0:00:14 So today we're going to continue our discussion of 0:00:17.025 networking. If you remember from the last 0:00:19.496 few times, we talked about these two different layers of the 0:00:23.139 network stack so far. We talked about the link layer, 0:00:26.35 and we talked about the network layer. 0:00:28.635 And today were going to talk about the end to end layer. 0:00:32.031 So what we've talked about so far has been a network stack 0:00:35.551 that provides this abstraction of being able to send a message 0:00:39.318 from one machine to another machine across a number of links 0:00:42.962 on the network. And this network stack that we 0:00:48.539 talked about is, if you will remember, 0:00:52.904 a best effort network. So a best effort network, 0:00:58.449 as you'll remember, is a network that is subject to 0:01:04.348 losses. So some messages may not be 0:01:08.503 properly transmitted from one point to the other point. 0:01:13.509 It's subject to the possibility of reordering of messages, 0:01:18.794 as messages may take, say for example, 0:01:21.854 different routes through the network. 0:01:25.192 And it's subject to delays and congestion typically due to 0:01:30.476 queuing within the network. So, today what we're going to 0:01:35.668 talk about is the end to end layer. And the end to end layer is 0:01:38.833 going to be the way that we're going to get that finally 0:01:41.998 addressing some of these best effort network properties we've 0:01:45.45 been kind of skirting around for the last few lectures. 0:01:48.557 Particularly, today we're going to focus on 0:01:50.974 the issue of loss. How do we avoid losses within 0:01:53.678 the network? And we'll talk a little bit 0:01:55.922 about this problem of reordering. 0:01:57.763 We're going to save the discussion of delays and 0:02:00.468 congestion for next time. 0:02:03 0:02:09 So the end to end layer in addition to helping us deal with 0:02:12.893 these limitations of the best effort network provides a few 0:02:16.787 other features that we need to mention. 0:02:19.337 So, the first thing that the end to end layer does is it 0:02:23.03 provides the ability to multiplex multiple applications 0:02:26.655 on top of the network. So the network that we talked 0:02:30.078 about so far is one in which there are these two endpoints, 0:02:33.972 to computers that are connected to each other. 0:02:38 And they are transmitting a sequence of messages. 0:02:40.238 But we haven't really said anything about how those 0:02:42.569 messages get dispatched to different applications that are 0:02:45.227 running above the network layer. 0:02:47 0:02:54 The other thing the end to end layer provides for us is the 0:02:57.86 ability to, is fragmentation of messages. 0:03:00.522 OK, and fragmentation is really about the fact that the link 0:03:04.649 itself may have some maximum size message that it can 0:03:08.11 physically transmit because that's, say for example, 0:03:11.504 the maximum size message is how long the sender and receiver can 0:03:15.698 remain synchronized with each other. 0:03:18.027 So what the end to end layer often does is it provides the 0:03:21.821 ability to take a longer message and fragment it up into smaller 0:03:26.014 chunks or fragments, and it transmits each of those 0:03:29.342 fragments independently as a separate message across the 0:03:33.003 network. So just to illustrate how these 0:03:37.066 things are dealt with within the end to end layer, 0:03:40.333 let's look at a little illustration. 0:03:42.666 So suppose we have some set of applications that are connected 0:03:46.733 up to the end to end layer. 0:03:49 0:03:54 OK, so these applications, the idea is going to be that 0:03:57.651 when a message arrives over the end to end layer, 0:04:00.896 it's going to be dispatched to one of these applications by 0:04:04.818 looking at a special number that's in the header of this 0:04:08.537 message that comes in. So, this number is often 0:04:11.647 referred to as a port number. And, each application is going 0:04:15.637 to be running on one of these ports. 0:04:18.003 So oftentimes these ports are sort of running at well-known 0:04:21.925 addresses. So we talked about these 0:04:24.224 numbers very briefly earlier, for example, 0:04:26.996 Web servers run at port number 80 in the TCP protocol. 0:04:32 So if you want to contact a Web server at a particular machine, 0:04:35.892 talk to port 80. Other times applications will 0:04:38.717 send the port number that they're listening at in a 0:04:41.856 message where two people will exchange the port numbers 0:04:45.246 through some out of band protocol, say by telling your 0:04:48.573 friend in an email that he should connect your server 0:04:51.838 because it's running on port X. So, messages now are going to 0:04:55.605 arrive into the end to end layer from the network layer. 0:05:00 And these messages are going to have in their header information 0:05:04.445 about which port they should be dispatched to. 0:05:07.62 So the other functionality of the end to end layer I said is 0:05:11.783 fragmentation. Then fragmentation is about 0:05:14.676 taking a message that's being sent down from one of these 0:05:18.627 applications into the end to end layer. 0:05:21.309 And then that message on its way to the network layer gets 0:05:25.33 fragmented up into a number of chunks. 0:05:29 So these are each of these little chunks in this message is 0:05:34.829 called a fragment. So these are sort of, 0:05:38.748 so oftentimes one common end to end layer is one that provides 0:05:44.879 an abstraction that's called a stream. 0:05:48.597 So a stream is simply a flow of messages or data from one 0:05:54.226 endpoint to the other, one application to the other, 0:05:59.351 and where the segments in that stream are guaranteed to be 0:06:05.08 delivered, are loss-free. So there are no missing 0:06:10.374 segments or missing messages. And they are in order. 0:06:13.421 So the application knows that the data that it receives is 0:06:16.827 going to be in the order that the receiver knows the data it 0:06:20.352 receives is going to be in the order that the sender sent it 0:06:23.877 out on the channel. And there won't be any missing 0:06:26.805 messages. So this sounds like a pretty 0:06:29.73 convenient abstraction. And it's one that's often used 0:06:33.005 in applications. And in fact, 0:06:34.735 this stream abstraction is the attraction that the TCP protocol 0:06:38.567 provides. OK, so just to sort of make it 0:06:40.977 clear, I just want to look quickly at a simplified end to 0:06:44.438 end header format. So we looked at the header 0:06:47.157 formats at the other layers last time. 0:06:49.443 And this is just showing some of the things that you might 0:06:52.966 expect to see an end to end header. 