(i.e., sends more than kx segments/sec), the network will become congested because it will be unable to deliver segments as fast as they are coming in.

What is needed is a mechanism that limits transmissions from the sender based on the network's carrying capacity rather than on the receiver's buffering capacity. Belsnes (1975) proposed using a sliding window flow-control scheme in which the sender dynamically adjusts the window size to match the network's carrying capacity. This means that a dynamic sliding window can implement both flow control and congestion control. If the network can handle c segments/sec and the round-trip time (including transmission, propagation, queueing, processing at the receiver, and return of the acknowledgement) is r, the sender's window should be cr. With a window of this size, the sender normally operates with the pipeline full. Any small decrease in network performance will cause it to block. Since the network capacity available to any given flow varies over time, the window size should be adjusted frequently, to track changes in the carrying capacity. As we will see later, TCP uses a similar scheme.

6.2.5 Multiplexing

Multiplexing, or sharing several conversations over connections, virtual circuits, and physical links plays a role in several layers of the network architecture. In the transport layer, the need for multiplexing can arise in a number of ways. For example, if only one network address is available on a host, all transport connections on that machine have to use it. When a segment comes in, some way is needed to tell which process to give it to. This situation, called **multiplexing**, is shown in Fig. 6-17(a). In this figure, four distinct transport connections all use the same network connection (e.g., IP address) to the remote host.

Multiplexing can also be useful in the transport layer for another reason. Suppose, for example, that a host has multiple network paths that it can use. If a user needs more bandwidth or more reliability than one of the network paths can provide, a way out is to have a connection that distributes the traffic among multiple network paths on a round-robin basis, as indicated in Fig. 6-17(b). This modus operandi is called **inverse multiplexing**. With k network connections open, the effective bandwidth might be increased by a factor of k. An example of inverse multiplexing is **SCTP** (**Stream Control Transmission Protocol**), which can run a connection using multiple network interfaces. In contrast, TCP uses a single network endpoint. Inverse multiplexing is also found at the link layer, when several low-rate links are used in parallel as one high-rate link.

6.2.6 Crash Recovery

If hosts and routers are subject to crashes or connections are long-lived (e.g., large software or media downloads), recovery from these crashes becomes an issue. If the transport entity is entirely within the hosts, recovery from network

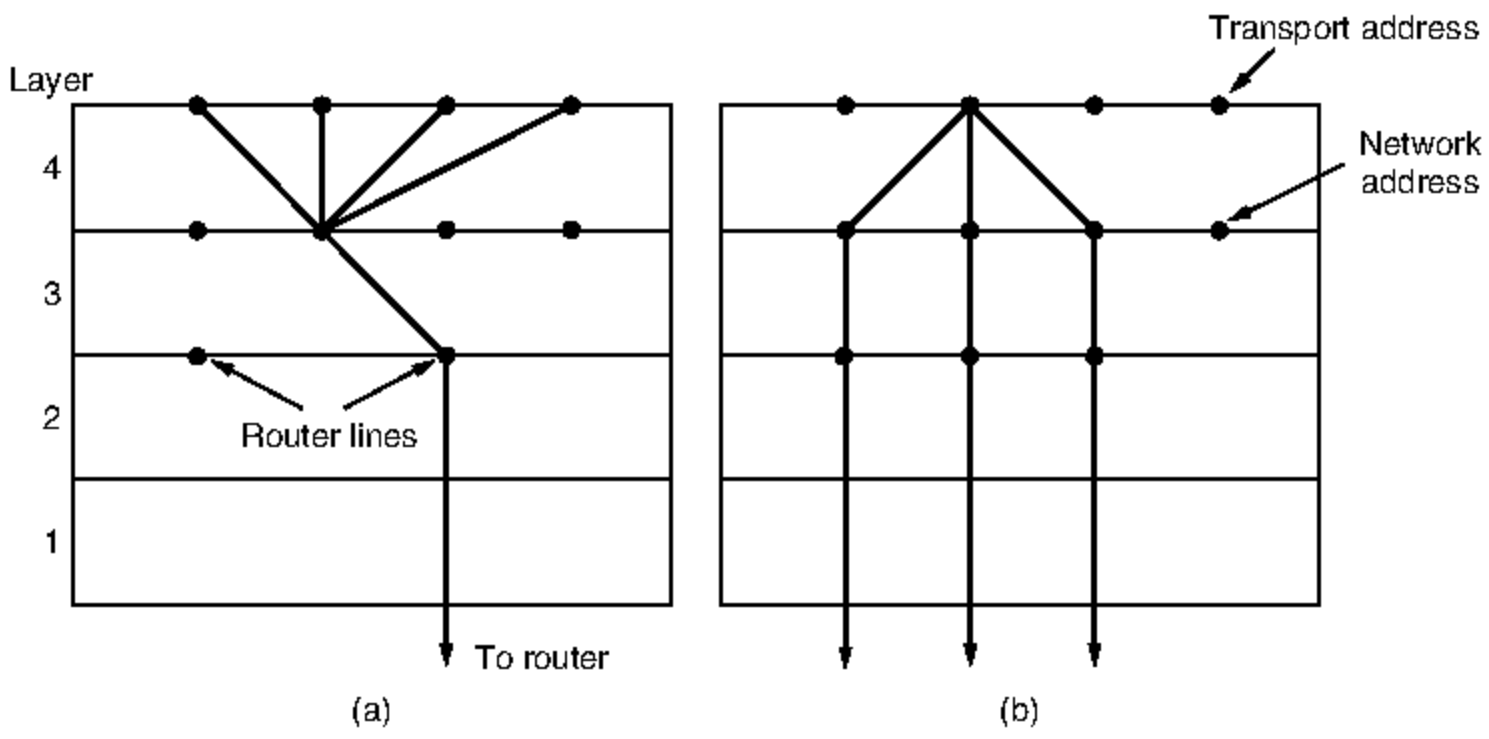

Figure 6-17. (a) Multiplexing. (b) Inverse multiplexing.

and router crashes is straightforward. The transport entities expect lost segments all the time and know how to cope with them by using retransmissions.

A more troublesome problem is how to recover from host crashes. In particular, it may be desirable for clients to be able to continue working when servers crash and quickly reboot. To illustrate the difficulty, let us assume that one host, the client, is sending a long file to another host, the file server, using a simple stop-and-wait protocol. The transport layer on the server just passes the incoming segments to the transport user, one by one. Partway through the transmission, the server crashes. When it comes back up, its tables are reinitialized, so it no longer knows precisely where it was.

In an attempt to recover its previous status, the server might send a broadcast segment to all other hosts, announcing that it has just crashed and requesting that its clients inform it of the status of all open connections. Each client can be in one of two states: one segment outstanding, *S1*, or no segments outstanding, *S0*. Based on only this state information, the client must decide whether to retransmit the most recent segment.

At first glance, it would seem obvious: the client should retransmit if and only if it has an unacknowledged segment outstanding (i.e., is in state *S1*) when it learns of the crash. However, a closer inspection reveals difficulties with this naive approach. Consider, for example, the situation in which the server's transport entity first sends an acknowledgement and then, when the acknowledgement has been sent, writes to the application process. Writing a segment onto the output stream and sending an acknowledgement are two distinct events that cannot be done simultaneously. If a crash occurs after the acknowledgement has been sent but before the write has been fully completed, the client will receive the

acknowledgement and thus be in state *S0* when the crash recovery announcement arrives. The client will therefore not retransmit, (incorrectly) thinking that the segment has arrived. This decision by the client leads to a missing segment.

At this point you may be thinking: "That problem can be solved easily. All you have to do is reprogram the transport entity to first do the write and then send the acknowledgement." Try again. Imagine that the write has been done but the crash occurs before the acknowledgement can be sent. The client will be in state *S1* and thus retransmit, leading to an undetected duplicate segment in the output stream to the server application process.

No matter how the client and server are programmed, there are always situations where the protocol fails to recover properly. The server can be programmed in one of two ways: acknowledge first or write first. The client can be programmed in one of four ways: always retransmit the last segment, never retransmit the last segment, retransmit only in state *S0*, or retransmit only in state *S1*. This gives eight combinations, but as we shall see, for each combination there is some set of events that makes the protocol fail.

Three events are possible at the server: sending an acknowledgement (A), writing to the output process (W), and crashing (C). The three events can occur in six different orderings: $AC(W)$, AWC, $C(AW)$, $C(WA)$, WAC, and $WC(A)$, where the parentheses are used to indicate that neither A nor W can follow C (i.e., once it has crashed, it has crashed). Figure 6-18 shows all eight combinations of client and server strategies and the valid event sequences for each one. Notice that for each strategy there is some sequence of events that causes the protocol to fail. For example, if the client always retransmits, the AWC event will generate an undetected duplicate, even though the other two events work properly.

Strategy used by receiving host

Strategy used by sending host	First ACK, then write			First write, then ACK		
	AC(W)	AWC	C(AW)	C(WA)	W AC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

OK = Protocol functions correctly
DUP = Protocol generates a duplicate message
LOST = Protocol loses a message

Figure 6-18. Different combinations of client and server strategies.

Making the protocol more elaborate does not help. Even if the client and server exchange several segments before the server attempts to write, so that the client knows exactly what is about to happen, the client has no way of knowing whether a crash occurred just before or just after the write. The conclusion is inescapable: under our ground rules of no simultaneous events—that is, separate events happen one after another not at the same time—host crash and recovery cannot be made transparent to higher layers.

Put in more general terms, this result can be restated as "recovery from a layer N crash can only be done by layer $N + 1$," and then only if the higher layer retains enough status information to reconstruct where it was before the problem occurred. This is consistent with the case mentioned above that the transport layer can recover from failures in the network layer, provided that each end of a connection keeps track of where it is.

This problem gets us into the issue of what a so-called end-to-end acknowledgement really means. In principle, the transport protocol is end-to-end and not chained like the lower layers. Now consider the case of a user entering requests for transactions against a remote database. Suppose that the remote transport entity is programmed to first pass segments to the next layer up and then acknowledge. Even in this case, the receipt of an acknowledgement back at the user's machine does not necessarily mean that the remote host stayed up long enough to actually update the database. A truly end-to-end acknowledgement, whose receipt means that the work has actually been done and lack thereof means that it has not, is probably impossible to achieve. This point is discussed in more detail by Saltzer et al. (1984).

6.3 CONGESTION CONTROL

If the transport entities on many machines send too many packets into the network too quickly, the network will become congested, with performance degraded as packets are delayed and lost. Controlling congestion to avoid this problem is the combined responsibility of the network and transport layers. Congestion occurs at routers, so it is detected at the network layer. However, congestion is ultimately caused by traffic sent into the network by the transport layer. The only effective way to control congestion is for the transport protocols to send packets into the network more slowly.

In Chap. 5, we studied congestion control mechanisms in the network layer. In this section, we will study the other half of the problem, congestion control mechanisms in the transport layer. After describing the goals of congestion control, we will describe how hosts can regulate the rate at which they send packets into the network. The Internet relies heavily on the transport layer for congestion control, and specific algorithms are built into TCP and other protocols.

6.3.1 Desirable Bandwidth Allocation

Before we describe how to regulate traffic, we must understand what we are trying to achieve by running a congestion control algorithm. That is, we must specify the state in which a good congestion control algorithm will operate the network. The goal is more than to simply avoid congestion. It is to find a good allocation of bandwidth to the transport entities that are using the network. A good allocation will deliver good performance because it uses all the available bandwidth but avoids congestion, it will be fair across competing transport entities, and it will quickly track changes in traffic demands. We will make each of these criteria more precise in turn.

Efficiency and Power

An efficient allocation of bandwidth across transport entities will use all of the network capacity that is available. However, it is not quite right to think that if there is a 100-Mbps link, five transport entities should get 20 Mbps each. They should usually get less than 20 Mbps for good performance. The reason is that the traffic is often bursty. Recall that in Sec. 5.3 we described the **goodput** (or rate of useful packets arriving at the receiver) as a function of the offered load. This curve and a matching curve for the delay as a function of the offered load are given in Fig. 6-19.

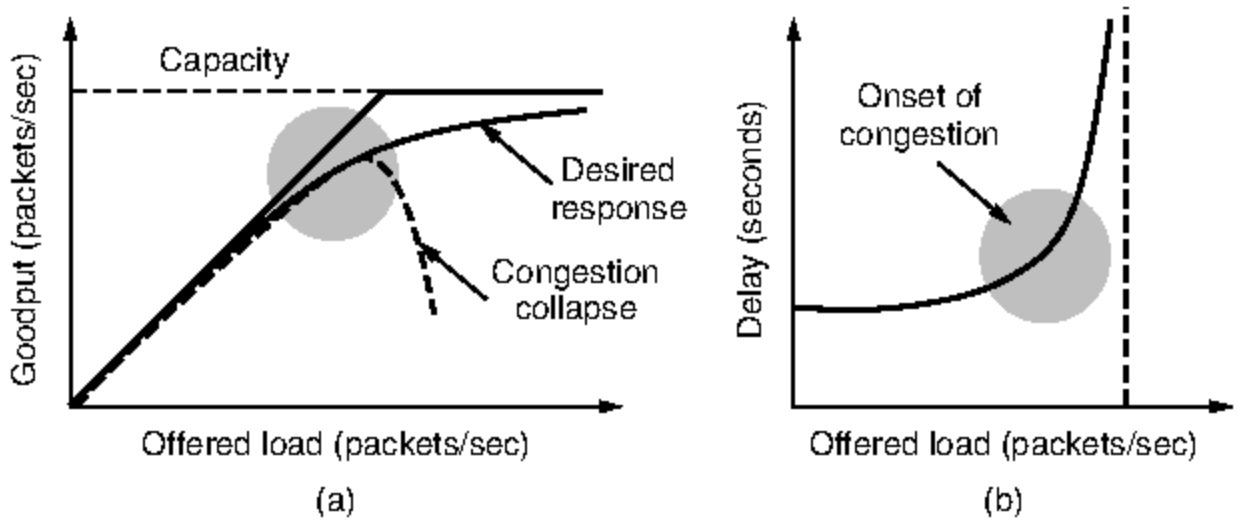

Figure 6-19. (a) Goodput and (b) delay as a function of offered load.

As the load increases in Fig. 6-19(a) goodput initially increases at the same rate, but as the load approaches the capacity, goodput rises more gradually. This falloff is because bursts of traffic can occasionally mount up and cause some losses at buffers inside the network. If the transport protocol is poorly designed and retransmits packets that have been delayed but not lost, the network can enter congestion collapse. In this state, senders are furiously sending packets, but increasingly little useful work is being accomplished.

The corresponding delay is given in Fig. 6-19(b) Initially the delay is fixed, representing the propagation delay across the network. As the load approaches the capacity, the delay rises, slowly at first and then much more rapidly. This is again because of bursts of traffic that tend to mound up at high load. The delay cannot really go to infinity, except in a model in which the routers have infinite buffers. Instead, packets will be lost after experiencing the maximum buffering delay.

For both goodput and delay, performance begins to degrade at the onset of congestion. Intuitively, we will obtain the best performance from the network if we allocate bandwidth up until the delay starts to climb rapidly. This point is below the capacity. To identify it, Kleinrock (1979) proposed the metric of **power**, where

$$power = \frac{load}{delay}$$

Power will initially rise with offered load, as delay remains small and roughly constant, but will reach a maximum and fall as delay grows rapidly. The load with the highest power represents an efficient load for the transport entity to place on the network.

Max-Min Fairness

In the preceding discussion, we did not talk about how to divide bandwidth between different transport senders. This sounds like a simple question to answer—give all the senders an equal fraction of the bandwidth—but it involves several considerations.

Perhaps the first consideration is to ask what this problem has to do with congestion control. After all, if the network gives a sender some amount of bandwidth to use, the sender should just use that much bandwidth. However, it is often the case that networks do not have a strict bandwidth reservation for each flow or connection. They may for some flows if quality of service is supported, but many connections will seek to use whatever bandwidth is available or be lumped together by the network under a common allocation. For example, IETF's differentiated services separates traffic into two classes and connections compete for bandwidth within each class. IP routers often have all connections competing for the same bandwidth. In this situation, it is the congestion control mechanism that is allocating bandwidth to the competing connections.

A second consideration is what a fair portion means for flows in a network. It is simple enough if N flows use a single link, in which case they can all have $1/N$ of the bandwidth (although efficiency will dictate that they use slightly less if the traffic is bursty). But what happens if the flows have different, but overlapping, network paths? For example, one flow may cross three links, and the other flows may cross one link. The three-link flow consumes more network resources. It might be fairer in some sense to give it less bandwidth than the one-link flows. It

should certainly be possible to support more one-link flows by reducing the bandwidth of the three-link flow. This point demonstrates an inherent tension between fairness and efficiency.

However, we will adopt a notion of fairness that does not depend on the length of the network path. Even with this simple model, giving connections an equal fraction of bandwidth is a bit complicated because different connections will take different paths through the network and these paths will themselves have different capacities. In this case, it is possible for a flow to be bottlenecked on a downstream link and take a smaller portion of an upstream link than other flows; reducing the bandwidth of the other flows would slow them down but would not help the bottlenecked flow at all.

The form of fairness that is often desired for network usage is **max-min fairness**. An allocation is max-min fair if the bandwidth given to one flow cannot be increased without decreasing the bandwidth given to another flow with an allocation that is no larger. That is, increasing the bandwidth of a flow will only make the situation worse for flows that are less well off.

Let us see an example. A max-min fair allocation is shown for a network with four flows, *A*, *B*, *C*, and *D*, in Fig. 6-20. Each of the links between routers has the same capacity, taken to be 1 unit, though in the general case the links will have different capacities. Three flows compete for the bottom-left link between routers *R4* and *R5*. Each of these flows therefore gets 1/3 of the link. The remaining flow, *A*, competes with *B* on the link from *R2* to *R3*. Since *B* has an allocation of 1/3, *A* gets the remaining 2/3 of the link. Notice that all of the other links have spare capacity. However, this capacity cannot be given to any of the flows without decreasing the capacity of another, lower flow. For example, if more of the bandwidth on the link between *R2* and *R3* is given to flow *B*, there will be less for flow *A*. This is reasonable as flow *A* already has more bandwidth. However, the capacity of flow *C* or *D* (or both) must be decreased to give more bandwidth to *B*, and these flows will have less bandwidth than *B*. Thus, the allocation is max-min fair.

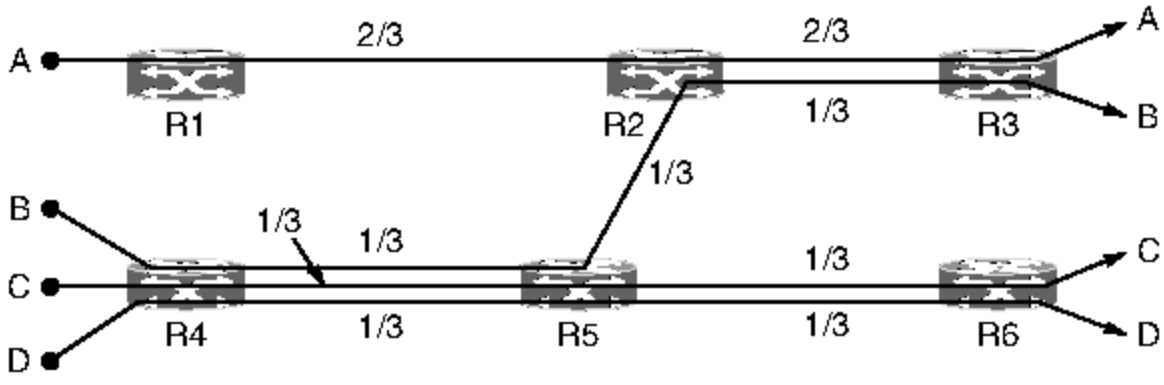

Figure 6-20. Max-min bandwidth allocation for four flows.

