So I'm back.

I guess by your presence here you've shown that for you people, DP1 is well under control.

And that's great.

OK, so today is the last lecture on this topic on the networking piece of 6.033. And, the topic for today is something called congestion control.

And what we're going to do is spend most of today talking about congestion control.

But let me remind you of where we are in networking.

And so if there's one thing you take away from 6.033 from this networking piece you should remember this picture.

So the way we're dealing with networking, and the way we always deal with networking in any complicated system is to layer network protocols.

And the particular learning model that we picked for 6.033 is a subset of what you'd see out in the real world.

And it's mostly accurate.

There is something called the link layer which deals with transmitting packets on the link.

And of that, you have the network layer.

And, the particular kind of network layer that we are talking about is a very popular one which provides a kind of service called a best effort service.

And the easiest way to understand best effort services: it just tries to get packets through from one end to another.

But it doesn't get all panicked if it can't get packets through.

It just lets a higher layer called the end to end layer deal with any problems such as lost packets, or missing packets, or corrupted packets, reordered packets, and so on.

And last time when Sam gave the lecture, he talked about a few things that the end to end layer does.

And in particular we talked about how the end to end layer achieves reliability using acknowledgments.

So the receiver acknowledges packets that it receives from the center, and the sender, if it misses an acknowledgment goes ahead and retransmits the packet.

And in order to do that, we spent some time talking about timers because you have to not try to retransmit immediately.

You have to wait some time until you're pretty sure that the packet's actually lost before you go ahead and try to retransmit the packet.

The next concept at the end to end layer that we talked about is something called a sliding window, where the idea was that if I send you a packet, got an acknowledgment back, and then sent you the next packet and did that, things are really slow.

And all that a sliding window is, is really an idea that you've already seen from an earlier chapter, and probably from 6.004 called pipelining, which is to have multiple outstanding things in the pipe or in the network at once as a way to get higher performance.

And the last thing that we talked about last time was flow control.

And the idea here is to make sure that the sender doesn't send too fast because if it's sent really fast, it would swamp the receiver, which might be slow trying to keep up processing the sender's packets.

So you don't want to swamp the receiver's buffer.

And we talked about how with every acknowledgment, the receiver can piggyback some information about how much space it has remaining in its buffer.

And if that clamped down to zero, then the sender would automatically slow down.

You would produce what's also known as back pressure back to the sender's thing.

I don't have any more space to slow down.

And the sender guarantees that it won't send more packets at any given time, more data at any given time than what the receiver says it can handle in its buffer.

So that was the plan.

Those are the things we talked about so far for the end to end layer.

And what we're going to do today is to go back to one of the main things about networks that was mentioned during the first networking lecture and talk about how we achieve that, which is sharing.

Ultimately, it's extremely inefficient to build networks where every computer is connected to every other computer in the world by a dedicated link or a dedicated path of its own.

Fundamentally, networks are efficient only if they allow computers connected to each other to share paths underneath.

And the moment you have sharing of links, you have to worry about sharing the resources, namely sharing the bandwidth of a link and that's what we're going to do today.

And we're going to do this.

Basically the goal for today is to talk about the problems that arise if you don't you do sharing properly, and then spend some time talking about how we solve these problems.

So imagine you have a network.

And I'm going to start with a simple example.

And we're going to use that example to write through, because it will turn out that the simple example will illustrate the essential problem with sharing.

So imagine you have a bunch of computers connected to one end of the network.

And at the other end you have other computers.

And imagine that these are senders and these are receivers.

And they share the network.

And you know a really simple form of sharing this network might be when you have all of these computers, take their links like the Ethernet connections and hook them up to a switch.

And maybe you hook it up to another switch.

And then there's other paths that eventually take you to the receivers that you want to talk to.

And imagine just for example that these are fake lengths.

So for example these might be 100 Mb per second links.

And then you go [SOUND OFF/THEN ON], and that might be, let's say it's a megabit per second link.

So this might be 1 Mb per second.

And these things are 100 Mb per second.

And of course you could have the receivers connected with extremely fast links as well, which means that if the sender and the receiver just looked at their own access links, these 100 Mb per second links, and just thought to themselves, well, I have 100 Mb per second links.

The sender has 100 Mb per second link, so clearly we could exchange data at 100 Mb per second.

That would be flawed because all of these things go through this relatively thin pipe of a megabit per second.

So really, the goal here is to take all of these connections, all of these end to end transfers that might be happening at any given point in time, and share every link in the network properly.

And I'll define what I mean by properly as we go along.

So let's write some notation first because it will help us pose the problem clearly.

So let's say that there is some offered load that these different senders offered to the network.

So you've decided to download a bunch of files, music files, and WebPages.

