MITOCW | ocw-6.033-lec13

0:00:00 So I'm back. I guess by your presence here 0:00:03.044 you've shown that for you people, DP1 is well under 0:00:06.757 control. And that's great. 0:00:08.613 OK, so today is the last lecture on this topic on the 0:00:12.475 networking piece of 6.033. And, the topic for today is 0:00:16.41 something called congestion control. 0:00:19.009 And what we're going to do is spend most of today talking 0:00:23.168 about congestion control. But let me remind you of where 0:00:27.252 we are in networking. And so if there's one thing you 0:00:31.917 take away from 6.033 from this networking piece you should 0:00:35.443 remember this picture. So the way we're dealing with 0:00:38.597 networking, and the way we always deal with networking in 0:00:42.061 any complicated system is to layer network protocols. 0:00:45.278 And the particular learning model that we picked for 6.033 0:00:48.804 is a subset of what you'd see out in the real world. 0:00:51.958 And it's mostly accurate. There is something called the 0:00:55.298 link layer which deals with transmitting packets on the 0:00:58.639 link. And of that, 0:01:00.721 you have the network layer. And, the particular kind of 0:01:03.967 network layer that we are talking about is a very popular 0:01:07.333 one which provides a kind of service called a best effort 0:01:10.699 service. And the easiest way to 0:01:12.502 understand best effort services: it just tries to get packets 0:01:16.109 through from one end to another. But it doesn't get all panicked 0:01:19.896 if it can't get packets through. It just lets a higher layer 0:01:23.442 called the end to end layer deal with any problems such as lost 0:01:27.169 packets, or missing packets, or corrupted packets, 0:01:30.114 reordered packets, and so on. 0:01:33 And last time when Sam gave the lecture, he talked about a few 0:01:36.931 things that the end to end layer does. 0:01:39.316 And in particular we talked about how the end to end layer 0:01:42.989 achieves reliability using acknowledgments. 0:01:45.696 So the receiver acknowledges packets that it receives from 0:01:49.37 the sender, and the sender, if it misses an acknowledgment 0:01:53.043 goes ahead and retransmits the packet. 0:01:55.428 And in order to do that, we spent some time talking 0:01:58.65 about timers because you have to not try to retransmit 0:02:02.066 immediately. You have to wait some time 0:02:05.755 until you're pretty sure that the packet's actually lost 0:02:09.47 before you go ahead and try to retransmit the packet. 0:02:12.981 The next concept at the end to end layer that we talked about 0:02:17.033 is something called a sliding window, where the idea was that 0:02:21.085 if I send you a packet, got an acknowledgment back, 0:02:24.462 and then sent you the next packet and did that, 0:02:27.568 things are really slow. And all that a sliding window 0:02:31.89 is, is really an idea that you've already seen from an 0:02:35.344 earlier chapter, and probably from 6.004 called 0:02:38.342 pipelining, which is to have multiple outstanding things in 0:02:42.122 the pipe or in the network at once as a way to get higher 0:02:45.772 performance. And the last thing that we 0:02:48.249 talked about the last time was flow control. 0:02:50.791 And the idea here is to make sure that the sender doesn't 0:02:54.441 send too fast because if it's sent really fast, 0:02:57.439 it would swamp the receiver, which might be slow trying to 0:03:01.154 keep up processing the sender's packets. 0:03:05 So you don't want to swamp the receiver's buffer. 0:03:07.728 And we talked about how with every acknowledgment, 0:03:10.513 the receiver can piggyback some information about how much space 0:03:14.094 it has remaining in its buffer. And if that clamped down to 0:03:17.39 zero, then the sender would automatically slow down. 0:03:20.289 You would produce what's also known as back pressure back to 0:03:23.642 the sender's saying, "I don't have any more space, 0:03:26.428 slow down." And the sender guarantees that 0:03:28.701 it won't send more packets at any given time, 0:03:31.202 more data at any given time than what the receiver says it 0:03:34.442 can handle in its buffer. So that was the plan. 0:03:38.391 Those are the things we talked about so far for the end to end 0:03:42.431 layer. And what we're going to do 0:03:44.551 today is to go back to one of the main things about networks 0:03:48.459 that was mentioned during the first networking lecture and 0:03:52.235 talk about how we achieve that, which is sharing. 0:03:55.414 Ultimately, it's extremely inefficient to build networks 0:03:59.057 where every computer is connected to every other 0:04:02.17 computer in the world by a dedicated link or a dedicated 0:04:05.814 path of its own. Fundamentally, 0:04:09.027 networks are efficient only if they allow computers connected 0:04:13.433 to each other to share paths underneath. 0:04:16.297 And the moment you have sharing of links, you have to worry 0:04:20.555 about sharing the resources, namely sharing the bandwidth of 0:04:24.888 a link and that's what we're going to do today. 0:04:29 And we're going to do this. Basically the goal for today is 0:04:32.73 to talk about the problems that arise if you don't you do 0:04:36.333 sharing properly, and then spend some time 0:04:38.97 talking about how we solve these problems. 0:04:41.608 So imagine you have a network. And I'm going to start with a 0:04:45.403 simple example. And we're going to use that 0:04:48.105 example to write through, because it will turn out that 0:04:51.578 the simple example will illustrate the essential problem 0:04:55.116 with sharing. So imagine you have a bunch of 0:04:57.883 computers connected to one end of the network. 0:05:02 And at the other end you have other computers. 0:05:04.954 And imagine that these are senders and these are receivers. 0:05:08.761 And they share the network. And you know a really simple 0:05:12.371 form of sharing this network might be when you have all of 0:05:16.113 these computers, take their links like the 0:05:18.805 Ethernet connections and hook them up to a switch. 0:05:22.021 And maybe you hook it up to another switch. 0:05:24.778 And then there's other paths that eventually take you to the 0:05:28.652 receivers that you want to talk to. 0:05:32 And imagine just for example that these are fixed lengths. 0:05:36.1 So for example these might be 100 Mb per second links. 0:05:39.98 And then you go out to your dorm room and then you have something connected, and that might be, 0:05:43.933 let's say it's a megabit per second link. 0:05:46.862 So this might be 1 Mb per second. 