MITOCW | 22. Sliding window analysis, Little's law

HARI BALAKRISHNAN: So this is actually almost near the end. So this is actually the last lecture on transport protocols. And then on Wednesday, my plan is to talk about how many of the things we have studied in this class apply to the internet. It will be a history lesson about communication networks.

And I'll talk in specific terms about two interesting problems. One of them is a problem we'll start on today, which is how you pick these window sizes. And I'll talk about how TCP does this. And it's one of the pretty amazing result that was only invented in the mid-1980s or late 1980s. And the second thing I want to talk about, when I talk about this history of the internet from, say, 1960 to today, I'll talk about how people can hijack other people's routes and be able to attract traffic that doesn't actually belong to them.

So apparently, now there are people who are doing it illegally. But apparently, now some governments are also doing this sort of thing. So it's an interesting thing to understand, how it is that some of the routing protocols we studied are not secure. So I'll do that on Wednesday. And then we'll wrap up next week.

So today, the plan is to continue to talk about transport protocols, in particular, about sliding windows. So just to refresh everyone's memory, the problem is that you have a best effort network, where packets could be lost, packets could be reordered, packets could be duplicated, and delays in the network are variable. And what we would like to provide to applications, like, for example, your web browser or your web client or server, is an abstraction where the application just writes data into some layer, and the application on the other side reads data from a layer, and this transport layer deals with providing in order reliable delivery.

So we looked at the first version of that protocol, which is stop and wait. And it had a few nice ideas in it. The first-- all simple ideas-- the first is to use sequence number, and then to have acknowledgments, and then to retransmit after a timeout. And I didn't actually talk about how to do adaptive timers and the low pass exponentially weighted moving average filter. But we started that recitation. And if I have time, I'll come back to that today. But my assumption is that you've already seen how to do that.

But then we concluded that the throughput of the stop and wait protocol is not very high. It's sometimes a good idea. For example, to get reliable delivery between this access point here and your computer right now, a stop and wait protocol is perfectly reasonable. We'll understand why later on. But the short answer why is because the round-trip time between this access point here and your laptop is quite small.

And because it's a really small round-trip time, you're able to get one packet per round-trip time-- or in [INAUDIBLE] packet losses, it's less than one packet per round-trip time, but roughly about one packet per round-trip time. But the round-trip time is on the order of microseconds. One packet per round-trip time can give you a throughput that's quite large.

And therefore, if the link speed is 10 megabits per second, and you're able to send 1,000-byte packet in, say, 20 microseconds, if you take the ratio, that's probably going to be bigger than the link speed. And therefore, you're going to get of the order of the link speed. And therefore, you're not going to underutilize the link.

But now, if the round-trip time were 100 milliseconds, and you were able to send just one packet every 100 milliseconds, it would be slow. And to solve that problem, we looked at this idea of a sliding window. This is just pipelining. It just says, rather than have one packet unacknowledged at any point in time, we're going to have a value w that the sender decides, decides upon this value.

And the semantics of a window are that we're going to have w unacknowledged packets in the system between the sender and the receiver. Now, technically, it's at most w packets. Because from time to time, you might have transients, where you have less than w packets, because you're about to send the next packet. Or if you get toward the end of the file, and you run out of data to send, you're clearly going to have fewer than w packets. So the technical definition of a window, a fixed-size window, says that if the window is w, then the semantics are that we will have no more than w unacknowledged packets in the system. Now, that's not the only possible definition of the window, but that's our current operating definition of the window.

So then the rules at the center are very simple. When you get an acknowledgment from the receiver, as long as it's an acknowledgment for a packet that you have sent, and that packet has not previously been acknowledged, then you now know that packet has been acknowledged. So you remove it from your list of knowledge packets, and you send a new packet. The packet you send, the new packet you send, is the smallest sequence number that you haven't sent so far. OK? It's the very simple rule.

And separately, there's a calculation of the timeout, an exponential moving average filter that calculates the average value of that timeout, the smooth estimate of the timeout, a similar calculation that finds the deviation from the mean. And you pick a return special timeout that is some number of deviations away from the mean, for example, four times the smoothed estimate. If that timer fires, and you haven't received an acknowledgment, you retransmit the packet. It's a very simple idea.

So I'm going to show now what happens in some pictures with the sliding window, when you have packet loss. So it's the same picture as the last time, except now we have a packet-- packet 2 is lost. The sender doesn't know that it's lost yet.

So packet 1 goes here. Packet 2 is lost. This side was supposed to be packet 3. The sent-- packet 4 is sent, and packet 5 is sent. And the window size in this example is 5.

So now, when the first packet gets its acknowledgment, the window slides to the right by 1. And at this point, we send packet 6. And the window is now packet 2 to 6, the sent packet 6. And now, in the meantime, what's going to happen is, when packet-- packet 2 doesn't reach, but packet 3 reaches. when packet 3 reaches, 3 gets an acknowledgment. The receiver says that it's received 3. When it receives 3, what's the next packet that's transmitted? The packet that's transmitted is 7.

