So today we're going to continue our discussion of networking.

If you remember from the last few times, we talked about these two different layers of the network stack so far.

We talked about the link layer, and we talked about the network layer.

And today were going to talk about the end to end layer.

So what we've talked about so far has been a network stack that provides this abstraction of being able to send a message from one machine to another machine across a number of links in the network.

And this network stack that we talked about is, if you will remember, a best effort network.

So a best effort network, as you'll remember, is a network that is subject to losses.

So some messages may not be properly transmitted from one point to the other point.

It's subject to the possibility of reordering of messages, as messages may take, for example, different routes through the network.

And it's subject to delays and congestion typically due to queuing within the network.

So, today what we're going to talk about it and end layer.

And the end to end layer is going to be the way that we're going to get that finally addressing some of these best effort network properties we've been kind of skirting around for the last few lectures.

Particularly, today we're going to focus on the issue of loss.

How do we avoid losses within the network?

And we'll talk a little bit about this problem of reordering.

We're going to save the discussion of delays and congestion for next time.

So the end to end layer in addition to helping us deal with these limitations of the best effort network provides a few other features that we need to mention.

So, the first thing that the end to end layer does is it provides the ability to multiplex multiple applications on top of the network.

So the network that we talked about so far is one in which there are these two endpoints, to computers that are connected to each other.

And they are transmitting a sequence of messages.

But we haven't really said anything about how those messages get dispatched to different applications that are running above the network layer.

The other thing the end to end layer provides for us is the ability to, is fragmentation of messages.

OK, and fragmentation is really about the fact that the link in itself may have some maximum size message that it can physically transmit because that's, say for example, the maximum size message is how long the sender and receiver can remain synchronized with each other.

So what the end to end layer often does is it provides the ability to take a longer message and fragment it up into smaller chunks or fragments, and it transmits each of those fragments independently as a separate message across the network.

So just to illustrate how these things are dealt with within the end to end layer, let's look at a little illustration.

So suppose we have some set of applications that are connected up to the end to end layer.

OK, so these applications, the idea is going to be that when a message arrives over the end to end layer, it's going to be dispatched to one of these applications by looking at a special number that's in the header of this message that comes in.

So, this number is often referred to as a port number.

And, each application is going to be running on one of these ports.

So oftentimes these ports are sort of running at well-known addresses.

So we talked about these numbers very briefly earlier, for example, Web servers run at port number 80 in the TCP protocol.

So if you want to contact a Web server at a particular machine, talk to port 80. Other times applications will send the port number that they're listening at in a message where two people will exchange the port numbers through some out of band protocol, say by telling your friend in an email that he should connect your server because it's running on port X.

So, messages now are going to arrive into the end to end layer from the network layer.

And these messages are going to have in their header information about which port they should be dispatched to.

So the other functionality of the end to end layer I said is fragmentation.

Then fragmentation is about taking a message that's being sent down from one of these applications into the end to end layer.

And then that message on its way to the network layer gets fragmented up into a number of chunks.

So these are each of these little chunks in this message is called a fragment.

So these are sort of, so oftentimes one common end to end layer is one that provides an abstraction that's called a stream.

So a stream is simply a flow of messages or data from one endpoint to the other, one application to the other, and where the segments in that stream are guaranteed to be delivered, are loss-free. So there are no missing segments or missing messages.

And they are in order.

So the application knows that the data that it receives is going to be in the order that the receiver knows the data it receives is going to be in the order that the sender sent it out on the channel.

And there won't be any missing messages.

So this sounds like a pretty convenient abstraction.

And it's one that's often used in applications.

And in fact, this stream abstraction is the attraction that the TCP protocol provides.

OK, so just to sort of make it clear, I just want to look quickly at a simplified end to end header format.

So we looked at the header formats at the other layers last time.

And this is just showing some of the things that you might expect to see an end to end header.

There are, of course, are additional things if you go look at the TCP header.

But these are the ones that mostly matter from the point of view of this class.

So there is a source port which specifies which port the sender of the message is listening at on the other side.

