

## 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 rules of thumb gained from hard experience and present examples taken from the real world. We have intentionally delayed this discussion until we studied the transport layer in TCP in order to be able to use TCP as an example in various places.

The transport layer is not the only place performance issues arise. We saw some of them in the network layer in the previous chapter. Nevertheless, the network layer tends to be largely concerned with routing and congestion control. The broader, system-oriented issues tend to be transport related, so this chapter is an appropriate place to examine them.

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

1. Performance problems.
2. Measuring network performance.
3. System design for better performance.
4. Fast TPDU processing.
5. Protocols for future high-performance networks.

As an aside, we need a generic name for the units exchanged by transport entities. The TCP term, segment, is confusing at best and is never used outside the TCP world in this context. The ATM terms (CS-PDU, SAR-PDU, and CPCS-PDU) are specific to ATM. Packets clearly refer to the network layer, and messages belong to the application layer. For lack of a standard term, we will go back to calling the units exchanged by transport entities TPDU. When we mean both TPDU and packet together, we will use packet as the collective term, as in “The CPU must be fast enough to process incoming packets in real time.” By this we mean both the network layer packet and the TPDU encapsulated in it.

### 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 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 CPU 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. For example, if a TPDU 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 TPDU is broadcast to 10,000 machines: each one might send back an error message. The resulting **broadcast storm** could cripple the network. UDP suffered from this problem until the protocol was changed to cause hosts to refrain from responding to errors in UDP TPDU sent to broadcast addresses.

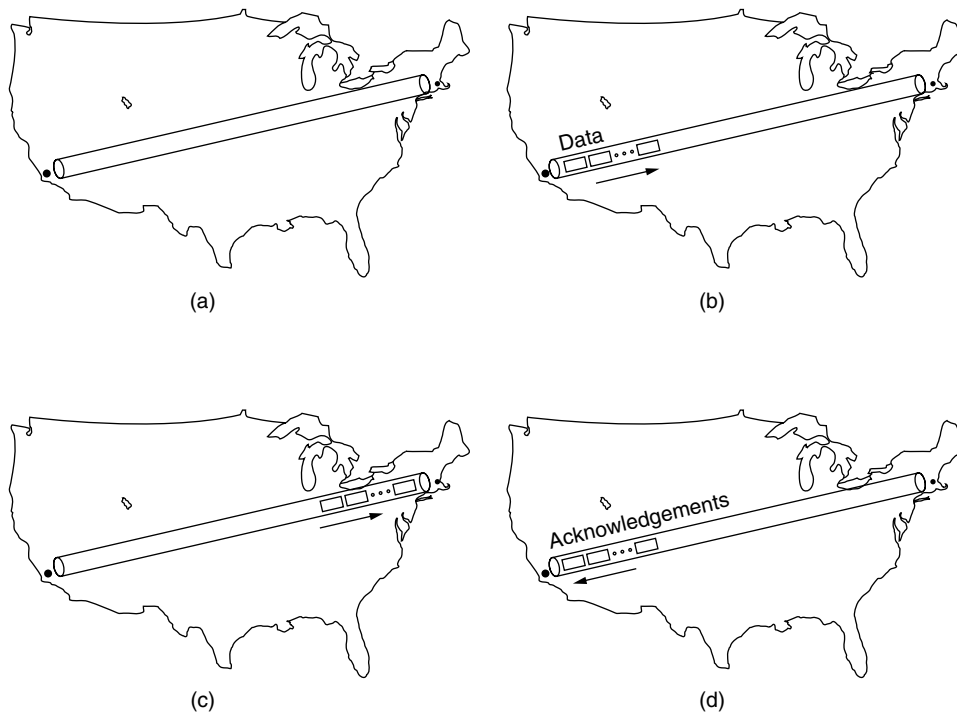
A second example of synchronous overload is what happens after an electrical power failure. When the power comes back on, all the machines simultaneously jump to their ROMs to start rebooting. A typical reboot sequence might require first going to some (DHCP) server to learn one's true identity, and then to some file server to get a copy of the operating system. If hundreds of machines 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, overruns will occur and TPDU will be lost. Similarly, if the scheduling algorithm does not give a high enough priority to processing incoming TPDU, some of them may be lost.