And each of those has a certain load that it's a certain size of a file.

A sender can actually push that data through at some rate.

So that's the offered load on the network.

And let's assume that senders [want?] through N here, and let's say that the offered load of the sender is L1. This is L2, all the way through LN.

OK, so in this simple picture, the total offered load on the network along this path is the summation of LI's, where I runs from one to N, right?

Hand out here, in this simple picture, we have all this offered load going through this one common link, which we're going to call the bottleneck link.

We don't actually know where; the senders don't know where the bottleneck is because it's not near them.

And in general, it's not near them.

It could be, but in general, they don't know where it is.

The receivers don't know where it is.

But there is some bottleneck link that's going to throttle the maximum rate at which you could send data.

Let's call that rate C.

So what we would like to be able to do is ensure that at all points in time that the sum of the load that's being offered by all senders that share any given link is less than the capacity of that link.

And so, C here, this picture, this would be C.

And notice that we don't actually know where these things are.

And for different connections, if you are surfing the web and downloading music and so on, the actual bottleneck for each of those transfers might in general be different.

But the general goal for congestion control is you look at any path between a sender and receiver, and there is, in general, some bottleneck there.

And you have to ensure that for every bottleneck link, in fact for every link, the total offered load presented to that link satisfies this relationship.

OK, so that's ultimate goal.

[SOUND OFF/THEN ON], OK?

[SOUND OFF/THEN ON] one other problem, that you want this.

This is not a trivial problem.

So one reason it's not trivial is something that might have been clear to you from the definition of the problem.

You want to share this for all the links in the network.

And the network in general might have, or will have, millions of links, and hundreds of millions of hosts.

So any solution you come up with has to be scalable.

And in particular, it has to scale to really large networks, networks that are as big as the public Internet, and the whole public Internet 10 years from now.

So it has to be something that scales.

It has to handle large values.

So it has to handle large networks.

It has to handle large values of N, where the number of N to N connections that are transferring data on this network, and it's an unknown value of N.

You don't really know what N is at any given point in time.

And it could be extremely large.

So you have to handle that.

And above all, and this is actually the most important point.

And it's often missed by a lot of descriptions of congestion control and sharing.

A good solution has to scale across many orders of magnitude of link properties.

On the Internet or in any decent packet switch network, link rates vary by perhaps seven, or eight, or nine orders of magnitude.

A single network could have links that send data in five or ten kilobits per second on one extreme, and 10 Gb per second, or 40 Gb per second on the other extreme.

And a single path might actually go through links whose capacities vary by many orders of magnitude.

And it's extremely important that a solution work across this range because otherwise its generality isn't that general.

And what you would like is a very general solution.

The second reason this problem as hard is that here we have this N, but really N is a function of time.

You're surfing the Web.

You click on a link, and all of a sudden 18 embedded objects come through.

In general, each of those is a different connection.

And then it's gone.

And the next time somebody else clicks on something, there's a bunch more objects.

So N varies with time.

And as a consequence, the offered load on the system, both the total offered load as well as the load offered by single connection varies with time.

So, LI varies with time as well.

So it's kind of really hard to understand what the steady-state behavior of a large network like the Internet is because there is no steady-state behavior.

And [SOUND OFF/THEN ON] is the thing that we're going to come back to and [SOUND OFF/THEN ON] the person who can control the rate at which data is being sent is the sender.

And maybe the receiver can help it control it by advertising these flow control windows.

So the control happens at these points.

But if you think about it, the resource that's being shared is somewhere else.

It's far away, right?

So, there is a delay between the congestion point, or the points where overload happens with these points in the network, and where the control can be exercised to control congestion.

And this situation is extremely different from most other resource management schemes for other computer systems.

If you talk about processor resource management, well, there's a schedule of the controls, what runs on the processor.

The processor is right there.

There's no long delay between the two.

So if the operating system decides to schedule [another?] process, well, it doesn't take a long time before that happens.

It happens pretty quickly.

And nothing else usually happens in the meantime related to something else, some other process.

If you talk about disk scheduling, well, the disk scheduler is very, very close to the disk.

It can make a decision, and exercises control according to whatever policy that it wants.

Here, if the network decides it's congested, and there is some plan by which the network will want to react to that load, well, it can't do very much.

It has to somehow arrange for feedback to reach the points of control, which are in general far away, and are not just far away in terms of delay, but that delay varies as well.

That's what makes the problem hard, which is that the resource is far from the control point.

So these are the congestion points, which is where the resources are.

And these are the control points, which is where you can exercise.

And the two are separated geographically.

I don't think I mentioned this.

This is what you want.

And any situation where the summation of LI, the offered load on a link, is larger than the capacity, that overload situation is what we're going to call congestion.