0:05:49.205 And these things are 100 Mb per second. 0:05:51.987 And of course you could have receivers connected with 0:05:56.087 extremely fast links as well, which means that if the sender 0:06:00.407 and the receiver just looked at their own access links, 0:06:04.36 these 100 Mb per second links, and just thought to themselves, 0:06:08.826 well, I have 100 Mb per second links. 0:06:13 The sender has 100 Mb per second link, so clearly we could 0:06:16.714 exchange data at 100 Mb per second. 0:06:18.929 That would be flawed because all of these things go through 0:06:22.709 this relatively thin pipe of a megabit per second. 0:06:25.902 So really, the goal here is to take all of these connections, 0:06:29.812 all of these end to end transfers that might be 0:06:32.809 happening at any given point in time, and share every link in 0:06:36.719 the network properly. And I'll define what we mean by 0:06:41.02 properly as we go along. So let's write some notation 0:06:44.523 first because it will help us pose the problem clearly. 0:06:48.16 So let's say that there is some offered load that these 0:06:51.797 different senders offered to the network. 0:06:54.492 So you've decided to download a bunch of files, 0:06:57.59 music files, and WebPages. 0:06:59.274 And each of those has a certain load that it's a certain size of 0:07:03.518 a file. A sender can actually push that 0:07:07.281 data through at some rate. So that's the offered load on 0:07:11.329 the network. And let's assume that senders 0:07:14.347 one through N here, and let's say that the offered 0:07:18.248 load of the sender is L1. This is L2, all the way through 0:07:22.37 LN. OK, so in this simple picture, 0:07:24.799 the total offered load on the network along this path is the 0:07:29.142 summation of LI's, where I runs from one to N, 0:07:32.454 right? And out here, 0:07:34.702 in this simple picture, we have all this offered load 0:07:37.312 going through this one common link, which we're going to call 0:07:40.324 the bottleneck link. We don't actually know where; 0:07:42.783 the senders don't know where the bottleneck is because it's 0:07:45.694 not near them. And in general, 0:07:47.15 it's not near them. It could be, 0:07:48.706 but in general, they don't know where it is. 0:07:50.864 The receivers don't know where it is. 0:07:52.671 But there is some bottleneck link that's going to throttle 0:07:55.532 the maximum rate at which you could send data. 0:07:57.791 Let's call that rate C. So what we would like to be 0:08:01.537 able to do is ensure that at all points in time that the sum of 0:08:05.069 the load that's being offered by all senders that share any given 0:08:08.715 link is less than the capacity of that link. 0:08:11.164 And so, C here, this picture, 0:08:12.759 this would be C. And notice that we don't 0:08:15.037 actually know where these things are. 0:08:17.088 And for different connections, if you are surfing the web and 0:08:20.506 downloading music and so on, the actual bottleneck for each 0:08:23.81 of those transfers might in general be different. 0:08:26.544 But the general goal for congestion control is you look 0:08:29.62 at any path between a sender and receiver, and there is, 0:08:32.753 in general, some bottleneck there. 0:08:36 And you have to ensure that for every bottleneck link, 0:08:41.18 in fact for every link, the total offered load 0:08:45.578 presented to that link satisfies this relationship. 0:08:50.466 OK, so that's ultimate goal. We're not going to be able to solve this, but we'll come prefectly close. 0:08:55.157 OK? So that's the statment of the 0:08:58.285 problem, that you want this. 0:09:02 0:09:08 This is not a trivial problem. So one reason it's not trivial 0:09:12.155 is something that might have been clear to you from the 0:09:15.894 definition of the problem. You want to share this for all 0:09:19.772 the links in the network. And the network in general 0:09:23.304 might have, or will have, millions of links, 0:09:26.282 and hundreds of millions of hosts. 0:09:28.567 So any solution you come up with has to be scalable. 0:09:33 And in particular, it has to scale to really large 0:09:36.404 networks, networks that are as big as the public Internet, 0:09:40.364 and the whole public Internet 10 years from now. 0:09:43.629 So it has to be something that scales. 0:09:46.2 It has to handle large values. So it has to handle large 0:09:50.021 networks. It has to handle large values 0:09:52.661 of N, where the number of N to N connections that are 0:09:56.273 transferring data on this network, and it's an unknown 0:09:59.955 value of N. You don't really know what N is 0:10:02.873 at any given point in time. And it could be extremely 0:10:07.751 large. So you have to handle that. 0:10:10.062 And above all, and this is actually the most 0:10:13.074 important point. And it's often missed by a lot 0:10:16.297 of descriptions of congestion control and sharing. 0:10:19.729 A good solution has to scale across many orders of magnitude 0:10:23.862 of link properties. On the Internet or in any 0:10:26.944 decent packet switch network, link rates vary by perhaps 0:10:30.797 seven, or eight, or nine orders of magnitude. 0:10:35 A single network could have links that send data in five or 0:10:38.766 ten kilobits per second on one extreme, and 10 Gb per second, 0:10:42.662 or 40 Gb per second on the other extreme. 0:10:45.259 And a single path might actually go through links whose 0:10:48.766 capacities vary by many orders of magnitude. 0:10:51.558 And it's extremely important that a solution work across this 0:10:55.454 range because otherwise its generality isn't that general. 0:11:00 And what you would like is a very general solution. 0:11:03.579 The second reason this problem is hard is that here we have 0:11:07.732 this N, but really N is a function of time. 0:11:10.739 You're surfing the Web. You click on a link, 0:11:13.817 and all of a sudden 18 embedded objects come through. 0:11:17.54 In general, each of those is a different connection. 0:11:21.191 And then it's gone. And the next time somebody else 0:11:24.771 clicks on something, there's a bunch more objects. 0:11:28.279 So N varies with time. And as a consequence, 0:11:32.729 the offered load on the system, both the total offered load as 0:11:37.752 well as the load offered by single connection varies with 0:11:42.364 time. So, LI varies with time as 0:11:44.917 well. So it's kind of really hard to 0:11:47.799 understand what the steady-state behavior of a large network like 0:11:53.07 the Internet is because there is no steady-state behavior. 