MIT 6.02 **DRAFT** *Lecture Notes*
Spring 2010 (Last update: April 20, 2010)
Comments, questions or bug reports?
Please contact 6.02-staff@mit.edu

LECTURE 20
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, etc. 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 lecture will not survey them all, but will instead discuss two protocols in some detail. The first protocol, called **stop-and-wait**, will solve the problem, 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 losses*, and *receiver buffering to cope with reordering*. 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 lecture and the next one will cover these issues, and discuss ways in which a reliable transport protocol can achieve high throughput.

■ 20.1 The Problem

The problem we're going to solve is 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 pack-

ets arbitratily, 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 looked at using *error-correcting codes* such as block codes, Reed-Solomon, or convolutional codes, to combat bit errors. In principle, one could apply such (or 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 the Internet today use retransmissions.

We will develop the key ideas in the context of two protocols: **stop-and-wait** and **simple sliding window**. 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 that would like to send and receive data in a reliable, in-order manner.

■ 20.2 Stop-and-Wait Protocol

The high-level idea is quite simple. The sender attaches a header to every packet (distinct from the network header that contains the destination address, hop limit, and header checksum discussed in the previous lectures) that includes a unique identifier for the packet. This identifier will never be reused for two different packets on the same stream. The receiver, upon receiving the 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 packet k" to the sender.

The sender sends the next 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* packet k.

The receiver's job is to deliver each packet it receives to the receiver application. Fig-

[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.

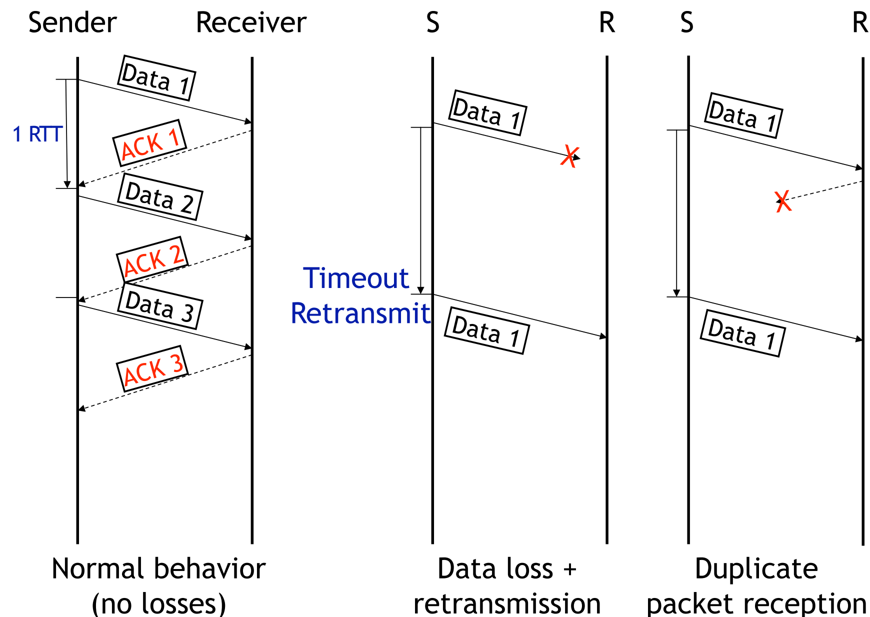


Figure 20-1: The stop-and-wait protocol. Each picture has a sender timeline and a receiver timeline. Time starts at the top of each vertical line and increases moving downward. The picture on the left shows what happens when there are no losses; the middle shows what happens on a data packet loss; and the right shows how duplicate packets may arrive at the receiver because of an ACK loss.

ure 20-1 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: how to pick unique identifiers, how this protocol may deliver duplicate packets to the receiver application, and how to pick the timeout.

■ 20.2.1 Selecting Unique Identifiers: Sequence Numbers

The sender may pick any unique identifier for a packet, but in most transport protocols, a convenient (and effective) way of doing so is to use incrementing sequence numbers. The simplest way to achieve this goal is for the sender and receiver to somehow 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 packet. Thus, the packet sent after the ACK for k is received by the sender will be 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 course, except to note that establishing and properly terminating these streams (which

are called connections) reliably is a non-trivial problem.

