MITOCW | 21. Reliable transport

HARI BALAKRISHNAN: So, good afternoon. Continuing our story about networks, what we've seen so far is a story where you have a network that you're trying to connect computers to communicate together. And we use a network with switches arranged in some topology to allow us to find paths between computers. And so we looked at the routing problem. We looked at two different routing protocols to solve that.

Earlier, we talked about this idea of a packet-switched network and there are queues in packet-switched networks. And when traffic comes in too fast, and the queues overflow, packets may get dropped. We also looked before at links which have errors on them. And so if you have errors on links, then your coding scheme isn't able to correct for those errors, packets may get lost. And when we looked at shared media networks with MAC protocols, depending on the MAC protocol you use, you may collisions, which means that packets may get lost.

So what you have is a packet-switched network that has the property of something called a best-effort network. And what "best effort" means is that the network has a few properties that you have to cope with. The first property of a best-effort network is that packets may get lost. The second problem in a best-effort issue that arises in a best-effort network is that packets have delays, but the delays are variable. And particular queuing delays that happen in switches are variable delays.

The good property of a packet-switched network is that each packet is treated independently by the network. So it could be that you have a stream of packets you want to send, say, belonging to a video stream or a file. And the sender sends them in some sequential order, but these packets may take different paths through the network.

And in fact, there may be switches in the network, for whatever reason, that may not treat packets in first in, first out order. They may reorder packets. But more generally, packets may take different paths through the network because the routing protocol might decide to change the paths on you, and so packets make get reordered.

And the fourth issue in a best-effort network is that, in fact, packets may get duplicated. So you may have the same packet show up multiple times even though it was sent only once for a bunch of reasons. One of them is that there could be problems in the implementation of the switches or the nodes that cause packets to get duplicated.

But it could also be that you may have a link with a high packet loss rate, or this may be a shared medium where you have a MAC protocol that has collisions. So you may have a retransmission protocol-- let me try to resend the packet a few times at the lowest layer over a shared medium or a link. And sometimes, multiple versions of the multiple copies of the same packet may get through. And we'll actually understand why that happens more today. So packets may get duplicated.

So in a way, a packet-switched network like the internet is great because it's very easy to build. And the reason it's easy to build in some sense is because about the only property that you're providing from the design of the network is to tell the endpoints, oh, I might get your package through. There's no guarantees on anything. As long as there's some non-zero probability of getting a packet through from one endpoint to the other, that's pretty much all it takes to declare that you have a conforming best-effort network.

So it's easy to build. But of course, it means that you have all these issues that you want to deal with if you actually want to run applications. So as an example of an application is, let's say you are trying to download a web page, or a set of web pictures and text on a page. What you would like is an abstraction that you can implement some sort of a scheme you can implement in the network or in between the endpoints that makes it so that an application sends a bunch of bytes or packets or sends a message. And at the receiving side, you get those bytes reliably.

So that's what we're going to understand today. We're going to look at today and next week, we're going to look at how to implement a protocol that provides reliable data transport. And the ideas we're going to look at are probably the ideas that are in the world's most popular computer program. It's the most popular in that it runs in the most number of places. And it's a protocol called TCP, which stands for the Transmission Control Protocol.

Now, we're not going through all the gory details of TCP. We're going to look at a simplified version of this protocol. So maybe it's TCP-lite. But it'll cover the main idea of how you can achieve reliability. And this particular program runs on pretty much every computer and every phone and every little device that's on the internet today. So it's really, really popular.

In fact, we're going to start with a simpler protocol that is a reliable data transport protocol that isn't used between endpoints. TCP is used between endpoints. Now we're going to look at a version of a protocol that's a simple version that actually runs in every 802.11 device-- your laptop, phone, and access points. So we'll study that protocol too first.

In the context of end to end, between endpoints, reliable data transport, the problem is the following-- you have some network here. It's a best-effort network with those properties. And what you want is you have an application at one end. And you have an application at another end running on some endpoints.

What this system provides that we're doing this study provides is an abstraction where you run software here-- you run software at this end, and all of the stuff sits on your end node. This is your endpoint. And the abstraction provides some nice properties. The application writes some data in here at the sender sending in. So let's call this the sender. And the other guy's the receiver.

The application writes stuff inside here. The network is a best-effort network. And there's some protocol between these two pieces of software that make sure that no matter what the network does, what goes up here into the application is exactly the data that was written from this application in exactly the same order in which it was written. So it provides reliable and in-order delivery of data, so reliable and in order. So every piece of data that's written shows up in exactly the same order exactly once at the receiver.

And these two ends are called transport. These two ends constitute the transport layer. And they [INAUDIBLE] at the end points, OK? So the application writes in here, the transport protocol delivers up to the application stuff that's reliable and in order. And in particular, it provides the semantics that you can think of as "exactly once" semantics. In other words, anything that's sent is delivered exactly once to the receiver, and it's delivered in order.

