond after line 4 by sending a fifth, SYN packet containing a different ISN_{A2} . For B the connection is now ESTABLISHED, and if B acknowledges this fifth packet but fails to update its record of A's ISN, the connection will fail as A and B would have different notions of ISN_A .

18.3.1 ISNs and spoofing

The clock-based ISN proved to have a significant weakness: it often allowed an attacker to guess the ISN a remote host might use. It did not help any that an early version of Berkeley Unix, instead of incrementing the ISN 250,000 times a second, incremented it once a second, by 250,000 (plus something for each connection). By guessing the ISN a remote host would choose, an attacker might be able to mimic a local, trusted host, and thus gain privileged access.

Specifically, suppose host A trusts its neighbor B, and executes with privileged status commands sent by B; this situation was typical in the era of the *rhost* command. A authenticates these commands because the connection comes from B's IP address. The bad guy, M, wants to send packets to A so as to *pretend* to be B, and thus get a privileged command invoked. The connection only needs to be *started*; if the ruse is discovered after the command is executed, it is too late. M can easily send a SYN packet to A with B's IP address in the source-IP field; M can probably temporarily disable B too, so that A's SYN-ACK response, which is sent to B, goes unnoticed. What is harder is for M to figure out how to guess how to ACK ISN_A . But if A generates ISNs with a slowly incrementing clock, M can guess the pattern of the clock with previous connection attempts, and can thus guess ISN_A with a considerable degree of accuracy. So M sends SYN to A with B as source, A sends SYN-ACK to B containing ISN_A , and M *guesses* this value and sends ACK(ISN_A+1) to A, again with B listed in the IP header as source, followed by a single-packet command.

This TCP-layer IP-spoofing technique was first described by Robert T Morris in [\[RTM85\]](#); Morris went on to launch the [Internet Worm of 1988](#) using unrelated attacks. The IP-spoofing technique was used in the 1994 Christmas Day attack against UCSD, launched from Loyola's own `apollo.it.luc.edu`; the attack was associated with [Kevin Mitnick](#) though apparently not actually carried out by him. Mitnick was arrested a few months later.

[RFC 1948](#), in May 1996, introduced a technique for introducing a degree of randomization in ISN selection, while still ensuring that the same ISN would not be used twice in a row for the same connection. The ISN is to be the sum of the 4-μs clock, $C(t)$, and a secure hash of the connection information as follows (compare with the local-port algorithm of [17.7 TCP and bind\(\)](#)):

$$\text{ISN} = C(t) + \text{hash}(\text{local_addr}, \text{local_port}, \text{remote_addr}, \text{remote_port}, \text{secret_key})$$

The `secret_key` value is a random value chosen by the host on startup. While M, above, can poll A for its current ISN, and can probably guess the hash function and the first four parameters above, without knowing the key it cannot determine (or easily guess) the ISN value A would have sent to B. Legitimate connections between A and B using the same port at each end, on the other hand, see the ISN increasing at the 4-μs rate, which potentially increases the chance of successful application of “TIMEWAIT connection reuse” as in [18.3 The Three-Way Handshake Revisited](#).

[RFC 5925](#) addresses spoofing and related attacks by introducing an optional TCP authentication mechanism: the TCP header includes an option containing a secure hash ([28.6 Secure Hashes](#)) of the rest of the TCP header and a shared secret key. The need for key management limits when this mechanism can be used; the classic use case is BGP connections between routers ([15 Border Gateway Protocol \(BGP\)](#)).

Another approach to the prevention of spoofing attacks is to ask sites and ISPs to refuse to forward outwards any IP packets with a source address not from within that site or ISP. If an attacker's ISP implements this, the attacker will be unable to launch spoofing attacks against the outside world. A concrete proposal can be found in [RFC 2827](#). Unfortunately, it has been (as of 2015) almost entirely ignored.

See also the discussion of SYN flooding at [17.3 TCP Connection Establishment](#), although that attack does not involve ISN manipulation.

18.4 Anomalous TCP scenarios

TCP, like any transport protocol, must address the transport issues in [16.3 Fundamental Transport Issues](#).

As we saw above, TCP addresses the Duplicate Connection Request (Duplicate SYN) issue by noting whether the ISN has changed. This is handled at the kernel level by TCP, versus TFTP's application-level (and rather desultory) approach to handing Duplicate RRs.

TCP addresses Loss of Final ACK through TIMEWAIT: as long as the TIMEWAIT period has not expired, if the final ACK is lost and the other side resends its final FIN, TCP will still be able to reissue that final ACK. TIMEWAIT in this sense serves a similar function to TFTP's DALLY state.

External Old Duplicates, arriving as part of a previous instance of the connection, are prevented by TIMEWAIT, and may also be prevented by the use of a clock-driven ISN.

Internal Old Duplicates, from the *same* instance of the connection, that is, sequence number wraparound, is only an issue for bandwidths exceeding 500 Mbps: only at bandwidths above that can 4 GB be sent in one 60-second MSL. TCP implementations now address this with PAWS: Protection Against Wrapped Segments ([RFC 1323](#)). PAWS adds a 32-bit "timestamp option" to the TCP header. The granularity of the timestamp clock is left unspecified; one tick must be small enough that sequence numbers cannot wrap in that interval (*eg* less than 3 seconds for 10,000 Mbps), and large enough that the timestamps cannot wrap in time MSL. On Linux systems the timestamp clock granularity is typically 1 to 10 ms; measurements on the author's systems have been 4 ms. With timestamps, an old duplicate due to sequence-number wraparound can now easily be detected.

The PAWS mechanism also requires ACK packets to echo back the sender's timestamp, in addition to including their own. This allows senders to accurately measure round-trip times.

Reboots are a potential problem as the host presumably has no record of what aborted connections need to remain in TIMEWAIT. TCP addresses this on paper by requiring hosts to implement Quiet Time on Startup: no new connections are to be accepted for $1 \times \text{MSL}$. No known implementations actually do this; instead, they assume that the restarting process itself will take at least one MSL. This is no longer as certain as it once was, but serious consequences have not ensued.

18.5 TCP Faster Opening

If a client wants to connect to a server, send a request and receive an immediate reply, TCP mandates one full RTT for the three-way handshake before data can be delivered. This makes TCP one RTT slower than UDP-based request-reply protocols. There have been periodic calls to allow TCP clients to include data with the first SYN packet *and* have it be delivered immediately upon arrival – this is known as **accelerated open**.

If there will be a series of requests and replies, the simplest way to deliver the data without a handshake delay is to **pipeline** all the requests and replies over one persistent connection; the handshake delay then applies only to the first request. If the pipeline connection is idle for a long-enough interval, it may be closed, and then reopened later if necessary.