Max-min allocations can be computed given a global knowledge of the network. An intuitive way to think about them is to imagine that the rate for all of the

flows starts at zero and is slowly increased. When the rate reaches a bottleneck for any flow, then that flow stops increasing. The other flows all continue to increase, sharing equally in the available capacity, until they too reach their respective bottlenecks.

A third consideration is the level over which to consider fairness. A network could be fair at the level of connections, connections between a pair of hosts, or all connections per host. We examined this issue when we were discussing WFQ (Weighted Fair Queueing) in Sec. 5.4 and concluded that each of these definitions has its problems. For example, defining fairness per host means that a busy server will fare no better than a mobile phone, while defining fairness per connection encourages hosts to open more connections. Given that there is no clear answer, fairness is often considered per connection, but precise fairness is usually not a concern. It is more important in practice that no connection be starved of bandwidth than that all connections get precisely the same amount of bandwidth. In fact, with TCP it is possible to open multiple connections and compete for bandwidth more aggressively. This tactic is used by bandwidth-hungry applications such as BitTorrent for peer-to-peer file sharing.

Convergence

A final criterion is that the congestion control algorithm converge quickly to a fair and efficient allocation of bandwidth. The discussion of the desirable operating point above assumes a static network environment. However, connections are always coming and going in a network, and the bandwidth needed by a given connection will vary over time too, for example, as a user browses Web pages and occasionally downloads large videos.

Because of the variation in demand, the ideal operating point for the network varies over time. A good congestion control algorithm should rapidly converge to the ideal operating point, and it should track that point as it changes over time. If the convergence is too slow, the algorithm will never be close to the changing operating point. If the algorithm is not stable, it may fail to converge to the right point in some cases, or even oscillate around the right point.

An example of a bandwidth allocation that changes over time and converges quickly is shown in Fig. 6-21. Initially, flow 1 has all of the bandwidth. One second later, flow 2 starts. It needs bandwidth as well. The allocation quickly changes to give each of these flows half the bandwidth. At 4 seconds, a third flow joins. However, this flow uses only 20% of the bandwidth, which is less than its fair share (which is a third). Flows 1 and 2 quickly adjust, dividing the available bandwidth to each have 40% of the bandwidth. At 9 seconds, the second flow leaves, and the third flow remains unchanged. The first flow quickly captures 80% of the bandwidth. At all times, the total allocated bandwidth is approximately 100%, so that the network is fully used, and competing flows get equal treatment (but do not have to use more bandwidth than they need).

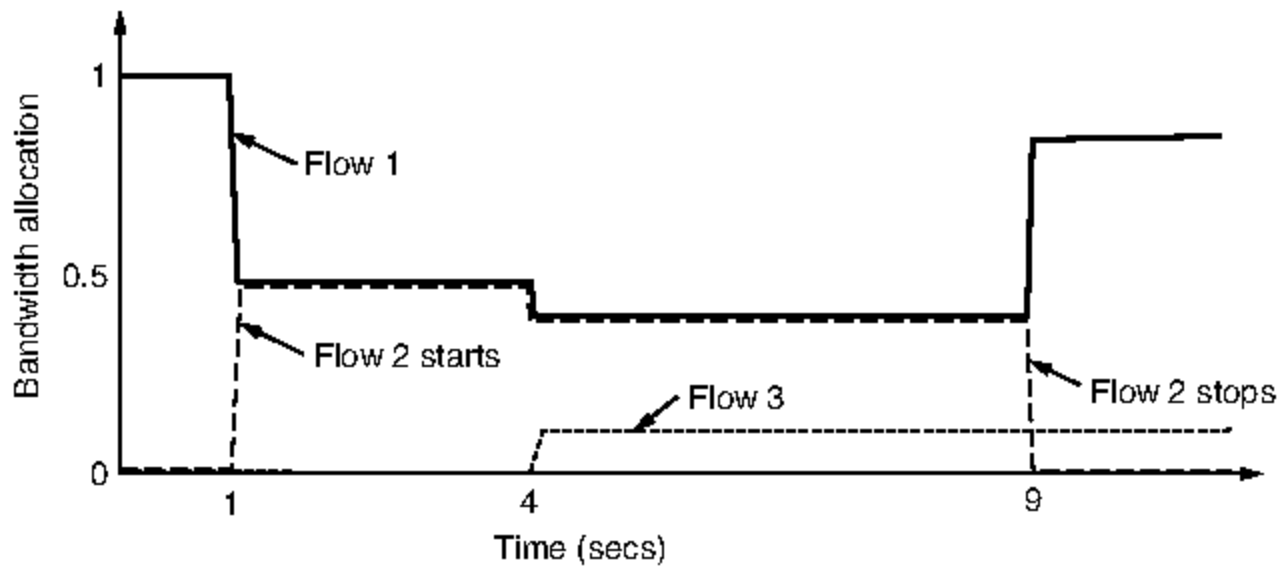

Figure 6-21. Changing bandwidth allocation over time.

6.3.2 Regulating the Sending Rate

Now it is time for the main course. How do we regulate the sending rates to obtain a desirable bandwidth allocation? The sending rate may be limited by two factors. The first is flow control, in the case that there is insufficient buffering at the receiver. The second is congestion, in the case that there is insufficient capacity in the network. In Fig. 6-22, we see this problem illustrated hydraulically. In Fig. 6-22(a), we see a thick pipe leading to a small-capacity receiver. This is a flow-control limited situation. As long as the sender does not send more water than the bucket can contain, no water will be lost. In Fig. 6-22(b), the limiting factor is not the bucket capacity, but the internal carrying capacity of the network. If too much water comes in too fast, it will back up and some will be lost (in this case, by overflowing the funnel).

These cases may appear similar to the sender, as transmitting too fast causes packets to be lost. However, they have different causes and call for different solutions. We have already talked about a flow-control solution with a variable-sized window. Now we will consider a congestion control solution. Since either of these problems can occur, the transport protocol will in general need to run both solutions and slow down if either problem occurs.

The way that a transport protocol should regulate the sending rate depends on the form of the feedback returned by the network. Different network layers may return different kinds of feedback. The feedback may be explicit or implicit, and it may be precise or imprecise.

An example of an explicit, precise design is when routers tell the sources the rate at which they may send. Designs in the literature such as XCP (eXplicit Congestion Protocol) operate in this manner (Katabi et al., 2002). An explicit, imprecise design is the use of ECN (Explicit Congestion Notification) with TCP. In this design, routers set bits on packets that experience congestion to warn the senders to slow down, but they do not tell them how much to slow down.

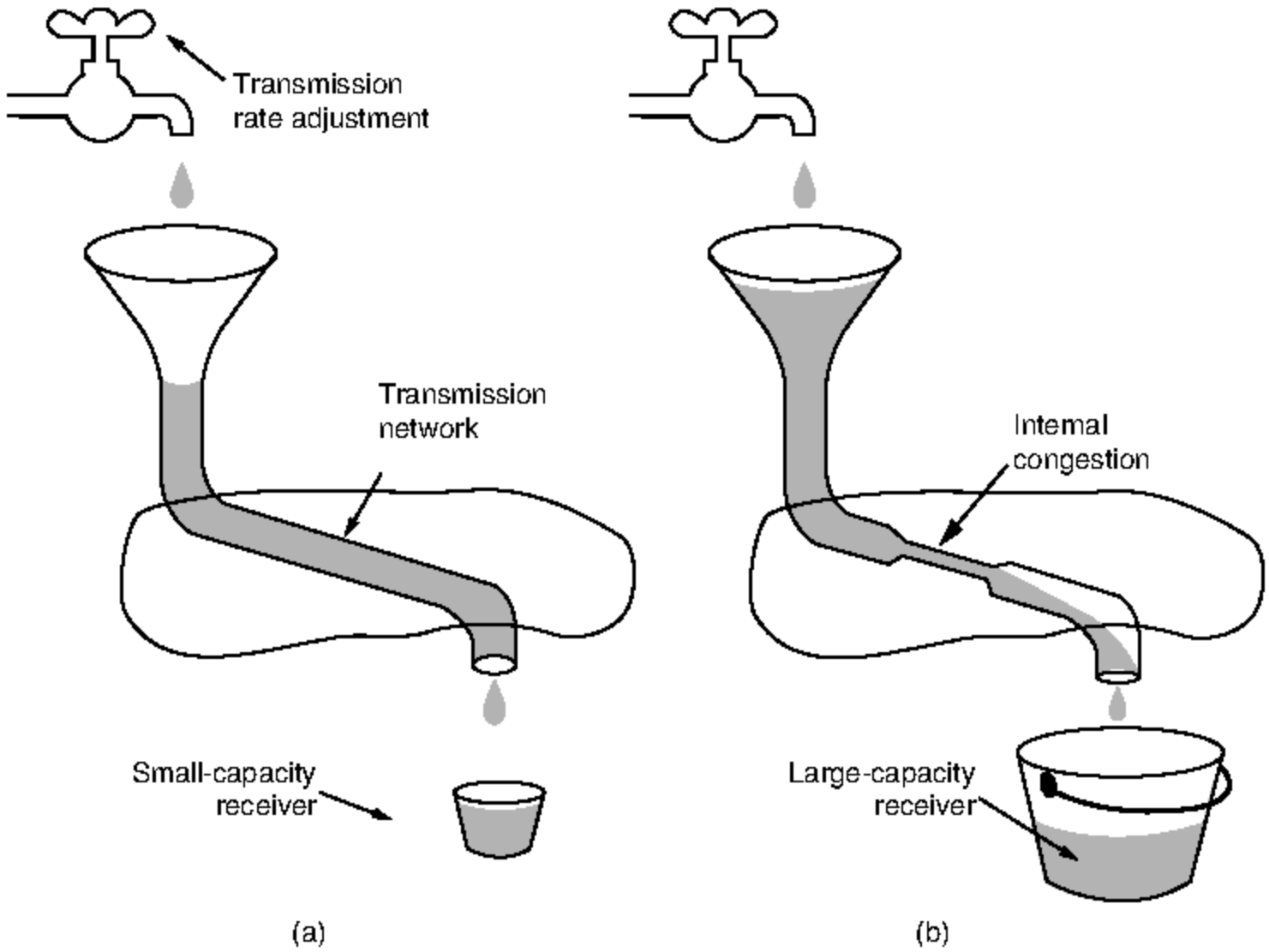

Figure 6-22. (a) A fast network feeding a low-capacity receiver. (b) A slow network feeding a high-capacity receiver.

In other designs, there is no explicit signal. FAST TCP measures the round-trip delay and uses that metric as a signal to avoid congestion (Wei et al., 2006). Finally, in the form of congestion control most prevalent in the Internet today, TCP with drop-tail or RED routers, packet loss is inferred and used to signal that the network has become congested. There are many variants of this form of TCP, including CUBIC TCP, which is used in Linux (Ha et al., 2008). Combinations are also possible. For example, Windows includes Compound TCP that uses both packet loss and delay as feedback signals (Tan et al., 2006). These designs are summarized in Fig. 6-23.

If an explicit and precise signal is given, the transport entity can use that signal to adjust its rate to the new operating point. For example, if XCP tells senders the rate to use, the senders may simply use that rate. In the other cases, however, some guesswork is involved. In the absence of a congestion signal, the senders should decrease their rates. When a congestion signal is given, the senders should decrease their rates. The way in which the rates are increased or decreased is given by a **control law**. These laws have a major effect on performance.

Protocol	Signal	Explicit?	Precise?
XCP	Rate to use	Yes	Yes
TCP with ECN	Congestion warning	Yes	No
FAST TCP	End-to-end delay	No	Yes
Compound TCP	Packet loss & end-to-end delay	No	Yes
CUBIC TCP	Packet loss	No	No
TCP	Packet loss	No	No

Figure 6-23. Signals of some congestion control protocols.

Chiu and Jain (1989) studied the case of binary congestion feedback and concluded that **AIMD** (**Additive Increase Multiplicative Decrease**) is the appropriate control law to arrive at the efficient and fair operating point. To argue this case, they constructed a graphical argument for the simple case of two connections competing for the bandwidth of a single link. The graph in Fig. 6-24 shows the bandwidth allocated to user 1 on the x-axis and to user 2 on the y-axis. When the allocation is fair, both users will receive the same amount of bandwidth. This is shown by the dotted fairness line. When the allocations sum to 100%, the capacity of the link, the allocation is efficient. This is shown by the dotted efficiency line. A congestion signal is given by the network to both users when the sum of their allocations crosses this line. The intersection of these lines is the desired operating point, when both users have the same bandwidth and all of the network bandwidth is used.

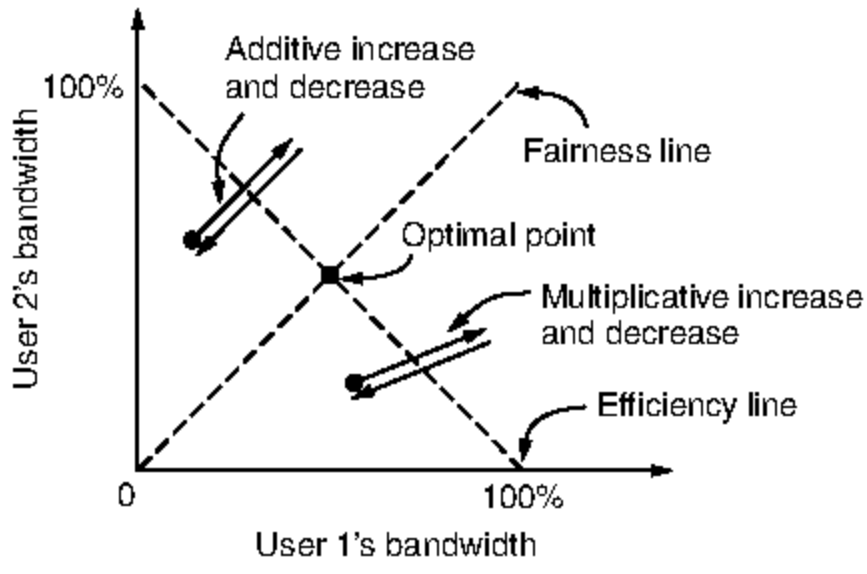

Figure 6-24. Additive and multiplicative bandwidth adjustments.

Consider what happens from some starting allocation if both user 1 and user 2 additively increase their respective bandwidths over time. For example, the users may each increase their sending rate by 1 Mbps every second. Eventually, the

operating point crosses the efficiency line and both users receive a congestion signal from the network. At this stage, they must reduce their allocations. However, an additive decrease would simply cause them to oscillate along an additive line. This situation is shown in Fig. 6-24. The behavior will keep the operating point close to efficient, but it will not necessarily be fair.

Similarly, consider the case when both users multiplicatively increase their bandwidth over time until they receive a congestion signal. For example, the users may increase their sending rate by 10% every second. If they then multiplicatively decrease their sending rates, the operating point of the users will simply oscillate along a multiplicative line. This behavior is also shown in Fig. 6-24. The multiplicative line has a different slope than the additive line. (It points to the origin, while the additive line has an angle of 45 degrees.) But it is otherwise no better. In neither case will the users converge to the optimal sending rates that are both fair and efficient.

Now consider the case that the users additively increase their bandwidth allocations and then multiplicatively decrease them when congestion is signaled. This behavior is the AIMD control law, and it is shown in Fig. 6-25. It can be seen that the path traced by this behavior does converge to the optimal point that is both fair and efficient. This convergence happens no matter what the starting point, making AIMD broadly useful. By the same argument, the only other combination, multiplicative increase and additive decrease, would diverge from the optimal point.

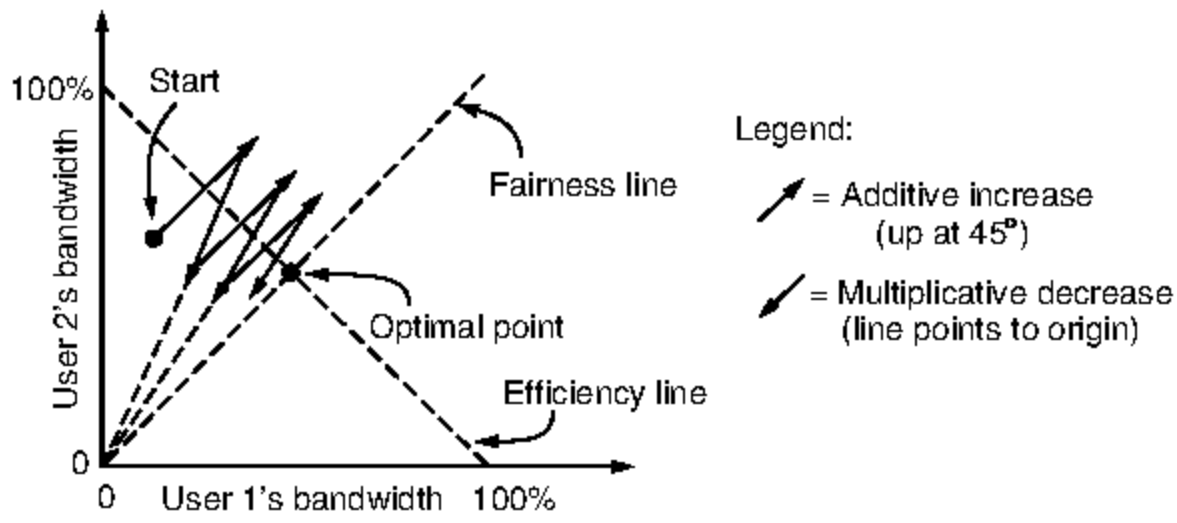

Figure 6-25. Additive Increase Multiplicative Decrease (AIMD) control law.

AIMD is the control law that is used by TCP, based on this argument and another stability argument (that it is easy to drive the network into congestion and difficult to recover, so the increase policy should be gentle and the decrease policy aggressive). It is not quite fair, since TCP connections adjust their window size by a given amount every round-trip time. Different connections will have different round-trip times. This leads to a bias in which connections to closer hosts receive more bandwidth than connections to distant hosts, all else being equal.

In Sec. 6.5, we will describe in detail how TCP implements an AIMD control law to adjust the sending rate and provide congestion control. This task is more difficult than it sounds because rates are measured over some interval and traffic is bursty. Instead of adjusting the rate directly, a strategy that is often used in practice is to adjust the size of a sliding window. TCP uses this strategy. If the window size is W and the round-trip time is RTT, the equivalent rate is W/RTT. This strategy is easy to combine with flow control, which already uses a window, and has the advantage that the sender paces packets using acknowledgements and hence slows down in one RTT if it stops receiving reports that packets are leaving the network.

As a final issue, there may be many different transport protocols that send traffic into the network. What will happen if the different protocols compete with different control laws to avoid congestion? Unequal bandwidth allocations, that is what. Since TCP is the dominant form of congestion control in the Internet, there is significant community pressure for new transport protocols to be designed so that they compete fairly with it. The early streaming media protocols caused problems by excessively reducing TCP throughput because they did not compete fairly. This led to the notion of **TCP-friendly** congestion control in which TCP and non-TCP transport protocols can be freely mixed with no ill effects (Floyd et al., 2000).

6.3.3 Wireless Issues

Transport protocols such as TCP that implement congestion control should be independent of the underlying network and link layer technologies. That is a good theory, but in practice there are issues with wireless networks. The main issue is that packet loss is often used as a congestion signal, including by TCP as we have just discussed. Wireless networks lose packets all the time due to transmission errors.

With the AIMD control law, high throughput requires very small levels of packet loss. Analyses by Padhye et al. (1998) show that the throughput goes up as the inverse square-root of the packet loss rate. What this means in practice is that the loss rate for fast TCP connections is very small; 1% is a moderate loss rate, and by the time the loss rate reaches 10% the connection has effectively stopped working. However, for wireless networks such as 802.11 LANs, frame loss rates of at least 10% are common. This difference means that, absent protective measures, congestion control schemes that use packet loss as a signal will unnecessarily throttle connections that run over wireless links to very low rates.