0:06:55.067 There are, of course, are additional things if you go 0:06:58.28 look at the TCP header. But these are the ones that 0:07:02.544 mostly matter from the point of view of this class. 0:07:05.404 So there is a source port which specifies which port the sender 0:07:08.95 of the message is listening at on the other side. 0:07:11.696 There is a destination port which specifies which port 0:07:14.727 number this message should be sent to on the receiver side. 0:07:18.045 There is something called the nonce. 0:07:20.047 The nonce is just a unique identifier. 0:07:22.163 It's just a thing that uniquely identifies this message as far 0:07:25.652 as the conversation between the two endpoints is concerned. 0:07:30 A common kind of the nonce to use is just a sequence number 0:07:33.164 that gets incremented by one every time an additional message 0:07:36.438 is sent. And then, oftentimes in the end 0:07:38.567 to end layer, there's also some check some 0:07:40.804 information, something that allows you to verify the 0:07:43.587 integrity of the messages that are transmitted out over the 0:07:46.752 network. And we do this at the end to 0:07:48.716 end layer. We saw that the checksum 0:07:50.571 sometimes appeared in the link layer before. 0:07:52.918 They also appear at the end to end layer because it's possible 0:07:56.246 that, as you guys read in the paper about end to end 0:07:59.029 arguments, oftentimes we want to verify that the message is 0:08:02.194 correct at the end to end layer or at the application layer even 0:08:05.632 if we have already may be verified but that was the case 0:08:08.633 at the link layer because the message could have been 0:08:11.47 corrupted somewhere above just the link layer, 0:08:13.926 right? So this is sort of the 0:08:17.208 abstraction that the end to end layer provides. 0:08:19.734 Notice that the header format here doesn't actually include, 0:08:22.974 for example, the addresses of the endpoints, 0:08:25.336 the IP addresses. That's because the IP addresses 0:08:27.972 are in the IP header. So, remember, 0:08:29.839 this is just the additional information that's added by the 0:08:33.025 end to end layer, and is used to dispatch from 0:08:35.496 the end to end layer to the applications that are running 0:08:38.572 above it. Once we are dealing with the 0:08:41.555 end to end header, all the packets have already 0:08:44.111 been transmitted across the network, and in fact we don't 0:08:47.222 need to know what the IP address is anymore because this is 0:08:50.444 happening on the local machine. All of the applications are 0:08:53.666 running at the same IP address. OK, so that the other things 0:08:56.944 that we said, so I said today we're going to 0:08:59.333 focus mostly on the ability of the end to end layer to mitigate 0:09:02.777 these problems of losses. So this is sort of the 0:09:06.595 abstraction that the end to end layer provides. 0:09:09.931 But what we want to look at now is how does the end to end layer 0:09:14.501 actually deal with loss? We're going to talk about two 0:09:18.345 different techniques. There's two different 0:09:21.391 components to dealing with loss. So the first thing we want to 0:09:25.815 do is we want to make sure that no packets get lost during 0:09:29.949 transmission. And the way we're going to do 0:09:32.995 that is providing what we call at least once delivery. 0:09:38 The reason I put at least once in quotes here is that I'm going 0:09:40.993 to talk you through a protocol. But, and this protocol is going 0:09:43.986 to guarantee that a message gets received by the receiver as long 0:09:47.075 as, for example, the receiver doesn't completely 0:09:49.344 fail or the network doesn't completely explode. 0:09:51.565 So it's always possible that messages can be lost because 0:09:54.268 there can be some physical failure that makes it impossible 0:09:57.068 for the messages to get through. But as long as it is possible 0:10:00.013 for the message to get through, there's a very high probability 0:10:03.006 that the message will, in fact, get through. 0:10:06 And, if the message doesn't get through, what this at least once 0:10:09.366 delivery protocol is going to guarantee is that the receiver 0:10:12.519 knows that the sender may not have actually received the 0:10:15.458 message, OK? So, and once we talk about it 0:10:17.648 at least once, we're going to talk about 0:10:19.732 something we call at most once delivery. 0:10:21.816 And the issue with at most once delivery is the at least once 0:10:25.022 protocol that I'm going to sketch out is going to generate 0:10:28.068 duplicates. And we're going to need to make 0:10:30.312 sure that we can get rid of some of the duplicates. 0:10:34 And these two things together are going to provide what's 0:10:38.97 known as exactly once delivery of messages. 0:10:42.698 OK, so let's start off by talking about our at least once 0:10:47.668 protocol. 0:10:49 0:10:55 This protocol is going to guarantee that if at all 0:10:57.819 possible, the message will be received by the receiver. 0:11:00.926 And the way we're going to do this is really very 0:11:03.687 straightforward. We are just going to have the 0:11:06.276 receiver send a message back to the sender that says I got the 0:11:09.786 message. So, the receiver, 0:11:11.224 when it gets the message, sends what's called an 0:11:13.928 acknowledgment back -- 0:11:16 0:11:22 -- sometimes abbreviated ACK, indicating that the message was 0:11:25.844 received at the other end. OK, this is just going to be 0:11:29.305 sent back to the receiver directly. 0:11:31.483 So in order to send this acknowledgment, 0:11:33.983 though, we're going to need a way for the receiver to refer to 0:11:37.892 the message that the sender sent, right? 0:11:40.391 So we want the receiver to be able to say I received this 0:11:43.979 message that you sent to me. And the simplest way to be able 0:11:47.76 to do that is to just use this nonce information that's in the 0:11:51.669 packets. We said the nonce is a unique 0:11:54.04 identifier as far as the two endpoints of the conversation 0:11:57.693 are concerned. So the acknowledgement is 0:12:01.567 basically just going to be the nonce of the message, 0:12:05.042 OK? So let's look at how this works 0:12:07.359 in a simple example. So the idea is that the sender 0:12:10.766 at some point is going to send a message to the receiver. 