MIT 6.02 **DRAFT** *Lecture Notes*
Last update: April 29, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 19
Reliable Data Transport Protocols

Packets in a *best-effort network* lead a rough life. They can be lost for any number of reasons, including queue overflows at switches because of congestion, repeated collisions over shared media, routing failures, and uncorrectable bit errors. In addition, packets can arrive out-of-order at the destination because different packets sent in sequence take different paths or because some switch en route reorders packets for some reason. They usually experience variable delays, especially whenever they encounter a queue. In some cases, the underlying network may even duplicate packets.

Many applications, such as Web page downloads, file transfers, and interactive terminal sessions would like a **reliable, in-order** stream of data, receiving exactly one copy of each byte in the same order in which it was sent. A **reliable transport protocol** does the job of hiding the vagaries of a best-effort network—packet losses, reordered packets, and duplicate packets—from the application, and provides it the abstraction of a reliable packet stream. We will develop protocols that also provide in-order delivery.

A large number of protocols have been developed that various applications use, and there are several ways to provide a reliable, in-order abstraction. This chapter will not discuss them all, but will instead discuss two protocols in some detail. The first protocol, called **stop-and-wait**, will solve the problem in perhaps the simplest possible way that works, but do so somewhat inefficiently. The second protocol will augment the first one with a **sliding window** to significantly improve performance.

All reliable transport protocols use the same powerful ideas: *redundancy to cope with packet losses* and *receiver buffering to cope with reordering*, and most use *adaptive timers*. The tricky part is figuring out exactly how to apply redundancy in the form of packet *retransmissions*, in working out exactly when retransmissions should be done, and in achieving good performance. This chapter will study these issues, and discuss ways in which a reliable transport protocol can achieve high throughput.

■ 19.1 The Problem

The problem we're going to solve is relatively easy to state. A sender application wants to send a stream of packets to a receiver application over a best-effort network, which

can drop packets arbitrarily, reorder them arbitrarily, delay them arbitrarily, and possibly even duplicate packets. The receiver wants the packets in exactly the same order in which the sender sent them, and wants exactly one copy of each packet.[1] Our goal is to devise mechanisms at the sending and receiving nodes to achieve what the receiver wants. These mechanisms involve rules between the sender and receiver, which constitute the protocol. In addition to correctness, we will be interested in calculating the throughput of our protocols, and in coming up with ways to maximize it.

All mechanisms to recover from losses, whether they are caused by packet drops or corrupted bits, employ *redundancy*. We have already studied *error-correcting codes* such as linear block codes and convolutional codes to mitigate the effect of bit errors. In principle, one could apply similar coding techniques over packets (rather than over bits) to recover from packet losses (as opposed to bit corruption). We are, however, interested not just in a scheme to reduce the effective packet loss rate, but to eliminate their effects altogether, and recover all lost packets. We are also able to rely on *feedback* from the receiver that can help the sender determine what to send at any point in time, in order to achieve that goal. Therefore, we will focus on carefully using *retransmissions* to recover from packet losses; one may combine retransmissions and error-correcting codes to produce a protocol that can further improve throughput under certain conditions. In general, experience has shown that if packet losses are not persistent and occur in bursts, and if latencies are not excessively long (i.e., not multiple seconds long), retransmissions by themselves are enough to recover from losses and achieve good throughput. Most practical reliable data transport protocols running over Internet paths today use only retransmissions on packets (individual links usually use the error correction methods, such as the ones we studied earlier, and may also augment them with a limited number of retransmissions to reduce the link-level packet loss rate.

We will develop the key ideas for two kinds of reliable data transport protocols: **stop-and-wait** and **sliding window with a fixed window size**. We will use the word "sender" to refer to the sending side of the transport protocol and the word "receiver" to refer to the receiving side. We will use "sender application" and "receiver application" to refer to the processes (applications) that would like to send and receive data in a reliable, in-order manner.

■ 19.2 Stop-and-Wait Protocol

The high-level idea in this protocol is simple. The sender attaches a *transport-layer header* to every data packet, which includes a *unique identifier* for the data packet (the transport-layer header is distinct from the *network-layer* packet header that contains the destination address, hop limit, and header checksum discussed in Chapters 17 and 18). Ideally, this unique identifier will never be reused for two different packets on the same stream.[2] The

[1]The reason for the "exactly one copy" requirement is that the mechanism used to solve the problem will end up retransmitting packets, so duplicates may occur that need to be filtered out. In some networks, it is possible that some links may end up duplicating packets because of mechanisms they employ to improve the packet delivery probability or bit-error rate over the link.

[2]In an ideal implementation, such reuse will never occur. In practice, however, a transport protocol may use a sequence number field whose width is not large enough and sequence numbers may wrap-around. In this case, it is important to ensure that two distinct unacknowledged data packets never have the same

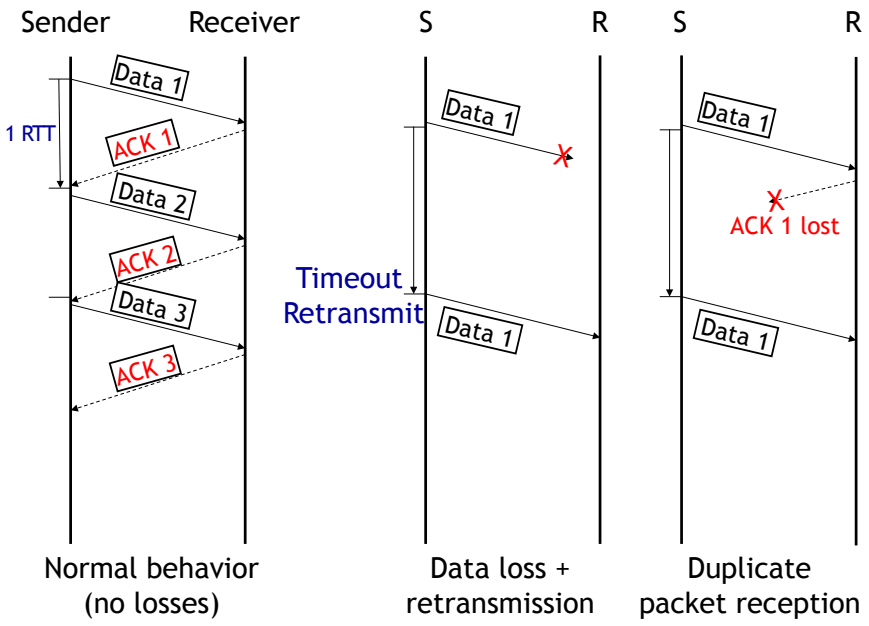


Figure 19-1: The stop-and-wait protocol. Each picture has a sender timeline and a receiver timeline. Time starts at the top of each vertical line and increases moving downward. The picture on the left shows what happens when there are no losses; the middle shows what happens on a data packet loss; and the right shows how duplicate packets may arrive at the receiver because of an ACK loss.

receiver, upon receiving the data packet with identifier k, will send an *acknowledgment* (ACK) to the sender; the header of this ACK contains k, so the receiver communicates "I got data packet k" to the sender. Both data packets and ACKs may get lost in the network.

In the stop-and-wait protocol, the sender sends the next data packet on the stream if, and only if, it receives an ACK for k. If it does not get an ACK within some period of time, called the *timeout*, the sender *retransmits* data packet k.

The receiver's job is to deliver each data packet it receives to the receiver application. Figure 19-1 shows the basic operation of the protocol when packets are not lost (left) and when data packets are lost (right).

Three properties of this protocol bear some discussion:

1. how to pick unique identifiers,

2. why this protocol may deliver duplicate data packets to the receiver application, and how the receiver can prevent that from occurring, and

3. how to pick the timeout.

We discuss each of these in turn below.

sequence number.

19.2.1 Selecting Unique Identifiers: Sequence Numbers

The sender may pick any unique identifier for a data packet. In most transport protocols, a convenient and effective choice of unique identifier is to use an *incrementing sequence number*. The simplest way to achieve this goal is for the sender and receiver to agree on the initial value of the identifier (which for our purposes will be taken to be 1), and then increment the identifier by 1 for each subsequent *new* data packet sent. Thus, the data packet sent after the ACK for k is received by the sender will have identifier $k + 1$. These incrementing identifiers are called **sequence numbers**.

In practice, transport protocols like TCP (Transmission Control Protocol), the standard Internet protocol for reliable data delivery, devote considerable effort to picking a good initial sequence number to avoid overlaps with previous instantiations of reliable streams between the same communicating processes. We won't worry about these complications in this chapter, except to note that establishing and properly terminating these streams (aka connections) reliably is a non-trivial problem. TCP also uses a sequence number that identifies the *starting byte offset* of the packet in the stream, to handle variable packet sizes.

19.2.2 Semantics of this Stop-and-Wait Protocol

It is easy to see that the stop-and-wait protocol achieves reliable data delivery as long as each of the links along the path have a non-zero packet delivery probability. However, it does not achieve *exactly once* semantics; its semantics are *at least once*—i.e., each packet will be delivered to the receiver application either once or *more than once*.