An early accelerated–open proposal was **T/TCP**, or TCP for Transactions, specified in [RFC 1644](#). T/TCP introduced a **connection count** TCP option, called CC; each participant would include a 32–bit CC value in its SYN; each participant’s own CC values were to be monotonically increasing. Accelerated open was allowed if the server side had the client’s previous CC in a cache, and the new CC value was strictly greater than this cached value. This ensured that the new SYN was not a duplicate of an older SYN.

Unfortunately, this also bypasses the duplicate–SYN detection and modest validation of the client’s IP address provided by the full three–way handshake, worsening the spoofing problem of [18.3.1 ISNs and spoofing](#). If malicious host M wants to pretend to be B when sending a privileged request to A, all M has to do is send a single SYN+Data packet with an extremely large value for CC. Generally, the accelerated open succeeded as long as the CC value presented was larger than the value A had cached for B; it did not have to be larger by exactly 1.

18.5.1 TCP Fast Open

The TCP Fast Open (TFO) mechanism, described in [RFC 7413](#), involves a secure “cookie” sent by the client as a TCP option; if a SYN+Data packet has a valid cookie, then the client has satisfactorily established its identity and the data may be released immediately to the receiving application.

Cookies can be either 4 or 16 bytes (probably, though not necessarily, corresponding to IPv4 and IPv6), and are requested by the client through a previous TCP handshake with a cookie–request option in the SYN packet. If a client includes a still–valid cookie in the SYN packet of a subsequent connection, the data accompanying that SYN packet is immediately released by the server to the application; the three–way handshake still completes but the data does not wait for it.

The same cookie can be reused multiple times. Cookies do have an expiration time, though, and also they are specific to the client IP address (though not to the TCP ports used). One implementation option is for the server to use encryption (not hashing) of the client IP address; in this model the cookie expires when the encryption key expires.

Because cookies must be requested ahead of time, TCP Fast Open is not fundamentally faster than the connection–pipeline option above, except that holding a TCP connection open uses more resources than simply storing a cookie. One likely application for TCP Fast Open is in accessing web servers. Web clients and servers already keep a persistent connection open for a while, but often “a while” here amounts only to several seconds; TCP Fast Open cookies could remain active for much longer. Another potential use is for TCP–based DNS queries, for which there is no established mechanism for connection reuse.

A serious practical problem with TCP Fast Open is that some middleboxes ([9.7.2 Middleboxes](#)) remove TCP options they do not understand, or even block the

connection attempt entirely. One consequence of this is that clients attempting to use TFO must log failures, and not attempt to reuse TFO again (at least for an appropriate time interval).

Also, SYN flooding attacks are still possible; for example, a large number of compromised clients can obtain cookies legitimately, and then each reuse their cookie many times in short order. Alternatively, the cookie from *one* client can be distributed to a large number of other hosts which then spoof the original client's IP address. To minimize the impact of such attacks, TFO requires that the fast-open option be ignored if the number of pending fast opens exceeds a given threshold. Connections would still open normally, but data would not be delivered to the server application until the three-way handshake completed.

Finally, TFO does introduce a small possibility of duplicate data delivery. Consider, for example, the following sequence:

1. Then client sends a SYN with valid TFO cookie, and some data
2. The ACK from the server is lost, or is never sent
3. The server processes the data
4. The server reboots
5. The client times out, and retransmits its SYN+cookie+data, which arrives at the server

The duplicate data arriving in the final step will be again processed by the server. This is not *likely*, but if either the client or the server cannot handle the possibility of duplication, then TFO should not be used. In particular, it must be possible to enable or disable TFO on a per-connection basis. Of course, if the request conveyed by the SYN+data is idempotent ([16.5.2 Sun RPC](#)), the duplication should not matter.

An alternative duplicate-data-delivery scenario involves the client sending a SYN+cookie+data packet and closing the connection. The client, then, goes into TIMEWAIT, but not the server. Meanwhile, the SYN+cookie+data packet somehow gets duplicated within the network, and this duplicate arrives at the server. This scenario (unlike the first) is not prevented by arranging for the server's cookie-generation key to become invalid after a reboot.

18.6 Path MTU Discovery

TCP connections are more efficient if they can keep large packets flowing between the endpoints. Once upon a time, TCP endpoints included just 512 bytes of data in each packet that was not destined for local delivery, to avoid fragmentation. TCP endpoints now typically engage in **Path MTU Discovery** which almost always allows them to send larger packets; backbone ISPs are now usually able to carry 1500-byte packets. The **Path MTU** is the largest packet size that can be sent along a path without fragmentation.

The IPv4 strategy is to send an initial data packet with the IPv4 DONT_FRAG bit set. If the ICMP message Frag_Required/DONT_FRAG_Set comes back, or if the packet times out, the sender tries a smaller size. If the sender receives a TCP ACK for the packet, on the other hand, indicating that it made it through to the other end, it might try a larger size. Usually, the size range of 512–1500 bytes is covered by less than a dozen discrete values; the point is not to find the exact Path MTU but to determine a reasonable approximation rapidly.

IPv6 has no DONT_FRAG bit. Path MTU Discovery over IPv6 involves the periodic sending of larger packets; if the ICMPv6 message Packet Too Big is received, a smaller packet size must be used. [RFC 1981](#) has details.

18.7 TCP Sliding Windows

TCP implements sliding windows, in order to improve throughput. Window sizes are measured in terms of bytes rather than packets; this leaves TCP free to packetize the data in whatever segment size it elects. In the initial three-way handshake, each side specifies the maximum window size it is willing to accept, in the **Window Size** field of the TCP header. This 16-bit field can only go to 64 kB, and a $1 \text{ Gbps} \times 100 \text{ ms}$ bandwidth \times delay product is 12 MB; as a result, there is a TCP **Window Scale** option that can also be negotiated in the opening handshake. The scale option specifies a power of 2 that is to be multiplied by the actual Window Size value. In the WireShark example above, the client specified a Window Size field of 5888 ($= 4 \times 1472$) in the third packet, but with a Window Scale value of $2^6 = 64$ in the first packet, for an effective window size of $64 \times 5888 = 256$ segments of 1472 bytes. The server side specified a window size of 5792 and a scaling factor of $2^5 = 32$.

TCP may either transmit a bulk stream of data, using sliding windows fully, or it may send slowly generated interactive data; in the latter case, TCP may never have even one full segment outstanding.

In the following chapter we will see that a sender frequently reduces the actual TCP window size, in order to avoid congestion; the window size included in the TCP header is known as the **Advertised Window Size**. On startup, TCP does not send a full window all at once; it uses a mechanism called “slow start”.