There is a destination port which specifies which port number this message should be sent to on the receiver side.

There is something called the nonce.

The nonce is just a unique identifier.

It's just a thing that uniquely identifies this message as far as the conversation between the two endpoints is concerned.

A common kind of the nonce to use is just a sequence number that gets incremented by one every time an additional message is sent.

And then, oftentimes in the end to end layer, there's also some check some information, something that allows you to verify the integrity of the messages that are transmitted out over the network.

And we do this at the end to end layer.

We saw that the checksum sometimes appeared in the link layer before.

They also appear at the end to end layer because it's possible that, as you guys read in the paper about end to end arguments, oftentimes we want to verify that the message is correct at the end to end layer or at the application layer even if we have already may be verified but that was the case at the link layer because the message could have been corrupted somewhere above just the link layer, right?

So this is sort of the abstraction that the end to end layer provides.

Notice that the header format here doesn't actually include, for example, the addresses of the endpoints, the IP addresses.

That's because the IP addresses are in the IP header.

So, remember, this is just the additional information that's added by the end to end layer, and is used to dispatch from the end to end layer to the applications that are running above it.

Once we are dealing with the end to end header, all the packets have already been transmitted across the network, and in fact we don't need to know what the IP address is anymore because this is happening on the local machine.

All of the applications are running at the same IP address.

OK, so that the other things that we said, so I said today we're going to focus mostly on the ability of the end to end layer to mitigate these problems of losses.

So this is sort of the abstraction that the end to end layer provides.

But what we want to look at now is how does the end to end layer actually deal with loss?

We're going to talk about two different techniques.

There's two different components to dealing with loss.

So the first thing we want to do is we want to make sure that no packets get lost during transmission.

And the way we're going to do that is providing what we call at least once delivery.

The reason I put at least once in quotes here is that I'm going to talk you through a protocol.

But, and this protocol is going to guarantee that a message gets received by the receiver as long as, for example, the receiver doesn't completely fail or the network doesn't completely explode.

So it's always possible that messages can be lost because there can be some physical failure that makes it impossible for the messages to get through.

But as long as it is possible for the message to get through, there's a very high probability that the message will, in fact, get through.

And, if the message doesn't get through, what this at least once delivery protocol is going to guarantee is that the receiver knows that the sender may not have actually received the message, OK?

So, and once we talk about it at least once, we're going to talk about something we call at most once delivery.

And the issue with at most once delivery is the at least once protocol that I'm going to sketch out is going to generate duplicates.

And we're going to need to make sure that we can get rid of some of the duplicates.

And these two things together are going to provide what's known as exactly once delivery of messages.

OK, so let's start off by talking about our at least once protocol.

This protocol is going to guarantee that if at all possible, the message will be received by the receiver.

And the way we're going to do this is really very straightforward.

We are just going to have the receiver send a message back to the sender that says I got the message.

So, the receiver, when it gets the message, sends what's called an acknowledgment back -- -- sometimes abbreviated ACK, indicating that the message was received at the other end.

OK, this is just going to be sent back to the receiver directly.

So in order to send this acknowledgment, though, we're going to need a way for the receiver to refer to the message that the sender sent, right?

So we want the receiver to be able to say I received this message that you sent to me.

And the simplest way to be able to do that is to just use this nonce information that's in the packets.

We said the nonce is a unique identifier as far as the two endpoints of the conversation are concerned.

So the knowledge man is basically just going to be the nonce of the message, OK?

So let's look at how this works in a simple example.

So the idea is that the sender at some point it's going to send a message to the receiver.

And this message is going to contain information like, well, it's going to have the address of the sender, the address of the receiver, the ports on both ends the message is supposed to be sent to, this nonce information that uniquely identifies the message, and then whatever the data that needs to go in the message.

Now when it sends this message, what's going to happen is that the sender is going to keep this table of pending messages.

So the sender is going to keep a list of all the messages that it hasn't yet heard acknowledged.

And then at some point, and it's going to keep, for example, if this message is, say, the name of this message is one and the nonce for this message is X, is going to add that information into its table.