One reason is that the network could drop ACKs, as shown in Figure 19-1 (right). A data packet may have reached the receiver, but the ACK doesn't reach the sender, and the sender will then timeout and retransmit the data packet. The receiver will get multiple copies of the data packet, and deliver both to the receiver application. Another reason is that the sender might have timed out, but the original data packet may not actually have been lost. Such a retransmission is called a *spurious retransmission*, and is a waste of bandwidth. The sender may strive to reduce the number of spurious retransmissions, but it is impossible to eliminate them in general.

Preventing duplicates: The solution to the problem of duplicate data packets arriving at the receiver is for the receiver to keep track of the last *in-sequence* data packet it has delivered to the application. At the receiver, let us maintain the sequence number of the last in-sequence data packet in the variable `rcv_seqnum`. If a data packet with sequence number less than or equal to `rcv_seqnum` arrives, then the receiver sends an ACK for the packet and discards it. Note that the only way a data packet with sequence number *smaller* than `rcv_seqnum` can arrive is if there were reordering in the network and the receiver gets an old data packet; for such packets, the receiver can safely not send an ACK because it knows that the sender knows about the receipt of the packet and has sent subsequent packets. This method prevents duplicate packets from being delivered to the receiving application.

If a data packet with sequence number `rcv_seqnum` + 1 arrives, then the receiver sends an ACK to the sender, delivers the data packet to the application, and increments `rcv_seqnum`. Note that a data packet with sequence number greater than `rcv_seqnum`

+ 1 should never arrive in this stop-and-wait protocol because that would imply that the sender got an ACK for `rcv_seqnum` + 1, but such an ACK would have been sent only if the receiver got the corresponding data packet. So, if such a data packet were to arrive, then *there must be a bug in the implementation* of either the sender or the receiver in this stop-and-wait protocol.

With this modification, the stop-and-wait protocol guarantees exactly-once delivery to the application.[3]

19.2.3 Setting Timeouts

The final design issue that we need to nail down in our stop-and-wait protocol is setting the value of the timeout. How soon after the transmission of a packet should the sender conclude that the data packet (or the ACK) was lost, and go ahead and retransmit? One approach might be to use some constant, but then the question is what it should be set to. Too small, and the sender may end up retransmitting data packets before giving enough time for the ACK for the original transmission to arrive, wasting network bandwidth. Too large, and one ends up wasting network bandwidth and simply idling before retransmitting.

The natural time-scale in the protocol is the time between the transmission of a data packet and the arrival of the ACK for the packet. This time is called the **round-trip time**, or **RTT**, and plays a crucial role in all reliable transport protocols. A good value of the timeout must clearly depend on the RTT; it makes no sense to use a timeout that is not bigger than the mean RTT (and in fact, it must be quite a bit bigger than the average, as we'll see).

The other reason the RTT is an important concept is that the throughput (in packets per second) achieved by the stop-and-wait protocol is inversely proportional to the RTT (see Section 19.4). In fact, the throughput of the sliding window protocol also depends on the RTT, as we will see.

The next section describes a procedure to estimate the RTT and set sender timeouts. This technique is general and applies to a variety of protocols, including both stop-and-wait and sliding window.

19.3 Adaptive RTT Estimation and Setting Timeouts

The RTT experienced by packets is variable because the delays in a best-effort network are variable. An example is shown in Figure 19-2, which shows the RTT of an Internet path between two hosts (blue) as a and the packet loss rate (red), both as a function of the time-of-day. The "rtt median-filtered" curve is the median RTT computed over a recent window of samples, and you can see that even that varies quite a bit. Picking a timeout equal to simply the mean or median RTT is not a good idea because there will be many RTT samples that are larger than the mean (or median), and we don't want to timeout prematurely and send *spurious retransmissions*.

[3]We are assuming here that the sender and receiver nodes and processes don't crash and restart; handling those cases make "exactly once" semantics considerably harder than described here and require stable storage that persists across crashes.

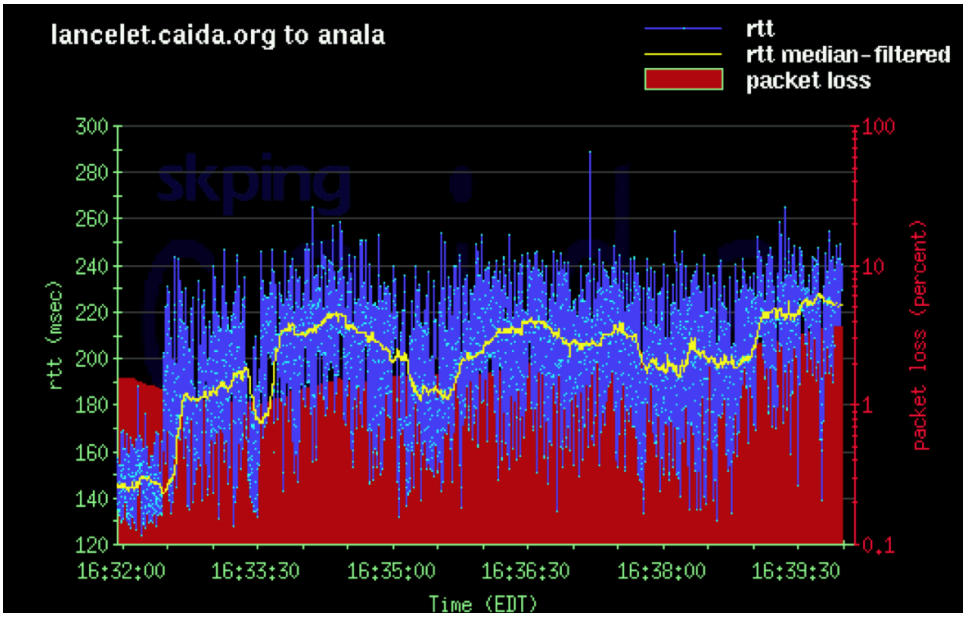


Figure 19-2: RTT variations are pronounced in many networks.

A good solution to the problem of picking the timeout value uses two tools we have seen earlier in the course: *probability distributions* (in our case, of the RTT estimates) and *a simple filter design*.

Suppose we are interested in estimating a good timeout *post facto*: i.e., suppose we run the protocol and collect a sequence of RTT samples, how would one use these values to pick a good timeout? We can take all the RTT samples and plot them as a probability distribution, and then see how any given timeout value will have performed in terms of the probability of a spurious retransmission. If the timeout value is T, then this probability may be estimated as the area under the curve to the right of "T" in the picture on the left of Figure 19-3, which shows the histogram of RTT samples. Equivalently, if we look at the cumulative distribution function of the RTT samples (the picture on the right of Figure 19-3, the probability of a spurious retransmission may be assumed to be the value of the y-axis corresponding to a value of T on the x-axis.

Real-world distributions of RTT are not actually Gaussian, but an interesting property of all distributions is that if you pick a threshold that is a sufficient number of standard deviations greater than the mean, the tail probability of a sample exceeding that threshold can be made arbitrarily small. (For the mathematically inclined, a useful result for arbitrary distributions is Chebyshev's inequality, which you might have seen in other courses already (or soon will): $P(|X - \mu| \geq k\sigma) \leq 1/k^2$, where μ is the mean and σ the standard deviation of the distribution. For Gaussians, the tail probability falls off *much faster* than $1/k^2$; for instance, when $k = 2$, the Gaussian tail probability is only about 0.05 and when $k = 3$, the tail probability is about 0.003.)

The protocol designer can use past RTT samples to determine an RTT cut-off so that only a small fraction f of the samples are larger. The choice of f depends on what spurious retransmission rate one is willing to tolerate, and depending on the protocol, the cost of such an action might be small or large. Empirically, Internet transport protocols tend to

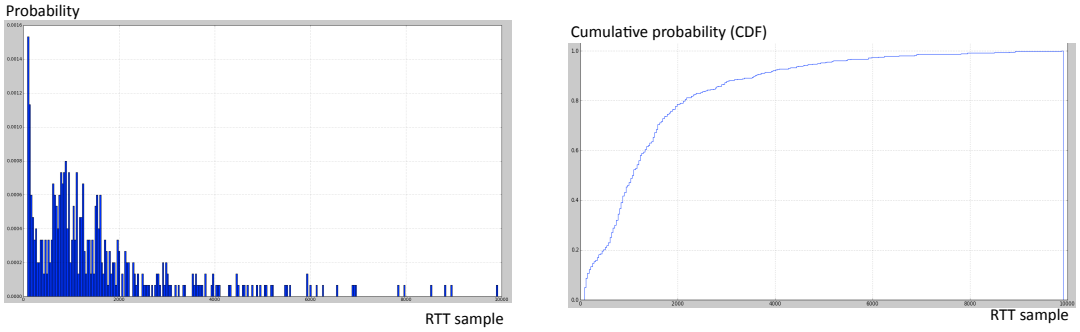


Figure 19-3: RTT variations on a wide-area cellular wireless network (Verizon Wireless's 3G CDMA Rev A service) across both idle periods and when data transfers are in progress, showing extremely high RTT values and high variability. The x-axis in both pictures is the RTT in milliseconds. The picture on the left shows the histogram (each bin plots the total probability of the RTT value falling within that bin), while the picture on the right is the cumulative distribution function (CDF). These delays suggest a poor network design with excessively long queues that do nothing more than cause delays to be very large. Of course, it means that the timeout method must adapt to these variations to the extent possible. (Data collected in November 2009 in Cambridge, MA and Belmont, MA.)

be conservative and use $k = 4$, in an attempt to make the likelihood of a spurious retransmission very small, because it turns out that the cost of doing one on an already congested network is rather large.

