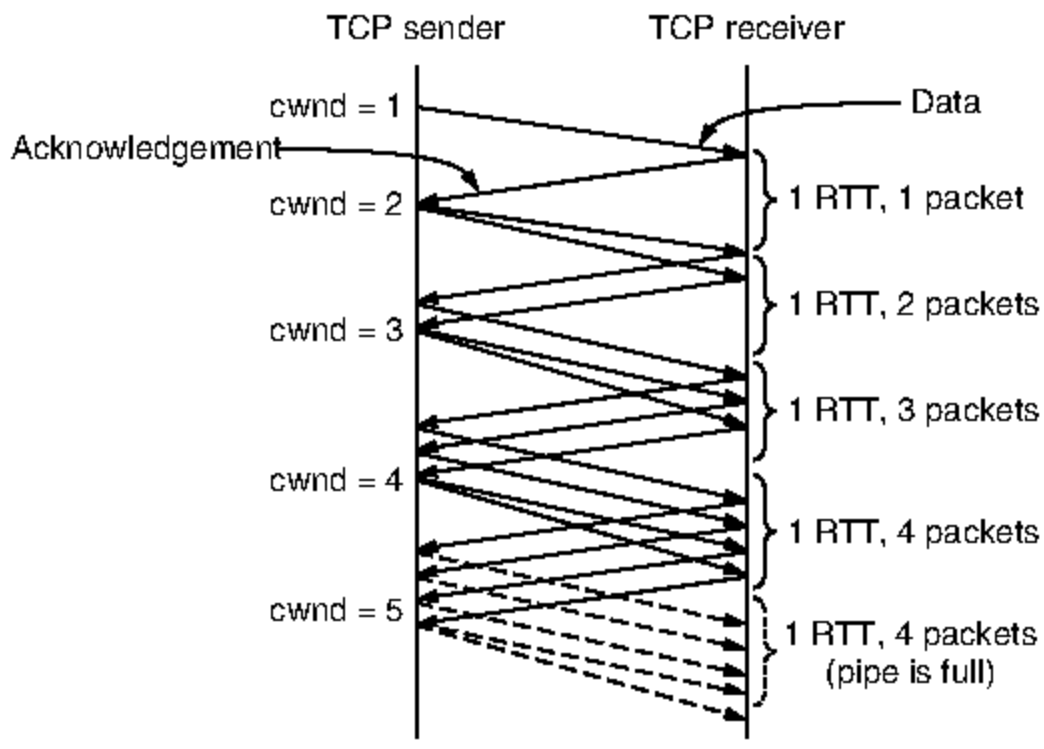

Figure 6-45. Additive increase from an initial congestion window of one segment.

acknowledgements that return to the sender. These acknowledgements bear the same acknowledgement number. They are called **duplicate acknowledgements**. Each time the sender receives a duplicate acknowledgement, it is likely that another packet has arrived at the receiver and the lost packet still has not shown up.

Because packets can take different paths through the network, they can arrive out of order. This will trigger duplicate acknowledgements even though no packets have been lost. However, this is uncommon in the Internet much of the time. When there is reordering across multiple paths, the received packets are usually not reordered too much. Thus, TCP somewhat arbitrarily assumes that three duplicate acknowledgements imply that a packet has been lost. The identity of the lost packet can be inferred from the acknowledgement number as well. It is the very next packet in sequence. This packet can then be retransmitted right away, before the retransmission timeout fires.

This heuristic is called **fast retransmission**. After it fires, the slow start threshold is still set to half the current congestion window, just as with a timeout. Slow start can be restarted by setting the congestion window to one packet. With this window size, a new packet will be sent after the one round-trip time that it takes to acknowledge the retransmitted packet along with all data that had been sent before the loss was detected.

An illustration of the congestion algorithm we have built up so far is shown in Fig. 6-46. This version of TCP is called TCP Tahoe after the 4.2BSD Tahoe release in 1988 in which it was included. The maximum segment size here is 1 KB. Initially, the congestion window was 64 KB, but a timeout occurred, so the threshold is set to 32 KB and the congestion window to 1 KB for transmission 0. The congestion window grows exponentially until it hits the threshold (32 KB). The

window is increased every time a new acknowledgement arrives rather than continuously, which leads to the discrete staircase pattern. After the threshold is passed, the window grows linearly. It is increased by one segment every RTT.

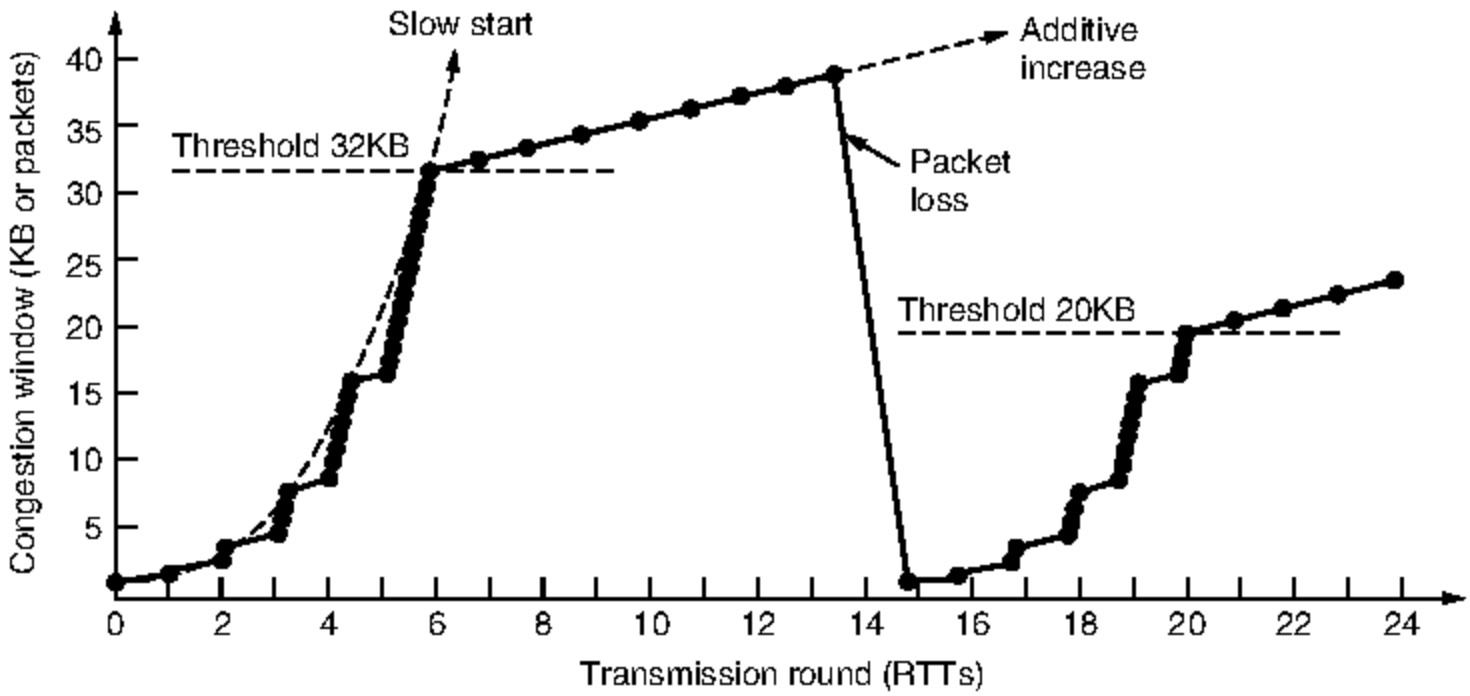

Figure 6-46. Slow start followed by additive increase in TCP Tahoe.

The transmissions in round 13 are unlucky (they should have known), and one of them is lost in the network. This is detected when three duplicate acknowledgements arrive. At that time, the lost packet is retransmitted, the threshold is set to half the current window (by now 40 KB, so half is 20 KB), and slow start is initiated all over again. Restarting with a congestion window of one packet takes one round-trip time for all of the previously transmitted data to leave the network and be acknowledged, including the retransmitted packet. The congestion window grows with slow start as it did previously, until it reaches the new threshold of 20 KB. At that time, the growth becomes linear again. It will continue in this fashion until another packet loss is detected via duplicate acknowledgements or a timeout (or the receiver's window becomes the limit).

TCP Tahoe (which included good retransmission timers) provided a working congestion control algorithm that solved the problem of congestion collapse. Jacobson realized that it is possible to do even better. At the time of the fast retransmission, the connection is running with a congestion window that is too large, but it is still running with a working ack clock. Every time another duplicate acknowledgement arrives, it is likely that another packet has left the network. Using duplicate acknowledgements to count the packets in the network, makes it possible to let some packets exit the network and continue to send a new packet for each additional duplicate acknowledgement.

Fast recovery is the heuristic that implements this behavior. It is a temporary mode that aims to maintain the ack clock running with a congestion window that is the new threshold, or half the value of the congestion window at the time of the

fast retransmission. To do this, duplicate acknowledgements are counted (including the three that triggered fast retransmission) until the number of packets in the network has fallen to the new threshold. This takes about half a round-trip time. From then on, a new packet can be sent for each duplicate acknowledgement that is received. One round-trip time after the fast retransmission, the lost packet will have been acknowledged. At that time, the stream of duplicate acknowledgements will cease and fast recovery mode will be exited. The congestion window will be set to the new slow start threshold and grows by linear increase.

The upshot of this heuristic is that TCP avoids slow start, except when the connection is first started and when a timeout occurs. The latter can still happen when more than one packet is lost and fast retransmission does not recover adequately. Instead of repeated slow starts, the congestion window of a running connection follows a **sawtooth** pattern of additive increase (by one segment every RTT) and multiplicative decrease (by half in one RTT). This is exactly the AIMD rule that we sought to implement.

This sawtooth behavior is shown in Fig. 6-47. It is produced by TCP Reno, named after the 4.3BSD Reno release in 1990 in which it was included. TCP Reno is essentially TCP Tahoe plus fast recovery. After an initial slow start, the congestion window climbs linearly until a packet loss is detected by duplicate acknowledgements. The lost packet is retransmitted and fast recovery is used to keep the ack clock running until the retransmission is acknowledged. At that time, the congestion window is resumed from the new slow start threshold, rather than from 1. This behavior continues indefinitely, and the connection spends most of the time with its congestion window close to the optimum value of the bandwidth-delay product.

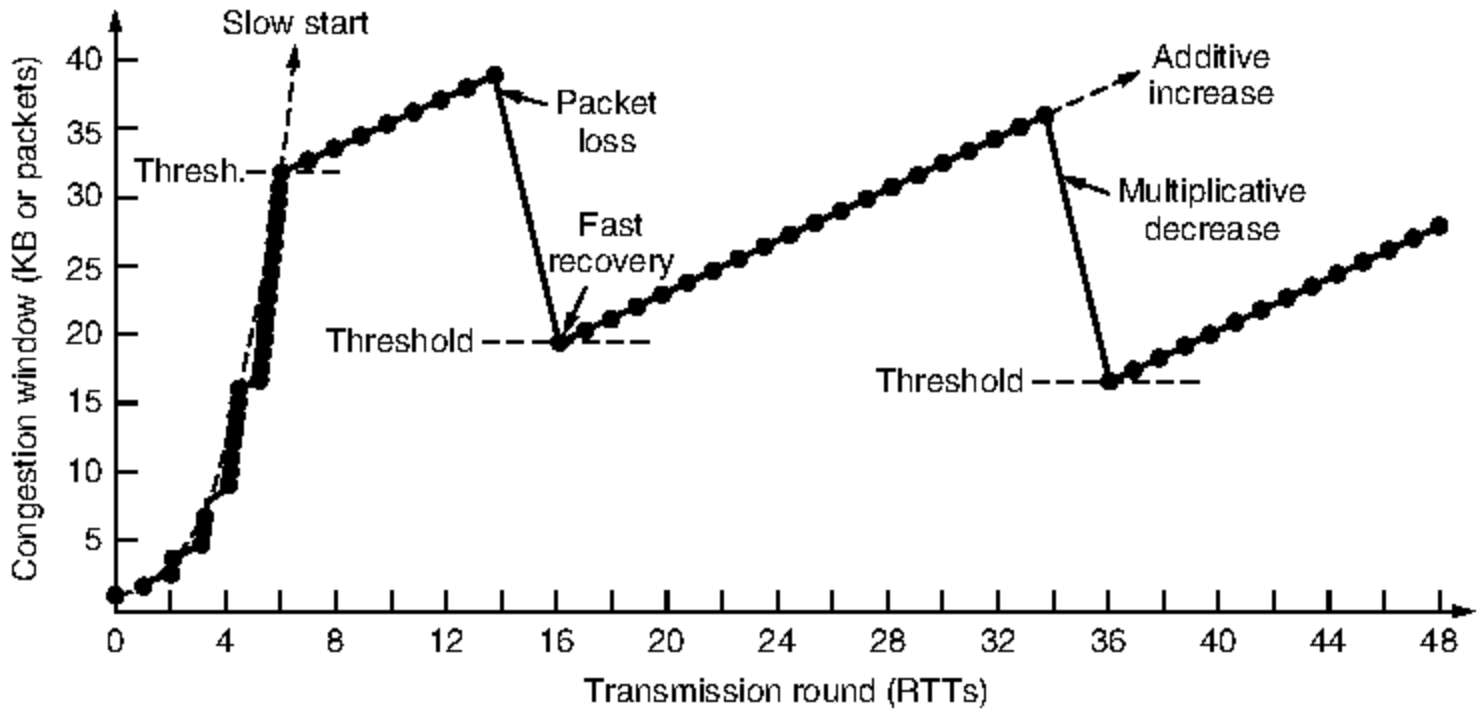

Figure 6-47. Fast recovery and the sawtooth pattern of TCP Reno.

TCP Reno with its mechanisms for adjusting the congestion window has formed the basis for TCP congestion control for more than two decades. Most of

the changes in the intervening years have adjusted these mechanisms in minor ways, for example, by changing the choices of the initial window and removing various ambiguities. Some improvements have been made for recovering from two or more losses in a window of packets. For example, the TCP NewReno version uses a partial advance of the acknowledgement number after a retransmission to find and repair another loss (Hoe, 1996), as described in RFC 3782. Since the mid-1990s, several variations have emerged that follow the principles we have described but use slightly different control laws. For example, Linux uses a variant called CUBIC TCP (Ha et al., 2008) and Windows includes a variant called Compound TCP (Tan et al., 2006).

Two larger changes have also affected TCP implementations. First, much of the complexity of TCP comes from inferring from a stream of duplicate acknowledgements which packets have arrived and which packets have been lost. The cumulative acknowledgement number does not provide this information. A simple fix is the use of **SACK** (**Selective ACKnowledgements**), which lists up to three ranges of bytes that have been received. With this information, the sender can more directly decide what packets to retransmit and track the packets in flight to implement the congestion window.

When the sender and receiver set up a connection, they each send the *SACK permitted* TCP option to signal that they understand selective acknowledgements. Once SACK is enabled for a connection, it works as shown in Fig. 6-48. A receiver uses the TCP *Acknowledgement number* field in the normal manner, as a cumulative acknowledgement of the highest in-order byte that has been received. When it receives packet 3 out of order (because packet 2 was lost), it sends a *SACK option* for the received data along with the (duplicate) cumulative acknowledgement for packet 1. The *SACK option* gives the byte ranges that have been received above the number given by the cumulative acknowledgement. The first range is the packet that triggered the duplicate acknowledgement. The next ranges, if present, are older blocks. Up to three ranges are commonly used. By the time packet 6 is received, two SACK byte ranges are used to indicate that packet 6 and packets 3 to 4 have been received, in addition to all packets up to packet 1. From the information in each *SACK option* that it receives, the sender can decide which packets to retransmit. In this case, retransmitting packets 2 and 5 would be a good idea.

SACK is strictly advisory information. The actual detection of loss using duplicate acknowledgements and adjustments to the congestion window proceed just as before. However, with SACK, TCP can recover more easily from situations in which multiple packets are lost at roughly the same time, since the TCP sender knows which packets have not been received. SACK is now widely deployed. It is described in RFC 2883, and TCP congestion control using SACK is described in RFC 3517.

The second change is the use of ECN (Explicit Congestion Notification) in addition to packet loss as a congestion signal. ECN is an IP layer mechanism to

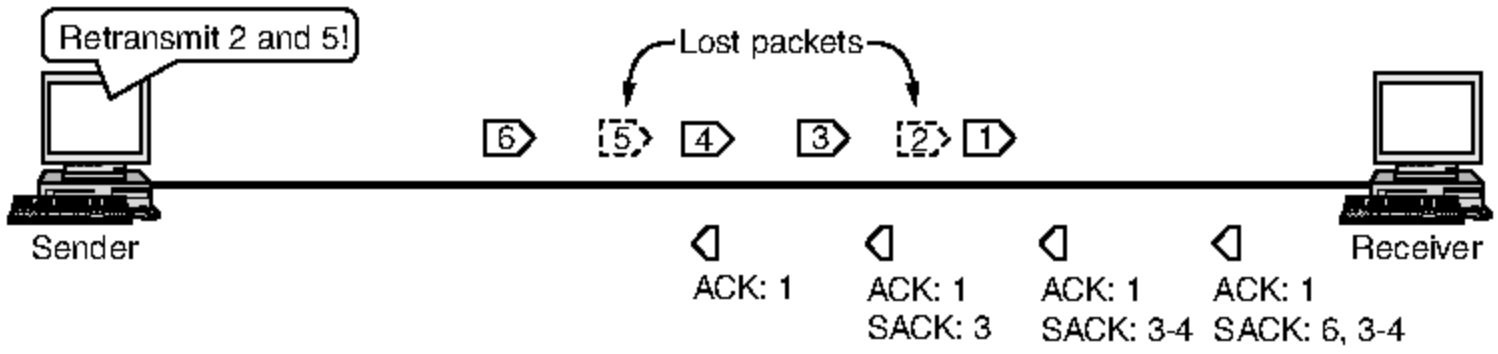

Figure 6-48. Selective acknowledgements.

notify hosts of congestion that we described in Sec. 5.3.4. With it, the TCP receiver can receive congestion signals from IP.

The use of ECN is enabled for a TCP connection when both the sender and receiver indicate that they are capable of using ECN by setting the *ECE* and *CWR* bits during connection establishment. If ECN is used, each packet that carries a TCP segment is flagged in the IP header to show that it can carry an ECN signal. Routers that support ECN will set a congestion signal on packets that can carry ECN flags when congestion is approaching, instead of dropping those packets after congestion has occurred.

The TCP receiver is informed if any packet that arrives carries an ECN congestion signal. The receiver then uses the *ECE* (ECN Echo) flag to signal the TCP sender that its packets have experienced congestion. The sender tells the receiver that it has heard the signal by using the *CWR* (Congestion Window Reduced) flag.

The TCP sender reacts to these congestion notifications in exactly the same way as it does to packet loss that is detected via duplicate acknowledgements. However, the situation is strictly better. Congestion has been detected and no packet was harmed in any way. ECN is described in RFC 3168. It requires both host and router support, and is not yet widely used on the Internet.