0:12:14.718 And this message is going to contain information like, 0:12:18.329 well, it's going to have the address of the sender, 0:12:21.736 the address of the receiver, the ports on both ends the 0:12:25.416 message is supposed to be sent to, this nonce information that 0:12:29.572 uniquely identifies the message, and then whatever the data that 0:12:33.865 needs to go in the message. Now when it sends this message, 0:12:38.978 what's going to happen is that the sender is going to keep this 0:12:42.936 table of pending messages. So the sender is going to keep 0:12:46.51 a list of all the messages that it hasn't yet heard 0:12:49.702 acknowledged. And then at some point, 0:12:52 and it's going to keep, for example, 0:12:54.234 if this message is, say, the name of this message 0:12:57.297 is one and the nonce for this message is X, 0:12:59.978 is going to add that information into its table. 0:13:04 Now at some point later the receiver is going to send in a 0:13:07.061 acknowledgement for this message one back. 0:13:09.157 And it just going to have the sender and receiver IP address, 0:13:12.38 the port number of the receiver, and the nonce, 0:13:14.851 which the sender is going to use in order to remove this 0:13:17.805 entry from its table. So once the sender receives an 0:13:20.545 acknowledgment for a message, it no longer needs to keep any 0:13:23.714 state about it because it knows that the receiver has received 0:13:26.991 it. OK, so how does this do us any 0:13:28.764 good? How is this at least once? 0:13:31.604 Well, let's see what happens when there is loss that occurs 0:13:35.185 within the network? So the idea is very simple. 0:13:38.024 Suppose that the sender sends out a message, 0:13:40.679 message two this time, and that message somehow gets 0:13:43.827 lost in transit through the network. 0:13:45.987 So the network drops the message either because of 0:13:49.012 congestion or because some links failed and it doesn't get 0:13:52.53 through. The idea is that the sender is 0:13:54.876 going to keep a timer associated with every message that it 0:13:58.456 sends. And this timer is going to be a 0:14:01.719 timeout value that's going to tell the sender when it should 0:14:04.99 retry transmitting this message. And the sender is just going to 0:14:08.484 retry transmitting messages over and over and over again until 0:14:11.866 the message actually gets through. 0:14:13.696 So in this case it sets this timer for time TR1 at the same 0:14:16.913 instant that it sends out this message. 0:14:19.02 And then when time TR1 arrives and the message hasn't yet been 0:14:22.347 acknowledged, the sender is, 0:14:23.844 so after this retry interval time, the sender is just going 0:14:27.06 to try and retransmit the message. 0:14:30 So now in this case, the receiver has successfully 0:14:32.55 received the message. But notice that the sender 0:14:34.997 doesn't actually know that the receiver has received it. 0:14:37.86 We can see it from the diagram, but there's been no feedback 0:14:40.931 that has come from the receiver again. 0:14:42.857 And now, suppose that the receiver sends its 0:14:45.095 acknowledgment for this message, and along the way the 0:14:47.854 acknowledgment gets lost, right? 0:14:49.468 So this could happen just as easily as the original message 0:14:52.487 being sent out. So now in this case, 0:14:54.309 our retry mechanism continues to work. 0:14:56.235 And after this retry interval and time TR2 is reached, 0:14:58.994 the message gets resent, and then in this case finally 0:15:01.753 the message is actually received and we can go ahead and remove 0:15:04.98 from the pending message list. So this process, 0:15:08.878 the sender is just going to continually retry transmitting 0:15:12.007 these messages until it gets an acknowledgment from the 0:15:14.971 receiver. Actually in practice, 0:15:16.618 it's the case that the receiver will only retry a fixed number 0:15:19.966 of times because as we said, there are certain situations in 0:15:23.205 which the network can just be simply unavailable. 0:15:25.84 Suppose there's no network connection available to the 0:15:28.749 transmitter to the sender. Of course at some point it's 0:15:32.602 going to make sense for it to give up and stop trying. 0:15:35.636 And then it will report an error to the user. 0:15:38.155 The other thing to notice about this protocol that we've 0:15:41.304 described here is that the receiver has received two copies 0:15:44.624 of this message. So that seems a little bit 0:15:47.028 problematic, right? If this is a message that says 0:15:49.833 withdraw $10,000 from your bank account, we don't probably want 0:15:53.382 to process that message twice, right? 0:15:55.443 That might be problematic. So we're going to address that 0:15:58.649 issue when we get to talking about at most once delivery. 0:16:03 But just bear in mind for now that these duplicates can occur. 0:16:06.553 There is another subtlety with this protocol that I've 0:16:09.64 described here, though. 0:16:10.922 Does anybody see something that's a little bit suspicious 0:16:14.184 about this diagram that I've shown here, a little bit weird 0:16:17.563 about the way I've shown it? Yeah? 0:16:19.485 OK, right, good. So there's this question about, 0:16:22.223 how are you going to set the retry interval for these 0:16:25.252 messages, right? So what I've shown here is that 0:16:27.99 the retry interval is short, and the first time we sent and 0:16:31.368 received this message, in fact the time it took us to 0:16:34.398 do that appeared to be quite long on this diagram, 0:16:37.252 right? And so in fact what would've 0:16:40.452 happened if we had done this is that the sender would have 0:16:43.408 retransmitted this message several times even though the 0:16:46.261 receiver had actually received the message, and the 0:16:48.854 acknowledgment was on the way back to us correctly. 0:16:51.448 It's just that we didn't wait long enough for that 0:16:53.989 acknowledgment to come back to us. 0:16:55.701 So, there's this question about, well, how are we going to 0:16:58.657 pick this timer interval so that it's appropriate for the network 0:17:01.977 that we're running on. And this turns out to be kind 0:17:06.416 of an interesting and challenging problem. 0:17:10 0:17:16 So there's this question about, how long to wait before a 0:17:21.272 retry? 0:17:22 0:17:28 So a simple answer for how long we should wait is, 0:17:31.197 well, whatever the round-trip time on the network is, 0:17:34.59 however long it takes for a message to reach the receiver 0:17:38.245 and then for the acknowledgement to be sent back to the sender; 0:17:42.291 so we call that the round-trip time up or the RTT. 0:17:45.488 So, we'd like to wait at least RTT, right? 