Notice that this approach is similar to something we did earlier in the course when we estimated the bit-error rate from the probability density function of voltage samples, where values above (or below) a threshold would correspond to a bit error. In our case, the "error" is a spurious retransmission.

So far, we have discussed how to set the timeout in a post-facto way, assuming we knew what the RTT samples were. We now need to talk about two important issues to complete the story:

1. How can the sender obtain RTT estimates?

2. How should the sender estimate the mean and deviation and pick a suitable timeout?

Obtaining RTT estimates. If the sender keeps track of when it sent each data packet, then it can obtain a sample of the RTT when it gets an ACK for the packet. The RTT sample is simply the difference in time between when the ACK arrived and when the data packet was sent. An elegant way to keep track of this information in a protocol is for the sender to include the current time in the header of each data packet that it sends in a "timestamp" field. The receiver then simply echoes this time in its ACK. When the sender gets an ACK, it just has to consult the clock for the current time, and subtract the echoed timestamp to obtain an RTT sample.

Calculating the timeout. As explained above, our plan is to pick a timeout that uses both the average and deviation of the RTT sample distribution. The sender must take two factors into account while estimating these values:

1. It must not get swayed by infrequent samples that are either too large or too small. That is, it must employ some sort of "smoothing".

2. It must weigh more recent estimates higher than old ones, because network conditions could have changed over multiple RTTs.

Thus, what we want is a way to track changing conditions, while at the same time not being swayed by sudden changes that don't persist.

Let's look at the first requirement. Given a sequence of RTT samples, $r_0, r_1, r_2, \ldots, r_n$, we want a sequence of smoothed outputs, $s_0, s_1, s_2, \ldots, s_n$ that avoids being swayed by sudden changes that don't persist. This problem sounds like a *filtering problem*, which we have studied earlier. The difference, of course, is that we aren't applying it to frequency division multiplexing, but the underlying problem is what a *low-pass filter* (LPF) does.

A simple LPF that provides what we need has the following form:

$$s_n = \alpha r_n + (1 - \alpha) s_{n-1}, \tag{19.1}$$

where $0 < \alpha < 1$.

To see why Eq. (19.1) is a low-pass filter, let's write down the frequency response, $H(\Omega)$. We know that if $r_n = e^{j\Omega n}$, then $s_n = H(\Omega)e^{j\Omega n}$. Letting $z = e^{j\Omega}$, we can rewrite Eq. (19.1) as

$$H(\Omega)z^n = \alpha z^n + (1 - \alpha)H(\Omega)z^{(n-1)},$$

which then gives us

$$H(\Omega) = \frac{\alpha z}{z - (1 - \alpha)}, \tag{19.2}$$

This filter has a single real pole, and is stable when $0 < \alpha < 1$. The peak of the frequency response is at $\Omega = 0$.

What does α do? Clearly, large values of α mean that we are weighing the current sample much more than the existing s estimate, so there's little memory in the system, and we're therefore letting higher frequencies through more than a smaller value of α. What α does is determine the rate at which the frequency response of the LPF tapers: small α makes lets fewer high-frequency components through, but at the same time, it takes more time to react to persistent changes in the RTT of the network. As α increases, we let more higher frequencies through. Figure 19-4 illustrates this point.

Figure 19-5 shows how different values of α react to a sudden non-persistent change in the RTT, while Figure 19-6 shows how they react to a sudden, but persistent, change in the RTT. Empirically, on networks prone to RTT variations due to congestion, researchers have found that α between 0.1 and 0.25 works well. In practice, TCP uses $\alpha = 1/8$.

The specific form of Equation 19.1 is very popular in many networks and computer systems, and has a special name: **exponential weighted moving average (EWMA)**. It is a "moving average" because the LPF produces a smoothed estimate of the average behavior. It is "exponentially weighted" because the weight given to older samples decays

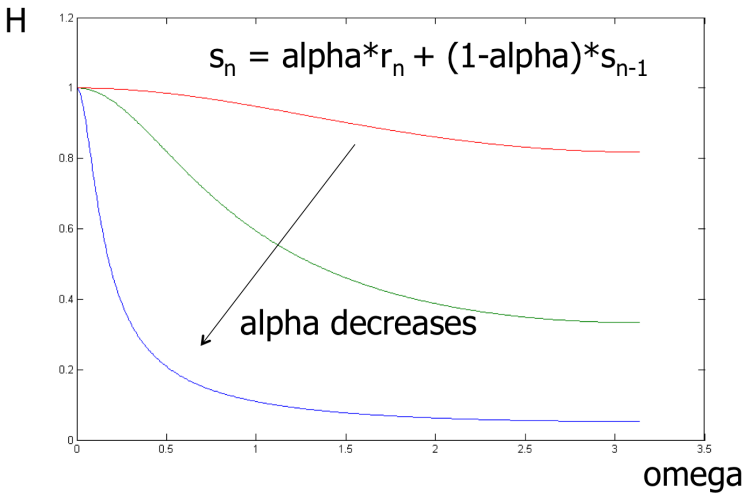


Figure 19-4: Frequency response of the exponential weighted moving average low-pass filter. As α decreases, the low-pass filter becomes even more pronounced. The graph shows the response for $\alpha = 0.9, 0.5, 0.1$, going from top to bottom.

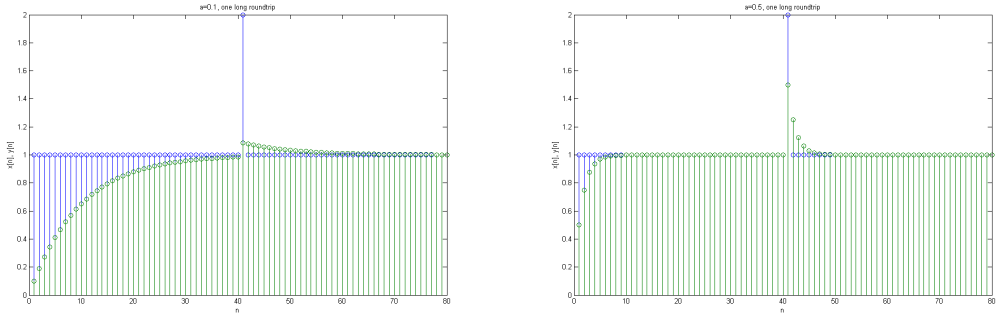


Figure 19-5: Reaction of the exponential weighted moving average filter to a non-persistent spike in the RTT (the spike is double the other samples). The smaller α (0.1, shown on the left) doesn't get swayed by it, whereas the bigger value (0.5, right) does. The output of the filter is shown in green, the input in blue.

geometrically: one can rewrite Eq. 19.1 as

$$s_n = \alpha r_n + \alpha(1-\alpha) r_{n-1} + \alpha(1-\alpha)^2 r_{n-2} + \ldots + \alpha(1-\alpha)^{n-1} r_1 + (1-\alpha)^n r_0, \qquad (19.3)$$

observing that each successive older sample's weight is a factor of $(1-\alpha)$ "less important" than the previous one's.

With this approach, one can compute the smoothed RTT estimate, `srtt`, quite easily using the pseudocode shown below, which runs each time an ACK arrives with an RTT estimate, r.

$$\texttt{srtt} \leftarrow \alpha r + (1-\alpha)\texttt{srtt}$$

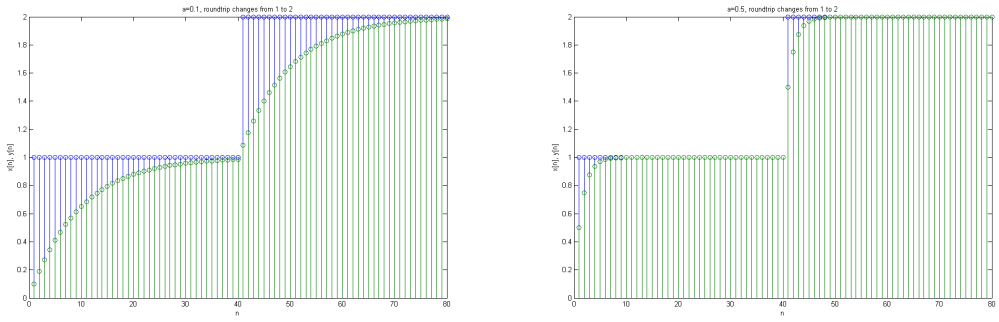


Figure 19-6: Reaction of the exponential weighted moving average filter to a persistent change (doubling) in the RTT. The smaller α (0.1, shown on the left) takes much longer to track the change, whereas the bigger value (0.5, right) responds much quicker. The output of the filter is shown in green, the input in blue.

What about the deviation? Ideally, we want the sample standard deviation, but it turns out to be a bit easier to compute the mean *linear deviation instead*.[4] The following elegant method performs this task:

$$\texttt{dev} \leftarrow |r - srtt|$$
$$\texttt{rttdev} \leftarrow \beta \cdot \texttt{dev} + (1 - \beta) \cdot \texttt{rttdev}$$

Here, $0 < \beta < 1$, and we apply an EWMA to estimate the linear deviation as well. TCP uses $\beta = 0.25$; again, values between 0.1 and 0.25 have been found to work well.