Now at some point later the receiver is going to send in a knowledge man for this message one back.

And it just going to have the sender and receiver IP address, the port number of the receiver, and the nonce, which the sender is going to use in order to remove this entry from its table.

So once the sender receives an acknowledgment for a message, it no longer needs to keep any state about it because it knows that the receiver has received it.

OK, so how does this do us any good?

How is this at least once?

Well, let's see what happens when there is loss that occurs within the network?

So the idea is very simple.

Suppose that the sender sends out a message, message two this time, and that message somehow gets lost in transit through the network.

So the network drops the message either because of congestion or because some links failed and it doesn't get through.

The idea is that the sender is going to keep a timer associated with every message that it sends.

And this timer is going to be a timeout value that's going to tell the sender when it should retry transmitting this message.

And the sender is just going to retry transmitting messages over and over and over again until the message actually gets through.

So in this case it sets this timer for time TR1 at the same instant that it sends out this message.

And then when time TR1 arrives in the message hasn't yet been acknowledged, the sender is, so after this retry interval time, the sender is just going to try and retransmit the message.

So now in this case, the receiver has successfully received the message.

But notice that the sender doesn't actually know that the receiver has received it.

We can see it from the diagram, but there's been no feedback that has come from the receiver again.

And now, suppose that the receiver sends its acknowledgment for this message, and along the way the acknowledgment gets lost, right?

So this could happen just as easily as the original message being sent out.

So now in this case, our retry mechanism continues to work.

And after this retry interval and time TR2 is reached, the message gets resent, and then in this case finally the message is actually received and we can go ahead and remove from the pending message list.

So this process, the sender is just going to continually retry transmitting these messages until it gets an acknowledgment from the receiver.

Actually in practice, it's the case that the receiver will only retry a fixed number of times because as we said, there are certain situations in which the network can just be simply unavailable.

Suppose there's no network connection available to the transmitter to the sender.

Of course at some point it's going to make sense for it to give up and stop trying.

And then it will report an error to the user.

The other thing to notice about this protocol that we've described here is that the receiver has received two copies of this message.

So that seems a little bit problematic, right?

If this is a message that says withdraw $10,000 from your bank account, we don't probably want to process that message twice, right?

That might be problematic.

So we're going to address that issue when we get to talking about at most once delivery.

But just bear in mind for now that these duplicates can occur.

There is another subtlety with this protocol that I've described here, though.

Does anybody see something that's a little bit suspicious about this diagram that I've shown here, a little bit weird about the way I've shown it?

Yeah?

OK, right, good.

So there's this question about, how are you going to set the retry interval for these messages, right?

So what I've shown here is that the retry interval is short, and the first time we sent and received this message, in fact the time it took us to do that appeared to be quite long on this diagram, right?

And so in fact what would've happened if we had done this is that the sender would have retransmitted this message several times even though the receiver had actually received the message, and the acknowledgment was on the way back to us correctly.

It's just that we didn't wait long enough for that acknowledgment to come back to us.

So, there's this question about, well, how are we going to pick this timer interval so that it's appropriate for the network that we're running on.

And this turns out to be kind of an interesting and challenging problem.

So there's this question about, how long do I wait before a retry?

So a simple answer for how long we should wait is, well, whatever the round-trip time on the network is, however long it takes for a message to reach the receiver and then for the knowledge meant to be sent back to the sender; so we call that the round-trip time up or the RTT.

So, we'd like to wait at least RTT, right?

But the problem is that RTT is this round-trip time is not necessarily going to be constant over the whole lifetime of the network.

So let me show you what I mean.

This is a plot of some round-trip time information from a wide area wireless link.

So these transit times are very long here over this link, their sort of order of thousands of milliseconds.

So the average transmission time is 2.4 seconds here.

The standard deviation, which is a measure of the variance between these different samples is 1.5 seconds.

So there's a lot of bouncing around of this signal.

And so it's not as though the round-trip time; expecting the round-trip time to simply be a single constant value isn't a very good idea.