0:11:57.764 And the third reason why this problem is hard is the thing that we're going to come 0:12:02.705 back to and when we talk about a solution. The person who can control the 0:12:07.729 rate at which data is being sent is the sender. 0:12:13 And maybe the receiver can help it control it by advertising 0:12:16.571 these flow control windows. So the control happens at these 0:12:20.082 points. But if you think about it, 0:12:22.079 the resource that's being shared is somewhere else. 0:12:25.106 It's far away, right? 0:12:26.317 So, there is a delay between the congestion point, 0:12:29.283 or the points where overload happens with these points in the 0:12:32.915 network, and where the control can be exercised to control 0:12:36.365 congestion. And this situation is extremely 0:12:39.584 different from most other resource management schemes for 0:12:42.446 other computer systems. If you talk about processor 0:12:45.002 resource management, well, there's a schedule of the 0:12:47.608 controls, what runs on the processor. 0:12:49.448 The processor is right there. There's no long delay between 0:12:52.413 the two. So if the operating system 0:12:54.151 decides to schedule another process, well, 0:12:56.399 it doesn't take a long time before that happens. 0:12:58.802 It happens pretty quickly. And nothing else usually 0:13:02.383 happens in the meantime related to something else, 0:13:05.209 some other process. If you talk about disk 0:13:07.573 scheduling, well, the disk scheduler is very, 0:13:10.11 very close to the disk. It can make a decision, 0:13:12.762 and exercises control according to whatever policy that it 0:13:16.049 wants. Here, if the network decides 0:13:18.009 it's congested, and there is some plan by which 0:13:20.662 the network will want to react to that load, 0:13:23.141 well, it can't do very much. It has to somehow arrange for 0:13:26.428 feedback to reach the points of control, which are in general 0:13:29.887 far away, and are not just far away in terms of delay, 0:13:32.943 but that delay varies as well. That's what makes the problem 0:13:37.87 hard, which is that the resource is far from the control point. 0:13:42 0:13:52 So these are the congestion points, which is where the 0:13:55.275 resources are. And these are the control 0:13:57.685 points, which is where you can exercise. 0:14:00.095 And the two are separated geographically. 0:14:03 I don't think I mentioned this. This is what you want. 0:14:06.47 And any situation where the summation of LI, 0:14:09.285 the offered load on a link, is larger than the capacity, 0:14:12.886 that overload situation is what we're going to call congestion. 0:14:16.946 OK, so any time you see LI bigger than C, 0:14:19.565 we're going to define that as congestion. 0:14:22.184 And then, I'll make this notion a little bit more precise. 0:14:25.916 And you'll see why we have to make it slightly more precise. 0:14:29.779 But for now, just say that if inequality is 0:14:32.529 swapped, that means it's congested. 0:14:36 OK, so that's the problem. We are going to want to solve 0:14:39.289 the problem. Now, every other problem that 0:14:41.742 we've encountered in networking so far, we've solved by going 0:14:45.33 back to that layered picture and saying, well, 0:14:48.022 we need to deliver, for example, 0:14:49.876 if you want to deliver packets across one link, 0:14:52.628 we say all right, we'll take this thing as a link 0:14:55.499 layer protocol. We'll define framing on top of 0:14:58.19 it, and define a way by which we can do things like 0:15:01.181 error correction if we need to on that link, 0:15:04.411 and we'll solve the problem right there. 0:15:08 Then we say, OK, we need to connect all 0:15:09.874 these computers up together and build up routing tables so we 0:15:12.834 can forward data. And we say, OK, 0:15:14.413 that's the network layer's problem. 0:15:16.091 We're going to solve the problem there. 0:15:17.965 And then packets get lost. Well, we'll deal with it at the 0:15:20.777 end to end layer. And, that model has worked out 0:15:23.096 extremely well because it allows us to run arbitrary applications 0:15:26.254 on top of this network layer without really having to build 0:15:29.115 up forwarding tables for every application anew and run it on 0:15:32.075 all sorts of links. And you can have paths 0:15:35.313 containing a variety of different links. 0:15:37.641 And everything works out like a charm. 0:15:39.85 But the problem with doing that for congestion control is that 0:15:43.492 this layered picture is actually getting in the way of solving 0:15:47.134 congestion control in a very clean manner. 0:15:49.582 And the reason for that is that the end to end layer runs at the 0:15:53.343 end points. And those are the points were 0:15:55.731 control is exercised to control the rate at which traffic is 0:15:59.253 being sent on to the network. But the congestion, 0:16:02.989 and the notice of any information about whether the 0:16:05.593 network is overloaded is deeply buried inside the network at the 0:16:08.875 network layer. So, what you need is a way by 0:16:11.114 which information from the network layer about whether 0:16:13.875 congestion is occurring, or whether congestion is not 0:16:16.583 occurring, or whether congestion is likely to occur even though 0:16:19.812 it's not yet occurred, that kind of information has 0:16:22.416 somehow to percolate toward the end to end layer. 0:16:24.916 And so far, we've modularized this very nicely by not really 0:16:27.989 having very much information propagate between the layers. 0:16:32 But now, to solve this problem of congestion, 0:16:35.097 precisely because of this separation between the resource 0:16:38.827 and the control point, and the fact that the control 0:16:42.417 point is at the end to end layer, at least in the way we're 0:16:46.5 going to solve the problem. And the fact that the resources 0:16:50.582 is at the network layer, it necessitates across layer 0:16:54.031 solution to the problem. So somehow we need information 0:16:57.832 to move between the two layers. So here's a general plan. 0:17:03.002 So we're going to arrange for the end to end layer at the sender 0:17:07.809 to send at a rate. This is going to be a rate that 0:17:11.734 changes with time. But let's say that the sender 0:17:15.5 at some point in time sends at a rate, RI where RI is measured in 0:17:20.627 bits per second. And that's actually true here. 0:17:24.312 In case it wasn't clear, these loads are in bits per 0:17:28.397 second. The capacity is in bits per 0:17:31.903 second as well. So the plan is for the sender 0:17:35.006 to send at a rate, RI. 0:17:37.262 And, for all of the switches