■ 20.2.2 Semantics of Our 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 20-1 (right). A packet may have reached the receiver, but the ACK doesn't reach the sender, and the sender will then timeout and retransmit the packet. The receiver will get multiple copies of the packet, and deliver both to the receiver application. Another reason is that the sender might have timed out, but the original packet may not actually have been lost. Such a retransmission is called a *spurious retransmission*, and is a waste of bandwidth.

Preventing duplicates: The solution to this problem is for the receiver to keep track of the last *in-sequence* packet it has delivered to the application. Call this value `rcv_seqnum`. If a packet with sequence number less than or equal to `rcv_seqnum` arrives, then send an ACK for it and discard the packet. This is the duplicate suppression step. If a packet with sequence number `rcv_seqnum` + 1 arrives, then send an ACK and deliver it to the application. Note that a packet with sequence number greater than `rcv_seqnum` + 1 should never arrive in this 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 packet. So, if such a 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.[2]

■ 20.2.3 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 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 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.

It should be clear that the natural time-scale in the protocol is the time between the transmission of a 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 average RTT (and in fact, it must be quite a bit bigger than the average, as we'll see).

[2]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.

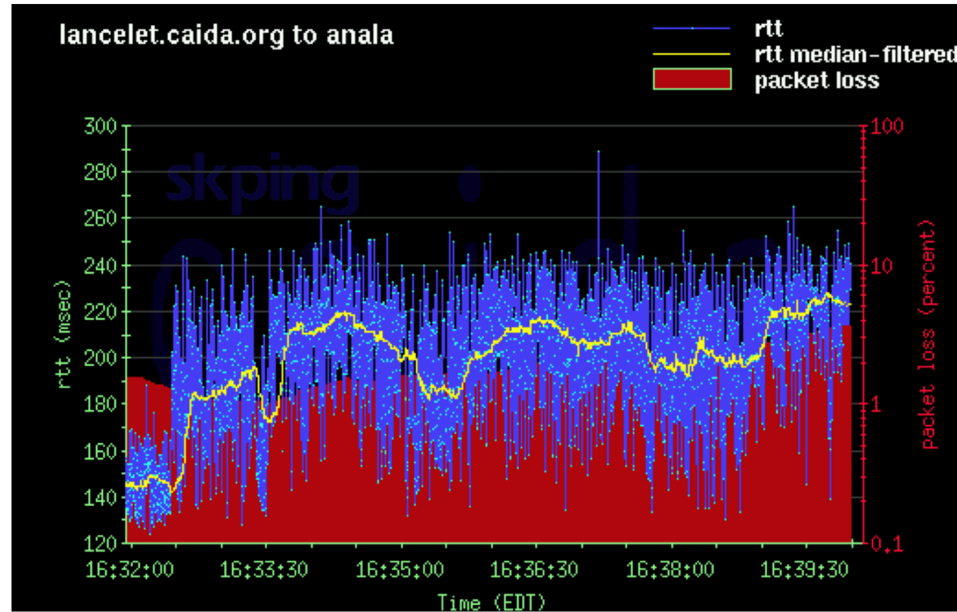


Figure 20-2: RTT variations are pronounced in many networks.

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 20.3.1). In fact, the throughput of many transport protocols depends on the RTT because the RTT is the time it takes to get feedback from the receiver about the fate of any given packet.

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.

■ 20.3 Adaptive RTT Estimation and Setting Timeouts

The RTT experienced by packets is variable because the delays in our network are variable. An example is shown in Figure 20-2, which shows the RTT of an Internet path between two hosts as a function of time-of-day (blue) and the packet loss rate (red). 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*.