0:17:48.164 But the problem is that RTT is this round-trip time is not 0:17:51.883 necessarily going to be constant over the whole lifetime of the 0:17:55.929 network. So let me show you what I mean. 0:17:58.474 This is a plot of some round-trip time information from 0:18:01.998 a wide area wireless link. So these transit times are very 0:18:07.165 long here over this link, their sort of order of 0:18:10.449 thousands of milliseconds. So the average transmission 0:18:14.152 time is 2.4 seconds here. The standard deviation, 0:18:17.506 which is a measure of the variance between these different 0:18:21.489 samples is 1.5 seconds. So there's a lot of bouncing 0:18:25.052 around of this signal. And so it's not as though the 0:18:28.615 round-trip time; expecting the round-trip time 0:18:31.759 to simply be a single constant value isn't a very good idea. 0:18:37 So if we want to set the timeout just for RTT, 0:18:40.677 that's going to cause us, if you think about this for a 0:18:45.091 minute, if we set it just to be RTT, which is say for example 0:18:49.995 may be the average round-trip time that we measured in a 0:18:54.49 signal like this, well, some significant 0:18:57.677 proportion of the time we're going to be above the RTT, 0:19:02.091 right, because just picking the average, there's going to be 0:19:06.913 many samples that are above the RTT. 0:19:11 So instead we want to do RTT plus some slop value, 0:19:14.113 some adjustment factor that gives us a little bit of extra 0:19:17.735 sort of leeway in how long we wait. 0:19:19.896 But of course we don't want to wait too long because if we wait 0:19:23.836 too long then we're not going to actually retransmit the messages 0:19:27.903 that were in fact lost. OK, so let's look at how, 0:19:32.012 so that's sort of a simple intuitive argument for how we 0:19:36.441 should do this. What I want to just do now is 0:19:39.984 just quickly take you through the way that these round-trip 0:19:44.654 times are done, actually, these round-trip 0:19:47.955 times are estimated in the TCP protocol. 0:19:51.096 And the way this is done is really pretty straightforward. 0:19:55.685 The idea is that we want to estimate the average RTT. 0:20:01 And then we also want to estimate the sort of variance 0:20:03.825 which is going to be our slop number. 0:20:05.744 OK, so one way we could compute the average RTT is to keep this

0:20:09.049 sort of set of samples of all the round-trip times. 0:20:11.715 So I have maybe 20 points, I have whatever it is, 0:20:14.273 20 points here that are samples of the round-trip time. 0:20:17.152 So I could take this set of 20 numbers and compute the average 0:20:20.404 of them. And then I could recompute the 0:20:22.43 average every time a new number comes in. 0:20:24.562 The problem with that is that I have to keep this window of all 0:20:27.867 the averages around. So instead, what we want to do 0:20:32.384 is to have some way of sort of updating the average without 0:20:36.994 having to keep all the previous values around. 0:20:40.571 And there's a simple technique that's commonly used in computer 0:20:45.499 systems called an exponentially weighted moving average, 0:20:49.87 which is a way that we can keep track of this average with just 0:20:54.798 a single number. So this is the EWMA. 0:20:57.659 And what the EWMA does is given a set of samples, 0:21:01.475 say S1 up to some most recent sample S new, 0:21:04.813 what the EWMA does is incrementally adjust the RTT 0:21:08.707 according to the following formula. 0:21:13 So, as the new RTT is going to be equal to one minus 0:21:16.991 alpha, so these samples are samples of round-trip times, 0:21:20.777 OK, like this number here. So, these are numbers that we 0:21:24.562 have observed over time as messages have been transmitted 0:21:28.417 back and forth. We're going to take some number 0:21:32.623 one minus alpha times S new, our newly observed roundtrip 0:21:37.363 time. And we're going to add to that 0:21:40.325 some alpha times the old round-trip time. 0:21:43.71 OK, so what this does is basically it computes the RTT as 0:21:48.449 some weighted combination of the old roundtrip time and the newly 0:21:53.866 observed roundtrip time. And, if you think about this 0:21:58.267 for a minute, if we make alpha, 0:22:00.806 so alpha in this case is going to be some number between zero 0:22:05.884 and one. And if you think about alpha 0:22:09.784 being zero, if alpha is zero, then the newly computed 0:22:13.098 round-trip time is just equal to S new, right? 0:22:15.965 And, if alpha is one, then the newly computed 0:22:18.769 roundtrip time is just equal to whatever the old round-trip time 0:22:22.784 was, right? So, the new sample has no 0:22:25.078 effect. So, as we move, vary alpha 0:22:27.117 between these two extremes, we are going to weight the new 0:22:30.75 roundtrip time more or less heavily. 0:22:34 OK, so this is not going to perfectly compute the average of 0:22:37.986 the samples over time. But it's going to give us some 0:22:41.5 estimate that sort of varies with time. 0:22:44.067 The other thing we said we wanted to do was compute what 0:22:47.783 the slop factor is. And the slop factor, 0:22:50.418 we just want this to be some measure of the variance of this 0:22:54.405 signal. So in particular, 0:22:56.027 what we want it to be is, I'm just going to push this up 0:22:59.743 so I can write, slop is going to be equal to 0:23:02.648 some factor beta times some variance of this, 0:23:05.621 some number that the variance. And what I mean by variance is 0:23:12.786 simply the difference between the predicted and actual 0:23:19.475 round-trip times. So if our formula says that 0:23:25.029 this round-trip time, given a sample, 0:23:29.572 the round-trip time should be 10 ms. 0:23:35 And the next sample comes in and it says the actual 0:23:37.79 round-trip time was 20 ms. Then the variance, 0:23:40.245 we would say that sort of this deviation, the difference 0:23:43.314 between those two things would be 10 ms. 0:23:45.491 So let's see how this works. I'll show you now the 0:23:48.225 pseudocode for how this actually works in the Internet. 0:23:51.238 And it should be pretty clear what's going on. 0:23:53.75 So what we're going to do is we are going to keep a set of 0:23:56.93 variables, one of which is called SRTT. 0:24:00 So this is almost exactly what the TCP protocol does. 0:24:02.855 So we're going to have SRTT, which is the current roundtrip 0:24:06.039 time estimate. And then we're going to have 0:24:08.346 this thing we're going to call RTTDEV, which is the 0:24:11.311 deviation, the current estimate of sort of the variance of this 0:24:14.715 round-trip time. And we're going to initialize 0:24:17.186 these two numbers to be something that seems reasonable. 0:24:20.206 We might say the round-trip time is 100 ms, 0:24:22.512 and this RTTDEV is 50 ms. And now what we're going to do 0:24:25.587 is every time a new one of these samples of the round-trip time 0:24:28.991 comes in, we're going to call this calc RTT function. 