To function well, the only packet losses that the congestion control algorithm should observe are losses due to insufficient bandwidth, not losses due to transmission errors. One solution to this problem is to mask the wireless losses by using retransmissions over the wireless link. For example, 802.11 uses a stop-and-wait protocol to deliver each frame, retrying transmissions multiple times if

need be before reporting a packet loss to the higher layer. In the normal case, each packet is delivered despite transient transmission errors that are not visible to the higher layers.

Fig. 6-26 shows a path with a wired and wireless link for which the masking strategy is used. There are two aspects to note. First, the sender does not necessarily know that the path includes a wireless link, since all it sees is the wired link to which it is attached. Internet paths are heterogeneous and there is no general method for the sender to tell what kind of links comprise the path. This complicates the congestion control problem, as there is no easy way to use one protocol for wireless links and another protocol for wired links.

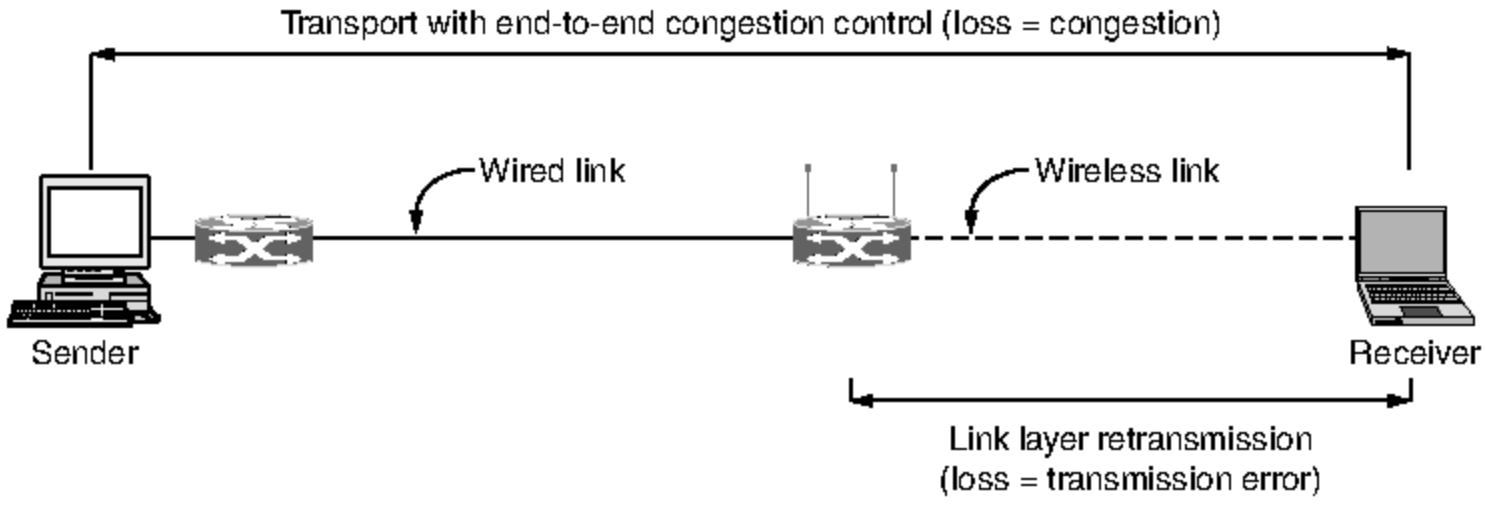

Figure 6-26. Congestion control over a path with a wireless link.

The second aspect is a puzzle. The figure shows two mechanisms that are driven by loss: link layer frame retransmissions, and transport layer congestion control. The puzzle is how these two mechanisms can co-exist without getting confused. After all, a loss should cause only one mechanism to take action because it is either a transmission error or a congestion signal. It cannot be both. If both mechanisms take action (by retransmitting the frame and slowing down the sending rate) then we are back to the original problem of transports that run far too slowly over wireless links. Consider this puzzle for a moment and see if you can solve it.

The solution is that the two mechanisms act at different timescales. Link layer retransmissions happen on the order of microseconds to milliseconds for wireless links such as 802.11. Loss timers in transport protocols fire on the order of milliseconds to seconds. The difference is three orders of magnitude. This allows wireless links to detect frame losses and retransmit frames to repair transmission errors long before packet loss is inferred by the transport entity.

The masking strategy is sufficient to let most transport protocols run well across most wireless links. However, it is not always a fitting solution. Some wireless links have long round-trip times, such as satellites. For these links other techniques must be used to mask loss, such as FEC (Forward Error Correction), or the transport protocol must use a non-loss signal for congestion control.

A second issue with congestion control over wireless links is variable capacity. That is, the capacity of a wireless link changes over time, sometimes abruptly, as nodes move and the signal-to-noise ratio varies with the changing channel conditions. This is unlike wired links whose capacity is fixed. The transport protocol must adapt to the changing capacity of wireless links, otherwise it will either congest the network or fail to use the available capacity.

One possible solution to this problem is simply not to worry about it. This strategy is feasible because congestion control algorithms must already handle the case of new users entering the network or existing users changing their sending rates. Even though the capacity of wired links is fixed, the changing behavior of other users presents itself as variability in the bandwidth that is available to a given user. Thus it is possible to simply run TCP over a path with an 802.11 wireless link and obtain reasonable performance.

However, when there is much wireless variability, transport protocols designed for wired links may have trouble keeping up and deliver poor performance. The solution in this case is a transport protocol that is designed for wireless links. A particularly challenging setting is a wireless mesh network in which multiple, interfering wireless links must be crossed, routes change due to mobility, and there is lots of loss. Research in this area is ongoing. See Li et al. (2009) for an example of wireless transport protocol design.

6.4 THE INTERNET TRANSPORT PROTOCOLS: UDP

The Internet has two main protocols in the transport layer, a connectionless protocol and a connection-oriented one. The protocols complement each other. The connectionless protocol is UDP. It does almost nothing beyond sending packets between applications, letting applications build their own protocols on top as needed. The connection-oriented protocol is TCP. It does almost everything. It makes connections and adds reliability with retransmissions, along with flow control and congestion control, all on behalf of the applications that use it.

In the following sections, we will study UDP and TCP. We will start with UDP because it is simplest. We will also look at two uses of UDP. Since UDP is a transport layer protocol that typically runs in the operating system and protocols that use UDP typically run in user space, these uses might be considered applications. However, the techniques they use are useful for many applications and are better considered to belong to a transport service, so we will cover them here.

6.4.1 Introduction to UDP

The Internet protocol suite supports a connectionless transport protocol called **UDP** (**User Datagram Protocol**). UDP provides a way for applications to send encapsulated IP datagrams without having to establish a connection. UDP is described in RFC 768.

UDP transmits **segments** consisting of an 8-byte header followed by the payload. The header is shown in Fig. 6-27. The two **ports** serve to identify the endpoints within the source and destination machines. When a UDP packet arrives, its payload is handed to the process attached to the destination port. This attachment occurs when the BIND primitive or something similar is used, as we saw in Fig. 6-6 for TCP (the binding process is the same for UDP). Think of ports as mailboxes that applications can rent to receive packets. We will have more to say about them when we describe TCP, which also uses ports. In fact, the main value of UDP over just using raw IP is the addition of the source and destination ports. Without the port fields, the transport layer would not know what to do with each incoming packet. With them, it delivers the embedded segment to the correct application.

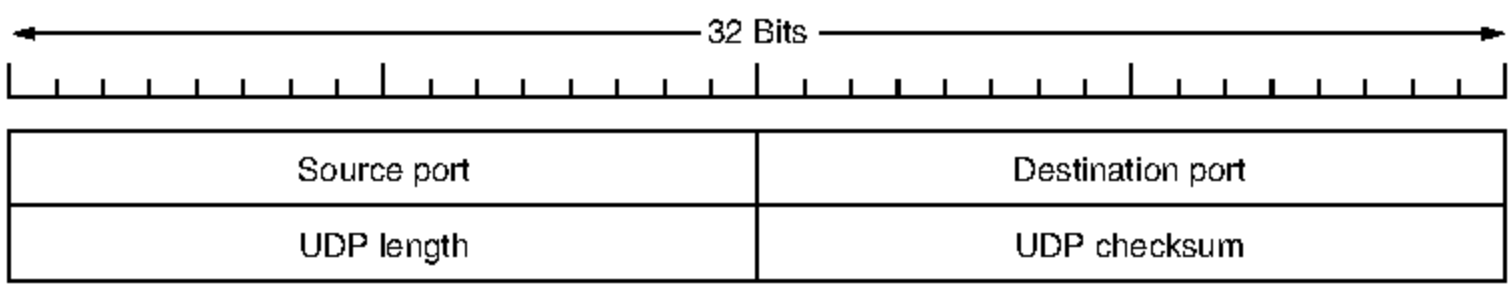

Figure 6-27. The UDP header.

The source port is primarily needed when a reply must be sent back to the source. By copying the *Source port* field from the incoming segment into the *Destination port* field of the outgoing segment, the process sending the reply can specify which process on the sending machine is to get it.

The *UDP length* field includes the 8-byte header and the data. The minimum length is 8 bytes, to cover the header. The maximum length is 65,515 bytes, which is lower than the largest number that will fit in 16 bits because of the size limit on IP packets.

An optional *Checksum* is also provided for extra reliability. It checksums the header, the data, and a conceptual IP pseudoheader. When performing this computation, the *Checksum* field is set to zero and the data field is padded out with an additional zero byte if its length is an odd number. The checksum algorithm is simply to add up all the 16-bit words in one's complement and to take the one's complement of the sum. As a consequence, when the receiver performs the calculation on the entire segment, including the *Checksum* field, the result should be 0. If the checksum is not computed, it is stored as a 0, since by a happy coincidence of one's complement arithmetic a true computed 0 is stored as all 1s. However, turning it off is foolish unless the quality of the data does not matter (e.g., for digitized speech).

The pseudoheader for the case of IPv4 is shown in Fig. 6-28. It contains the 32-bit IPv4 addresses of the source and destination machines, the protocol number for UDP (17), and the byte count for the UDP segment (including the header). It

is different but analogous for IPv6. Including the pseudoheader in the UDP checksum computation helps detect misdelivered packets, but including it also violates the protocol hierarchy since the IP addresses in it belong to the IP layer, not to the UDP layer. TCP uses the same pseudoheader for its checksum.

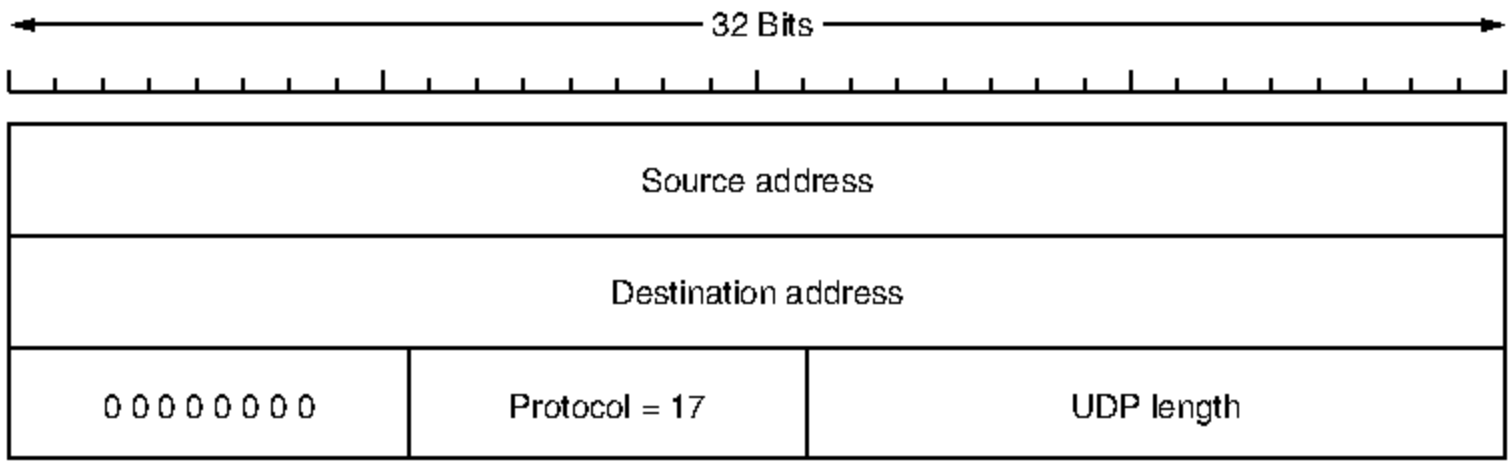

Figure 6-28. The IPv4 pseudoheader included in the UDP checksum.

It is probably worth mentioning explicitly some of the things that UDP does *not* do. It does not do flow control, congestion control, or retransmission upon receipt of a bad segment. All of that is up to the user processes. What it does do is provide an interface to the IP protocol with the added feature of demultiplexing multiple processes using the ports and optional end-to-end error detection. That is all it does.

For applications that need to have precise control over the packet flow, error control, or timing, UDP provides just what the doctor ordered. One area where it is especially useful is in client-server situations. Often, the client sends a short request to the server and expects a short reply back. If either the request or the reply is lost, the client can just time out and try again. Not only is the code simple, but fewer messages are required (one in each direction) than with a protocol requiring an initial setup like TCP.

An application that uses UDP this way is DNS (Domain Name System), which we will study in Chap. 7. In brief, a program that needs to look up the IP address of some host name, for example, *www.cs.berkeley.edu*, can send a UDP packet containing the host name to a DNS server. The server replies with a UDP packet containing the host's IP address. No setup is needed in advance and no release is needed afterward. Just two messages go over the network.

6.4.2 Remote Procedure Call

In a certain sense, sending a message to a remote host and getting a reply back is a lot like making a function call in a programming language. In both cases, you start with one or more parameters and you get back a result. This observation has led people to try to arrange request-reply interactions on networks to be cast in the

form of procedure calls. Such an arrangement makes network applications much easier to program and more familiar to deal with. For example, just imagine a procedure named *get_IP_address*(*host_name*) that works by sending a UDP packet to a DNS server and waiting for the reply, timing out and trying again if one is not forthcoming quickly enough. In this way, all the details of networking can be hidden from the programmer.

The key work in this area was done by Birrell and Nelson (1984). In a nutshell, what Birrell and Nelson suggested was allowing programs to call procedures located on remote hosts. When a process on machine 1 calls a procedure on machine 2, the calling process on 1 is suspended and execution of the called procedure takes place on 2. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing is visible to the application programmer. This technique is known as **RPC** (**Remote Procedure Call**) and has become the basis for many networking applications. Traditionally, the calling procedure is known as the client and the called procedure is known as the server, and we will use those names here too.

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In the simplest form, to call a remote procedure, the client program must be bound with a small library procedure, called the **client stub**, that represents the server procedure in the client's address space. Similarly, the server is bound with a procedure called the **server stub**. These procedures hide the fact that the procedure call from the client to the server is not local.

The actual steps in making an RPC are shown in Fig. 6-29. Step 1 is the client calling the client stub. This call is a local procedure call, with the parameters pushed onto the stack in the normal way. Step 2 is the client stub packing the parameters into a message and making a system call to send the message. Packing the parameters is called **marshaling**. Step 3 is the operating system sending the message from the client machine to the server machine. Step 4 is the operating system passing the incoming packet to the server stub. Finally, step 5 is the server stub calling the server procedure with the unmarshaled parameters. The reply traces the same path in the other direction.

The key item to note here is that the client procedure, written by the user, just makes a normal (i.e., local) procedure call to the client stub, which has the same name as the server procedure. Since the client procedure and client stub are in the same address space, the parameters are passed in the usual way. Similarly, the server procedure is called by a procedure in its address space with the parameters it expects. To the server procedure, nothing is unusual. In this way, instead of I/O being done on sockets, network communication is done by faking a normal procedure call.

Despite the conceptual elegance of RPC, there are a few snakes hiding under the grass. A big one is the use of pointer parameters. Normally, passing a pointer to a procedure is not a problem. The called procedure can use the pointer in the same way the caller can because both procedures live in the same virtual address

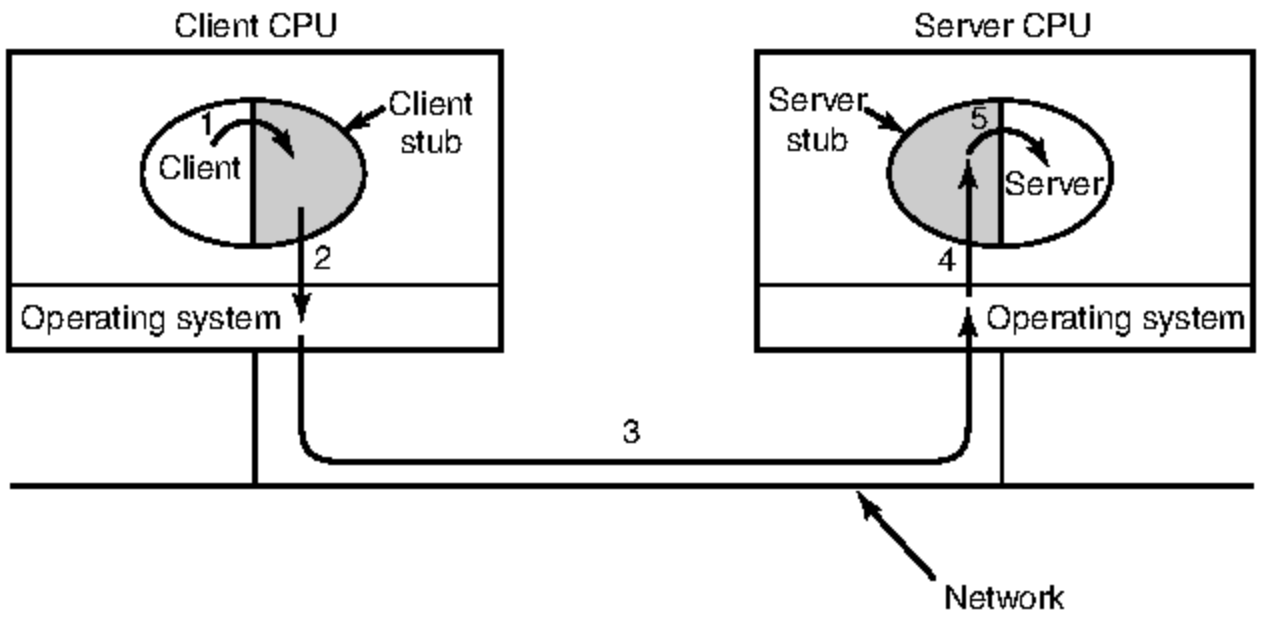

Figure 6-29. Steps in making a remote procedure call. The stubs are shaded.

space. With RPC, passing pointers is impossible because the client and server are in different address spaces.

In some cases, tricks can be used to make it possible to pass pointers. Suppose that the first parameter is a pointer to an integer, *k*. The client stub can marshal *k* and send it along to the server. The server stub then creates a pointer to *k* and passes it to the server procedure, just as it expects. When the server procedure returns control to the server stub, the latter sends *k* back to the client, where the new *k* is copied over the old one, just in case the server changed it. In effect, the standard calling sequence of call-by-reference has been replaced by call-by-copy-restore. Unfortunately, this trick does not always work, for example, if the pointer points to a graph or other complex data structure. For this reason, some restrictions must be placed on parameters to procedures called remotely, as we shall see.

A second problem is that in weakly typed languages, like C, it is perfectly legal to write a procedure that computes the inner product of two vectors (arrays), without specifying how large either one is. Each could be terminated by a special value known only to the calling and called procedures. Under these circumstances, it is essentially impossible for the client stub to marshal the parameters: it has no way of determining how large they are.