inside the network, 0:17:40.576 to keep track in some fashion of whether they are being 0:17:44.384 congested or not, and it's going to be really 0:17:47.487 simple after that. If the senders are sending too 0:17:50.871 fast, or if a given sender is sending too fast, 0:17:54.115 then we're going to tell them to slow down. 0:17:57.076 The network layer is somehow going to tell the sender to slow 0:18:01.307 down. And likewise, 0:18:03.77 if they are sending too slow, and we have yet to figure out 0:18:07.209 how you know that you're sending too slow, and that you could 0:18:10.766 send a little faster, you're sending too slow there's 0:18:13.849 going to be some plan by which the sender can speed up. 0:18:17.051 And the nature of our solution is going to be that there is 0:18:20.49 really not going to be a steady rate at which senders are going 0:18:24.166 to send. In fact, by the definition of 0:18:26.359 the problem, the steady rate is a fool's errand because N 0:18:29.857 varies N varies and the load varies. 0:18:33 So that means on any given link, the traffic varies with 0:18:35.946 time. So you really don't want a 0:18:37.607 static rate. What you would like to do is if 0:18:39.91 there's any extra capacity, and the sender has load to fill 0:18:43.017 that capacity, you want that sender to use 0:18:45.214 that capacity. So this rate is going to adapt 0:18:47.571 and change with time in response to some feedback that we're 0:18:50.732 going to obtain based on network layer information essentially 0:18:54 from the network layer about whether people are sending too 0:18:57.107 fast or people are sending too slow. 0:19:00 OK and all congestion control schemes that people have come up 0:19:03.796 with, all algorithms for solving this problem, 0:19:06.597 and there have been dozens if not a few hundred of them, 0:19:10.02 all of these things in various variants of various solutions, 0:19:13.755 all of them are according to the basic plan. 0:19:16.431 OK, the network layer gives some feedback. 0:19:18.983 If it's too fast, slow down. 0:19:20.663 If it's too slow, speed up. 0:19:22.282 The analogy is a little bit like a water pipeline network. 0:19:25.829 Imagine you have all sorts of pipes feeding water in. 0:19:30 You have this massive pipeline network. 0:19:32.459 And what you can control are the valves at the end points. 0:19:36.148 And by controlling the valves, you can decide whether to let 0:19:39.966 water rush in or not. And anytime you're getting 0:19:43.008 clogged, you have to slow down and close the valves. 0:19:46.308 So the devil's actually in the details. 0:19:48.768 So were going to dive in to actually seeing how we're going 0:19:52.521 to solve this problem. And the first component of any 0:19:55.887 solution to congestion control is something we've already seen. 0:20:01 And it's buffering. Any network that has 0:20:04.021 asynchronous multiplex and, which Sam talked about the 0:20:08.126 first time, any network that has the following behavior, 0:20:12.387 at any given point in time you might have multiple packets 0:20:16.802 arrive in to a given switch. And, the switch can only send 0:20:21.218 one of them out on an outgoing link at any given point in time, 0:20:26.021 which means if you weren't careful, you would have to drop 0:20:30.436 the other packet that showed up. And so, almost everyone who 0:20:35.832 builds a decent asynchronously multiplex network puts in some 0:20:39.906 queues to buffer packets until the link is free so they can 0:20:43.843 then send the packet out onto the link. 0:20:46.422 So the key question when it comes to buffering that you have 0:20:50.36 to ask is how much? I don't actually mean how much 0:20:53.686 in terms of how expensive it is, but in terms of how much should 0:20:57.963 the buffering be? Should you have one packet of 0:21:02.354 buffering? Two packets? 0:21:04.141 Four packets? And how big should the buffer 0:21:07.552 be? Well, the one way to answer the 0:21:10.313 question of how much, first you can ask, 0:21:13.48 what happens if it's too little? 0:21:15.997 So what happens if the buffering is too little in a 0:21:20.058 switch? Like you put one packet or two 0:21:23.062 packets of buffering: what happens? 0:21:25.823 I'm sorry, what? Well, congestion by definition 0:21:29.559 has happened when the load exceeds the capacity. 0:21:35 So you get congestion. But what's the effect of that? 0:21:37.941 So a bunch of packets show up. You've got two packets of 0:21:41.052 buffering. So you could lose packets. 0:21:43.257 This is pretty clear, right? 0:21:44.785 Good. So if it's too little, 0:21:46.312 what ends up happening is that you drop packets, 0:21:48.97 which suggests that you can't have too little buffering, 0:21:52.081 OK? So, at the other end of the 0:21:53.778 spectrum, you could just say, well, I'm going to design my 0:21:57.002 network so it never drops a packet. 0:22:00 Memory is cheap. We learned about Moore's Law. 0:22:03.082 So let's just over provision, also called too much buffering. 0:22:07.191 OK, it's not really that expensive. 0:22:09.52 So what happens if there's too much buffering? 0:22:12.602 Well, if you think about it, the only thing that happens 0:22:16.369 when you have too much buffering as well, packets won't get lost. 0:22:20.753 But all you've done is traded off packet loss for packet 0:22:24.52 delay. Adding more buffering doesn't 0:22:26.917 make your link go any faster. They should probably learn that 0:22:31.693 at Disneyland and places like that. 0:22:33.551 I mean, these lines there are just so long. 0:22:35.846 I mean, they may as well tell people to go away and come back 0:22:39.125 later. It's the same principle. 0:22:40.765 I mean, just adding longer lines and longer queues doesn't 0:22:43.879 mean that the link goes any faster. 0:22:45.737 So it's really problematic to add excessive buffers. 0:22:48.524 And the reason is quite subtle. The reason it actually has to 0:22:51.803 do with something we talked about the last time, 0:22:54.371 or at lease one reason, a critical reason has to do 0:22:57.103 with timers and retransmissions. Recall that all you do when you 0:23:01.747 have too much buffering is you eliminate packet loss or at 0:23:04.961 least reduce it greatly at the expense of increasing delays. 0:23:08.287 But the problem with increasing delays is that your timers that 0:23:11.783 you're trying to set up to figure out when to retransmit a 0:23:14.997 packet, you would like them to adapt to increasing delays. 0:23:18.21 So he you build this exponentially weighted moving 0:23:20.973 average, and you pick a timeout interval that's based on the 0:23:24.299 mean value, and the standard deviation. 