0:24:33 What the calc RTT function is going to do isn't going to 0:24:37.035 update the old, update the round-trip time 0:24:40.043 using this formula that we've seen here. 0:24:42.904 And in the case of TCP, people have sort of 0:24:45.986 experimented with different values, and sort of the number 0:24:50.168 that is typically used is that a number that is commonly used is 0:24:54.79 that sort of used alpha, you set alpha to be seven 0:24:58.385 eighths. So that means that you sort of 0:25:01.705 add in one eighth of the new number, and use seven eighths of 0:25:05.117 the old number. So, a new number that varies a 0:25:07.675 lot isn't going to change the overall round-trip time, 0:25:10.688 estimate of the round-trip time terribly dramatically. 0:25:13.702 And we're going to compute the deviation as I've shown here. 0:25:17.056 We're going to take the absolute value of it. 0:25:19.558 And then, we're going to keep some running estimate of the 0:25:22.799 round-trip time again using one of these sort of exponentially 0:25:26.267 weighted things. So in this case we're going to 0:25:28.882 weight with this setting, the value of the weight here to 0:25:31.953 three quarters. OK, so that's a simple way to 0:25:35.569 compute the round-trip time. And now, given the computation 0:25:38.707 of the round-trip time, what we need to do is to 0:25:41.25 compute the timeout value that we should use. 0:25:43.63 So what we said is the timeout value you want to use is RTT 0:25:46.768 plus some slop. And, in the case of TCP, 0:25:48.878 a commonly used slop value might be four times the estimate 0:25:52.016 of the deviation. See, the idea is that we want 0:25:54.505 the slop value to be larger if the round-trip time varies more. 0:25:57.86 If the round-trip time is practically constant, 0:26:00.348 we don't want it to vary much. We are sort of happy; 0:26:04.496 if the round-trip time is practically constant, 0:26:07.775 then the timeout shouldn't be very much longer than that 0:26:11.695 round-trip time because that's going to suggest we are going to 0:26:16.114 be waiting longer than we need to time out. 0:26:19.107 If the round-trip time varies very dramatically, 0:26:22.457 we need to wait a relatively long time in order to be sure 0:26:26.52 that the message in fact has been lost as opposed to simply 0:26:30.654 taking a long time for the acknowledgment to get back to 0:26:34.574 us. OK, so what we've seen so far 0:26:38.23 now is we've seen how we can build up at least once semantics 0:26:42.846 using acknowledgments. And we talked about how we can 0:26:46.846 go ahead and set these timers in order to allow us to calculate 0:26:51.615 the round-trip time for a message. 0:26:54.153 And these timers are going to allow us to sort of decide when 0:26:58.769 we should retransmit a message. But we also saw how we have a 0:27:04.276 little bit of a problem in the at least once protocol. 0:27:08.436 And the problem is that we can generate duplicates. 0:27:12.361 So in order to avoid duplicates, we need to introduce 0:27:16.443 this notion of at most once. OK, so the idea with at most 0:27:20.839 once is that we want to suppress duplicates. 0:27:25 0:27:35 And duplicate suppression turns out it works a lot like the way 0:27:38.893 that acknowledgments work on the receiver side or on the sender 0:27:42.786 side. So on the receiver side we're 0:27:44.92 going to keep a table of all of the nonces, of all the messages 0:27:48.813 that we've heard, and we're only going to process 0:27:51.827 a message when we haven't already processed that message. 0:27:55.344 And we're going to tell whether we've already processed it by 0:27:59.111 looking in this table of nonces. So let's look at an example. 0:28:03.882 So here we are. This is sort of showing you the 0:28:06.974 protocol, a stage in the at least once protocol that we were 0:28:10.941 in before. So we've already sent message 0:28:13.563 one. It's been successfully 0:28:15.31 acknowledged. And you notice that we have 0:28:18 this table of nonces, received messages, 0:28:20.621 that is at the receiver. And in this table, 0:28:23.579 we have received this message with nonce X. 0:28:26.403 OK, so now when the sender starts to decide to send a 0:28:29.899 message two with nonce Y, it sends it out. 0:28:34 The message doesn't arrive. We time out. 0:28:36.657 We retry. And this time the message is 0:28:39.178 successfully received. So what we do is we go ahead 0:28:42.584 and add the nonce for this message into the table on the 0:28:46.331 receiver. And then the receiver goes 0:28:48.716 ahead and sends the acknowledgment. 0:28:51.032 But the acknowledgment is lost. Again, the sender times out, 0:28:55.052 resends the message. And this time, 0:28:57.369 when the receiver receives this message, it's going to look it 0:29:01.525 up in the receive messages table. 0:29:05 And it's going to see that this is a duplicate. 0:29:07.53 It already has seen a message with nonce Y. 0:29:09.841 So, it's not going to process this. 0:29:11.711 But it needs to still be sure that it sends the acknowledgment 0:29:15.067 of the message. So it doesn't actually do 0:29:17.268 anything. It doesn't actually process 0:29:19.249 this message. It doesn't pass it up to the 0:29:21.504 application so the application can look at it. 0:29:23.98 But it still sends the acknowledgment so that the 0:29:26.621 receiver knows that the message has been received. 0:29:29.317 OK, so this is fine. But if you think about this for 0:29:32.122 a second, this table of received messages is now just going to be 0:29:35.644 kind of growing without bound, right? 0:29:39 Because every time we receive a message, we're going to add a 0:29:42.848 new message to this table of nonces, right? 0:29:45.542 And this is a problem. I mean if we're sending 0:29:48.428 thousands of messages out over the network, then this table is 0:29:52.341 going to become very, very large. 0:29:54.393 So what are we going to do about it? 0:29:56.638 Well, we're going to do sort of again the obvious thing. 0:30:01 We're going to have the sender send some additional information 0:30:04.761 that lets the receiver know which messages the receiver has 0:30:08.279 actually heard. So a common way that this might 0:30:11.069 be done is to simply, along with each message that 0:30:14.042 gets sent, piggyback a little bit of information, 0:30:16.954 for example that contains the list of messages that the sender 0:30:20.654 knows that the receiver has actually received. 0:30:23.384 Right, so when the sender receives an acknowledgment for a 0:30:26.841 message, it knows that it's never going to have to request, 0:30:30.36 never going to resend the message anymore. 0:30:34 And so there's no reason for the receiver to keep that 0:30:37.459 message in its table of received messages because it's never 0:30:41.31 going to be asked to acknowledge that message again. 