queue each time around. This results in each flow getting $1/n$th of the bandwidth when there are n flows. With WFQ, however, one queue might have a weight of 2, a second queue might have a weight of 1, and a third queue might have a weight of 3. Assuming that each queue always contains a packet waiting to be transmitted, the first flow will get one-third of the available bandwidth, the second will get one-sixth of the available bandwidth, and the third will get one-half of the available bandwidth.

While we have described WFQ in terms of flows, note that it could be implemented on *classes* of traffic, where classes are defined in some other way than the simple flows introduced at the start of this chapter. For example, we could use some bits in the IP header to identify classes and allocate a queue and a weight to each class. This is exactly what is proposed as part of the Differentiated Services architecture described in Section 6.5.3.

Note that a router performing WFQ must learn what weights to assign to each queue from somewhere, either by manual configuration or by some sort of signalling from the sources. In the latter case, we are moving toward a reservation-based model. Just assigning a weight to a queue provides a rather weak form of reservation because these weights are only indirectly related to the bandwidth the flow receives. (The bandwidth available to a flow also depends, for example, on how many other flows are sharing the link.) We will see in Section 6.5.2 how WFQ can be used as a component of a reservation-based resource allocation mechanism.

Finally, we observe that this whole discussion of queue management illustrates an important system design principle known as *separating policy and mechanism*. The idea is to view each mechanism as a black box that provides a multifaceted service that can be controlled by a set of knobs. A policy specifies a particular setting of those knobs but does not know (or care) about how the black box is implemented. In this case, the mechanism in question is the queuing discipline, and the policy is a particular setting of which flow gets what level of service (e.g., priority or weight). We discuss some policies that can be used with the WFQ mechanism in Section 6.5.

6.3 TCP CONGESTION CONTROL

This section describes the predominant example of end-to-end congestion control in use today, that implemented by TCP. The essential strategy of TCP is to send packets into the network without a reservation and then to react to observable events that occur. TCP assumes only FIFO queuing in the network's routers, but also works with fair queuing.

TCP congestion control was introduced into the Internet in the late 1980s by Van Jacobson, roughly eight years after the TCP/IP protocol stack had become operational. Immediately preceding this time, the Internet was suffering from congestion collapse—hosts would send their packets into the Internet as fast as the advertised window would allow, congestion would occur at some router (causing packets to be dropped), and the hosts would time out and retransmit their packets, resulting in even more congestion.

Broadly speaking, the idea of TCP congestion control is for each source to determine how much capacity is available in the network, so that it knows how many packets it can safely have in transit. Once a given source has this many packets in transit, it uses the arrival of an ACK as a signal that one of its packets has left the network and that it is therefore safe to insert a new packet into the network without adding to the level of congestion. By using ACKs to pace the transmission of packets, TCP is said to be *self-clocking*. Of course, determining the available capacity in the first place is no easy task. To make matters worse, because other connections come and go, the available bandwidth changes over time, meaning that any given source must be able to adjust the number of packets it has in transit. This section describes the algorithms used by TCP to address these and other problems.

Note that, although we describe the TCP congestion-control mechanisms one at a time, thereby giving the impression that we are talking about three independent mechanisms, it is only when they are taken as a whole that we have TCP congestion control. Also, while we are going to begin here with the variant of TCP congestion control most often referred to as *standard TCP*, we will see that there are actually quite a few variants of TCP congestion control in use today, and researchers continue to explore new approaches to addressing this problem. Some of these new approaches are discussed below.

6.3.1 Additive Increase/Multiplicative Decrease

TCP maintains a new state variable for each connection, called CongestionWindow, which is used by the source to limit how much data it is allowed to have in transit at a given time. The congestion window is congestion control's counterpart to flow control's advertised window.

TCP is modified such that the maximum number of bytes of unacknowledged data allowed is now the minimum of the congestion window and the advertised window. Thus, using the variables defined in Section 5.2.4, TCP's effective window is revised as follows:

$$\text{MaxWindow} = \text{MIN}(\text{CongestionWindow}, \text{AdvertisedWindow})$$

$$\text{EffectiveWindow} = \text{MaxWindow} - (\text{LastByteSent} - \text{LastByteAcked}).$$

That is, MaxWindow replaces AdvertisedWindow in the calculation of EffectiveWindow. Thus, a TCP source is allowed to send no faster than the slowest component—the network or the destination host—can accommodate.