A third problem is that it is not always possible to deduce the types of the parameters, not even from a formal specification or the code itself. An example is *printf*, which may have any number of parameters (at least one), and the parameters can be an arbitrary mixture of integers, shorts, longs, characters, strings, floating-point numbers of various lengths, and other types. Trying to call *printf* as a remote procedure would be practically impossible because C is so permissive. However, a rule saying that RPC can be used provided that you do not program in C (or C++) would not be popular with a lot of programmers.

A fourth problem relates to the use of global variables. Normally, the calling and called procedure can communicate by using global variables, in addition to communicating via parameters. But if the called procedure is moved to a remote machine, the code will fail because the global variables are no longer shared.

These problems are not meant to suggest that RPC is hopeless. In fact, it is widely used, but some restrictions are needed to make it work well in practice.

In terms of transport layer protocols, UDP is a good base on which to implement RPC. Both requests and replies may be sent as a single UDP packet in the simplest case and the operation can be fast. However, an implementation must include other machinery as well. Because the request or the reply may be lost, the client must keep a timer to retransmit the request. Note that a reply serves as an implicit acknowledgement for a request, so the request need not be separately acknowledged. Sometimes the parameters or results may be larger than the maximum UDP packet size, in which case some protocol is needed to deliver large messages. If multiple requests and replies can overlap (as in the case of concurrent programming), an identifier is needed to match the request with the reply.

A higher-level concern is that the operation may not be idempotent (i.e., safe to repeat). The simple case is idempotent operations such as DNS requests and replies. The client can safely retransmit these requests again and again if no replies are forthcoming. It does not matter whether the server never received the request, or it was the reply that was lost. The answer, when it finally arrives, will be the same (assuming the DNS database is not updated in the meantime). However, not all operations are idempotent, for example, because they have important side-effects such as incrementing a counter. RPC for these operations requires stronger semantics so that when the programmer calls a procedure it is not executed multiple times. In this case, it may be necessary to set up a TCP connection and send the request over it rather than using UDP.

6.4.3 Real-Time Transport Protocols

Client-server RPC is one area in which UDP is widely used. Another one is for real-time multimedia applications. In particular, as Internet radio, Internet telephony, music-on-demand, videoconferencing, video-on-demand, and other multimedia applications became more commonplace, people have discovered that each application was reinventing more or less the same real-time transport protocol. It gradually became clear that having a generic real-time transport protocol for multiple applications would be a good idea.

Thus was **RTP** (**Real-time Transport Protocol**) born. It is described in RFC 3550 and is now in widespread use for multimedia applications. We will describe two aspects of real-time transport. The first is the RTP protocol for transporting audio and video data in packets. The second is the processing that takes place, mostly at the receiver, to play out the audio and video at the right time. These functions fit into the protocol stack as shown in Fig. 6-30.

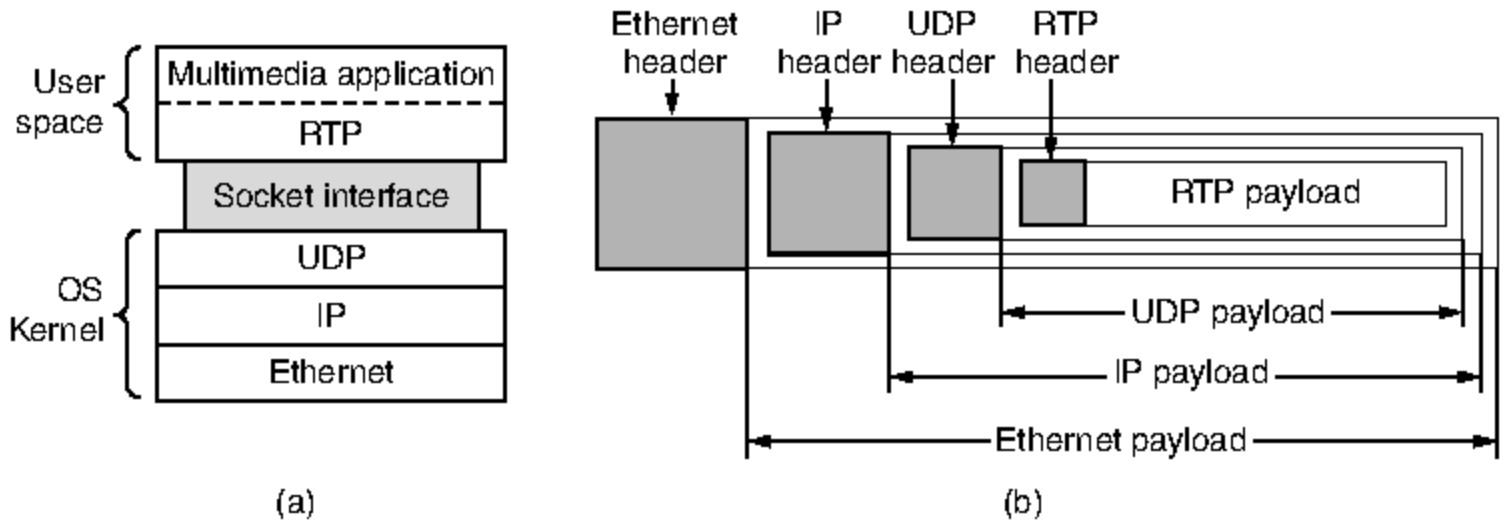

Figure 6-30. (a) The position of RTP in the protocol stack. (b) Packet nesting.

RTP normally runs in user space over UDP (in the operating system). It operates as follows. The multimedia application consists of multiple audio, video, text, and possibly other streams. These are fed into the RTP library, which is in user space along with the application. This library multiplexes the streams and encodes them in RTP packets, which it stuffs into a socket. On the operating system side of the socket, UDP packets are generated to wrap the RTP packets and handed to IP for transmission over a link such as Ethernet. The reverse process happens at the receiver. The multimedia application eventually receives multimedia data from the RTP library. It is responsible for playing out the media. The protocol stack for this situation is shown in Fig. 6-30(a). The packet nesting is shown in Fig. 6-30(b).

As a consequence of this design, it is a little hard to say which layer RTP is in. Since it runs in user space and is linked to the application program, it certainly looks like an application protocol. On the other hand, it is a generic, application-independent protocol that just provides transport facilities, so it also looks like a transport protocol. Probably the best description is that it is a transport protocol that just happens to be implemented in the application layer, which is why we are covering it in this chapter.

RTP—The Real-time Transport Protocol

The basic function of RTP is to multiplex several real-time data streams onto a single stream of UDP packets. The UDP stream can be sent to a single destination (unicasting) or to multiple destinations (multicasting). Because RTP just uses normal UDP, its packets are not treated specially by the routers unless some normal IP quality-of-service features are enabled. In particular, there are no special guarantees about delivery, and packets may be lost, delayed, corrupted, etc.

The RTP format contains several features to help receivers work with multimedia information. Each packet sent in an RTP stream is given a number one

higher than its predecessor. This numbering allows the destination to determine if any packets are missing. If a packet is missing, the best action for the destination to take is up to the application. It may be to skip a video frame if the packets are carrying video data, or to approximate the missing value by interpolation if the packets are carrying audio data. Retransmission is not a practical option since the retransmitted packet would probably arrive too late to be useful. As a consequence, RTP has no acknowledgements, and no mechanism to request retransmissions.

Each RTP payload may contain multiple samples, and they may be coded any way that the application wants. To allow for interworking, RTP defines several profiles (e.g., a single audio stream), and for each profile, multiple encoding formats may be allowed. For example, a single audio stream may be encoded as 8-bit PCM samples at 8 kHz using delta encoding, predictive encoding, GSM encoding, MP3 encoding, and so on. RTP provides a header field in which the source can specify the encoding but is otherwise not involved in how encoding is done.

Another facility many real-time applications need is timestamping. The idea here is to allow the source to associate a timestamp with the first sample in each packet. The timestamps are relative to the start of the stream, so only the differences between timestamps are significant. The absolute values have no meaning. As we will describe shortly, this mechanism allows the destination to do a small amount of buffering and play each sample the right number of milliseconds after the start of the stream, independently of when the packet containing the sample arrived.

Not only does timestamping reduce the effects of variation in network delay, but it also allows multiple streams to be synchronized with each other. For example, a digital television program might have a video stream and two audio streams. The two audio streams could be for stereo broadcasts or for handling films with an original language soundtrack and a soundtrack dubbed into the local language, giving the viewer a choice. Each stream comes from a different physical device, but if they are timestamped from a single counter, they can be played back synchronously, even if the streams are transmitted and/or received somewhat erratically.

The RTP header is illustrated in Fig. 6-31. It consists of three 32-bit words and potentially some extensions. The first word contains the *Version* field, which is already at 2. Let us hope this version is very close to the ultimate version since there is only one code point left (although 3 could be defined as meaning that the real version was in an extension word).

The *P* bit indicates that the packet has been padded to a multiple of 4 bytes. The last padding byte tells how many bytes were added. The *X* bit indicates that an extension header is present. The format and meaning of the extension header are not defined. The only thing that is defined is that the first word of the extension gives the length. This is an escape hatch for any unforeseen requirements.

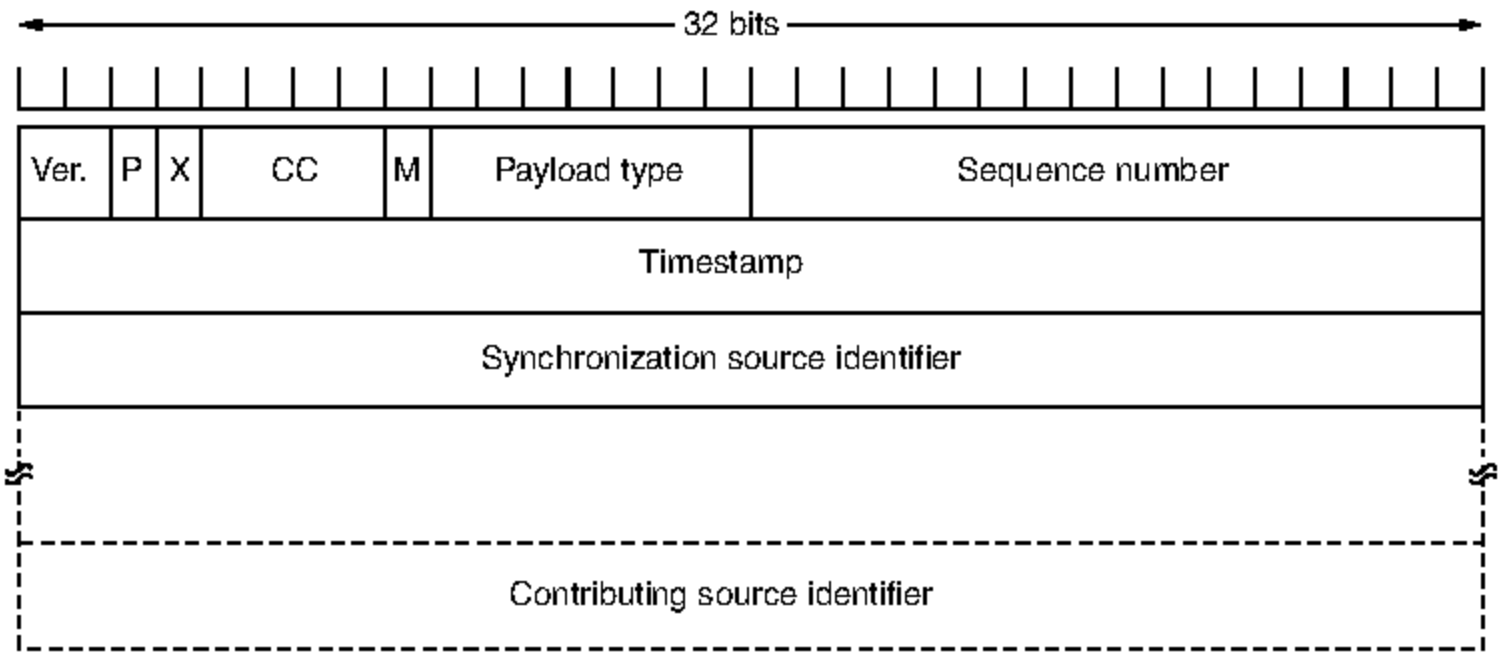

Figure 6-31. The RTP header.

The *CC* field tells how many contributing sources are present, from 0 to 15 (see below). The *M* bit is an application-specific marker bit. It can be used to mark the start of a video frame, the start of a word in an audio channel, or something else that the application understands. The *Payload type* field tells which encoding algorithm has been used (e.g., uncompressed 8-bit audio, MP3, etc.). Since every packet carries this field, the encoding can change during transmission. The *Sequence number* is just a counter that is incremented on each RTP packet sent. It is used to detect lost packets.

The *Timestamp* is produced by the stream's source to note when the first sample in the packet was made. This value can help reduce timing variability called jitter at the receiver by decoupling the playback from the packet arrival time. The *Synchronization source identifier* tells which stream the packet belongs to. It is the method used to multiplex and demultiplex multiple data streams onto a single stream of UDP packets. Finally, the *Contributing source identifiers*, if any, are used when mixers are present in the studio. In that case, the mixer is the synchronizing source, and the streams being mixed are listed here.

RTCP—The Real-time Transport Control Protocol

RTP has a little sister protocol (little sibling protocol?) called **RTCP** (**Real-time Transport Control Protocol**). It is defined along with RTP in RFC 3550 and handles feedback, synchronization, and the user interface. It does not transport any media samples.

The first function can be used to provide feedback on delay, variation in delay or jitter, bandwidth, congestion, and other network properties to the sources. This information can be used by the encoding process to increase the data rate (and give better quality) when the network is functioning well and to cut back the data

rate when there is trouble in the network. By providing continuous feedback, the encoding algorithms can be continuously adapted to provide the best quality possible under the current circumstances. For example, if the bandwidth increases or decreases during the transmission, the encoding may switch from MP3 to 8-bit PCM to delta encoding as required. The *Payload type* field is used to tell the destination what encoding algorithm is used for the current packet, making it possible to vary it on demand.

An issue with providing feedback is that the RTCP reports are sent to all participants. For a multicast application with a large group, the bandwidth used by RTCP would quickly grow large. To prevent this from happening, RTCP senders scale down the rate of their reports to collectively consume no more than, say, 5% of the media bandwidth. To do this, each participant needs to know the media bandwidth, which it learns from the sender, and the number of participants, which it estimates by listening to other RTCP reports.

RTCP also handles interstream synchronization. The problem is that different streams may use different clocks, with different granularities and different drift rates. RTCP can be used to keep them in sync.

Finally, RTCP provides a way for naming the various sources (e.g., in ASCII text). This information can be displayed on the receiver's screen to indicate who is talking at the moment.

More information about RTP can be found in Perkins (2003).

Playout with Buffering and Jitter Control

Once the media information reaches the receiver, it must be played out at the right time. In general, this will not be the time at which the RTP packet arrived at the receiver because packets will take slightly different amounts of time to transit the network. Even if the packets are injected with exactly the right intervals between them at the sender, they will reach the receiver with different relative times. This variation in delay is called **jitter**. Even a small amount of packet jitter can cause distracting media artifacts, such as jerky video frames and unintelligible audio, if the media is simply played out as it arrives.

The solution to this problem is to **buffer** packets at the receiver before they are played out to reduce the jitter. As an example, in Fig. 6-32 we see a stream of packets being delivered with a substantial amount of jitter. Packet 1 is sent from the server at $t = 0$ sec and arrives at the client at $t = 1$ sec. Packet 2 undergoes more delay and takes 2 sec to arrive. As the packets arrive, they are buffered on the client machine.

At $t = 10$ sec, playback begins. At this time, packets 1 through 6 have been buffered so that they can be removed from the buffer at uniform intervals for smooth play. In the general case, it is not necessary to use uniform intervals because the RTP timestamps tell when the media should be played.

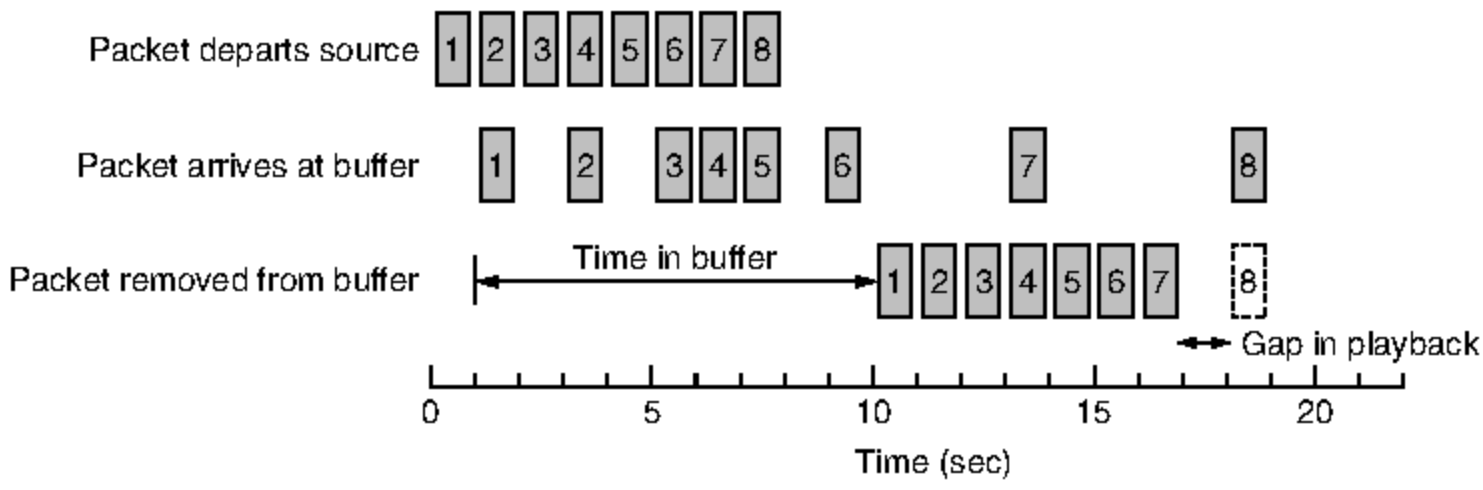

Figure 6-32. Smoothing the output stream by buffering packets.

Unfortunately, we can see that packet 8 has been delayed so much that it is not available when its play slot comes up. There are two options. Packet 8 can be skipped and the player can move on to subsequent packets. Alternatively, playback can stop until packet 8 arrives, creating an annoying gap in the music or movie. In a live media application like a voice-over-IP call, the packet will typically be skipped. Live applications do not work well on hold. In a streaming media application, the player might pause. This problem can be alleviated by delaying the starting time even more, by using a larger buffer. For a streaming audio or video player, buffers of about 10 seconds are often used to ensure that the player receives all of the packets (that are not dropped in the network) in time. For live applications like videoconferencing, short buffers are needed for responsiveness.

A key consideration for smooth playout is the **playback point**, or how long to wait at the receiver for media before playing it out. Deciding how long to wait depends on the jitter. The difference between a low-jitter and high-jitter connection is shown in Fig. 6-33. The average delay may not differ greatly between the two, but if there is high jitter the playback point may need to be much further out to capture 99% of the packets than if there is low jitter.

To pick a good playback point, the application can measure the jitter by looking at the difference between the RTP timestamps and the arrival time. Each difference gives a sample of the delay (plus an arbitrary, fixed offset). However, the delay can change over time due to other, competing traffic and changing routes. To accommodate this change, applications can adapt their playback point while they are running. However, if not done well, changing the playback point can produce an observable glitch to the user. One way to avoid this problem for audio is to adapt the playback point between **talkspurts**, in the gaps in a conversation. No one will notice the difference between a short and slightly longer silence. RTP lets applications set the *M* marker bit to indicate the start of a new talkspurt for this purpose.