18.8 TCP Delayed ACKs

TCP receivers are allowed briefly to delay their ACK responses to new data. This offers perhaps the most benefit for interactive applications that exchange small packets, such as ssh and telnet. If A sends a data packet to B and expects an immediate response, delaying B’s ACK allows the receiving *application* on B time to wake up and generate that application-level response, which can then be sent together with B’s ACK. Without delayed ACKs, the kernel layer on B may send its ACK before the receiving application on

B has even been scheduled to run. If response packets are small, that doubles the total traffic. The maximum ACK delay is 500 ms, according to [RFC 1122](#) and [RFC 2581](#), though 200 ms is more common.

For bulk traffic, delayed ACKs simply mean that the ACK traffic volume is reduced. Because ACKs are cumulative, one ACK from the receiver can in principle acknowledge multiple data packets from the sender. Unfortunately, acknowledging too many data packets with one ACK can interfere with the self-clocking aspect of sliding windows; the arrival of that ACK will then trigger a burst of additional data packets, which would otherwise have been transmitted at regular intervals. Because of this, the RFCs above specify that an ACK be sent, at a minimum, for every other data packet. For a discussion of how the sender should respond to delayed ACKs, see [19.2.1 TCP Reno Per-ACK Responses](#).

The TCP ACK-delay time can usually be adjusted globally as a system parameter. Linux offers a `TCP_QUICKACK` option, as a flag to `setsockopt()`, to disable delayed ACKs on a per-connection basis, but only until the next TCP system call (including reads and writes). It must be invoked immediately after every receive operation to disable delayed ACKs entirely. This option is also not very portable.

The TSO option of [17.5 TCP Offloading](#), used at the receiver, can also reduce the number of ACKs sent. If every two arriving data packets are consolidated via TSO into a single packet, then the receiver will appear to the sender to be acknowledging every other data packet. The ACK delay introduced by TSO is, however, usually quite small.

18.9 Nagle Algorithm

Like delayed ACKs, the Nagle algorithm ([RFC 896](#)) also attempts to improve the behavior of interactive small-packet applications. It specifies that a TCP endpoint generating small data segments – segments of less than the maximum size – should queue them until either it accumulates a full segment’s worth or receives an ACK for all the previously sent packets (small or not). If the full-segment threshold is not reached at the sender, this means that only one segment will be sent per RTT, containing all the data generated during that RTT.

Bandwidth Conservation

Delayed ACKs and the Nagle algorithm both originated in a bygone era, when bandwidth was in much shorter supply than it is today. In [RFC 896](#), John Nagle writes (in 1984, well before TCP Reno, [19 TCP Reno and Congestion Management](#)) “In general, we have not been able to afford the luxury of excess long-haul bandwidth that the ARPANET possesses, and our long-haul links are heavily loaded during peak periods. Transit times of several seconds are thus common in our network.” Today, it is unlikely that a modest number of small packets would cause detectable, let alone significant, problems. That said, abandoning the Nagle algorithm has the potential to

unleash onto the Internet backbone large numbers of small, mostly-header packets; [MM01] suggests “it would be a mistake to stop using it.”

As an example, suppose A wishes to send to B packets containing consecutive letters, starting with “a”. The application on A generates these every 100 ms, but the RTT is 501 ms. At $T=0$, A transmits “a”. The application on A continues to generate “b”, “c”, “d”, “e” and “f” at times 100 ms through 500 ms, but A does not send them immediately. At $T=501$ ms, ACK(“a”) arrives; at this point A transmits its backlogged “bcdef”. The ACK for this arrives at $T=1002$, by which point A has queued “ghijk”. The end result is that A sends a fifth as many separate packets as it would without the Nagle algorithm. If these letters are generated by a user typing them with telnet, and the ACKs also include the echoed responses, then if the user pauses the echoed responses will very soon catch up.

The Nagle algorithm does not always interact well with delayed ACKs. If an application generates a 2 KB transaction that is divided between a full-sized packet and a followup small packet, then the Nagle algorithm means that the second packet cannot be sent until the first is acknowledged. However, a receiver using delayed ACKs may wait up to 500 ms to send the ACK that allows that second packet to be sent. This delays the entire transaction by the delayed-ACK time. Worse, this may happen to every one of a lengthy *series* of transactions. Internet Draft [A Proposed Modification to Nagle’s Algorithm](#) addresses this by, in effect, always allowing senders to send *one* small packet without receiving an acknowledgment, to compensate for the possibility that a delayed-ACK receiver will need to receive that one small packet before it sends its ACK. More specifically, the modification is to forbid the sending of small packets as long as there is an earlier unacknowledged *small* packet outstanding, versus the original rule forbidding the sending of small packets as long as there are *any* earlier unacknowledged packets, small or not.

For other examples, see exercises 1.0 and 2.0; the first is an example of how the Nagle algorithm can have surprising user-interface consequences. The Nagle algorithm can usually be disabled on a per-connection basis, in the BSD socket library by calling `setsockopt()` with the `TCP_NODELAY` flag.

18.10 TCP Flow Control

It is possible for a TCP sender to send data faster than the receiver can process it. When this happens, a TCP receiver may reduce the advertised Window Size value of an open connection, thus informing the sender to switch to a smaller window size. This provides support for **flow control**.

The window-size reduction appears in the ACKs sent back by the receiver. A given ACK is not supposed to reduce the window size by so much that the upper end of the window gets smaller. A window might shrink from the byte range [20,000..28,000] to [22,000..28,000] but never to [20,000..26,000].

If a TCP receiver uses this technique to shrink the advertised window size to 0, this means that the sender may not send data. The receiver has thus informed the sender that, yes, the data was received, but that, no, more may not yet be sent. This corresponds to the ACK_{WAIT} suggested in [8.1.3 Flow Control](#). Eventually, when the receiver is ready to receive data, it will send an ACK increasing the advertised window size again.

If the TCP sender has its window size reduced to 0, and the ACK from the receiver increasing the window is lost, then the connection would be deadlocked. TCP has a special feature specifically to avoid this: if the window size is reduced to zero, the sender sends dataless packets to the receiver, at regular intervals. Each of these “polling” packets elicits the receiver’s current ACK; the end result is that the sender will receive the eventual window-enlargement announcement reliably. These “polling” packets are regulated by the so-called **persist** timer.

18.11 Silly Window Syndrome

The silly-window syndrome is a term for a scenario in which TCP transfers only small amounts of data at a time. Because TCP/IP packets have a minimum fixed header size of 40 bytes, sending small packets uses the network inefficiently. The silly-window syndrome can occur when either by the receiving application consuming data slowly or when the sending application generating data slowly.

As an example involving a slow-consuming receiver, suppose a TCP connection has a window size of 1000 bytes, but the receiving application consumes data only 10 bytes at a time, at intervals about equal to the RTT. The following can then happen:

- The sender sends bytes 1–1000. The receiving application consumes 10 bytes, numbered 1–10. The receiving TCP buffers the remaining 990 bytes and sends an ACK reducing the window size to 10, per [18.10 TCP Flow Control](#).
- Upon receipt of the ACK, the sender sends 10 bytes numbered 1001–1010, the most it is permitted. In the meantime, the receiving application has consumed bytes 11–20. The window size therefore remains at 10 in the next ACK.
- the sender sends bytes 1011–1020 while the application consumes bytes 21–30. The window size remains at 10.