The problem, of course, is how TCP comes to learn an appropriate value for CongestionWindow. Unlike the AdvertisedWindow, which is sent by the receiving side of the connection, there is no one to send a suitable CongestionWindow to the sending side of TCP. The answer is that the TCP source sets the CongestionWindow based on the level of congestion it perceives to exist in the network. This involves decreasing the congestion window when the level of congestion goes up and increasing the congestion window when the level of congestion goes down. Taken together, the mechanism is commonly called *additive increase/multiplicative decrease* (AIMD); the reason for this mouthful of a name will become apparent below.

The key question, then, is how does the source determine that the network is congested and that it should decrease the congestion window? The answer is based on the observation that the main reason packets are not delivered, and a timeout results, is that a packet was dropped due to congestion. It is rare that a packet is dropped because of an error during transmission. Therefore, TCP interprets timeouts as a sign of congestion and reduces the rate at which it is transmitting. Specifically, each time a timeout occurs, the source sets CongestionWindow to half of its previous value. This halving of the CongestionWindow for each timeout corresponds to the "multiplicative decrease" part of AIMD.

Although CongestionWindow is defined in terms of bytes, it is easiest to understand multiplicative decrease if we think in terms of whole packets. For example, suppose the CongestionWindow is currently set to 16 packets. If a loss is detected, CongestionWindow is set to 8. (Normally, a loss is detected when a timeout occurs, but as we see below, TCP has another mechanism to detect dropped packets.) Additional losses cause

CongestionWindow to be reduced to 4, then 2, and finally to 1 packet. CongestionWindow is not allowed to fall below the size of a single packet, or in TCP terminology, the *maximum segment size* (MSS).

A congestion-control strategy that only decreases the window size is obviously too conservative. We also need to be able to increase the congestion window to take advantage of newly available capacity in the network. This is the "additive increase" part of AIMD, and it works as follows. Every time the source successfully sends a CongestionWindow's worth of packets—that is, each packet sent out during the last round-trip time (RTT) has been ACKed—it adds the equivalent of 1 packet to CongestionWindow. This linear increase is illustrated in Figure 6.8. Note that, in practice, TCP does not wait for an entire window's worth of ACKs to add 1 packet's worth to the congestion window, but instead increments CongestionWindow by a little for each ACK that arrives. Specifically, the

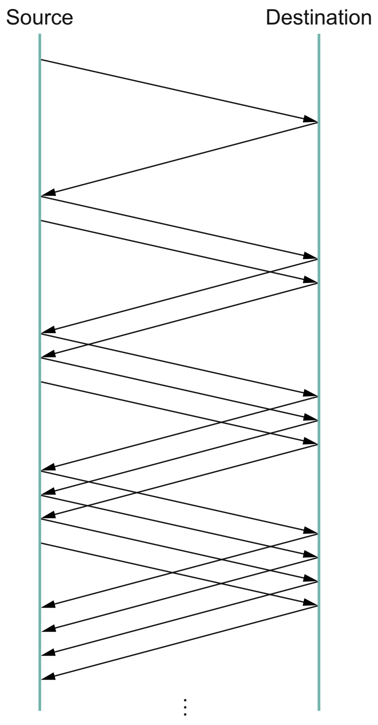


FIGURE 6.8 Packets in transit during additive increase, with one packet being added each RTT.

congestion window is incremented as follows each time an ACK arrives:

$$\text{Increment} = \text{MSS} \times (\text{MSS}/\text{CongestionWindow})$$

$$\text{CongestionWindow}+ = \text{Increment}$$

That is, rather than incrementing CongestionWindow by an entire MSS bytes each RTT, we increment it by a fraction of MSS every time an ACK is received. Assuming that each ACK acknowledges the receipt of MSS bytes, then that fraction is MSS/CongestionWindow.