Finally, the timeout is calculated very easily as follows:

$$\texttt{timeout} \leftarrow \texttt{srtt} + 4 \cdot \texttt{rttdev}$$

This procedure to calculate the timeout runs every time an ACK arrives. It does a great deal of useful work essential to the correct functioning of any reliable transport protocol, and it can be implemented in less than 10 lines of code in most programming languages! The reader should note that this procedure does not depend on whether the transport protocol is stop-and-wait or sliding window; the same method works for both.

Exponential back-off of the timeout. When a timeout occurs and the sender retransmits a data packet, it might be lost again (or its ACK might be lost). In that case, it is possible (in networks where congestion is the main reason for packet loss) that the network is heavily congested. Rather than using the same timeout value and retransmitting, it would be prudent to take a leaf from the exponential back-off idea we studied earlier with contention MAC protocols and double the timeout value. Eventually, when the retransmitted data packet is acknowledged, the sender can revert to the timeout value calculated from the mean RTT and its linear deviation. Most reliable transport protocols use an adaptive timer with such an exponential back-off mechanism.

[4]The mean linear deviation is always at least as big as the sample standard deviation, so picking a timeout equal to the mean plus k times the linear deviation has a tail probability no larger than picking a timeout equal to the mean plus k times the sample standard deviation.

19.4 Throughput of Stop-and-Wait

We now show how to calculate the throughput of the stop-and-wait protocol. Clearly, the maximum throughput occurs when there are no packet losses. The sender sends one packet every RTT, so the maximum throughput is exactly that.

We can also calculate the throughput of stop-and-wait when the network has a packet loss rate of ℓ. For convenience, we will treat ℓ as the *bi-directional* loss rate; i.e., the probability of any given packet *or* its ACK getting lost is ℓ. We will assume that the packet loss distribution is independent and identically distributed. What is the throughput of the stop-and-wait protocol in this case?

The answer clearly depends on the timeout that's used. Let's assume that the retransmission timeout is RTO, which we will assume to be a constant for simplicity (i.e., it is the same throughout the connection and the sender doesn't use any exponential back-off). These assumptions mean that the calculation below may be viewed as a (good) upper bound on the throughput.

Let T denote the expected time taken to send a data packet and get an ACK for it. Observe that with probability $1 - \ell$, the data packet reaches the receiver and its ACK reaches the sender. On the other hand, with probability ℓ, the sender needs to time out and retransmit a data packet. We can use this property to write an expression for T:

$$T = (1 - \ell) \cdot \text{RTT} + \ell(\text{RTO} + T), \tag{19.4}$$

because once the sender times out, the expected time to send a data packet and get an ACK is exactly T, the number we want to calculate. Solving Equation (19.4), we find that $T = \text{RTT} + \frac{\ell}{1-\ell} \cdot \text{RTO}$.

The expected throughput of the protocol is then equal to $1/T$ packets per second.[5]

The good thing about the stop-and-wait protocol is that it is very simple, and should be used under two circumstances: first, when throughput isn't a concern and one wants good reliability, and second, when the network path has a small RTT such that sending one data packet every RTT is enough to saturate the bandwidth of the link or path between sender and receiver.

On the other hand, a typical Internet path between Boston and San Francisco might have an RTT of about 100 milliseconds. If the network path has a bit rate of 1 megabit/s, and we use a data packet size of 10,000 bits, then the maximum throughput of stop-and-wait would be only 10% of the possible rate. And in the face of packet loss, it would be much lower than that.

The next section describes a protocol that provides considerably higher throughput. It builds on all the mechanisms used in the stop-and-wait protocol.

[5]The careful reader or purist may note that we have only calculated T, the *expected time* between the transmission of a data packet and the receipt of an ACK for it. We have then assumed that the expected value of the reciprocal of X, which is a random variable whose expected value is T, is equal to $1/T$. In general, however, $1/E[X]$ is not equal to $E[1/X]$. But the formula for the expected throughput we have written does in fact hold. Intuitively, to see why, define $Y_n = X_1 + X_2 + \ldots X_n$. As $n \to \infty$, one can show using the Chebyshev inequality that the probability that $|Y_n - nT| > \delta n$ goes to 0 or any positive δ. That is, when viewed over a long period of time, the random variable X looks like a constant—which is the only distribution for which the expected value of the reciprocal is equal to the reciprocal of the expectation.

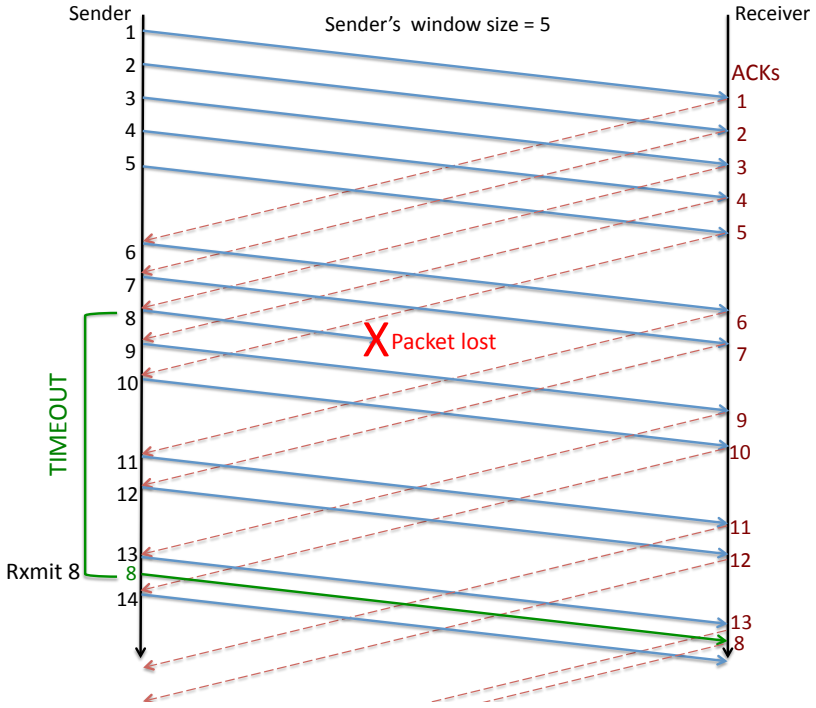


Figure 19-7: The sliding window protocol in action ($W = 5$ here).

■ 19.5 Sliding Window Protocol

The idea is to use a *window* of data packets that are *outstanding* along the path between sender and receiver. By "outstanding", we mean "unacknowledged". The idea then is to overlap data packet transmissions with ACK receptions. For our purposes, a window size of W data packets means that the sender has at most W outstanding data packets at any time. Our protocol will allow the sender to pick W, and the sender will try to have W outstanding data packets in the network at all times. The receiver is almost exactly the same as in the stop-and-wait case, except that it must also buffer data packets that might arrive out-of-order so that it can deliver them in order to the receiving application. This enhancement makes the receiver more complicated than before, but this complexity is worth the improvement in throughput in most situations.

The key idea in the protocol is that the window *slides* every time the sender gets an ACK. The reason is that the receipt of an ACK is a positive signal that one data packet left the network, and so the sender can add another to replenish the window. This plan is shown in Figure 19-7 that shows a sender (top line) with $W = 5$ and the receiver (bottom line) sending ACKs (dotted arrows) whenever it gets a data packet (solid arrow). Time moves from left to right here.

There are at least two different ways of defining a window in a reliable transport protocol. Here, we will use the following:

> **A window size of W means that the maximum number of outstanding (unacknowledged) data packets between sender and receiver is W.**

When there are no packet losses, the operation of the sliding window protocol is fairly straightforward. The sender transmits the next in-sequence data packet every time an ACK arrives; if the ACK is for data packet k and the window is W, the data packet sent out has sequence number $k + W$. The receiver ACKs each data packet echoing the sender's timestamp and delivers packets in sequence number order to the receiving application. The sender uses the ACKs to estimate the smoothed RTT and linear deviations and sets a timeout. Of course, the timeout will only be used if an ACK doesn't arrive for a data packet within that duration.

We now consider what happens when a packet is lost. Suppose the receiver has received data packets 0 through $k - 1$ and the sender doesn't get an ACK for data packet k. If the subsequent data packets in the window reach the receiver, then each of those packets triggers an ACK. So the sender will have the following ACKs assuming no further packets are lost: $k + 1, k + 2, \ldots, k + W - 1$. Moreover, upon the receipt of each of these ACKs, an additional new data packet will get sent with an even higher sequence number. But somewhere in the midst of these new data packet transmissions, the sender's timeout for data packet k will occur, and the sender will retransmit that packet. If that data packet reaches, then it will trigger an ACK, and if that ACK reaches the sender, yet another new data packet with a new sequence number one larger than the last sent so far will be sent.

Hence, this protocol tries hard to keep as many data packets outstanding as possible, *but not exceeding the window size*, W. If ℓ data packets or ACKs get lost, then the effective number of outstanding data packets reduces to $W - \ell$, until one of them times out, is retransmitted and received successfully by the receiver, and its ACK received successfully at the sender.