Now, that's the abstraction that TCP as well provides. And that's the abstraction of our 602 protocol. It's reliable, in-order, exactly-once delivery. Now, there are other implementations you can have. There are protocols you could have that provide reliability but not in order. So I'll give you all the data that you send, but it may show up in different order, and it's your problem to fix it.

Or you might provide a protocol that provides in order, but not reliable. So if I'm doing real-time video conferencing, say, Skype, Skype would probably want to provide a protocol that's in order but not reliable, because if I speak, you'd like to actually get those things into the Skype application in order.

But it's not really required that it be reliable because if a message shows up, say, more than 100 or 200 milliseconds after I spoke it, it's going to distort the conversation. It's not going to be intelligible to you. And the human ear is wonderful at-- the human brain is wonderful at dealing with some clippiness in the voice. You know, occasional packets get lost. It's not the end of the world.

So there are lots of interesting applications. But in order is useful, but not perfectly reliable. Applications where reliable is useful, but not perfectly in order. BitTorrent would be an example of that. An application where eventually you want all of those movies that you're trying to get. But who cares what order they come in? You're not going to start watching it until the whole file is assembled. And so the protocol that BitTorrent uses in effect-- it's a complicated protocol. It's not point to point. But in effect, it provides reliability without worrying about ordering.

So there's lots of combinations. The combination we care about is reliable, in order, essentially giving you the illusion that you have a circuit between the two endpoints, a wire between the two endpoints, OK? So is the abstraction clear? Everyone understands what we're trying to solve?

And in between this, just think, there's an adversary or some network in the middle that you sent packets, and the thing is just throwing packets away. And every once in a while, just for the heck of it, it decides to delay a packet for a long time. And every once in a while, it decides to send packets along different paths. And your job is to deal with all of that and design the sending side and the receiving sides so that stuff still shows up reliably and in the same order in which it was sent.

So we're going to try to solve this problem. We're going to solve it first by coming up with a protocol. It has a nice name to it called stop and wait. It's a very simple idea. And this will be a protocol that works, but it's slow. But the good news is it works. It's correct. It gives the semantics we want. And then we'll try to improve its performance.

It's a very simple idea. I'm sure you have-- just think about this for three minutes. You'll come up with something that looks like this. You just take the message you want to send, whatever file, stream, or video, whatever it is, and break it up into packets. So far, there's nothing new here.

The main first idea is we're going to number every packet with a sequence number. So that's what's shown here as "Data 1," "Data 2," "Data 3," and so forth. So we're going to use a sequence number on every packet.

Now, again, there are many ways to implement sequence numbers. The way that's the simplest and most conceptually clean is every packet has a sequence number that increments by 1 for every subsequent packet that's sent. And you might initially start the sequence numbering at 0 or 1 or whatever. The sender and the receiver have to agree on that.

Now, in reality, TCP in the real world is a little more complicated. TCP provides sequence numbering by numbering the bytes with the byte offset in the stream. So if you send a packet which is 25 bytes, and the next packet is 200 bytes, the first packet is going to have a sequence number of, let's say, 0. The second packet's going to have a sequence number of 26 because it numbers the starting of the byte offset. But these are all details that to first order, we don't have to really worry about.

The important point is there is a sequence number. And a sequence number is a unique identifier for the packet. In other words, if I later send a packet with the same sequence number, I have to guarantee that the material inside the packet is the same as it was before. So the assumption is that this is a unique identifier for the contents of the packet. So it's a unique identifier. We want to use it for some other set of bytes. We'll always use it again for the same set of bytes if we ever retransmit a packet with the same sequence number.

When the receiver gets the packet with a certain sequence number, it does what the post office does if you send registered post-- you turn around, and you send an acknowledgment. And to allow the sender to know which packet's being acknowledged, you stick in the sequence number of the packet that's being acknowledged. So you send sequence 1. Data 1, you get ACK 1. Data 2, you get ACK 2, and everything is wonderful. It's easy, easy protocol.

So what happens when a packet's lost, this data is lost? What's going to happen is the sender is not going to get an acknowledgment. And after some period of time called the timeout, the sender decides that it wants to retry that packet. And it tries to resend the packet. And if it works, it gets an acknowledgment. When it gets that acknowledgment, that's when it goes and sends the next packet.

So the property of stop-and-wait protocol is that you send a packet only after you get an acknowledgment. You sent packet k plus 1 only after you get an acknowledgment for packet k. If you don't get an acknowledgment for packet k, you wait for a period of time called a timeout. And after that timeout elapses, you retransmit the packet that you considered was lost, that you thought was lost. OK, simple.

Now, is this protocol reliable? And when I ask that question, you have to assume that the network may drop and reorder and do whatever it does to packets. But there's always a non-zero probability that a packet or data packet or acknowledgment packet sent on the network has a non-zero probability of reaching the other side. Because if the probability of packet loss is 1, now no one can help us.