This pattern of continually increasing and decreasing the congestion window continues throughout the lifetime of the connection. In fact, if you plot the current value of CongestionWindow as a function of time, you get a sawtooth pattern, as illustrated in Figure 6.9. The important concept to understand about AIMD is that the source is willing to reduce its congestion window at a much faster rate than it is willing to increase its congestion window. This is in contrast to an additive increase/additive decrease strategy in which the window would be increased by 1 packet when an ACK arrives and decreased by 1 when a timeout occurs. It has been shown that AIMD is a necessary condition for a congestion-control mechanism to be stable (see the Further Reading section). One intuitive reason to decrease the window aggressively and increase it conservatively is that the consequences of having too large a window are much worse than those of it being too small. For example, when the window is too large, packets that are dropped will be retransmitted, making congestion even worse; thus, it is important to get out of this state quickly.

Finally, since a timeout is an indication of congestion that triggers multiplicative decrease, TCP needs the most accurate timeout mechanism it

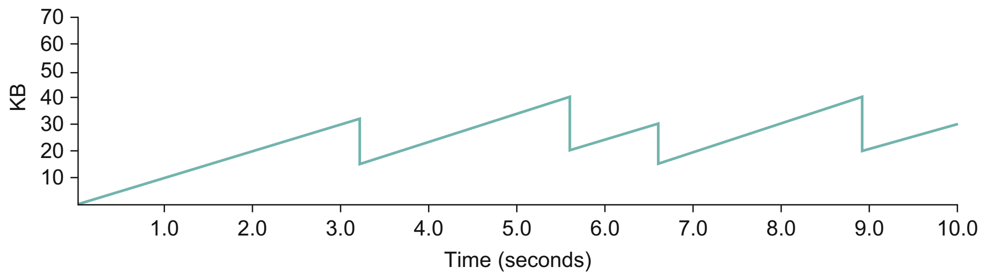


FIGURE 6.9 Typical TCP sawtooth pattern.

When Loss Doesn't Mean Congestion: TCP Over Wireless

There is one situation in which TCP congestion control has a tendency to fail spectacularly. When a link drops packets at a relatively high rate due to bit errors—something that is fairly common on wireless links—TCP misinterprets this as a signal of congestion. Consequently, the TCP sender reduces its rate, which typically has no effect on the rate of bit errors, so the situation can continue until the send window drops to a single packet. At this point, the throughput achieved by TCP will deteriorate to one packet per round-trip time, which may be much less than the appropriate rate for a network that is not actually experiencing congestion.

Given this situation, you may wonder how it is that TCP works at all over wireless networks. Fortunately, there are a number of ways to address the problem. Most commonly, some steps are taken at the link layer to reduce or hide packet losses due to bit errors. For example, 802.11 networks apply forward error correction (FEC) to the transmitted packets so that some number of errors can be corrected by the receiver. Another approach is to do link-layer retransmission, so that even if a packet is corrupted and dropped it eventually gets sent successfully, and the initial loss never becomes apparent to TCP. Each of these approaches has its problems: FEC wastes some bandwidth and will sometimes still fail to correct errors, while retransmission increases both the RTT of the connection and its variance, leading to worse performance.

Another approach used in some situations is to split the TCP connection into wireless and wired segments. There are many variations on this idea, but the basic approach is to treat losses on the wired segment as congestion signals but treat losses on the wireless segment as being caused by bit errors. This sort of technique has been used in satellite networks, where the RTT is so long already that you really don't want to make it any longer. Unlike the link-layer approaches, however, this one is a fundamental change to the end-to-end operation of the protocol; it also means that the forward and reverse paths of the connection have to pass through the same "middlebox" that is doing the splitting of the connection.