For more information on the complete set of congestion control behaviors that are implemented in TCP, see RFC 5681.

6.5.11 The Future of TCP

As the workhorse of the Internet, TCP has been used for many applications and extended over time to give good performance over a wide range of networks. Many versions are deployed with slightly different implementations than the classic algorithms we have described, especially for congestion control and robustness against attacks. It is likely that TCP will continue to evolve with the Internet. We will mention two particular issues.

The first one is that TCP does not provide the transport semantics that all applications want. For example, some applications want to send messages or records whose boundaries need to be preserved. Other applications work with a group of

related conversations, such as a Web browser that transfers several objects from the same server. Still other applications want better control over the network paths that they use. TCP with its standard sockets interface does not meet these needs well. Essentially, the application has the burden of dealing with any problem not solved by TCP. This has led to proposals for new protocols that would provide a slightly different interface. Two examples are SCTP (Stream Control Transmission Protocol), defined in RFC 4960, and SST (Structured Stream Transport) (Ford, 2007). However, whenever someone proposes changing something that has worked so well for so long, there is always a huge battle between the "Users are demanding more features" and "If it ain't broke, don't fix it" camps.

The second issue is congestion control. You may have expected that this is a solved problem after our deliberations and the mechanisms that have been developed over time. Not so. The form of TCP congestion control that we described, and which is widely used, is based on packet losses as a signal of congestion. When Padhye et al. (1998) modeled TCP throughput based on the sawtooth pattern, they found that the packet loss rate must drop off rapidly with increasing speed. To reach a throughput of 1 Gbps with a round-trip time of 100 ms and 1500 byte packets, one packet can be lost approximately every 10 minutes. That is a packet loss rate of 2×10^{-8}, which is incredibly small. It is too infrequent to serve as a good congestion signal, and any other source of loss (e.g., packet transmission error rates of 10^{-7}) can easily dominate it, limiting the throughput.

This relationship has not been a problem in the past, but networks are getting faster and faster, leading many people to revisit congestion control. One possibility is to use an alternate congestion control in which the signal is not packet loss at all. We gave several examples in Sec. 6.2. The signal might be round-trip time, which grows when the network becomes congested, as is used by FAST TCP (Wei et al., 2006). Other approaches are possible too, and time will tell which is the best.

6.6 PERFORMANCE ISSUES

Performance issues are very important in computer networks. When hundreds or thousands of computers are interconnected, complex interactions, with unforeseen consequences, are common. Frequently, this complexity leads to poor performance and no one knows why. In the following sections, we will examine many issues related to network performance to see what kinds of problems exist and what can be done about them.

Unfortunately, understanding network performance is more an art than a science. There is little underlying theory that is actually of any use in practice. The best we can do is give some rules of thumb gained from hard experience and present examples taken from the real world. We have delayed this discussion until we studied the transport layer because the performance that applications receive

depends on the combined performance of the transport, network and link layers, and to be able to use TCP as an example in various places.

In the next sections, we will look at six aspects of network performance:

1. Performance problems.
2. Measuring network performance.
3. Host design for fast networks.
4. Fast segment processing.
5. Header compression.
6. Protocols for "long fat" networks.

These aspects consider network performance both at the host and across the network, and as networks are increased in speed and size.

6.6.1 Performance Problems in Computer Networks

Some performance problems, such as congestion, are caused by temporary resource overloads. If more traffic suddenly arrives at a router than the router can handle, congestion will build up and performance will suffer. We studied congestion in detail in this and the previous chapter.

Performance also degrades when there is a structural resource imbalance. For example, if a gigabit communication line is attached to a low-end PC, the poor host will not be able to process the incoming packets fast enough and some will be lost. These packets will eventually be retransmitted, adding delay, wasting bandwidth, and generally reducing performance.

Overloads can also be synchronously triggered. As an example, if a segment contains a bad parameter (e.g., the port for which it is destined), in many cases the receiver will thoughtfully send back an error notification. Now consider what could happen if a bad segment is broadcast to 1000 machines: each one might send back an error message. The resulting **broadcast storm** could cripple the network. UDP suffered from this problem until the ICMP protocol was changed to cause hosts to refrain from responding to errors in UDP segments sent to broadcast addresses. Wireless networks must be particularly careful to avoid unchecked broadcast responses because broadcast occurs naturally and the wireless bandwidth is limited.

A second example of synchronous overload is what happens after an electrical power failure. When the power comes back on, all the machines simultaneously start rebooting. A typical reboot sequence might require first going to some (DHCP) server to learn one's true identity, and then to some file server to get a copy of the operating system. If hundreds of machines in a data center all do this at once, the server will probably collapse under the load.

Even in the absence of synchronous overloads and the presence of sufficient resources, poor performance can occur due to lack of system tuning. For example, if a machine has plenty of CPU power and memory but not enough of the memory has been allocated for buffer space, flow control will slow down segment reception and limit performance. This was a problem for many TCP connections as the Internet became faster but the default size of the flow control window stayed fixed at 64 KB.

Another tuning issue is setting timeouts. When a segment is sent, a timer is set to guard against loss of the segment. If the timeout is set too short, unnecessary retransmissions will occur, clogging the wires. If the timeout is set too long, unnecessary delays will occur after a segment is lost. Other tunable parameters include how long to wait for data on which to piggyback before sending a separate acknowledgement, and how many retransmissions to make before giving up.

Another performance problem that occurs with real-time applications like audio and video is jitter. Having enough bandwidth on average is not sufficient for good performance. Short transmission delays are also required. Consistently achieving short delays demands careful engineering of the load on the network, quality-of-service support at the link and network layers, or both.

6.6.2 Network Performance Measurement

When a network performs poorly, its users often complain to the folks running it, demanding improvements. To improve the performance, the operators must first determine exactly what is going on. To find out what is really happening, the operators must make measurements. In this section, we will look at network performance measurements. Much of the discussion below is based on the seminal work of Mogul (1993).

Measurements can be made in different ways and at many locations (both in the protocol stack and physically). The most basic kind of measurement is to start a timer when beginning some activity and see how long that activity takes. For example, knowing how long it takes for a segment to be acknowledged is a key measurement. Other measurements are made with counters that record how often some event has happened (e.g., number of lost segments). Finally, one is often interested in knowing the amount of something, such as the number of bytes processed in a certain time interval.

Measuring network performance and parameters has many potential pitfalls. We list a few of them here. Any systematic attempt to measure network performance should be careful to avoid these.

Make Sure That the Sample Size Is Large Enough

Do not measure the time to send one segment, but repeat the measurement, say, one million times and take the average. Startup effects, such as the 802.16 NIC or cable modem getting a bandwidth reservation after an idle period, can

slow the first segment, and queueing introduces variability. Having a large sample will reduce the uncertainty in the measured mean and standard deviation. This uncertainty can be computed using standard statistical formulas.

Make Sure That the Samples Are Representative

Ideally, the whole sequence of one million measurements should be repeated at different times of the day and the week to see the effect of different network conditions on the measured quantity. Measurements of congestion, for example, are of little use if they are made at a moment when there is no congestion. Sometimes the results may be counterintuitive at first, such as heavy congestion at 11 A.M., and 1 P.M., but no congestion at noon (when all the users are at lunch).

With wireless networks, location is an important variable because of signal propagation. Even a measurement node placed close to a wireless client may not observe the same packets as the client due to differences in the antennas. It is best to take measurements from the wireless client under study to see what it sees. Failing that, it is possible to use techniques to combine the wireless measurements taken at different vantage points to gain a more complete picture of what is going on (Mahajan et al., 2006).

Caching Can Wreak Havoc with Measurements

Repeating a measurement many times will return an unexpectedly fast answer if the protocols use caching mechanisms. For instance, fetching a Web page or looking up a DNS name (to find the IP address) may involve a network exchange the first time, and then return the answer from a local cache without sending any packets over the network. The results from such a measurement are essentially worthless (unless you want to measure cache performance).

Buffering can have a similar effect. TCP/IP performance tests have been known to report that UDP can achieve a performance substantially higher than the network allows. How does this occur? A call to UDP normally returns control as soon as the message has been accepted by the kernel and added to the transmission queue. If there is sufficient buffer space, timing 1000 UDP calls does not mean that all the data have been sent. Most of them may still be in the kernel, but the performance test program thinks they have all been transmitted.

Caution is advised to be absolutely sure that you understand how data can be cached and buffered as part of a network operation.

Be Sure That Nothing Unexpected Is Going On during Your Tests

Making measurements at the same time that some user has decided to run a video conference over your network will often give different results than if there is no video conference. It is best to run tests on an idle network and create the

entire workload yourself. Even this approach has pitfalls, though. While you might think nobody will be using the network at 3 A.M., that might be when the automatic backup program begins copying all the disks to tape. Or, there might be heavy traffic for your wonderful Web pages from distant time zones.

Wireless networks are challenging in this respect because it is often not possible to separate them from all sources of interference. Even if there are no other wireless networks sending traffic nearby, someone may microwave popcorn and inadvertently cause interference that degrades 802.11 performance. For these reasons, it is a good practice to monitor the overall network activity so that you can at least realize when something unexpected does happen.

Be Careful When Using a Coarse-Grained Clock

Computer clocks function by incrementing some counter at regular intervals. For example, a millisecond timer adds 1 to a counter every 1 msec. Using such a timer to measure an event that takes less than 1 msec is possible but requires some care. Some computers have more accurate clocks, of course, but there are always shorter events to measure too. Note that clocks are not always as accurate as the precision with which the time is returned when they are read.

To measure the time to make a TCP connection, for example, the clock (say, in milliseconds) should be read out when the transport layer code is entered and again when it is exited. If the true connection setup time is 300 μsec, the difference between the two readings will be either 0 or 1, both wrong. However, if the measurement is repeated one million times and the total of all measurements is added up and divided by one million, the mean time will be accurate to better than 1 μsec.

Be Careful about Extrapolating the Results

Suppose that you make measurements with simulated network loads running from 0 (idle) to 0.4 (40% of capacity). For example, the response time to send a voice-over-IP packet over an 802.11 network might be as shown by the data points and solid line through them in Fig. 6-49. It may be tempting to extrapolate linearly, as shown by the dotted line. However, many queueing results involve a factor of $1/(1-\rho)$, where ρ is the load, so the true values may look more like the dashed line, which rises much faster than linearly when the load gets high. That is, beware contention effects that become much more pronounced at high load.

6.6.3 Host Design for Fast Networks

Measuring and tinkering can improve performance considerably, but they cannot substitute for good design in the first place. A poorly designed network can be improved only so much. Beyond that, it has to be redesigned from scratch.

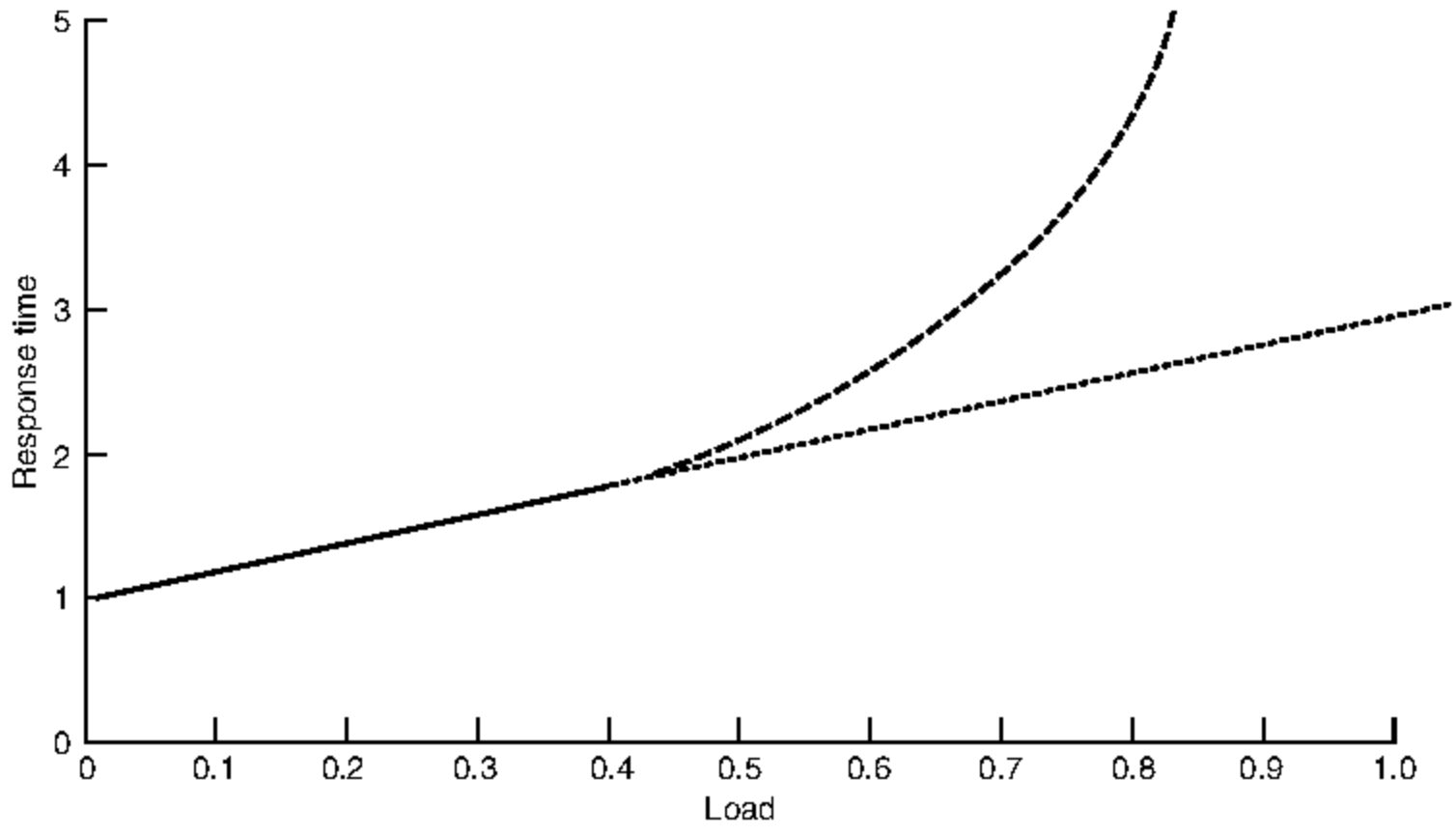

Figure 6-49. Response as a function of load.

In this section, we will present some rules of thumb for software implementation of network protocols on hosts. Surprisingly, experience shows that this is often a performance bottleneck on otherwise fast networks, for two reasons. First, NICs (Network Interface Cards) and routers have already been engineered (with hardware support) to run at "wire speed." This means that they can process packets as quickly as the packets can possibly arrive on the link. Second, the relevant performance is that which applications obtain. It is not the link capacity, but the throughput and delay after network and transport processing.

Reducing software overheads improves performance by increasing throughput and decreasing delay. It can also reduce the energy that is spent on networking, which is an important consideration for mobile computers. Most of these ideas have been common knowledge to network designers for years. They were first stated explicitly by Mogul (1993); our treatment largely follows his. Another relevant source is Metcalfe (1993).

Host Speed Is More Important Than Network Speed

Long experience has shown that in nearly all fast networks, operating system and protocol overhead dominate actual time on the wire. For example, in theory, the minimum RPC time on a 1-Gbps Ethernet is 1 μsec, corresponding to a minimum (512-byte) request followed by a minimum (512-byte) reply. In practice, overcoming the software overhead and getting the RPC time anywhere near there is a substantial achievement. It rarely happens in practice.

Similarly, the biggest problem in running at 1 Gbps is often getting the bits from the user's buffer out onto the network fast enough and having the receiving host process them as fast as they come in. If you double the host (CPU and memory) speed, you often can come close to doubling the throughput. Doubling the network capacity has no effect if the bottleneck is in the hosts.

Reduce Packet Count to Reduce Overhead

Each segment has a certain amount of overhead (e.g., the header) as well as data (e.g., the payload). Bandwidth is required for both components. Processing is also required for both components (e.g., header processing and doing the checksum). When 1 million bytes are being sent, the data cost is the same no matter what the segment size is. However, using 128-byte segments means 32 times as much per-segment overhead as using 4-KB segments. The bandwidth and processing overheads add up fast to reduce throughput.

Per-packet overhead in the lower layers amplifies this effect. Each arriving packet causes a fresh interrupt if the host is keeping up. On a modern pipelined processor, each interrupt breaks the CPU pipeline, interferes with the cache, requires a change to the memory management context, voids the branch prediction table, and forces a substantial number of CPU registers to be saved. An n-fold reduction in segments sent thus reduces the interrupt and packet overhead by a factor of n.

You might say that both people and computers are poor at multitasking. This observation underlies the desire to send MTU packets that are as large as will pass along the network path without fragmentation. Mechanisms such as Nagle's algorithm and Clark's solution are also attempts to avoid sending small packets.

Minimize Data Touching

The most straightforward way to implement a layered protocol stack is with one module for each layer. Unfortunately, this leads to copying (or at least accessing the data on multiple passes) as each layer does its own work. For example, after a packet is received by the NIC, it is typically copied to a kernel buffer. From there, it is copied to a network layer buffer for network layer processing, then to a transport layer buffer for transport layer processing, and finally to the receiving application process. It is not unusual for an incoming packet to be copied three or four times before the segment enclosed in it is delivered.

All this copying can greatly degrade performance because memory operations are an order of magnitude slower than register–register instructions. For example, if 20% of the instructions actually go to memory (i.e., are cache misses), which is likely when touching incoming packets, the average instruction execution time is slowed down by a factor of 2.8 ($0.8 \times 1 + 0.2 \times 10$). Hardware assistance will not help here. The problem is too much copying by the operating system.

A clever operating system will minimize copying by combining the processing of multiple layers. For example, TCP and IP are usually implemented together (as "TCP/IP") so that it is not necessary to copy the payload of the packet as processing switches from network to transport layer. Another common trick is to perform multiple operations within a layer in a single pass over the data. For example, checksums are often computed while copying the data (when it has to be copied) and the newly computed checksum is appended to the end.

Minimize Context Switches

A related rule is that context switches (e.g., from kernel mode to user mode) are deadly. They have the bad properties of interrupts and copying combined. This cost is why transport protocols are often implemented in the kernel. Like reducing packet count, context switches can be reduced by having the library procedure that sends data do internal buffering until it has a substantial amount of them. Similarly, on the receiving side, small incoming segments should be collected together and passed to the user in one fell swoop instead of individually, to minimize context switches.

In the best case, an incoming packet causes a context switch from the current user to the kernel, and then a switch to the receiving process to give it the newly arrived data. Unfortunately, with some operating systems, additional context switches happen. For example, if the network manager runs as a special process in user space, a packet arrival is likely to cause a context switch from the current user to the kernel, then another one from the kernel to the network manager, followed by another one back to the kernel, and finally one from the kernel to the receiving process. This sequence is shown in Fig. 6-50. All these context switches on each packet are wasteful of CPU time and can have a devastating effect on network performance.