Now, let me ask you a question here. The sender got packet A1 and packet A3-- sorry, acknowledgment A1 and acknowledgment A3. Now, it the sender were calculating the expected next acknowledgment, it knows that after A1, it should get A2, and it now got A3. So why doesn't it just re-send packet 3 right now? Yeah?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: It could have been delayed. Now, yes, it could have been delayed. But if it were delayed, wouldn't 3, packet 3, have also been delayed?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: Why not?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: All packets are delayed. So the question is, what is it about-- the delay is one part of the answer, but what is it specifically about that delay that has caused this to-- I mean, if a packet gets delayed, and packets are sitting in a queue, if the first packet in the queue was delayed, then the remaining packets are also going to get delayed, because they're sitting behind that packet in the queue. Yes, sir?

AUDIENCE: Does it depend on the size of the packet?

HARI BALAKRISHNAN: Well, so far, let's assume that you have a network where packets are delayed, and delays are variable. And you have a switch, right? And the switch has a queue in it. And let's say that you have-- in this example, you have packet 1 and 2 and 3. But in fact, packet 2 was lost, and you don't know that, this packet 3. That's one case.

But in the other case, if packet 2 were not actually lost-- if packet 2 were lost, and you got an acknowledgment for 1 and an acknowledgment for 3, if packet 2 were legitimately lost, then it's certainly correct behavior for the sender to send, when it receives A3, to retransmit packet 2. So clearly, you're going after a case here, where 2 exists, but wasn't lost.

In other words, if the sender were to retransmit packet 3-- oh, sorry, packet 2 when it receives A3, she said that that's wrong, because it could have gotten delayed. But what kind of delay would delay packet 2 but not packet 3? Or what kind of delay equivalently would delay A2 but not A3?

If it's sitting behind the same queue, and the queue is serviced in that order, I mean, if this packet was delayed, this package would also be delayed. And this packet's behind this packet in the queue. So what else could it be? Yeah?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: Yeah. So the word I'm looking for is that packets could get reordered in the network. In fact, the reordering could happen even if there were no variable delays, like, no queuing delays in the network. I mean, you could just have a switch that you have a switch here. It could be that packet 2 gets sent that way, and packet 3 gets sent this way.

Here's a very concrete example of how this would happen, from your previous lab. It could be that the network had a certain set of routes, and packets were going along this path. And then maybe there was a failure before, and a new link showed up. Or the failure healed. And the routing protocol converged to pick this path, going forward. And this new packet 3, that showed up after 2, gets sent along this path.

And it could easily be the case that this path has a lot shorter delay to the destination than that path. And therefore, what would happen is that the receiver, packet 3 would arrive before packet 2. So in other words, if I had a network where no packets ever got reordered, no acknowledgments got-- no data packets got reordered, and no ACKs got reordered, then, in fact, it was perfectly good behavior for the sender, at this point, when it observes A3, to go ahead and resend packet 2. Because I'm guaranteeing to you that there's no reordering in the network.

But in general, the networks and packet switch networks, I mean, they get a lot of their robustness to failure and resilience to failure because they send packets any which way. Their only job is to get packets to the destination with as high a likelihood as they can, which means packets are allowed to get reordered. And therefore, it's not correct for packet 2 to get retransmitted when you get A3. OK?

So let's keep going. So what is the next packet that's going to be sent, when you get A3, in this picture? It's 7. Because the sender's rule is very simple. Have I seen the ACK before? No. Is this ACK corresponding to a packet that I've sent before?

Remember, we need that check, because it's possible that a flaky-- there's some bug on the receiver side. So is it is an ACK that corresponds to a packet I've sent before? Yes. Send the next in-sequence packet. So it sends packet 7.

At this point, we're going to lose the beautiful animation. Because each of these things takes an endless amount of time. So I just produced the full picture. I just wish I had the patience to sit and do the full animation. But I ran out of patience.

So anyway, you sent packet 7 at this point. And then, when you receive A4, you send packet 8. When you receive A5, you send packet 9; A6, you send packet 10. Now, let me ask this question. At this point in time, when you receive this acknowledgment, A5, and you sent packet 9, what is the window? That is, what is the set of packets in the window?

The window is 5, the window size, but the window size corresponds to some list of packets that are in the window. What is that set of packets or that list of packets? Yeah?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: 2, 7, 8, 9, 10. This is important. This is 2, 7, 8, 9, 10. These packets are not in sequence. It's very tempting to say, the windows [INAUDIBLE] five packets. So if I've sent out 10, the window must be 10, 9, 8, 7, 6.

Well, that's not true. The window just says, here's the number of unacknowledged packets. The number of unacknowledged packets is 5, in this case. All right, let's keep going. When 10 arrives, when 10 is sent out here, and then at some later point in time, we get an acknowledgment for 7, we send out 11. When we get 8, we send out 12.