0:23:26.442 And the problem with too much buffering is that it makes these 0:23:29.881 adaptive timers extremely hard to implement because your 0:23:32.982 timeout value has to depend on both the mean and standard 0:23:36.139 deviation. And if you have too much 0:23:39.866 buffering, the range of round-trip time values is too 0:23:43.911 high. And the result of that you end 0:23:46.633 up with this potential for something called congestion 0:23:50.755 collapse. 0:23:52 0:24:07 And let me explain this with a picture. 0:24:09.873 So your adaptive timers are trying to estimate the 0:24:13.579 round-trip time, and the average round-trip time 0:24:17.134 in the standard deviation or at the linear deviation of the 0:24:21.521 round-trip time, and at some point they're going 0:24:25.075 to make a decision as to whether to retransmit a packet or not. 0:24:29.764 What might happen, and what does happen and has 0:24:33.243 happened when you have too much networks with too much buffering 0:24:38.008 is that you end up with a queue that's really big. 0:24:43 OK, and this is some link on which the packet's going out, 0:24:46.66 and you might have packet one sitting here, 0:24:49.357 and two sitting here, and three, and all the way out. 0:24:52.697 There is a large number of packets sitting there. 0:24:55.779 Notice that the end to end sender is trying to decide 0:24:59.119 whether packet one for whose acknowledgment it still hasn't 0:25:02.844 heard. It's trying to decide whether 0:25:05.592 one is still in transit, or has actually been dropped. 0:25:08.503 And it should retransmit the packet only after one has been 0:25:11.689 dropped. But if you have too much 0:25:13.446 buffering, the range of these values is so high that it makes 0:25:16.742 these adaptive timers quite difficult to tune. 0:25:19.214 And the result often is that one is still sitting here. 0:25:22.179 But it had been stuck behind a large number of packets. 0:25:25.145 So the delay was extremely long. 0:25:26.848 And the end to end sender timed out. 0:25:28.77 And when it times out, it retransmits one into the 0:25:31.517 queue. And, soon after that, 0:25:34.214 often it might retransmit two, and retransmit three, 0:25:37.165 and retransmit four, and so on. 0:25:38.9 And these packets are sort of just stuck in the queue. 0:25:41.966 They're not actually lost. And if you think about what 0:25:45.033 will happen then, this link, which is already a 0:25:47.694 congested link, because queues have been 0:25:49.95 building up here, long queues have been building 0:25:52.669 up, this link is not starting to use more and more of its 0:25:55.909 capacity to send the same packet twice, right, 0:25:58.512 because it sent one out. And the sender retransmitted it 0:26:03.08 thinking it was lost when it was just stuck behind a long queue. 0:26:07.307 And now one is being sent again. 0:26:09.387 And two is being sent again, and so on. 0:26:11.937 And graphically, if you look at this, 0:26:14.352 what you end up with is a picture that looks like the 0:26:17.841 following. This picture plots the total 0:26:20.391 offered load on the X axis, and the throughput of the 0:26:23.88 system on the Y axis where the throughput is defined as the 0:26:27.772 number of useful bits per second that you get. 0:26:32 So if you send the packet one twice, only one of those packets 0:26:35.812 is actually useful. Now, initially when the offered 0:26:38.937 load is low and the network is not congested,

0:26:41.687 and the offered load is less than the capacity of the link, 0:26:45.312 this curve is just a straight line
with slope one, 0:26:48.375 right, because everything you offer is below the link's 0:26:51.75 capacity. And it's
going through. 0:26:53.75 Now at some point, it hits the link's capacity, 0:26:56.625 right, the bottleneck link's
capacity. 0:27:00 And after that, any extra offered load that you 0:27:02.4 pump into the network is not going to go
out any faster. 0:27:05.271 The throughput is still going to remain the same. 0:27:07.776 And, it's just going to be
flat for a while. 0:27:10.02 And the reason it's flat is that queues are building up. 0:27:12.89 And, that's one reason
that you can't send things any faster. 0:27:16.021 The only thing that's going on is that queues are building up.
0:27:19.205 So this curve remains flat. Now, in a decent network that 0:27:22.127 doesn't have congestion
collapse, if you don't do 0:27:24.632 anything else, but somehow manage to keep the 0:27:26.929 system working
here, this curve might remain flat 0:27:29.434 forever. No matter what the offered 0:27:32.646 load, you know,
you can pump more and more load 0:27:35.56 in the system. And the throughput remains flat 0:27:38.41 at the
capacity. But the problem is that it has 0:27:41.324 interactions with things that the higher layer is are doing,
0:27:45.061 such as these retransmissions and timers. 0:27:47.595 And, eventually, more and more of the
capacity 0:27:50.445 starts being used uselessly for these redundant transmissions. 0:27:54.309 And you might
end up in a situation where the throughput 0:27:57.792 dies down. OK, and if that happens, 0:27:59.946 this
situation is called congestion collapse. 0:28:04 There is more and more work being presented to the system.
0:28:07.431 If you reach a situation where the actual amount of useful work 0:28:11.163 that's been done starts
reducing as more work gets presented to 0:28:14.895 the system, that's the situation of congestion collapse.
0:28:18.205 And this kind of thing shows up in many other systems, 0:28:21.396 for example, in things like Web
servers that 0:28:23.984 are built out of many stages where you have a high load 0:28:27.234 presented at the
system, you might see congestion 0:28:30.304 collapse in situations where, let's say you get a Web 0:28:33.434
request. And what you have to do is 0:28:36.323 seven stages of processing on the Web request. 0:28:38.615 But
what you do with the processor is you take a request, 0:28:41.466 and you process it for three of those stages.
0:28:43.706 And then you decide to go to the next request and process at 0:28:46.71 three stages. And, you go to
the next request 0:28:48.951 and processes at three stages. So, [\no request gets 0:28:51.649 complete. Your
CPU's 100% utilized. 0:28:53.38 But the throughput is essentially diving down to zero. 0:28:56.079 That's another
situation where you have congestion collapse. 0:29:00 But when it occurs in networks, it turns out it's more
0:29:03.868 complicated to solve the networks because of these 0:29:07.445 reasons that I outlined before.