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

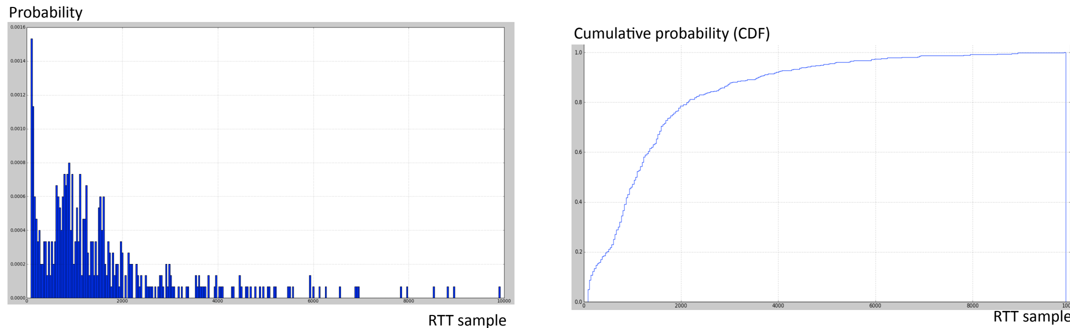


Figure 20-3: RTT variations on a wide-area cellular wireless network (Verizon Wireless's 3G CDMA Rev A service) across both idle periods and when data transfers are in progress, showing extremely high RTT values and high variability. The x-axis in both pictures is the RTT in milliseconds. The picture 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 density 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.)

terms of the probability of a spurious retransmission, as shown by the tail probability in Figure 20-3 (the area under the curve to the right of the "Timeout" mark). Real-world distributions of RTT are not 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 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 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 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 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:

$$s_n = \alpha r_n + (1 - \alpha) s_{n-1}, \tag{20.1}$$

where $0 < \alpha < 1$.

A little algebra shows that this equation is an LPF whose frequency response is:

$$H(e^{j\Omega}) = \frac{\alpha}{1 - (1 - \alpha) z}, \tag{20.2}$$

where $z = e^{-j\Omega}$. It 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

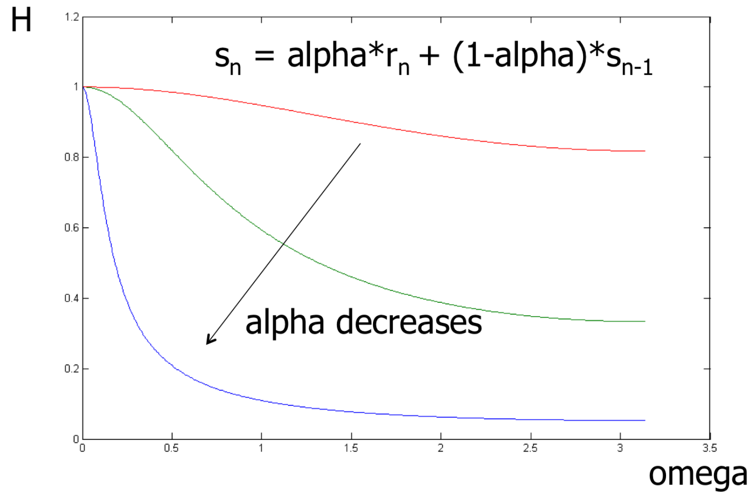


Figure 20-4: Frequency response of the exponential weighted moving average low-pass filter. As α decreases, the low-pass filter becomes even more pronounced. The graph shows the response for $\alpha = 0.9, 0.5, 0.1$, going from top to bottom.

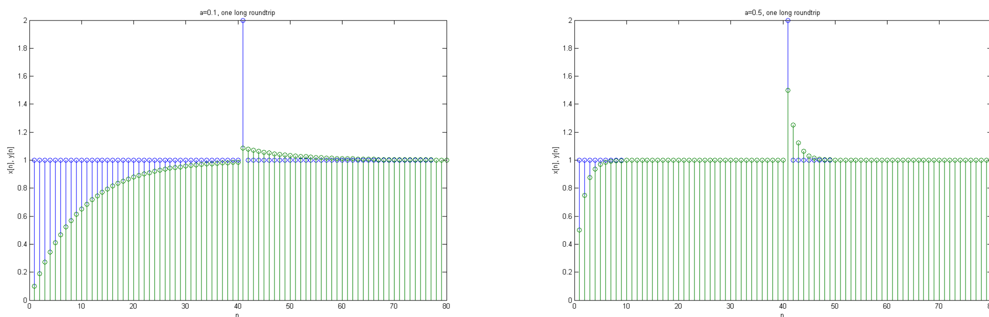


Figure 20-5: Reaction of the exponential weighted moving average filter to a non-persistent spike in the RTT (the spike is double the other samples). The smaller α (0.1, shown on the left) doesn't get swayed by it, whereas the bigger value (0.5, right) does. The output of the filter is shown in green, the input in blue.