The sender may end up sending 10 bytes at a time indefinitely. This is of no benefit to either side; the sender might as well send larger packets less often. The standard fix, set forth in [RFC 1122](#), is for the receiver to use its ACKs to keep the window at 0 until it has consumed one full packet’s worth (or half the window, for small window sizes). At that point the sender is invited – by an appropriate window-size advertisement in the next ACK – to send another full packet of data.

The silly-window syndrome can also occur if the sender is *generating* data slowly, say 10 bytes at a time. The Nagle algorithm, above, can be used to prevent this, though for

interactive applications sending small amounts of data in separate but closely spaced packets may actually be useful.

18.12 TCP Timeout and Retransmission

When TCP sends a packet containing user data (this excludes ACK-only packets), it sets a timeout. If that timeout expires before the packet data is acknowledged, it is retransmitted. Acknowledgments are sent for every arriving data packet (unless Delayed ACKs are implemented, [18.8 TCP Delayed ACKs](#)); this amounts to receiver-side retransmit-on-duplicate of [8.1.1 Packet Loss](#). Because ACKs are cumulative, and so a later ACK can replace an earlier one, lost ACKs are seldom a problem.

For TCP to work well for both intra-server-room and trans-global connections, with RTTs ranging from well under 1 ms to close to 1 second, the length of the timeout interval must *adapt*. TCP manages this by maintaining a running estimate of the RTT, EstRTT. In the original version, TCP then set $\text{Timeout} = 2 \times \text{EstRTT}$ (in the literature, the TCP Timeout value is often known as RTO, for Retransmission Timeout). EstRTT itself was a running average of periodically measured SampleRTT values, according to

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

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

There is a potential RTT measurement ambiguity: if a packet is sent twice, the ACK received could be in response to the first transmission or the second. The Karn/Partridge algorithm resolves this: on packet loss (and retransmission), the sender

- Doubles Timeout
- Stops recording SampleRTT
- Uses the doubled Timeout as EstRTT when things resume

Setting $\text{Timeout} = 2 \times \text{EstRTT}$ proved too short during congestion periods and too long other times. Jacobson and Karels ([\[JK88\]](#)) introduced a way of calculating the Timeout value based on the statistical variability of EstRTT. After each SampleRTT value was collected, the sender would also update EstDeviation according to

$$\begin{aligned} \text{SampleDev} &= | \text{SampleRTT} - \text{EstRTT} | \\ \text{EstDeviation} &= \beta \times \text{EstDeviation} + (1 - \beta) \times \text{SampleDev} \end{aligned}$$

for a fixed β , $0 < \beta < 1$. Timeout was then set to $\text{EstRTT} + 4 \times \text{EstDeviation}$. EstDeviation is an estimate of the so-called *mean deviation*; 4 mean deviations corresponds (for normally distributed data) to about 5 *standard* deviations. If the SampleRTT values were normally distributed (which they are not), this would mean that the chance that a non-lost packet would arrive outside the Timeout period is vanishingly small.

For further details, see [\[JK88\]](#) and [\[AP99\]](#).

In most implementations, a TCP sender maintains just one retransmission timer no matter how many packets are outstanding. Here is the recommended timer-management algorithm, from [RFC 6298](#):

- If a packet is sent and the timer is not running, restart it for time Timeout.
- If an ACK arrives that acknowledges all outstanding data, turn off the timer.
- If an ACK arrives that acknowledges new data, but not all outstanding data, reset the timer to time Timeout.

If the sender transmits a steady stream of packets, none of which is lost, the last clause will ensure that the timer never fires, which is as desired. However, if a series of earlier ACKs arrives slowly, but just fast enough to keep resetting the timer, a lost packet may not time out until some multiple of Timeout has elapsed; while not ideal, this is not considered serious. See exercise 6.0.

18.13 KeepAlive

There is no reason that a TCP connection should not be idle for a long period of time; ssh/telnet connections, for example, might go unused for days. However, there is the turned-off-at-night problem: a workstation might telnet into a server, and then be shut off (not shut down gracefully) at the end of the day. The connection would now be half-open, but the server would not generate any traffic and so might never detect this; the connection itself would continue to tie up resources.

KeepAlive in action

One evening long ago, when dialed up (yes, that long ago) into the Internet, my phone line disconnected while I was typing an email message in an ssh window. I dutifully reconnected, expecting to find my message in the file “dead.letter”, which is what would have happened had I been disconnected while using the even-older tty dialup. Alas, nothing was there. I reconstructed my email as best I could and logged off.

The next morning, there was my lost email in a file “dead.letter”, dated two hours after the initial crash! What had happened, apparently, was that the original ssh connection on the server side just hung there, half-open. Then, after two hours, KeepAlive kicked in, and aborted the connection. At that point ssh sent my mail program the HangUp signal, and the mail program wrote out what it had in “dead.letter”.

To avoid this, TCP supports an optional **KeepAlive** mechanism: each side “polls” the other with a dataless packet. The original [RFC 1122](#) KeepAlive timeout was 2 hours, but this could be reduced to 15 minutes. If a connection failed the KeepAlive test, it would be closed.

Supposedly, some TCP implementations are not exactly [RFC 1122](#)-compliant: either KeepAlives are enabled by default, or the KeepAlive interval is much smaller than called for in the specification.

18.14 TCP timers

To summarize, TCP maintains the following four kinds of timers. All of them can be maintained by a single timer list, above.

- **TimeOut**: a per-segment timer; TimeOut values vary widely
- **2×MSL TIMEWAIT**: a per-connection timer
- **Persist**: the timer used to poll the receiving end when `winsize = 0`
- **KeepAlive**, above

18.15 Variants and Alternatives

One alternative to TCP is UDP with programmer-implemented timeout and retransmission; many RPC implementations ([16.5 Remote Procedure Call \(RPC\)](#)) do exactly this, with reasonable results. Within a LAN a static timeout of around half a second usually works quite well (unless the LAN has some tunneled links), and implementation of a simple timeout-retransmission mechanism is quite straightforward, although implementing adaptive timeouts as in [18.12 TCP Timeout and Retransmission](#) is a bit more complex. QUIC ([16.1.1 QUIC](#)) is an example of this strategy.

We here consider four other protocols. The first, MPTCP, is based on TCP itself. The second, SCTP, is a message-oriented alternative to TCP that is an entirely separate protocol. The last two, DCCP and QUIC, are attempts to create a TCP-like transport layer on top of UDP.

