MIT 6.02 **DRAFT** *Lecture Notes*
Spring 2009 (Last update: April 29, 2009)
Comments, questions or bug reports?
Please contact Hari Balakrishnan (hari@mit.edu) or 6.02-staff@mit.edu

LECTURE 22
Reliable Data Delivery

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 (e.g., because different packets sent in sequence take different paths) and they experience variable delays (typically because of queueing). Most applications, such as Web page downloads, file transfers, and interactive terminal sessions would like a *reliable*, *in-order* stream of data. A *transport protocol* does the job of hiding the vagaries of a best-effort network, particularly lost and reordered packets, from the application.

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 significanntly improve performance.

All reliable transport protocols use the same powerful idea: use 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, and in working out exactly when retransmissions should be done, and how to achieve good performance. This lecture will cover these issues.

■ 22.1 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 packets arbitratily, reorder them arbitrarily, and delay them arbitrarily. 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] Devise mechanisms at the sending and receiving nodes to achieve

[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

what the receiver wants.

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 coding over packets to recover from packet losses. However, these mechanisms are considerably more sophisticated and complicated than the ones we will study, which use *retransmissions*. Most practical reliable data transport protocols running 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.

22.2 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. Figure 22-1 (left and middle) shows the basic operation of the protocol when packets are not lost and when data packets are lost.

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.

22.2.1 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 0), 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

packet delivery probability or bit-error rate over the link.

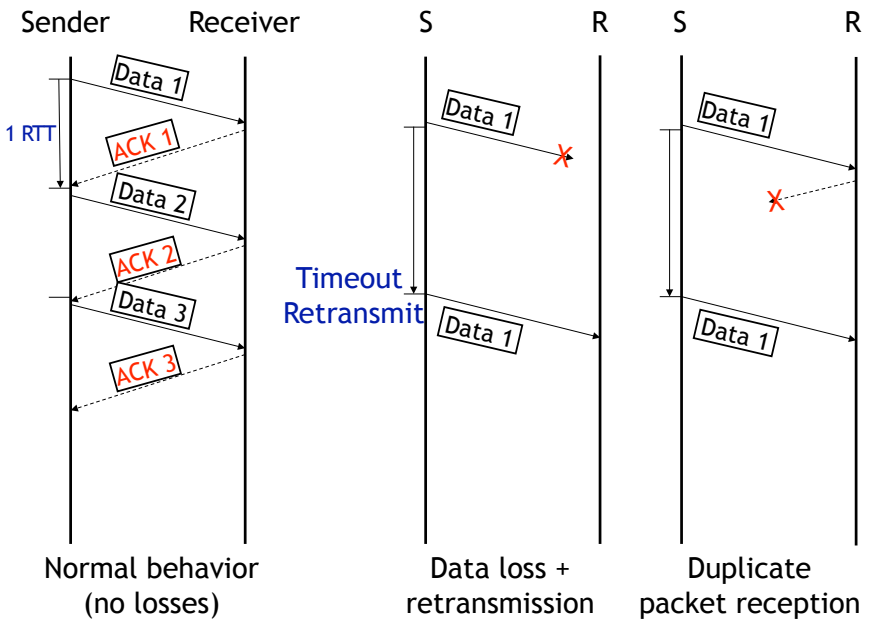

Figure 22-1: The stop-and-wait protocol. Each picture has a sender timeline and a receiver timeline. Time starts at the top of each vertical line and increases moving downward. The picture on the left shows what happens when there are no losses; the middle shows what happens on a data packet loss; and the right shows how duplicate packets may arrive at the receiver because of an ACK loss.

initial sequence number to avoid overlaps with previous instantiations of reliable streams between (incarnations of) the same communicating processes. We won't worry about these complications in this course, except to note that establishing and properly terminating connections (streams) reliably is a non-trivial problem.

■ 22.2.2 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 22-1. 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]

■ 22.2.3 Setting Timouts: Adaptive Timers

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 waiting 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* (RTT), and plays a crucial role in all reliable transport protocols. One reason is that a good value of the timeout must depend on the RTT. The other reason is that the throughput (in packets per second) achieved by these protocols (including stop-and-wait) is inversely proportional to the RTT; for example, in the absence of packet and ACK losses, the stop-and-wait protocol achieves a throghput of 1 packet per RTT.