If the absolute delay until media is played out is too long, live applications will suffer. Nothing can be done to reduce the propagation delay if a direct path is

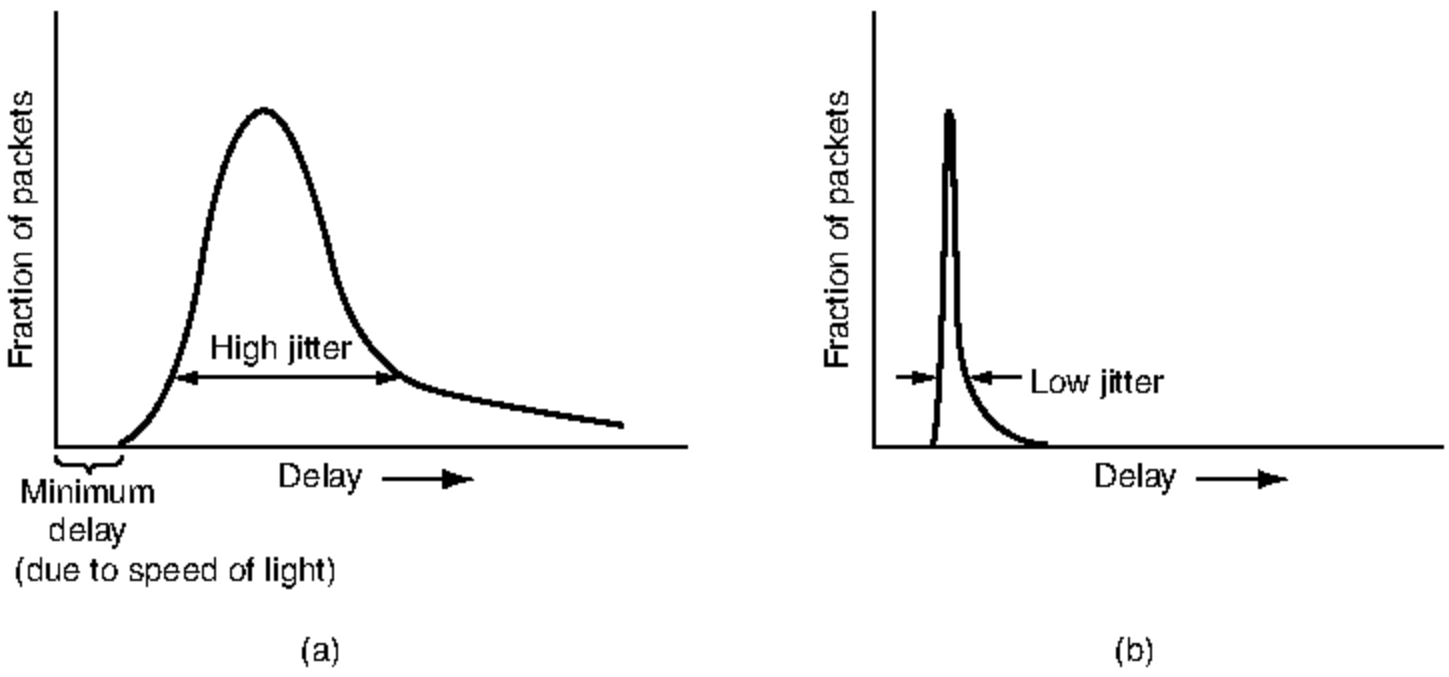

Figure 6-33. (a) High jitter. (b) Low jitter.

already being used. The playback point can be pulled in by simply accepting that a larger fraction of packets will arrive too late to be played. If this is not acceptable, the only way to pull in the playback point is to reduce the jitter by using a better quality of service, for example, the expedited forwarding differentiated service. That is, a better network is needed.

6.5 THE INTERNET TRANSPORT PROTOCOLS: TCP

UDP is a simple protocol and it has some very important uses, such as client-server interactions and multimedia, but for most Internet applications, reliable, sequenced delivery is needed. UDP cannot provide this, so another protocol is required. It is called TCP and is the main workhorse of the Internet. Let us now study it in detail.

6.5.1 Introduction to TCP

TCP (**Transmission Control Protocol**) was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork. An internetwork differs from a single network because different parts may have wildly different topologies, bandwidths, delays, packet sizes, and other parameters. TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures.

TCP was formally defined in RFC 793 in September 1981. As time went on, many improvements have been made, and various errors and inconsistencies have been fixed. To give you a sense of the extent of TCP, the important RFCs are

now RFC 793 plus: clarifications and bug fixes in RFC 1122; extensions for high-performance in RFC 1323; selective acknowledgements in RFC 2018; congestion control in RFC 2581; repurposing of header fields for quality of service in RFC 2873; improved retransmission timers in RFC 2988; and explicit congestion notification in RFC 3168. The full collection is even larger, which led to a guide to the many RFCs, published of course as another RFC document, RFC 4614.

Each machine supporting TCP has a TCP transport entity, either a library procedure, a user process, or most commonly part of the kernel. In all cases, it manages TCP streams and interfaces to the IP layer. A TCP entity accepts user data streams from local processes, breaks them up into pieces not exceeding 64 KB (in practice, often 1460 data bytes in order to fit in a single Ethernet frame with the IP and TCP headers), and sends each piece as a separate IP datagram. When datagrams containing TCP data arrive at a machine, they are given to the TCP entity, which reconstructs the original byte streams. For simplicity, we will sometimes use just "TCP" to mean the TCP transport entity (a piece of software) or the TCP protocol (a set of rules). From the context it will be clear which is meant. For example, in "The user gives TCP the data," the TCP transport entity is clearly intended.

The IP layer gives no guarantee that datagrams will be delivered properly, nor any indication of how fast datagrams may be sent. It is up to TCP to send datagrams fast enough to make use of the capacity but not cause congestion, and to time out and retransmit any datagrams that are not delivered. Datagrams that do arrive may well do so in the wrong order; it is also up to TCP to reassemble them into messages in the proper sequence. In short, TCP must furnish good performance with the reliability that most applications want and that IP does not provide.

6.5.2 The TCP Service Model

TCP service is obtained by both the sender and the receiver creating end points, called **sockets**, as discussed in Sec. 6.1.3. Each socket has a socket number (address) consisting of the IP address of the host and a 16-bit number local to that host, called a **port**. A port is the TCP name for a TSAP. For TCP service to be obtained, a connection must be explicitly established between a socket on one machine and a socket on another machine. The socket calls are listed in Fig. 6-5.

A socket may be used for multiple connections at the same time. In other words, two or more connections may terminate at the same socket. Connections are identified by the socket identifiers at both ends, that is, (*socket1*, *socket2*). No virtual circuit numbers or other identifiers are used.

Port numbers below 1024 are reserved for standard services that can usually only be started by privileged users (e.g., root in UNIX systems). They are called **well-known ports**. For example, any process wishing to remotely retrieve mail from a host can connect to the destination host's port 143 to contact its IMAP

daemon. The list of well-known ports is given at *www.iana.org*. Over 700 have been assigned. A few of the better-known ones are listed in Fig. 6-34.

Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

Figure 6-34. Some assigned ports.

Other ports from 1024 through 49151 can be registered with IANA for use by unprivileged users, but applications can and do choose their own ports. For example, the BitTorrent peer-to-peer file-sharing application (unofficially) uses ports 6881–6887, but may run on other ports as well.

It would certainly be possible to have the FTP daemon attach itself to port 21 at boot time, the SSH daemon attach itself to port 22 at boot time, and so on. However, doing so would clutter up memory with daemons that were idle most of the time. Instead, what is commonly done is to have a single daemon, called **inetd** (**Internet daemon**) in UNIX, attach itself to multiple ports and wait for the first incoming connection. When that occurs, *inetd* forks off a new process and executes the appropriate daemon in it, letting that daemon handle the request. In this way, the daemons other than *inetd* are only active when there is work for them to do. Inetd learns which ports it is to use from a configuration file. Consequently, the system administrator can set up the system to have permanent daemons on the busiest ports (e.g., port 80) and *inetd* on the rest.

All TCP connections are full duplex and point-to-point. Full duplex means that traffic can go in both directions at the same time. Point-to-point means that each connection has exactly two end points. TCP does not support multicasting or broadcasting.

A TCP connection is a byte stream, not a message stream. Message boundaries are not preserved end to end. For example, if the sending process does four 512-byte writes to a TCP stream, these data may be delivered to the receiving process as four 512-byte chunks, two 1024-byte chunks, one 2048-byte chunk (see Fig. 6-35), or some other way. There is no way for the receiver to detect the unit(s) in which the data were written, no matter how hard it tries.

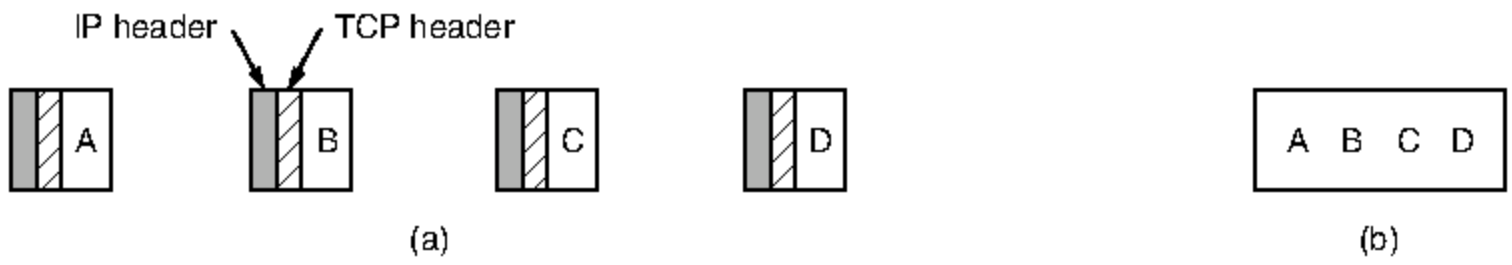

Figure 6-35. (a) Four 512-byte segments sent as separate IP datagrams. (b) The 2048 bytes of data delivered to the application in a single READ call.

Files in UNIX have this property too. The reader of a file cannot tell whether the file was written a block at a time, a byte at a time, or all in one blow. As with a UNIX file, the TCP software has no idea of what the bytes mean and no interest in finding out. A byte is just a byte.

When an application passes data to TCP, TCP may send it immediately or buffer it (in order to collect a larger amount to send at once), at its discretion. However, sometimes the application really wants the data to be sent immediately. For example, suppose a user of an interactive game wants to send a stream of updates. It is essential that the updates be sent immediately, not buffered until there is a collection of them. To force data out, TCP has the notion of a PUSH flag that is carried on packets. The original intent was to let applications tell TCP implementations via the PUSH flag not to delay the transmission. However, applications cannot literally set the PUSH flag when they send data. Instead, different operating systems have evolved different options to expedite transmission (e.g., TCP_NODELAY in Windows and Linux).

For Internet archaeologists, we will also mention one interesting feature of TCP service that remains in the protocol but is rarely used: **urgent data**. When an application has high priority data that should be processed immediately, for example, if an interactive user hits the CTRL-C key to break off a remote computation that has already begun, the sending application can put some control information in the data stream and give it to TCP along with the URGENT flag. This event causes TCP to stop accumulating data and transmit everything it has for that connection immediately.

When the urgent data are received at the destination, the receiving application is interrupted (e.g., given a signal in UNIX terms) so it can stop whatever it was doing and read the data stream to find the urgent data. The end of the urgent data is marked so the application knows when it is over. The start of the urgent data is not marked. It is up to the application to figure that out.

This scheme provides a crude signaling mechanism and leaves everything else up to the application. However, while urgent data is potentially useful, it found no compelling application early on and fell into disuse. Its use is now discouraged because of implementation differences, leaving applications to handle their own signaling. Perhaps future transport protocols will provide better signaling.

6.5.3 The TCP Protocol

In this section, we will give a general overview of the TCP protocol. In the next one, we will go over the protocol header, field by field.

A key feature of TCP, and one that dominates the protocol design, is that every byte on a TCP connection has its own 32-bit sequence number. 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. At modern network speeds, the sequence numbers can be consumed at an alarming rate, as we will see later. Separate 32-bit sequence numbers are carried on packets for the sliding window position in one direction and for acknowledgements in the reverse direction, as discussed below.

The sending and receiving TCP entities exchange data in the form of segments. A **TCP segment** consists of a fixed 20-byte header (plus an optional part) followed by zero or more data bytes. The TCP software decides how big segments should be. It can accumulate data from several writes into one segment or can split data from one write over multiple segments. Two limits restrict the segment size. First, each segment, including the TCP header, must fit in the 65,515-byte IP payload. Second, each link has an **MTU** (**Maximum Transfer Unit**). Each segment must fit in the MTU at the sender and receiver so that it can be sent and received in a single, unfragmented packet. In practice, the MTU is generally 1500 bytes (the Ethernet payload size) and thus defines the upper bound on segment size.

However, it is still possible for IP packets carrying TCP segments to be fragmented when passing over a network path for which some link has a small MTU. If this happens, it degrades performance and causes other problems (Kent and Mogul, 1987). Instead, modern TCP implementations perform **path MTU discovery** by using the technique outlined in RFC 1191 that we described in Sec. 5.5.5. This technique uses ICMP error messages to find the smallest MTU for any link on the path. TCP then adjusts the segment size downwards to avoid fragmentation.

The basic protocol used by TCP entities is the sliding window protocol with a dynamic window size. When a sender transmits a segment, it also starts a timer. When the segment arrives at the destination, the receiving TCP entity sends back a segment (with data if any exist, and otherwise without) bearing an acknowledgement number equal to the next sequence number it expects to receive and the remaining window size. If the sender's timer goes off before the acknowledgement is received, the sender transmits the segment again.

Although this protocol sounds simple, there are many sometimes subtle ins and outs, which we will cover below. Segments can arrive out of order, so bytes 3072–4095 can arrive but cannot be acknowledged because bytes 2048–3071 have not turned up yet. Segments can also be delayed so long in transit that the sender times out and retransmits them. The retransmissions may include different byte

ranges than the original transmission, requiring careful administration to keep track of which bytes have been correctly received so far. However, since each byte in the stream has its own unique offset, it can be done.

TCP must be prepared to deal with these problems and solve them in an efficient way. A considerable amount of effort has gone into optimizing the performance of TCP streams, even in the face of network problems. A number of the algorithms used by many TCP implementations will be discussed below.

6.5.4 The TCP Segment Header

Figure 6-36 shows the layout of a TCP segment. Every segment begins with a fixed-format, 20-byte header. The fixed header may be followed by header options. After the options, if any, up to $65,535 - 20 - 20 = 65,495$ data bytes may follow, where the first 20 refer to the IP header and the second to the TCP header. Segments without any data are legal and are commonly used for acknowledgements and control messages.

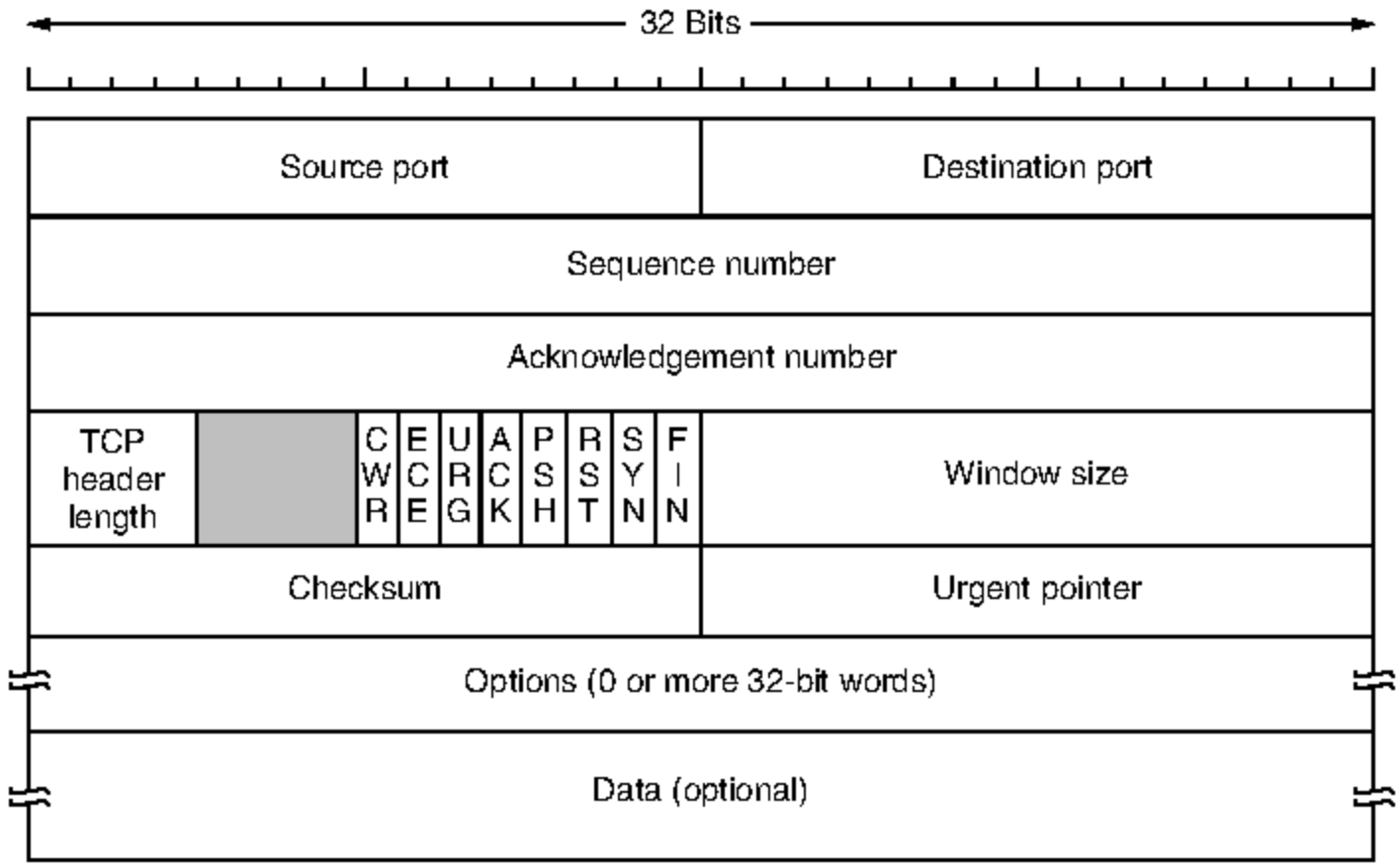

Figure 6-36. The TCP header.

Let us dissect the TCP header field by field. The *Source port* and *Destination port* fields identify the local end points of the connection. A TCP port plus its host's IP address forms a 48-bit unique end point. The source and destination end points together identify the connection. This connection identifier is called a **5 tuple** because it consists of five pieces of information: the protocol (TCP), source IP and source port, and destination IP and destination port.

The *Sequence number* and *Acknowledgement number* fields perform their usual functions. Note that the latter specifies the next in-order byte expected, not the last byte correctly received. It is a **cumulative acknowledgement** because it summarizes the received data with a single number. It does not go beyond lost data. Both are 32 bits because every byte of data is numbered in a TCP stream.

The *TCP header length* tells how many 32-bit words are contained in the TCP header. This information is needed because the *Options* field is of variable length, so the header is, too. Technically, this field really indicates the start of the data within the segment, measured in 32-bit words, but that number is just the header length in words, so the effect is the same.

Next comes a 4-bit field that is not used. The fact that these bits have remained unused for 30 years (as only 2 of the original reserved 6 bits have been reclaimed) is testimony to how well thought out TCP is. Lesser protocols would have needed these bits to fix bugs in the original design.