Another tuning issue is setting timeouts correctly. When a TPDU is sent, a timer is typically set to guard against loss of the TPDU. 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 TPDU 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 before giving up.

Gigabit networks bring with them new performance problems. 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. 6-41(a). Only 500  $\mu$ sec later, in Fig. 6-41(b), all the TPDU are out on the fiber. The lead TPDU 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.

After 20 msec, the lead TPDU hits Boston, as shown in Fig. 6-41(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 0.5 msec out of 40, the efficiency is about 1.25 percent. This situation is typical of older protocols running over gigabit lines.



**Figure 6-41.** The state of transmitting one megabit from San Diego to Boston.  
 (a) At  $t = 0$ . (b) After  $500 \mu\text{sec}$ . (c) After  $20 \text{ msec}$ . (d) After  $40 \text{ msec}$ .

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. 6-41 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 percent efficiency: it is only 1.25 percent 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, preferably somewhat larger since the receiver may not respond instantly. For a transcontinental gigabit line, at least 5 megabytes are required.

If the efficiency is terrible for sending a megabit, imagine what it is like for a short request of a few hundred bytes. Unless some other use can be found for the

line while the first client is waiting for its reply, a gigabit line is no better than a megabit line, just more expensive.

Another performance problem that occurs with time-critical applications like audio and video is jitter. Having a short mean transmission time is not enough. A small standard deviation is also required. Achieving a short mean transmission time along with a small standard deviation demands a serious engineering effort.

## 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. The discussion below is based on the work of Mogul (1993).

The basic loop used to improve network performance contains the following steps:

1. Measure the relevant network parameters and performance.
2. Try to understand what is going on.
3. Change one parameter.

These steps are repeated until the performance is good enough or it is clear that the last drop of improvement has been squeezed out.

Measurements can be made in many ways and at many locations (both physically and in the protocol stack). 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 TPDU 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 TPDU's). 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. Below we list a few of them. 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 TPDU, but repeat the measurement, say, one million times and take the average. 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 system loads 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 10, 11, 1, and 2 o'clock, but no congestion at noon (when all the users are away at lunch).

### **Be Careful When Using a Coarse-Grained Clock**

Computer clocks work 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.)

To measure the time to send a TPDU, for example, the system clock (say, in milliseconds) should be read out when the transport layer code is entered and again when it is exited. If the true TPDU send time is 300  $\mu$ sec, 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 added up and divided by one million, the mean time will be accurate to better than 1  $\mu$ sec.

### **Be Sure That Nothing Unexpected Is Going On during Your Tests**

Making measurements on a university system the day some major lab project has to be turned in may give different results than if made the next day. Likewise, if some researcher has decided to run a video conference over your network during your tests, you may get a biased result. It is best to run tests on an idle system 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 precisely when the automatic backup program begins copying all the disks to tape. Furthermore, there might be heavy traffic for your wonderful World Wide Web pages from distant time zones.

### **Caching Can Wreak Havoc with Measurements**

The obvious way to measure file transfer times is to open a large file, read the whole thing, close it, and see how long it takes. Then repeat the measurement many more times to get a good average. The trouble is, the system may cache the file, so only the first measurement actually involves network traffic. The rest are just reads from the local cache. The results from such a measurement are essentially worthless (unless you want to measure cache performance).

Often you can get around caching by simply overflowing the cache. For example, if the cache is 10 MB, the test loop could open, read, and close two 10-

MB files on each pass, in an attempt to force the cache hit rate to 0. Still, caution is advised unless you are absolutely sure you understand the caching algorithm.

Buffering can have a similar effect. One popular TCP/IP performance utility program has been known to report that UDP can achieve a performance substantially higher than the physical line 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 utility thinks they have all been transmitted.