OK, so any time you see LI bigger than C, we're going to define that as congestion.

And then, I'll make this notion a little bit more precise.

And you'll see why we have to make it slightly more precise.

But for now, just say that if inequality is swapped, that means it's congested.

OK, so that's the problem.

We are going to want to solve the problem.

Now, every other problem that we've encountered in networking so far, we've solved by going back to that layered picture and saying, well, we need to deliver, for example, if you want to deliver packets across one link, we say all right, we'll take this thing as a link layer protocol.

We'll define framing on top of it, and define a way by which we can do things like [UNINTELLIGIBLE] correction if we need to on that link, and we'll solve the problem right there.

Then we say, OK, we need to connect all these computers up together and build up routing tables so we can forward data.

And we say, OK, that's the network layer's problem.

We're going to solve the problem there.

And then packets get lost.

Well, we'll deal with it at the end to end layer.

And, that model has worked out extremely well because it allows us to run arbitrary applications on top of this network layer without really having to build up forwarding tables for every application anew and run it on all sorts of links.

And you can have paths containing a variety of different links.

And everything works out like a charm.

But the problem with doing that for congestion control is that this layered picture is actually getting in the way of solving congestion control in a very clean manner.

And the reason for that is that the end to end layer runs at the end points.

And those are the points were control is exercised to control the rate at which traffic is being sent on to the network.

But the congestion, and the notice of any information about whether the network is overloaded is deeply buried inside the network at the network layer.

So, what you need is a way by which information from the network layer about whether congestion is occurring, or whether congestion is not occurring, or whether congestion is likely to occur even though it's not yet occurred, that kind of information has somehow to percolate toward the end to end layer.

And so far, we've modularized this very nicely by not really having very much information propagate between the layers.

But now, to solve this problem of congestion, precisely because this separation between the resource and the control point, and the fact that the control point is at the end to end layer, at least in the way we're going to solve the problem.

And the fact that the resources at the network layer, it necessitates across layer solution to the problem.

So somehow we need information to move between the two layers.

So here's a general plan.

So we're going to arrange for them to end layer at the sender to send at a rate.

This is going to be a rate that changes with time.

But let's say that the sender at some point in time sends at a rate, RI where RI is measured in bits per second.

And that's actually true here.

In case it wasn't clear, these loads are in bits per second.

The capacity is in bits per second as well.

So the plan is for the sender to send at a [layered?] rate, RI.

And, for all of the switches inside the network, to keep track in some fashion of whether they are being congested or not, and it's going to be really simple after that.

If the senders are sending too fast, or if a given sender is sending too fast, then we're going to tell them to slow down.

The network layer is somehow going to tell the sender to slow down.

And likewise, if they are sending too slow, and we have yet to figure out how you know that you're sending too slow, and that you could send a little faster, you're sending too slow there's going to be some plan by which the sender can speed up.

And the nature of our solution is going to be that there is really not going to be a steady rate at which senders are going to send.

In fact, by the definition of the problem, the steady rate is a [fool's errand?] because N varies N varies and the load varies.

So that means on any given link, the traffic varies with time.

So you really don't want a static rate.

What you would like to do is if there's any extra capacity, and the sender has load to fill that capacity, you want that sender to use that capacity.

So this rate is going to adapt and change with time in response to some feedback that we're going to obtain based on network layer information essentially from the network layer about whether people are sending too fast or people are sending too slow.

OK and all congestion control schemes that people have come up with, all algorithms for solving this problem, and there have been dozens if not a few hundred of them, all of these things in various variants of various solutions, all of them are according to the basic plan.

OK, the network layer gives some feedback.

If it's too fast, slow down.

If it's too slow, speed up.

The analogy is a little bit like a water pipeline network.

Imagine have all sorts of pipes feeding water in.

You have this massive pipeline network.

And what you can control are the valves at the end points.

And by controlling the valves, you can decide whether to let water rush in or not.

And anytime you're getting clogged, you have to slow down and close the valves.

So the devil's actually in the details.

So were going to dive in to actually seeing how we're going to solve this problem.

And the first component of any solution to congestion control is something we've already seen.

And it's buffering.

Any network that has asynchronous multiplex and, which Sam talked about the first time, any network that has the following behavior, at any given point in time you might have multiple packets arrive in to a given switch.

And, the switch can only send one of them out on an outgoing link at any given point in time, which means if you weren't careful, you would have to drop the other packet that showed up.

And so, almost everyone who builds a decent asynchronously multiplex network puts in some queues to buffer packets until the link is free so they can then send the packet out onto the link.

So the key question when it comes to buffering but you have to ask is how much?

I don't actually mean how much in terms of how expensive it is, but in terms of how much should the buffering be?