18.15.1 MPTCP

Multipath TCP, or MPTCP, allows connections to use multiple network interfaces on a host, either sequentially or simultaneously. MPTCP architectural principles are outlined in [RFC 6182](#); implementation details are in [RFC 6824](#).

To carry the actual traffic, MPTCP arranges for the creation of multiple standard-TCP **subflows** between the sending and receiving hosts; these subflows typically connect between different pairs of IP addresses on the respective hosts.

For example, a connection to a server can start using the client's wired Ethernet interface, and continue via Wi-Fi after the user has unplugged. If the client then moves out of Wi-Fi range, the connection might continue via a mobile network. Alternatively, MPTCP allows the parallel use of multiple Ethernet interfaces on both client and server for higher throughput.

MPTCP officially forbids the creation of multiple TCP connections between a single pair of interfaces in order to simulate Highspeed TCP (22.5 Highspeed TCP); [RFC 6356](#) spells out an MWTCP congestion-control algorithm to enforce this.

Suppose host A, with two interfaces with IP addresses A_1 and A_2 , wishes to connect to host B with IP addresses B_1 and B_2 . Connection establishment proceeds via the ordinary TCP three-way handshake, between one of A's IP addresses, say A_1 , and one of B's, B_1 . The SYN packets must each carry the MP_CAPABLE TCP option, to signal one another that MPTCP is supported. As part of the MP_CAPABLE option, A and B also exchange pseudorandom 64-bit connection keys, sent unencrypted; these will be used to sign later messages as in 28.6.1 Secure Hashes and Authentication. This first connection is the initial subflow.

Once the MPTCP initial subflow has been established, additional subflow connections can be made. Usually these will be initiated from the client side, here A, though the B side can also do this. At this point, however, A does not know of B's address B_2 , so the only possible second subflow will be from A_2 to B_1 . New subflows will carry the MP_JOIN option with their initial SYN packets, along with digital signatures signed by the original connection keys verifying that the new subflow is indeed part of this MPTCP connection.

At this point A and B can send data to one another using both connections simultaneously. To keep track of data, each side maintains a 64-bit data sequence number, DSN, for the data it sends; each side also maintains a mapping between the DSN and the subflow sequence numbers. For example, A might send 1000-byte blocks of data alternating between the A_1 and A_2 connections; the blocks might have DSN values 10000, 11000, 12000, 13000, The A_1 subflow would then carry blocks 10000, 12000, *etc*, numbering these consecutively (perhaps 20000, 21000, ...) with its own sequence numbers. The sides exchange DSN mapping information with a DSS TCP option. This mechanism means that all data transmitted over the MWTCP connection can be delivered in the proper order, and that if one subflow fails, its data can be retransmitted on another subflow.

B can inform A of its second IP address, B_2 , using the ADD_ADDR option. Of course, it is possible that B_2 is not directly reachable by A; for example, it might be behind a NAT router. But if B_2 is reachable, A can now open two more subflows A_1 — B_2 and A_2 — B_2 .

All the above works equally well if either or both of A's addresses is behind a NAT router, simply because the NAT router is able to properly forward the subflow TCP connections. Addresses sent from one host to another, such as B's transmission of its address B_2 , may be rendered invalid by NAT, but in this case A's attempt to open a connection to B_2 simply fails.

Generally, hosts can be configured to use multiple subflows in parallel, or to use one interface only as a backup, when the primary interface is unplugged or out of range. APIs

have been proposed that allow an control over MPTCP behavior on a per-connection basis.

18.15.2 SCTP

The Stream Control Transmission Protocol, SCTP, is an entirely separate protocol from TCP, running directly above IP. It is, in effect, a message-oriented alternative to TCP: an application writes a sequence of messages and SCTP delivers each one as a unit, fragmenting and reassembling it as necessary. Like TCP, SCTP is connection-oriented and reliable. SCTP uses a form of sliding windows, and, like TCP, adjusts the window size to manage congestion.

An SCTP connection can support multiple **message streams**; the exact number is negotiated at startup. A retransmission delay in one stream never blocks delivery in other streams. Within each stream, SCTP messages are sequentially numbered, and are normally delivered in order of message number. A receiver can request, however, to receive messages immediately upon successful delivery, that is, potentially out of order. Either way, the data within each message is guaranteed to be delivered in order and without loss.

Internally, message data is divided into SCTP **chunks** for inclusion in packets. One SCTP packet can contain data chunks from different messages and different streams; packets can also contain control chunks.

Messages themselves can be quite large; there is no set limit. Very large messages may need to be received in multiple system calls (*eg* calls to `recvmsg()`).

SCTP supports an MPTCP-like feature by which each endpoint can use multiple network interfaces.

SCTP connections are set up using a four-way handshake, versus TCP's three-way handshake. The extra packet provides some protection against so-called SYN flooding (17.3 TCP Connection Establishment). The central idea is that if client A initiates a connection request with server B, then B allocates no resources to the connection until after B has received a response to its own message to A. This means that, at a minimum, A is a real host with a real IP address.

The full four-way handshake between client A and server B is, in brief, as follows:

- A sends B an INIT chunk (corresponding to SYN), along with a pseudorandom Tag_A .
- B sends A an INIT ACK, with Tag_B and a **state cookie**. The state cookie contains all the information B needs to allocate resources to the connection, and is digitally signed (28.6.1 Secure Hashes and Authentication) with a key known only to B. Crucially, B does **not** at this point allocate any resources to the incipient connection.

- A returns the state cookie to B in a `COOKIE ECHO` packet.
- B enters the `ESTABLISHED` state and sends a `COOKIE ACK` to A. Upon receipt, A enters the `ESTABLISHED` state.

When B receives the `COOKIE ECHO`, it verifies the signature. At this point B knows that it sent the cookie to A and received a response, so A must exist. Only then does B allocate memory resources to the connection. Spoofed INITs in the first step cost B essentially nothing.

The `TagA` and `TagB` in the first two packets are called **verification tags**. From this point on, B will include `TagA` in every packet it sends to A, and vice-versa. Although these tags are sent unencrypted, they nonetheless make it much harder for an attacker to inject data into the connection.

Data can be included in the third and fourth packets above; *i.e.* A can begin sending data after one RTT.

Unfortunately for potential SCTP applications, few if any NAT routers recognize SCTP; this limits the use of SCTP to Internet paths along which NAT is not used. In principle SCTP could simplify delivery of web pages, transmitting one page component per message, but lack of NAT support makes this infeasible. SCTP is also blocked by some middleboxes (9.7.2 Middleboxes) on the grounds that it is an unknown protocol, and therefore suspect. While this is not quite as common as the NAT problem, it is common enough to prevent by itself the widespread adoption of SCTP in the general Internet. SCTP *is* widely used for telecommunications signaling, both within and between providers, where NAT and recalcitrant middleboxes can be banished.