We will use a *fixed size* window in our discussion in this chapter. The sender picks a maximum window size and does not change that during a stream. In practice, most practical transport protocols on the Internet should implement a *congestion control* strategy to adjust the window size to prevailing network conditions (level of congestion, speed of data delivery, packet loss rates, round-trip times, etc.)

■ 19.5.1 Sliding Window Sender

We now describe the salient features of the sender side of this fixed-size sliding window protocol. The sender maintains `unacked_pkts`, a buffer of unacknowledged data packets. Every time the sender is called (by a fine-grained timer, which we assume fires each slot), it first checks to see whether any data packets were sent greater than "timeout" seconds ago (assuming time is maintained in seconds). If so, the sender retransmits each of these data packets, and takes care to change the packet transmission time of each of these packets to be the current time. For convenience, we usually maintain the time at which each packet was last sent in the packet data structure, though other ways of keeping track of this information are also possible.

After checking for retransmissions, the sender proceeds to see whether any new data packets can be sent. To properly check if any new packets can be sent, the sender maintains a variable, `outstanding`, which keeps track of the current number of outstanding data packets. If this value is smaller than the maximum window size, the sender sends a new data packet, setting the sequence number to be `max_seq` + 1, where `max_seq` is the highest sequence number sent so far. Of course, we should remember to update `max_seq` as well,

and increment `outstanding` by 1.

Whenever the sender gets an ACK, it should remove the acknowledged data packet from `unacked_pkts` (assuming it hasn't already been removed), decrement outstanding, and call the procedure to calculate the timeout (which will use the timestamp echoed in the current ACK to update the EWMA filters and update the timeout value).

We would like `outstanding` to keep track of the number of unackowledged data packets between sender and receiver. We have described the method to do this task as follows: increment it by 1 on each new data packet transmission, and decrement it by 1 on each ACK that was not previously seen by the sender, corresponding to a packet the sender had previously sent that is being acknowledged (as far as the sender is concerned) for the first time. The question now is whether `outstanding` should be adjusted when a *retransmission* is done. A little thought will show that it does not. The reason is that it is precisely on a timeout of a data packet that the sender believes that the packet was actually lost, and in the sender's view, the packet has left the network. But the retransmission immediately adds a data packet to the network, so the effect is that the number of outstanding packets is exactly the same. Hence, no change is required in the code.

Implementing a sliding window protocol is sometimes error-prone even when one completely understands the protocol in one's mind. Three kinds of errors are common. First, the timeouts are set too low because of an error in the EWMA estimators, and data packets end up being retransmitted too early, leading to spurious retransmissions. In addition to keeping track of the sender's smoothed round-trip time (srtt), RTT deviation, and timeout estimates,[6] it is a good idea to maintain a counter for the number of retransmissions done for each data packet. If the network has a certain total loss rate between sender and receiver and back (i.e., the bi-directional loss rate), p_l, the number of retransmissions should be on the order of $\frac{1}{1-p_l} - 1$, assuming that each packet is lost independently and with the same probability. (It is a useful exercise to work out why this formula holds.) If your implementation shows a much larger number than this prediction, it is very likely that there's a bug in it.

Second, the number of outstanding data packets might be larger than the configured window, which is an error. If that occurs, and especially if a bug causes the number of outstanding packets to grow unbounded, delays will increase and it is also possible that packet loss rates caused by congestion will increase. It is useful to place an assertion or two that checks that the outstanding number of data packets does not exceed the configured window.

Third, when retransmitting a data packet, the sender must take care to modify the time at which the packet is sent. Otherwise, that packet will end up getting retransmitted repeatedly, a pretty serious bug that will cause the throughput to diminish.

■ 19.5.2 Sliding Window Receiver

At the receiver, the biggest change to the stop-and-wait case is to maintain a list of received data packets that are out-of-order. Call this list `rcvbuf`. Each data packet that arrives is added to this list, assuming it is not already on the list. It's convenient to store this list in increasing sequence order. Then, check to see whether one or more contiguous data packets starting from `rcv_seqnum` + 1 are in `rcvbuf`. If they are, deliver them to the

[6] In our lab, this information will be printed when you click on the sender node.

application, remove them from `rcvbuf`, and remember to update `rcv_seqnum`.

19.5.3 Throughput

What is the throughput of the sliding window protocol we just developed? Clearly, we send W data packets per RTT when there are no data packet or ACK losses, so the throughput in the absence of losses is W/RTT packets per second. So the question one should ask is, what should we set W to in order to maximize throughput, at least when there are no data packet or ACK losses? After answering this question, we will provide a simple formula for the throughput of the protocol in the absence of losses, and then finally consider packet losses.

Setting W

One can address the question of how to choose W using Little's law. Think of the entire bi-directional path between the sender and receiver as a single queue (in reality it's more complicated than a single queue, but the abstraction of a single queue still holds). W is the number of (unacknowledged) packets in the system and RTT is the mean delay between the transmission of a data packet and the receipt of its ACK at the sender (upon which the sender transmits a new data packet). We would like to maximize the processing rate of this system. Note that this rate cannot exceed the bit rate of the slowest, or *bottleneck*, link between the sender and receiver (i.e., the rate of the *bottleneck link*) . If that rate is B packets per second, then by Little's law, setting $W = B \times$ RTT will ensure that the protocol comes close to achieving a thoroughput equal to the available bit rate.

But what should the RTT be in the above formula? After all, the definition of a "RTT sample" is the time that elapses between the transmission of a data packet and the receipt of an ACK for it. As such, it depends on other data using the path. Moreover, if one looks at the formula $B = W/$ RTT, it suggests that one can simply increase the window size W to any value and B may correspondingly just increase. Clearly, that can't be right!

Consider the simple case when there is only one connection active over a network path. Observe that the RTT experienced by a packet P sent on the connection may be broken into two parts: one part that does not depend on any queueing delay (i.e., the sum of the propagation, transmission, and processing delays of the packet and its ACK), and one part that depends on how many other packets were ahead of P in the bottleneck queue. (Here we are assuming that ACKs experience no queueing, for simplicity.) Denote the RTT in the absence of queuing as RTT_{min}, the minimum possible round-trip time that the connection can experience.

Now, suppose the RTT of the connection is equal to RTT_{min}. That is, there is no queue building up at the bottleneck link. Then, the throughput of the connection is $W/\text{RTT} = W/\text{RTT}_{\text{min}}$. We would like this throughput to be the bottleneck link rate, B. Setting $W/\text{RTT}_{\text{min}} = B$, we find that W should be equal to $B \cdot \text{RTT}_{\text{min}}$.

This quantity—$B \cdot \text{RTT}_{\text{min}}$—is an important concept for sliding window protocols (all sliding window protocols, not just the one we have studied). It is called the **bandwidth-delay product** of the connection and is a property of the bi-directional network path between sender and receiver. When the window size is strictly smaller than the bandwidth-delay product, the throughput will be strictly smaller than the bottleneck rate, B, and the queueing delay will be non-existent. In this phase, the connection's throughput *linearly*

increases as we increase the window size, W, assuming no other traffic intervenes. The smallest window size for which the throughput will be equal to B is the bandwidth-delay product.

This discussion shows that for our sliding window protocol, setting $W = B \times \text{RTT}_{\text{min}}$ achieves the maximum possible throughput, B, *in the absence of any data packet or ACK losses*. When packet losses occur, the window size will need to be higher to get maximum throughput (utilization), because we need a sufficient number of unacknowledged data packets to keep a $B \times \text{RTT}_{\text{min}}$ worth of packets even when losses occur. A smaller window size will achieve sub-optimal throughput, linear in the window size, and inversely proportional to RTT_{min}.

But once W exceeds $B \times \text{RTT}_{\text{min}}$, the RTT experienced by the connection includes queueing as well, and the RTT will *no longer be a constant independent of* W! That is, increasing W will cause RTT to also increase, but the rate, B, will no longer increase. What is the throughput in this case?

We can answer this question by applying Little's law *twice*. Once at the bottleneck link's queue, and once on the entire network path. We will show the intuitive result that if $W > B \times \text{RTT}_{\text{min}}$, then the throughput is B packets per second.

First, let the average number of packets at the queue of the bottleneck link be Q. By Little's law applied to this queue, we know that $Q = B \cdot \tau$, where B is the rate at which the queue drains (i.e., the bottleneck link rate), and τ is the average delay in the queue, so $\tau = Q/B$.

We also know that

$$\text{RTT} = \text{RTT}_{\text{min}} + \tau = \text{RTT}_{\text{min}} + Q/B. \tag{19.5}$$

Now, consider the window size, W, which is the number of unacknowledged packets. We know that all these packets, by conservation of packets, must either be in the bottleneck queue, or in the non-queueing part of the system. That is,

$$W = Q + B \cdot \text{RTT}_{\text{min}}. \tag{19.6}$$

Finally, from Little's law applied to the entire bi-directional network path,

$$\begin{aligned} \text{Throughput} &= \frac{W}{\text{RTT}} && (19.7)\\ &= \frac{B \cdot \text{RTT}_{\text{min}} + Q}{\text{RTT}_{\text{min}} + (Q/B)} && (19.8)\\ &= B && (19.9) \end{aligned}$$

Thus, we can conclude that, in the absence of any data packet or ACK losses, the connection's throughput is as shown schematically in Figure 19-8.