So is this protocol reliable? OK. Is this protocol in order? It is in order in the way of-- I'm not actually describing what the receiver does. But I should tell you that the receiver's semantics here are when the receiver gets the packet, it delivers it to the application.

Now, if you turn out that this protocol is not necessarily in order the way I described it-- and I'll come back to why. But so far, it looks like the protocol is in order. But remember what I said about the receiver? When the receiver gets a data packet, it delivers it up to the application. So is the protocol potentially not in order? It's not actually in order. We'll get back to why. You have a question?

AUDIENCE: Yeah, I was just wondering if the receiver gets a data packet and then tries to send an acknowledgement back, if the acknowledgement gets wrong, I guess, the sender will resend data. Does the receiver then compare to figure out?

HARI BALAKRISHNAN: Right, so I haven't specified that. And you are one step ahead of-- you're at the next picture here. What happens in this case? You get a duplicate packet. And in fact, that's precisely for this reason that this protocol is not actually-- it's kind of in order, but in order means that you deliver packets in the same order in which they were sent.

And in the way I've described the description, given the description of this protocol, this protocol does not provide exactly one semantics, right? It provides at least one semantics. In other words, every package is delivered at least once to the application. And what you would like us to deliver every packet exactly once to the application in order. So what would you have to do at the receiver in the software that you write and the receiver transport to take the same idea and make it be a reliable, in-order, exactly-once protocol? Yes?

AUDIENCE: Loop up if you received that sequence number.

HARI BALAKRISHNAN: Sorry, say again.

AUDIENCE: Look up if you've received that sequence number.

HARI BALAKRISHNAN: Look up if we received that sequence number.

AUDIENCE: I think that's right.

HARI BALAKRISHNAN: Good. So one implementation is you perhaps keep track of all the sequence numbers you've ever received and delivered up to the application. If the new guy comes in, you look and see if it's in your list, and deliver it if not. You could do better. You have to do all that work. You have to keep track of the list of all the sequence numbers you've ever received in order for this protocol to work.

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: Yeah, is it enough to keep track of simply the very last one you've delivered and also guarantee that you'll only deliver stuff in order? So if you get up to packet number 17 and you now get 18, you deliver it up to the application and update your counter to be set from 17 to 18 of the last sequence number you've delivered.

If your last sequence number delivered in order is 17, and you get 16, throw it out. If you get 17, you throw it out. If you get anything-- if you get 19, which probably shouldn't happen in this protocol unless there's a mistake in the implementation that they're sending in, if the last sequence number I got was 17, can the sender send 19?

AUDIENCE: No.

HARI BALAKRISHNAN: Why not?

AUDIENCE: Because we have 17 acknowledgements [INAUDIBLE].

HARI BALAKRISHNAN: Because-- that's right. So unless there's a bug in either side of implementation, which, trust me, when you implement it, you'll probably end up having some bugs, and you'd know something is amiss. But there are these invariants that have to hold. The sender can send k plus 1 only if it gets an ACK for k. The sender gets an ACK for k only if the receiver got k.

And therefore, if the sender's last in-order sequence number received and delivered the application was 17, it can't actually get a 19 in a correctly implemented protocol. But if it does, because in the real world, you don't know who the heck wrote the sending side. You might have done your receiver, and the sender might have been done by, oh, I don't know, Microsoft. And it may have an issue with it, or Apple, or whoever. I mean, you don't want to trust it, right?

So you have to be careful about making sure that you might want to assume the protocols-- you don't want assume necessarily that the other guys implemented the protocol right because he might not have. And so, who knows what might happen? So your rule as the receiver is to rigidly obey whatever the discipline is, which is you deliver up a packet exactly in order.

OK, so we wanted exactly-once semantics. And the way you get that is you get that by keeping track of the very last sequence number that you received. So this protocol-- so the first idea is sequence numbers. The second protocol is a retransmission after a timeout. Now, how big should this timeout be? This whole protocol rests on this magic timeout. What should it be? 15, 17? What are the units of the timeout? Actually, what are the units of this timeout? It's time. So it's, like, seconds or milliseconds or something. How big should it be?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: What?

AUDIENCE: 5 milliseconds.

HARI BALAKRISHNAN: 5 milliseconds?

AUDIENCE: [INAUDIBLE] milliseconds.

AUDIENCE: Units [INAUDIBLE] milliseconds.

HARI BALAKRISHNAN: Yeah, units are seconds or milliseconds. Good. But how would you pick it?

AUDIENCE: You know the round-trip time.

HARI BALAKRISHNAN: OK, good. So that's a good idea. There's this thing I've written on the left called the round-trip time. But you don't know the round-trip time, but you could measure the round-trip time. And I'll talk about how you measure it a little bit later.