So if we want to set the timeout just for RTT, that's going to cause us, if you think about this for a minute, if we set it just to be RTT, which is say for example may be the average round-trip time that we measured in a signal like this, well, some significant proportion of the time we're going to be above the RTT, right, because just picking the average, there's going to be many samples that are above the RTT.

So instead we want to do RTT plus some slop value, some adjustment factor that gives us a little bit of extra sort of leeway in how long we wait.

But of course we don't want to wait too long because if we wait too long then we're not going to actually retransmit the messages that were in fact lost.

OK, so let's look at how, so that's sort of a simple intuitive argument for how we should do this.

What I want to just do now is just quickly take you through the way that these round-trip times are done, actually, these round-trip times are estimated in the TCP protocol.

And the way this is done is really pretty straightforward.

The idea is that we want to estimate the average RTT.

And then we also want to estimate the sort of variance which is going to be our slop number.

OK, so one way we could compute the average RTT is to keep this sort of set of samples of all the round-trip times.

So I have maybe 20 points, I have whatever it is, 20 points here that are samples of the round-trip time.

So I could take this set of 20 numbers and compute the average of them.

And then I could recompute the average every time a new number comes in.

The problem with that is that I have to keep this window of all the averages around.

So instead, what we want to do is to have some way of sort of updating the average without having to keep all the previous values around.

And there's a simple technique that's commonly used in computer systems called an exponentially weighted moving average, which is a way that we can keep track of this average with just a single number.

So this is the EWMA.

And what the EWMA does is given a set of samples, say S1 up to some most recent sample S gnu, what the EWMA does is incrementally adjust the RTT according to the following formula.

So, as the [new/gnu?] RTT is going to be equal to one minus alpha, so these samples are samples of round-trip times, OK, like this number here.

So, these are numbers that we have observed over time as messages have been transmitted back and forth.

We're going to take some number one minus alpha times S gnu, our newly observed roundtrip time.

And we're going to add to that some alpha times the old round-trip time.

OK, so what this does is basically it computes the RTT as some weighted combination of the old roundtrip time and the newly observed roundtrip time.

And, if you think about this for a minute, if we make alpha, so alpha in this case is going to be some number between zero and one.

And if you think about alpha being zero, if alpha is zero, then the newly computed round-trip time is just equal to S gnu, right?

And, if alpha is one, then the newly computed roundtrip time is just equal to whatever the old round-trip time was, right?

So, the new sample has no effect.

So, as we move very alpha between these two extremes, we are going to weight the new roundtrip time more or less heavily.

OK, so this is not going to perfectly compute the average of the samples over time.

But it's going to give us some estimate that sort of varies with time.

The other thing we said we wanted to do was compute what the slop factor is.

And the slop factor, we just want this to be some measure of the variance of this signal.

So in particular, what we want it to be is, I'm just going to push this up so I can write, slop is going to be equal to some factor beta times some variants of this, some number that the variance.

And what I mean by variance is simply the difference between the predicted and actual round-trip times.

So if our formula says that this round-trip time, given a sample, the round-trip time should be 10 ms.

And the next sample comes in and it says the actual round-trip time was 20 ms.

Then the variance, we would say that sort of this deviation, the difference between those two things would be 10 ms.

So let's see how this works.

I'll show you now the pseudocode for how this actually works in the Internet.

And it should be pretty clear what's going on.

So what we're going to do is we are going to keep a set of variables, one of which is called SRTT.

So this is almost exactly what the TCP protocol does.

So we're going to have SRTT, which is the current roundtrip time estimate.

And then we're going to have this thing we're going to call RTT [DEV?], which is the deviation, the current estimate of sort of the variance of this round-trip time.

And we're going to initialize these two numbers to be something that seems reasonable.

We might say the round-trip time is 100 ms, and this RTT DEV is 50 ms.

And now what we're going to do is every time a new one of these samples of the round-trip time comes in, we're going to call this calc RTT function.

What the calc RTT function is going to do isn't going to update the old, update the round-trip time using this formula that we've seen here.