Another set of approaches tries to distinguish intelligently between the two difference classes of loss: congestion and bit errors. There are clues that losses are due to congestion, such as increasing RTT and correlation among successive losses. Explicit Congestion Notification (ECN) marking (see Section 6.4.2) can also provide an indication that congestion is imminent, so a subsequent loss is more likely to be congestion related. Clearly, if you can detect the difference between the two types of loss, then TCP doesn't need to reduce its window for bit-error-related losses. Unfortunately, it is hard to make this determination with 100% accuracy, and this issue continues to be an area of active research.

can afford. We already covered TCP's timeout mechanism in Section 5.2.6, so we do not repeat it here. The two main things to remember about that mechanism are that (1) timeouts are set as a function of both the average RTT and the standard deviation in that average, and (2) due to the cost of measuring each transmission with an accurate clock, TCP only samples the round-trip time once per RTT (rather than once per packet) using a coarse-grained (500-ms) clock.

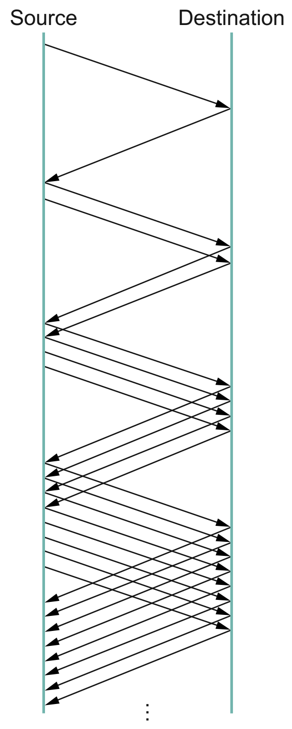
6.3.2 Slow Start

The additive increase mechanism just described is the right approach to use when the source is operating close to the available capacity of the network, but it takes too long to ramp up a connection when it is starting from scratch. TCP therefore provides a second mechanism, ironically called *slow start*,[5] which is used to increase the congestion window rapidly from a cold start. Slow start effectively increases the congestion window exponentially, rather than linearly.

Specifically, the source starts out by setting CongestionWindow to one packet. When the ACK for this packet arrives, TCP adds 1 to CongestionWindow and then sends two packets. Upon receiving the corresponding two ACKs, TCP increments CongestionWindow by 2—one for each ACK—and next sends four packets. The end result is that TCP effectively doubles the number of packets it has in transit every RTT. Figure 6.10 shows the growth in the number of packets in transit during slow start. Compare this to the linear growth of additive increase illustrated in Figure 6.8.

Why any exponential mechanism would be called "slow" is puzzling at first, but it can be explained if put in the proper historical context. We need to compare slow start not against the linear mechanism of the previous subsection, but against the original behavior of TCP. Consider what happens when a connection is established and the source first starts to send packets—that is, when it currently has no packets in transit. If the source sends as many packets as the advertised window allows—which is exactly what TCP did before slow start was developed—then even if there is a fairly large amount of bandwidth available in the network, the

[5]Even though the original paper describing slow start called it "slow-start," the unhyphenated term is more commonly used today, so we omit the hyphen here.



■ **FIGURE 6.10** Packets in transit during slow start.

routers may not be able to consume this burst of packets. It all depends on how much buffer space is available at the routers. Slow start was therefore designed to space packets out so that this burst does not occur. In other words, even though its exponential growth is faster than linear growth, slow start is much "slower" than sending an entire advertised window's worth of data all at once.

There are actually two different situations in which slow start runs. The first is at the very beginning of a connection, at which time the source has no idea how many packets it is going to be able to have in transit at a given time. (Keep in mind that TCP runs over everything from 9600-bps links to 2.4-Gbps links, so there is no way for the source to know the network's capacity.) In this situation, slow start continues to double CongestionWindow each RTT until there is a loss, at which time a timeout causes multiplicative decrease to divide CongestionWindow by 2.

The second situation in which slow start is used is a bit more subtle; it occurs when the connection goes dead while waiting for a timeout to occur. Recall how TCP's sliding window algorithm works—when a packet is lost, the source eventually reaches a point where it has sent as much data as the advertised window allows, and so it blocks while waiting for an ACK that will not arrive. Eventually, a timeout happens, but by this time there are no packets in transit, meaning that the source will receive no ACKs to "clock" the transmission of new packets. The source will instead receive a single cumulative ACK that reopens the entire advertised window, but, as explained above, the source then uses slow start to restart the flow of data rather than dumping a whole window's worth of data on the network all at once.