At this point in time, the window is 12, 11, 10 9, and 2. At some point, the center times out. And the timeout is picked to be conservative. That's why we take the smooth average. We take the deviation. Because we don't actually want to transmit a packet that hasn't genuinely been lost.

The reason for that is oftentimes, when you start seeing weird behavior like this, like a presumed missing packet, you're not actually sure if it's missing or if it's just delayed, as was pointed out before, because it took a different route. Something strange is going on in the network. And causing retransmission to happen of a packet that hasn't actually been lost makes things worse, because it adds more load onto the system right about the time when there's something fishy going on in the network.

So the last thing you want to do, when something is under stress, is to add more stress to it. That's why the timeouts are conservative. Any time the sender, in any protocol like this, retransmits a packet that is not actually lost, that's considered a spurious retransmission. It's considered a retransmission that is just not a good thing.

Now, actually, our protocol, as we've described, it has some wonderful nice properties. And I'll show later on-- maybe you can read about this in the book-- that it actually is the best possible protocol you can come up with in an asymptomatic sense. In other words, no other protocol, if you ran it for a long time, would actually get higher throughput in a network that had losses. So it has some nice properties.

But it has this one bad property, which is that, in fact, this protocol, in the way these acknowledgments are structured, ends up with a lot of spurious retransmissions. Or they could end up with a lot of spurious retransmissions. Can you kind of see why this protocol could be that we follow this discipline extremely nicely, that timeouts are conservative. So the only timeout, when we are really, really sure when a packet-- we don't get an acknowledgment, is when we time out, and we wait a long time.

But the protocol could still have spurious retransmissions. Can anyone see why there's a very peculiar behavior of this property of this protocol that comes from the way in which these acknowledgments work?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: Yeah. So this protocol has a peculiar problem, which is that all packets and acknowledgments are essentially the same. They contain the same information. If you lose a packet or you lose an acknowledgment for that packet, the sender can't tell the difference.

Now, this is, therefore, not necessarily the best protocol, in the sense that, if you have a path-- here is an extreme case. So I have a path where there's no packet losses going from me to you, and I'm sending data to you. And coming back, the packet loss rate is 25%. This protocol has this unfortunate property that I believe that 25% of my transmissions are lost to you.

In fact, you've got every single packet I've sent. It's just that I don't see the acknowledgments for those specific packets. And therefore, I'm going to retransmit all those packets to you, leading to spurious retransmissions.

So we don't have to worry about it for the lab or for the class, but as a design problem, can you invent a protocol that fix this problem? Can you modify this protocol, or come up with an idea of your own, which has the property that-- pick the design point that is the sender-to-receiver path is generally loss-free. But let's say that the receiver-to-send path has a high loss.

And by the way, this is not hypothetical. This is what happens in wireless networks a lot. Because that base station sitting in some cell tower that has a huge amount of power-- it's powered in-- it consumes probably kilowatts these days. So they can blast at whatever is the maximum allowed by the FCC. And your poor, little, dinky phone is sending acknowledgments. And the thing is running out of battery all the time. So they're carefully trying to figure out, what's the minimum power at which I can transmit?

So in fact, these asymmetric conditions are not unrealistic. They're quite realistic. So if I ran this protocol on a network like that, it would be probably a bad thing. So what would you do to the protocol? Yes? What?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: Send multiple acknowledgments-- yeah, you could send multiple acknowledgments to every time. So you'd be doubling. Yeah, that's a little bit-- but it's the right kind of idea. You want some sort of redundancy. Yeah?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: Yeah. That's actually not a bad idea. In a sense, you're sending multiple acknowledgments, but you're not just blindly sending multiple acknowledgment. But any time you send an acknowledgment, you could also say-- sending the list of all packets you've received so far is a huge amount. Because if I send you a gigabyte movie, or something, I mean, by the end of that movie, you're just sending me a lot of acknowledgments. So you don't quite want to do that.

But remember, the receiver has some idea. If it knew the window size, it would have some idea of what the number of things at the center. You could do something even simpler than all of that.

One thing you could do is that the receiver, when it acknowledged the packet, wouldn't just acknowledge packet 7, when it got packet 7, but it might be able to send a cumulative acknowledgment. In other words, it could say, that when I send an acknowledgment, I guarantee to you that everything I've received up to this point-- I'm sorry. I guarantee to you that all packets up to that point I have received.

So if I tell you that my acknowledgment is 17, I guarantee you that there's nothing before 17 that I've not received. And then I could, in addition, in the acknowledgment, tell you a little bit about some of the later packets I've received or some of the later packets that might be missing. So you can make this protocol have a little bit more redundancy.

And if you do that, and you apply almost everything else I've taught, you get TCP, which is an extremely popular protocol. But that's about the only difference of significance between our protocol and TCP. Now, interestingly, our protocol, when you actually have loss rates in the forward and reverse directions that are roughly the same, our protocol actually does a little better than what TCP happens to do. But TCP is good at dealing with the reverse path having a higher degree of packet loss. OK.