The RTT experienced by packets is variable because the delays in our network are variable. An example is shown in Figure 22-2, 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 terms of the probability of a spurious retransmission, as shown by the tail probability in Figure 22-3 (the area under the curve to the right of the "Timeout" mark). Real-world distributions of RTT

[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.

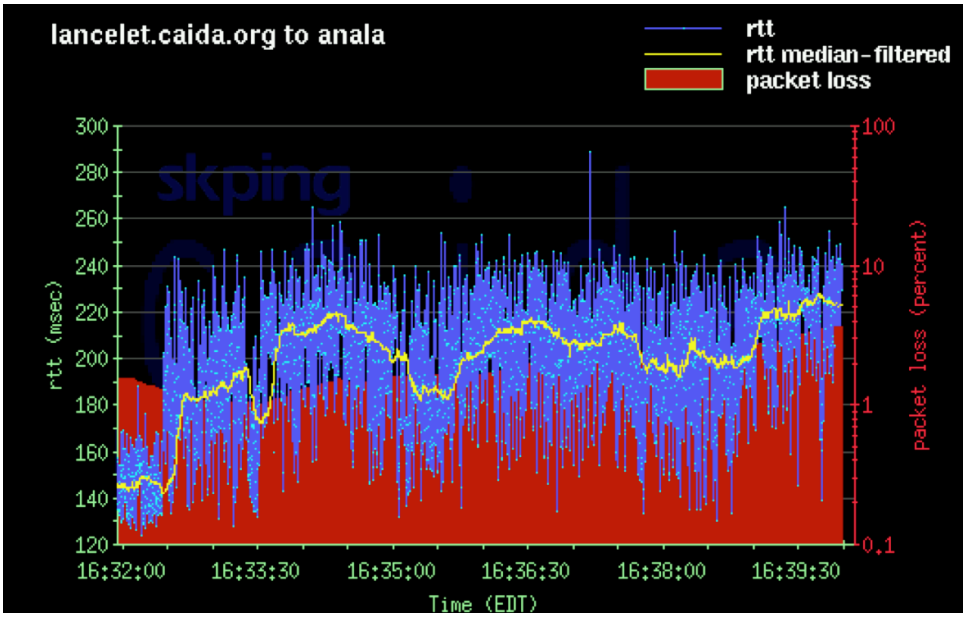

Figure 22-2: RTT variations are pronounced in many networks.

are not Gaussian, but an interesting property of all (reasonable) distributions is that if you pick a timeout threshold that is a sufficient number of standard deviations greater than the mean, the tail probability of a sample exceeding that timeout 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): $P(|X - \mu| \leq ksigma) \leq 1/k^2$. 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?

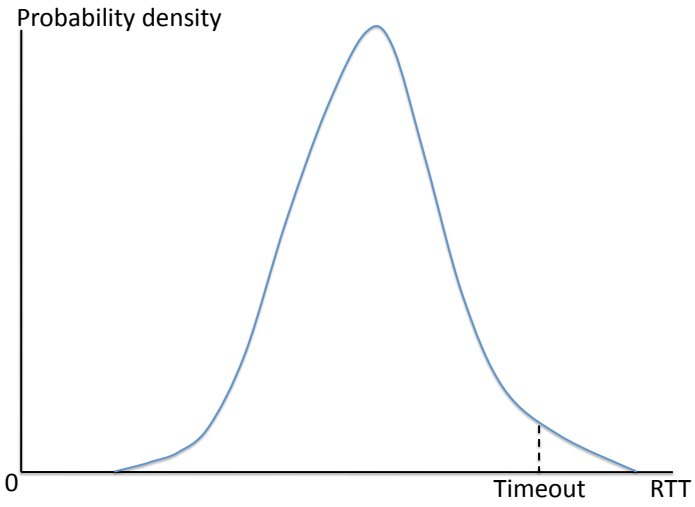

Figure 22-3: RTT variations on a wide-area cellular wireless network under high load, showing extremely high RTT values and high variability. These delays suggest a poor network design with excessively long queues that do nothing more than cause delays to be very large.

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.