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 20-4 illustrates this point.

Figure 20-5 shows how different values of α react to a sudden non-persistent change in the RTT, while Figure 20-6 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 20.1 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

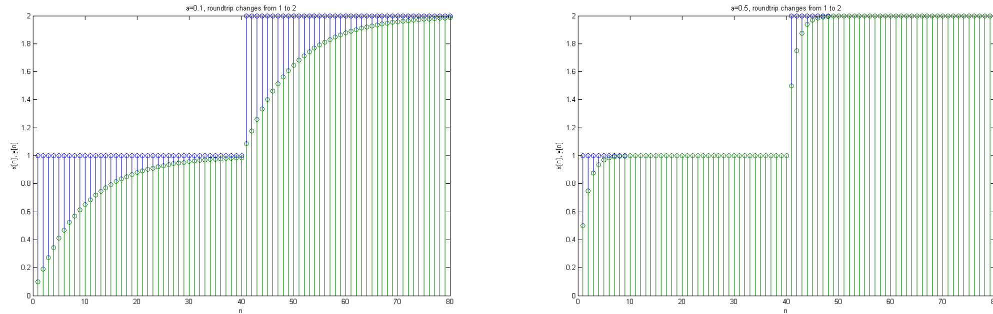


Figure 20-6: Reaction of the exponential weighted moving average filter to a persistent change (doubling) in the RTT. The smaller α (0.1, shown on the left) takes much longer to track the change, whereas the bigger value (0.5, right) responds much quicker. The output of the filter is shown in green, the input in blue.

geometrically: one can rewrite Eq. 20.1 as

$$s_n = \alpha r_n + \alpha(1-\alpha)r_{n-1} + \alpha(1-\alpha)^2 r_{n-2} + \ldots + \alpha(1-\alpha)^{n-1} r_1 + (1-\alpha)^n r_0, \qquad (20.3)$$

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.

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

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*.[3] 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!

[3]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.

■ 20.3.1 Throughput of Stop-and-Wait

We can now calculate the maximum throughput of the stop-and-wait protocol quite easily. Clearly, the maximum 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. What is the throughput of stop-and-wait in this case?

The answer clearly depends on the timeout that's used. Let's assume that the timeout is τ. To calculate the throughput, first observe that with probability $1 - \ell$, the 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 packet. We can use this property to write an expression for T, the expected time taken to send a packet and get an ACK for it:

$$T = (1 - \ell) \cdot \mathrm{RTT} + \ell(\tau + T), \tag{20.4}$$

because once the sender times out, the expected time to send a packet and get an ACK is exactly T, the number we want to calculate. Solving Equation (20.4), we find that $T = \mathrm{RTT} + \frac{\ell}{1-\ell} \cdot \tau$. The throughput is equal to $1/T$ packets per second.

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 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 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.

■ 20.4 Sliding Window Protocol

The key idea is to use a *window* of packets that are *outstanding* along the path between sender and receiver. By "outstanding", we mean "unacknowledged". The idea then is to overlap packet transmissions with ACK receptions. For our purposes, a window size of W packets means that the sender has at most W outstanding packets at any time. Our protocol will allow the sender to pick W, and the sender will try to have W outstanding 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 packets that might arrive out-of-order so that it can deliver them in order to the receiving application. This addition makes the receiver a bit more complex than before, but this complexity is worth the extra 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 packet left the