So the other question I want to ask people here, at this point, is let's say that you have a receiver that's running on an extremely simple device. So you don't want to have a lot of storage. Now, why would you need storage, before I get to that question? Let's take this picture here.

So packet 2 hasn't yet been received. But in the meantime, the receiver has gotten packets 3 and 4 and 5, all the way up to 12. So what does the receiver have to do? Well, the receiver, remember, before it delivers it to the application, it has to hold on to those packets. It can't deliver packet 3 to the application and packet 4 to the application. Because the guarantee that the receiver is giving is that all packets will be delivered in exactly the same order in which they were sent.

So the receiver has got to hold onto those packets until packet 2 shows up. Does that make sense? OK. How big can that receiver's buffer become? How big do you need to make it? Like, if you were implementing this on a computer, if you want to allocate memory for it, how big do you need to make it?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: What?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: Big enough to handle the timeout-- good. How big can the timeout be? Well, the timeout can be some finite number. But think about what happens. Think about the timeout happens. You retransmit packet 2. And packet 2 is lost again.

Now, in the meantime, the protocol is going to continue. Because all these other packets are going to keep getting acknowledgments, and they're going to keep causing the sender to keep sending packets. So if packet 2's retransmission were lost, we're going to be at this point here. We're still going to be sending, at this point in time, packet 13 and whatever-- 13 and 14 and 15 and 16, and so forth, right?

Now, packet 2 could just keep getting lost. I mean, it may happen with low probability, but there's a probability that it'll happen. So how big does the receiver's buffer have to be in this implementation, in the worst case? Well, let's say that you don't know how big the file is. It's a continuous stream of packets that are sent.

I mean, is that a bound? In the worst case, is that a real bound on the size of the buffer? Or can it grow to be as big as the entire stream that you're sending? It can grow to be really, really big.

Now, this is a potential problem. Because it can keep growing and growing and growing. At some point, you're going to run out. You might run out of space. When you start to run out of space, it's tempting to just throw things out.

So let's say that the receiver implements it. Somebody implements this protocol and just says, I'm going to just have 100 packets. The sender is running with a window size of 5. I'm just going to have a buffer of 100 packets, which says, the maximum number of packets I'm going to hold in my buffer, before I start discarding later packets, is 100. Does this protocol work? Is it correct if I do that? Yes?

AUDIENCE: [INAUDIBLE] like a receiver just never acknowledges [INAUDIBLE] receives it [INAUDIBLE].

HARI BALAKRISHNAN: OK, but what if I acknowledge a packet as soon as I get it? OK, if you acknowledge a packet as soon as you get it, the receiver's discipline, the guarantee it should provide, is if it acknowledges a packet, then it's told the sender that it's got the packet, which means the sender will never retransmit it, which means it shouldn't throw the packet away. So as long as the receiver only throws out packets that it doesn't acknowledge, you're OK. Does that make sense?

So the discipline is it's just like writing a legal contract, right? That's what protocols are. It's just a bunch of legal contracts, and you try to make them as simple as possible. And you try hard, and you end up with 200 pages. But that's what lawyers also say-- yeah, it's really simple. But then you've got all these clauses.

But the reality is that you've got to deal with all these [INAUDIBLE] cases. So protocols are nothing more than contracts that both sides agree upon. And the contract here from the receiver is actually pretty simple. It says, if I send you an acknowledgment, it means that I'm not throwing the packet away.

What happens if I treat this protocol to do a little bit differently at the receiver? When I get a packet, if it's in order, I deliver it to the application. And after I deliver it to the application, I send an acknowledgment. OK? So in other words, I only send an acknowledgment to the sender after it's delivered up to the application. Otherwise, I don't.

What happens to this protocol if I do that? Does it perform the same as what I describe? And remember, there's a subtle difference. The only difference is, in this protocol as I've described it, the receiver gets a packet, sends an acknowledgment, and then holds on to it in a buffer, if the packet's not the next packet in order.

The modification I'm proposing is the receiver gets a packet, and only when it delivers it up to the application, does it send an acknowledgment. Otherwise, it doesn't send acknowledgment. Yes?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: Is it just like stop and wait?

AUDIENCE: I think so, because [INAUDIBLE].

HARI BALAKRISHNAN: But if packets are not being lost, it's doing a lot better than stop and wait, right? If packets are not getting lost, it's doing-- would you agree that if packets are not lost, it does better than stop and wait? In fact, if packets are not lost, is there any difference between my protocol and this [INAUDIBLE] modified? No. OK.

But yet, you had a good thought. It looks like stop and wait. When does it look like stop and wait?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: Yeah. So that modification is, when packets are lost, it looks like stop and wait. Now, this is not a mere academic thing. So it turns out that there was a period of time in the '90s, where somebody in Linux TCP had the bright idea-- it seemed like a bright idea-- that that's what they would do. So for a period of time, there was a Linux TCP, where they said, well, it's all very complicated.