And in the case of TCP, people have sort of experimented with different values, and sort of the number that is typically used is that a number that is commonly used is that sort of used alpha, you set alpha to be seven eighths.

So that means that you sort of add in one eighth of the new number, and use seven eighths of the old number.

So, a new number that varies a lot isn't going to change the overall round-trip time, estimate of the round-trip time terribly dramatically.

And we're going to compute the deviation as I've shown here.

We're going to take the absolute value of it.

And then, we're going to keep some running estimate of the round-trip time again using one of these sort of exponentially weighted things.

So in this case we're going to wait with this setting, the value of the weight here to three quarters.

OK, so that's a simple way to compute the round-trip time.

And now, given the computation of the round-trip time, what we need to do is to compute the timeout value that we should use.

So what we said is the timeout value you want to use is RTT plus some slop.

And, in the case of TCP, a commonly used slop value might be four times the estimate of the deviation.

See, the idea is that we want the slop value to be larger if the round-trip time varies more.

If the round-trip time is practically constant, we don't want it to vary much.

We are sort of happy; if the round-trip time is practically constant, then the timeout shouldn't be very much longer than that round-trip time because that's going to suggest we are going to be waiting longer than we need to time out.

If the round-trip time varies very dramatically, we need to wait a relatively long time in order to be sure that the message in fact has been lost as opposed to simply taking a long time for the acknowledgment to get back to us.

OK, so what we've seen so far now is we've seen how we can build up at least once semantics using acknowledgments.

And we talked about how we can go ahead and set these timers in order to allow us to calculate the round-trip time for a message.

And these timers are going to allow us to sort of decide when we should retransmit a message.

But we also saw how we have a little bit of a problem in the at least once protocol.

And the problem is that we can generate duplicates.

So in order to avoid duplicates, we need to introduce this notion of at most once.

OK, so the idea with at most once is that we want to suppress duplicates.

And duplicate suppression turns out it works a lot like the way that acknowledgments work on the receiver side or on the sender side.

So on the receiver side we're going to keep a table of all of the nonces, of all the messages that we've heard, and we're only going to process a message when we haven't already processed that message.

And we're going to tell whether we've already processed it by looking in this table of nonces.

So let's look at an example.

So here we are.

This is sort of showing you the protocol, a stage in the at least once protocol that we were in before.

So we've already sent message one.

It's been successfully acknowledged.

And you notice that we have this table of nonces, received messages, that is that the receiver.

And in this table, we have received this message with nonce X.

OK, so now when the sender starts to decide to send a message two with nonce Y, it sends it out.

The message doesn't arrive.

We time out.

We retry.

And this time the message is successfully received.

So what we do is we go ahead and add the nonce for this message into the table on the receiver.

And then the receiver goes ahead and sends the acknowledgment.

But the acknowledgment is lost.

Again, the sender times out, resends the message.

And this time, when the receiver receives this message, it's going to look it up in the receive messages table.

And it's going to see that this is a duplicate.

It already has seen a message with nonce Y.

So, it's not going to process this.

But it needs to still be sure that it sends the acknowledgment of the message.

So it doesn't actually do anything.

It doesn't actually process this message.

It doesn't pass it up to the application so the application can look at it.

But it still sends the acknowledgment so that the receiver knows that the message has been received.

OK, so this is fine.

But if you think about this for a second, this table of received messages is now just going to be kind of growing without bound, right?

Because every time we receive a message, we're going to add a new message to this table of nonces, right?

And this is a problem.

I mean if we're sending thousands of messages out over the network, then this table is going to become very, very large.

So what are we going to do about it?

Well, we're going to do sort of again the obvious thing.

We're going to have the sender send some additional information that lets the receiver know which messages the receiver has actually heard.

So a common way that this might be done is to simply, along with each message that gets sent, piggyback a little bit of information, for example that contains the list of messages that the sender knows that the receiver has actually received.

Right, so when the sender receives an acknowledgment for a message, it knows that it's never going to have to request, never going to resend the message anymore.

And so there's no reason for the receiver to keep that message in its table of received messages because it's never going to be asked to acknowledge that message again.