18.15.3 DCCP

As we saw in 16.1.2 DCCP, DCCP is a UDP-based transport protocol that supports, among other things, connection establishment. While it is used much less often than TCP, it provides an alternative example of how transport can be done.

DCCP defines a set of distinct packet types, rather than TCP's independent packet flags; this disallows unforeseen combinations such as TCP SYN+RST. Connection establishment involves Request and Respond; data transmission involves Data, ACK and DataACK, and teardown involves CloseReq, Close and Reset. While one cannot have, for example, a Respond+ACK, Respond packets do carry an acknowledgment field.

Like TCP, DCCP uses a three-way handshake to open a connection; here is a diagram:

The `OPEN` state corresponds to TCP's `ESTABLISHED` state. Like TCP, each side chooses an ISN (not shown in the diagram). Because packet delivery is not reliable, and because ACKs are not cumulative, the client remains in `PARTOPEN` state until it has confirmed

that the server has received its ACK of the server's Response. While in state PARTOPEN, the client can send ACK and DataACK but not ACK-less Data packets.

Packets are numbered sequentially. The numbering includes all packets, not just Data packets, and is by packet rather than by byte.

The DCCP state diagram is shown below. It is simpler than the TCP state diagram because DCCP does not support simultaneous opens.

To close a connection, one side sends Close and the other responds with Reset. Reset is used for normal close as well as for exceptional conditions. Because whoever sends the Close is then stuck with TIMEWAIT, the server side may send CloseReq to ask the client to send Close.

There are also two special packet formats, Sync and SyncAck, for resynchronizing sequence numbers after a burst of lost packets.

The other major TCP-like feature supported by DCCP is congestion control; see [21.3.3 DCCP Congestion Control](#).

18.15.4 QUIC Revisited

Like DCCP, QUIC (see also [16.1.1 QUIC](#)) is a UDP-based transport protocol, aimed rather squarely at HTTP plus TLS ([29.5.2 TLS](#)). The fundamental goal of QUIC is to provide TLS encryption protection with as little overhead as possible, in a manner that competes fairly with TCP in the presence of congestion. Opening a QUIC connection, encryption included, takes a single RTT. QUIC can also be seen, however, as a complete rewrite of TCP from the ground up; a reading of specific features sheds quite a bit of light on how the corresponding TCP features have fared over the past thirty-odd years. As of 2021, QUIC was finally edited to RFC status:

- [RFC 8999](#): Version-independent Properties of QUIC
- [RFC 9000](#): QUIC: A UDP-Based Multiplexed and Secure Transport (good overview)
- [RFC 9001](#): Using TLS to Secure QUIC
- [RFC 9002](#): QUIC Loss Detection and Congestion Control
- [RFC 9114](#): HTTP/3 (which uses QUIC)

The move to base HTTP/3 on QUIC began officially in 2018; [RFC 9114](#) was published in June 2022. QUIC's standardization may represent the beginning of the end for TCP, given that most Internet connections are for HTTP or HTTPS. Still, many TCP design issues ([19 TCP Reno and Congestion Management](#), [20 Dynamics of TCP](#), [22 Newer TCP Implementations](#)) carry over very naturally to QUIC; a shift from TCP to QUIC should best be viewed as evolutionary. (And, by the same token, the 1995 standardization of

IPv6 presumably represents the beginning of the end for IPv4, but that was over 25 years ago.)

The design of QUIC was influenced by the fate of SCTP above; the latter, as a new protocol above IP, was often blocked by overly security-conscious middleboxes (9.7.2 Middleboxes).

The fact that the QUIC layer resides within an application (or within a library) rather than within the kernel has meant that QUIC is able to evolve much faster than TCP. The long-term consequences of having the transport layer live outside the kernel are not yet completely clear, however; it may, for example, make it easier for users to install unfair congestion-management schemes.

18.15.4.1 Headers

We will start with the QUIC header. While there are some alternative forms, the basic header is diagrammed below, with a 1-byte Type field, an 8-byte Connection ID, and 4-byte Version and Packet Number fields.

Perhaps the most striking thing about this header is that 4-byte alignment – used consistently in the IPv4, IPv6, UDP and TCP headers – has been completely abandoned. On most contemporary processors, the performance advantages of alignment are negligible; see the last paragraph at 9.1 The IPv4 Header.

IP packets are identified as such by the Ethernet type field, and TCP and UDP packets are identified as such by the IPv4-header Protocol field. But QUIC packets are *not* identified as such by any flag in the preceding IP or UDP headers; there is in fact no place in those headers for a QUIC marker to go. QUIC appears to an observer as just another form of UDP traffic. This acts as a form of middlebox defense; QUIC packets cannot be identified as such in isolation. WireShark, sidebar below, identifies QUIC packets by looking at the whole history of the connection, and even then must make some (educated) guesses. Middleboxes could do that too, but it would take work.

The initial Connection ID consists of 64 random bits chosen by the client. The server, upon accepting the connection, may change the Connection ID; at that point the Connection ID is fixed for the lifetime of the connection. The Connection ID may be omitted for packets whose connection can be determined from the associated IP address and port values; this is signaled by the Type field. The Connection ID can also be used to migrate a connection to a different IP address and port, as might happen if a mobile device moves out of range of Wi-Fi and the mobile-data plan continues the communication. This may also happen if a connection passes through a NAT router. The NAT forwarding entry may time out (see the comment on UDP and inactivity at 9.7 Network Address Translation), and the connection may be assigned a different

outbound UDP port if it later resumes. QUIC uses the Connection ID to recognize that the reassigned connection is still the same one as before.

The Version field gets dropped as soon as the version is negotiated. As part of the version negotiation, a packet might have multiple version fields. Such packets put a *random* value into the low-order seven bits of the Type field, as a prevention against middleboxes' blocking unknown types. This way, aggressive middlebox behavior should be discovered early, before it becomes widespread.

QUIC-watching

QUIC packets can be observed in [WireShark](#) by using the filter string "quic". To generate QUIC traffic, use a [Chromium-based browser](#) and go to a Google-operated site, say, [google.com](#). Often the only non-encrypted fields are the Type field and the packet number. It may be necessary to enable QUIC in the browser, done in Chrome via `chrome://flags`; see also `chrome://net-internals`.

The packet number can be reduced to one or two bytes once the connection is established; this is signaled by the Type field. Internally, QUIC uses packet numbers in the range 0 to 2^{62} ; these internal numbers are not allowed to wrap around. The low-order 32 bits (or 16 bits or 8 bits) of the internal number are what is transmitted in the packet header. A packet receiver infers the high-order bits from the most recent acknowledgment.

The initial packet number is to be chosen randomly in the range 0 to $2^{32}-1025$. This corresponds to TCP's use of Initial Sequence Numbers.