Because what would happen was that sometimes, the machine would crash. And the sender thought that the packet had been acknowledged, but it hadn't actually been delivered up to the application. So let's just make it so the packets get delivered to the application, and only when the application does the read-- for those of you who've done this sort of thing-- from the socket buffer, and it's been out in the application, and out of the operating system, that's when we'll send the acknowledgment.

And you [? read, ?] seemed OK. People said, that seems reasonable. And Linux, the way it seems, things seem to work, as people try out a lot of stuff. And then I suggest, from time to time, somebody declares that something is right.

So anyway, they tried this out. And it actually didn't work as well. And the reason for that is, if you don't run on a high enough packet loss rate network, then what could happen is that you may get stuck. And it's very hard to notice these performance problems. Correctness problems are one thing, because the other side stops. It stops working. And you can corner it down.

But this simple tweak, that looks perfectly reasonable, is actually a performance problem. And it doesn't show up all the time. It actually shows up only when the packet loss rate is reasonably high. So these are all examples of reasons why these protocols are not completely obvious and require a fair amount of care to get it to work. Are there any questions about any of the stuff? Is this all clear? OK.

What I want to do now is to show a picture of something called a sequence plot, which is a very useful tool in understanding how these protocols actually work. So what you do to produce one of these plots is you run your protocol. And you plot out-- at the sender, you plot out the times at which the sender sent out every sequence number, every time it transmitted a packet. And you plot that out as a function of time.

The y-axis is the sequence number. The x-axis is time. And similarly, every time the sender gets an acknowledgment, you plot that out on the trace as well. So you look at these two traces. OK, this is a trace of packet transmissions, data packet transmissions, and this is a trace of ACK packet receptions. And you look at this picture.

Now, the moment you get a picture like this, there's a few things you can immediately conclude. The first thing you can conclude is, that if I look at the distance between the data and the ACK, when there are no losses happening, if I look at that distance, that tells me the window size. Because that's the number of packets. Because the last acknowledgment, every time an acknowledgment happens, you send out a new packet. Therefore, the distance in sequence numbers between-- in one of these vertical slices, when there are no packet losses, is the window size.

You can also read off the typical round-trip time of the connection. Because the round-trip time is the time between when a packet, data packet, was sent and when you got an acknowledgment for it. So you can read that off as well. Those two pictures, there's an easy way for you in your lab 9 to produce these pictures.

So if you're running into things where things look slow, things look bad, you should just put up one of these pictures, and then it'll usually become pretty apparent what's going on. What may happen is that initially, things look like this. And all of a sudden, things stop. And you can start to see, well, I'm not getting acknowledgments, or I'm not sending data the right way. And these are very useful to understand what is going on.

And generally speaking, these are useful to uncover performance issues rather than correctness. I mean, correctness, usually, you can iron out before you get to this stage. The retransmission timeout is the time between when you send a packet and when you send the retransmission for the packet. In this particular picture, the deviation from the mean was small. And that's why the retransmission timeout is only a little bit bigger than the mean round-trip time.

Every time you see a packet that's off of that sequence [INAUDIBLE]-- so you see packets here. The pluses are data packets. And then you see something going normally. And then you see a lower sequence number retransmitted, sent here. That's a retransmission. So you see, normally, the new packets are all sent there. But the retransmissions show up before. So these are examples of retransmissions, and these are examples of packets that were retransmitted more than once, because they're timing out multiple times. Yes?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: Yeah. So the window size-- what's the definition of the window size? The maximum number of unacknowledged packets. So the maximum number of unacknowledged packets, when there are no packet losses that have happened, is the difference between the last packet you transmitted and the last acknowledgment you got. Because every time you got an acknowledgment, you send a new packet.

And initially, you send out w packets. So if you continue that, so you initially send 1 to 5, then you send 2 to 6, 3 to 7. And the last ACK you had was 2, when you sent out 3 to 7. So that distance tells you the window size. I might be off by 1. It's probably the last packet you sent minus the last acknowledgment you got plus 1, is the window size, or minus 1, something like that. You've got to get that right on the quiz. Fortunately, I don't have to get it right here.

[LAUGHTER]

And then, some of these things here are later x's. And these are acknowledgments that show up. So these are packets that got retransmitted multiple times. These are acknowledgments that are for these retransmitted packets. And I say most probably, because I can't actually be sure.

In principle, it could be that this acknowledgment here is for this packet, is for this data packet, that was actually originally transmitted over here, rather than for this retransmission. It's, in principle, possible that this acknowledgment was sent by the receiver upon the reception of a packet over here. It's just that it's unlikely. It's more likely that it was this, because that's the round-trip time that's consistent with that RTT. But you can't actually be sure.

All you know is that this was an acknowledgment for that data packet. But most likely, it was for the retransmission. OK, so these sequence traces are generally pretty helpful and useful in understanding the performance of transport protocols, particularly sliding window protocols. So any questions? OK.