Throughput of the sliding window protocol with packet losses

Assuming that one sets the window size properly, i.e., to be large enough so that $W \geq B \times \text{RTT}_{\text{min}}$ always, even in the presence of data or ACK losses, what is the maximum throughput of our sliding window protocol if the network has a certain probability of packet loss?

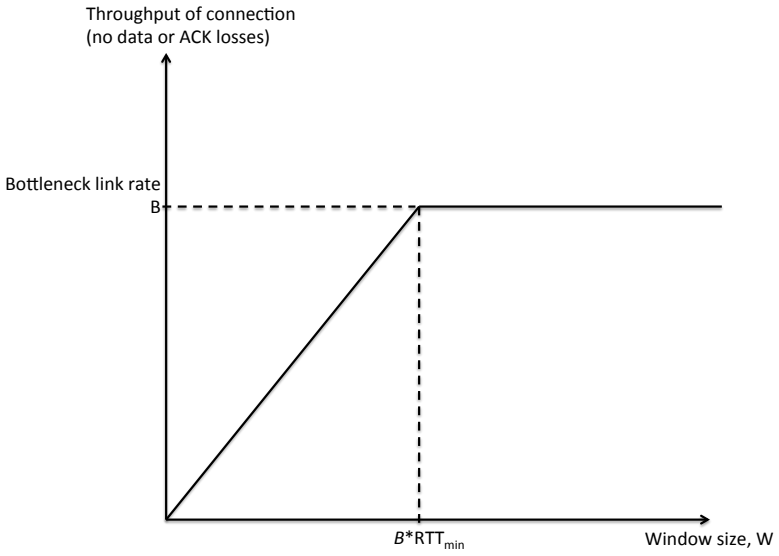


Figure 19-8: Throughput of the sliding window protocol as a function of the window size in a network with no other traffic. The bottleneck link rate is B packets per second and the RTT without any queueing is RTT_{min}. The product of these two quantities is the bandwidth-delay product.

Consider a simple model in which the network path loses any packet—data or ACK—such that the probability of either a data packet being lost *or* its ACK being lost is equal to ℓ, and the packet loss random process is independent and identically distributed (the same model as in our analysis of stop-and-wait). Then, the utilization achieved by our sliding window reliable transport protocol is at most $1 - \ell$. Moreover, for a large-enough window size, W, our sliding window protocol comes close to achieving it.

The reason for the upper bound on utilization is that in this protocol, a data packet is acknowledged only when the sender gets an ACK explicitly for that packet. Now consider the number of transmissions that any given data packet must incur before its ACK is received by the sender. With probability $1 - \ell$, we need one transmission, with probability $\ell(1 - \ell)$, we need two transmissions, and so on, giving us an *expected number of transmissions* of $\frac{1}{1-\ell}$. If we make this number of transmissions, one data packet is successfully sent and acknowledged. Hence, the utilization of the protocol can be at most $\frac{1}{\frac{1}{1-\ell}} = 1 - \ell$. In fact, it turns out the $1 - \ell$ is the *capacity* (i.e., upper-bound on throughput) for *any* channel (network path) with packet loss rate ℓ.

If the sender picks a window size sufficiently larger than the bandwidth-minimum-RTT product, so that at least bandwidth-minimum-RTT packets are in transit (unacknowledged) even in the face of data and ACK losses, then the protocol's utilization will be close to the maximum value of $1 - \ell$.

Is a good timeout important for the sliding window protocol?

Given that our sliding window protocol always sends a data packet every time the sender gets an ACK, one might reasonably ask whether setting a good timeout value, which under even the best of conditions involves a hard trade-off, is essential. The answer turns out to be subtle: it's true that the timeout can be quite large, because data packets will continue to flow as long as some ACKs are arriving. However, as data packets (or ACKs) get lost, the effective window size keeps falling, and eventually the protocol will stall until the sender retransmits. So one can't ignore the task of picking a timeout altogether, but one can pick a more conservative (longer) timeout than in the stop-and-wait protocol. However, the longer the timeout, the bigger the stalls experienced by the receiver application—even though the receiver's transport protocol would have received the data packets, they can't be delivered to the application because it wants the data to be delivered *in order*. Therefore, a good timeout is still quite useful, and the principles discussed in setting it are widely useful.

Secondly, we note that the longer the timeout, the bigger the receiver's buffer has to be when there are losses; in fact, in the worst case, there is no bound on how big the receiver's buffer can get. To see why, think about what happens if we were unlucky and a data packet with a particular sequence number kept getting lost, but everything else got through.

The two factors mentioned above affect the throughput of the transport protocol, but the biggest consequence of a long timeout is the effect on the *latency* perceived by applications (and users). The reason is that data packets are delivered in-order by the protocol to the application, which means that a missing packet with sequence number k will cause the application to stall, even though data packets with sequence numbers larger than k have arrived and are in the transport protocol's receiver buffer. Hence, an excessively long timeout hurts interactivity and degrades the user's experience.

■ 19.6 Summary

This chapter described the key concepts in the design on a reliable data transport protocol. The big idea is to use redundancy in the form of careful retransmissions, for which we developed the idea of using sequence numbers to uniquely identify data packets and acknowledgments for the receiver to signal the successful reception of a data packet to the sender. We discussed how the sender can set a good timeout, balancing between the ability to track a persistent change of the round-trip times against the ability to ignore non-persistent glitches. The method to calculate the timeout involved estimating a smoothed mean and linear deviation using an exponential weighted moving average, which is a single real-zero low-pass filter. The timeout itself is set at the mean + 4 times the deviation to ensure that the tail probability of a spurious retransmission is small. We used these ideas in developing the simple stop-and-wait protocol.

We then developed the idea of a sliding window to improve performance, and showed how to modify the sender and receiver to use this concept. Both the sender and receiver are now more complicated than in the stop-and-wait protocol, but when there are no losses, one can set the window size to the bandwidth-delay product and achieve high throughput in this protocol. We also studied how increasing the window size increases the throughput linearly up to a point, after only the (queueing) delay increases, and not the throughput of

the connection.

Acknowledgments

Thanks to Karl Berggren, Katrina LaCurts, Alexandre Megretski, and Sari Canelake for suggesting significant and helpful improvements to this chapter.

Problems and Questions

1. Consider a best-effort network with variable delays and losses. In such a network, Louis Reasoner suggests that the receiver does not need to send the sequence number in the ACK in a correctly implemented stop-and-wait protocol, where the sender sends data packet $k+1$ *only after* the ACK for data packet k is received. Explain whether he is correct or not.

2. The 802.11 (WiFi) link-layer uses a stop-and-wait protocol to improve link reliability. The protocol works as follows:

   (a) The sender transmits data packet $k+1$ to the receiver as soon as it receives an ACK for the data packet k.
   (b) After the receiver gets the entire data packet, it computes a checksum (CRC). The processing time to compute the CRC is T_p and you may assume that it does not depend on the packet size.
   (c) If the CRC is correct, the receiver sends a link-layer ACK to the sender. The ACK has negligible size and reaches the sender instantaneously.

   The sender and receiver are near each other, so you can ignore the propagation delay. The bit rate is $R = 54$ Megabits/s, the smallest data packet size is 540 bits, and the largest data packet size is 5,400 bits.

   What is the maximum processing time T_p that ensures that the protocol will achieve a throughput of *at least 50%* of the bit rate of the link in the absence of data packet and ACK losses, *for any data packet size*?

3. Alyssa P. Hacker sets up a wireless network in her home to enable her computer ("client") to communicate with an Access Point (AP). The client and AP communicate with each other using a stop-and-wait protocol.

   The data packet size is 10000 bits. The total round-trip time (RTT) between the AP and client is equal to 0.2 milliseconds (that includes the time to process the packet, transmit an ACK, and process the ACK at the sender) **plus** the transmission time of the 10000 bit packet over the link.

   Alyssa can configure two possible transmission bit rates for her link, with the following properties:

Bit rate	Bi-directional packet loss probability	RTT
10 Megabits/s	1/11	____
20 Megabits/s	1/4	____

Alyssa's goal is to select the bit rate that provides the higher throughput for a stream of packets that need to be delivered reliably between the AP and client using stop-and-wait. For both bit rates, the **retransmission timeout (RTO) is 2.4 milliseconds**.

(a) Calculate the round-trip time (RTT) for each bit rate?

(b) For each bit rate, calculate the **expected time**, in milliseconds, to successfully deliver a packet and get an ACK for it. **Show your work.**

(c) Using the above calculations, which bit rate would you choose to achieve Alyssa's goal?

4. Suppose the sender in a reliable transport protocol uses an EWMA filter to estimate the smoothed round trip time, srtt, every time it gets an ACK with an RTT sample r.

   $\text{srtt} \rightarrow \alpha \cdot r + (1-\alpha) \cdot \text{srtt}$

   We would like every data packet in a window to contribute a weight of at least 1% to the srtt calculation. As the window size increases, should α increase, decrease, or remain the same, to achieve this goal? (You should be able to answer this question without writing any equations.)