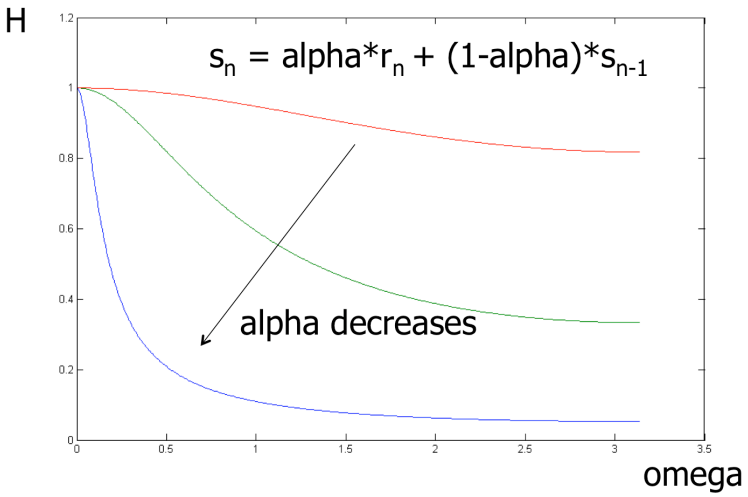

Figure 22-4: Frequency response of the exponential weighted moving average low-pass filter. As α decreases, the low-pass filter becomes even more pronounced. The graph shows the response for $\alpha = 0.9, 0.5, 0.1$, going from top to bottom.

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

$$s_n = \alpha r_n + (1 - \alpha) s_{n-1}, \tag{22.1}$$

where $0 < \alpha < 1$.

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

$$H(e^{j\Omega}) = \frac{\alpha}{1 - (1 - \alpha)z}, \tag{22.2}$$

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 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 22-4 illustrates this point.

Figure 22-5 shows how different values of α react to a sudden non-persistent change in the RTT, while Figure 22-6 shows how they react to a sudden, but persistent, change in the RTT.

Empirically, on networks prone to RTT variations due to congestion, α between 0.1 and 0.25 has been found to work well. In practice, TCP uses $\alpha = 1/8$ in its EWMA.

The specific form of Equation 22.1 is very popular in many networks and computer systems, and has a special name: *exponential weighted moving average*. 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 geometrically: one can

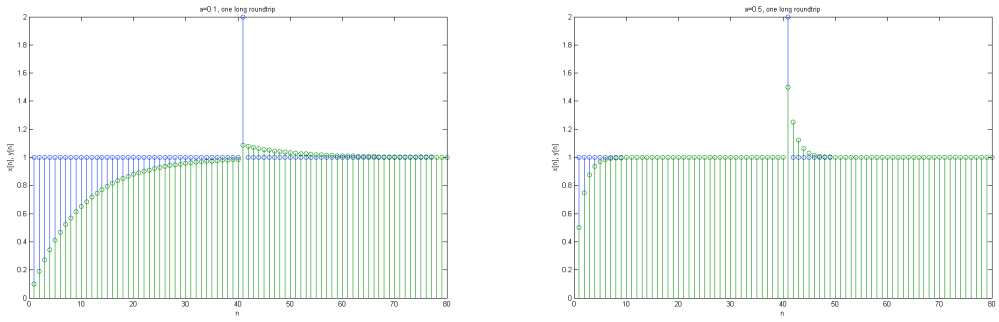

Figure 22-5: Reaction of the exponential weighted moving average filter to a non-persistent spike in the RTT (the spike is double the other samples). The smaller α (0.1, shown on the left) doesn't get swayed by it, whereas the bigger value (0.5, right) does. The output of the filter is shown in green, the input in blue.