So now I'm going to turn to the last remaining issue for these transport protocols, which is analogous to we did a calculation of the throughput of the stop and wait protocol. I want to look at the throughput of the sliding window protocol. OK. And I want to explain that by first actually explaining what the problem is. And then I want to go back and tell you about a very beautiful result, very widely applicable result, applies to everything from networking to how long you're going to wait to get served at a restaurant, called Little's Law. It's a remarkable result, very simple, and widely applicable. Everybody should know it.

So the question here is, what's the throughput of sliding window? And in particular, if I had run a protocol in a network that looks like this-- so I have a sender. I have a receiver. There's some network path in between. And of course, this has a bunch of switches here. And I want to know what is the throughput of the protocol.

And what I would like a few to tell you what the throughput is in terms of. So the sender has a window size w, according to this protocol. For now, we'll assume that there's no packet loss. That is, acknowledgment data packets are not lost, and acknowledgments are not lost.

If I have time today, I'll come back to explaining what happens with packet loss. Otherwise, we'll pick it up in recitation tomorrow, or I'll point you to the place in the book. It's just a simple calculation that expands. The more important part is when there are no losses.

Now, I'll also assume that there's links of different rates here. And one of these links on the path between sender and receiver is the link that is the bottleneck link. In other words, no matter what you do or who you bribe, you cannot send packets faster than the speed of that link. For simplicity, I'll assume that there's one bottleneck.

The general results apply, even though there are multiple bottlenecks. But I'll assume that there's some bottleneck here. And I'll assume that its rate as c packets per second. And I will assume here, that because there's a bottleneck, in general, packets may show up faster than the bottleneck can handle. And if they do, they sit in a queue.

And because I've constructed the problems so packets don't get lost, the queue can have an arbitrary length. It could be potentially-- it could grow unbounded. Though, in reality, it won't, because the sender has a fixed window size of w.

Now, all of this analysis and calculation will apply when there are many, many people transmitting data, sharing this bottleneck. So you can have multiple set senders to multiple receivers, and they'll all share this link in some way. And for now, today, all I'll assume is that there's one user of the network. It's not hard to extend the same calculation to multiple users.

And the question is, what is the throughput in terms of the window size and in terms of these other things? Now, in order to answer this question, it'll turn out that the throughput depends on the window size, and also on the round-trip time, and also on the loss rate, and also on-- in a certain mode, it will depend on c. It can't exceed c. OK?

But in order to understand how to solve these kinds of questions, there's a more general result that's more widely applicable, called Little's Law, which I want to tell you about. So Little's Law applies to any queuing system. It applies to any system where there's some big black box here, and the black box has a queue sitting inside it, and the queue drains at some rate.

So you have a queue sitting here. Things arrive into the queue. I'll call that the arrival process, which I'll represent by A. And then things come out of the queue, according to some service process, which I'll represent by S. By the way, Little is a professor at MIT. I think he wrote this result, this law. Well, I don't think he called it Little's Law, but other people did. So he did this work, I think, in the 1950s.

And what's beautiful about this result is that it relates three parameters. It relates the-- I'll call it N. It relates N, the average number of items that you have in this system, in the queue, or in this black box. It relates that to the service rate and to the average delay experienced by an item that sits inside this black box.

So let me relate the three again. It relates N, which is the average number, to D, which is the average delay-- so I'm going to put a bar above the fact that it's an average-- to lambda, which is the average rate. Now, the result applies to a stable system. What that means is it applies to a system where the queue doesn't grow unbounded to infinity.

In other words, it applies to a system where the service rate-- if the arrivals are persistently bigger than the service, then it doesn't matter what you do. The queue is going to grow to infinity, and the delay is going to grow to infinity, and N is going to grow to infinity. So you're going to get a relationship that's not of much practical use.

But otherwise, if the rate at which things come out of the system, in a stable system, is lambda-- which, if it's a stable system, the rate at which things enter the system can't exceed it either. But it relates the service rate lambda for a stable system that doesn't grow unbounded to N entity. OK? So let me give this first by example.

How many of you guys have used the food truck? All right, so last week, I did a little experiment there. And I found that-- this is all real data. I found that, at least the [? Thai ?] truck, they seem to take about 20 seconds per person, on average. OK?

And when I showed up there-- and this wasn't an average calculation. But I showed up there, and there were 20 people ahead of me in the line. And the question, of course, is I don't care how many people there are in line. What I care about is, how long do I have to wait?

Assuming that the random sample I did was the average, which who knows if it was or not, looking at these two numbers, what's the waiting time? In other words, what's D? 10 minutes? Is it? I didn't wait 10 minutes. How do you get 10?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: I see. It might be. So I got 30. So I had it as 20. All right, it might be 10. Why is it 10? How do you conclude that it was 10? Who said 10?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: Why?

AUDIENCE: Well, so it'd be like, [? if you have ?] 30 people, then you have [INAUDIBLE] [? per person ?] [INAUDIBLE].