0:30:44.638 So we can attach a little bit of information to the messages 0:30:48.489 that we send that indicates sort of which messages we have 0:30:52.209 definitely completed up to this point. 0:30:54.624 OK, so this is a simple way in which we can sort of eliminate 0:30:58.54 these messages that are hanging around. 0:31:02 These messages that are sort of left in our table of received 0:31:05.705 messages are sometimes referred to as tombstones, 0:31:08.669 which is kind of a funny name. But the idea is that there are 0:31:12.374 these messages that are kind of, that are sort of remnants of a 0:31:16.203 dead message that's hanging around that we're never going to 0:31:19.846 need to process again. But it might just be sitting in 0:31:23.119 this table. And we are able to get rid of 0:31:25.589 some of the tombstones by piggybacking this information on 0:31:29.109 to the ends of the messages that we retransmit. 0:31:33 But you have to realize that there's always going to be a few 0:31:37.039 messages left over in this receive messages table because 0:31:40.81 if we use this piggybacking technique because we are 0:31:44.244 piggybacking a list of done messages onto messages that are 0:31:48.149 sent. So if the sender never sends 0:31:50.371 any more messages, then the receiver is never 0:31:53.334 going to be able to eliminate any of the tombstones from its 0:31:57.306 list of received messages. OK, so now what we've seen is a 0:32:03.158 simple way to provide this sort of notion of at least once and 0:32:09.372 at most once delivery. So as I said before, 0:32:13.651 taken together, this is sometimes called an 0:32:17.93 exactly once protocol. And this variant of this 0:32:22.616 protocol that we've seen is called a lockstep protocol. 0:32:28.117 OK, so it's called lockstep because the sender and receiver 0:32:34.026 are operating in lockstep. The sender sends a message, 0:32:40.103 and then it waits for the receiver to send an acknowledgement 0:32:44.467 before it sends any additional messages. 0:32:47.974 Right, so we are always sort of sitting here waiting for, 0:32:52.337 the sender is basically spending a lot of time idle 0:32:56.233 waiting to receive an acknowledgment. 0:32:59.038 If you think about what this means in terms of the throughput 0:33:03.714 of the network, it's kind of a limitation. 0:33:08 So let's do a simple calculation. 0:33:10.419 So suppose that we said that packets, the segments that are 0:33:14.805 being sent in this network, these things that are being 0:33:18.888 sent back and forth in being acknowledged are, 0:33:22.291 say, 512 bytes large. So, suppose we said that it 0:33:25.92 takes the round-trip time in the network is, say, 0:33:29.55 100 ms, which might be a common roundtrip time in a traditional 0:33:34.238 network, well, the throughput of this network 0:33:37.565 is going to be, the maximum throughput of this 0:33:40.967 network is going to be limited by this number, 0:33:44.37 512 bytes divided by 100 ms. The round-trip time is 100 ms. 0:33:50.033 And so we have to wait 100 ms between each transmission of 0:33:53.898 messages. And the sort of size of a 0:33:56.203 message is, if the message is 512 bytes, then we're going to 0:34:00.203 be able to send 10 of these messages per second. 0:34:04 So we're going to send sort of approximately 50 kb per second. 0:34:08.935 OK, that's not very fast, right? 0:34:11.443 If we want to send, modern networks are often 0:34:15.003 capable of sending tens or hundreds of megabytes, 0:34:18.886 megabits a second. So we would like to be able to 0:34:22.77 make this number higher. So this is a bit of a 0:34:26.411 performance problem. 0:34:29 0:34:34 And the way we're going to do this is by making it so that the 0:34:39.35 sender doesn't wait to receive its acknowledgments before it 0:34:44.526 goes and sends the next message. So the sender is going to start 0:34:50.052 sending the next message before it's even heard the first 0:34:54.964 message even being acknowledged. So we're going to have multiple 0:35:00.491 overlapping transmissions, OK? 0:35:04 So let's see a really simple example of how this works. 0:35:07.14 So the idea is now that the sender, it's going to send a 0:35:10.339 message. And then before it's even heard 0:35:12.607 the acknowledgement from the receiver, it's going to go ahead 0:35:16.097 and start sending the message. At the same time, 0:35:18.831 the receiver can go ahead and acknowledge the messages that 0:35:22.204 have already been sent. So you sort of see what's 0:35:24.996 happening here is that as time passes, the additional messages 0:35:28.544 are being sent, and the acknowledgment for 0:35:30.928 those messages start being sent as soon as possible. 0:35:35 So we have a whole bunch of messages that are kind of flying 0:35:38.6 back and forth within this network. 0:35:40.675 And I've only shown the sort of yellow, red, and blue messages 0:35:44.397 actually being acknowledged here. 0:35:46.35 The white messages aren't being acknowledged. 0:35:49.035 But of course there would be acknowledgments pulling for 0:35:52.574 those as well. So this seems really good. 0:35:55.015 Right now we can send messages basically as fast as we can cram 0:35:58.799 them onto the network. And we sort of don't have to 0:36:01.85 wait for the acknowledgments to come back anymore. 0:36:06 So in effect, what we've said is now the 0:36:08.122 throughput is simply constrained by how fast we can cram the 0:36:11.333 bytes onto the network. But this is a little bit of an 0:36:14.217 oversimplification, right, because this is sort of 0:36:16.884 ignoring what it is that the receiver, suppose the sender is 0:36:20.095 just sending data as fast as it can. 0:36:22 Well the receiver has to receive that data, 0:36:24.285 has to do something with it. It has to process it. 0:36:26.952 It has to take some action on it, right? 0:36:30 And so it's very possible or very likely in fact that if we 0:36:33.584 cram data at the receiver in this way, that the receiver is 0:36:37.168 going to become overloaded, right? 0:36:39.207 The receiver has some limited amount of data that it can 0:36:42.606 buffer or that it can hold on to. 0:36:44.584 And we are going to overflow those buffers. 0:36:47.179 And we're going to create a problem. 0:36:49.342 So what we want to do is to have some way to allow the 0:36:52.617 receiver to kind of throttle the transmission of the sender to 0:36:56.387 ask the sender to back off a little bit, and not send so 0:36:59.786 aggressively. So we call this -- 0:37:03 0:37:07 -- the technique that we're going to use is called flow 0:37:10.58 control. And it's basically just a way 0:37:13.033 in which the receiver can tell the sender what rate it would 0:37:17.011 like to receive data at. 0:37:19 0:37:26 OK, so this is going to be sort of receiver driven feedback. 