5. TCP computes an average round-trip time (RTT) for the connection using an EWMA estimator, as in the previous problem. Suppose that at time 0, the initial estimate, srtt, is equal to the true value, r_0. Suppose that immediately after this time, the RTT for the connection increases to a value R and remains at that value for the remainder of the connection. You may assume that $R >> r_0$.

   Suppose that the TCP retransmission timeout value at step n, $RTO(n)$, is set to $\beta \cdot \text{srtt}$. Calculate the number of RTT samples before we can be sure that there will be no spurious retransmissions. Old TCP implementations used to have $\beta = 2$ and $\alpha = 1/8$. How many samples does this correspond to before spurious retransmissions are avoided, for this problem? (As explained in Section 19.3, TCP now uses the mean linear deviation as its RTO formula. Originally, TCP didn't incorporate the linear deviation in its RTO formula.)

6. Consider a sliding window protocol between a sender and a receiver. The receiver should deliver data packets reliably and in order to its application.

   The sender correctly maintains the following state variables:
   `unacked_pkts` – the buffer of unacknowledged data packets
   `first_unacked` – the lowest unacked sequence number (undefined if all data packets have been acked)
   `last_unacked` – the highest unacked sequence number (undefined if all data

packets have been acked)

`last_sent` – the highest sequence number sent so far (whether acknowledged or not)

If the receiver gets a data packet that is strictly larger than the next one in sequence, it adds the packet to a buffer if not already present. We want to ensure that the size of this buffer of data packets awaiting delivery *never exceeds* a value $W \geq 0$. Write down the check(s) that the sender should perform before sending a new data packet in terms of the variables mentioned above that ensure the desired property.

7. Alyssa P. Hacker measures that the network path between two computers has a round-trip time (RTT) of 100 milliseconds. The queueing delay is negligible. The speed of the bottleneck link between them is 1 Mbyte/s. Alyssa implements the reliable sliding window protocol studied in 6.02 and runs it between these two computers. The data packet size is fixed at 1000 bytes (you can ignore the size of the acknowledgments). There is no other traffic.

   (a) Alyssa sets the window size to 10 data packets. What is the resulting maximum utilization of the bottleneck link? Explain your answer.

   (b) Alyssa's implementation of a sliding window protocol uses an 8-bit field for the sequence number in each data packet. Assuming that the RTT remains the same, what is the smallest value of the bottleneck link bandwidth (in Mbytes/s) that will cause the protocol to stop working correctly when packet losses occur? Assume that the definition of a window in her protocol is the difference between the last transmitted sequence number and the last in-sequence ACK.

   (c) Suppose the window size is 10 data packets and that the value of the sender's retransmission timeout is 1 second. A data packet gets lost before it reaches the receiver. The protocol continues *and no other data packets or acks are lost*. The receiver wants to deliver data to the application in order.

   What is the maximum size, in packets, that the buffer at the receiver can grow to in the sliding window protocol? Answer this question for the two different definitions of a "window" below.

      i. When the window is the maximum difference between the last transmitted data packet and the last in-sequence ACK received at the sender:

      ii. When the window is the maximum number of unacknowledged data packets at the sender:

8. In the reliable transport protocols we studied, the receiver sends an acknowledgment (ACK) saying "I got k" whenever it receives a data packet with sequence number k. Ben Bitdiddle invents a different method using **cumulative ACKs**: whenever the receiver gets a data packet, whether in order or not, it sends an ACK saying "I got every data packet up to and including ℓ", where ℓ is the **highest, in-order** data packet received so far.

   The definition of the window is the same as before: a window size of W means that the maximum number of unacknowledged data packets is W. Every time the sender gets an ACK, it may transmit one or more data packets, within the constraint of the

window size. It also implements a timeout mechanism to retransmit data packets that it believes are lost using the algorithm described in these notes. The protocol runs over a best-effort network, but *no data packet or ACK is duplicated at the network or link layers*.

The sender sends a stream of new data packets according to the sliding window protocol, and in response gets the following cumulative ACKs from the receiver:

1 2 3 4 4 4 4 4 4 4

(a) Now, suppose that the sender times out and retransmits the first unacknowledged data packet. When the receiver gets that retransmitted data packet, what can you say about the ACK, a, that it sends?

i. $a = 5$.
ii. $a \geq 5$.
iii. $5 \leq a \leq 11$.
iv. $a = 11$.
v. $a \leq 11$.

(b) Assuming no ACKs were lost, what is the *minimum* window size that can produce the sequence of ACKs shown above?

(c) Is it possible for the given sequence of cumulative ACKs to have arrived at the sender even when no data packets were lost en route to the receiver when they were sent?

(d) A little bit into the data transfer, the sender observes the following sequence of cumulative ACKs sent from the receiver:

21 22 23 25 28

The window size is 8 packets. What data packet(s) should the sender transmit upon receiving each of the above ACKs, if it wants to maximize the number of unacknowledged data packets?

On getting ACK #	→ Send ??	On getting ACK #	→ Send ??
21	→	22	→
23	→	25	→
28	→		

9. Give one example of a situation where the cumulative ACK protocol described in the previous problem gets higher throughput than the sliding window protocol described in this chapter.

10. A sender S and receiver R communicate reliably over a series of links using a sliding window protocol with some window size, W packets. The path between S and R has one bottleneck link (i.e., one link whose rate bounds the throughput that can be achieved), whose data rate is C packets/second. When the window size is W, the queue at the bottleneck link is always **full**, with Q data packets in it. The round trip time (RTT) of the connection between S and R during this data transfer with window

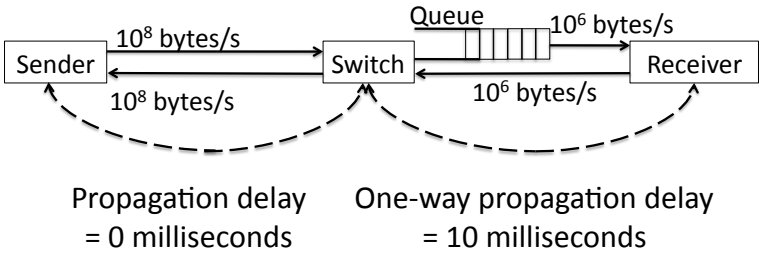
size W is T seconds, *including the queueing delay*. There are no data packet or ACK losses in this case, and there are no other connections sharing this path.

(a) Write an expression for W in terms of the other parameters specified above.

(b) We would like to reduce the window size from W and still achieve high utilization. What is the minimum window size, W_{min}, which will achieve 100% utilization of the bottleneck link? Express your answer as a function of C, T, and Q.

(c) Now suppose the sender starts with a window size set to W_{min}. If all these data packets get acknowledged and no packet losses occur in the window, the sender increases the window size by 1. The sender keeps increasing the window size in this fashion until it reaches a window size that causes a data packet loss to occur. What is the smallest window size at which the sender observes a data packet loss caused by the bottleneck queue overflowing? Assume that no ACKs are lost.

11. Ben Bitdiddle decides to use the sliding window transport protocol described in these notes on the network shown in Figure 19-9. The receiver sends **end-to-end ACKs** to the sender. The switch in the middle simply forwards packets in best-effort fashion.



Max queue size = 100 packets
Packet size = 1000 bytes
ACK size = 40 bytes
Initial sender window size = 10 packets

Figure 19-9: Ben's network.

(a) The sender's window size is 10 packets. At what approximate rate (in packets per second) will the protocol deliver a multi-gigabyte file from the sender to the receiver? Assume that there is no other traffic in the network and packets can only be lost because the queues overflow.

i. Between 900 and 1000.

ii. Between 450 and 500.

iii. Between 225 and 250.

iv. Depends on the timeout value used.

(b) You would like to double the throughput of this sliding window transport protocol running on the network shown on the previous page. To do so, you can apply **one** of the following techniques alone:

i. Double the window size.

ii. Halve the propagation time of the links.

iii. Double the speed of the link between the Switch and Receiver.

For each of the following sender window sizes, list which of the above techniques, **if any, can approximately double the throughput**. If no technique does the job, say "None". There might be more than one answer for each window size, in which case you should list them all. Each technique works in isolation.

1. $W = 10$: _____
2. $W = 50$: _____
3. $W = 30$: _____

12. Eager B. Eaver starts MyFace, a next-generation social networking web site in which the only pictures allowed are users' faces. MyFace has a simple request-response interface. The client sends a request (for a face), the server sends a response (the face). Both request and response fit in one packet (the faces in the responses are small pictures!). When the client gets a response, it immediately sends the next request. The size of the largest packet is $S = 1000$ bytes.

Eager's server is in Cambridge. Clients come from all over the world. Eager's measurements show that one can model the typical client as having a 100 millisecond round-trip time (RTT) to the server (i.e., the network component of the request-response delay, not counting the additional processing time taken by the server, is 100 milliseconds).

If the client does not get a response from the server in a time τ, it resends the request. It keeps doing that until it gets a response.

(a) Is the protocol described above "at least once", "at most once", or "exactly once"?

(b) Eager needs to provision the link bandwidth for MyFace. He anticipates that at any given time, the largest number of clients making a request is 2000. What minimum outgoing link bandwidth from MyFace will ensure that the link connecting MyFace to the Internet will not experience congestion?