### **Understand What You Are Measuring**

When you measure the time to read a remote file, your measurements depend on the network, the operating systems on both the client and server, the particular hardware interface boards used, their drivers, and other factors. If the measurements are done carefully, you will ultimately discover the file transfer time for the configuration you are using. If your goal is to tune this particular configuration, these measurements are fine.

However, if you are making similar measurements on three different systems in order to choose which network interface board to buy, your results could be thrown off completely by the fact that one of the network drivers is truly awful and is only getting 10 percent of the performance of the board.

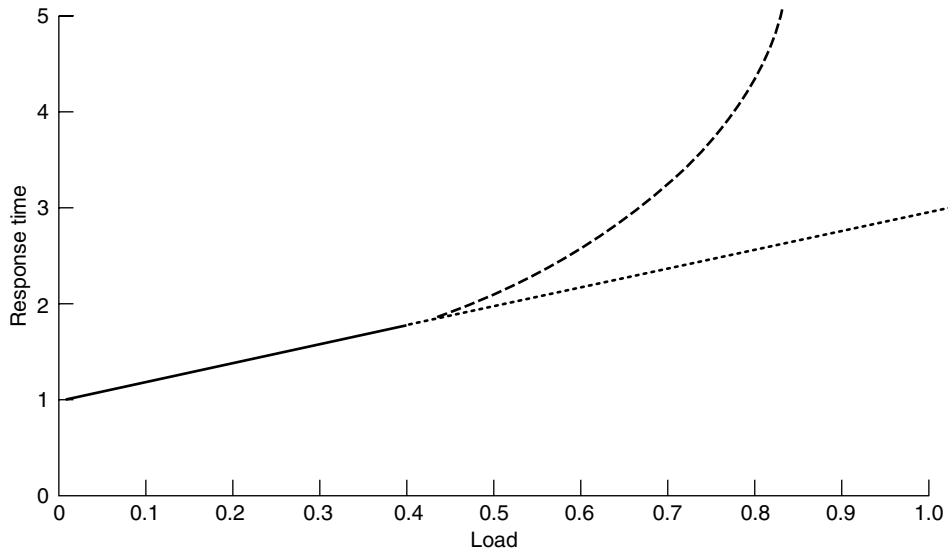
### **Be Careful about Extrapolating the Results**

Suppose that you make measurements of something with simulated network loads running from 0 (idle) to 0.4 (40 percent of capacity), as shown by the data points and solid line through them in Fig. 6-42. It may be tempting to extrapolate linearly, as shown by the dotted line. However, many queuing results involve a factor of  $1/(1 - \rho)$ , where  $\rho$  is the load, so the true values may look more like the dashed line, which rises much faster than linearly.

### **6.6.3 System Design for Better Performance**

Measuring and tinkering can often 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.

In this section, we will present some rules of thumb based on hard experience with many networks. These rules relate to system design, not just network design, since the software and operating system are often more important than the routers and interface boards. Most of these ideas have been common knowledge to net-



**Figure 6-42.** Response as a function of load.

work designers for years and have been passed on from generation to generation by word of mouth. They were first stated explicitly by Mogul (1993); our treatment largely follows his. Another relevant source is (Metcalfe, 1993).

### **Rule #1: CPU Speed Is More Important Than Network Speed**

Long experience has shown that in nearly all networks, operating system and protocol overhead dominate actual time on the wire. For example, in theory, the minimum RPC time on an Ethernet is 102  $\mu$ sec, corresponding to a minimum (64-byte) request followed by a minimum (64-byte) reply. In practice, overcoming the software overhead and getting the RPC time anywhere near there is a substantial achievement.

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

### **Rule #2: Reduce Packet Count to Reduce Software Overhead**

Processing a TPDU has a certain amount of overhead per TPDU (e.g., header processing) and a certain amount of processing per byte (e.g., doing the checksum). When 1 million bytes are being sent, the per-byte overhead is the same no

matter what the TPDU size is. However, using 128-byte TPDU's means 32 times as much per-TPDU overhead as using 4-KB TPDU's. This overhead adds up fast.