So we can attach a little bit of information to the messages that we send that indicates sort of which messages we have definitely completed up to this point.

OK, so this is a simple way in which we can sort of eliminate these messages that are hanging around.

These messages that are sort of left in our table of received messages are sometimes referred to as tombstones, which is kind of a funny name.

But the idea is that there are these messages that are kind of, that are sort of remnants of a dead message that's hanging around that we're never going to need to process again.

But it might just be sitting in this table.

And we are able to get rid of some of the tombstones by piggybacking this information on to the ends of the messages that we retransmit.

But you have to realize that there's always going to be a few messages left over in this receive messages table because if we use this piggybacking technique because we are piggybacking a list of done messages onto messages that are sent.

So if the sender never sends any more messages, then the receiver is never going to be able to eliminate any of the tombstones from its list of received messages.

OK, so now what we've seen is a simple way to provide this sort of notion of at least once and at most once delivery.

So as I said before, taken together, this is sometimes called an exactly once protocol.

And this variant of this protocol that we've seen is called a lockstep protocol.

OK, so it's called lockstep because the sender and receiver are operating in lockstep.

The sender sends a message, and then it waits for the receiver to send in a knowledge meant before it sends any additional messages.

Right, so we are always sort of sitting here waiting for, the sender is basically spending a lot of time idle waiting to receive an acknowledgment.

If you think about what this means in terms of the throughput of the network, it's kind of a limitation.

So let's do a simple calculation.

So suppose that we said that packets, the segments that are being sent in this network, these things that are being sent back and forth in being acknowledged are, say, 512 bytes large.

So, suppose we said that it takes the round-trip time in the network is, say, 100 ms, which might be a common roundtrip time in a traditional network, well, the throughput of this network is going to be, the maximum throughput of this network is going to be limited by this number, 512 bytes divided by 100 ms.

The round-trip time is 100 ms.

And so we have to wait 100 ms between each transmission of messages.

And the sort of size of a message is, if the message is 512 bytes, then we're going to be able to send ten of these messages per second.

So we're going to send sort of approximately 50 kb per second.

OK, that's not very fast, right?

If we want to send, modern networks are often capable of sending tens or hundreds of megabytes, megabits a second.

So we would like to be able to make this number higher.

So this is a bit of a performance problem.

And the way we're going to do this is by making it so that the sender doesn't wait to receive its acknowledgments before it goes and sends the next message.

So the sender is going to start sending the next message before it's even heard the first message even being acknowledged.

So we're going to have multiple overlapping transmissions, OK?

So let's see a really simple example of how this works.

So the idea is now that the sender, it's going to send a message.

And then before it's even heard the knowledge meant from the receiver, it's going to go ahead and start sending the message.

At the same time, the receiver can go ahead and acknowledge the messages that have already been sent.

So you sort of see what's happening here is that as time passes, the additional messages are being sent, and the acknowledgment for those messages start being sent as soon as possible.

So we have a whole bunch of messages that are kind of flying back and forth within this network.

And I've only shown the sort of yellow, red, and blue messages actually being acknowledged here.

The white messages aren't being acknowledged.

But of course there would be acknowledgments [pulling?] for those as well.

So this seems really good.

Right now we can send messages basically as fast as we can cram them onto the network.

And we sort of don't have to wait for the acknowledgments to come back anymore.

So in effect, what we've said is now the throughput is simply constrained by how fast we can cram the bytes onto the network.

But this is a little bit of an oversimplification, right, because this is sort of ignoring what it is that the receiver, suppose the sender is just sending data as fast as it can.

Well the receiver has to receive that data, has to do something with it.

It has to process it.

It has to take some action on it, right?

And so it's very possible or very likely in fact that if we cram data at the receiver in this way, that the receiver is going to become overloaded, right?

The receiver has some limited amount of data that it can buffer or that it can hold on to.

And we are going to overflow those buffers.

And we're going to create a problem.

So what we want to do is to have some way to allow the receiver to kind of throttle the transmission of the sender to ask the sender to back off a little bit, and not send so aggressively.