But it's important to realize that if you make the timeout be smaller than the round-trip time, where the round-trip time is defined as the time at which you sent a packet to when you got an acknowledgment for that packet-- if you make the timeout smaller than the round-trip time, what happens in this protocol?

Let me first ask, is the protocol still correct? By correct, I mean, does it provide reliable, in-order delivery? OK, it's correct, because that correctness does not rest on how we pick the timeout. However, what is the problem with making the timeout smaller than round-trip time?

AUDIENCE: [INAUDIBLE]

HARI BALAKRISHNAN: Yeah, if this protocol's going to be here, you're going to be retransmitting and retransmitting and using up a lot more of the network's resources than you need to in order for you to actually get your protocol to work correctly. And you might, if the time out is really, really small, you would probably congest the network.

OK, so the timeout has to be bigger than the round-trip time. The trouble in a packet-switched network is that delays are variable in a best-effort network. And, in fact, packets may be reordered. There may be weird things going on in the network, which means that the round-trip times are actually not constant. They vary with time. They vary with other traffic. They vary with lots of other factors.

And so what you want is an adaptive method that would measure the round-trip time, estimate the round-trip time, and then come up with some sort of an algorithm to compute or to set the timeout as a function of the observations of the round-trip time. I'll get back to that later on today. And we'll also talk about this in recitation tomorrow. It's actually a very nice application of a very simple low-pass filter. So we'll actually come back to this idea.

But what I want to you to have in your head right now is this idea that there's a timeout. And the timeout has to be-- which I'll call RTO for Retransmission TimeOut. We have this idea that our retransmission timeout has to be bigger than the round-trip time, OK? So what I need to tell you still is how to measure and estimate the round-trip time and how to use these estimates of the round-trip time to pick the timeout.

But let's subcontract that problem to someone. Let's say that there's the black box that will tell you what the timeout should be, and now you have this protocol. So assuming we have that black box and someone telling you the retransmission timeout, what I would like to do now is to spend some time telling you how well this protocol works. I'd like to understand what is the throughput, which is the data rate that you get if you're on the stop-and-wait protocol. So that's what I want to do now-- throughput of stop and wait.

So the input here is-- I'm going to assume a very, very simple model. I'm going to assume for a minute that the round-trip time doesn't change a whole lot. This is a very simplifying assumption, but there's some average round-trip time. I'm going to assume that the round-trip time is RTT. The same result holds if the round-trip time varies, but just simple model.

Let's just assume the round-trip time is fixed. And let's assume that somebody tells us what the retransmission timeout is. And I need one more parameter. I'm going to assume that I know the network's packet loss rate because intuitively, if the network's packet loss rate is zero-- that is, no packets are lost. No data packets, no acknowledgments are lost-- then you would expect this protocol has higher throughput than if packets were lost, right?

If the packet loss rate is 50%, you would expect that what would happen is, well, half the packets or ACKs are getting lost, which means you have to retransmit the packet. And every time you retransmit the packet, the protocol comes to a wait. And you have to wait until the timeout happens. So the bigger the packet loss rate, you would expect the protocol to be slower.

So I'm going to assume that we have RTT and RTO. And we have a packet loss rate of L. So what does that mean? What it means is that if I send a large number of packets to the network, a fraction, L, of them will get lost. And I'll just assume in the simplifying model that the packet losses are independent. So they're sort of Bernoulli losses. You know, every packet gets lost independently with some probability.

Now, I also will assume in this protocol-- does it matter to the performance of the protocol if the data packet is lost or if the ACK packet is lost? It doesn't matter. As far as the sender's-- and this is an important point to understand. As far as the sender is concerned, if a timeout happens, it has no way of knowing whether the timeout happened because the data was lost or because the ACK was lost.

This is, like, absolutely-- the receiver knows. Or actually, the receiver doesn't know if a timeout happened, but the receiver does know whether it got a data packet or not. But the sender-- the only thing it's acting on is the absence of an ACK. And the absence of an ACK indicates either that the data was lost or that the ACK was lost, and it has no idea which. Therefore, we could assume for this analysis that this packet loss rate of L is actually a bidirectional packet loss rate. What I mean by that is L is the probability that either a data packet is lost, or its ACK was lost, OK?

Now, if I give you the one-way loss probability, you can do the calculation. That's a probability calculation to find out what is the probability that either the packet was lost or the data was lost. That's an easy calculation. But let me just assume that the probability that either the packet was lost, data packet was lost, or its ACK was lost, is L.

So given these numbers, what I want to do is, given these things, I want to know what the throughput is. In other words, how many packets per second am I transmitting, am I able to transmit, or am I able to receive at the receiver? So if you want to look at what happens in this picture, if you draw time like that-- you send a packet, and maybe you get an ACK here. So D1, A1. You send D2 immediately. And you get A2 after some time.