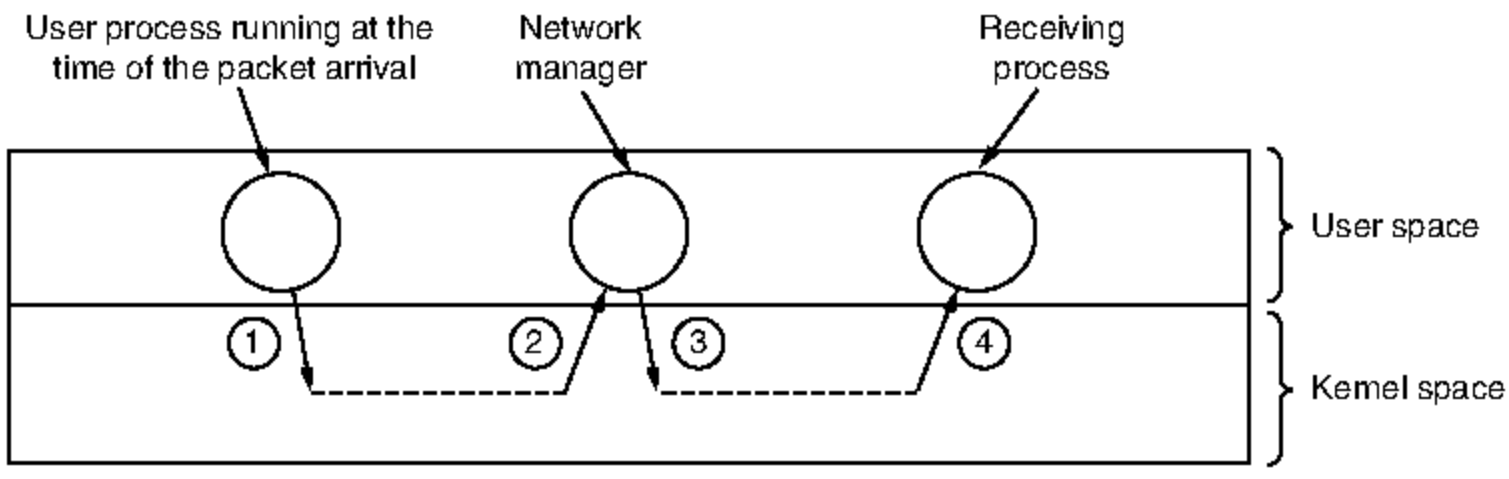

Figure 6-50. Four context switches to handle one packet with a user-space network manager.

Avoiding Congestion Is Better Than Recovering from It

The old maxim that an ounce of prevention is worth a pound of cure certainly holds for network congestion. When a network is congested, packets are lost, bandwidth is wasted, useless delays are introduced, and more. All of these costs are unnecessary, and recovering from congestion takes time and patience. Not having it occur in the first place is better. Congestion avoidance is like getting your DTP vaccination: it hurts a little at the time you get it, but it prevents something that would hurt a lot more in the future.

Avoid Timeouts

Timers are necessary in networks, but they should be used sparingly and timeouts should be minimized. When a timer goes off, some action is generally repeated. If it is truly necessary to repeat the action, so be it, but repeating it unnecessarily is wasteful.

The way to avoid extra work is to be careful that timers are set a little bit on the conservative side. A timer that takes too long to expire adds a small amount of extra delay to one connection in the (unlikely) event of a segment being lost. A timer that goes off when it should not have uses up host resources, wastes bandwidth, and puts extra load on perhaps dozens of routers for no good reason.

6.6.4 Fast Segment Processing

Now that we have covered general rules, we will look at some specific methods for speeding up segment processing. For more information, see Clark et al. (1989), and Chase et al. (2001).

Segment processing overhead has two components: overhead per segment and overhead per byte. Both must be attacked. The key to fast segment processing is to separate out the normal, successful case (one-way data transfer) and handle it specially. Many protocols tend to emphasize what to do when something goes wrong (e.g., a packet getting lost), but to make the protocols run fast, the designer should aim to minimize processing time when everything goes right. Minimizing processing time when an error occurs is secondary.

Although a sequence of special segments is needed to get into the *ESTABLISHED* state, once there, segment processing is straightforward until one side starts to close the connection. Let us begin by examining the sending side in the *ESTABLISHED* state when there are data to be transmitted. For the sake of clarity, we assume here that the transport entity is in the kernel, although the same ideas apply if it is a user-space process or a library inside the sending process. In Fig. 6-51, the sending process traps into the kernel to do the SEND. The first thing the transport entity does is test to see if this is the normal case: the state is *ESTABLISHED*, neither side is trying to close the connection, a regular (i.e., not an

out-of-band) full segment is being sent, and enough window space is available at the receiver. If all conditions are met, no further tests are needed and the fast path through the sending transport entity can be taken. Typically, this path is taken most of the time.

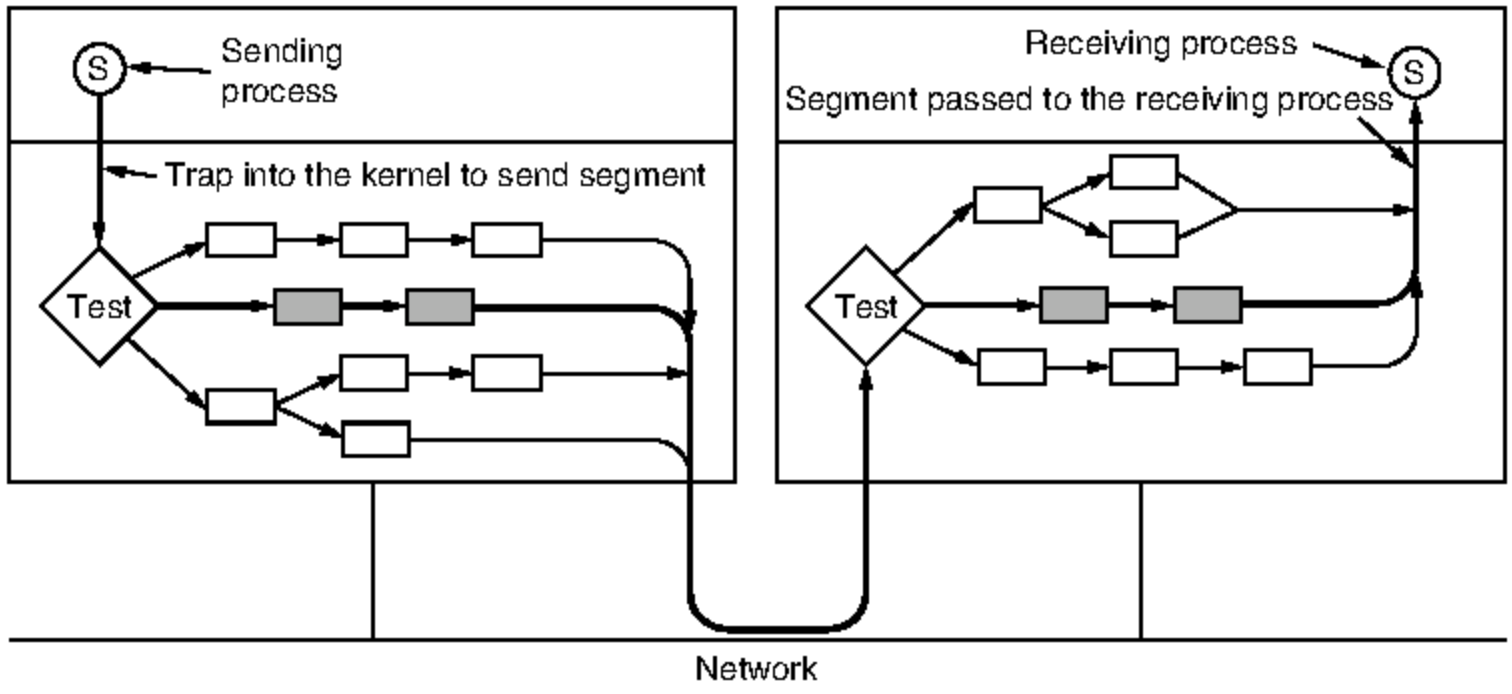

Figure 6-51. The fast path from sender to receiver is shown with a heavy line. The processing steps on this path are shaded.

In the usual case, the headers of consecutive data segments are almost the same. To take advantage of this fact, a prototype header is stored within the transport entity. At the start of the fast path, it is copied as fast as possible to a scratch buffer, word by word. Those fields that change from segment to segment are overwritten in the buffer. Frequently, these fields are easily derived from state variables, such as the next sequence number. A pointer to the full segment header plus a pointer to the user data are then passed to the network layer. Here, the same strategy can be followed (not shown in Fig. 6-51). Finally, the network layer gives the resulting packet to the data link layer for transmission.

As an example of how this principle works in practice, let us consider TCP/IP. Fig. 6-52(a) shows the TCP header. The fields that are the same between consecutive segments on a one-way flow are shaded. All the sending transport entity has to do is copy the five words from the prototype header into the output buffer, fill in the next sequence number (by copying it from a word in memory), compute the checksum, and increment the sequence number in memory. It can then hand the header and data to a special IP procedure for sending a regular, maximum segment. IP then copies its five-word prototype header [see Fig. 6-52(b)] into the buffer, fills in the *Identification* field, and computes its checksum. The packet is now ready for transmission.

Now let us look at fast path processing on the receiving side of Fig. 6-51. Step 1 is locating the connection record for the incoming segment. For TCP, the

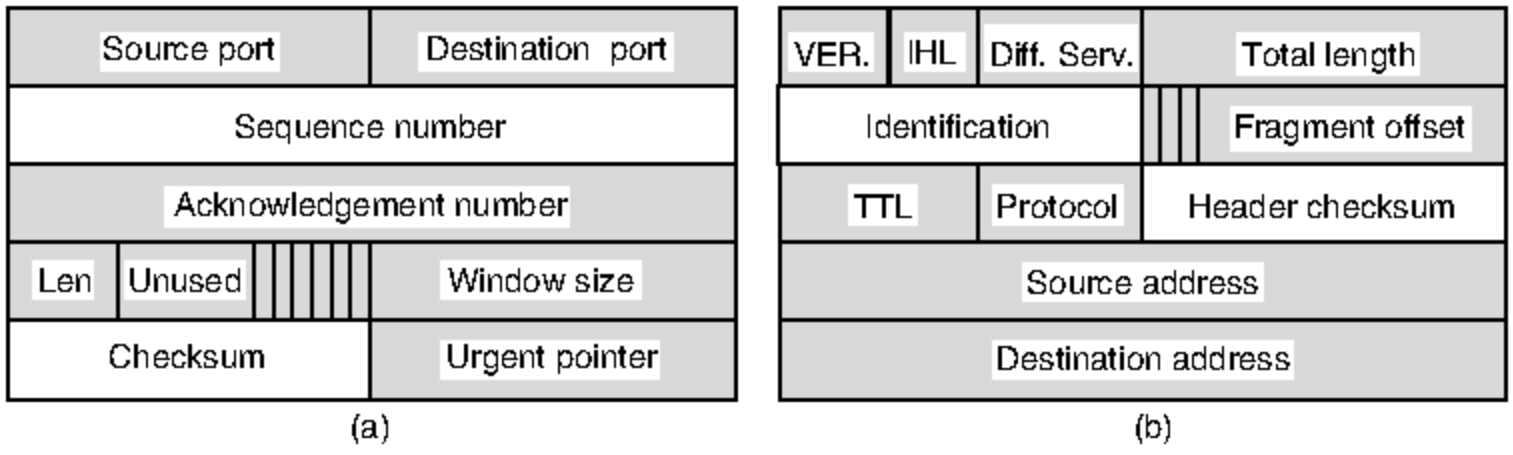

Figure 6-52. (a) TCP header. (b) IP header. In both cases, they are taken from the prototype without change.

connection record can be stored in a hash table for which some simple function of the two IP addresses and two ports is the key. Once the connection record has been located, both addresses and both ports must be compared to verify that the correct record has been found.

An optimization that often speeds up connection record lookup even more is to maintain a pointer to the last one used and try that one first. Clark et al. (1989) tried this and observed a hit rate exceeding 90%.

The segment is checked to see if it is a normal one: the state is *ESTABLISHED*, neither side is trying to close the connection, the segment is a full one, no special flags are set, and the sequence number is the one expected. These tests take just a handful of instructions. If all conditions are met, a special fast path TCP procedure is called.

The fast path updates the connection record and copies the data to the user. While it is copying, it also computes the checksum, eliminating an extra pass over the data. If the checksum is correct, the connection record is updated and an acknowledgement is sent back. The general scheme of first making a quick check to see if the header is what is expected and then having a special procedure handle that case is called **header prediction**. Many TCP implementations use it. When this optimization and all the other ones discussed in this chapter are used together, it is possible to get TCP to run at 90% of the speed of a local memory-to-memory copy, assuming the network itself is fast enough.

Two other areas where major performance gains are possible are buffer management and timer management. The issue in buffer management is avoiding unnecessary copying, as mentioned above. Timer management is important because nearly all timers set do not expire. They are set to guard against segment loss, but most segments and their acknowledgements arrive correctly. Hence, it is important to optimize timer management for the case of timers rarely expiring.

A common scheme is to use a linked list of timer events sorted by expiration time. The head entry contains a counter telling how many ticks away from expiry it is. Each successive entry contains a counter telling how many ticks after the

previous entry it is. Thus, if timers expire in 3, 10, and 12 ticks, respectively, the three counters are 3, 7, and 2, respectively.

At every clock tick, the counter in the head entry is decremented. When it hits zero, its event is processed and the next item on the list becomes the head. Its counter does not have to be changed. This way, inserting and deleting timers are expensive operations, with execution times proportional to the length of the list.

A much more efficient approach can be used if the maximum timer interval is bounded and known in advance. Here, an array called a **timing wheel** can be used, as shown in Fig. 6-53. Each slot corresponds to one clock tick. The current time shown is $T = 4$. Timers are scheduled to expire at 3, 10, and 12 ticks from now. If a new timer suddenly is set to expire in seven ticks, an entry is just made in slot 11. Similarly, if the timer set for $T + 10$ has to be canceled, the list starting in slot 14 has to be searched and the required entry removed. Note that the array of Fig. 6-53 cannot accommodate timers beyond $T + 15$.

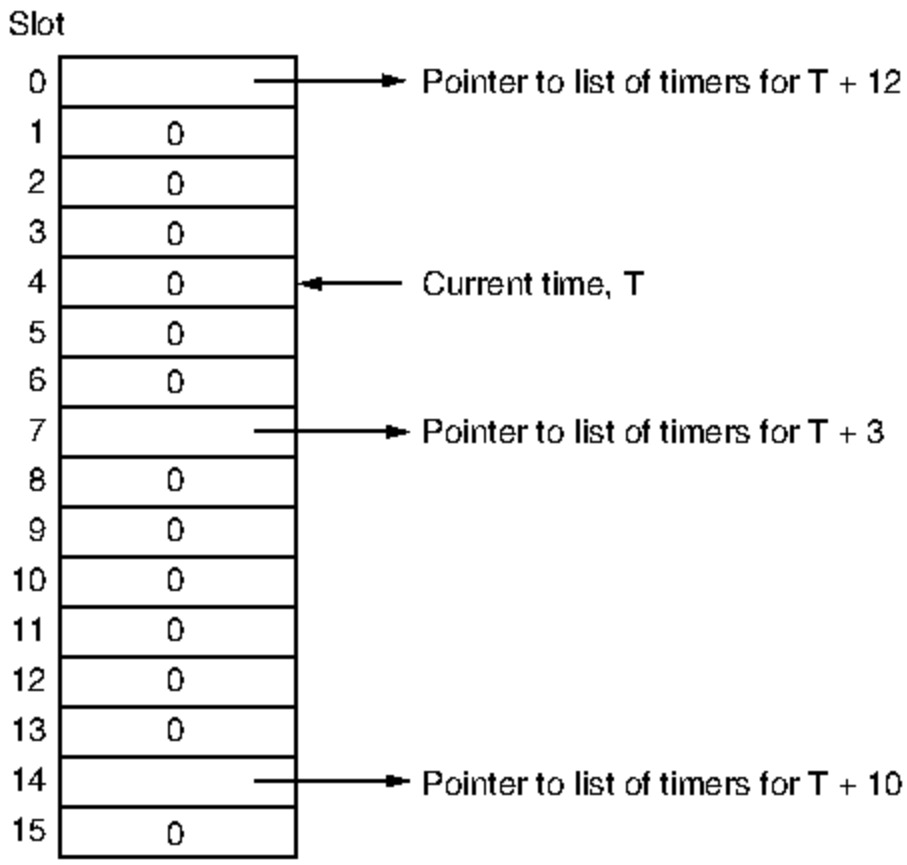

Figure 6-53. A timing wheel.

When the clock ticks, the current time pointer is advanced by one slot (circularly). If the entry now pointed to is nonzero, all of its timers are processed. Many variations on the basic idea are discussed by Varghese and Lauck (1987).

6.6.5 Header Compression

We have been looking at fast networks for too long. There is more out there. Let us now consider performance on wireless and other networks in which bandwidth is limited. Reducing software overhead can help mobile computers run

more efficiently, but it does nothing to improve performance when the network links are the bottleneck.

To use bandwidth well, protocol headers and payloads should be carried with the minimum of bits. For payloads, this means using compact encodings of information, such as images that are in JPEG format rather than a bitmap, or document formats such as PDF that include compression. It also means application-level caching mechanisms, such as Web caches that reduce transfers in the first place.

What about for protocol headers? At the link layer, headers for wireless networks are typically compact because they were designed with scarce bandwidth in mind. For example, 802.16 headers have short connection identifiers instead of longer addresses. However, higher layer protocols such as IP, TCP and UDP come in one version for all link layers, and they are not designed with compact headers. In fact, streamlined processing to reduce software overhead often leads to headers that are not as compact as they could otherwise be (e.g., IPv6 has a more loosely packed headers than IPv4).

The higher-layer headers can be a significant performance hit. Consider, for example, voice-over-IP data that is being carried with the combination of IP, UDP, and RTP. These protocols require 40 bytes of header (20 for IPv4, 8 for UDP, and 12 for RTP). With IPv6 the situation is even worse: 60 bytes, including the 40-byte IPv6 header. The headers can wind up as the majority of the transmitted data and consume more than half the bandwidth.

Header compression is used to reduce the bandwidth taken over links by higher-layer protocol headers. Specially designed schemes are used instead of general purpose methods. This is because headers are short, so they do not compress well individually, and decompression requires all prior data to be received. This will not be the case if a packet is lost.

Header compression obtains large gains by using knowledge of the protocol format. One of the first schemes was designed by Van Jacobson (1990) for compressing TCP/IP headers over slow serial links. It is able to compress a typical TCP/IP header of 40 bytes down to an average of 3 bytes. The trick to this method is hinted at in Fig. 6-52. Many of the header fields do not change from packet to packet. There is no need, for example, to send the same IP TTL or the same TCP port numbers in each and every packet. They can be omitted on the sending side of the link and filled in on the receiving side.