(c) Suppose the probability of the client receiving a response from the server for any given request is p. What is the expected time for a client's request to obtain a response from the server? Your answer will depend on p, RTT, and τ.

13. Lem E. Tweetit is designing a new protocol for Tweeter, a Twitter rip-off. All tweets in Tweeter are 1000 bytes in length. Each tweet sent by a client and received by the Tweeter server is immediately acknowledged by the server; if the client does not receive an ACK within a timeout, it re-sends the tweets, and repeats this process until it gets an ACK.

Sir Tweetsalot uses a device whose data transmission rate is 100 Kbytes/s, which you can assume is the bottleneck rate between his client and the server. The round-trip propagation time between his client and the server is 10 milliseconds. Assume that there is no queueing on any link between client and server and that the processing time along the path is 0. You may also assume that the ACKs are very small in size, so consume neglible bandwidth and transmission time (of course, they still need to propagate from server to client). Do not ignore the transmission time of a tweet.

(a) What is the smallest value of the timeout, in *milliseconds*, that will avoid spurious retransmissions?

(b) Suppose that the timeout is set to 90 milliseconds. Unfortunately, the probability that a given client transmission gets an ACK is only 75%. What is the *utilization* of the network?

14. A sender A and a receiver B communicate using the stop-and-wait protocol studied in this chapter. There are n links on the path between A and B, each with a data rate of R bits per second. The size of a data packet is S bits and the size of an ACK is K bits. Each link has a physical distance of D meters and the speed of signal propagation over each link is c meters per second. The total processing time experienced by a data packet *and* its ACK is T_p seconds. ACKs traverse the same links as data packets, except in the opposite direction on each link (the propagation time and data rate are the same in both directions of a link). There is no queueing delay in this network. Each link has a packet loss probability of p, with packets being lost independently.

    What are the following four quantities in terms of the parameters given?

    (a) Transmission time for a data packet *on one link* between A and B: _____.

    (b) Propagation time for a data packet across n links between A and B: _____.

    (c) Round-trip time (RTT) between A and B? _____.
    (The RTT is defined as the elapsed time between the start of transmission of a data packet and the completion of receipt of the ACK sent in response to the data packet's reception by the receiver.)

    (d) Probability that a data packet sent by A will reach B: _____.

15. Ben Bitdiddle gets rid of the timestamps from the packet header in this chapter's stop-and-wait transport protocol running over a best-effort network. The network may lose or reorder packets, but it never duplicates a packet. In the protocol, the receiver sends an ACK for each data packet it receives, echoing the sequence number of the packet that was just received.

    The sender uses the following method to estimate the round-trip time (RTT) of the connection:

1. When the sender transmits a packet with sequence number k, it stores the time on its machine at which the packet was sent, t_k. If the transmission is a retransmission of sequence number k, then t_k is updated.
2. When the sender gets an ACK for packet k, if it has not already gotten an ACK for k so far, it observes the current time on its machine, a_k, and measures the RTT sample as $a_k - t_k$.

If the ACK received by the sender at time a_k was sent by the receiver in response to a data packet sent at time t_k, then the RTT sample $a_k - t_k$ is said to be correct. Otherwise, it is incorrect.

State **True** or **False** for the following statements, with an explanation for your choice.

(a) If the sender never retransmits a data packet during a data transfer, then all the RTT samples produced by Ben's method are correct.

(b) If data and ACK packets are never reordered in the network, then all the RTT samples produced by Ben's method are correct.

(c) If the sender makes no spurious retransmissions during a data transfer (i.e., it only retransmits a data packet if all previous transmissions of data packets with the same sequence number did in fact get dropped before reaching the receiver), then all the RTT samples produced by Ben's method are correct.

16. Opt E. Miser implements this chapter's stop-and-wait reliable transport protocol with one modification: being stingy, he replaces the sequence number field with a 1-bit field, deciding to reuse sequence numbers across data packets. The first data packet has sequence number 1, the second has number 0, the third has number 1, the fourth has number 0, and so on. Whenever the receiver gets a packet with sequence number s(= 0 or 1), it sends an ACK to the sender echoing s. The receiver delivers a data packet to the application if, and only if, its sequence number is different from the last one delivered, and upon delivery, updates the last sequence number delivered.

    He runs this protocol over a best-effort network that can lose packets (with probability < 1) or reorder them, and whose delays are variable. Explain whether the modified protocol always provides reliable, in-order delivery of a stream of packets.

17. Consider a reliable transport connection using this chapter's sliding window protocol on a network path whose RTT in the absence of queueing is $\text{RTT}_{\text{min}} = 0.1$ seconds. The connection's bottleneck link has a rate of $C = 100$ packets per second, and the queue in front of the bottleneck link has space for $Q = 20$ packets.

    Assume that the sender uses a sliding window protocol with fixed window size. There is no other traffic on the path.

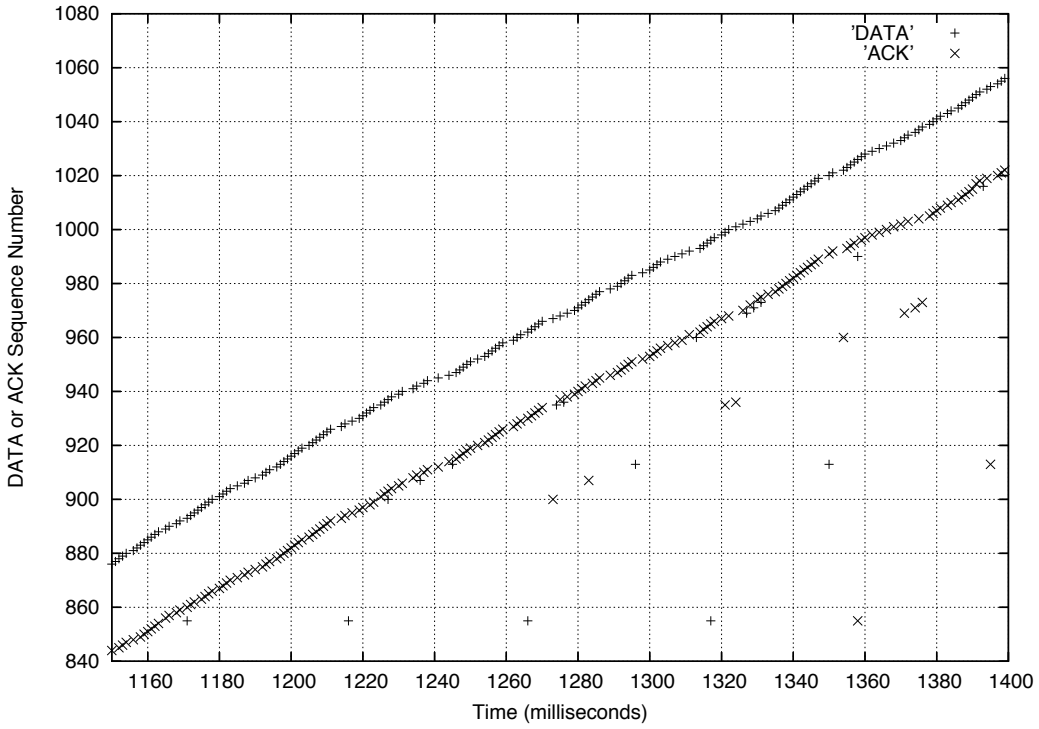
    (a) If the window size is 8 packets, then what is the throughput of the connection?

    (b) If the window size is 16 packets, then what is the throughput of the connection?

    (c) What is the smallest window size for which the connection's RTT exceeds RTT_{min}?

(d) What is the largest value of the sender window size for which no packets are lost due to a queue overflow?

18. Annette Werker correctly implements the fixed-size sliding window protocol described in this chapter. She instruments the sender to store the time at which each DATA packet is sent and the time at which each ACK is received. A snippet of the DATA and ACK traces from an experiment is shown in the picture below. Each $+$ is a DATA packet transmission, with the x-axis showing the transmission time and the y-axis showing the sequence number. Each $\times$ is an ACK reception, with the x-axis showing the ACK reception time and the y-axis showing the ACK sequence number. All DATA packets have the same size.



Answer the following questions, providing a brief explanation for each one.

(a) Estimate any one sample round-trip time (RTT) of the connection.

(b) Estimate the sender's retransmission timeout (RTO) for this trace.

(c) On the picture, circle DATA packet retransmissions for four different sequence numbers.

(d) Some DATA packets in this trace may have incurred more than one retransmission? On the picture, draw a square around one such retransmission.

(e) What is your best estimate of the sender's window size?

(f) What is your best estimate of the throughput in packets per second of the connection?

(g) Considering only sequence numbers > 880, what is your best estimate of the packet loss rate experienced by DATA packets?

19. Consider the same setup as the previous problem. Suppose the window size for the connection is equal to twice the bandwidth-delay product of the network path.

For each change to the parameters of the network path or the sender given below, explain if the connection's throughput (not utilization) will increase, decrease, or remain the same. In each statement, nothing other than what is specified in that statement changes.

(a) The packet loss rate, ℓ, decreases to $\ell/3$.

(b) The minimum value of the RTT, R, increases to $1.8R$.

(c) The window size, W, decreases to $W/3$.