In addition to the TPDU overhead, there is overhead in the lower layers to consider. Each arriving packet causes an interrupt. On a modern pipelined processor, each interrupt breaks the CPU pipeline, interferes with the cache, requires a change to the memory management context, and forces a substantial number of CPU registers to be saved. An  $n$ -fold reduction in TPDU's sent thus reduces the interrupt and packet overhead by a factor of  $n$ .

This observation argues for collecting a substantial amount of data before transmission in order to reduce interrupts at the other side. Nagle's algorithm and Clark's solution to the silly window syndrome are attempts to do precisely this.

### **Rule #3: Minimize Context Switches**

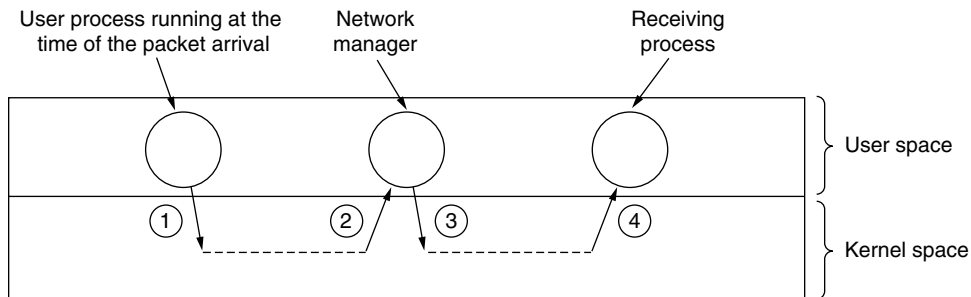
Context switches (e.g., from kernel mode to user mode) are deadly. They have the same bad properties as interrupts, the worst being a long series of initial cache misses. Context switches can be reduced by having the library procedure that sends data do internal buffering until it has a substantial amount of them. Similarly, on the receiving side, small incoming TPDU's 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 many 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. 6-43. All these context switches on each packet are very wasteful of CPU time and will have a devastating effect on network performance.

### **Rule #4: Minimize Copying**

Even worse than multiple context switches are multiple copies. It is not unusual for an incoming packet to be copied three or four times before the TPDU enclosed in it is delivered. After a packet is received by the network interface in a special on-board hardware buffer, it is typically copied to a kernel buffer. From there it is copied to a network layer buffer, then to a transport layer buffer, and finally to the receiving application process.

A clever operating system will copy a word at a time, but it is not unusual to require about five instructions per word (a load, a store, incrementing an index



**Figure 6-43.** Four context switches to handle one packet with a user-space network manager.

register, a test for end-of-data, and a conditional branch). Making three copies of each packet at five instructions per 32-bit word copied requires  $15/4$  or about four instructions per byte copied. On a 500-MIPS CPU, an instruction takes 2 nsec so each byte needs 8 nsec of processing time or about 1 nsec per bit, giving a maximum rate of about 1 Gbps. When overhead for header processing, interrupt handling, and context switches is factored in, 500 Mbps might be achievable, and we have not even considered the actual processing of the data. Clearly, handling a 10-Gbps Ethernet running at full blast is out of the question.

In fact, probably a 500-Mbps line cannot be handled at full speed either. In the computation above, we have assumed that a 500-MIPS machine can execute any 500 million instructions/sec. In reality, machines can only run at such speeds if they are not referencing memory. Memory operations are often a factor of ten slower than register-register instructions (i.e., 20 nsec/instruction). If 20 percent of the instructions actually reference memory (i.e., are cache misses), which is likely when touching incoming packets, the average instruction execution time is 5.6 nsec ( $0.8 \times 2 + 0.2 \times 20$ ). With four instructions/byte, we need 22.4 nsec/byte, or 2.8 nsec/bit, which gives about 357 Mbps. Factoring in 50 percent overhead gives us 178 Mbps. Note that hardware assistance will not help here. The problem is too much copying by the operating system.

### **Rule #5: You Can Buy More Bandwidth but Not Lower Delay**

The next three rules deal with communication, rather than protocol processing. The first rule states that if you want more bandwidth, you can just buy it. Putting a second fiber next to the first one doubles the bandwidth but does nothing to reduce the delay. Making the delay shorter requires improving the protocol software, the operating system, or the network interface. Even if all of these improvements are made, the delay will not be reduced if the bottleneck is the transmission time.