Now come eight 1-bit flags. *CWR* and *ECE* are used to signal congestion when ECN (Explicit Congestion Notification) is used, as specified in RFC 3168. *ECE* is set to signal an *ECN-Echo* to a TCP sender to tell it to slow down when the TCP receiver gets a congestion indication from the network. *CWR* is set to signal *Congestion Window Reduced* from the TCP sender to the TCP receiver so that it knows the sender has slowed down and can stop sending the *ECN-Echo*. We discuss the role of ECN in TCP congestion control in Sec. 6.5.10.

URG is set to 1 if the *Urgent pointer* is in use. The *Urgent pointer* is used to indicate a byte offset from the current sequence number at which urgent data are to be found. This facility is in lieu of interrupt messages. As we mentioned above, this facility is a bare-bones way of allowing the sender to signal the receiver without getting TCP itself involved in the reason for the interrupt, but it is seldom used.

The *ACK* bit is set to 1 to indicate that the *Acknowledgement number* is valid. This is the case for nearly all packets. If *ACK* is 0, the segment does not contain an acknowledgement, so the *Acknowledgement number* field is ignored.

The *PSH* bit indicates PUSHed data. The receiver is hereby kindly requested to deliver the data to the application upon arrival and not buffer it until a full buffer has been received (which it might otherwise do for efficiency).

The *RST* bit is used to abruptly reset a connection that has become confused due to a host crash or some other reason. It is also used to reject an invalid segment or refuse an attempt to open a connection. In general, if you get a segment with the *RST* bit on, you have a problem on your hands.

The *SYN* bit is used to establish connections. The connection request has $SYN = 1$ and $ACK = 0$ to indicate that the piggyback acknowledgement field is not in use. The connection reply does bear an acknowledgement, however, so it has $SYN = 1$ and $ACK = 1$. In essence, the *SYN* bit is used to denote both CONNECTION REQUEST and CONNECTION ACCEPTED, with the *ACK* bit used to distinguish between those two possibilities.

The *FIN* bit is used to release a connection. It specifies that the sender has no more data to *transmit*. However, after closing a connection, the closing process may continue to *receive* data indefinitely. Both *SYN* and *FIN* segments have sequence numbers and are thus guaranteed to be processed in the correct order.

Flow control in TCP is handled using a variable-sized sliding window. The *Window size* field tells how many bytes may be sent starting at the byte acknowledged. A *Window size* field of 0 is legal and says that the bytes up to and including *Acknowledgement number* − 1 have been received, but that the receiver has not had a chance to consume the data and would like no more data for the moment, thank you. The receiver can later grant permission to send by transmitting a segment with the same *Acknowledgement number* and a nonzero *Window size* field.

In the protocols of Chap. 3, acknowledgements of frames received and permission to send new frames were tied together. This was a consequence of a fixed window size for each protocol. In TCP, acknowledgements and permission to send additional data are completely decoupled. In effect, a receiver can say: "I have received bytes up through k but I do not want any more just now, thank you." This decoupling (in fact, a variable-sized window) gives additional flexibility. We will study it in detail below.

A *Checksum* is also provided for extra reliability. It checksums the header, the data, and a conceptual pseudoheader in exactly the same way as UDP, except that the pseudoheader has the protocol number for TCP (6) and the checksum is mandatory. Please see Sec. 6.4.1 for details.

The *Options* field provides a way to add extra facilities not covered by the regular header. Many options have been defined and several are commonly used. The options are of variable length, fill a multiple of 32 bits by using padding with zeros, and may extend to 40 bytes to accommodate the longest TCP header that can be specified. Some options are carried when a connection is established to negotiate or inform the other side of capabilities. Other options are carried on packets during the lifetime of the connection. Each option has a Type-Length-Value encoding.

A widely used option is the one that allows each host to specify the **MSS** (**Maximum Segment Size**) it is willing to accept. Using large segments is more efficient than using small ones because the 20-byte header can be amortized over more data, but small hosts may not be able to handle big segments. During connection setup, each side can announce its maximum and see its partner's. If a host does not use this option, it defaults to a 536-byte payload. All Internet hosts are required to accept TCP segments of $536 + 20 = 556$ bytes. The maximum segment size in the two directions need not be the same.

For lines with high bandwidth, high delay, or both, the 64-KB window corresponding to a 16-bit field is a problem. For example, on an OC-12 line (of roughly 600 Mbps), it takes less than 1 msec to output a full 64-KB window. If the round-trip propagation delay is 50 msec (which is typical for a transcontinental

fiber), the sender will be idle more than 98% of the time waiting for acknowledgements. A larger window size would allow the sender to keep pumping data out. The **window scale** option allows the sender and receiver to negotiate a window scale factor at the start of a connection. Both sides use the scale factor to shift the *Window size* field up to 14 bits to the left, thus allowing windows of up to 2^{30} bytes. Most TCP implementations support this option.

The **timestamp** option carries a timestamp sent by the sender and echoed by the receiver. It is included in every packet, once its use is established during connection setup, and used to compute round-trip time samples that are used to estimate when a packet has been lost. It is also used as a logical extension of the 32-bit sequence number. On a fast connection, the sequence number may wrap around quickly, leading to possible confusion between old and new data. The **PAWS** (**Protection Against Wrapped Sequence numbers**) scheme discards arriving segments with old timestamps to prevent this problem.

Finally, the **SACK** (**Selective ACKnowledgement**) option lets a receiver tell a sender the ranges of sequence numbers that it has received. It supplements the *Acknowledgement number* and is used after a packet has been lost but subsequent (or duplicate) data has arrived. The new data is not reflected by the *Acknowledgement number* field in the header because that field gives only the next in-order byte that is expected. With SACK, the sender is explicitly aware of what data the receiver has and hence can determine what data should be retransmitted. SACK is defined in RFC 2108 and RFC 2883 and is increasingly used. We describe the use of SACK along with congestion control in Sec. 6.5.10.

6.5.5 TCP Connection Establishment

Connections are established in TCP by means of the three-way handshake discussed in Sec. 6.2.2. To establish a connection, one side, say, the server, passively waits for an incoming connection by executing the LISTEN and ACCEPT primitives in that order, either specifying a specific source or nobody in particular.

The other side, say, the client, executes a CONNECT primitive, specifying the IP address and port to which it wants to connect, the maximum TCP segment size it is willing to accept, and optionally some user data (e.g., a password). The CONNECT primitive sends a TCP segment with the *SYN* bit on and *ACK* bit off and waits for a response.

When this segment arrives at the destination, the TCP entity there checks to see if there is a process that has done a LISTEN on the port given in the *Destination port* field. If not, it sends a reply with the *RST* bit on to reject the connection.

If some process is listening to the port, that process is given the incoming TCP segment. It can either accept or reject the connection. If it accepts, an acknowledgement segment is sent back. The sequence of TCP segments sent in the normal case is shown in Fig. 6-37(a). Note that a *SYN* segment consumes 1 byte of sequence space so that it can be acknowledged unambiguously.

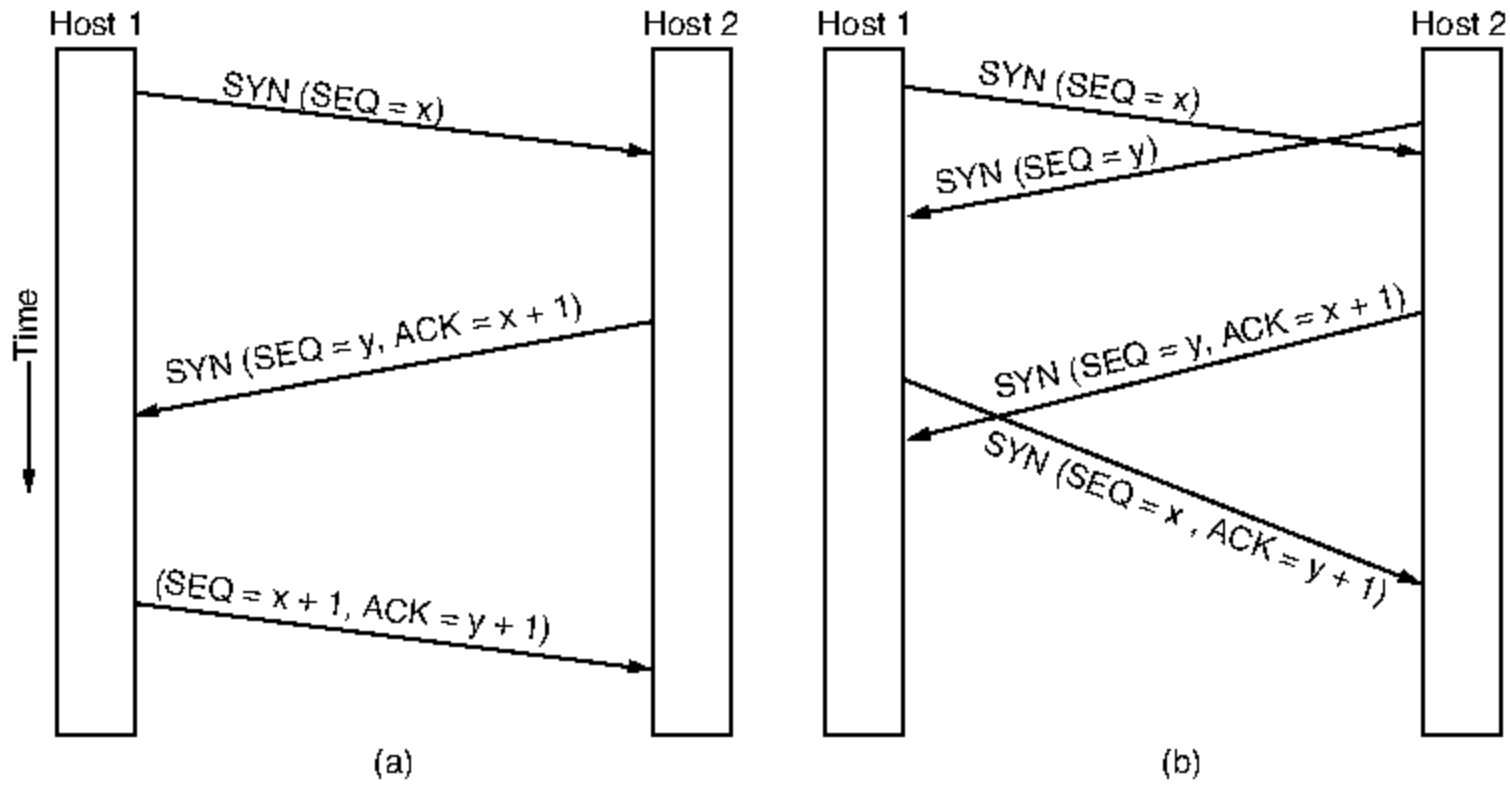

Figure 6-37. (a) TCP connection establishment in the normal case. (b) Simultaneous connection establishment on both sides.

In the event that two hosts simultaneously attempt to establish a connection between the same two sockets, the sequence of events is as illustrated in Fig. 6-37(b). The result of these events is that just one connection is established, not two, because connections are identified by their end points. If the first setup results in a connection identified by (x, y) and the second one does too, only one table entry is made, namely, for (x, y).

Recall that the initial sequence number chosen by each host should cycle slowly, rather than be a constant such as 0. This rule is to protect against delayed duplicate packets, as we discussed in Sec 6.2.2. Originally this was accomplished with a clock-based scheme in which the clock ticked every 4 μsec.

However, a vulnerability with implementing the three-way handshake is that the listening process must remember its sequence number as soon it responds with its own *SYN* segment. This means that a malicious sender can tie up resources on a host by sending a stream of *SYN* segments and never following through to complete the connection. This attack is called a **SYN flood**, and it crippled many Web servers in the 1990s.

One way to defend against this attack is to use **SYN cookies**. Instead of remembering the sequence number, a host chooses a cryptographically generated sequence number, puts it on the outgoing segment, and forgets it. If the three-way handshake completes, this sequence number (plus 1) will be returned to the host. It can then regenerate the correct sequence number by running the same cryptographic function, as long as the inputs to that function are known, for example, the other host's IP address and port, and a local secret. This procedure allows the host to check that an acknowledged sequence number is correct without having to

remember the sequence number separately. There are some caveats, such as the inability to handle TCP options, so SYN cookies may be used only when the host is subject to a SYN flood. However, they are an interesting twist on connection establishment. For more information, see RFC 4987 and Lemon (2002).

6.5.6 TCP Connection Release

Although TCP connections are full duplex, to understand how connections are released it is best to think of them as a pair of simplex connections. Each simplex connection is released independently of its sibling. To release a connection, either party can send a TCP segment with the *FIN* bit set, which means that it has no more data to transmit. When the *FIN* is acknowledged, that direction is shut down for new data. Data may continue to flow indefinitely in the other direction, however. When both directions have been shut down, the connection is released. Normally, four TCP segments are needed to release a connection: one *FIN* and one *ACK* for each direction. However, it is possible for the first *ACK* and the second *FIN* to be contained in the same segment, reducing the total count to three.

Just as with telephone calls in which both people say goodbye and hang up the phone simultaneously, both ends of a TCP connection may send *FIN* segments at the same time. These are each acknowledged in the usual way, and the connection is shut down. There is, in fact, no essential difference between the two hosts releasing sequentially or simultaneously.

To avoid the two-army problem (discussed in Sec. 6.2.3), timers are used. If a response to a *FIN* is not forthcoming within two maximum packet lifetimes, the sender of the *FIN* releases the connection. The other side will eventually notice that nobody seems to be listening to it anymore and will time out as well. While this solution is not perfect, given the fact that a perfect solution is theoretically impossible, it will have to do. In practice, problems rarely arise.

6.5.7 TCP Connection Management Modeling

The steps required to establish and release connections can be represented in a finite state machine with the 11 states listed in Fig. 6-38. In each state, certain events are legal. When a legal event happens, some action may be taken. If some other event happens, an error is reported.

Each connection starts in the *CLOSED* state. It leaves that state when it does either a passive open (LISTEN) or an active open (CONNECT). If the other side does the opposite one, a connection is established and the state becomes *ESTABLISHED*. Connection release can be initiated by either side. When it is complete, the state returns to *CLOSED*.

The finite state machine itself is shown in Fig. 6-39. The common case of a client actively connecting to a passive server is shown with heavy lines—solid for the client, dotted for the server. The lightface lines are unusual event sequences.

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIME WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

Figure 6-38. The states used in the TCP connection management finite state machine.

Each line in Fig. 6-39 is marked by an *event/action* pair. The event can either be a user-initiated system call (CONNECT, LISTEN, SEND, or CLOSE), a segment arrival (*SYN*, *FIN*, *ACK*, or *RST*), or, in one case, a timeout of twice the maximum packet lifetime. The action is the sending of a control segment (*SYN*, *FIN*, or *RST*) or nothing, indicated by —. Comments are shown in parentheses.

One can best understand the diagram by first following the path of a client (the heavy solid line), then later following the path of a server (the heavy dashed line). When an application program on the client machine issues a CONNECT request, the local TCP entity creates a connection record, marks it as being in the *SYN SENT* state, and shoots off a *SYN* segment. Note that many connections may be open (or being opened) at the same time on behalf of multiple applications, so the state is per connection and recorded in the connection record. When the *SYN+ACK* arrives, TCP sends the final *ACK* of the three-way handshake and switches into the *ESTABLISHED* state. Data can now be sent and received.

When an application is finished, it executes a CLOSE primitive, which causes the local TCP entity to send a *FIN* segment and wait for the corresponding *ACK* (dashed box marked "active close"). When the *ACK* arrives, a transition is made to the state *FIN WAIT 2* and one direction of the connection is closed. When the other side closes, too, a *FIN* comes in, which is acknowledged. Now both sides are closed, but TCP waits a time equal to twice the maximum packet lifetime to guarantee that all packets from the connection have died off, just in case the acknowledgement was lost. When the timer goes off, TCP deletes the connection record.

Now let us examine connection management from the server's viewpoint. The server does a LISTEN and settles down to see who turns up. When a *SYN*

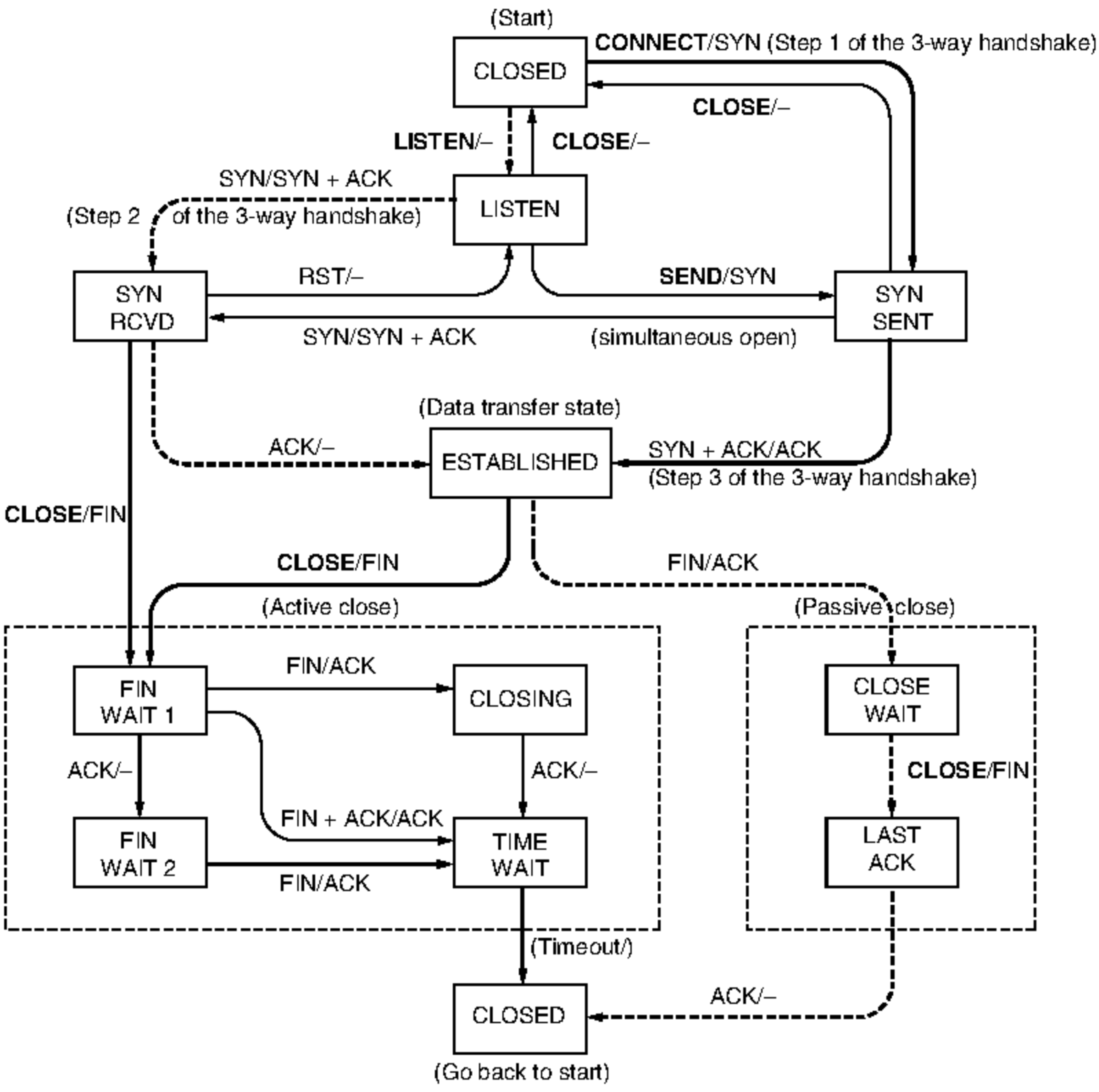

Figure 6-39. TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. Each transition is labeled with the event causing it and the action resulting from it, separated by a slash.

comes in, it is acknowledged and the server goes to the *SYN RCVD* state. When the server's *SYN* is itself acknowledged, the three-way handshake is complete and the server goes to the *ESTABLISHED* state. Data transfer can now occur.