Use of 16-bit or 8-bit transmitted packet numbers is restricted to cases where there can be no ambiguity. At a minimum, this means that the number of outstanding packets (the QUIC winsize) cannot exceed 2^7-1 for 8-bit packet numbering or $2^{15}-1$ for 16-bit packet numbering. These maximum winsizes represent the ideal case where there is no packet reordering; smaller values are likely to be used in practice. (See [8.5 Exercises](#), exercise 9.0.)

18.15.4.2 Frames and streams

Data in a QUIC packet is partitioned into one or more **frames**. Each frame's data is prefixed by a simple frame header indicating its length and type. Some frames contain management information; frames containing higher-layer data are called STREAM frames. Each frame must be fully contained in one packet.

The application's data can be divided into multiple **streams**, depending on the application requirements. This is particularly useful with HTTP, as a client may request a large number of different resources (html, images, javascript, *etc*) simultaneously. Stream data is contained in STREAM frames. Streams are numbered, with Stream 0

reserved for the TLS cryptographic handshake. The HTTP/2 protocol has introduced its own notion of streams; these map neatly onto QUIC streams.

The two low-order bits of each stream number indicate whether the stream was initiated by the client or by the server, and whether it is bi- or uni-directional. This design decision means that either side can create a stream and send data on it immediately, *without negotiation*; this is important for reducing unnecessary RTTs.

Each individual stream is guaranteed in-order delivery, but there are no ordering guarantees between different streams. Within a packet, the data for a particular stream is contained in a frame for that stream.

One packet can contain stream frames for multiple streams. However, if a packet is lost, streams that have frames contained in that packet are blocked until retransmission. Other streams can continue without interruption. This creates an incentive for keeping separate streams in separate packets.

Stream frames contain the byte offset of the frame's block of stream data (starting from 0), to enable in-order stream reassembly. TCP, as we have seen, uses this byte-numbering approach exclusively, though starting with the Initial Sequence Number rather than zero. QUIC's stream-level numbering by byte is unrelated to its top-level numbering by packet.

In addition to stream frames, there are a large number of management frames. Here are a few of them:

- RST_STREAM: like TCP RST, but for one stream only.
- MAX_DATA: this corresponds to the TCP advertised window size. As with TCP, it can be reduced to zero to pause the flow of data and thereby implement flow control. There is also a similar MAX_STREAM_DATA, applying per stream.
- PING and PONG: to verify that the other endpoint is still responding. These serve as the equivalent of TCP KEEPALIVES, among other things.
- CONNECTION_CLOSE and APPLICATION_CLOSE: these initiate termination of the connection; they differ only in that a CONNECTION_CLOSE might be accompanied by a QUIC-layer error or explanation message while an APPLICATION_CLOSE might be accompanied by, say, an HTTP error/explanation message.
- PAD: to pad out the packet to a larger size.
- ACK: for acknowledgments, below.

18.15.4.3 Acknowledgments

QUIC assigns a new, sequential packet number (the Packet ID) to every packet, including retransmissions. TCP, by comparison, assigns sequence numbers to each byte. (QUIC stream frames do number data by byte, as noted above.)

Lost QUIC packets are retransmitted, but with a new packet number. This makes it impossible for a receiver to send cumulative acknowledgments, as lost packets will never be acknowledged. The receiver handles this as below. At the sender side, the sender maintains a list of packets it has sent that are both unacknowledged and also not known to be lost. These represent the packets **in flight**. When a packet is retransmitted, its old packet number is removed from this list, as lost, and the new packet number replaces it.

To the extent possible given this retransmission–renumbering policy, QUIC follows the spirit of sliding windows. It maintains a state variable `bytes_in_flight`, corresponding to TCP's `winsize`, listing the total size of all the packets in flight. As with TCP, new acknowledgments allow new transmissions.

Acknowledgments themselves are sent in special acknowledgment frames. These begin with the number of the highest packet received. This is followed by a list of pairs, as long as will fit into the frame, consisting of the length of the next block of contiguous packets received followed by the length of the intervening gap of packets *not* received. The TCP Selective ACK (19.6 [Selective Acknowledgments \(SACK\)](#)) option is similar, but is limited to three blocks of received packets. It is quite possible that some of the gaps in a QUIC ACK frame refer to lost packets that were long since retransmitted with new packet numbers, but this does not matter.

The sender is allowed to skip packet numbers occasionally, to prevent the receiver from trying to increase throughput by acknowledging packets not yet received. Unlike with TCP, acknowledging an unsent packet is considered to be a fatal error, and the connection is terminated.

As with TCP, there is a delayed–ACK timer, but, while TCP's is typically 250 ms, QUIC's is 25 ms. QUIC also includes in each ACK frame the receiver's best estimate of the elapsed time between arrival of the most recent packet and the sending of the ACK it triggered; this allows the sender to better estimate the RTT. The primary advantage of the design decision not to reuse packet IDs is that there is never any ambiguity as to a retransmitted packet's RTT, as there is in TCP (18.12 [TCP Timeout and Retransmission](#)). Note, however, that because QUIC runs in a user process and not the kernel, it may not be able to respond immediately to an arriving packet, and so the time–delay estimate may be slightly short.

ACK frames are not themselves acknowledged. This means that, in a one–way data flow, the receiver may have no idea if its ACKs are getting through (a TCP receiver may be in the same situation). The QUIC receiver may send a PING frame to the sender, which will respond not only with a matching PONG frame but also an ACK frame acknowledging the receiver's recent acknowledgment packets.

QUIC adjusts its `bytes_in_flight` value to manage congestion, much as TCP manages its `winsize` (or more properly its `cwnd`, 19 [TCP Reno and Congestion Management](#)) for the same purpose. Specifically, QUIC attempts to mimic the congestion response of TCP

Cubic, [22.15 TCP CUBIC](#), and so should in theory compete fairly with TCP Cubic connections. However, it is straightforward to arrange for QUIC to model the behavior of any other flavor of TCP ([22 Newer TCP Implementations](#)).

18.15.4.4 Connection handshake and TLS encryption

The opening of a QUIC connection makes use of the TLS handshake, [29.5.2 TLS](#), specifically TLS v1.3, [29.5.2.4.3 TLS version 1.3](#). A client wishing to connect sends a QUIC Initial packet, containing the TLS ClientHello message. The server responds (with a ServerHello) in a QUIC Handshake packet. (There is also a Retry packet, for special situations.) The TLS negotiation is contained in QUIC's Stream 0. While the TLS and QUIC handshake rules are rather precise, there is as yet no formal state-diagram description of connection opening.

The Initial packet also contains a set of QUIC **transport parameters** declared unilaterally by the client; the server makes a similar declaration in its response. These parameters include, among other things, the maximum packet size, the connection's idle timeout, and initial value for MAX_DATA, above.