HARI BALAKRISHNAN: Yeah. Right. So what this says-- what you did was you just said that D must be equal to N over lambda, right? Or N is lambda times D. So if you say that it's 20 seconds per person, is 3 people per minute. So what you do is you do 30 people divided by 3 per minute. And so you get 10 minutes. So that's about right. That is exactly right.

So Little's Law just tells you that the average number of items in a system-- this is all applicable to various conditions on stable systems, and so forth-- it says the average number of items, or packets, or people, or whatever, is equal to the product of the rate at which the system is servicing them multiplied by the average delay that they experience. So knowing two of them, you can calculate the third.

And what's truly, truly remarkable about the result is that it applies to anything that you do in the system. Packets could arrive, or jobs could arrive, or people could arrive, in some arbitrary distribution. They could be serviced according to some completely arbitrary distribution.

They don't have to be serviced in the order in which they arrive. They could be shuffled around. You could make it so people who come in last get serviced first. You could do whatever, and the result still applies. Yes?

AUDIENCE: [INAUDIBLE] delay [INAUDIBLE]?

HARI BALAKRISHNAN: No. Well, I kind of cheated here a little bit. This is 20 seconds per person. But whenever I tell you a number like that, what this really says, that this is 3 people per minute. So it looks like a delay, but this is really an inverse of a rate in the way-- this is the inverse of the rate in the way I've described it.

I mean, it's intuitive to say they take 20 seconds per person. But when I tell you that it takes 20 seconds per packet or 20 seconds per person, it looks like a delay. But it's really a rate. So it's important. That's a good question. Yeah, so this is a rate. So this is inverse time. And this is whatever quantity you're dealing with. So if you then take the ratio of N to lambda, you get time.

OK, so why is Little's Law true? So here's a very simple pictorially proof of Little's Law. And it applies under specific conditions. But it turns out these conditions are good enough for our use.

So let's say we draw a picture like this of a queue. So I'm going to assume that packets enter the queue and leave the queue. Now, the fact that there's a single queue, versus not a queue, doesn't matter. It's any black box. So p.packets could get-- or information or messages or items could get sent from the sender. They enter a black box, and they get stuck out at the receiver. And the thing applies to that as well.

So let me plot the number of packets in the queue, or the number of items in the queue, as a function of time. So I'm going to assume here that capital T is extremely long. Whenever I deal with rates, I have to look at what happens over a long period of time, and then I can calculate a rate.

So you can see that what I've done here is, that every time a packet arrives or an item arrives, the queue increments by 1. So you can see that the y-axis, the height of each of those little snippets here is 1. And then every time it leaves, it drops by 1. So you get-- in a particular execution of whatever the queue does, you get a trace that looks like this.

Now, of course, in a different execution, the details might be different. But if you do it for a long enough time, you're going to sample all the possible evolutions of this thing, or at least enough of it, so you can make meaningful statements. So whenever a packet arrives, I've shown it in a color. And I think I've matched the color up against whenever that packet leaves.

But in fact, the result applies-- this particular example is a first-in, first-out queue. So packets leave in the same order they were sent. But that doesn't have to be true. So let me label these packets as shown here.

Now, what I'm going to try to do is to relate the rate at which packets have entered or left the queue to the number of items in the queue and the average delay experienced by each item in the queue, in this pictorially proof. So the way you do that is everything has to do with the fact that there are two different ways of looking at the area under this curve. And there's two different ways. One of them relates to the rate, and the other relates to the average delay. And then we're going to say, all right, the area under the curve is the same, and therefore, we're going to equate two numbers.

So the first thing I'm going to do is I'm going to divide this up, the area under the curve, and divide it up into rectangles like this and associate with each rectangle a packet, or an item. So I'm going to say that A showed up here, and it left at that point. So this entire period of time here corresponds to packet A sitting in the queue.

This entire period of time corresponds to B. A left at this point in time. So now my queue was three packets, and they're B, C, D. And then, at this point in time, E showed up. So we now have C and D sitting here. But now E showed up, and then F showed up, and so forth.

So you agree that I can divide this up into rectangles and associate with each little rectangle, whose height is 1, a particular packet. And that is the same packet in the queue. So the height represents a particular packet, and I associate every little piece of this queue picture with a given packet.

Now, let's assume that we run this experiment for a long time T, capital T. And P packets were forwarded through the system. So what is the rate?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: P packets per T seconds-- so the rate is clearly lambda is P over T. Right? This is easy. OK. Great. Now, let's assume that the area under the curve is A. This is the entire area under the curve here.

Now, this is an area under the curve of N of T, which is the number of packets, as a function of T. So if I take this area under the curve, which is the same-- if you think of it in continuous domain, it's the integral of N of T-- and I divide by T, I get that number. Right? You agree that the mean number of packets in the queue is the integral of N of T, which is the number of packets in the queue at any point in time. If I take that integral, and I divide by capital T, I get the mean number of packets in the queue.

All it says is this is the total number of packets in the queue over-- aggregated across all time. Therefore, to find the average, I take the integral, and I divide by T. That's the definition of the mean.