Similarly, other fields change in a predictable manner. For example, barring loss, the TCP sequence number advances with the data. In these cases, the receiver can predict the likely value. The actual number only needs to be carried when it differs from what is expected. Even then, it may be carried as a small change from the previous value, as when the acknowledgement number increases when new data is received in the reverse direction.

With header compression, it is possible to have simple headers in higher-layer protocols and compact encodings over low bandwidth links. **ROHC** (**RObust Header Compression**) is a modern version of header compression that is defined

as a framework in RFC 5795. It is designed to tolerate the loss that can occur on wireless links. There is a profile for each set of protocols to be compressed, such as IP/UDP/RTP. Compressed headers are carried by referring to a context, which is essentially a connection; header fields may easily be predicted for packets of the same connection, but not for packets of different connections. In typical operation, ROHC reduces IP/UDP/RTP headers from 40 bytes to 1 to 3 bytes.

While header compression is mainly targeted at reducing bandwidth needs, it can also be useful for reducing delay. Delay is comprised of propagation delay, which is fixed given a network path, and transmission delay, which depends on the bandwidth and amount of data to be sent. For example, a 1-Mbps link sends 1 bit in 1 μsec. In the case of media over wireless networks, the network is relatively slow so transmission delay may be an important factor in overall delay and consistently low delay is important for quality of service.

Header compression can help by reducing the amount of data that is sent, and hence reducing transmission delay. The same effect can be achieved by sending smaller packets. This will trade increased software overhead for decreased transmission delay. Note that another potential source of delay is queueing delay to access the wireless link. This can also be significant because wireless links are often heavily used as the limited resource in a network. In this case, the wireless link must have quality-of-service mechanisms that give low delay to real-time packets. Header compression alone is not sufficient.

6.6.6 Protocols for Long Fat Networks

Since the 1990s, there have been gigabit networks that transmit data over large distances. Because of the combination of a fast network, or "fat pipe," and long delay, these networks are called **long fat networks**. When these networks arose, people's first reaction was to use the existing protocols on them, but various problems quickly arose. In this section, we will discuss some of the problems with scaling up the speed and delay of network protocols.

The first problem is that many protocols use 32-bit sequence numbers. When the Internet began, the lines between routers were mostly 56-kbps leased lines, so a host blasting away at full speed took over 1 week to cycle through the sequence numbers. To the TCP designers, 2^{32} was a pretty decent approximation of infinity because there was little danger of old packets still being around a week after they were transmitted. With 10-Mbps Ethernet, the wrap time became 57 minutes, much shorter, but still manageable. With a 1-Gbps Ethernet pouring data out onto the Internet, the wrap time is about 34 seconds, well under the 120-sec maximum packet lifetime on the Internet. All of a sudden, 2^{32} is not nearly as good an approximation to infinity since a fast sender can cycle through the sequence space while old packets still exist.

The problem is that many protocol designers simply assumed, without stating it, that the time required to use up the entire sequence space would greatly exceed

the maximum packet lifetime. Consequently, there was no need to even worry about the problem of old duplicates still existing when the sequence numbers wrapped around. At gigabit speeds, that unstated assumption fails. Fortunately, it proved possible to extend the effective sequence number by treating the timestamp that can be carried as an option in the TCP header of each packet as the high-order bits. This mechanism is called PAWS (Protection Against Wrapped Sequence numbers) and is described in RFC 1323.

A second problem is that the size of the flow control window must be greatly increased. Consider, for example, sending a 64-KB burst of data from San Diego to Boston in order to fill the receiver's 64-KB buffer. Suppose that the link is 1 Gbps and the one-way speed-of-light-in-fiber delay is 20 msec. Initially, at $t = 0$, the pipe is empty, as illustrated in Fig. 6-54(a). Only 500 μsec later, in Fig. 6-54(b), all the segments are out on the fiber. The lead segment will now be somewhere in the vicinity of Brawley, still deep in Southern California. However, the transmitter must stop until it gets a window update.

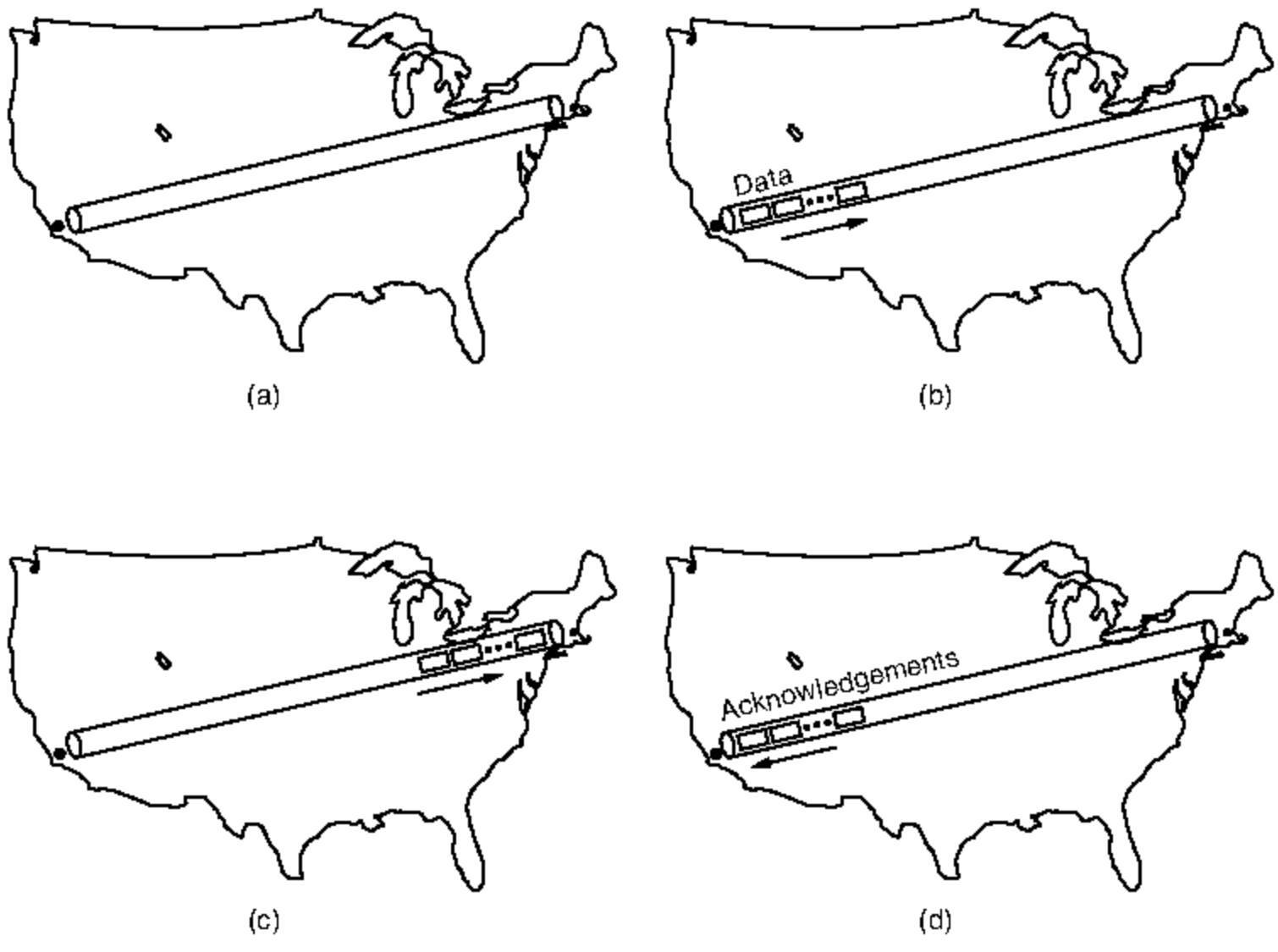

Figure 6-54. The state of transmitting 1 Mbit from San Diego to Boston. (a) At $t = 0$. (b) After 500 μsec. (c) After 20 msec. (d) After 40 msec.

After 20 msec, the lead segment hits Boston, as shown in Fig. 6-54(c), and is acknowledged. Finally, 40 msec after starting, the first acknowledgement gets

back to the sender and the second burst can be transmitted. Since the transmission line was used for 1.25 msec out of 100, the efficiency is about 1.25%. This situation is typical of an older protocols running over gigabit lines.

A useful quantity to keep in mind when analyzing network performance is the **bandwidth-delay product**. It is obtained by multiplying the bandwidth (in bits/sec) by the round-trip delay time (in sec). The product is the capacity of the pipe from the sender to the receiver and back (in bits).

For the example of Fig. 6-54, the bandwidth-delay product is 40 million bits. In other words, the sender would have to transmit a burst of 40 million bits to be able to keep going full speed until the first acknowledgement came back. It takes this many bits to fill the pipe (in both directions). This is why a burst of half a million bits only achieves a 1.25% efficiency: it is only 1.25% of the pipe's capacity.

The conclusion that can be drawn here is that for good performance, the receiver's window must be at least as large as the bandwidth-delay product, and preferably somewhat larger since the receiver may not respond instantly. For a transcontinental gigabit line, at least 5 MB are required.

A third and related problem is that simple retransmission schemes, such as the go-back-n protocol, perform poorly on lines with a large bandwidth-delay product. Consider, the 1-Gbps transcontinental link with a round-trip transmission time of 40 msec. A sender can transmit 5 MB in one round trip. If an error is detected, it will be 40 msec before the sender is told about it. If go-back-n is used, the sender will have to retransmit not just the bad packet, but also the 5 MB worth of packets that came afterward. Clearly, this is a massive waste of resources. More complex protocols such as selective-repeat are needed.

A fourth problem is that gigabit lines are fundamentally different from megabit lines in that long gigabit lines are delay limited rather than bandwidth limited. In Fig. 6-55 we show the time it takes to transfer a 1-Mbit file 4000 km at various transmission speeds. At speeds up to 1 Mbps, the transmission time is dominated by the rate at which the bits can be sent. By 1 Gbps, the 40-msec round-trip delay dominates the 1 msec it takes to put the bits on the fiber. Further increases in bandwidth have hardly any effect at all.

Figure 6-55 has unfortunate implications for network protocols. It says that stop-and-wait protocols, such as RPC, have an inherent upper bound on their performance. This limit is dictated by the speed of light. No amount of technological progress in optics will ever improve matters (new laws of physics would help, though). Unless some other use can be found for a gigabit line while a host is waiting for a reply, the gigabit line is no better than a megabit line, just more expensive.

A fifth problem is that communication speeds have improved faster than computing speeds. (Note to computer engineers: go out and beat those communication engineers! We are counting on you.) In the 1970s, the ARPANET ran at 56 kbps and had computers that ran at about 1 MIPS. Compare these numbers to

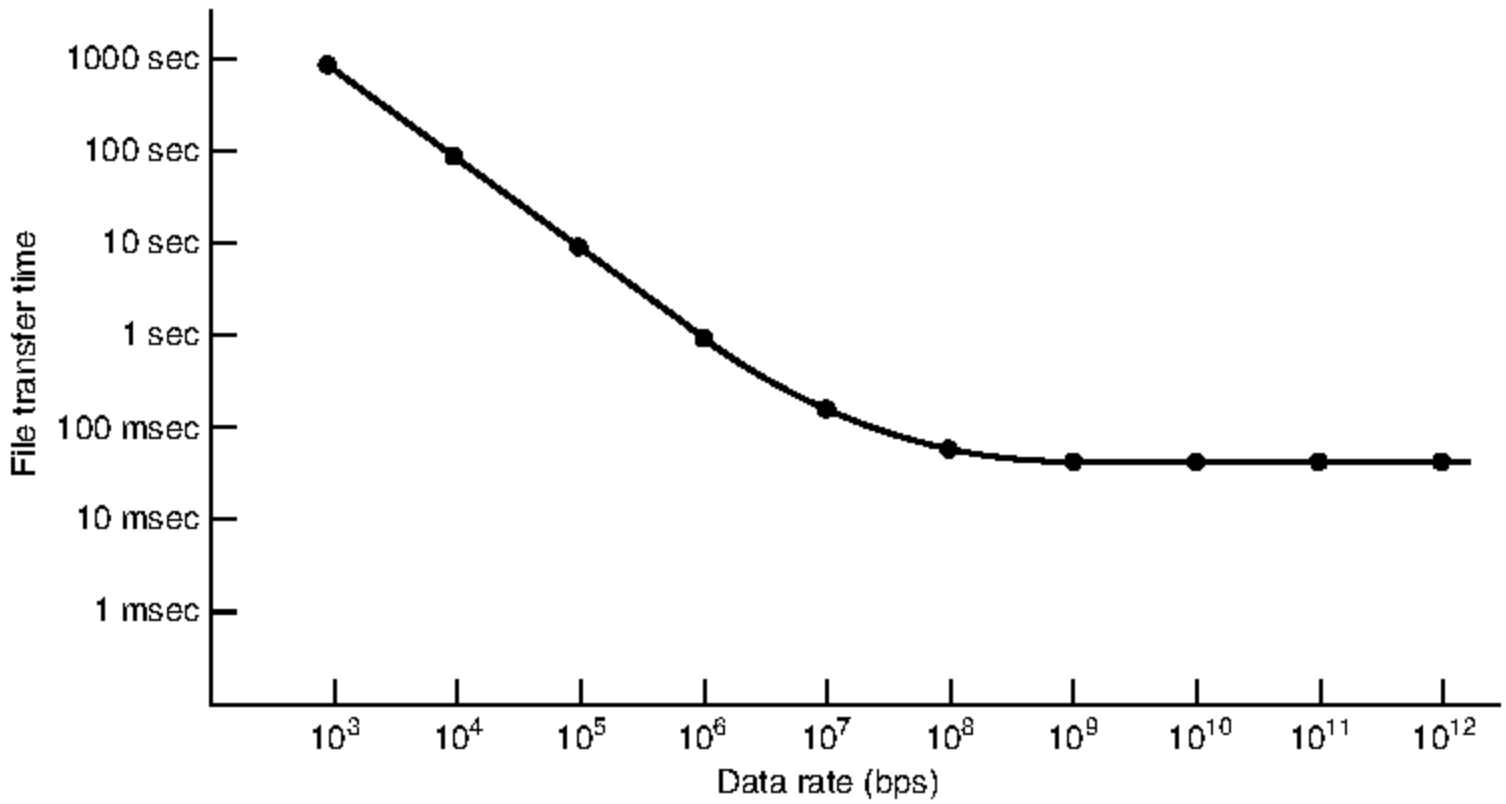

Figure 6-55. Time to transfer and acknowledge a 1-Mbit file over a 4000-km line.

1000-MIPS computers exchanging packets over a 1-Gbps line. The number of instructions per byte has decreased by more than a factor of 10. The exact numbers are debatable depending on dates and scenarios, but the conclusion is this: there is less time available for protocol processing than there used to be, so protocols must become simpler.

Let us now turn from the problems to ways of dealing with them. The basic principle that all high-speed network designers should learn by heart is:

Design for speed, not for bandwidth optimization.

Old protocols were often designed to minimize the number of bits on the wire, frequently by using small fields and packing them together into bytes and words. This concern is still valid for wireless networks, but not for gigabit networks. Protocol processing is the problem, so protocols should be designed to minimize it. The IPv6 designers clearly understood this principle.

A tempting way to go fast is to build fast network interfaces in hardware. The difficulty with this strategy is that unless the protocol is exceedingly simple, hardware just means a plug-in board with a second CPU and its own program. To make sure the network coprocessor is cheaper than the main CPU, it is often a slower chip. The consequence of this design is that much of the time the main (fast) CPU is idle waiting for the second (slow) CPU to do the critical work. It is a myth to think that the main CPU has other work to do while waiting. Furthermore, when two general-purpose CPUs communicate, race conditions can occur, so elaborate protocols are needed between the two processors to synchronize

them correctly and avoid races. Usually, the best approach is to make the protocols simple and have the main CPU do the work.

Packet layout is an important consideration in gigabit networks. The header should contain as few fields as possible, to reduce processing time, and these fields should be big enough to do the job and be word-aligned for fast processing. In this context, "big enough" means that problems such as sequence numbers wrapping around while old packets still exist, receivers being unable to advertise enough window space because the window field is too small, etc. do not occur.

The maximum data size should be large, to reduce software overhead and permit efficient operation. 1500 bytes is too small for high-speed networks, which is why gigabit Ethernet supports jumbo frames of up to 9 KB and IPv6 supports jumbogram packets in excess of 64 KB.

Let us now look at the issue of feedback in high-speed protocols. Due to the (relatively) long delay loop, feedback should be avoided: it takes too long for the receiver to signal the sender. One example of feedback is governing the transmission rate by using a sliding window protocol. Future protocols may switch to rate-based protocols to avoid the (long) delays inherent in the receiver sending window updates to the sender. In such a protocol, the sender can send all it wants to, provided it does not send faster than some rate the sender and receiver have agreed upon in advance.

A second example of feedback is Jacobson's slow start algorithm. This algorithm makes multiple probes to see how much the network can handle. With high-speed networks, making half a dozen or so small probes to see how the network responds wastes a huge amount of bandwidth. A more efficient scheme is to have the sender, receiver, and network all reserve the necessary resources at connection setup time. Reserving resources in advance also has the advantage of making it easier to reduce jitter. In short, going to high speeds inexorably pushes the design toward connection-oriented operation, or something fairly close to it.

Another valuable feature is the ability to send a normal amount of data along with the connection request. In this way, one round-trip time can be saved.

6.7 DELAY-TOLERANT NETWORKING

We will finish this chapter by describing a new kind of transport that may one day be an important component of the Internet. TCP and most other transport protocols are based on the assumption that the sender and the receiver are continuously connected by some working path, or else the protocol fails and data cannot be delivered. In some networks there is often no end-to-end path. An example is a space network as LEO (Low-Earth Orbit) satellites pass in and out of range of ground stations. A given satellite may be able to communicate to a ground station only at particular times, and two satellites may never be able to communicate with each other at any time, even via a ground station, because one of the satellites

may always be out of range. Other example networks involve submarines, buses, mobile phones, and other devices with computers for which there is intermittent connectivity due to mobility or extreme conditions.

In these occasionally connected networks, data can still be communicated by storing them at nodes and forwarding them later when there is a working link. This technique is called **message switching**. Eventually the data will be relayed to the destination. A network whose architecture is based on this approach is called a **DTN** (**Delay-Tolerant Network**, or a **Disruption-Tolerant Network**).

Work on DTNs started in 2002 when IETF set up a research group on the topic. The inspiration for DTNs came from an unlikely source: efforts to send packets in space. Space networks must deal with intermittent communication and very long delays. Kevin Fall observed that the ideas for these Interplanetary Internets could be applied to networks on Earth in which intermittent connectivity was the norm (Fall, 2003). This model gives a useful generalization of the Internet in which storage and delays can occur during communication. Data delivery is akin to delivery in the postal system, or electronic mail, rather than packet switching at routers.

Since 2002, the DTN architecture has been refined, and the applications of the DTN model have grown. As a mainstream application, consider large datasets of many terabytes that are produced by scientific experiments, media events, or Web-based services and need to be copied to datacenters at different locations around the world. Operators would like to send this bulk traffic at off-peak times to make use of bandwidth that has already been paid for but is not being used, and are willing to tolerate some delay. It is like doing the backups at night when other applications are not making heavy use of the network. The problem is that, for global services, the off-peak times are different at locations around the world. There may be little overlap in the times when datacenters in Boston and Perth have off-peak network bandwidth because night for one city is day for the other.