Although the source is using slow start again, it now knows more information than it did at the beginning of a connection. Specifically, the source has a current (and useful) value of CongestionWindow; this is the value of CongestionWindow that existed prior to the last packet loss, divided by 2 as a result of the loss. We can think of this as the *target* congestion window. Slow start is used to rapidly increase the sending rate up to this value, and then additive increase is used beyond this point. Notice that we have a small bookkeeping problem to take care of, in that we want to remember the target congestion window resulting from multiplicative decrease as well as the *actual* congestion window being used by slow start. To address this problem, TCP introduces a temporary variable to store the target window, typically called CongestionThreshold, that is set equal to the CongestionWindow value that results from multiplicative decrease. The variable CongestionWindow is then reset to one packet, and it is incremented by one packet for every ACK that is received until it reaches CongestionThreshold, at which point it is incremented by one packet per RTT.

In other words, TCP increases the congestion window as defined by the following code fragment:

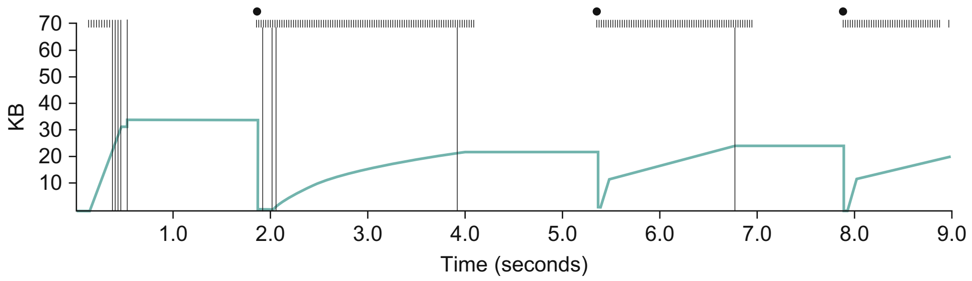
```
{
    u_int    cw = state->CongestionWindow;
    u_int    incr = state->maxseg;

    if (cw > state->CongestionThreshold)
        incr = incr * incr / cw;
        state->CongestionWindow = MIN(cw + incr, TCP_MAXWIN);
    }
```

where state represents the state of a particular TCP connection and TCP_MAXWIN defines an upper bound on how large the congestion window is allowed to grow.

Figure 6.11 traces how TCP's CongestionWindow increases and decreases over time and serves to illustrate the interplay of slow start and additive increase/multiplicative decrease. This trace was taken from an actual TCP connection and shows the current value of CongestionWindow—the colored line—over time.

There are several things to notice about this trace. The first is the rapid increase in the congestion window at the beginning of the connection. This corresponds to the initial slow start phase. The slow start phase continues until several packets are lost at about 0.4 seconds into the connection, at which time CongestionWindow flattens out at about 34 KB. (Why so many packets are lost during slow start is discussed below.) The reason why the congestion window flattens is that there are no ACKs arriving, due to the fact that several packets were lost. In fact, no new packets are sent during this time, as denoted by the lack of hash marks at the top of the graph. A timeout eventually happens at approximately 2 seconds, at which time the congestion window is divided by 2 (i.e., cut from approximately 34 KB to around 17 KB) and CongestionThreshold is set to this value. Slow start then causes CongestionWindow to be reset to one packet and to start ramping up from there.



■ FIGURE 6.11 Behavior of TCP congestion control. Colored line = value of CongestionWindow over time; solid bullets at top of graph = timeouts; hash marks at top of graph = time when each packet is transmitted; vertical bars = time when a packet that was eventually retransmitted was first transmitted.