When the client is done transmitting its data, it does a CLOSE, which causes a *FIN* to arrive at the server (dashed box marked "passive close"). The server is then signaled. When it, too, does a CLOSE, a *FIN* is sent to the client. When the

client's acknowledgement shows up, the server releases the connection and deletes the connection record.

6.5.8 TCP Sliding Window

As mentioned earlier, window management in TCP decouples the issues of acknowledgement of the correct receipt of segments and receiver buffer allocation. For example, suppose the receiver has a 4096-byte buffer, as shown in Fig. 6-40. If the sender transmits a 2048-byte segment that is correctly received, the receiver will acknowledge the segment. However, since it now has only 2048 bytes of buffer space (until the application removes some data from the buffer), it will advertise a window of 2048 starting at the next byte expected.

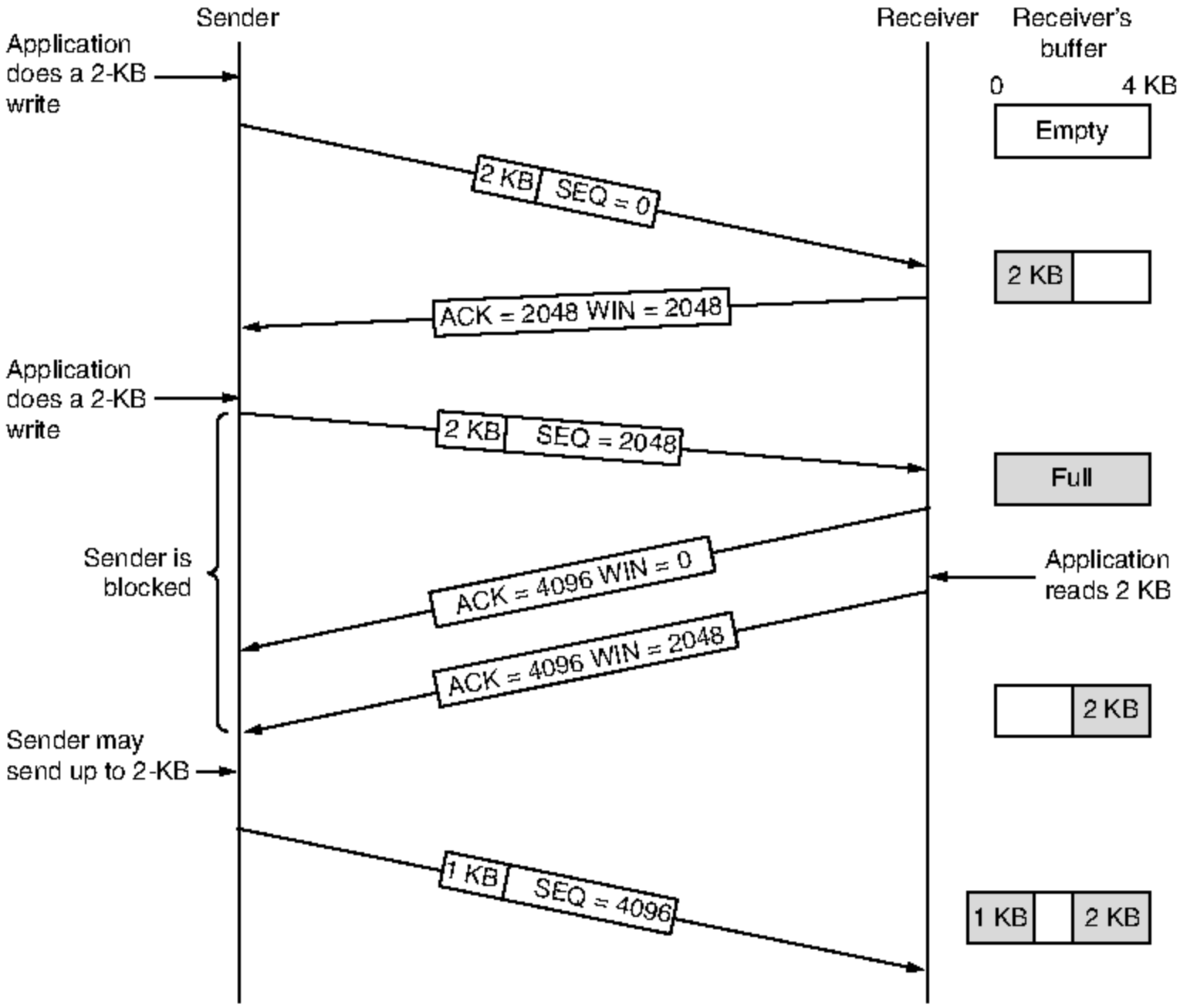

Figure 6-40. Window management in TCP.

Now the sender transmits another 2048 bytes, which are acknowledged, but the advertised window is of size 0. The sender must stop until the application

process on the receiving host has removed some data from the buffer, at which time TCP can advertise a larger window and more data can be sent.

When the window is 0, the sender may not normally send segments, with two exceptions. First, urgent data may be sent, for example, to allow the user to kill the process running on the remote machine. Second, the sender may send a 1-byte segment to force the receiver to reannounce the next byte expected and the window size. This packet is called a **window probe**. The TCP standard explicitly provides this option to prevent deadlock if a window update ever gets lost.

Senders are not required to transmit data as soon as they come in from the application. Neither are receivers required to send acknowledgements as soon as possible. For example, in Fig. 6-40, when the first 2 KB of data came in, TCP, knowing that it had a 4-KB window, would have been completely correct in just buffering the data until another 2 KB came in, to be able to transmit a segment with a 4-KB payload. This freedom can be used to improve performance.

Consider a connection to a remote terminal, for example using SSH or telnet, that reacts on every keystroke. In the worst case, whenever a character arrives at the sending TCP entity, TCP creates a 21-byte TCP segment, which it gives to IP to send as a 41-byte IP datagram. At the receiving side, TCP immediately sends a 40-byte acknowledgement (20 bytes of TCP header and 20 bytes of IP header). Later, when the remote terminal has read the byte, TCP sends a window update, moving the window 1 byte to the right. This packet is also 40 bytes. Finally, when the remote terminal has processed the character, it echoes the character for local display using a 41-byte packet. In all, 162 bytes of bandwidth are used and four segments are sent for each character typed. When bandwidth is scarce, this method of doing business is not desirable.

One approach that many TCP implementations use to optimize this situation is called **delayed acknowledgements**. The idea is to delay acknowledgements and window updates for up to 500 msec in the hope of acquiring some data on which to hitch a free ride. Assuming the terminal echoes within 500 msec, only one 41-byte packet now need be sent back by the remote side, cutting the packet count and bandwidth usage in half.

Although delayed acknowledgements reduce the load placed on the network by the receiver, a sender that sends multiple short packets (e.g., 41-byte packets containing 1 byte of data) is still operating inefficiently. A way to reduce this usage is known as **Nagle's algorithm** (Nagle, 1984). What Nagle suggested is simple: when data come into the sender in small pieces, just send the first piece and buffer all the rest until the first piece is acknowledged. Then send all the buffered data in one TCP segment and start buffering again until the next segment is acknowledged. That is, only one short packet can be outstanding at any time. If many pieces of data are sent by the application in one round-trip time, Nagle's algorithm will put the many pieces in one segment, greatly reducing the bandwidth used. The algorithm additionally says that a new segment should be sent if enough data have trickled in to fill a maximum segment.

Nagle's algorithm is widely used by TCP implementations, but there are times when it is better to disable it. In particular, in interactive games that are run over the Internet, the players typically want a rapid stream of short update packets. Gathering the updates to send them in bursts makes the game respond erratically, which makes for unhappy users. A more subtle problem is that Nagle's algorithm can sometimes interact with delayed acknowledgements to cause a temporary deadlock: the receiver waits for data on which to piggyback an acknowledgement, and the sender waits on the acknowledgement to send more data. This interaction can delay the downloads of Web pages. Because of these problems, Nagle's algorithm can be disabled (which is called the *TCP_NODELAY* option). Mogul and Minshall (2001) discuss this and other solutions.

Another problem that can degrade TCP performance is the **silly window syndrome** (Clark, 1982). This problem occurs when data are passed to the sending TCP entity in large blocks, but an interactive application on the receiving side reads data only 1 byte at a time. To see the problem, look at Fig. 6-41. Initially, the TCP buffer on the receiving side is full (i.e., it has a window of size 0) and the sender knows this. Then the interactive application reads one character from the TCP stream. This action makes the receiving TCP happy, so it sends a window update to the sender saying that it is all right to send 1 byte. The sender obliges and sends 1 byte. The buffer is now full, so the receiver acknowledges the 1-byte segment and sets the window to 0. This behavior can go on forever.

Clark's solution is to prevent the receiver from sending a window update for 1 byte. Instead, it is forced to wait until it has a decent amount of space available and advertise that instead. Specifically, the receiver should not send a window update until it can handle the maximum segment size it advertised when the connection was established or until its buffer is half empty, whichever is smaller. Furthermore, the sender can also help by not sending tiny segments. Instead, it should wait until it can send a full segment, or at least one containing half of the receiver's buffer size.

Nagle's algorithm and Clark's solution to the silly window syndrome are complementary. Nagle was trying to solve the problem caused by the sending application delivering data to TCP a byte at a time. Clark was trying to solve the problem of the receiving application sucking the data up from TCP a byte at a time. Both solutions are valid and can work together. The goal is for the sender not to send small segments and the receiver not to ask for them.

The receiving TCP can go further in improving performance than just doing window updates in large units. Like the sending TCP, it can also buffer data, so it can block a READ request from the application until it has a large chunk of data for it. Doing so reduces the number of calls to TCP (and the overhead). It also increases the response time, but for noninteractive applications like file transfer, efficiency may be more important than response time to individual requests.

Another issue that the receiver must handle is that segments may arrive out of order. The receiver will buffer the data until it can be passed up to the application

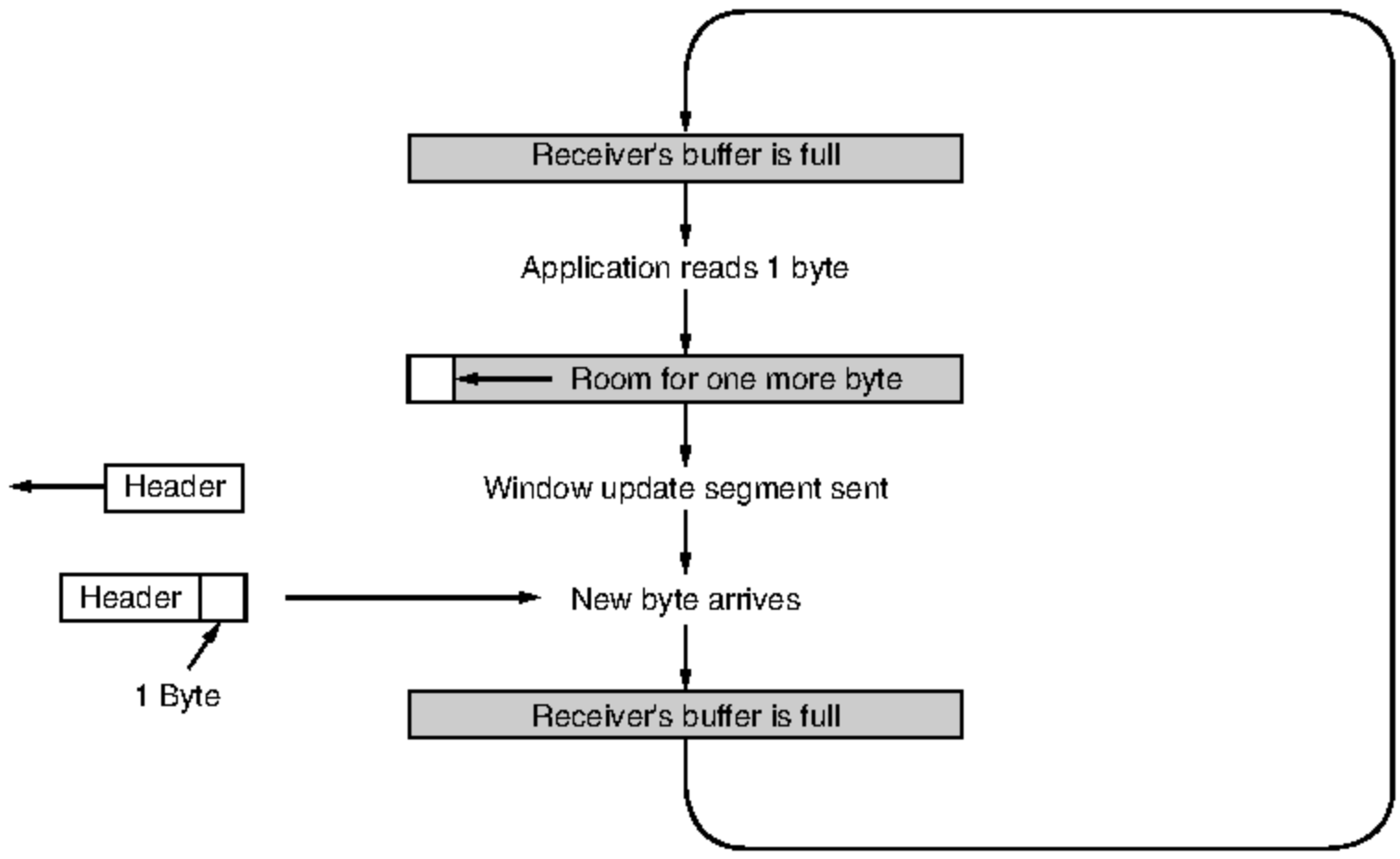

Figure 6-41. Silly window syndrome.

in order. Actually, nothing bad would happen if out-of-order segments were discarded, since they would eventually be retransmitted by the sender, but it would be wasteful.

Acknowledgements can be sent only when all the data up to the byte acknowledged have been received. This is called a **cumulative acknowledgement**. If the receiver gets segments 0, 1, 2, 4, 5, 6, and 7, it can acknowledge everything up to and including the last byte in segment 2. When the sender times out, it then retransmits segment 3. As the receiver has buffered segments 4 through 7, upon receipt of segment 3 it can acknowledge all bytes up to the end of segment 7.

6.5.9 TCP Timer Management

TCP uses multiple timers (at least conceptually) to do its work. The most important of these is the **RTO** (**Retransmission TimeOut**). When a segment is sent, a retransmission timer is started. If the segment is acknowledged before the timer expires, the timer is stopped. If, on the other hand, the timer goes off before the acknowledgement comes in, the segment is retransmitted (and the timer os started again). The question that arises is: how long should the timeout be?

This problem is much more difficult in the transport layer than in data link protocols such as 802.11. In the latter case, the expected delay is measured in

microseconds and is highly predictable (i.e., has a low variance), so the timer can be set to go off just slightly after the acknowledgement is expected, as shown in Fig. 6-42(a). Since acknowledgements are rarely delayed in the data link layer (due to lack of congestion), the absence of an acknowledgement at the expected time generally means either the frame or the acknowledgement has been lost.

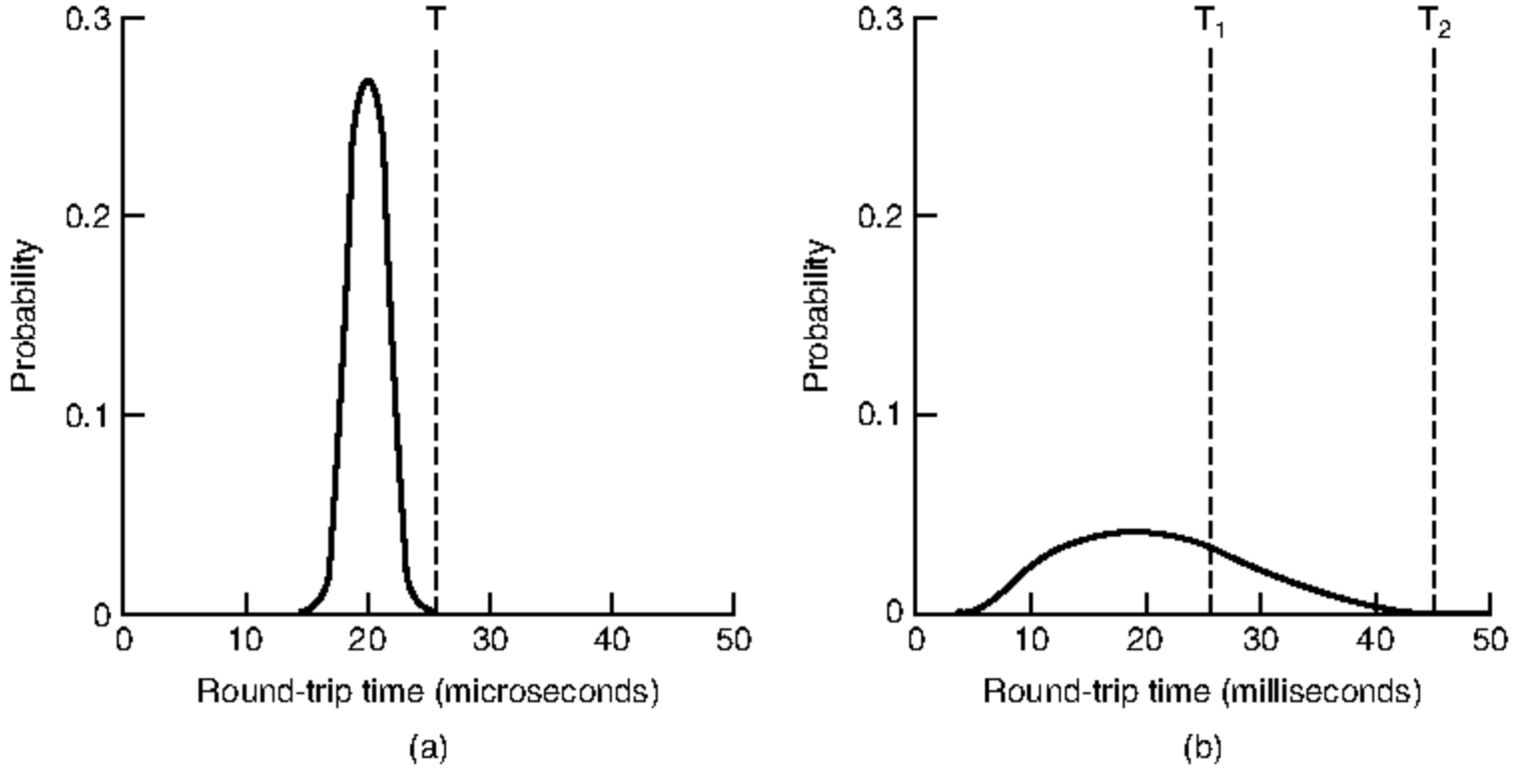

Figure 6-42. (a) Probability density of acknowledgement arrival times in the data link layer. (b) Probability density of acknowledgement arrival times for TCP.

TCP is faced with a radically different environment. The probability density function for the time it takes for a TCP acknowledgement to come back looks more like Fig. 6-42(b) than Fig. 6-42(a). It is larger and more variable. Determining the round-trip time to the destination is tricky. Even when it is known, deciding on the timeout interval is also difficult. If the timeout is set too short, say, T_1 in Fig. 6-42(b), unnecessary retransmissions will occur, clogging the Internet with useless packets. If it is set too long (e.g., T_2), performance will suffer due to the long retransmission delay whenever a packet is lost. Furthermore, the mean and variance of the acknowledgement arrival distribution can change rapidly within a few seconds as congestion builds up or is resolved.