However, DTN models allow for storage and delays during transfer. With this model, it becomes possible to send the dataset from Boston to Amsterdam using off-peak bandwidth, as the cities have time zones that are only 6 hours apart. The dataset is then stored in Amsterdam until there is off-peak bandwidth between Amsterdam and Perth. It is then sent to Perth to complete the transfer. Laoutaris et al. (2009) have studied this model and find that it can provide substantial capacity at little cost, and that the use of a DTN model often doubles that capacity compared with a traditional end-to-end model.

In what follows, we will describe the IETF DTN architecture and protocols.

6.7.1 DTN Architecture

The main assumption in the Internet that DTNs seek to relax is that an end-to-end path between a source and a destination exists for the entire duration of a communication session. When this is not the case, the normal Internet protocols

fail. DTNs get around the lack of end-to-end connectivity with an architecture that is based on message switching, as shown in Fig. 6-56. It is also intended to tolerate links with low reliability and large delays. The architecture is specified in RFC 4838.

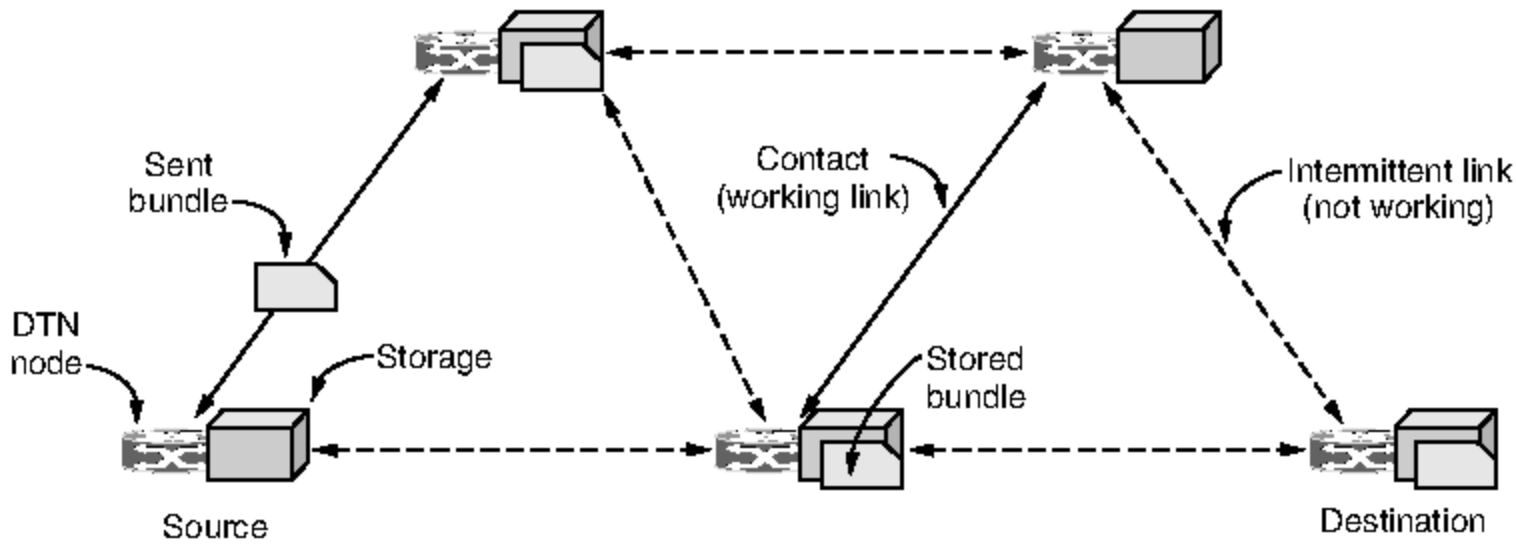

Figure 6-56. Delay-tolerant networking architecture.

In DTN terminology, a message is called a **bundle**. DTN nodes are equipped with storage, typically persistent storage such as a disk or flash memory. They store bundles until links become available and then forward the bundles. The links work intermittently. Fig. 6-56 shows five intermittent links that are not currently working, and two links that are working. A working link is called a **contact**. Fig. 6-56 also shows bundles stored at two DTN nodes awaiting contacts to send the bundles onward. In this way, the bundles are relayed via contacts from the source to their destination.

The storing and forwarding of bundles at DTN nodes sounds similar to the queueing and forwarding of packets at routers, but there are qualitative differences. In routers in the Internet, queueing occurs for milliseconds or at most seconds. At DTN nodes, bundles may be stored for hours, until a bus arrives in town, while an airplane completes a flight, until a sensor node harvests enough solar energy to run, until a sleeping computer wakes up, and so forth. These examples also point to a second difference, which is that nodes may move (with a bus or plane) while they hold stored data, and this movement may even be a key part of data delivery. Routers in the Internet are not allowed to move. The whole process of moving bundles might be better known as "store-carry-forward."

As an example, consider the scenario shown in Fig. 6-57 that was the first use of DTN protocols in space (Wood et al., 2008). The source of bundles is an LEO satellite that is recording Earth images as part of the Disaster Monitoring Constellation of satellites. The images must be returned to the collection point. However, the satellite has only intermittent contact with three ground stations as it orbits the Earth. It comes into contact with each ground station in turn. Each of the satellite, ground stations, and collection point act as a DTN node. At each contact, a

bundle (or a portion of a bundle) is sent to a ground station. The bundles are then sent over a backhaul terrestrial network to the collection point to complete the transfer.

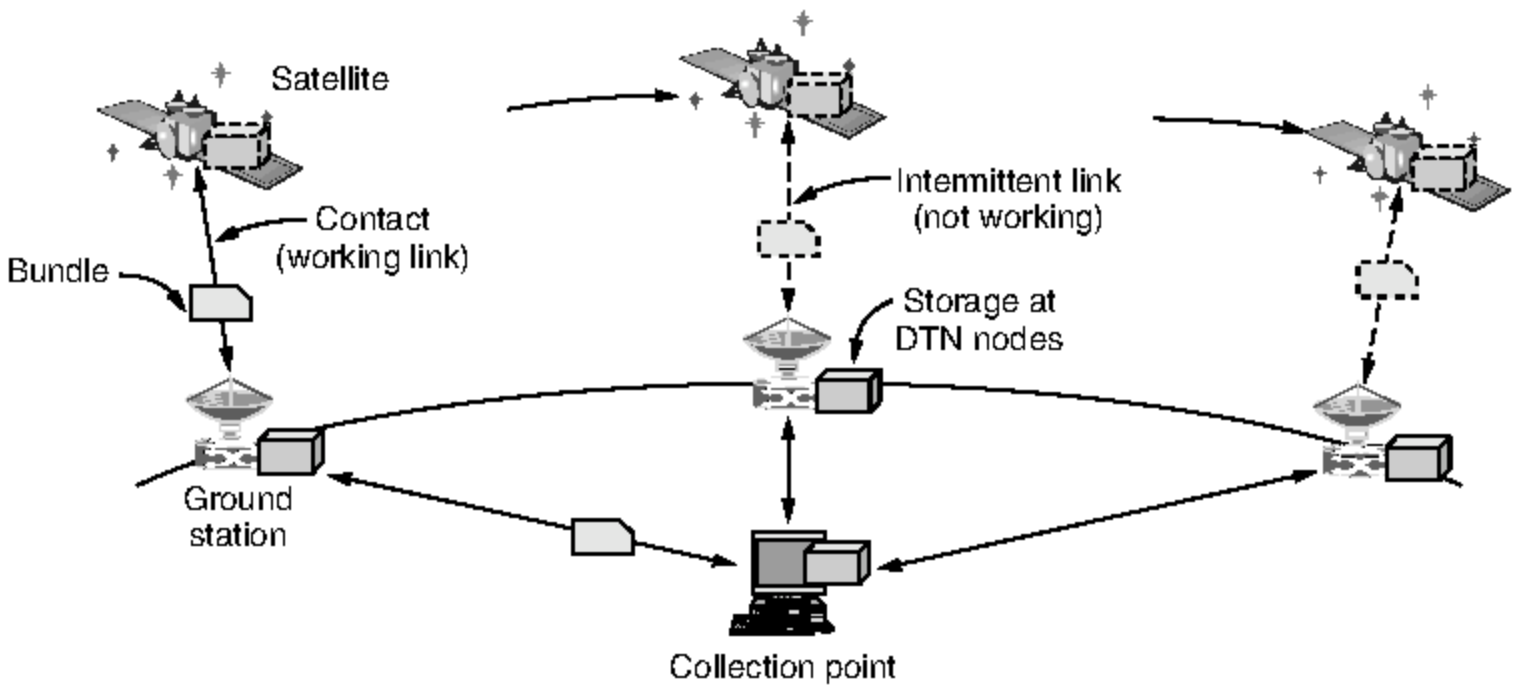

Figure 6-57. Use of a DTN in space.

The primary advantage of the DTN architecture in this example is that it naturally fits the situation of the satellite needing to store images because there is no connectivity at the time the image is taken. There are two further advantages. First, there may be no single contact long enough to send the images. However, they can be spread across the contacts with three ground stations. Second, the use of the link between the satellite and ground station is decoupled from the link over the backhaul network. This means that the satellite download is not limited by a slow terrestrial link. It can proceed at full speed, with the bundle stored at the ground station until it can be relayed to the collection point.

An important issue that is not specified by the architecture is how to find good routes via DTN nodes. A route in this path to use. Good routes depend on the nature of the architecture describes when to send data, and also which contacts. Some contacts are known ahead of time. A good example is the motion of heavenly bodies in the space example. For the space experiment, it was known ahead of time when contacts would occur, that the contact intervals ranged from 5 to 14 minutes per pass with each ground station, and that the downlink capacity was 8.134 Mbps. Given this knowledge, the transport of a bundle of images can be planned ahead of time.

In other cases, the contacts can be predicted, but with less certainty. Examples include buses that make contact with each other in mostly regular ways, due to a timetable, yet with some variation, and the times and amount of off-peak bandwidth in ISP networks, which are predicted from past data. At the other extreme, the contacts are occasional and random. One example is carrying data from user

to user on mobile phones depending on which users make contact with each other during the day. When there is unpredictability in contacts, one routing strategy is to send copies of the bundle along different paths in the hope that one of the copies is delivered to the destination before the lifetime is reached.

6.7.2 The Bundle Protocol

To take a closer look at the operation of DTNs, we will now look at the IETF protocols. DTNs are an emerging kind of network, and experimental DTNs have used different protocols, as there is no requirement that the IETF protocols be used. However, they are at least a good place to start and highlight many of the key issues.

The DTN protocol stack is shown in Fig. 6-58. The key protocol is the **Bundle protocol**, which is specified in RFC 5050. It is responsible for accepting messages from the application and sending them as one or more bundles via store-carry-forward operations to the destination DTN node. It is also apparent from Fig. 6-58 that the Bundle protocol runs above the level of TCP/IP. In other words, TCP/IP may be used over each contact to move bundles between DTN nodes. This positioning raises the issue of whether the Bundle protocol is a transport layer protocol or an application layer protocol. Just as with RTP, we take the position that, despite running over a transport protocol, the Bundle protocol is providing a transport service to many different applications, and so we cover DTNs in this chapter.

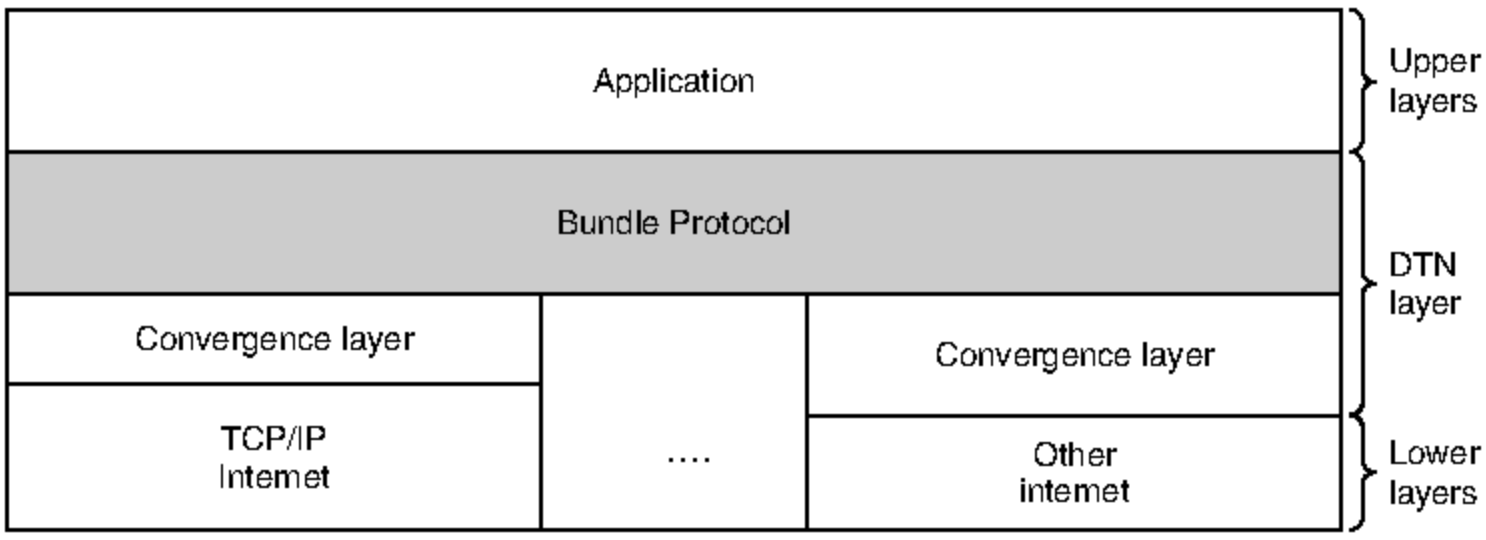

Figure 6-58. Delay-tolerant networking protocol stack.

In Fig. 6-58, we see that the Bundle protocol may be run over other kinds of protocols such as UDP, or even other kinds of internets. For example, in a space network the links may have very long delays. The round-trip time between Earth and Mars can easily be 20 minutes depending on the relative position of the planets. Imagine how well TCP acknowledgements and retransmissions will work over that link, especially for relatively short messages. Not well at all. Instead,

another protocol that uses error-correcting codes might be used. Or in sensor networks that are very resource constrained, a more lightweight protocol than TCP may be used.

Since the Bundle protocol is fixed, yet it is intended to run over a variety of transports, there is must be a gap in functionality between the protocols. That gap is the reason for the inclusion of a convergence layer in Fig. 6-58. The convergence layer is just a glue layer that matches the interfaces of the protocols that it joins. By definition there is a different convergence layer for each different lower layer transport. Convergence layers are commonly found in standards to join new and existing protocols.

The format of Bundle protocol messages is shown in Fig. 6-59. The different fields in these messages tell us some of the key issues that are handled by the Bundle protocol.

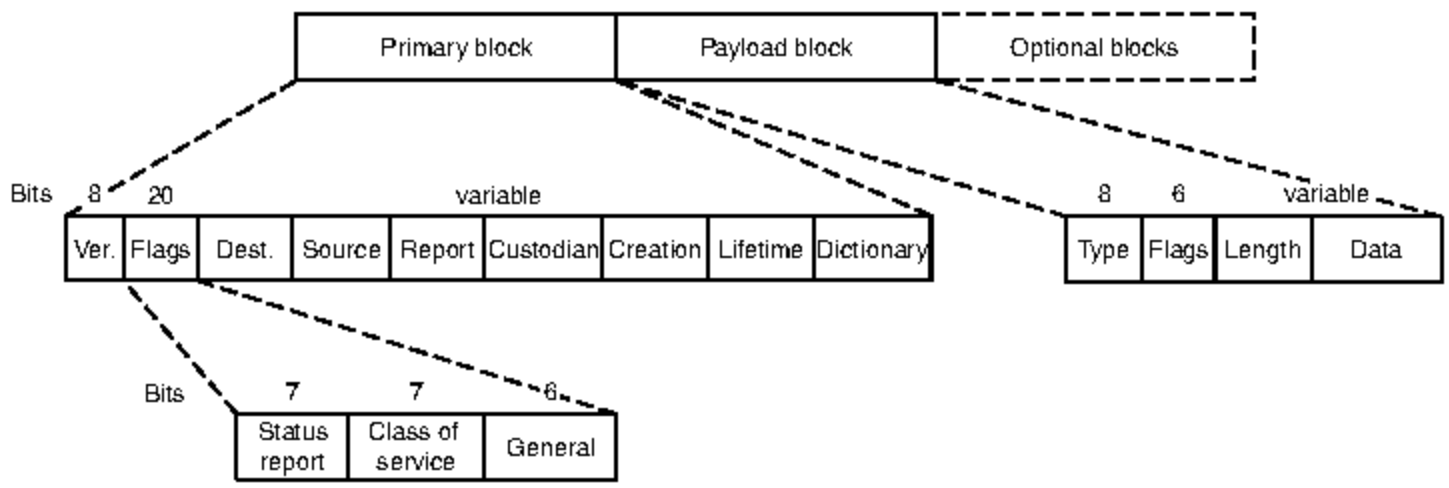

Figure 6-59. Bundle protocol message format.

Each message consists of a primary block, which can be thought of as a header, a payload block for the data, and optionally other blocks, for example to carry security parameters. The primary block begins with a *Version* field (currently 6) followed by a *Flags* field. Among other functions, the flags encode a class of service to let a source mark its bundles as higher or lower priority, and other handling requests such as whether the destination should acknowledge the bundle.

Then come addresses, which highlight three interesting parts of the design. As well as a *Destination* and *Source* identifier field, there is a *Custodian* identifier. The custodian is the party responsible for seeing that the bundle is delivered. In the Internet, the source node is usually the custodian, as it is the node that retransmits if the data is not ultimately delivered to the destination. However, in a DTN, the source node may not always be connected and may have no way of knowing whether the data has been delivered. DTNs deal with this problem using the notion of **custody transfer**, in which another node, closer to the destination, can assume responsibility for seeing the data safely delivered. For example, if a bundle is stored on an airplane for forwarding at a later time and location, the airplane may become the custodian of the bundle.

The second interesting aspect is that these identifiers are *not* IP addresses. Because the Bundle protocol is intended to work across a variety of transports and internets, it defines its own identifiers. These identifiers are really more like high-level names, such as Web page URLs, than low-level addresses, such as IP addresses. They give DTNs an aspect of application-level routing, such as email delivery or the distribution of software updates.

The third interesting aspect is the way the identifiers are encoded. There is also a *Report* identifier for diagnostic messages. All of the identifiers are encoded as references to a variable length *Dictionary* field. This provides compression when the custodian or report nodes are the same as the source or the destination. In fact, much of the message format has been designed with both extensibility and efficiency in mind by using a compact representation of variable length fields. The compact representation is important for wireless links and resource-constrained nodes such as in a sensor network.