There is not enough detail in the trace to see exactly what happens when a couple of packets are lost just after 2 seconds, so we jump ahead to the linear increase in the congestion window that occurs between 2 and 4 seconds. This corresponds to additive increase. At about 4 seconds, CongestionWindow flattens out, again due to a lost packet. Now, at about 5.5 seconds:

1. A timeout happens, causing the congestion window to be divided by 2, dropping it from approximately 22 KB to 11 KB, and CongestionThreshold is set to this amount.
2. CongestionWindow is reset to one packet, as the sender enters slow start.
3. Slow start causes CongestionWindow to grow exponentially until it reaches CongestionThreshold.
4. CongestionWindow then grows linearly.

The same pattern is repeated at around 8 seconds when another timeout occurs.

We now return to the question of why so many packets are lost during the initial slow start period. At this point, TCP is attempting to learn how much bandwidth is available on the network. This is a very difficult task. If the source is not aggressive at this stage—for example, if it only increases the congestion window linearly—then it takes a long time for it to discover how much bandwidth is available. This can have a dramatic impact on the throughput achieved for this connection. On the other hand, if the source is aggressive at this stage, as TCP is during exponential growth, then the source runs the risk of having half a window's worth of packets dropped by the network.

To see what can happen during exponential growth, consider the situation in which the source was just able to successfully send 16 packets through the network, causing it to double its congestion window to 32. Suppose, however, that the network happens to have just enough capacity to support 16 packets from this source. The likely result is that 16 of the 32 packets sent under the new congestion window will be dropped by the network; actually, this is the worst-case outcome, since some of the packets will be buffered in some router. This problem will become increasingly severe as the delay $\times$ bandwidth product of networks increases. For example, a delay $\times$ bandwidth product of 500 KB means that each connection has the potential to lose up to 500 KB of data at the beginning of each connection. Of course, this assumes that both the source and the destination implement the "big windows" extension.

Some protocol designers have proposed alternatives to slow start, whereby the source tries to estimate the available bandwidth by more sophisticated means. A recent example is the *quick-start* mechanism undergoing standardization at the IETF. The basic idea is that a TCP sender can ask for an initial sending rate greater than slow start would allow by putting a requested rate in its SYN packet as an IP option. Routers along the path can examine the option, evaluate the current level of congestion on the outgoing link for this flow, and decide if that rate is acceptable, if a lower rate would be acceptable, or if standard slow start should be used. By the time the SYN reaches the receiver, it will contain either a rate that was acceptable to all routers on the path or an indication that one or more routers on the path could not support the quick-start request. In the former case, the TCP sender uses that rate to begin transmission; in the latter case, it falls back to standard slow start. If TCP is allowed to start off sending at a higher rate, a session could more quickly reach the point of filling the pipe, rather than taking many round-trip times to do so.

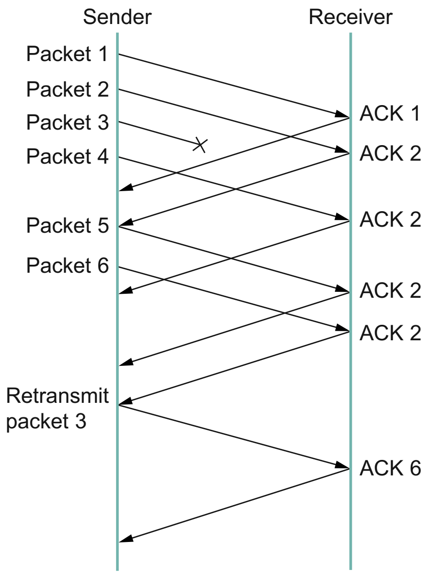
Clearly one of the challenges to this sort of enhancement to TCP is that it requires substantially more cooperation from the routers than standard TCP does. If a single router in the path does not support quick-start, then the system reverts to standard slow start. Thus, it could be a long time before these types of enhancements could make it into the Internet; for now, they are more likely to be used in controlled network environments (e.g., research networks).

6.3.3 Fast Retransmit and Fast Recovery

The mechanisms described so far were part of the original proposal to add congestion control to TCP. It was soon discovered, however, that the coarse-grained implementation of TCP timeouts led to long periods of time during which the connection went dead while waiting for a timer to expire. Because of this, a new mechanism called *fast retransmit* was added to TCP. Fast retransmit is a heuristic that sometimes triggers the retransmission of a dropped packet sooner than the regular timeout mechanism. The fast retransmit mechanism does not replace regular timeouts; it just enhances that facility.