All right, so now we have two things. We have the rate is P over T. And we have that the mean number of packets in the queue is A over T, where A is the area under the curve. Now, to complete the puzzle, what we have to observe is, that if you look at the same area under the curve, you can look at it in two ways. The one way to look at it is the mean number of packets in the queue is some line through here, which is the area under that curve divided by T.

But each of these things accounts for a certain delay. And the mean delay experienced by the packet is simply the area under this entire curve, but it's divided into all of the packets that ever got forwarded through the system. So through this experiment, P packets got forwarded by the system. And the area under the curve also represents a total aggregate delay. Because if I look at it with this axis here, that's the total time. So that's the total time spent.

And if I take this entire area under the curve, and I look at that area under the curve, and I divide by the number of packets that I sent, that gives me the average time that a given packet spent in the queue, which means that the mean delay is A over P. So if I take A over P and multiply it by P over T, what I get is A over T, which is equal to N. And that's Little's Law.

So now we're going to apply Little's Law. I mean, it's actually a very intuitive idea. It just says, that if I take the rate, average rate, and the average delay, and I multiply the two, I get the average number that's sitting in the system. So in order to complete the picture for the throughput of this sliding window protocol, what we're going to do is to apply Little's Law in a couple of different ways.

We're going to say, that if the window size is w in that protocol, and the round-trip time is RTT-- that's the time between when I send a packet and get an acknowledgment back-- I first apply Little's Law. So now I have a big black box. I send out packets. And every time I receive an acknowledgment, I send out another packet. And I never send out more than w packets.

So the average delay between when I send a packet and when I get an acknowledgment for it is RTT. So that's the D in the Little's Law formula. The number of things that I have sitting in this black box inside the network, the number of outstanding things that I have that are waiting to be processed, is w. And therefore, the rate is, by Little's Law, N over D, which is w over RTT.

So therefore, the throughput of this protocol is simply equal to w over RTT. So if I increase w, I get higher throughput. So if I draw this as a function of the window size w, I look at the throughput here. I get a linear increase like that.

Now, the problem with this is, of course, you look at this and go, well, the best way to get higher and higher and higher throughput is to keep increasing the window size. So what happens if I-- does this keep going on forever, that all I have to do is to keep increasing the window size, and then I'm getting infinite throughput? That's clearly not happening.

So what happens? Why is it that it's completely true that w over RTT is the throughput. So why is it that I can't just keep increasing the window size and get infinite throughput? Yeah?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: Well, it's true you're bounded by c. But yet, this formula is true. Right? It's true that there's some round-trip time. So what's really going on, of course, is that if you increase the window size more than a certain amount, all that's going to happen is the packets are going to get stuck in this queue here. And they're going to start draining at some rate, c packets per second. But they're just going to get stuck at the back end of the queue.

When they get stuck at the back end of the queue, the RTT is no longer fixed. The RTT now also starts going. So in other words, the throughput is always this formula. But initially, when the packets are no longer in the queue-- until a certain point, initially, you said one packet. It goes through.

You have a window of two packets. They go through, and you get ACKs. Three packets, they go through and get ACKs. At some point in time, they start to fill up the queue. And when they start to fill up the queue, w keeps growing. But RTT keeps growing. And what ends up happening is this ratio doesn't exceed c. So you end up with throughput that looks like that.

And the point at which this happens here, this point here is actually a product of the minimum RTT of the system, which is the round-trip time in the absence of any queuing. I'm going to call that RTT min. And that depends on the propagation delay and the transmission delay, but not on the queuing delay. If there's no queues, and there's a certain minimum round-trip time-- like, it takes 100 milliseconds to go to California and back, or whatever.

Now, when queues start to grow, that RTT starts increasing. But until that point happens, the round-trip time is RTT min. And if I take that, and I multiply that by c, that's the critical window size up to which point there are no queue packets that build up in the queue.

But after that, packets start to build up in the queue. And there's a name given to this product of the bottleneck link speed, or bandwidth, and RTT min. It's called the bandwidth delay product. It's the product of the bandwidth and the delay, where the delay is the minimum round-trip time.

And if I were to draw an analogous picture of the actual round-trip time as a function of the window size, initially, when the window size is small, the round-trip time is RTT min, with some value. And then, at this point in time-- I want to mimic this thing here-- you get to this point in time, which is the bandwidth delay product. And then the round-trip time starts to grow.

So this is the actual delay. And so you look at this picture. And a well-designed, well-running protocol will run with a window size roughly around here, where it gets the highest possible throughput at the lowest possible delay. But sometimes, you might end up running with a bigger window size. You're not going to get any faster throughput, but what you would see is increasingly higher delay.

Now, in real networks, designing protocols that run at this nice, sweet spot is an extremely challenging problem. I'll get back to this problem on Wednesday and talk about how people work on it. It's still a somewhat open problem. In fact, it's still an open problem in things like cellular wireless networks. So I'll come back to this point. But the main point here is this idea of a bandwidth delay product.