Should you have one packet of buffering?

Two packets?

Four packets?

And how big should the buffer be?

Well, the one way to answer the question of how much, first you can ask, what happens if it's too little?

So what happens if the buffering is too little in a switch?

Like you put one packet or two packets of buffering: what happens?

I'm sorry, what?

Well, congestion by definition has happened when the load exceeds the capacity.

So you get congestion.

But what's the effect of that?

So a bunch of packets show up.

You've got two packets of buffering.

[So you could lose?] packets.

This is pretty clear, right?

Good.

So if it's too little, what ends up happening is that you drop packets, which suggests that you can't have too little buffering, OK?

So, at the other end of the spectrum, you could just say, well, I'm going to design my network so it never drops a packet.

Memory is cheap.

We learned about Moore's Law.

So let's just over provision, also called too much buffering.

OK, it's not really that expensive.

So what happens if there's too much buffering?

Well, if you think about it, the only thing that happens when you have too much buffering as well, packets won't get lost.

But all you've done is traded off packet loss for packet delay.

Adding more buffering doesn't make your link go any faster.

They should probably learn that at Disneyland and places like that.

I mean, these lines there are just so long.

I mean, they may as well tell people to go away and come back later.

It's the same principle.

I mean, just adding longer lines and longer queues doesn't mean that the link goes any faster.

So it's really problematic to add excessive buffers.

And the reason is quite subtle.

The reason it actually has to do with something we talked about the last time, or at lease one reason, a critical reason has to do with timers and retransmissions.

Recall that all you do when you have too much buffering is you eliminate packet loss or at least reduce it greatly at the expense of increasing delays.

But the problem with increasing delays is that your timers that you're trying to set up to figure out when to retransmit a packet, you would like them to adapt to increasing delays.

So he you build this exponentially weighted moving average, and you pick a timeout interval that's based on the mean value, and the standard deviation.

And the problem with too much buffering is that it makes these adaptive timers extremely hard to implement because your timeout value has to depend on both the mean and standard deviation.

And if you have too much buffering, the range of round-trip time values is too high.

And the result of that you end up with this potential for something called congestion collapse.

And let me explain this with a picture.

So your adaptive timers are trying to estimate the round-trip time, and the average round-trip time in the standard deviation or at the linear deviation of the round-trip time, and at some point they're going to make a decision as to whether to retransmit a packet or not.

What might happen, and what does happen and has happened when you have too much networks with too much buffering is that you end up with a queue that's really big.

OK, and this is some link on which the packet's going out, and you might have packet one sitting here, and two sitting here, and three, and all the way out.

There is a large number of packets sitting there.

Notice that the end to end sender is trying to decide whether packet one for whose acknowledgment it still hasn't heard.

It's trying to decide whether one is still in transit, or has actually been dropped.

And it should retransmit the packet only after one has been dropped.

But if you have too much buffering, the range of these values is so high that it makes these adaptive timers quite difficult to tune.

And the result often is that one is still sitting here.

But it had been stuck behind a large number of packets.

So the delay was extremely long.

And the end to end sender timed out.

And when it times out, get retransmits one into the queue.

And, soon after that, often it might retransmit two, and retransmit three, and retransmit four, and so on.

And these packets are sort of just stuck in the queue.

They're not actually lost.

And if you think about what will happen then, this link, which is already a congested link, because queues have been building up here, long queues have been building up, this link is not starting to use more and more of its capacity to send the same packet twice, right, because it sent one out.

And the sender retransmitted it thinking it was lost when it was just stuck behind a long queue.

And now one is being sent again.

And two is being sent again, and so on.

And graphically, if you look at this, what you end up with is a picture that looks like the following.

This picture plots the total offered load on the X axis, and the throughput of the system on the Y axis where the throughput is defined as the number of useful bits per second that you get.

So if you send the packet one twice, only one of those packets is actually useful.

Now, initially when the offered load is low and the network is not congested, and the offered load is less than the capacity of the link, this curve is just a straight line with slope one, right, because everything you offer is below the link's capacity.

And it's going through.

Now at some point, it hits the link's capacity, right, the bottleneck link's capacity.

And after that, any extra offered load that you pump into the network is not going to go out any faster.

The throughput is still going to remain the same.

And, it's just going to be flat for a while.

And the reason it's flat is that queues are building up.

And, that's one reason that you can't send things any faster.

The only thing that's going on is that queues are building up.

So this curve remains flat.

Now, in a decent network that doesn't have congestion collapse, if you don't do anything else, but somehow manage to keep the system working here, this curve might remain flat forever.

No matter what the offered load, you know, you can pump more and more load in the system.

And the throughput remains flat at the capacity.