The idea of fast retransmit is straightforward. Every time a data packet arrives at the receiving side, the receiver responds with an acknowledgment, even if this sequence number has already been acknowledged. Thus, when a packet arrives out of order—when TCP cannot yet acknowledge the data the packet contains because earlier data has not yet arrived—TCP resends the same acknowledgment it sent the last time. This second transmission of the same acknowledgment is called a *duplicate ACK*. When the sending side sees a duplicate ACK, it knows that the other side must have received a packet out of order, which suggests that an earlier packet might have been lost. Since it is also possible that the earlier packet has only been delayed rather than lost, the sender waits until it sees some number of duplicate ACKs and then retransmits the missing packet. In practice, TCP waits until it has seen three duplicate ACKs before retransmitting the packet.

Figure 6.12 illustrates how duplicate ACKs lead to a fast retransmit. In this example, the destination receives packets 1 and 2, but packet 3 is lost in the network. Thus, the destination will send a duplicate ACK for packet 2 when packet 4 arrives, again when packet 5 arrives, and so on. (To simplify this example, we think in terms of packets 1, 2, 3, and so on, rather

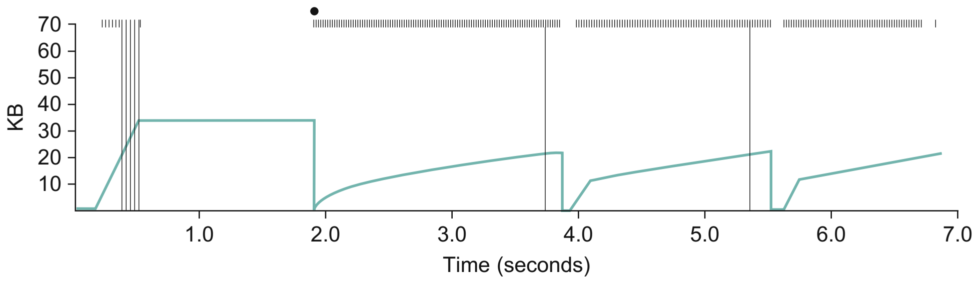


■ **FIGURE 6.12** Fast retransmit based on duplicate ACKs.

than worrying about the sequence numbers for each byte.) When the sender sees the third duplicate ACK for packet 2—the one sent because the receiver had gotten packet 6—it retransmits packet 3. Note that when the retransmitted copy of packet 3 arrives at the destination, the receiver then sends a cumulative ACK for everything up to and including packet 6 back to the source.

Figure 6.13 illustrates the behavior of a version of TCP with the fast retransmit mechanism. It is interesting to compare this trace with that given in Figure 6.11, where fast retransmit was not implemented—the long periods during which the congestion window stays flat and no packets are sent has been eliminated. In general, this technique is able to eliminate about half of the coarse-grained timeouts on a typical TCP connection, resulting in roughly a 20% improvement in the throughput over what could otherwise have been achieved. Notice, however, that the fast retransmit strategy does not eliminate all coarse-grained timeouts. This is because for a small window size there will not be enough packets in transit to cause enough duplicate ACKs to be delivered. Given enough lost packets—for example, as happens during the initial slow start phase—the sliding window algorithm eventually blocks the sender until a timeout occurs. Given the current 64-KB maximum advertised window size, TCP's fast retransmit mechanism is able to detect up to three dropped packets per window in practice.

Finally, there is one last improvement we can make. When the fast retransmit mechanism signals congestion, rather than drop the congestion window all the way back to one packet and run slow start, it is



■ FIGURE 6.13 Trace of TCP with fast retransmit. Colored line = CongestionWindow; solid bullet = timeout; hash marks = time when each packet is transmitted; vertical bars = time when a packet that was eventually retransmitted was first transmitted.

possible to use the ACKs that are still in the pipe to clock the sending of packets. This mechanism, which is called *fast recovery*, effectively removes the slow start phase that happens between when fast retransmit detects a lost packet and additive increase begins. For example, fast recovery avoids the slow start period between 3.8 and 4 seconds in Figure 6.13 and instead simply cuts the congestion window in half (from 22 KB to 11 KB) and resumes additive increase. In other words, slow start is only used at the beginning of a connection and whenever a coarse-grained timeout occurs. At all other times, the congestion window is following a pure additive increase/multiplicative decrease pattern.