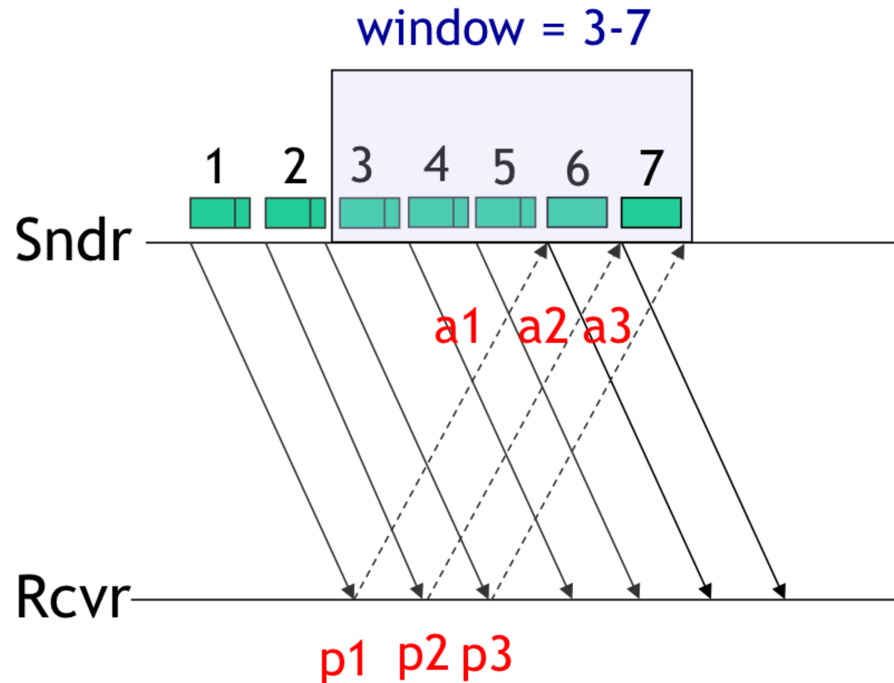


Figure 20-7: The sliding window protocol in action (W=5 here).

network, and so the sender can add another to replenish the window. This plan is shown in Figure 20-7 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 stick to the following:

> **A window size of W means that the maximum number of outstanding (unacknowledged) 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 packet every time an ACK arrives; if the ACK is for packet k and the window is W, the packet sent out has sequence number $k + W$. The receiver ACKs each 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 packet within that duration.

We now consider what happens when a packet is lost. Suppose the receiver has received packets 0 through $k - 1$ and the sender doesn't get an ACK for packet k. If the subsequent 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 packet will get sent with even higher sequence number. But somewhere in the midst of these new packet transmissions, the sender's timeout for packet k will occur, and the

sender will retransmit that packet. If that packet reaches, then it will trigger an ACK, and if that ACK reaches the sender, yet another new 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 packets outstanding as possible, but not exceeding the window size, W. If ℓ packets (or ACKs) get lost, then the effective number of outstanding packets reduces to $W - \ell$, until one of them times out, is 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 course. The sender picks a maximum window size and does not change that during a stream.

20.4.1 Sliding Window Sender

We now describe the salient features of the sender side of this protocol. The sender maintains `unacked_pkts`, a buffer of unacknowledged 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 packets were sent greater than TIMEOUT slots ago (assuming time is maintained in "slots"). If so, the sender retransmits each of these 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 packets can be sent. To do this properly, it maintains a variable, `outstanding`, which keeps track of the current number of outstanding packets. If this value is smaller than the maximum window size, the sender sends a new 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 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 packets between sender and receiver. We have described the method to do this task as follows: increment it by 1 on each new 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 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 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 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,[4] it is a good idea to maintain a counter for the number of retransmissions done for each 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 packets might be larger than the configured window, which is an error. If that occurs, and especially if a bug causes the window 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 packets does not exceed the configured window.

Third, when retransmitting a 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.

■ 20.4.2 Sliding Window Receiver

At the receiver, the biggest change to the stop-and-wait case is to maintain a list of received packets that are out-of-order. Call this list `rcvbuf`. Each 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 packets starting from `rcv_seqnum` + 1 are in `rcvbuf`. If they are, deliver them to the application, remove them from `rcvbuf`, and remember to update `rcv_seqnum`.