But the problem is that it has interactions with things that the higher layer is are doing, such as these retransmissions and timers.

And, eventually, more and more of the capacity starts being used uselessly for these redundant transmissions.

And you might end up in a situation where the throughput dies down.

OK, and if that happens, this situation is called congestion collapse.

There is more and more work being presented to the system.

If you reach a situation where the actual amount of useful work that's been done starts reducing as more work gets presented to the system, that's the situation of congestion collapse.

And this kind of thing shows up in many other systems, for example, in things like Web servers that are built out of many stages where you have a high load presented [at?] the system, you might see congestion collapse in situations where, let's say you get a Web request.

And what you have to do is seven stages of processing on the Web request.

But what you do with the processor is you take a request, and you process it for three of those stages.

And then you decide to go to the next request and process at three stages.

And, you go to the next request [that?] passes at three stages.

So, [no?] request gets complete.

Your CPU's 100% utilized.

But the throughput is essentially diving down to zero.

That's another situation where you have congestion collapse.

But when it occurs in networks, it turns out it's more complicated to solve the networks because of these reasons that I outlined before.

So has everyone seen this?

So let's go back to our problem.

Our goal is to, we want this thing to be true.

The aggregate offered load to be smaller than the capacity of a link for every link in the system.

But it turns out that's not enough of a specification because there are many ways to achieve this goal.

For example, a really cheesy way to achieve this goal is to make sure that everybody remains quiet.

If nobody sends any packets, you're going to get that to be true.

So we're going to actually have to define the problem a little bit more completely.

And let's first define what we don't want.

The first thing we don't want is congestion collapse.

So any solution should have the property that it never gets into that situation.

And in fact, good congestion control schemes operate at the left knee of the curve.

But we're not going to get too hung up on that because if we operate a little bit in the middle, we're going to say that's fine because that'll turn out to be good enough in practice.

And often is just fine.

And the additional complexity that you might have to bring to bear on a solution to work nearer [the knee?] might not be worth it.

OK, but really we're going to worry about not falling off the cliff at the right edge, OK?

And then having done that, we're going to want reasonable utilization also called efficiency.

So, what this says is that if you have a network link that's often congested, you want to make sure that that link isn't underutilized when there is offered load around.

So for example, people are presenting 100 kb per second of load, and the network link has that capacity; you want that to be used.

You don't want to shut them up too much.

So what this really means in practice is that if you have slowed down, and excess capacity has presented itself, then you have to make sure that you speed up.

So that's what it means in practice.

And the third part of the solution is there's another thing we need to specify.

And the reason is that you can solve these two problems by making sure that only one person transmits in the network.

OK, if that person has enough offered load, then you just got everybody out altogether, and essentially allocate that resource in sort of a monopolistic fashion to this one person.

That's not going to be very good because we would like our network to be used by a number of people.

So I'm going to define that as a goal of any good solution.

I'm going to call it equitable allocation.

I'm not going to say fair allocation because fair suggests that it's really a strong condition that every connection gets a roughly equal throughput if it has that offered load.

I mean, that turns out to be, I think, in my opinion, achieving perfect fairness to TCP connections is just a waste of time because in reality, fairness is governed by who's paying what for access to the network.

So we're not going to get into that in this class.

But we are going to want solutions that don't eliminate, don't starve certain connections out.

And we'll be happy with that because that will turn out to work out just fine in practice.

Now, to understand this problem a little bit better, we're going to want to understand this requirement a little bit more closely.

And the problem is that this requirement of the aggregate rate specified by the offered load, being smaller than the link capacity is a condition on rates.

It just says the offered load is 100 kb per second.

The capacity is 150 kb per second.

That means it's fine.

The problem is that you have to really ask offered load over what timescale?

For example, if the overall offered load on your network is, let's say, a megabit per second, and the capacity of the network is half a megabit per second, and in that condition lasts for a whole day, OK, so for a whole day, your website got slash dotted.

Take that example.

And you have this little wimpy access link through your DSL line.

And, ten times that much in terms of requests are being presented to your website.

And, it lasts for the whole day.

Guess what.

Nothing we're going to talk about is really going to solve that problem in a way that allows every request to get through in a timely fashion.

At some point, you throw up your hands and say, you know what?

If I want my website to be popular, I'd better put it on a 10 Mb per second network and make sure that everybody can gain access to it.

So we're not really going to solve the problem at that time scale.

On the other hand, if your website suddenly got a little bit popular, and a bunch of connections came to it, but it didn't last for a whole day, but it lasted for a few seconds, we're going to want to deal with that problem.

So if you think about it, there are three timescales that matter here.

And these timescales arise in an actual way because of this network where congestion happens here.