Next comes a *Creation* field carrying the time at which the bundle was created, along with a sequence number from the source for ordering, plus a *Lifetime* field that tells the time at which the bundle data is no longer useful. These fields exist because data may be stored for a long period at DTN nodes and there must be some way to remove stale data from the network. Unlike the Internet, they require that DTN nodes have loosely synchronized clocks.

The primary block is completed with the *Dictionary* field. Then comes the payload block. This block starts with a short *Type* field that identifies it as a payload, followed by a small set of *Flags* that describe processing options. Then comes the *Data* field, preceded by a *Length* field. Finally, there may be other, optional blocks, such as a block that carries security parameters.

Many aspects of DTNs are being explored in the research community. Good strategies for routing depend on the nature of the contacts, as was mentioned above. Storing data inside the network raises other issues. Now congestion control must consider storage at nodes as another kind of resource that can be depleted. The lack of end-to-end communication also exacerbates security problems. Before a DTN node takes custody of a bundle, it may want to know that the sender is authorized to use the network and that the bundle is probably wanted by the destination. Solutions to these problems will depend on the kind of DTN, as space networks are different from sensor networks.

6.8 SUMMARY

The transport layer is the key to understanding layered protocols. It provides various services, the most important of which is an end-to-end, reliable, connection-oriented byte stream from sender to receiver. It is accessed through service primitives that permit the establishment, use, and release of connections. A common transport layer interface is the one provided by Berkeley sockets.

Transport protocols must be able to do connection management over unreliable networks. Connection establishment is complicated by the existence of delayed duplicate packets that can reappear at inopportune moments. To deal with them, three-way handshakes are needed to establish connections. Releasing a connection is easier than establishing one but is still far from trivial due to the two-army problem.

Even when the network layer is completely reliable, the transport layer has plenty of work to do. It must handle all the service primitives, manage connections and timers, allocate bandwidth with congestion control, and run a variable-sized sliding window for flow control.

Congestion control should allocate all of the available bandwidth between competing flows fairly, and it should track changes in the usage of the network. The AIMD control law converges to a fair and efficient allocation.

The Internet has two main transport protocols: UDP and TCP. UDP is a connectionless protocol that is mainly a wrapper for IP packets with the additional feature of multiplexing and demultiplexing multiple processes using a single IP address. UDP can be used for client-server interactions, for example, using RPC. It can also be used for building real-time protocols such as RTP.

The main Internet transport protocol is TCP. It provides a reliable, bidirectional, congestion-controlled byte stream with a 20-byte header on all segments. A great deal of work has gone into optimizing TCP performance, using algorithms from Nagle, Clark, Jacobson, Karn, and others.

Network performance is typically dominated by protocol and segment processing overhead, and this situation gets worse at higher speeds. Protocols should be designed to minimize the number of segments and work for large bandwidth-delay paths. For gigabit networks, simple protocols and streamlined processing are called for.

Delay-tolerant networking provides a delivery service across networks that have occasional connectivity or long delays across links. Intermediate nodes store, carry, and forward bundles of information so that it is eventually delivered, even if there is no working path from sender to receiver at any time.

PROBLEMS

1. In our example transport primitives of Fig. 6-2, LISTEN is a blocking call. Is this strictly necessary? If not, explain how a nonblocking primitive could be used. What advantage would this have over the scheme described in the text?

2. Primitives of transport service assume asymmetry between the two end points during connection establishment, one end (server) executes LISTEN while the other end (client) executes CONNECT. However, in peer to peer applications such file sharing

systems, e.g. BitTorrent, all end points are peers. There is no server or client functionality. How can transport service primitives may be used to build such peer to peer applications?

3. In the underlying model of Fig. 6-4, it is assumed that packets may be lost by the network layer and thus must be individually acknowledged. Suppose that the network layer is 100 percent reliable and never loses packets. What changes, if any, are needed to Fig. 6-4?

4. In both parts of Fig. 6-6, there is a comment that the value of *SERVER_PORT* must be the same in both client and server. Why is this so important?

5. In the Internet File Server example (Figure 6-6), can the connect( ) system call on the client fail for any reason other than listen queue being full on the server? Assume that the network is perfect.

6. One criteria for deciding whether to have a server active all the time or have it start on demand using a process server is how frequently the service provided is used. Can you think of any other criteria for making this decision?

7. Suppose that the clock-driven scheme for generating initial sequence numbers is used with a 15-bit wide clock counter. The clock ticks once every 100 msec, and the maximum packet lifetime is 60 sec. How often need resynchronization take place

(a) in the worst case?
(b) when the data consumes 240 sequence numbers/min?

8. Why does the maximum packet lifetime, T, have to be large enough to ensure that not only the packet but also its acknowledgements have vanished?

9. Imagine that a two-way handshake rather than a three-way handshake were used to set up connections. In other words, the third message was not required. Are deadlocks now possible? Give an example or show that none exist.

10. Imagine a generalized n-army problem, in which the agreement of any two of the blue armies is sufficient for victory. Does a protocol exist that allows blue to win?

11. Consider the problem of recovering from host crashes (i.e., Fig. 6-18). If the interval between writing and sending an acknowledgement, or vice versa, can be made relatively small, what are the two best sender-receiver strategies for minimizing the chance of a protocol failure?

12. In Figure 6-20, suppose a new flow E is added that takes a path from $R1$ to $R2$ to $R6$. How does the max-min bandwidth allocation change for the five flows?

13. Discuss the advantages and disadvantages of credits versus sliding window protocols.

14. Some other policies for fairness in congestion control are Additive Increase Additive Decrease (AIAD), Multiplicative Increase Additive Decrease (MIAD), and Multiplicative Increase Multiplicative Decrease (MIMD). Discuss these three policies in terms of convergence and stability.

15. Why does UDP exist? Would it not have been enough to just let user processes send raw IP packets?

16. Consider a simple application-level protocol built on top of UDP that allows a client to retrieve a file from a remote server residing at a well-known address. The client first sends a request with a file name, and the server responds with a sequence of data packets containing different parts of the requested file. To ensure reliability and sequenced delivery, client and server use a stop-and-wait protocol. Ignoring the obvious performance issue, do you see a problem with this protocol? Think carefully about the possibility of processes crashing.

17. A client sends a 128-byte request to a server located 100 km away over a 1-gigabit optical fiber. What is the efficiency of the line during the remote procedure call?

18. Consider the situation of the previous problem again. Compute the minimum possible response time both for the given 1-Gbps line and for a 1-Mbps line. What conclusion can you draw?

19. Both UDP and TCP use port numbers to identify the destination entity when delivering a message. Give two reasons why these protocols invented a new abstract ID (port numbers), instead of using process IDs, which already existed when these protocols were designed.

20. Several RPC implementations provide an option to the client to use RPC implemented over UDP or RPC implemented over TCP. Under what conditions will a client prefer to use RPC over UDP and under what conditions will he prefer to use RPC over TCP?

21. Consider two networks, $N1$ and $N2$, that have the same average delay between a source A and a destination D. In $N1$, the delay experienced by different packets is unformly distributed with maximum delay being 10 seconds, while in $N2$, 99% of the packets experience less than one second delay with no limit on maximum delay. Discuss how RTP may be used in these two cases to transmit live audio/video stream.

22. What is the total size of the minimum TCP MTU, including TCP and IP overhead but not including data link layer overhead?

23. Datagram fragmentation and reassembly are handled by IP and are invisible to TCP. Does this mean that TCP does not have to worry about data arriving in the wrong order?

24. RTP is used to transmit CD-quality audio, which makes a pair of 16-bit samples 44,100 times/sec, one sample for each of the stereo channels. How many packets per second must RTP transmit?

25. Would it be possible to place the RTP code in the operating system kernel, along with the UDP code? Explain your answer.

26. A process on host 1 has been assigned port p, and a process on host 2 has been assigned port q. Is it possible for there to be two or more TCP connections between these two ports at the same time?

27. In Fig. 6-36 we saw that in addition to the 32-bit *acknowledgement* field, there is an *ACK* bit in the fourth word. Does this really add anything? Why or why not?

28. The maximum payload of a TCP segment is 65,495 bytes. Why was such a strange number chosen?

29. Describe two ways to get into the *SYN RCVD* state of Fig. 6-39.

30. Consider the effect of using slow start on a line with a 10-msec round-trip time and no congestion. The receive window is 24 KB and the maximum segment size is 2 KB. How long does it take before the first full window can be sent?

31. Suppose that the TCP congestion window is set to 18 KB and a timeout occurs. How big will the window be if the next four transmission bursts are all successful? Assume that the maximum segment size is 1 KB.

32. If the TCP round-trip time, *RTT*, is currently 30 msec and the following acknowledgements come in after 26, 32, and 24 msec, respectively, what is the new *RTT* estimate using the Jacobson algorithm? Use $\alpha = 0.9$.

33. A TCP machine is sending full windows of 65,535 bytes over a 1-Gbps channel that has a 10-msec one-way delay. What is the maximum throughput achievable? What is the line efficiency?

34. What is the fastest line speed at which a host can blast out 1500-byte TCP payloads with a 120-sec maximum packet lifetime without having the sequence numbers wrap around? Take TCP, IP, and Ethernet overhead into consideration. Assume that Ethernet frames may be sent continuously.

35. To address the limitations of IP version 4, a major effort had to be undertaken via IETF that resulted in the design of IP version 6 and there are still is significant reluctance in the adoption of this new version. However, no such major effort is needed to address the limitations of TCP. Explain why this is the case.

36. In a network whose max segment is 128 bytes, max segment lifetime is 30 sec, and has 8-bit sequence numbers, what is the maximum data rate per connection?

37. Suppose that you are measuring the time to receive a segment. When an interrupt occurs, you read out the system clock in milliseconds. When the segment is fully processed, you read out the clock again. You measure 0 msec 270,000 times and 1 msec 730,000 times. How long does it take to receive a segment?

38. A CPU executes instructions at the rate of 1000 MIPS. Data can be copied 64 bits at a time, with each word copied costing 10 instructions. If an coming packet has to be copied four times, can this system handle a 1-Gbps line? For simplicity, assume that all instructions, even those instructions that read or write memory, run at the full 1000-MIPS rate.

39. To get around the problem of sequence numbers wrapping around while old packets still exist, one could use 64-bit sequence numbers. However, theoretically, an optical fiber can run at 75 Tbps. What maximum packet lifetime is required to make sure that future 75-Tbps networks do not have wraparound problems even with 64-bit sequence numbers? Assume that each byte has its own sequence number, as TCP does.

40. In Sec. 6.6.5, we calculated that a gigabit line dumps 80,000 packets/sec on the host, giving it only 6250 instructions to process it and leaving half the CPU time for applications. This calculation assumed a 1500-byte packet. Redo the calculation for an ARPANET-sized packet (128 bytes). In both cases, assume that the packet sizes given include all overhead.

41. For a 1-Gbps network operating over 4000 km, the delay is the limiting factor, not the bandwidth. Consider a MAN with the average source and destination 20 km apart. At what data rate does the round-trip delay due to the speed of light equal the transmission delay for a 1-KB packet?

42. Calculate the bandwidth-delay product for the following networks: (1) T1 (1.5 Mbps), (2) Ethernet (10 Mbps), (3) T3 (45 Mbps), and (4) STS-3 (155 Mbps). Assume an RTT of 100 msec. Recall that a TCP header has 16 bits reserved for Window Size. What are its implications in light of your calculations?

43. What is the bandwidth-delay product for a 50-Mbps channel on a geostationary satellite? If the packets are all 1500 bytes (including overhead), how big should the window be in packets?

44. The file server of Fig. 6-6 is far from perfect and could use a few improvements. Make the following modifications.

(a) Give the client a third argument that specifies a byte range.
(b) Add a client flag –w that allows the file to be written to the server.

45. One common function that all network protocols need is to manipulate messages. Recall that protocols manipulate messages by adding/striping headers. Some protocols may break a single message into multiple fragments, and later join these multiple fragments back into a single message. To this end, design and implement a message management library that provides support for creating a new message, attaching a header to a message, stripping a header from a message, breaking a message into two messages, combining two messages into a single message, and saving a copy of a message. Your implementation must minimize data copying from one buffer to another as much as possible. It is critical that the operations that manipulate messages do not touch the data in a message, but rather, only manipulate pointers.

46. Design and implement a chat system that allows multiple groups of users to chat. A chat coordinator resides at a well-known network address, uses UDP for communication with chat clients, sets up chat servers for each chat session, and maintains a chat session directory. There is one chat server per chat session. A chat server uses TCP for communication with clients. A chat client allows users to start, join, and leave a chat session. Design and implement the coordinator, server, and client code.

7

THE APPLICATION LAYER

Having finished all the preliminaries, we now come to the layer where all the applications are found. The layers below the application layer are there to provide transport services, but they do not do real work for users. In this chapter, we will study some real network applications.

However, even in the application layer there is a need for support protocols, to allow the applications to function. Accordingly, we will look at an important one of these before starting with the applications themselves. The item in question is DNS, which handles naming within the Internet. After that, we will examine three real applications: electronic mail, the World Wide Web, and multimedia. We will finish the chapter by saying more about content distribution, including by peer-to-peer networks.

7.1 DNS—THE DOMAIN NAME SYSTEM

Although programs theoretically could refer to Web pages, mailboxes, and other resources by using the network (e.g., IP) addresses of the computers on which they are stored, these addresses are hard for people to remember. Also, browsing a company's Web pages from *128.111.24.41* means that if the company moves the Web server to a different machine with a different IP address, everyone needs to be told the new IP address. Consequently, high-level, readable names were introduced in order to decouple machine names from machine addresses. In

this way, the company's Web server might be known as *www.cs.washington.edu* regardless of its IP address. Nevertheless, since the network itself understands only numerical addresses, some mechanism is required to convert the names to network addresses. In the following sections, we will study how this mapping is accomplished in the Internet.

Way back in the ARPANET days, there was simply a file, *hosts.txt*, that listed all the computer names and their IP addresses. Every night, all the hosts would fetch it from the site at which it was maintained. For a network of a few hundred large timesharing machines, this approach worked reasonably well.

However, well before many millions of PCs were connected to the Internet, everyone involved with it realized that this approach could not continue to work forever. For one thing, the size of the file would become too large. However, even more importantly, host name conflicts would occur constantly unless names were centrally managed, something unthinkable in a huge international network due to the load and latency. To solve these problems, **DNS** (**Domain Name System**) was invented in 1983. It has been a key part of the Internet ever since.

The essence of DNS is the invention of a hierarchical, domain-based naming scheme and a distributed database system for implementing this naming scheme. It is primarily used for mapping host names to IP addresses but can also be used for other purposes. DNS is defined in RFCs 1034, 1035, 2181, and further elaborated in many others.

Very briefly, the way DNS is used is as follows. To map a name onto an IP address, an application program calls a library procedure called the **resolver**, passing it the name as a parameter. We saw an example of a resolver, *gethostbyname*, in Fig. 6-6. The resolver sends a query containing the name to a local DNS server, which looks up the name and returns a response containing the IP address to the resolver, which then returns it to the caller. The query and response messages are sent as UDP packets. Armed with the IP address, the program can then establish a TCP connection with the host or send it UDP packets.

7.1.1 The DNS Name Space

Managing a large and constantly changing set of names is a nontrivial problem. In the postal system, name management is done by requiring letters to specify (implicitly or explicitly) the country, state or province, city, street address, and name of the addressee. Using this kind of hierarchical addressing ensures that there is no confusion between the Marvin Anderson on Main St. in White Plains, N.Y. and the Marvin Anderson on Main St. in Austin, Texas. DNS works the same way.

For the Internet, the top of the naming hierarchy is managed by an organization called **ICANN** (**Internet Corporation for Assigned Names and Numbers**). ICANN was created for this purpose in 1998, as part of the maturing of the Internet to a worldwide, economic concern. Conceptually, the Internet is divided into

over 250 **top-level domains**, where each domain covers many hosts. Each domain is partitioned into subdomains, and these are further partitioned, and so on. All these domains can be represented by a tree, as shown in Fig. 7-1. The leaves of the tree represent domains that have no subdomains (but do contain machines, of course). A leaf domain may contain a single host, or it may represent a company and contain thousands of hosts.

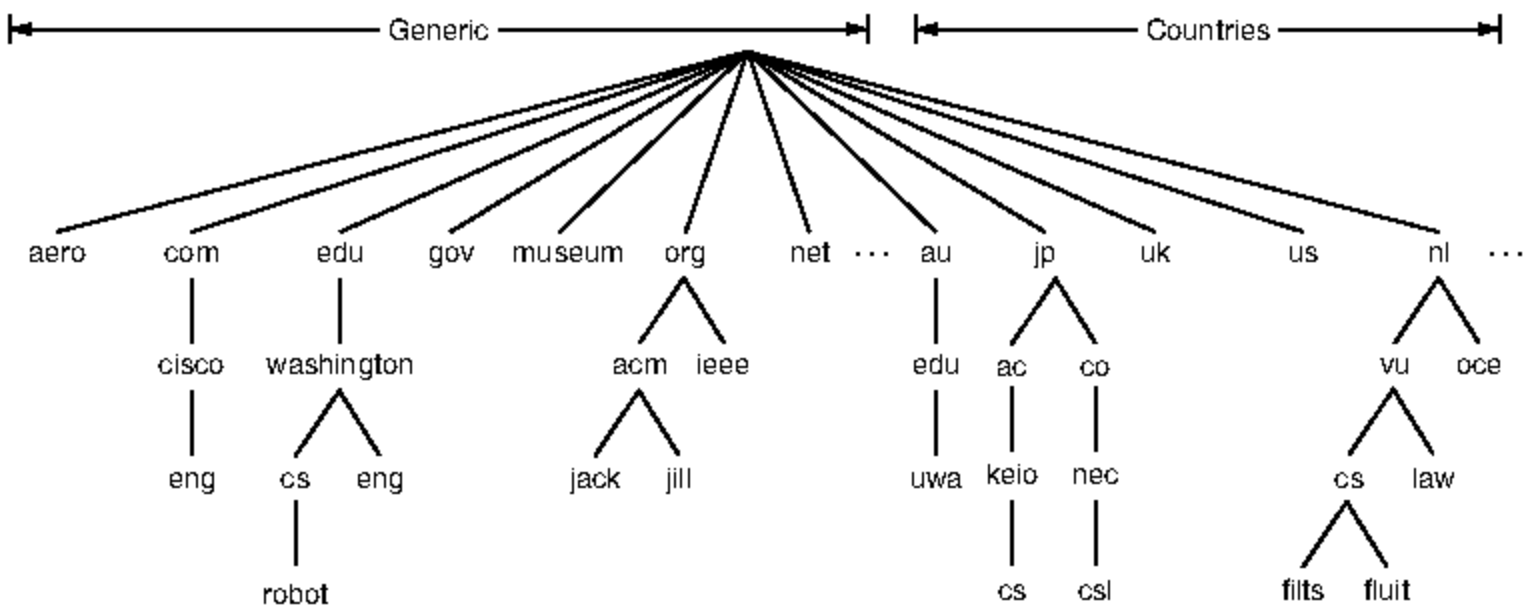

Figure 7-1. A portion of the Internet domain name space.