So we call this -- -- the technique that we're going to use is called flow control.

And it's basically just a way in which the receiver can tell the sender what great it would like to receive data at.

OK, so this is going to be sort of receiver driven feedback.

And we're going to look at two techniques basically for doing this.

The first one is a technique called fixed windows.

And the other technique is a technique called sliding windows.

OK, and what we mean by window here: a window is simply the size of the data that the receiver can accept, the number of messages that the receiver can accept -- -- can accept at one time.

So the idea is we're going to send a window's worth of data all continuously.

And then once the receiver says that it's done processing that window, we're going to be able to go ahead and send a next window to it.

So this first scheme that we're going to look at is called the fixed windows scheme.

And the idea is really very straightforward.

The sender and the receiver at the beginning of communication can negotiate.

They're going to exchange some information about what the window size is.

So the sender is going to request the connection, the open, and then the receiver is going to say, for example, OK, let's go ahead and start having this conversation, and by the way, my window size is four segments.

So now, the sender can send four segments all at once, and the receiver can go ahead and acknowledge them.

So we can have four segments that are sort of in flight at any one time.

And then after those messages have all been acknowledged, the receiver is going to have to sort of chew on those messages and process them for a little while, during which time basically the sender is simply waiting.

It's simply sitting there waiting.

But notice that we were at least able to, the sender was able to do a little bit of extra work.

It was able to send all four of these messages to simultaneously.

And then when the receiver has finished processing these messages, it's going to go ahead and ask for some additional set of messages to be transmitted out over the network.

So notice that there is a little assumption here which is that acknowledgments are being sent before the sort of receiver has actually finished processing the messages.

So the protocol that I'm showing here is that the receiver receives a message, and it immediately acknowledges it without, even though it hasn't actually, the application hasn't necessarily processed this message yet.

And the reason we want to do this is that application process, remember it's already hard enough for us to estimate this round-trip time using this EWMA approach.

And so, trying to sort of also estimate how long it would take the application to process the data would further complicate this process of setting timers.

So we usually just sending knowledge meant right away.

OK, so this is the fixed size windows scheme.

And it's nice because now basically we've set this thing up so that the sender can sort of send more messages without waiting for the acknowledgments.

But the receiver has a way that it can kind of throttle how fast the sender sends.

It knows that it only has buffers for four messages.

So it says my window size is four.

But we would like to do a little bit better, right?

In particular, we would like to avoid this sort of situation where, both so in this case we sort of have long periods where the network is kind of sitting idle, where we are not using available bandwidth within the network because we're sort of waiting.

The sender has, perhaps, data to send, but it hasn't received the message to send, the receiver can accept more messages.

And the receiver may be has processed some of the messages that it's received, but it hasn't yet sent this message; it hasn't yet asked the sender to go ahead and send any additional data.

So the way that we're going to fix this is to use something called the sliding window technique.

And the idea is that when the receiver, rather than the receiver waiting until it is processed the whole window's worth of data, when the receiver processes each message within its window, so if this window is more messages big, once it's processed the first message, it's going to go ahead and indicate that to the sender so that the sender can then go ahead and send additional messages.

So rather than sort of getting four new messages at a time, what we're going to do is we're going to send our first four messages.

And then we're going to just send one additional message at a time.

So that's why we say the window is sliding, because we're going to sort of allow the sender to send an additional message whenever it is that the receiver is finished processing just one message.

And the way we are going to do this again, we're going to initially negotiate a window size.

And then when the sender starts sending, it's going to do the same thing.

It's going to send all four of these messages.

I've sort of halted this animation in the middle of it so that I can show you an intermediate step.

But what would really be happening here is the sender would sort of send all the messages that were in the window, and then the receiver would begin processing them.

So the receiver begins processing the first message.

So suppose it finishes processing segment one before it receives this segment two from the sender.

So now what it can do is in its acknowledgment for segment two, it can piggyback again using this idea, it can stick a little bit of information on to the knowledge meant that says, oh, and by the way, I have finished processing one of these messages.