And then we're going to have some feedback, go back to the sender, and the sender exercises control.

And the only timescale in this whole system is the round-trip time because that's the timescale, the order of magnitude of the timescale around which any feedback is going to come to us.

So there are three timescales of interest.

There is smaller than one round-trip time, which says this inequality is not satisfied for really small.

The time where there is a little spurt, a burst, it's also called a burst, a burst of packets show up, and then you have to handle them.

But then after that, things get to be fine.

So there is smaller than one round-trip time.

So this is summation LI is greater than C, i.e. the network is congested.

It could be congested at really short durations that are smaller than a roundtrip time.

It could be between one and I'm going to say 100 round-trip times.

And just for real numbers, a round-trip time is typically order of 100 ms.

So we are talking here of less than 100 ms up to about ten seconds, and then bigger than this number, OK?

Bigger than 100. And these are all orders of magnitude.

I mean, it could be 500 RTT's.

OK, those are the three time scales to worry about.

When congestion happens at less than one roundtrip time, we're going to solve that problem using this technique.

We're going to solve it using buffering, OK?

And, that's the plan.

The reason is that it's really hard to do anything else because the congestion lasts for certain burst.

And by the time you figure that out and tell the sender of that, the congestion has gone away, which means telling the sender that the congestion has gone away, which means telling the sender that the congestion has gone away was sort of a waste because the sender would have slowed down.

But the congestion anyway went away.

So, why bother, right?

Why did you tell the sender that?

So that's the kind of thing you want to solve at the network layer inside the switches.

And that's going to be done using buffering.

And that sort of suggests, and there's a bit of sleight-of-hand here.

And we're not going to talk about why there's a sleight-of-hand. But this really suggests that if you design a network and you want to put buffering in the network, you'd better not put more than about a round-trip time's worth of buffering in that switch.

If you put buffering longer than a round-trip time, you're getting in the way of things that the higher layers are going to be doing.

And it's going to confuse them.

In fact, Sam showed you this picture of these round-trip times varying a lot between the last lecture where the mean was two and a half seconds.

And the standard deviation was one and a half seconds.

That was not a made-up picture.

That was from a real wireless network where the designers had incorrectly put a huge amount of buffering.

And this is extremely common.

Almost every modem that you buy, cellular modem or phone modem, has way too much buffering in it.

And, the only thing that happens is these queues build up.

It's just a mistake.

So less than one roundtrip time: deal with it using buffering.

Between one and 100 round-trip times, we're going to deal with that problem using the techniques that we are going to talk about today, the next five or ten minutes.

And then bigger than 100 round-trip times or 1,000 round-trip times are things where congestion is just sort of persistent for many, many, many seconds or minutes or hours.

And there are ways of dealing with this problem using protocols and using algorithms.

But ultimately you have to ask yourself whether you are really under provisioned, and you really ought to be buying or provisioning your network to be higher, maybe put your [UNINTELLIGIBLE] on a different network that has higher capacity.

And these are longer times congestion effects for which decisions based on provisioning have to probably come to play in solving the problem.

So, provisioning is certainly an important problem.

And when you have congestion lasting really long periods of time, that might be the right solution.

But for everything else, there is solutions that are much easier to understand, or at least as far as using the tools that we've built so far in 6.033. So, the first component of the solution is some buffering to deal with the smaller than one round-trip time situation.

And then when your buffers start getting filled up, and congestion is lasting for more than a round-trip time, then your buffers start getting filled up.

At that point, you're starting to see congestion that can't be hidden by pure buffering.

And that can't be hidden because queues are building up, and that's causing increased delay to show up at the sender, and perhaps packets may start getting dropped when the queue overflows.

And that causes the sender to observe this congestion, which means that what we want is a plan by which, when congestion happens lasting over half a round-trip time or close to a round-trip time, we're going to want to provide feedback to the sender, OK?

And then the third part of our solution is when the sender gets this feedback, what it's going to do is adapt.

And the way it's going to adapt is actually easy.

It's going to do it by modifying the rate at which it sends packets.

And that's what are set up.

The sender sends at rate RI.

What you do is you change the rate or the speed at which the sender sends.

So, there's two things we have to figure out.

One is, how does the network give feedback?

And the second is, what exactly is going on with changing the speed?

How does it work?

There are many ways to give feedback, and sort of the first order of things you might think about are, well, when the queue starts to fill up, I'll send a message to the sender, or I'll send a bit in the packet header and get to the endpoint, and it'll send this feedback back.

And any variant that you can think of in the next five or ten minutes I can assure you has been thought of and investigated.

And you might think of new ways of doing it which would be great.

I mean, this is a topic of active work.

People are working on this stuff still.