And maybe you have a timeout. So you send D3. And then you have a period of time, which is the RTO. No ACK happens. You send D3 again. And maybe no ACK happens for a while. You have another RTO. I'll assume that the RTO was fixed here. And you send D3 again. And you get an ACK here. And then you send D4 here, and so forth, right? That's an example of what could happen in a particular time evolution of the protocol.

What I mean by throughput is that I would like to run such an experiment for a very long time, or run many, many such experiments, which is sort of equivalent to running an experiment for a very long time, and then count how many packets did I successfully get at the receiver. Or equivalently, I can ask how many ACKs did I get at the sender over that long experiment, right? And the number of ACKs that I get at the center divided by the time of that experiment will tell me the number of packets per second.

Or put another way, if I run the experiment for some long period of time, and I receive n packets coming back-- right? If I receive n acknowledgments-- and if the expected time here between when I send a data packet-- I send a data packet. I get an ACK. I send a data packet. I get an ACK. I send a data packet, and I get an ACK. I send a data packet, and I get an ACK.

If I take the expected value of that time-- that is, the expected time between when I send a packet and one I get an ACK-- the 1 over that number, 1 over the expected time, is equal to my throughput in packets per second. Because if I run the experiment for a long time, I'm going to get some number of acknowledgments.

So if I run it for some period of time where n times e of t-- where e of t is number here, and I get back an acknowledgment, n divided by n times e of t is my throughput. And therefore, 1 over the expected time is the throughput of my experiment, right?

So this should be intuitive, because what's really happening is, with a little bit of handwaving, actually, that I send data. I get an ACK. Send data, I get an ACK. There's a certain expected amount of time, so I will send 1 over that packets per second, OK? So in other words, the throughput is the reciprocal of the expected amount of time between when I send a packet and when I get an acknowledgment. So it's enough for us to compute the expected value of this time, right, or the mean value of that time.

All right, so we could do that calculation in a simpler way. There's the sort of tedious way to do it, and there's a very simple, nice way to do it. So we want to calculate expected time between data and ACK. And one way to do this is to say that-- let's say I send a data packet. One of two things can happen. I either get an ACK for it, or I don't get an ACK for it. What's the probability that if I send a data packet, I get an ACK for it?

Well, the probability that I send a packet and I don't get an ACK for it is L. Therefore, the probability that if I send a data packet, I get ACK for it, is 1 minus L, right? So with probability 1 minus L, I send a data packet, and I immediately get-- and when I say "immediately," I get an ACK for that data packet, right?

And how long does that take? If I get an ACK for it, the ACK comes back to me in a time which is equal to RTT, the Round-Trip Time, right? So therefore, I can write a formula that looks like this. I can write this expected time which I'm trying to calculate as being equal to 1 minus L. With probability 1 minus L, the expected time between when I send a data packet and when I get an ACK for it is equal to the RTT, right? Because 1 minus L is, by definition, the probability that I send a packet and I get an ACK for it. Send a data packet and get an ACK.

Now, what happens with probability L? With probability L, I send a data packet, and I don't get an ACK for it. So now I want to compute the expected time given that I don't get an ACK for it. The first thing that has to happen is I need to take a timeout. So I have to wait for a period of time shown in this picture given by the RTO.

And then once I wait for that RTO, and I now start by sending a data packet, the expected amount of time before I get an ACK for that data packet is exactly equal to the original expected time that I'm trying to calculate, right? Because it doesn't matter what happened in the past. Let's say I take a timeout. And now, I come back here and [INAUDIBLE]. I'm not going to send a data packet. What's the expected time before I get an ACK? Well, that's exactly equal to the same answer that we're trying to calculate, this expected time over here.

Therefore, I could write this recursion type of relationship. The expected time is 1 minus L the RTT plus L times the RTO plus the same expected time that I'm trying to calculate, right? What this says is with probably 1 minus L, the time it'll take for me to get an ACK is equal to the RTT.

And with probability L, it's equal to-- first of all, this RTO-- I have to wait for that retransmission timeout. And then once I do that, well, I have to add some more time. And that time that I have to add is exactly equal to the same expected time from the left-hand side that I'm trying to calculate. Does this makes sense?

You could kind of do this in a more tedious way. You could say, well, with probability 1 minus L, my time is RTT. With probability L times 1 minus L, the time is equal to RTT plus RTO. With probability L squared times 1 minus L, this is, like, two losses. And then a retransmission-- the time is 2 times the RTO plus RTT. With probability LQ times 1 minus L, it's that. If you do all of that stuff, you'll get the same thing. But this is the more-- there's a simpler way to do it.

So if you run take the expected time over to one side and solve this equation, what you'll end up with is that the expected time is equal to RTT plus L over 1 minus L times the RTO. I mean, as the packet loss rate becomes larger and larger and larger, this term starts to dominate because L over 1 minus L starts to be bigger and bigger and bigger, which is what you would expect.