■ 20.4.3 Throughput

What is the throughput of the sliding window protocol? Clearly, we send at most W packets per RTT, so the throughput can't exceed 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 packet or ACK losses?

One can answer this question using a straightforward application of Little's law. W is the number of packets in the system, RTT is the mean delay of a packet (as far as the sender is concerned, since it introduces a new packet 1 RTT after some previous one in the window). What we would like is to maximize the processing rate, which of course cannot exceed the bit rate of the slowest link between the sender and receiver. 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.

This quantity, $B \cdot$ RTT is also called the *bandwidth-delay product* of the network path and is a crucial determinant of the performance of any sliding window protocol.

Given that our sliding window protocol always sends a 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 packets will continue to flow as long as some ACKs are arriving. However, as packets (or ACKs) get lost, the

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

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 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.

Finally, we also 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 packet with a particular sequence number kept getting lost, but everything else got through.

■ 20.5 Summary

This lecture discussed the key concepts involved 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 packets and acknowledgments for the receiver to signal the successful reception of a 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.

■ Acknowledgments

Thanks to Sarina Canelake for carefully proofreading and commenting on these notes.

■ Problems and Questions

1. Consider a best-effort network with variable delays and losses. Here, 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 packet $k+1$ *only after* the ACK for packet k is received. Explain whether he is correct or not.

2. ***PSet*** 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 packet $k+1$ to the receiver as soon as it receives an ACK for the packet k.
   (b) After the receiver gets the entire 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 packet size is 540 bits, and the largest 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 packet and ACK losses, for any packet size?

3. 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.

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

   We would like every 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.)

4. ***PSet*** 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 $\beta \cdot$ 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 these notes, TCP now uses the mean linear deviation as its RTO formula. Originally, TCP didn't incorporate the linear deviation in its RTO.)

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

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

have been acked)

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

If the receiver gets a 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 packets awaiting delivery *never exceeds* a value $W \geq 0$. Write down the check(s) that the sender should perform before sending a new packet in terms of the variables mentioned above that ensure the desired property.

6. 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 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 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 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-order ACK.

   (c) Suppose the window size is 10 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 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 packet and the last in-order acknowledgment received at the sender:

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

7. In the reliable transport protocols we studied, the receiver sends an acknowledgment (ACK) saying "I got k" whenever it receives a packet with sequence number k. Ben Bitdiddle invents a different method using **cumulative ACKs**: whenever the receiver gets a packet, whether in order or not, it sends an ACK saying "I got every packet up to and including ℓ", where ℓ is the **highest, in-order** 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 packets is W. Every time the sender gets an ACK, it may transmit one or more packets, within the constraint of the window size. It also implements a timeout mechanism to retransmit packets that it believes

are lost using the algorithm described in these notes. The protocol runs over a best-effort network, but *no packet or ACK is duplicated at the network or link layers*.

The sender sends a stream of new 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) ***PSet*** Now, suppose that the sender times out and retransmits the first unacknowledged packet. When the receiver gets that retransmitted 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 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:

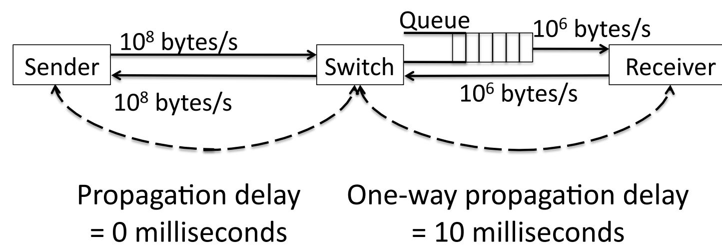
21 22 23 25 28

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

On getting ACK # $\rightarrow$ Send ??		On getting ACK # $\rightarrow$ Send ??	
21	$\rightarrow$	22	$\rightarrow$
23	$\rightarrow$	25	$\rightarrow$
28	$\rightarrow$		

8. 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 the notes and in lecture.

9. Ben Bitdiddle decides to use the sliding window transport protocol described in these notes on the network shown in Figure 20-8. 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 20-8: 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. $W = 30$: ________________________________

10. 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.