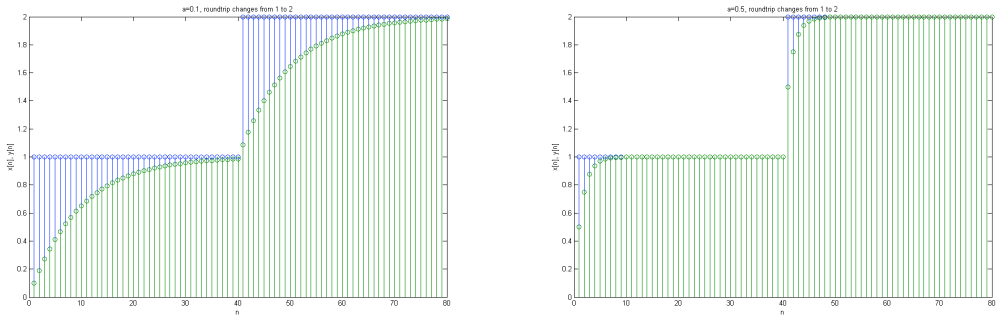

Figure 22-6: Reaction of the exponential weighted moving average filter to a persistent change (doubling) in the RTT. The smaller α (0.1, shown on the left) takes much longer to track the change, whereas the bigger value (0.5, right) responds much quicker. The output of the filter is shown in green, the input in blue.

rewrite Eq. 22.1 as

$$s_n = \alpha r_n + \alpha(1-\alpha)r_{n-1} + \alpha(1-\alpha)^2 r_{n-2} + \ldots + \alpha(1-\alpha)^{n-1} r_1 + (1-\alpha)^n r_0, \qquad (22.3)$$

observing that each successive older sample's weight is a factor of $(1-\alpha)$ "less important" than the previous one's.[3]

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.

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

[3]Note: The formula written on the blackboard in lecture on 4/29/09 for the expansion of s_n contained an algebraic error.

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 does 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 reasonably 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 Python code!

22.2.4 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.

The good thing about this 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 network 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.

The next section describes a protocol that does considerably better.

22.3 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.

[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 former has a tail probability no larger than picking a timeout equal to the mean plus k times the latter.

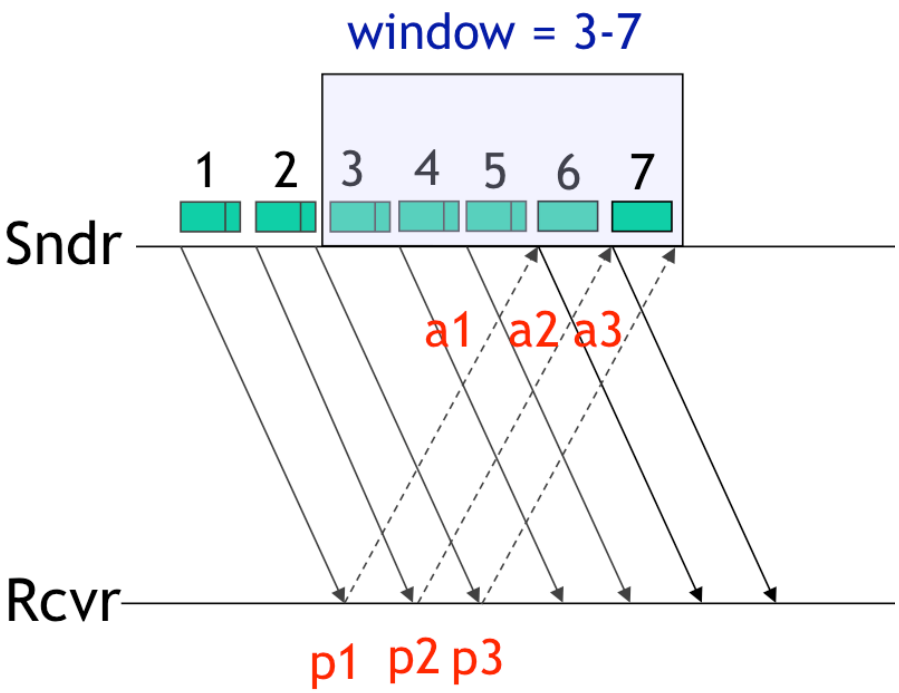

Figure 22-7: The sliding window protocol in action (W=5 here).

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 network, and so the sender can add another to replenish the window. This plan is shown in Figure 22-7 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.

■ 22.3.1 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.

In implementing a sliding window protocol, three errors are commonly encountered. 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. It is therefore

useful for the implementor 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 you're implementation shows a much larger number than this prediction, it is very likely there's a problem.

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. Therefore, it is useful to keep track of the sender's smoothed round-trip time, RTT deviation, and timeout estimates.[5] It is also 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 be very low.

■ 22.3.2 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`.

■ 22.3.3 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

[5] In Lab 10, this information will be printed when you click on the sender node.

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 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.

■ 22.4 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.