A Faster TCP?

Many times in the last two decades the argument over how fast TCP can be made to run has reared its head. First there was the claim that TCP was too complex to run fast in host software as networks headed toward the gigabit range. This claim was repeatedly disproved. More recently however, an important theoretical result has shown that there are limits to how well standard TCP can perform in very high bandwidth-delay environments. An analysis of the congestion-control behavior of TCP has shown that, in the steady state, TCP's throughput is approximately

$$Rate = \left(\frac{1.2 \times MSS}{RTT \times \sqrt{\rho}}\right)$$

In a network with an RTT of 100 ms and 10-Gbps links, it follows that a single TCP connection will only be able to achieve a throughput close to link speed if the loss rate is below one per 5 billion packets—equivalent to one congestion event every 100 minutes. Even very rare packet losses due to bit errors on the fiber will typically produce a considerably higher loss rate than this, making it impossible to fill the pipe with a single TCP connection.

A number of proposals to improve on TCP's behavior in networks with very high bandwidth delay products have been put forward, and they range from the incremental to the dramatic. Observing the dependency on MSS, one simple change that has been proposed is to increase the packet size. Unfortunately, increasing packet sizes also increases the chance that a given packet will suffer from a bit error, so at some point increasing the MSS alone may not be sufficient. Other proposals that have been advanced at the IETF and elsewhere make changes to the way TCP avoids congestion, in an attempt to make TCP better able to use bandwidth that is available. The challenges here are to be fair to standard TCP implementations and also to avoid the congestion collapse issues that led to the current behavior of TCP.

The HighSpeed TCP proposal, now an experimental RFC, makes TCP more aggressive only when it is clearly operating in a very high bandwidth-delay product environment and not competing with a lot of other traffic. In essence, when the congestion window gets very large, HighSpeed TCP starts to increase CongestionWindow by a larger amount that standard TCP. In the normal environment where CongestionWindow is relatively small (about 40 $\times$ MSS), HighSpeed TCP is indistinguishable from standard TCP. Many other proposals have been made in this vein, some of which are listed in the Further Reading section. Notably, the default TCP behavior in the Linux operating system is now based on a TCP variant called *CUBIC*, which also expands the congestion window aggressively in high bandwidth-delay product regimes, while maintaining compatibility with older TCP variants in more bandwidth-constrained environments.

The Quick-Start proposal, which changes the start-up behavior of TCP, was mentioned above. Since it can enable a TCP connection to ramp up its sending rate more quickly, its effect on TCP performance is most noticeable when connections are short, or when an application periodically stops sending data and TCP would otherwise return to slow start.

Yet another proposal, FAST TCP, takes an approach similar to TCP Vegas described in the next section. The basic idea is to anticipate the onset of congestion and avoid it, thereby not taking the performance hit associated with decreasing the congestion window.

Several proposals that involve more dramatic changes to TCP or even replace it with a new protocol have been developed. These have considerable potential to fill the pipe quickly and fairly in high bandwidth-delay environments, but they also face higher deployment challenges. We refer the reader to the end of this chapter for references to ongoing work in this area.

6.4 CONGESTION-AVOIDANCE MECHANISMS

It is important to understand that TCP's strategy is to control congestion once it happens, as opposed to trying to avoid congestion in the first place. In fact, TCP repeatedly increases the load it imposes on the network in an effort to find the point at which congestion occurs, and then it backs off from this point. Said another way, TCP *needs* to create losses to find the available bandwidth of the connection. An appealing alternative, but one that has not yet been widely adopted, is to predict when congestion is about to happen and then to reduce the rate at which hosts send data just before packets start being discarded. We call such a strategy *congestion avoidance*, to distinguish it from *congestion control*.