**Rule #6: 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. 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.

**Rule #7: 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.

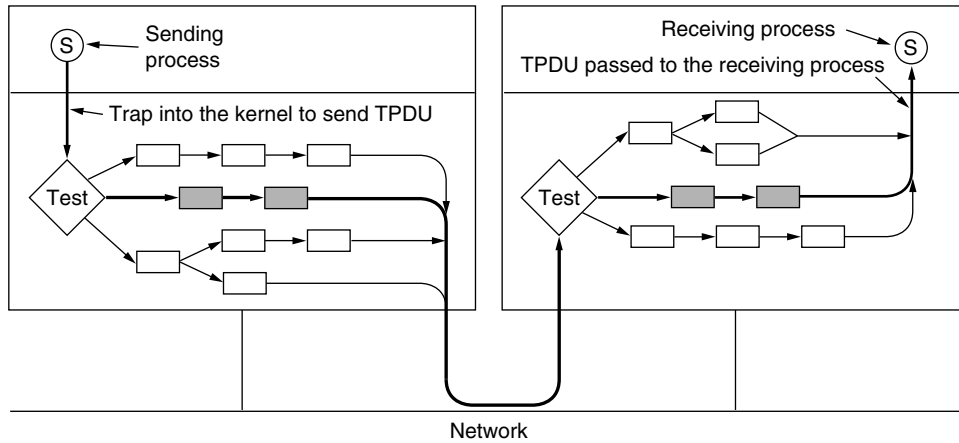
The way to avoid extra work is to be careful that timers are set a little bit on the conservative side. A timer that takes too long to expire adds a small amount of extra delay to one connection in the (unlikely) event of a TPDU being lost. A timer that goes off when it should not have uses up scarce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

**6.6.4 Fast TPDU Processing**

The moral of the story above is that the main obstacle to fast networking is protocol software. In this section we will look at some ways to speed up this software. For more information, see (Clark et al., 1989; and Chase et al., 2001).

TPDU processing overhead has two components: overhead per TPDU and overhead per byte. Both must be attacked. The key to fast TPDU processing is to separate out the normal case (one-way data transfer) and handle it specially. Although a sequence of special TPDU is needed to get into the *ESTABLISHED* state, once there, TPDU 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. 6-44, 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 TPDU 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.

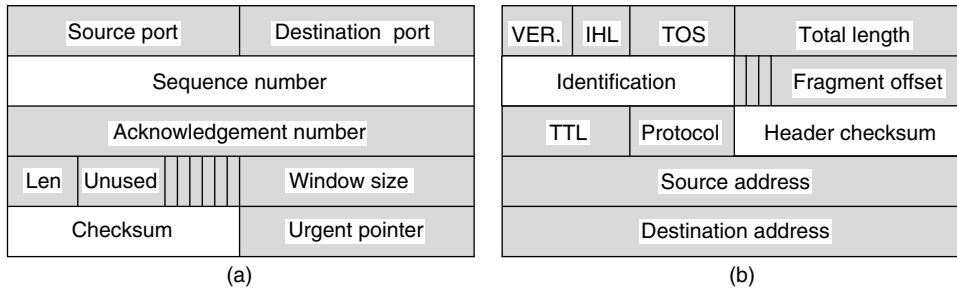


**Figure 6-44.** The fast path from sender to receiver is shown with a heavy line. The processing steps on this path are shaded.

In the usual case, the headers of consecutive data TPDU's 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 TPDU to TPDU are then overwritten in the buffer. Frequently, these fields are easily derived from state variables, such as the next sequence number. A pointer to the full TPDU 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. 6-44). 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. 6-45(a) shows the TCP header. The fields that are the same between consecutive TPDU's 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 TPDU. IP then copies its five-word prototype header [see Fig. 6-45(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. 6-44. Step 1 is locating the connection record for the incoming TPDU. For TCP, the 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.