The top-level domains come in two flavors: generic and countries. The generic domains, listed in Fig. 7-2, include original domains from the 1980s and domains introduced via applications to ICANN. Other generic top-level domains will be added in the future.

The country domains include one entry for every country, as defined in ISO 3166. Internationalized country domain names that use non-Latin alphabets were introduced in 2010. These domains let people name hosts in Arabic, Cyrillic, Chinese, or other languages.

Getting a second-level domain, such as *name-of-company.com*, is easy. The top-level domains are run by **registrars** appointed by ICANN. Getting a name merely requires going to a corresponding registrar (for *com* in this case) to check if the desired name is available and not somebody else's trademark. If there are no problems, the requester pays the registrar a small annual fee and gets the name.

However, as the Internet has become more commercial and more international, it has also become more contentious, especially in matters related to naming. This controversy includes ICANN itself. For example, the creation of the *xxx* domain took several years and court cases to resolve. Is voluntarily placing adult content in its own domain a good or a bad thing? (Some people did not want adult content available at all on the Internet while others wanted to put it all in one domain so nanny filters could easily find and block it from children). Some of the domains self-organize, while others have restrictions on who can obtain a name, as noted in Fig. 7-2. But what restrictions are appropriate? Take the *pro* domain,

Domain	Intended use	Start date	Restricted?
com	Commercial	1985	No
edu	Educational institutions	1985	Yes
gov	Government	1985	Yes
int	International organizations	1988	Yes
mil	Military	1985	Yes
net	Network providers	1985	No
org	Non-profit organizations	1985	No
aero	Air transport	2001	Yes
biz	Businesses	2001	No
coop	Cooperatives	2001	Yes
info	Informational	2002	No
museum	Museums	2002	Yes
name	People	2002	No
pro	Professionals	2002	Yes
cat	Catalan	2005	Yes
jobs	Employment	2005	Yes
mobi	Mobile devices	2005	Yes
tel	Contact details	2005	Yes
travel	Travel industry	2005	Yes
xxx	Sex industry	2010	No

Figure 7-2. Generic top-level domains.

for example. It is for qualified professionals. But who is a professional? Doctors and lawyers clearly are professionals. But what about freelance photographers, piano teachers, magicians, plumbers, barbers, exterminators, tattoo artists, mercenaries, and prostitutes? Are these occupations eligible? According to whom?

There is also money in names. Tuvalu (the country) sold a lease on its *tv* domain for $50 million, all because the country code is well-suited to advertising television sites. Virtually every common (English) word has been taken in the *com* domain, along with the most common misspellings. Try household articles, animals, plants, body parts, etc. The practice of registering a domain only to turn around and sell it off to an interested party at a much higher price even has a name. It is called **cybersquatting**. Many companies that were slow off the mark when the Internet era began found their obvious domain names already taken when they tried to acquire them. In general, as long as no trademarks are being violated and no fraud is involved, it is first-come, first-served with names. Nevertheless, policies to resolve naming disputes are still being refined.

Each domain is named by the path upward from it to the (unnamed) root. The components are separated by periods (pronounced "dot"). Thus, the engineering department at Cisco might be *eng.cisco.com.*, rather than a UNIX-style name such as */com/cisco/eng*. Notice that this hierarchical naming means that *eng.cisco.com.* does not conflict with a potential use of *eng* in *eng.washington.edu.*, which might be used by the English department at the University of Washington.

Domain names can be either absolute or relative. An absolute domain name always ends with a period (e.g., *eng.cisco.com.*), whereas a relative one does not. Relative names have to be interpreted in some context to uniquely determine their true meaning. In both cases, a named domain refers to a specific node in the tree and all the nodes under it.

Domain names are case-insensitive, so *edu*, *Edu*, and *EDU* mean the same thing. Component names can be up to 63 characters long, and full path names must not exceed 255 characters.

In principle, domains can be inserted into the tree in either generic or country domains. For example, *cs.washington.edu* could equally well be listed under the *us* country domain as *cs.washington.wa.us*. In practice, however, most organizations in the United States are under generic domains, and most outside the United States are under the domain of their country. There is no rule against registering under multiple top-level domains. Large companies often do so (e.g., *sony.com*, *sony.net*, and *sony.nl*).

Each domain controls how it allocates the domains under it. For example, Japan has domains *ac.jp* and *co.jp* that mirror *edu* and *com*. The Netherlands does not make this distinction and puts all organizations directly under *nl*. Thus, all three of the following are university computer science departments:

1. *cs.washington.edu* (University of Washington, in the U.S.).
2. *cs.vu.nl* (Vrije Universiteit, in The Netherlands).
3. *cs.keio.ac.jp* (Keio University, in Japan).

To create a new domain, permission is required of the domain in which it will be included. For example, if a VLSI group is started at the University of Washington and wants to be known as *vlsi.cs.washington.edu*, it has to get permission from whoever manages *cs.washington.edu*. Similarly, if a new university is chartered, say, the University of Northern South Dakota, it must ask the manager of the *edu* domain to assign it *unsd.edu* (if that is still available). In this way, name conflicts are avoided and each domain can keep track of all its subdomains. Once a new domain has been created and registered, it can create subdomains, such as *cs.unsd.edu*, without getting permission from anybody higher up the tree.

Naming follows organizational boundaries, not physical networks. For example, if the computer science and electrical engineering departments are located in the same building and share the same LAN, they can nevertheless have distinct

domains. Similarly, even if computer science is split over Babbage Hall and Turing Hall, the hosts in both buildings will normally belong to the same domain.

7.1.2 Domain Resource Records

Every domain, whether it is a single host or a top-level domain, can have a set of **resource records** associated with it. These records are the DNS database. For a single host, the most common resource record is just its IP address, but many other kinds of resource records also exist. When a resolver gives a domain name to DNS, what it gets back are the resource records associated with that name. Thus, the primary function of DNS is to map domain names onto resource records.

A resource record is a five-tuple. Although they are encoded in binary for efficiency, in most expositions resource records are presented as ASCII text, one line per resource record. The format we will use is as follows:

```
Domain_name  Time_to_live  Class  Type  Value
```

The *Domain_name* tells the domain to which this record applies. Normally, many records exist for each domain and each copy of the database holds information about multiple domains. This field is thus the primary search key used to satisfy queries. The order of the records in the database is not significant.

The *Time_to_live* field gives an indication of how stable the record is. Information that is highly stable is assigned a large value, such as 86400 (the number of seconds in 1 day). Information that is highly volatile is assigned a small value, such as 60 (1 minute). We will come back to this point later when we have discussed caching.

The third field of every resource record is the *Class*. For Internet information, it is always *IN*. For non-Internet information, other codes can be used, but in practice these are rarely seen.

The *Type* field tells what kind of record this is. There are many kinds of DNS records. The important types are listed in Fig. 7-3.

An *SOA* record provides the name of the primary source of information about the name server's zone (described below), the email address of its administrator, a unique serial number, and various flags and timeouts.

The most important record type is the *A* (Address) record. It holds a 32-bit IPv4 address of an interface for some host. The corresponding *AAAA*, or "quad A," record holds a 128-bit IPv6 address. Every Internet host must have at least one IP address so that other machines can communicate with it. Some hosts have two or more network interfaces, in which case they will have two or more type *A* or *AAAA* resource records. Consequently, DNS can return multiple addresses for a single name.

A common record type is the *MX* record. It specifies the name of the host prepared to accept email for the specified domain. It is used because not every

Type	Meaning	Value
SOA	Start of authority	Parameters for this zone
A	IPv4 address of a host	32-Bit integer
AAAA	IPv6 address of a host	128-Bit integer
MX	Mail exchange	Priority, domain willing to accept email
NS	Name server	Name of a server for this domain
CNAME	Canonical name	Domain name
PTR	Pointer	Alias for an IP address
SPF	Sender policy framework	Text encoding of mail sending policy
SRV	Service	Host that provides it
TXT	Text	Descriptive ASCII text

Figure 7-3. The principal DNS resource record types.

machine is prepared to accept email. If someone wants to send email to, for example, *bill@microsoft.com*, the sending host needs to find some mail server located at *microsoft.com* that is willing to accept email. The *MX* record can provide this information.

Another important record type is the *NS* record. It specifies a name server for the domain or subdomain. This is a host that has a copy of the database for a domain. It is used as part of the process to look up names, which we will describe shortly.

CNAME records allow aliases to be created. For example, a person familiar with Internet naming in general and wanting to send a message to user *paul* in the computer science department at M.I.T. might guess that *paul@cs.mit.edu* will work. Actually, this address will not work, because the domain for M.I.T.'s computer science department is *csail.mit.edu*. However, as a service to people who do not know this, M.I.T. could create a *CNAME* entry to point people and programs in the right direction. An entry like this one might do the job:

```
cs.mit.edu   86400   IN   CNAME   csail.mit.edu
```

Like *CNAME*, *PTR* points to another name. However, unlike *CNAME*, which is really just a macro definition (i.e., a mechanism to replace one string by another), *PTR* is a regular DNS data type whose interpretation depends on the context in which it is found. In practice, it is nearly always used to associate a name with an IP address to allow lookups of the IP address and return the name of the corresponding machine. These are called **reverse lookups**.

SRV is a newer type of record that allows a host to be identified for a given service in a domain. For example, the Web server for *cs.washington.edu* could be identified as *cockatoo.cs.washington.edu*. This record generalizes the *MX* record that performs the same task but it is just for mail servers.

SPF is also a newer type of record. It lets a domain encode information about what machines in the domain will send mail to the rest of the Internet. This helps receiving machines check that mail is valid. If mail is being received from a machine that calls itself *dodgy* but the domain records say that mail will only be sent out of the domain by a machine called *smtp*, chances are that the mail is forged junk mail.

Last on the list, *TXT* records were originally provided to allow domains to identify themselves in arbitrary ways. Nowadays, they usually encode machine-readable information, typically the *SPF* information.

Finally, we have the *Value* field. This field can be a number, a domain name, or an ASCII string. The semantics depend on the record type. A short description of the *Value* fields for each of the principal record types is given in Fig. 7-3.

For an example of the kind of information one might find in the DNS database of a domain, see Fig. 7-4. This figure depicts part of a (hypothetical) database for the *cs.vu.nl* domain shown in Fig. 7-1. The database contains seven types of resource records.

```
; Authoritative data for cs.vu.nl
cs.vu.nl.        86400   IN   SOA      star boss (9527,7200,7200,241920,86400)
cs.vu.nl.        86400   IN   MX       1 zephyr
cs.vu.nl.        86400   IN   MX       2 top
cs.vu.nl.        86400   IN   NS       star
star             86400   IN   A        130.37.56.205
zephyr           86400   IN   A        130.37.20.10
top              86400   IN   A        130.37.20.11
www              86400   IN   CNAME    star.cs.vu.nl
ftp              86400   IN   CNAME    zephyr.cs.vu.nl

flits            86400   IN   A        130.37.16.112
flits            86400   IN   A        192.31.231.165
flits            86400   IN   MX       1 flits
flits            86400   IN   MX       2 zephyr
flits            86400   IN   MX       3 top

rowboat                  IN   A        130.37.56.201
                         IN   MX       1 rowboat
                         IN   MX       2 zephyr

little-sister            IN   A        130.37.62.23

laserjet                 IN   A        192.31.231.216
```

Figure 7-4. A portion of a possible DNS database for *cs.vu.nl*.

The first noncomment line of Fig. 7-4 gives some basic information about the domain, which will not concern us further. Then come two entries giving the first

and second places to try to deliver email sent to *person@cs.vu.nl*. The *zephyr* (a specific machine) should be tried first. If that fails, the *top* should be tried as the next choice. The next line identifies the name server for the domain as *star*.

After the blank line (added for readability) come lines giving the IP addresses for the *star*, *zephyr*, and *top*. These are followed by an alias, *www.cs.vu.nl*, so that this address can be used without designating a specific machine. Creating this alias allows *cs.vu.nl* to change its World Wide Web server without invalidating the address people use to get to it. A similar argument holds for *ftp.cs.vu.nl*.

The section for the machine *flits* lists two IP addresses and three choices are given for handling email sent to *flits.cs.vu.nl*. First choice is naturally the *flits* itself, but if it is down, the *zephyr* and *top* are the second and third choices.

The next three lines contain a typical entry for a computer, in this case, *rowboat.cs.vu.nl*. The information provided contains the IP address and the primary and secondary mail drops. Then comes an entry for a computer that is not capable of receiving mail itself, followed by an entry that is likely for a printer that is connected to the Internet.

7.1.3 Name Servers

In theory at least, a single name server could contain the entire DNS database and respond to all queries about it. In practice, this server would be so overloaded as to be useless. Furthermore, if it ever went down, the entire Internet would be crippled.

To avoid the problems associated with having only a single source of information, the DNS name space is divided into nonoverlapping **zones**. One possible way to divide the name space of Fig. 7-1 is shown in Fig. 7-5. Each circled zone contains some part of the tree.

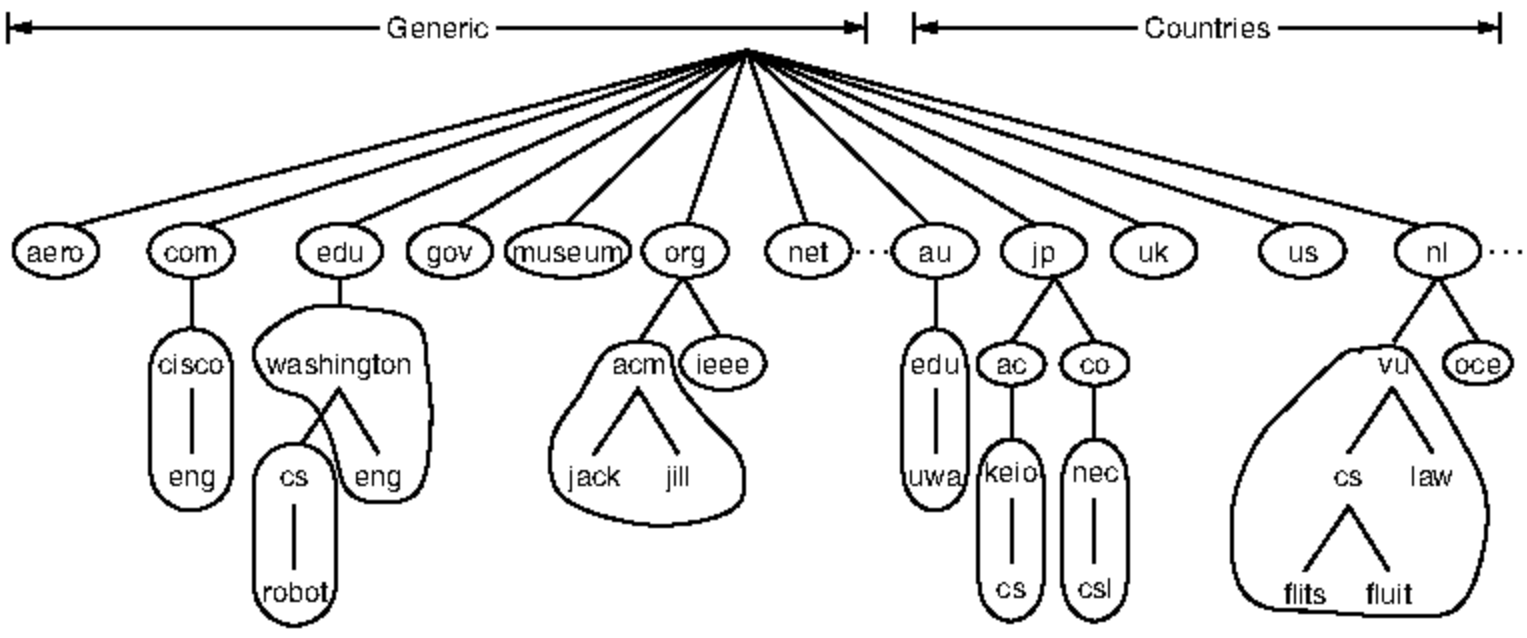

Figure 7-5. Part of the DNS name space divided into zones (which are circled).

Where the zone boundaries are placed within a zone is up to that zone's administrator. This decision is made in large part based on how many name servers are desired, and where. For example, in Fig. 7-5, the University of Washington has a zone for *washington.edu* that handles *eng.washington.edu* but does not handle *cs.washington.edu*. That is a separate zone with its own name servers. Such a decision might be made when a department such as English does not wish to run its own name server, but a department such as Computer Science does.

Each zone is also associated with one or more name servers. These are hosts that hold the database for the zone. Normally, a zone will have one primary name server, which gets its information from a file on its disk, and one or more secondary name servers, which get their information from the primary name server. To improve reliability, some of the name servers can be located outside the zone.

The process of looking up a name and finding an address is called **name resolution**. When a resolver has a query about a domain name, it passes the query to a local name server. If the domain being sought falls under the jurisdiction of the name server, such as *top.cs.vu.nl* falling under *cs.vu.nl*, it returns the authoritative resource records. An **authoritative record** is one that comes from the authority that manages the record and is thus always correct. Authoritative records are in contrast to **cached records**, which may be out of date.

What happens when the domain is remote, such as when *flits.cs.vu.nl* wants to find the IP address of *robot.cs.washington.edu* at UW (University of Washington)? In this case, and if there is no cached information about the domain available locally, the name server begins a remote query. This query follows the process shown in Fig. 7-6. Step 1 shows the query that is sent to the local name server. The query contains the domain name sought, the type (*A*), and the class(*IN*).