0:29:10 0:29:16 So has everyone seen this? So let's go back to our 0:29:20.44 problem. Our goal is to,
0:29:22.524 we want this thing to be true. The aggregate offered load to 0:29:27.87 be smaller than the capacity
of a link for every link in the 0:29:33.216 system. But it turns out that's not 0:29:36.297 enough of a specification
because there are many ways to 0:29:41.281 achieve this goal. For example, 0:29:44.698 a really cheesy way to
achieve this goal is to make sure that 0:29:48.191 everybody remains quiet. If nobody sends any packets,
0:29:51.219 you're going to get that to be true. 0:29:53.256 So we're going to actually have to define the problem
a little 0:29:56.808 bit more completely. And let's first define what we 0:29:59.719 don't want. The first thing we
don't want 0:30:02.561 is congestion collapse. So any solution should have the 0:30:05.47 property that it never
gets into that situation. 0:30:08.001 And in fact, good congestion control schemes 0:30:10.317 operate at the left
knee of the curve. 0:30:12.31 But we're not going to get too hung up on that because if we 0:30:15.488 operate a
little bit in the middle, we're going to say 0:30:18.342 that's fine because that'll turn out to be good enough in
0:30:21.359 practice. And often is just fine. 0:30:23.082 And the additional complexity that you might have to bring
to 0:30:26.314 bear on a solution to work nearer the knee might not be 0:30:29.384 worth it. OK, but really we're
going to 0:30:33.092 worry about not falling off the cliff at the right edge, 0:30:37.061 OK? And then having done
that, 0:30:39.154 we're going to want reasonable utilization also called 0:30:42.979 efficiency. So, what this says is
that if 0:30:45.865 you have a network link that's often congested, 0:30:49.185 you want to make sure that that
link isn't underutilized when 0:30:53.515 there is offered load around. So for example, 0:30:56.69 people are
presenting 100 kb per second of load, 0:31:00.082 and the network link has that capacity; you want that to be
0:31:04.268 used. You don't want to shut them up 0:31:08.004 too much. So what this really means in
0:31:10.476 practice is that if you have slowed down, and excess capacity 0:31:14.486 has presented itself, then
you have to make sure that 0:31:17.96 you speed up. So that's what it means in 0:31:20.566 practice. And the
third part of the 0:31:22.838 solution is there's another thing we need to specify. 0:31:26.313 And the reason is
that you can solve these two problems by 0:31:30.122 making sure that only one person transmits in the network.
0:31:35 OK, if that person has enough offered load, 0:31:37.625 then you just got everybody out altogether, and
essentially 0:31:41.25 allocate that resource in sort of a monopolistic fashion to 0:31:44.875 this one person.
That's not going to be very 0:31:47.562 good because we would like our network to be used by a number
0:31:51.312 of people. So I'm going to define that as 0:31:53.812 a goal of any good solution. I'm going to call it
equitable 0:31:57.437 allocation. I'm not going to say fair 0:32:00.445 allocation because fair suggests that it's
really a strong 0:32:03.742 condition that every connection gets a roughly equal throughput 0:32:07.328 if it has
that offered load. I mean, that turns out to be, 0:32:10.625 I think, in my opinion, achieving perfect fairness to
0:32:13.632 TCP connections is just a waste of time because in reality, 0:32:16.986 fairness is governed by who's
paying what for access to the 0:32:20.341 network. So we're not going to get into 0:32:22.539 that in this class.
But we are going to want 0:32:25.026 solutions that don't eliminate, don't starve certain 0:32:27.975 connections
out. And we'll be happy with that 0:32:32.228 because that will turn out to work out just fine in practice.
0:32:37.002 Now, to understand this problem a little bit better, 0:32:41.061 we're going to want to understand this
requirement a 0:32:45.119 little bit more closely. And the problem is that this 0:32:49.257 requirement of the
aggregate rate specified by the offered 0:32:53.793 load, being smaller than the link capacity is a condition on
0:32:58.488 rates. It just says the offered load 0:33:01.806 is 100 kb per second. The capacity is 150 kb per
0:33:04.735 second. That means it's fine. 0:33:06.479 The problem is that you have to really ask offered load over
0:33:10.155 what timescale? For example, 0:33:11.838 if the overall offered load on your network is, 0:33:14.704

let's say, a megabit per second, and the capacity of the 0:33:18.13 network is half a megabit per second, and that condition 0:33:21.744 lasts for a whole day, OK, so for a whole day, 0:33:24.548 your website got slash dotted. Take that example. 0:33:27.538 And you have this little wimpy access link through your DSL 0:33:31.152 line. And, ten times that much in 0:33:33.146 terms of requests are being presented to your website. 0:33:36.448 And, it lasts for the whole day. 0:33:40 Guess what. Nothing we're going to talk 0:33:41.809 about is really going to solve that problem in a way that 0:33:44.476 allows every request to get through in a timely fashion. 0:33:47.095 At some point, you throw up your hands and 0:33:49.047 say, you know what? If I want my website to be 0:33:51.19 popular, I'd better put it on a 10 Mb per second network and 0:33:54 make sure that everybody can gain access to it. 0:33:56.19 So we're not really going to solve the problem at that time 0:33:58.952 scale. On the other hand, 0:34:01.09 if your website suddenly got a little bit popular, 0:34:04.06 and a bunch of connections came to it, but it didn't last for a 0:34:07.818 whole day, but it lasted for a few seconds, we're going to want 0:34:11.575 to deal with that problem. So if you think about it, 0:34:14.666 there are three timescales that matter here. 0:34:17.272 And these timescales arise in an actual way because of this 0:34:20.787 network where congestion happens here. 0:34:23.03 And then we're going to have some feedback, 0:34:25.575 go back to the sender, and the sender exercises 0:34:28.363 control. And the only timescale in this 0:34:31.91 whole system is the round-trip time because that's the 0:34:35.286 timescale, the order of magnitude of the timescale 0:34:38.407 around which any feedback is going to come to us. 0:34:41.464 So there are three timescales of interest. 0:34:44.076 There is smaller than one round-trip time, 0:34:46.687 which says this inequality is not satisfied for really small. 0:34:50.509 The time where there is a little spurt, 0:34:52.929 a burst, it's also called a burst, a burst of packets show 0:34:56.56 up, and then you have to handle them. 0:35:00 But then after that, things get to be fine. 0:35:03.255 So there is smaller than one round-trip time. 0:35:06.666 So this is summation LI is greater than C, 0:35:09.844 i.e. the network is congested. 