0:37:32.571 And we're going to look at two techniques basically for doing 0:37:39.254 this. The first one is a technique 0:37:42.93 called fixed windows. And the other technique is a 0:37:48.388 technique called sliding windows. 0:37:51.952 OK, and what we mean by window here: a window is simply the 0:37:58.413 size of the data that the receiver can accept, 0:38:03.425 the number of messages that the receiver can accept -- 0:38:11 0:38:22 -- can accept at one time. So the idea is we're going to 0:38:27.817 send a window's worth of data all continuously. 0:38:33 And then once the receiver says that it's done processing that 0:38:36.959 window, we're going to be able to go ahead and send a next 0:38:40.658 window to it. So this first scheme that we're 0:38:43.514 going to look at is called the fixed windows scheme. 0:38:46.824 And the idea is really very straightforward. 0:38:49.615 The sender and the receiver at the beginning of communication 0:38:53.509 are going to negotiate. They're going to exchange some 0:38:56.365 information about what the window size is. 0:39:00 So the sender is going to request the connection, 0:39:02.98 the open, and then the receiver is going to say, 0:39:05.898 for example, OK, let's go ahead and start 0:39:08.382 having this conversation, and by the way, 0:39:10.866 my window size is four segments. 0:39:12.79 So now, the sender can send four segments all at once, 0:39:16.081 and the receiver can go ahead and acknowledge them. 0:39:19.186 So we can have four segments that are sort of in flight at 0:39:22.725 any one time. And then after those messages 0:39:25.333 have all been acknowledged, the receiver is going to have 0:39:28.81 to sort of chew on those messages and process them for a 0:39:32.225 little while, during which time basically the 0:39:34.957 sender is simply waiting. It's simply sitting there 0:39:39.391 waiting. But notice that we were at 0:39:41.283 least able to, the sender was able to do a 0:39:43.565 little bit of extra work. It was able to send all four of 0:39:46.682 these messages to simultaneously. 0:39:48.463 And then when the receiver has finished processing these 0:39:51.525 messages, it's going to go ahead and ask for some additional set 0:39:55.031 of messages to be transmitted out over the network. 0:39:57.814 So notice that there is a little assumption here which is 0:40:00.931 that acknowledgments are being sent before the sort of receiver 0:40:04.382 has actually finished processing the messages. 0:40:08 So the protocol that I'm showing here is that the 0:40:11.031 receiver receives a message, and it immediately acknowledges 0:40:14.757 it without, even though it hasn't actually, 0:40:17.41 the application hasn't necessarily processed this 0:40:20.442 message yet. And the reason we want to do 0:40:22.968 this is that application process, remember it's already 0:40:26.378 hard enough for us to estimate this round-trip time using this 0:40:30.231 EWMA approach. And so, trying to sort of also 0:40:33.546 estimate how long it would take the application to process the 0:40:36.692 data would further complicate this process of setting timers. 0:40:39.786 So we usually just send acknowledgements right away. 0:40:42.519 OK, so this is the fixed size windows scheme. 0:40:44.788 And it's nice because now basically we've set this thing 0:40:47.624 up so that the sender can sort of send more messages without 0:40:50.666 waiting for the acknowledgments. But the receiver has a way that 0:40:53.915 it can kind of throttle how fast the sender sends. 0:40:56.441 It knows that it only has buffers for four messages. 0:41:00 So it says my window size is four. 0:41:02.142 But we would like to do a little bit better, 0:41:04.935 right? In particular, 0:41:06.233 we would like to avoid this sort of situation where, 0:41:09.545 both so in this case we sort of have long periods where the 0:41:13.311 network is kind of sitting idle, where we are not using 0:41:16.818 available bandwidth within the network because we're sort of 0:41:20.649 waiting. The sender has, 0:41:22.142 perhaps, data to send, but it hasn't received the 0:41:25.259 message to send, the receiver can accept more 0:41:28.116 messages. And the receiver may be has 0:41:31.519 processed some of the messages that it's received, 0:41:34.278 but it hasn't yet sent this message; it hasn't yet asked the 0:41:37.599 sender to go ahead and send any additional data. 0:41:40.245 So the way that we're going to fix this is to use something 0:41:43.511 called the sliding window technique. 0:41:45.481 And the idea is that when the receiver, rather than the 0:41:48.521 receiver waiting until it has processed the whole window's 0:41:51.674 worth of data, when the receiver processes 0:41:53.982 each message within its window, so if this window is four 0:41:57.134 messages big, once it's processed the first 0:41:59.499 message, it's going to go ahead and indicate that to the sender 0:42:02.989 so that the sender can then go ahead and send additional 0:42:06.085 messages. So rather than sort of getting 0:42:09.675 four new messages at a time, what we're going to do is we're 0:42:12.969 going to send our first four messages. 0:42:15.035 And then we're going to just send one additional message at a 0:42:18.385 time. So that's why we say the window 0:42:20.395 is sliding, because we're going to sort of allow the sender to 0:42:23.802 send an additional message whenever it is that the receiver 0:42:27.04 is finished processing just one message. 0:42:30 And the way we are going to do this again, we're going to 0:42:33.372 initially negotiate a window size. 0:42:35.359 And then when the sender starts sending, it's going to do the 0:42:38.973 same thing. It's going to send all four of 0:42:41.442 these messages. I've sort of halted this 0:42:43.79 animation in the middle of it so that I can show you an 0:42:47.042 intermediate step. But what would really be 0:42:49.572 happening here is the sender would sort of send all the 0:42:52.824 messages that were in the window, and then the receiver 0:42:56.076 would begin processing them. So the receiver begins 0:42:59.087 processing the first message. So suppose it finishes 0:43:02.158 processing segment one before it receives this segment two from 0:43:05.892 the sender. So now what it can do is in its 0:43:09.897 acknowledgment for segment two, it can piggyback again using 0:43:13.51 this idea, it can stick a little bit of information on to the 0:43:17.183 acknowledgement that says, oh, and by the way, 0:43:19.938 I have finished processing one of these messages. 0:43:22.877 You can go ahead and send one more additional message. 0:43:26.122 You can slide the window by one, OK? 0:43:29 So I'm going to hit the next step of this animation. 