An important feature of TLS v1.3 is that, if the client has connected to the server previously and still has the key negotiated in that earlier session, it can use that old key to send an encrypted application-layer request (in a STREAM frame) immediately following the Initial packet. This is called **0-RTT** protection (or encryption). The advantage of this is that the client may receive an answer from the server within a single RTT, versus four RTTs for traditional TCP (one for the TCP three-way handshake, two for TLS negotiation, and one for the application request/reply). As discussed at [29.5.2.4.4 TLS v1.3 0-RTT mode](#), requests submitted with 0-RTT protection must be idempotent, to prevent replay attacks.

Once the server's first Handshake packet makes it back to the client, the client is in possession of the key negotiated by the new session, and will encrypt everything using that going forward. This is known as the **1-RTT** key, and all further data is said to be 1-RTT protected. The negotiated key is initially calculated by the TLS layer, which then exports it to QUIC. The QUIC layer then encrypts the entire data portion of its packets, using the format of [RFC 5116](#).

The QUIC header is not encrypted, but is still covered by an authentication checksum, making it impossible for middleboxes to rewrite anything. Such rewriting has been observed for TCP, and has sometimes complicated TCP evolution.

The type field of a QUIC packet contains a special code to mark 0-RTT data, ensuring that the receiver will know what level of protection is in effect.

When a QUIC server receives the ClientHello and sends off its ServerHello, it has not yet received any evidence that the client "owns" the IP address it claims to have; that is, that the client is not spoofing its IP address ([18.3.1 ISNs and spoofing](#)). Because of the

idempotency restriction on responses to 0-RTT data, the server cannot give away privileges if spoofed in this way by a client. The server may, however, be an unwitting participant in a **traffic-amplification** attack, if the real client can trigger the sending by the server to a spoofed client of a larger response than the real client sends directly. The solution here is to require that the QUIC Initial packet, containing the ClientHello, be at least 1200 bytes. The server's Handshake response is likely to be smaller, and so represents no amplification of traffic.

To close the connection, one side sends a CONNECTION_CLOSE or APPLICATION_CLOSE. It may continue to send these in response to packets from the other side. When the other side receives the CLOSE packet, it should send its own, and then enter the so-called **draining** state. When the initiator of the close receives the other side's echoed CLOSE, it too will enter the draining state. Once in this state, an endpoint may not send any packets. The draining state corresponds to TCP's TIMEWAIT (18.2 TIMEWAIT), for the purpose of any lost final ACKs; it should last three RTT's. There is no need of a TIMEWAIT analog to prevent old duplicates, as a second QUIC connection will select a new Connection ID.

QUIC connection closing has no analog of TCP's feature in which one side sends FIN and the other continues to send data indefinitely, 17.8.1 Closing a connection. This use of FIN, however, is allowed in bidirectional streams; the per-stream (and per-direction) FIN bit lives in the stream header. Alternatively, one side can send its request and close its stream, and the other side can then answer on a different stream.

18.16 Epilog

At this point we have covered the basic mechanics of TCP, but have one important topic remaining: how TCP manages its window size so as to limit congestion, while maintaining fairness. This turns out to be complex, and will be the focus of the next three chapters.

18.17 Exercises

Exercises may be given fractional (floating point) numbers, to allow for interpolation of new exercises.

1.0. A user moves the computer mouse and sees the mouse-cursor's position updated on the screen. Suppose the mouse-position updates are being transmitted over a TCP connection with a relatively long RTT. The user attempts to move the cursor to a specific point. How will the user perceive the mouse's motion

- (a). with the Nagle algorithm
- (b). without the Nagle algorithm

2.0. Host A sends two single-byte packets, one containing “x” and the other containing “y”, to host B. A implements the Nagle algorithm and B implements delayed ACKs, with a 500 ms maximum delay. The RTT is negligible. How long does the transmission take? Draw a ladder diagram.

3.0. Suppose you have fallen in with a group that wants to add to TCP a feature so that, if A and B1 are connected, then B1 can **hand off** its connection to a different host B2; the end result is that A and B2 are connected and A has received an uninterrupted stream of data. Either A or B1 can initiate the handoff.

(a). Suppose B1 is the host to send the final FIN (or HANDOFF) packet to A. How would you handle appropriate analogues of the TIMEWAIT state for host B1? Does the fact that A is continuing the connection, just not with B1, matter?

(b). Now suppose A is the party to send the final FIN/HANDOFF, to B1. What changes to TIMEWAIT would have to be made at A’s end? Note that A may potentially hand off the connection again and again, *eg* to B3, B4 and then B5.

4.0. Suppose A connects to B via TCP, and sends the message “Attack at noon”, followed by FIN. Upon receiving this, B is sure it has received the entire message.

(a). What can A be sure of upon receiving B’s own FIN+ACK?

(b). What can B be sure of upon receiving A’s final ACK?

(c). What is A not absolutely sure of after sending its final ACK?

5.0. Host A connects to the Internet via Wi-Fi, receiving IPv4 address 10.0.0.2, and then opens a TCP connection *conn1* to remote host B. After *conn1* is established, A’s Ethernet cable is plugged in. A’s Ethernet interface receives IP address 10.0.0.3, and A automatically selects this new Ethernet connection as its default route. **Assume** that A now starts using 10.0.0.3 as the source address of packets it sends as part of *conn1* (contrary to [RFC 1122](#)).

Assume also that A’s TCP implementation is such that when a packet arrives from $\langle B_{IP}, B_{port} \rangle$ to $\langle A_{IP}, A_{port} \rangle$ and this socketpair is to be matched to an existing TCP connection, the field A_{IP} is allowed to be any of A’s IP addresses (that is, either 10.0.0.2 or 10.0.0.3); it does not have to match the IP address with which the connection was originally negotiated.

(a). Explain why *conn1* will now fail, as soon as any packet is sent from A. Hint: the packet will be sent from 10.0.0.3. What will B send in response? In light of the second assumption, how will A react to B’s response packet?

(The author regularly sees connections fail this way. Perhaps some justification for this behavior is that, at the time of establishment of *conn1*, A was not yet multihomed.)

(b). Now suppose all four fields of the socketpair ($\langle B_{IP}, B_{port} \rangle$, $\langle A_{IP}, A_{port} \rangle$) are used to match an incoming packet to its corresponding TCP connection. The connection *conn1* still fails, though not as immediately. Explain what happens.

See also [10.2.5 ARP and multihomed hosts](#), [9 IP version 4](#) exercise 4.0, and [13 Routing-Update Algorithms](#) exercise 16.0.

6.0. Draw a ladder diagram showing a lost packet transmitted at time T , and yet the retransmission timer does not go off until at least $T + 3 \cdot \text{Timeout}$ (there is nothing special about 3 here). Assume that the algorithm of [18.12 TCP Timeout and Retransmission](#) is used. Hint: show a series of ACKs for *previous* packets arriving at intervals of just under Timeout, causing a series of resets of the timer.