0:35:12.093 It could be congested at really short durations that are smaller 0:35:16.976 than a roundtrip time. It could be between one and I'm 0:35:21.085 going to say 100 round-trip times. 0:35:23.643 And just for real numbers, a round-trip time is typically 0:35:27.984 order of 100 ms. So we are talking here of less 0:35:31.55 than 100 ms up to about ten seconds, and then bigger than 0:35:35.891 this number, OK? Bigger than 100. 0:35:40 And these are all orders of magnitude. 0:35:42.458 I mean, it could be 500 RTT's. OK, those are the three time 0:35:46.312 scales to worry about. When congestion happens at less 0:35:49.833 than one roundtrip time, we're going to solve that 0:35:53.089 problem using this technique. We're going to solve it using 0:35:56.943 buffering, OK? And, that's the plan. 0:36:00 The reason is that it's really hard to do anything else because 0:36:03.105 the congestion lasts for a certain burst. 0:36:05.008 And by the time you figure that out and tell the sender of that, 0:36:08.163 the congestion has gone away, which means telling the sender 0:36:11.117 that the congestion has gone away, which means telling the 0:36:13.972 sender that the congestion has gone away was sort of a waste 0:36:16.927 because the sender would have slowed down. 0:36:18.98 But the congestion anyway went away. 0:36:20.733 So, why bother, right? 0:36:21.785 Why did you tell the sender that? 0:36:23.387 So that's the kind of thing you want to solve at the network 0:36:26.342 layer inside the switches. And that's going to be done 0:36:28.996 using buffering. And that sort of suggests, 0:36:32.387 and there's a bit of sleight-of-hand here. 0:36:34.576 And we're not going to talk about why there's a 0:36:37.031 sleight-of-hand. But this really suggests that 0:36:39.433 if you design a network and you want to put buffering in the 0:36:42.582 network, you'd better not put more than about a round-trip 0:36:45.624 time's worth of buffering in that switch. 0:36:47.759 If you put buffering longer than a round-trip time, 0:36:50.428 you're getting in the way of things that the higher layers 0:36:53.47 are going to be doing. And it's going to confuse them. 0:36:56.299 In fact, Sam showed you this picture of these round-trip 0:36:59.235 times varying a lot between the last lecture where the mean was 0:37:02.544 two and a half seconds. And the standard deviation was 0:37:06.879 one and a half seconds. That was not a made-up picture. 0:37:10.261 That was from a real wireless network where the designers had 0:37:14.019 incorrectly put a huge amount of buffering. 0:37:16.65 And this is extremely common. Almost every modem that you 0:37:20.157 buy, cellular modem or phone modem, has way too much 0:37:23.352 buffering in it. And, the only thing that 0:37:25.857 happens is these queues build up. 0:37:27.861 It's just a mistake. So less than one roundtrip 0:37:30.742 time: deal with it using buffering. 0:37:34 Between one and 100 round-trip times, we're going to deal with 0:37:37.884 that problem using the techniques that we are going to 0:37:41.259 talk about today, the next five or ten minutes. 0:37:44.188 And then bigger than 100 round-trip times or 1,000 0:37:47.308 round-trip times are things where congestion is just sort of 0:37:51.066 persistent for many, many, many seconds or minutes 0:37:54.186 or hours. And there are ways of dealing 0:37:56.606 with this problem using protocols and using algorithms. 0:38:01 But ultimately you have to ask yourself whether you are really 0:38:04.682 under provisioned, and you really ought to be 0:38:07.338 buying or provisioning your network to be higher, 0:38:10.236 maybe put your server on a different network that has 0:38:13.979 higher capacity. And these are longer times 0:38:16.514 congestion effects for which decisions based on provisioning 0:38:20.076 have to probably come to play in solving the problem. 0:38:24 0:38:29 So, provisioning is certainly an important problem. 0:38:32.791 And when you have congestion lasting really long periods of 0:38:37.189 time, that might be the right solution. 0:38:40.071 But for everything else, there is solutions that are 0:38:43.938 much easier to understand, or at least as far as using the 0:38:48.26 tools that we've built so far in 6.033. 0:38:51.142 So, the first component of the solution is some buffering to 0:38:55.616 deal with the smaller than one round-trip time situation. 0:39:01 And then when your buffers start getting filled up, 0:39:04.129 and congestion is lasting for more than a round-trip time, 0:39:07.697 then your buffers start getting filled up. 0:39:10.263 At that point, you're starting to see 0:39:12.516 congestion that can't be hidden by pure buffering. 0:39:15.583 And that can't be hidden because queues are building up, 0:39:19.026 and that's causing increased delay to show up at the sender, 0:39:22.719 and perhaps packets may start getting dropped when the queue 0:39:26.411 overflows. And that causes the sender to 0:39:28.852 observe this congestion, which means that what we want 0:39:32.17 is a plan by which, when congestion happens lasting 0:39:35.299 over half a round-trip time or close to a round-trip time, 0:39:38.867 we're going to want to provide feedback to the sender, 0:39:42.184 OK? And then the third part of our 0:39:46.101 solution is when the sender gets this feedback, 0:39:49.322 what it's going to do is adapt. And the way it's going to adapt 0:39:53.665 is actually easy. It's going to do it by 0:39:56.396 modifying the rate at which it sends packets. 0:39:59.478 And that's what are set up. The sender sends at rate RI. 0:40:03.806 What you do is you change the rate or the speed at which the 0:40:07.612 sender sends. So, there's two things we have 0:40:10.387 to figure out. One is, how does the network 0:40:13.096 give feedback? And the second is, 0:40:15.161 what exactly is going on with changing the speed? 0:40:18.258 How does it work? There are many ways to give 0:40:21.096 feedback, and sort of the first order of things you might think 0:40:25.096 about are, well, when the queue starts to fill 0:40:28 up, I'll send a message to the sender, or I'll send a bit in 0:40:31.806 the packet header and get to the endpoint, and it'll send this 0:40:35.741 feedback back. And any variant that you can 0:40:39.608 think of in the next five or ten minutes I can assure you has 0:40:43.055 been thought of and investigated. 0:40:44.893 And you might think of new ways of doing it which would be 0:40:48.168 great. I mean, this is a topic of 0:40:50.006 active work. People are working on this 0:40:52.189 stuff still. But my opinion is that the best 0:40:54.659 way to solve this problem of feedback is the simplest 0:40:57.646 possible thing you could think of. 0:40:59.542 The simplest technique that really works all the time is 0:41:02.702 just drop the packet. In particular, 0:41:05.606 if the queue overflows, and it's going to get dropped 0:41:07.857 anyway, and that's a sign of congestion, but if at any point 0:41:10.411 in time you decide that the network is getting congested and 0:41:12.965 remains so for a long enough time scale that you want the 0:41:15.389 sender to react, just throw the packet away. 0:41:17.251 The nice thing about throwing a packet away is that it's 0:41:19.632 extremely hard to implement it wrong because when the queue 0:41:22.142 overflows, the packet's going to be dropped anyway. 