If the directional packet loss rate is large, you'd expect the RTO terms to start to dominate, and the expected time is larger and larger and larger. If the packet loss rate is zero, then the expected time is exactly equal to the RTT. You send a packet. You get an ACK. And with an RTT, you send the next packet. You get an ACK. And of course, the throughput's equal to 1 over the expected time. That's the reciprocal of the expected time, OK?

Now, what's the best case here? The best case here is that you get one packet per round-trip time. The worst case is arbitrarily back depending on the packet loss rate. But the important point here is that even in the best case, you're able to send only one packet, at most, one packet per round-trip time. The question is, how good or bad is one packet per round-trip time?

Is this clear, this intuition behind why this is one packet per round-trip time in the best case? That should be pretty obvious, right? I send a packet. I get an ACK. Send a packet. I get an ACK. This calculation just shows a little bit more detail about what happens when the packet loss rate's non-zero. So if the packet loss rate is, say, 20%, you take 1/5 over 4/5. So it's RTT plus 1/4 of the retransmission timeout. That's what it says the expected time is. And 1 over that is throughput.

Now, how bad or good-- is it clear? Any questions? OK, so now, how good or bad is this 1 over the round trip time? So let's say that you have a network between Boston to-- I don't know-- San Francisco. And if you do these pings or whatever, let's-- I mean, I don't know the real numbers, but let's say it's 80 milliseconds. Just for the calculation to be easier, let's assume it's 100 milliseconds.

And let's say that a packet on the internet it's about 10,000 bits. So let's make it bytes. Let's say that it's 1,000, say, 1,500 bytes. So what this says is that the throughput that I would get with the stop-and-wait protocol if I ran it on this internet path would be 1,500 bytes divided by 100 milliseconds. So that's 15,000 bytes per second, 15 kilobytes a second, which might have been really, really good in 1985, but no one's going to be happy with this today.

I mean, you might have a link that's a megabyte a second or a gigabyte a second or 10-- you know, bigger than that. But no matter how fast the network links are, this protocol is completely dominated by the delay or the latency, the round-trip latency, between the sender and the receiver. And you end up with a throughput that's pegged to a small value. And so, people don't like that.

So question is, how can you do better? What can you do now to this protocol? Or come up with a new method, a new protocol that would improve the throughput of this system. Because if people pay money for network links, they'd like to actually get higher performance from it. So what could you do?

AUDIENCE: Larger packets?

HARI BALAKRISHNAN: What?

AUDIENCE: Larger packets?

HARI BALAKRISHNAN: Larger packets. Well, yeah, larger packets is-- yeah, why don't we make our packets as big as the five we want to send? Actually, I digress. Why don't we make packets really big? Like, I got a megabyte file or a gigabyte file to send. Why do I have to break it up into smaller packets?

AUDIENCE: Larger packets use more bandwidth?

HARI BALAKRISHNAN: Well, to send the data, no matter if we break it up small or big, you're going to use the same bandwidth. I mean, that's a good question. Yeah, you have an answer?

AUDIENCE: [INAUDIBLE] how to [INAUDIBLE] over time.

HARI BALAKRISHNAN: That's kind of true. You know, if a packet is, you know, let's say a gigabyte file you want to transmit. And you send that in one atomic unit, and goes through four hops in the network, and then it gets dropped on the first hop, you end up having to send an entire gigabyte again over all those other hops. That's actually not good.

But in fact, really large packets are probably a bad idea even for networks which don't drop any packets. I mean, think of the case when I have a gigabyte file to send, and you have a gigabyte file to send. The problem if you make these packets really big is that one of us is-- on a shared link, only one of us can send that packet, which means the other guy is going to be waiting a really, really long time for him to send that packet.

So the reason why, in the end, packets are modest size has to do with our wanting to share the network evenly over smaller time scales. It's because we want to give fairness across smaller time scales, allowing everybody who's competing access to the network. So even if we have big amounts of data to send, we prefer to break them up into smaller chunks among other reasons, one reason being we don't want to start other connections and prevent them from gaining access to the network because there's some huge transfer sitting in front. So that's part of the reason.

So anyway, so bigger packets doesn't quite cut it. So what else could you do? Yes?

AUDIENCE: [INAUDIBLE] to send. So if you cannot [INAUDIBLE].

HARI BALAKRISHNAN: OK, you know, well, I'll come back to this on Monday. That's actually a really good idea. But when would you stop for 8, 16, 32? I mean, at some point, this is like--

AUDIENCE: [INAUDIBLE] at some point, it's going to tail.

HARI BALAKRISHNAN: Because packets are lost.

AUDIENCE: Yeah, so you go back.

HARI BALAKRISHNAN: OK, this is a really good idea. We're not actually going to teach that here in this course. This is actually what TCP does in the beginning of the connection. But before we-- what else could you do? That's a good idea. Yeah.