**Figure 6-45.** (a) TCP header. (b) IP header. In both cases, the shaded fields are taken from the prototype without change.

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 percent. Other lookup heuristics are described in (McKenney and Dove, 1992).

The TPDU is then checked to see if it is a normal one: the state is *ESTABLISHED*, neither side is trying to close the connection, the TPDU 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.

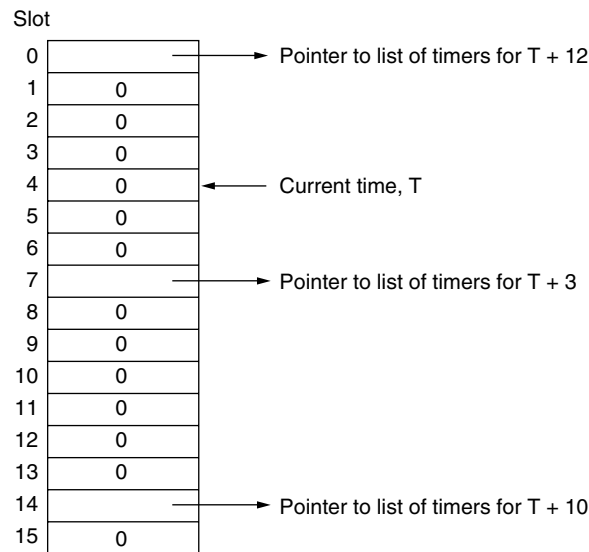
The fast path updates the connection record and copies the data to the user. While it is copying, it also computes the checksum, eliminating an extra pass over the data. If the checksum is correct, the connection record is updated and an acknowledgement is sent back. The general scheme of first making 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 percent 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 TPDU loss, but most TPDUs arrive correctly and their acknowledgements also 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. In this scheme, inserting and deleting timers are expensive operations, with execution times proportional to the length of the list.

A 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. 6-46. 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. 6-46 cannot accommodate timers beyond  $T + 15$ .



**Figure 6-46.** 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 in (Varghese and Lauck, 1987).

### 6.6.5 Protocols for Gigabit Networks

At the start of the 1990s, gigabit networks began to appear. People's first reaction was to use the old protocols on them, but various problems quickly arose. In this section we will discuss some of these problems and the directions new protocols are taking to solve them as we move toward ever faster networks.

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 sender can cycle through the sequence space while old packets still exist. RFC 1323 provides an escape hatch, though.

The problem is that many protocol designers simply assumed, without stating it, that the time 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.

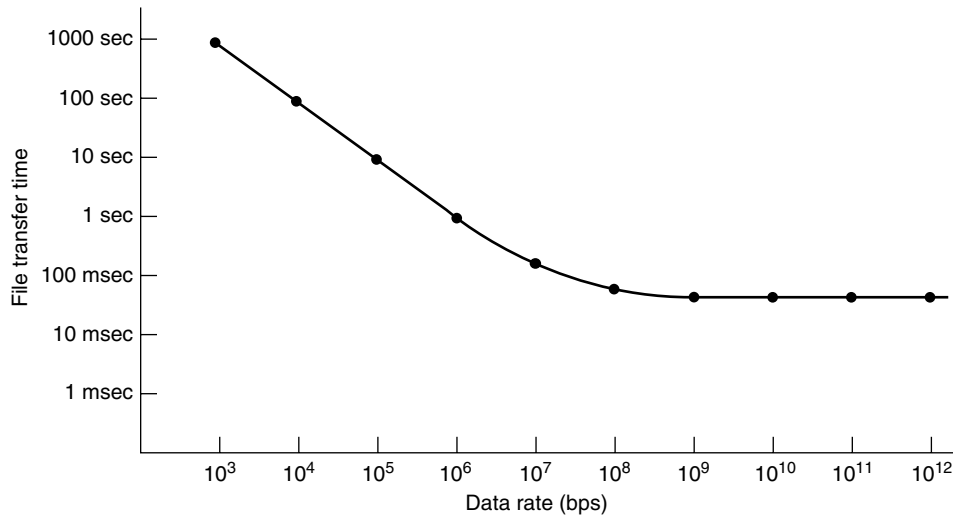
A second problem is that communication speeds have improved much faster than computing speeds. (Note to computer engineers: Go out and beat those communication engineers! We are counting on you.) In the 1970s, the ARPANET ran at 56 kbps and had computers that ran at about 1 MIPS. Packets were 1008 bits, so the ARPANET was capable of delivering about 56 packets/sec. With almost 18 msec available per packet, a host could afford to spend 18,000 instructions processing a packet. Of course, doing so would soak up the entire CPU, but it could devote 9000 instructions per packet and still have half the CPU left to do real work.