But my opinion is that the best way to solve this problem of feedback is the simplest possible thing you could think of.

The simplest technique that really works all the time is just drop the packet.

In particular, if the queue overflows, and it's going to get dropped anyway, and that's a sign of congestion, but if at any point in time you decide that the network is getting congested and remains so for a long enough time scale that you want the sender to react, just throw the packet away.

The last thing about throwing a packet away is that it's extremely hard to implement it wrong because when the queue overflows, the packet's going to be dropped anyway.

So that feedback is going to be made available to the sender.

So what we're going to assume for now is that all packet drops that happen in networks are a sign of congestion, OK?

And when a packet gets dropped, the sender gets that feedback.

And, the reason it gets that feedback is, remember that every packet's being acknowledged.

So if it misses an acknowledgment, then it knows that the packet's been dropped.

And it says, ah, the network is congested.

And then I'm going to change my speed.

And, I'm going to change my speed by reducing the rate at which I'm going to send packets.

So now, we have to ask how the sender adapts to congestion, how the speed thing actually works out.

Well?

It says up.

It's already up.

Yeah, it doesn't have a down button.

What's that?

Sam?

How many professors does it take to -- He turned the thing off.

He turned the light off on that.

This might actually work.

Can you see that?

I bet it's a race condition.

All right, so, can you see that?

OK, let's just move from here.

OK, the way we're going to figure out the rate at which we're going to send our packets is to use this idea of a sliding window that we talked about the last time.

And since Sam did such a nice animation, I'm going to display that again, just a refresher.

Wonderful.

All right.

[APPLAUSE] And since Sam made this nice animation, I'm going to play that back.

I've always wanted to say, play it again, Sam.

So, what's going on here is that whenever the receiver receives a packet, it sends in acknowledgment back.

And the sender's pipelining packets going through.

And whatever the sender gets an acknowledgment, it sends a new packet off.

And that's where you see happening.

Whenever it gets an acknowledgment, a new packet goes out onto the other end.

And, the window's sliding by one every time it gets an acknowledgement.

So this is sort of the way in which this steady-state thing works out for us in this network.

So the main point to note about this is that this scheme has an effect that I'll call self pacing.

OK, it's self pacing because the sender doesn't actually have to worry very much about when it should exactly send a packet because it's being told every time an acknowledgment arrives, it's being told that the reason I got an acknowledgment is because one packet left the network, which means I can send one packet into the network.

And as long as things weren't congested before, they won't be congested now, right, because packets have left the pipe.

For every packet that leaves, you're putting one packet in.

This is an absolutely beautiful idea called self-pacing. And, the idea here is that [acts?] essentially [strobe?] data packets.

And, like all a really nice ideas, it's extremely simple to get it.

And it turns out to have lots of implications for why the congestion control system is actually stable in reality.

So what this means, for example, is let's say that things are fine.

And, all of a sudden a bunch of packets come in from somewhere else, and the network starts to get congested.

What's going to happen is that queues are going to start building up.

But, when queues start building up, what happens is that transmissions that are ongoing for this connection are just going to slow down a little bit because they're going to get interleaved with other packets in the queue, which means the acts are going to come back slower because how can the acts come back any faster than the network can deliver them, which means automatically the sender has a slowing down effect dealing with transient congestion.

This is a really, really nice behavior.

And as a consequence of the way in which the self pacing of, the way in which the sliding window thing here works.

So now, at some point the network might become congested, and a packet may get dropped.

So, the way that manifests itself as one packet gets dropped, the corresponding acknowledgment doesn't get back to the sender.

And when an acknowledgment doesn't get back to the sender, what the sender does is reduce its window size by two.

OK, so I should mention here there are really two windows going on.

One window is the window that we talked about last time where the receiver tells the sender how much buffer space that it has.

And the other window, another variable that we just introduced which is maintained by the sender, and it's also called the congestion window, OK?

And this is a dynamic variable that constantly changing as the sender gets feedback from the receiver.

And in response to a missing act when you determine that a packet is lost, you say its congestion is going on.

And I want to reduce my window size.

And there are many ways to reduce it.

We're going to just multiplicatively decrease it.

And the reason for that has to do with the reason why, for example, on the Ethernet you found that you were doing exponential back off.

You're sort of geometrically increasing the spacing with which you are sending packets there.

You're doing the same kind of thing here reducing the rate by a factor of two.

So that's how you slowdown.

At some point you might get in a situation where no acknowledgments come back for a while.

Say, many packets gets lost, right?

And that could happen.

All of a sudden there's a lot of congestion going on.

You get nothing back.

At that point, you've lost this ability for acknowledgments to strobe new data packets.

OK, so the acts have acted as a clock allowing you to strobe packets through.