AUDIENCE: You could send a fixed number.

HARI BALAKRISHNAN: Yeah, you could do a fixed number. You know, somebody could pick-- I actually kind of-- it is a really good idea to do 1, 2, 4, 8. And then if it fails, you come back down to, say, 1 or 1/2 of whatever worked the last time and then continue from that. That particular thing has a name to it. That protocol is called slow start. It's ironic because it's really fast. It's exponential, right-- 1, 2, 4, 8. But yet, it's called slow start. I'll probably tell you more about it on Wednesday.

But we'll ease into that solution. We'll do something simpler. We'll use something called a sliding window protocol with a fixed-size window. You just make that 1 be 7 or 4 or 6 or 8. I'll tell you later next time how you pick that value, OK? And one way to pick that value is to do it dynamically like the gentleman in the front said. It's more complicated. But let's just pick a fixed-size value.

So the idea is actually very, very simple. Now that I have one packet outstanding. We use this idea in computer science. We use this over and over again-- pipelining. So you just send multiple of them and have multiple outstanding packets. By "outstanding," I mean a packet that hasn't yet been acknowledged. A data packet that hasn't yet been acknowledged is called an outstanding data packet. And you have multiple of these outstanding.

And every time you get an acknowledgment, you send one more packet. So that's shown in this timeline here, right? So you start here. You send a packet. I don't know why this isn't working. Ah, there we go. You send a packet. You get an acknowledgment. When you get an acknowledgment, you send another packet. Get an acknowledgment. You send another packet.

But in the meantime, there are these other acknowledgments coming in. And the rule is very simple-- every time you get an acknowledgment that you have not seen before, send the next packet in sequence. So the sender just keeps sending packets in sequence order. Every time it gets an acknowledgment that it hasn't seen before for a packet that it had sent before, it sends the next incrementing sequence number.

So this painstaking animation will attempt to show you that, assuming it's correct. So the window here is five packets, OK? I'll tell you later some guidelines on how to pick this window size. But this number of packets here is called the window, the number of outstanding packets, or the number of unacknowledged packets. It's always going to be 5. It's going to be 5 in this example. It's always going to be a fixed value in our protocol, OK?

So you send the first packet. When you get an acknowledgment for that first packet, you slide the window forward by 1, and you send packet 6. When you get an acknowledgment for a packet 2, you slide the window forward, and you send packet 7. When you get an acknowledgment for 3, you slide the window forward, and you send packet 8. This is-- sorry? Yeah.

AUDIENCE: So it appears [INAUDIBLE] out of order.

HARI BALAKRISHNAN: That's a good question. I'll get to that in a moment. The answer is that the sender's rule is always the same. Yes, you get acknowledgments out of order. As long as it's an acknowledgment for a packet-- sorry, as long as it's an acknowledgment that you have not seen before for a packet that you have actually sent, you slide the window forward by 1 and send a new packet, OK? And you keep track of the fact that you received an acknowledgment, so you know that you should never retransmit that packet.

I want to define this thing and pause here. I want you to understand the definition of a window and internalize it. If the window size is W, what it means is that the maximum number of unacknowledged packets that you can have in the connection is W. There are many different ways of defining a window. In fact, TCP inside it has two windows. This definition is one of those windows. I won't talk about the second definition here. I'll get to it next week. It's not important for us right now.

So again, to repeat, if the window size is W, it means that the maximum number of unacknowledged packets in the system in the protocol is W. So the rule of the sender is going to be to very religiously adhere to this rule. In other words, every time it gets an acknowledgment, it waits and sees whether it's an acknowledgment for a packet it has sent before that it has not seen before.

If you get an acknowledgment like that, it means that some packet has been received, which means you can get rid of that packet from the stack of unacknowledged packets that you have and send a new packet. Because you can send a new packet because you know that the number of unacknowledged packets reduced by 1 because you got an ACK, which means you can now send a new packet, OK? It's a very simple rule if you just follow that idea to implement. It also is surprisingly easy to get wrong. Yeah.

AUDIENCE: So the window doesn't necessarily have to be consecutive?

HARI BALAKRISHNAN: The window doesn't have to be consecutive. This is a really, really good point. And it's very tempting to implement a window that's consecutive. And you'll find that after a while, if you follow that idea, and you do it wrongly, the protocol will just stall. And every time, there's about a quarter of the students, the first time they implemented this, it just stops working after a while as the packet loss rates grow.

So it's important that in this definition of the protocol in the way it's defined here, the window of unacknowledged packets-- it's not necessarily consecutive. So you could have packets 1, 2, 3, 4, 8, 9, 10, 11 outstanding if your window size is 8. The other guys may have gotten acknowledged. That's absolutely true, yes. OK?