0:43:32.427 And it's going to zip by really fast. 0:43:34.847 I'll explain what's happening, but don't try and worry too 0:43:38.678 much about exactly following the details. 0:43:41.366 OK, so now what happens is that the receiver sort of continues 0:43:45.466 to process these messages as it arrives, and then it continues 0:43:49.566 to piggyback this information about the fact that it has 0:43:53.262 finished sending some messages onto the acknowledgments, 0:43:56.959 finished processing some messages onto the 0:43:59.714 acknowledgments. If the receiver doesn't have 0:44:03.772 any acknowledgments to send, which is the case for these 0:44:07.253 last two messages that it sent out here that I've labeled send 0:44:11.113 more, it may need to send these sort of slide the window 0:44:14.594 messages out by itself without acknowledgments. 0:44:17.443 So here it sends a couple of additional messages that say go 0:44:21.177 ahead and send me some more data. 0:44:23.202 OK so in this way now what we've done is we've managed to 0:44:26.746 make it so that the sender can send additional information 0:44:30.354 before all of the sort of messages in the initial window 0:44:33.835 were processed by the receiver. So you see that before the 0:44:39 sender sends, before the receiver actually 0:44:42.153 requests, sends a message requesting a whole new window 0:44:46.307 worth of data, so you see now that some of the 0:44:49.769 sort of, go ahead and send more messages from the receiver 0:44:54.153 arrive at the sender after the time that the sender starts 0:44:58.538 sending messages, this fifth and sixth message to 0:45:02.23 go ahead and processed. So we've managed to sort of 0:45:06.771 increase the amount of concurrent processing that we 0:45:10.117 can do in this network. But there still are periods 0:45:13.397 where the network is sort of idle where the sender and 0:45:16.874 receiver are sort of not transmitting data. 0:45:19.63 So we haven't done quite as good a job as maybe we would 0:45:23.238 like. And the reason we haven't done 0:45:25.534 quite as good a job as maybe we would like is when the receiver, 0:45:29.667 so the property that we would like to enforce is that when the 0:45:33.669 receiver says go ahead and send me a new message. 0:45:38 When the receiver says slide the window by one, 0:45:41.113 by the time that the next message to process arrives from 0:45:44.904 the sender, we would like the receiver to not have reached the 0:45:49.033 point where it's idle, where it has to wait. 0:45:51.944 So we'd like the receivers buffered to be big enough such 0:45:55.735 that by the time its request for more data reaches the sender, 0:45:59.864 and by the time the sender's response comes back, 0:46:03.113 we would like the receiver to still have some data to process, 0:46:07.243 whereas what's happened here is that the receiver finished 0:46:11.101 processing all four messages in its buffer before this next 0:46:15.027 message came back with additional data for the receiver 0:46:18.683 to process from the sender. Right, so basically the problem 0:46:24.341 was that the receiver's buffer wasn't quite large enough for it 0:46:29.025 to be able to continuously process data; 0:46:31.971 it didn't have quite enough data for it to be able to 0:46:35.899 continuously process while it waited for the additional data 0:46:40.356 to arrive from the sender. What this suggests that we want 0:46:46 is to set this buffer size, to set the size of the window 0:46:51.419 to be the appropriate size in order for the receiver to be 0:46:56.935 able to sort of continuously process information if at all 0:47:02.451 possible. So there's this question about, 0:47:07.012 how do we pick the window size? So let's assume, 0:47:11.579 so the problem we have is that small windows imply 0:47:16.341 underutilization of the network, OK, because we have both the 0:47:22.172 sender and receiver sort of sitting, waiting. 0:47:26.448 There's times when there's no data that's being transferred 0:47:32.084 across the network. So the question is then, 0:47:36.627 how big should we make the window? 0:47:39 0:47:45 And what we said is we want the window size to be greater than 0:47:50.809 the amount of time -- 0:47:53 0:47:58 We want it to be long enough so that the receiver which, 0:48:01.586 say for example, if the receiver can process 0:48:04.521 some number of messages, rate messages per second, 0:48:07.717 OK, the receiver can process this many messages, 0:48:10.782 we want its buffer to be large enough that if it processes that 0:48:14.826 many messages per second, that the amount of time for 0:48:18.217 additional messages for it to process, that it will still have

0:48:22.195 messages to process by the time additional messages arrive from 0:48:26.239 the sender. So the amount of time it takes 0:48:30.021 for additional messages to arrive from the sender from the 0:48:33.861 time that this guy first sends this OK to send more message, 0:48:37.835 right, is one round-trip time of the network. 0:48:40.8 So it takes one round-trip time of the network for the receiver 0:48:44.976 to receive additional messages to process from the sender, 0:48:48.816 and the number of messages that will process during that 0:48:52.387 round-trip time is whatever its message processing rate is times 0:48:56.631 the round-trip time of the network. 0:49:00 And so, therefore, this is sort of ideally how we 0:49:03.162 would set the window size in this network. 0:49:05.863 Of course, we talked about before how it's tricky to 0:49:09.222 estimate this sort of EWMA thing that we do to estimate the 0:49:13.043 window size is not a perfect estimator. 0:49:15.547 But we're going to try and sort of, this is going to be a good 0:49:19.565 choice for a window size to use. And so, typically the receiver 0:49:23.65 is going to try and sort of use some rough estimate of what it 0:49:27.668 thinks the round-trip time is and the rate at which it can 0:49:31.423 process messages when it informs the sender, of the window size at 0:49:35.508 the initiation of the connection. 0:49:39 OK, so this basically is going to wrap up our discussion of the 0:49:42.795 end-to-end layer, or of the sort of loss recovery 0:49:45.734 issues in the end to end layer. What we're going to talk about 0:49:49.469 next time is this issue of how we deal with congestion, 0:49:52.775 right? So we said that in one of these 0:49:55.04 networks, there can be all these delays due to queuing, 0:49:58.346 and that these delays can introduce additional loss. 0:50:01.469 And we call these delays congestion. 0:50:03.612 And so what we're going to talk about next time is how we 0:50:07.04 actually deal with congestion and how we respond to 0:50:10.102 congestion. OK so we'll see you on 0:50:13.334 Wednesday.