You can go ahead and send one more additional message.

You can slide the window by one, OK?

So I'm going to hit the next step of this animation.

And it's going to zip by really fast.

I'll explain what's happening, but don't try and worry too much about exactly following the details.

OK, so now what happens is that the receiver sort of continues to process these messages as it arrives, and then it continues to piggyback this information about the fact that it has finished sending some messages onto the acknowledgments, finished processing some messages onto the acknowledgments.

If the receiver doesn't have any acknowledgments to send, which is the case for these last two messages that it sent out here that I've labeled send more, it may need to send these sort of slide the window messages out by itself without acknowledgment.

So here it sends a couple of additional messages that say go ahead and send me some more data.

OK so in this way now what we've done is we've managed to make it so that the sender can send additional information before all of the sort of messages in the initial window were processed by the receiver.

So you see that before the sender sends, before the receiver actually requests, sends a message requesting a whole new window worth of data, so you see now that some of the sort of, go ahead and send more messages from the receiver arrive at the sender after the time that the sender starts sending messages, this fifth and sixth message to go ahead and processed.

So we've managed to sort of increase the amount of concurrent processing that we can do in this network.

But there still are periods where the network is sort of idle where the sender and receiver are sort of not transmitting data.

So we haven't done quite as good a job as maybe we would like.

And the reason we haven't done quite as good a job as maybe we would like is when the receiver, so the property that we would like to enforce is that when the receiver says go ahead and send me a new message.

When the receiver says slide the window by one, by the time that the next message to process arrives from the sender, we would like the receiver to not have reached the point where it's idle, where it has to wait.

So we'd like the receivers buffered to be big enough such that by the time its request for more data reaches the sender, and by the time the sender's response comes back, we would like the receiver to still have some data to process, whereas what's happened here is that the receiver finished processing all four messages in its buffer before this next message came back with additional data for the receiver to process from the sender.

Right, so basically the problem was that the receiver's buffer wasn't quite large enough for it to be able to continuously process data; it didn't have quite enough data for it to be able to continuously process while it waited for the additional data to arrive from the sender.

What this suggests that we want is to set this buffer size, to set the size of the window to be the appropriate size in order for the receiver to be able to sort of continuously process information if at all possible.

So there's this question about, how do we pick the window size?

So let's assume, so the problem we have is that small windows imply underutilization of the network, OK, because we have both the sender and receiver sort of sitting, waiting.

There's times when there's no data that's being transferred across the network.

So the question is then, how big should we make the window?

And what we said is we want the window size to be greater than the amount of time -- We want it to be long enough so that the receiver which, say for example, if the receiver can processed some number of messages, rate messages per second, OK, the receiver can process this many messages, we want its buffer to be large enough that if it processes that many messages per second, that the amount of time for additional messages for it to process, that it will still have messages to process by the time additional messages arrive from the sender.

So the amount of time it takes for additional messages to arrive from the sender from the time that this guy first since this OK to send more message, right, is one round-trip time of the network.

So it takes one round-trip time of the network for the receiver to receive additional messages to process from the sender, and a number of messages that will process during that round-trip time is whatever its message processing rate is times the round-trip time of the network.

And so, therefore, this is sort of ideally how we would set the window size in this network.

Of course, we talked about before how it's tricky to estimate this sort of EWMA thing that we do to estimate the window size is not a perfect estimator.

But we're going to try and sort of, this is going to be a good choice for a window size to use.

And so, typically the receiver is going to try and sort of use some rough estimate of what it thinks the round-trip time is and the rate at which it can process messages when it informs the sender, the window size at the initiation of the connection.

OK, so this basically is going to wrap up our discussion of the end-to-end layer, or of the sort of loss recovery issues in the end to end layer.

What we're going to talk about next time is this issue of how we deal with congestion, right?

So we said that in one of these networks, there can be all these delays due to queuing, and that these delays can introduce additional loss.

And we call these delays congestion.

And so what we're going to talk about next time is how we actually deal with congestion and how we respond to congestion.

OK so we'll see you on Wednesday.