All right, now what happens under all these other weird cases that are going to happen here? So let me first show you a timeline of how a timeout is dealt with. So let's say in this case, the window size is 5 again like it was before. So everything is going wonderfully well here. And let's say now you move on. You send packets 6, 7, 8. And let's say packet 8 is lost. What the sender is going to do is it's going to send packet 9. It's going to send packet 10 based on acknowledgments for 4 and 3 that it received before.

So, sorry, when it got acknowledgment 3, it sent packet 8. 8 was lost. The sender didn't know that at this point. When it got an acknowledgment for 4, it sends 9. When it got an acknowledgment for 5, it sends 10. When it gets an acknowledgment for 6, it goes ahead and sends 11. When it gets an acknowledgment for 7, it goes ahead and sends 12. So at this point, the sender actually has outstanding 12, 11, 10, 9, 8, OK?

Now, at some point, it discovers that-- in fact, this picture continues. In this picture, what happened is that you sent out 9. You've got an acknowledgment for 9. And at that point, you send out 13 because whenever you get an acknowledgment, you send out the next consecutive packet you should be sending out. So at this point in time, the sender has a bunch of outstanding packets in it, and it's got acknowledgment. And this is an interesting case because packet 8 was lost. 9 was sent later, and 9 got acknowledged.

But we still haven't timed out on 8. So at this point in time, the outstanding packets in the window are 13, 12, 11, 10, and 8, giving you that nonconsecutive observation that you notice. And then at some point in time, based on the round-trip time, based on the black box, the guy timed out, and 8 got retransmitted. And then 8 got retransmitted. You got an acknowledgment for 8, and the protocol sort of continues in that fashion.

Does this makes sense? So it's actually not that hard when you think about it. What the sender does is a very simple idea, which is every time it gets a new acknowledgment for a packet it had sent before but hadn't seen an acknowledgment for before, it just goes ahead and sends the next packet. And then it has a separate process by which it maintains this timeout. And whenever an acknowledgment does not arrive within a timeout, it goes ahead and retransmits that back.

Now, when it retransmits the packet, the assumption here is that the original packet was actually lost. So the timeout has to be bigger than-- for the system to work well, the timeout has to be bigger than the maximum time that a packet can sit around in the system. So if the timeout is too small, and you retransmit 8 too early, it could be that 8 is not lost, but 8 is just being reordered in the network and going on some very long circulous path. You know, I recently read that someone got a letter in New York 70 years-- it was sent in 1943, and it showed up, like, two weeks ago.

So, I mean, it could happen on the internet too. I mean, quite literally, if you're on an Amtrak train, and you're using their wireless network, some packets come to you in 300 or 400 milliseconds, which is arguably very long. But literally, there are packets that will come back to you a minute after you sent them-- reach the receiver a minute after you sent them. And they could be out of order.

So in fact, this is my student's, and I call this the great Amtrak network. It's really good because it gives us really interesting research for others to work on. But the people who are on that train probably-- it's miserable.

So anyway, this could happen. But, and so, the timeout is a heuristic. It could be that 8 was retransmitted wrongly in a spurious way. But our hope is that if the timeout is long enough, it could be that-- if the timeout's long enough, the idea is that you retransmit 8 because the original 8 was lost. And now the outstanding packets in the window are 8, 13, 12, 11, and 10. But the 8 here is not that 8, but this 8. But as long as the contents of 8 are the same, it doesn't matter which 8 it is.

But of course, if the timeout is too small, there are two 8's sitting in the network. And now you actually have more than W packets in the window. But as far as the sender is concerned, it has exactly those 8, 10, 11, 12, and 13. It's true that there's one more packet if the timeout happened too early. That's the network's problem. As far as the sender is concerned, it has five outstanding packets.

The receiver is a little trickier than in the other case because what it has to do-- I mean, it's trickier in that it has to maintain a buffer of packets. So the receiver has a little more of a job to do. In the previous stop-and-wait protocol, any time it got an out-of-order packet, it's probably because the sender is badly implemented, right? If the last sequence number I deliver to the application was 17, and I got a 19, it's a bug.

Whereas here, is the last sequence number I delivered to the application is 17, and I get a 19, it means that, well, there's a window. And maybe 18 was lost, or who knows what happened, right? Maybe 18 will show up later. So the receiver now has an interesting job.

And this is important because when you implement this stuff in this protocol [INAUDIBLE], you've got to make sure that whenever you deliver packets from the receiver protocol to the application, you deliver it in order and update the last sequence number you delivered, OK? So that's important to do. But if you do that and then acknowledge a packet when it's received, just send an acknowledgment for it. The protocol will continue, and it will work well.

So this is what you'll be looking at in the lab, the piece that's going to go out today. This is the last lab. And then on tomorrow in recitation, we'll look at how the timeout is selected. And then on Monday, I'll talk more about an analysis of this protocol.