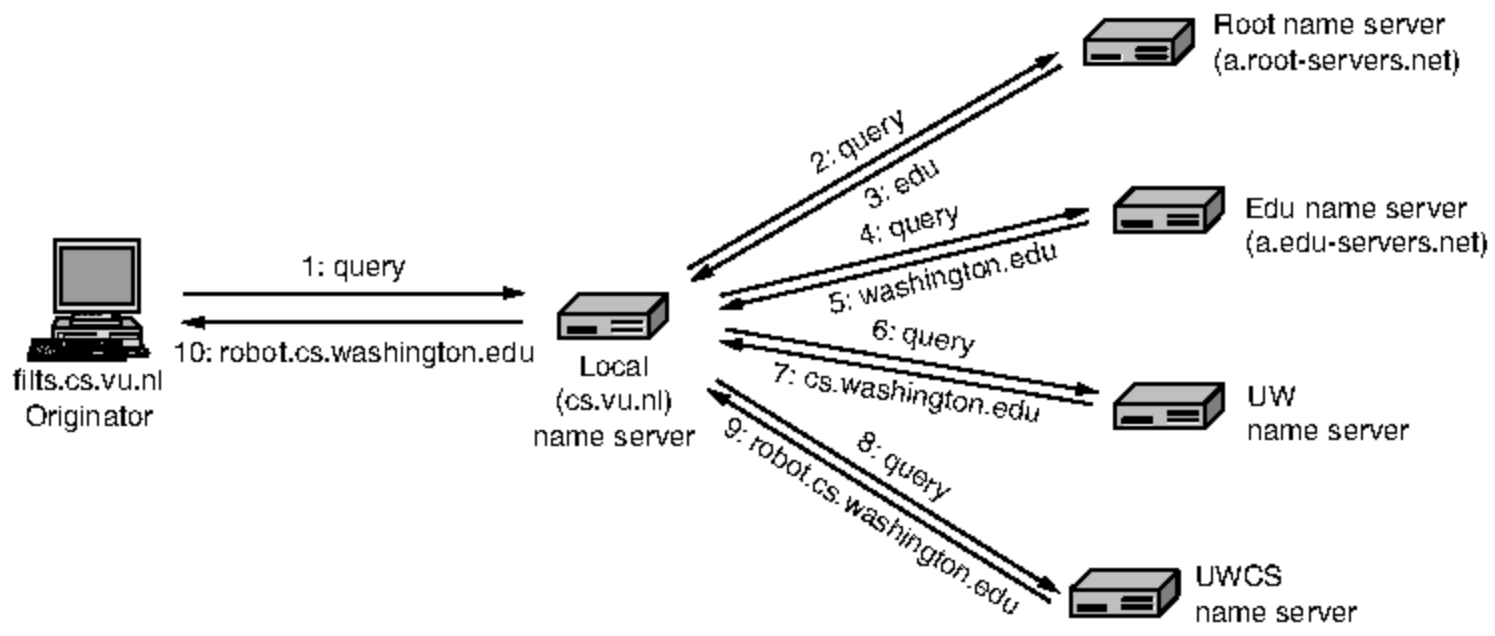

Figure 7-6. Example of a resolver looking up a remote name in 10 steps.

The next step is to start at the top of the name hierarchy by asking one of the **root name servers**. These name servers have information about each top-level

domain. This is shown as step 2 in Fig. 7-6. To contact a root server, each name server must have information about one or more root name servers. This information is normally present in a system configuration file that is loaded into the DNS cache when the DNS server is started. It is simply a list of *NS* records for the root and the corresponding *A* records.

There are 13 root DNS servers, unimaginatively called *a-root-servers.net* through *m.root-servers.net*. Each root server could logically be a single computer. However, since the entire Internet depends on the root servers, they are powerful and heavily replicated computers. Most of the servers are present in multiple geographical locations and reached using anycast routing, in which a packet is delivered to the nearest instance of a destination address; we described anycast in Chap. 5 The replication improves reliability and performance.

The root name server is unlikely to know the address of a machine at UW, and probably does not know the name server for UW either. But it must know the name server for the *edu* domain, in which *cs.washington.edu* is located. It returns the name and IP address for that part of the answer in step 3.

The local name server then continues its quest. It sends the entire query to the *edu* name server (*a.edu-servers.net*). That name server returns the name server for UW. This is shown in steps 4 and 5. Closer now, the local name server sends the query to the UW name server (step 6). If the domain name being sought was in the English department, the answer would be found, as the UW zone includes the English department. But the Computer Science department has chosen to run its own name server. The query returns the name and IP address of the UW Computer Science name server (step 7).

Finally, the local name server queries the UW Computer Science name server (step 8). This server is authoritative for the domain *cs.washington.edu*, so it must have the answer. It returns the final answer (step 9), which the local name server forwards as a response to *flits.cs.vu.nl* (step 10). The name has been resolved.

You can explore this process using standard tools such as the *dig* program that is installed on most UNIX systems. For example, typing

```
dig@a.edu-servers.net robot.cs.washington.edu
```

will send a query for *robot.cs.washington.edu* to the *a.edu-servers.net* name server and print out the result. This will show you the information obtained in step 4 in the example above, and you will learn the name and IP address of the UW name servers.

There are three technical points to discuss about this long scenario. First, two different query mechanisms are at work in Fig. 7-6. When the host *flits.cs.vu.nl* sends its query to the local name server, that name server handles the resolution on behalf of *flits* until it has the desired answer to return. It does *not* return partial answers. They might be helpful, but they are not what the query was seeking. This mechanism is called a **recursive query**.

On the other hand, the root name server (and each subsequent name server) does not recursively continue the query for the local name server. It just returns a partial answer and moves on to the next query. The local name server is responsible for continuing the resolution by issuing further queries. This mechanism is called an **iterative query**.

One name resolution can involve both mechanisms, as this example showed. A recursive query may always seem preferable, but many name servers (especially the root) will not handle them. They are too busy. Iterative queries put the burden on the originator. The rationale for the local name server supporting a recursive query is that it is providing a service to hosts in its domain. Those hosts do not have to be configured to run a full name server, just to reach the local one.

The second point is caching. All of the answers, including all the partial answers returned, are cached. In this way, if another *cs.vu.nl* host queries for *robot.cs.washington.edu* the answer will already be known. Even better, if a host queries for a different host in the same domain, say *galah.cs.washington.edu*, the query can be sent directly to the authoritative name server. Similarly, queries for other domains in *washington.edu* can start directly from the *washington.edu* name server. Using cached answers greatly reduces the steps in a query and improves performance. The original scenario we sketched is in fact the worst case that occurs when no useful information is cached.

However, cached answers are not authoritative, since changes made at *cs.washington.edu* will not be propagated to all the caches in the world that may know about it. For this reason, cache entries should not live too long. This is the reason that the *Time_to_live* field is included in each resource record. It tells remote name servers how long to cache records. If a certain machine has had the same IP address for years, it may be safe to cache that information for 1 day. For more volatile information, it might be safer to purge the records after a few seconds or a minute.

The third issue is the transport protocol that is used for the queries and responses. It is UDP. DNS messages are sent in UDP packets with a simple format for queries, answers, and name servers that can be used to continue the resolution. We will not go into the details of this format. If no response arrives within a short time, the DNS client repeats the query, trying another server for the domain after a small number of retries. This process is designed to handle the case of the server being down as well as the query or response packet getting lost. A 16-bit identifier is included in each query and copied to the response so that a name server can match answers to the corresponding query, even if multiple queries are outstanding at the same time.

Even though its purpose is simple, it should be clear that DNS is a large and complex distributed system that is comprised of millions of name servers that work together. It forms a key link between human-readable domain names and the IP addresses of machines. It includes replication and caching for performance and reliability and is designed to be highly robust.

We have not covered security, but as you might imagine, the ability to change the name-to-address mapping can have devastating consequences if done maliciously. For that reason, security extensions called DNSSEC have been developed for DNS. We will describe them in Chap. 8.

There is also application demand to use names in more flexible ways, for example, by naming content and resolving to the IP address of a nearby host that has the content. This fits the model of searching for and downloading a movie. It is the movie that matters, not the computer that has a copy of it, so all that is wanted is the IP address of any nearby computer that has a copy of the movie. Content distribution networks are one way to accomplish this mapping. We will describe how they build on the DNS later in this chapter, in Sec. 7.5.

7.2 ELECTRONIC MAIL

Electronic mail, or more commonly **email**, has been around for over three decades. Faster and cheaper than paper mail, email has been a popular application since the early days of the Internet. Before 1990, it was mostly used in academia. During the 1990s, it became known to the public at large and grew exponentially, to the point where the number of emails sent per day now is vastly more than the number of **snail mail** (i.e., paper) letters. Other forms of network communication, such as instant messaging and voice-over-IP calls have expanded greatly in use over the past decade, but email remains the workhorse of Internet communication. It is widely used within industry for intracompany communication, for example, to allow far-flung employees all over the world to cooperate on complex projects. Unfortunately, like paper mail, the majority of email—some 9 out of 10 messages—is junk mail or **spam** (McAfee, 2010).

Email, like most other forms of communication, has developed its own conventions and styles. It is very informal and has a low threshold of use. People who would never dream of calling up or even writing a letter to a Very Important Person do not hesitate for a second to send a sloppily written email to him or her. By eliminating most cues associated with rank, age, and gender, email debates often focus on content, not status. With email, a brilliant idea from a summer student can have more impact than a dumb one from an executive vice president.

Email is full of jargon such as BTW (By The Way), ROTFL (Rolling On The Floor Laughing), and IMHO (In My Humble Opinion). Many people also use little ASCII symbols called **smileys**, starting with the ubiquitous ":-)". Rotate the book 90 degrees clockwise if this symbol is unfamiliar. This symbol and other **emoticons** help to convey the tone of the message. They have spread to other terse forms of communication, such as instant messaging.

The email protocols have evolved during the period of their use, too. The first email systems simply consisted of file transfer protocols, with the convention that the first line of each message (i.e., file) contained the recipient's address. As time

went on, email diverged from file transfer and many features were added, such as the ability to send one message to a list of recipients. Multimedia capabilities became important in the 1990s to send messages with images and other non-text material. Programs for reading email became much more sophisticated too, shifting from text-based to graphical user interfaces and adding the ability for users to access their mail from their laptops wherever they happen to be. Finally, with the prevalence of spam, mail readers and the mail transfer protocols must now pay attention to finding and removing unwanted email.

In our description of email, we will focus on the way that mail messages are moved between users, rather than the look and feel of mail reader programs. Nevertheless, after describing the overall architecture, we will begin with the user-facing part of the email system, as it is familiar to most readers.

7.2.1 Architecture and Services

In this section, we will provide an overview of how email systems are organized and what they can do. The architecture of the email system is shown in Fig. 7-7. It consists of two kinds of subsystems: the **user agents**, which allow people to read and send email, and the **message transfer agents**, which move the messages from the source to the destination. We will also refer to message transfer agents informally as **mail servers**.

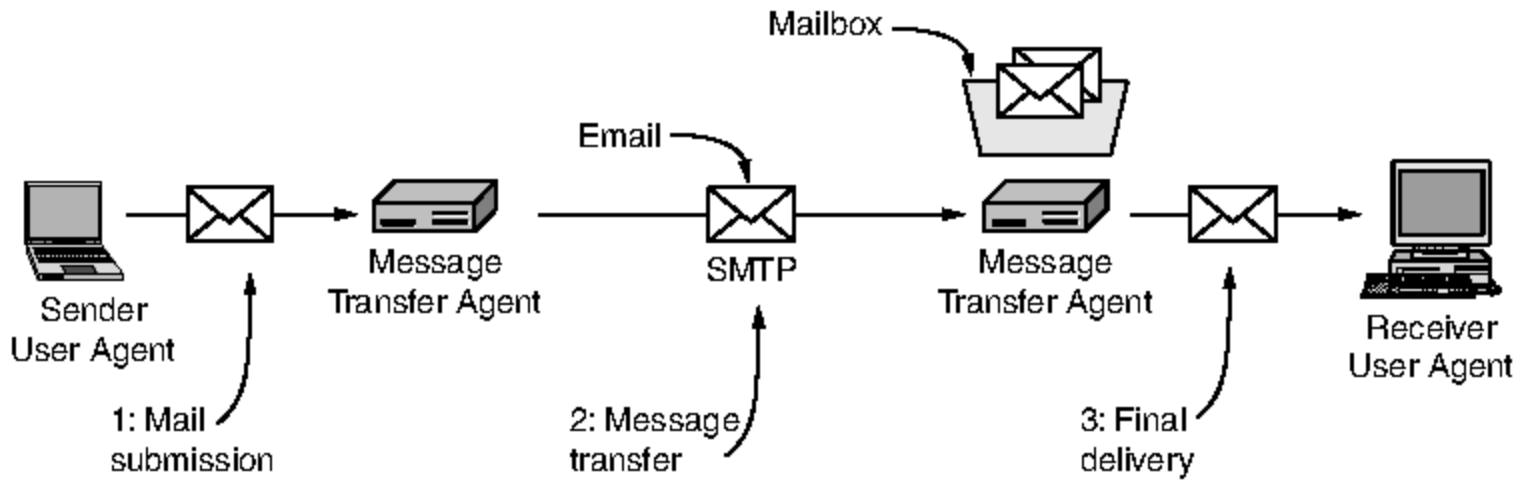

Figure 7-7. Architecture of the email system.

The user agent is a program that provides a graphical interface, or sometimes a text- and command-based interface that lets users interact with the email system. It includes a means to compose messages and replies to messages, display incoming messages, and organize messages by filing, searching, and discarding them. The act of sending new messages into the mail system for delivery is called **mail submission**.

Some of the user agent processing may be done automatically, anticipating what the user wants. For example, incoming mail may be filtered to extract or

deprioritize messages that are likely spam. Some user agents include advanced features, such as arranging for automatic email responses ("I'm having a wonderful vacation and it will be a while before I get back to you"). A user agent runs on the same computer on which a user reads her mail. It is just another program and may be run only some of the time.

The message transfer agents are typically system processes. They run in the background on mail server machines and are intended to be always available. Their job is to automatically move email through the system from the originator to the recipient with **SMTP** (**Simple Mail Transfer Protocol**). This is the message transfer step.

SMTP was originally specified as RFC 821 and revised to become the current RFC 5321. It sends mail over connections and reports back the delivery status and any errors. Numerous applications exist in which confirmation of delivery is important and may even have legal significance ("Well, Your Honor, my email system is just not very reliable, so I guess the electronic subpoena just got lost somewhere'').

Message transfer agents also implement **mailing lists**, in which an identical copy of a message is delivered to everyone on a list of email addresses. Other advanced features are carbon copies, blind carbon copies, high-priority email, secret (i.e., encrypted) email, alternative recipients if the primary one is not currently available, and the ability for assistants to read and answer their bosses' email.

Linking user agents and message transfer agents are the concepts of mailboxes and a standard format for email messages. **Mailboxes** store the email that is received for a user. They are maintained by mail servers. User agents simply present users with a view of the contents of their mailboxes. To do this, the user agents send the mail servers commands to manipulate the mailboxes, inspecting their contents, deleting messages, and so on. The retrieval of mail is the final delivery (step 3) in Fig. 7-7. With this architecture, one user may use different user agents on multiple computers to access one mailbox.

Mail is sent between message transfer agents in a standard format. The original format, RFC 822, has been revised to the current RFC 5322 and extended with support for multimedia content and international text. This scheme is called MIME and will be discussed later. People still refer to Internet email as RFC 822, though.

A key idea in the message format is the distinction between the **envelope** and its contents. The envelope encapsulates the message. It contains all the information needed for transporting the message, such as the destination address, priority, and security level, all of which are distinct from the message itself. The message transport agents use the envelope for routing, just as the post office does.

The message inside the envelope consists of two separate parts: the **header** and the **body**. The header contains control information for the user agents. The body is entirely for the human recipient. None of the agents care much about it. Envelopes and messages are illustrated in Fig. 7-8.

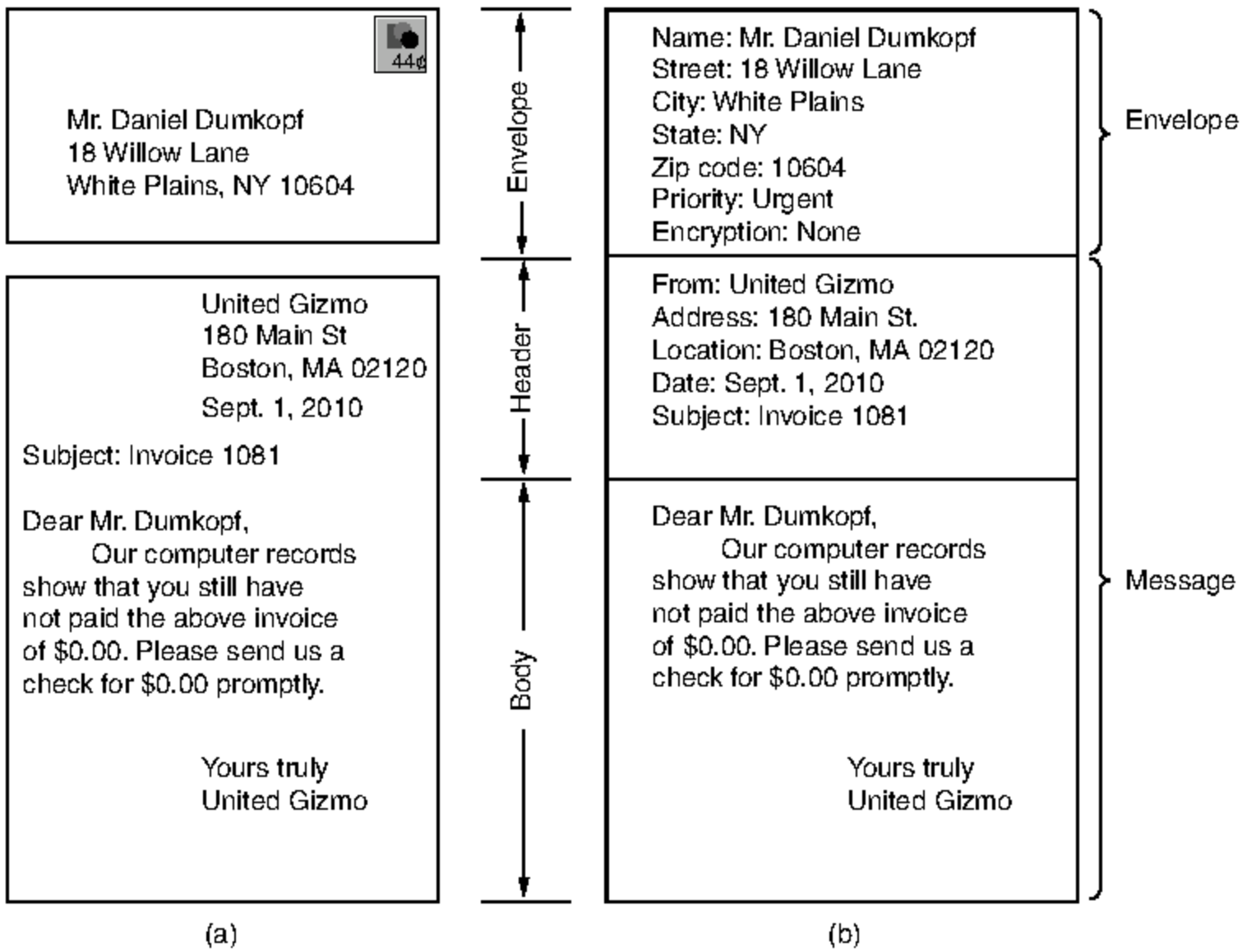

Figure 7-8. Envelopes and messages. (a) Paper mail. (b) Electronic mail.

We will examine the pieces of this architecture in more detail by looking at the steps that are involved in sending email from one user to another. This journey starts with the user agent.

7.2.2 The User Agent

A user agent is a program (sometimes called an **email reader**) that accepts a variety of commands for composing, receiving, and replying to messages, as well as for manipulating mailboxes. There are many popular user agents, including Google gmail, Microsoft Outlook, Mozilla Thunderbird, and Apple Mail. They can vary greatly in their appearance. Most user agents have a menu- or icon-driven graphical interface that requires a mouse, or a touch interface on smaller mobile devices. Older user agents, such as Elm, mh, and Pine, provide text-based interfaces and expect one-character commands from the keyboard. Functionally, these are the same, at least for text messages.

The typical elements of a user agent interface are shown in Fig. 7-9. Your mail reader is likely to be much flashier, but probably has equivalent functions.