Compare these numbers to 1000-MIPS computers exchanging 1500-byte packets over a gigabit line. Packets can flow in at a rate of over 80,000 per second, so packet processing must be completed in 6.25  $\mu$ sec if we want to reserve half the CPU for applications. In 6.25  $\mu$ sec, a 1000-MIPS computer can execute 6250 instructions, only 1/3 of what the ARPANET hosts had available. Furthermore, modern RISC instructions do less per instruction than the old CISC instructions did, so the problem is even worse than it appears. The conclusion is this: there is less time available for protocol processing than there used to be, so protocols must become simpler.

A third problem is that the go back n protocol performs poorly on lines with a large bandwidth-delay product. Consider, for example, a 4000-km line operating at 1 Gbps. The round-trip transmission time is 40 msec, in which time a sender can transmit 5 megabytes. 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 megabytes worth of packets that came afterward. Clearly, this is a massive waste of resources.

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. 6-47 we show the time it takes to transfer a 1-megabit 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 6-47.** Time to transfer and acknowledge a 1-megabit file over a 4000-km line.

Figure 6-47 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).

A fifth problem that is worth mentioning is not a technological or protocol one like the others, but a result of new applications. Simply stated, it is that for many gigabit applications, such as multimedia, the variance in the packet arrival times is as important as the mean delay itself. A slow-but-uniform delivery rate is often preferable to a fast-but-jumpy one.

Let us now turn from the problems to ways of dealing with them. We will first make some general remarks, then look at protocol mechanisms, packet layout, and protocol software.

The basic principle that all gigabit 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.

Nowadays, there is plenty of bandwidth. 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. Usually, the best approach is to make the protocols simple and have the main CPU do the work.

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. To avoid the (long) delays inherent in the receiver sending window updates to the sender, it is better to use a rate-based protocol. 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. Of course, if bandwidth becomes so plentiful in the future that nobody cares about wasting lots of it, the design rules will become very different.

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 ease of 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, and so on do not occur.

The header and data should be separately checksummed, for two reasons. First, to make it possible to checksum the header but not the data. Second, to verify that the header is correct before copying the data into user space. It is desirable to do the data checksum at the time the data are copied to user space, but if

the header is incorrect, the copy may go to the wrong process. To avoid an incorrect copy but to allow the data checksum to be done during copying, it is essential that the two checksums be separate.

The maximum data size should be large, to permit efficient operation even in the face of long delays. Also, the larger the data block, the smaller the fraction of the total bandwidth devoted to headers. 1500 bytes is too small.

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.

Finally, a few words about the protocol software are appropriate. A key thought is concentrating on the successful case. Many older protocols tend to emphasize what to do when something goes wrong (e.g., a packet getting lost). To make the protocols run fast, the designer should aim for minimizing processing time when everything goes right. Minimizing processing time when an error occurs is secondary.

A second software issue is minimizing copying time. As we saw earlier, copying data is often the main source of overhead. Ideally, the hardware should dump each incoming packet into memory as a contiguous block of data. The software should then copy this packet to the user buffer with a single block copy. Depending on how the cache works, it may even be desirable to avoid a copy loop. In other words, to copy 1024 words, the fastest way may be to have 1024 back-to-back MOVE instructions (or 1024 load-store pairs). The copy routine is so critical it should be carefully handcrafted in assembly code, unless there is a way to trick the compiler into producing precisely the optimal code.