And you've lost that.

At this point, your best plan is just to stop.

And you halt for a while.

It's also called a timeout where you just be quiet for awhile, often on the order of a second or more.

And then you try to start over again.

Of course, now we have to figure out how you start over again.

And that's the same problem you have when you start a connection in the beginning.

When a connection starts up, how does it know how fast to send?

It's the same problem as when many, many packet drops happen, and you timeout.

And, in TCP, that startup is done using something called, a technique called slow start.

And I'll describe the algorithm.

It's a really elegant algorithm, and it's very, very simple to implement.

So you have a sender and receiver.

And the plan is initially, the sender wants to initiate a connection to the receiver.

It sends a request.

It gets the response back with the flow control window.

It just says how much buffer space the receiver currently has.

OK, we're not going to use that anymore in this picture.

But it's just there saying, which means the sender can never send more than eight packets within any given roundtrip.

Then what happens is the sender sends the first segment, and it gets an act.

So, initially this congestion window value starts off at one.

OK, so you send one packet.

One segment, you get an acknowledgment back.

And now the plan is that the algorithm is as follows.

For every acknowledgment you get back, you increase the congestion window by one.

OK, so what this means is when you get this acknowledgment back, you send out two packets.

And each of those two packets and sends you back two acknowledgements.

Now, for each of those two acknowledgements, you increase the window by one, which means that after you got both those acknowledgements, you've increased your window by two.

In previously, the window was two.

So, now you can send four packets.

So what happens is in response to the rape acknowledgment, you send out two packets.

In response to the light blue acknowledgement, you send out two more packets.

And now, those get through to the other side, they send back acknowledgments to you.

In response to each one of those, you increase the window by one.

So increasing the window by one on each acknowledgment means you can send two packets.

The reason is that one packet went through.

And you got an acknowledgment.

So in response to the acknowledgment, you could send one packet to fill in the space occupied by the packet that just got acknowledged.

In addition, you increase the window by one, which means you can send one more packet.

So really, what's going on with this algorithm is if on each act, you send two segments, which means you increase the window by one, the congestion window by one, really what happens is the rate at which this window opens is exponential in time because in every roundtrip, from one roundtrip to the next, you're doubling the entire window, OK?

So although it's called slow start, it's really quite fast.

And, it's an exponential increase.

So if you put it all together, it will turn out that the way in which algorithms like the TCP algorithm works is it starts off at a value of the congestion window.

If this is the congestion window, and this is time, it starts off at some small value like one.

And, it increases exponentially.

And then, if you continually increase exponentially, this is a recipe for congestion.

It's surely going to overflow some buffer and slowdown.

So what really happens in practice is after a while, based on some threshold, it just automatically decides it's probably not worth going exponentially fast.

And, it turns out slows down to go at a linear increase.

So every round-trip time it's increasing the window by a constant amount.

And then at some point, congestion might happen.

And when congestion happens, we drop the window by a factor of two.

So, we cut down to a factor of two of the current window.

And then, we continue to increase linearly.

And then you might have congestion coming again.

For instance, we might have lost all of the acknowledgements, not got any of the acknowledgements, losing a lot of packets, which means we go down to this thing where we've lost the act clock.

So, we have to time out.

And, it's like a new connection.

So, we remain silent for a while.

And this period is the time out period that's governed by the equations that we talked about the last time based on the roundtrip time and the standard deviation.

We remain quiet for a while, and then we start over as if it's a new connection.

And we increase exponentially.

And this time, we don't exponentially increase as we did the very first time.

We actually exponentially increase only for a little bit because we knew that congestion happened here.

So, we go up to some fraction of that, and then increase linearly.

So the exact details are not important.

The thing that's important is just understanding that when congestion occurs, you will drop your window by a factor [of?] some multiplicative degrees.

And at the beginning of connection, you're trying to quickly figure out what the sustainable capacity is.

So he you are going in this increase phase here.

But after awhile, you don't really want to be exponentially increasing all the time because that's almost guaranteed to overrun buffers.

So you want to go into a more [ginger mode?].

And, people have proposed congestion control schemes that do all sorts of things, all sorts of functions.

But this is the thing that's just out there on all of the computers.

And it works remarkably well.

So, there's two basic points you have to take home from here.

The main goal of congestion control is to avoid congestion collapse, which is the picture that's being hidden behind that, which is that as often load increases, you don't want throughput to drop like a cliff.

And this is a cross layer problem.

And the essence of all solutions is to use a combination of network layer feedback and end system control, end to end control.

And, good solutions work across eight orders of magnitude of bandwidth, and four orders of magnitude of delay.

So, with that, we'll start. DP1 is due tomorrow.

Have a great spring break, and see you after spring break.