The solution is to use a dynamic algorithm that constantly adapts the timeout interval, based on continuous measurements of network performance. The algorithm generally used by TCP is due to Jacobson (1988) and works as follows. For each connection, TCP maintains a variable, *SRTT* (Smoothed Round-Trip Time), that is the best current estimate of the round-trip time to the destination in question. When a segment is sent, a timer is started, both to see how long the acknowledgement takes and also to trigger a retransmission if it takes too long. If

the acknowledgement gets back before the timer expires, TCP measures how long the acknowledgement took, say, *R*. It then updates *SRTT* according to the formula

$$SRTT = \alpha\, SRTT + (1 - \alpha)\, R$$

where α is a smoothing factor that determines how quickly the old values are forgotten. Typically, $\alpha = 7/8$. This kind of formula is an **EWMA** (**Exponentially Weighted Moving Average**) or low-pass filter that discards noise in the samples.

Even given a good value of *SRTT*, choosing a suitable retransmission timeout is a nontrivial matter. Initial implementations of TCP used $2xRTT$, but experience showed that a constant value was too inflexible because it failed to respond when the variance went up. In particular, queueing models of random (i.e., Poisson) traffic predict that when the load approaches capacity, the delay becomes large and highly variable. This can lead to the retransmission timer firing and a copy of the packet being retransmitted although the original packet is still transiting the network. It is all the more likely to happen under conditions of high load, which is the worst time at which to send additional packets into the network.

To fix this problem, Jacobson proposed making the timeout value sensitive to the variance in round-trip times as well as the smoothed round-trip time. This change requires keeping track of another smoothed variable, *RTTVAR* (Round-Trip Time VARiation) that is updated using the formula

$$RTTVAR = \beta\, RTTVAR + (1 - \beta)\, |SRTT - R|$$

This is an EWMA as before, and typically $\beta = 3/4$. The retransmission timeout, *RTO*, is set to be

$$RTO = SRTT + 4 \times RTTVAR$$

The choice of the factor 4 is somewhat arbitrary, but multiplication by 4 can be done with a single shift, and less than 1% of all packets come in more than four standard deviations late. Note that *RTTVAR* is not exactly the same as the standard deviation (it is really the mean deviation), but it is close enough in practice. Jacobson's paper is full of clever tricks to compute timeouts using only integer adds, subtracts, and shifts. This economy is not needed for modern hosts, but it has become part of the culture that allows TCP to run on all manner of devices, from supercomputers down to tiny devices. So far nobody has put it on an RFID chip, but someday? Who knows.

More details of how to compute this timeout, including initial settings of the variables, are given in RFC 2988. The retransmission timer is also held to a minimum of 1 second, regardless of the estimates. This is a conservative value chosen to prevent spurious retransmissions based on measurements (Allman and Paxson, 1999).

One problem that occurs with gathering the samples, *R*, of the round-trip time is what to do when a segment times out and is sent again. When the acknowledgement comes in, it is unclear whether the acknowledgement refers to the first

transmission or a later one. Guessing wrong can seriously contaminate the retransmission timeout. Phil Karn discovered this problem the hard way. Karn is an amateur radio enthusiast interested in transmitting TCP/IP packets by ham radio, a notoriously unreliable medium. He made a simple proposal: do not update estimates on any segments that have been retransmitted. Additionally, the timeout is doubled on each successive retransmission until the segments get through the first time. This fix is called **Karn's algorithm** (Karn and Partridge, 1987). Most TCP implementations use it.

The retransmission timer is not the only timer TCP uses. A second timer is the **persistence timer**. It is designed to prevent the following deadlock. The receiver sends an acknowledgement with a window size of 0, telling the sender to wait. Later, the receiver updates the window, but the packet with the update is lost. Now the sender and the receiver are each waiting for the other to do something. When the persistence timer goes off, the sender transmits a probe to the receiver. The response to the probe gives the window size. If it is still 0, the persistence timer is set again and the cycle repeats. If it is nonzero, data can now be sent.

A third timer that some implementations use is the **keepalive timer**. When a connection has been idle for a long time, the keepalive timer may go off to cause one side to check whether the other side is still there. If it fails to respond, the connection is terminated. This feature is controversial because it adds overhead and may terminate an otherwise healthy connection due to a transient network partition.

The last timer used on each TCP connection is the one used in the *TIME WAIT* state while closing. It runs for twice the maximum packet lifetime to make sure that when a connection is closed, all packets created by it have died off.

6.5.10 TCP Congestion Control

We have saved one of the key functions of TCP for last: congestion control. When the load offered to any network is more than it can handle, congestion builds up. The Internet is no exception. The network layer detects congestion when queues grow large at routers and tries to manage it, if only by dropping packets. It is up to the transport layer to receive congestion feedback from the network layer and slow down the rate of traffic that it is sending into the network. In the Internet, TCP plays the main role in controlling congestion, as well as the main role in reliable transport. That is why it is such a special protocol.

We covered the general situation of congestion control in Sec. 6.3. One key takeaway was that a transport protocol using an AIMD (Additive Increase Multiplicative Decrease) control law in response to binary congestion signals from the network would converge to a fair and efficient bandwidth allocation. TCP congestion control is based on implementing this approach using a window and with packet loss as the binary signal. To do so, TCP maintains a **congestion window**

whose size is the number of bytes the sender may have in the network at any time. The corresponding rate is the window size divided by the round-trip time of the connection. TCP adjusts the size of the window according to the AIMD rule.

Recall that the congestion window is maintained *in addition* to the flow control window, which specifies the number of bytes that the receiver can buffer. Both windows are tracked in parallel, and the number of bytes that may be sent is the smaller of the two windows. Thus, the effective window is the smaller of what the sender thinks is all right and what the receiver thinks is all right. It takes two to tango. TCP will stop sending data if either the congestion or the flow control window is temporarily full. If the receiver says "send 64 KB" but the sender knows that bursts of more than 32 KB clog the network, it will send 32 KB. On the other hand, if the receiver says "send 64 KB" and the sender knows that bursts of up to 128 KB get through effortlessly, it will send the full 64 KB requested. The flow control window was described earlier, and in what follows we will only describe the congestion window.

Modern congestion control was added to TCP largely through the efforts of Van Jacobson (1988). It is a fascinating story. Starting in 1986, the growing popularity of the early Internet led to the first occurrence of what became known as a **congestion collapse**, a prolonged period during which goodput dropped precipitously (i.e., by more than a factor of 100) due to congestion in the network. Jacobson (and many others) set out to understand what was happening and remedy the situation.

The high-level fix that Jacobson implemented was to approximate an AIMD congestion window. The interesting part, and much of the complexity of TCP congestion control, is how he added this to an existing implementation without changing any of the message formats, which made it instantly deployable. To start, he observed that packet loss is a suitable signal of congestion. This signal comes a little late (as the network is already congested) but it is quite dependable. After all, it is difficult to build a router that does not drop packets when it is overloaded. This fact is unlikely to change. Even when terabyte memories appear to buffer vast numbers of packets, we will probably have terabit/sec networks to fill up those memories.

However, using packet loss as a congestion signal depends on transmission errors being relatively rare. This is not normally the case for wireless links such as 802.11, which is why they include their own retransmission mechanism at the link layer. Because of wireless retransmissions, network layer packet loss due to transmission errors is normally masked on wireless networks. It is also rare on other links because wires and optical fibers typically have low bit-error rates.

All the Internet TCP algorithms assume that lost packets are caused by congestion and monitor timeouts and look for signs of trouble the way miners watch their canaries. A good retransmission timer is needed to detect packet loss signals accurately and in a timely manner. We have already discussed how the TCP retransmission timer includes estimates of the mean and variation in round-trip

times. Fixing this timer, by including the variation factor, was an important step in Jacobson's work. Given a good retransmission timeout, the TCP sender can track the outstanding number of bytes, which are loading the network. It simply looks at the difference between the sequence numbers that are transmitted and acknowledged.

Now it seems that our task is easy. All we need to do is to track the congestion window, using sequence and acknowledgement numbers, and adjust the congestion window using an AIMD rule. As you might have expected, it is more complicated than that. A first consideration is that the way packets are sent into the network, even over short periods of time, must be matched to the network path. Otherwise the traffic will cause congestion. For example, consider a host with a congestion window of 64 KB attached to a 1-Gbps switched Ethernet. If the host sends the entire window at once, this burst of traffic may travel over a slow 1-Mbps ADSL line further along the path. The burst that took only half a millisecond on the 1-Gbps line will clog the 1-Mbps line for half a second, completely disrupting protocols such as voice over IP. This behavior might be a good idea for a protocol designed to cause congestion, but not for a protocol to control it.

However, it turns out that we can use small bursts of packets to our advantage. Fig. 6-43 shows what happens when a sender on a fast network (the 1-Gbps link) sends a small burst of four packets to a receiver on a slow network (the 1-Mbps link) that is the bottleneck or slowest part of the path. Initially the four packets travel over the link as quickly as they can be sent by the sender. At the router, they are queued while being sent because it takes longer to send a packet over the slow link than to receive the next packet over the fast link. But the queue is not large because only a small number of packets were sent at once. Note the increased length of the packets on the slow link. The same packet, of 1 KB say, is now longer because it takes more time to send it on a slow link than on a fast one.

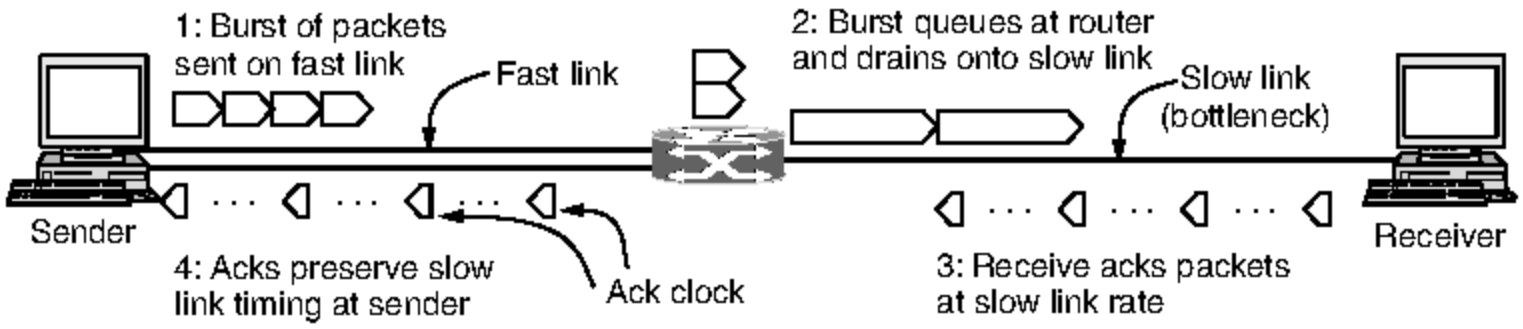

Figure 6-43. A burst of packets from a sender and the returning ack clock.

Eventually the packets get to the receiver, where they are acknowledged. The times for the acknowledgements reflect the times at which the packets arrived at the receiver after crossing the slow link. They are spread out compared to the original packets on the fast link. As these acknowledgements travel over the network and back to the sender they preserve this timing.

The key observation is this: the acknowledgements return to the sender at about the rate that packets can be sent over the slowest link in the path. This is precisely the rate that the sender wants to use. If it injects new packets into the network at this rate, they will be sent as fast as the slow link permits, but they will not queue up and congest any router along the path. This timing is known as an **ack clock**. It is an essential part of TCP. By using an ack clock, TCP smoothes out traffic and avoids unnecessary queues at routers.

A second consideration is that the AIMD rule will take a very long time to reach a good operating point on fast networks if the congestion window is started from a small size. Consider a modest network path that can support 10 Mbps with an RTT of 100 msec. The appropriate congestion window is the bandwidth-delay product, which is 1 Mbit or 100 packets of 1250 bytes each. If the congestion window starts at 1 packet and increases by 1 packet every RTT, it will be 100 RTTs or 10 seconds before the connection is running at about the right rate. That is a long time to wait just to get to the right speed for a transfer. We could reduce this startup time by starting with a larger initial window, say of 50 packets. But this window would be far too large for slow or short links. It would cause congestion if used all at once, as we have just described.

Instead, the solution Jacobson chose to handle both of these considerations is a mix of linear and multiplicative increase. When a connection is established, the sender initializes the congestion window to a small initial value of at most four segments; the details are described in RFC 3390, and the use of four segments is an increase from an earlier initial value of one segment based on experience. The sender then sends the initial window. The packets will take a round-trip time to be acknowledged. For each segment that is acknowledged before the retransmission timer goes off, the sender adds one segment's worth of bytes to the congestion window. Plus, as that segment has been acknowledged, there is now one less segment in the network. The upshot is that every acknowledged segment allows two more segments to be sent. The congestion window is doubling every round-trip time.

This algorithm is called **slow start**, but it is not slow at all—it is exponential growth—except in comparison to the previous algorithm that let an entire flow control window be sent all at once. Slow start is shown in Fig. 6-44. In the first round-trip time, the sender injects one packet into the network (and the receiver receives one packet). Two packets are sent in the next round-trip time, then four packets in the third round-trip time.

Slow-start works well over a range of link speeds and round-trip times, and uses an ack clock to match the rate of sender transmissions to the network path. Take a look at the way acknowledgements return from the sender to the receiver in Fig. 6-44. When the sender gets an acknowledgement, it increases the congestion window by one and immediately sends two packets into the network. (One packet is the increase by one; the other packet is a replacement for the packet that has been acknowledged and left the network. At all times, the number of

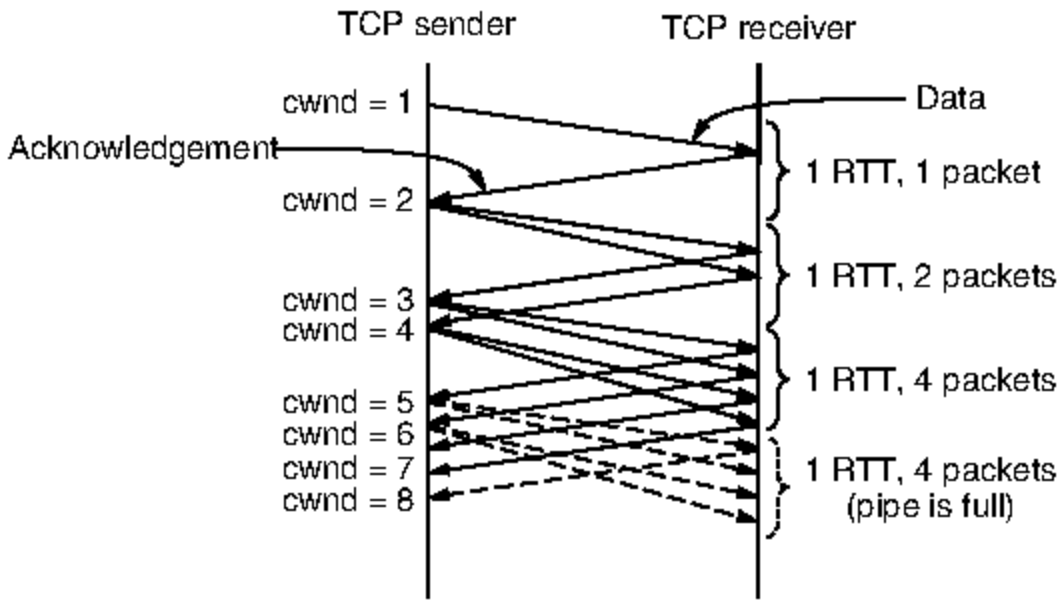

Figure 6-44. Slow start from an initial congestion window of one segment.

unacknowledged packets is given by the congestion window.) However, these two packets will not necessarily arrive at the receiver as closely spaced as when they were sent. For example, suppose the sender is on a 100-Mbps Ethernet. Each packet of 1250 bytes takes 100 μsec to send. So the delay between the packets can be as small as 100 μsec. The situation changes if these packets go across a 1-Mbps ADSL link anywhere along the path. It now takes 10 msec to send the same packet. This means that the minimum spacing between the two packets has grown by a factor of 100. Unless the packets have to wait together in a queue on a later link, the spacing will remain large.

In Fig. 6-44, this effect is shown by enforcing a minimum spacing between data packets arriving at the receiver. The same spacing is kept when the receiver sends acknowledgements, and thus when the sender receives the acknowledgements. If the network path is slow, acknowledgements will come in slowly (after a delay of an RTT). If the network path is fast, acknowledgements will come in quickly (again, after the RTT). All the sender has to do is follow the timing of the ack clock as it injects new packets, which is what slow start does.

Because slow start causes exponential growth, eventually (and sooner rather than later) it will send too many packets into the network too quickly. When this happens, queues will build up in the network. When the queues are full, one or more packets will be lost. After this happens, the TCP sender will time out when an acknowledgement fails to arrive in time. There is evidence of slow start growing too fast in Fig. 6-44. After three RTTs, four packets are in the network. These four packets take an entire RTT to arrive at the receiver. That is, a congestion window of four packets is the right size for this connection. However, as these packets are acknowledged, slow start continues to grow the congestion window, reaching eight packets in another RTT. Only four of these packets can reach the receiver in one RTT, no matter how many are sent. That is, the network pipe is full. Additional packets placed into the network by the sender will build up in

router queues, since they cannot be delivered to the receiver quickly enough. Congestion and packet loss will occur soon.

To keep slow start under control, the sender keeps a threshold for the connection called the **slow start threshold**. Initially this value is set arbitrarily high, to the size of the flow control window, so that it will not limit the connection. TCP keeps increasing the congestion window in slow start until a timeout occurs or the congestion window exceeds the threshold (or the receiver's window is filled).

Whenever a packet loss is detected, for example, by a timeout, the slow start threshold is set to be half of the congestion window and the entire process is restarted. The idea is that the current window is too large because it caused congestion previously that is only now detected by a timeout. Half of the window, which was used successfully at an earlier time, is probably a better estimate for a congestion window that is close to the path capacity but will not cause loss. In our example in Fig. 6-44, growing the congestion window to eight packets may cause loss, while the congestion window of four packets in the previous RTT was the right value. The congestion window is then reset to its small initial value and slow start resumes.

Whenever the slow start threshold is crossed, TCP switches from slow start to additive increase. In this mode, the congestion window is increased by one segment every round-trip time. Like slow start, this is usually implemented with an increase for every segment that is acknowledged, rather than an increase once per RTT. Call the congestion window *cwnd* and the maximum segment size *MSS*. A common approximation is to increase *cwnd* by $(MSS \times MSS)/cwnd$ for each of the $cwnd/MSS$ packets that may be acknowledged. This increase does not need to be fast. The whole idea is for a TCP connection to spend a lot of time with its congestion window close to the optimum value—not so small that throughput will be low, and not so large that congestion will occur.

Additive increase is shown in Fig. 6-45 for the same situation as slow start. At the end of every RTT, the sender's congestion window has grown enough that it can inject an additional packet into the network. Compared to slow start, the linear rate of growth is much slower. It makes little difference for small congestion windows, as is the case here, but a large difference in the time taken to grow the congestion window to 100 segments, for example.

There is something else that we can do to improve performance too. The defect in the scheme so far is waiting for a timeout. Timeouts are relatively long because they must be conservative. After a packet is lost, the receiver cannot acknowledge past it, so the acknowledgement number will stay fixed, and the sender will not be able to send any new packets into the network because its congestion window remains full. This condition can continue for a relatively long period until the timer fires and the lost packet is retransmitted. At that stage, TCP slow starts again.

There is a quick way for the sender to recognize that one of its packets has been lost. As packets beyond the lost packet arrive at the receiver, they trigger