0:41:25 So that feedback is going to be made available to the sender. 0:41:28.89 So what we're going to assume for now is that all packet drops 0:41:32.846 that happen in networks are a sign of congestion, 0:41:35.959 OK? And when a packet gets dropped, 0:41:38.163 the sender gets that feedback. And, the reason it gets that 0:41:41.924 feedback is, remember that every packet's being acknowledged. 0:41:45.815 So if it misses an acknowledgment, 0:41:47.955 then it knows that the packet's been dropped. 0:41:50.808 And it says, ah, the network is congested. 0:41:53.467 And then I'm going to change my speed. 0:41:55.866 And, I'm going to change my speed by reducing the rate at 0:41:59.498 which I'm going to send packets. So now, we have to ask how the 0:42:05.121 sender adapts to congestion, how the speed thing actually 0:42:09.08 works out. 0:42:10 0:42:16 Well? 0:42:17 0:42:22 It says up. It's already up. 0:42:25.461 Yeah, it doesn't have a down button. 0:42:29.948 What's that? 0:42:32 0:42:39 Sam? 0:42:41 0:42:55 How many professors does it take to -- 0:42:58 0:43:09 He turned the thing off. He turned the light off on 0:43:13.189 that. This might actually work. 0:43:15.703 Can you see that? I bet it's a race condition. 0:43:19.474 All right, so, can you see that? 0:43:22.072 OK, let's just move from here. OK, the way we're going to 0:43:26.765 figure out the rate at which we're going to send our packets 0:43:31.709 is to use this idea of a sliding window that we talked about the 0:43:36.988 last time. And since Sam did such a nice 0:43:41.537 animation, I'm going to display that again, just a refresher. 0:43:46.787 Wonderful. All right. 0:43:48.537 [APPLAUSE] And since Sam made this nice animation, 0:43:52.824 I'm going to play that back. I've always wanted to say, 0:43:57.55 play it again, Sam. 0:44:00 So, what's going on here is that whenever the receiver 0:44:03.382 receives a packet, it sends an acknowledgment 0:44:06.191 back. And the sender's pipelining 0:44:08.234 packets going through. And whenever the sender gets an 0:44:11.617 acknowledgment, it sends a new packet off. 0:44:14.234 And that's where you see happening. 0:44:16.404 Whenever it gets an acknowledgment, 0:44:18.574 a new packet goes out onto the other end. 0:44:21.127 And, the window's sliding by one every time it gets an 0:44:24.51 acknowledgement. So this is sort of the way in 0:44:27.382 which this steady-state thing works out for us in this 0:44:30.765 network. So the main point to note about 0:44:33.255 this is that this scheme has an effect that I'll call self 0:44:36.893 pacing. OK, it's self pacing because 0:44:40.359 the sender doesn't actually have to worry very much about when it 0:44:43.467 should exactly send a packet because it's being told every 0:44:46.236 time an acknowledgment arrives, it's being told that the reason 0:44:49.247 I got an acknowledgment is because one packet left the 0:44:51.821 network, which means I can send one packet into the network. 0:44:54.686 And as long as things weren't congested before, 0:44:56.92 they won't be congested now, right, because packets have 0:44:59.591 left the pipe. For every packet that leaves, 0:45:02.693 you're putting one packet in. This is an absolutely beautiful 0:45:06.197 idea called self-pacing. And, the idea here is that 0:45:09.116 acts essentially strobe data packets. 0:45:11.569 And, like all a really nice ideas, it's extremely simple to 0:45:14.956 get it. And it turns out to have lots 0:45:17.058 of implications for why the congestion control system is 0:45:20.27 actually stable in reality. So what this means, 0:45:22.956 for example, is let's say that things are 0:45:25.291 fine. And, all of a sudden a bunch of 0:45:27.394 packets come in from somewhere else, and the network starts to 0:45:30.956 get congested. What's going to happen is that 0:45:34.436 queues are going to start building up. 0:45:36.208 But, when queues start building up, what happens is that 0:45:38.842 transmissions that are ongoing for this connection are just 0:45:41.62 going to slow down a little bit because they're going to get 0:45:44.446 interleaved with other packets in the queue, 0:45:46.505 which means the acts are going to come back slower because how 0:45:49.427 can the acts come back any faster than the network can 0:45:51.965 deliver them, which means automatically the 0:45:53.977 sender has a slowing down effect dealing with transient 0:45:56.563 congestion. This is a really, 0:45:59.231 really nice behavior. And as a consequence of the way 0:46:03 in which the self pacing of, the way in which the sliding 0:46:07.057 window thing here works. So now, at some point the 0:46:10.608 network might become congested, and a packet may get dropped. 0:46:14.956 So, the way that manifests itself as one packet gets 0:46:18.652 dropped, the corresponding acknowledgment doesn't get back 0:46:22.782 to the sender. And when an acknowledgment 0:46:25.681 doesn't get back to the sender, what the sender does is reduce 0:46:30.101 its window size by two. 0:46:33 0:46:39 OK, so I should mention here there are really two windows 0:46:42.458 going on. One window is the window that 0:46:44.805 we talked about the last time where the receiver tells the sender 0:46:48.511 how much buffer space that it has. 0:46:50.55 And the other window, another variable that we just 0:46:53.638 introduced which is maintained by the sender, 0:46:56.355 and it's also called the congestion window, 0:46:58.949 OK? And this is a dynamic variable 0:47:01.53 that constantly changing as the sender gets feedback from the 0:47:04.59 receiver. And in response to a missing 0:47:06.477 act when you determine that a packet is lost, 0:47:08.721 you say its congestion is going on. 0:47:10.455 And I want to reduce my window size. 0:47:12.241 And there are many ways to reduce it. 0:47:14.077 We're going to just multiplicatively decrease it. 0:47:16.525 And the reason for that has to do with the reason why, 0:47:19.228 for example, on the Ethernet you found that 0:47:21.37 you were doing exponential back off. 0:47:23.156 You're sort of geometrically increasing the spacing with 0:47:25.961 which you are sending packets there. 0:47:27.746 You're doing the same kind of thing here reducing the rate by 0:47:30.806 a factor of two. So that's how you slowdown. 0:47:34.549 At some point you might get in situation where no 0:47:37.418 acknowledgments come back for a while. 0:47:39.54 Say, many packets gets lost, right? 0:47:41.491 And that could happen. All of a sudden there's a lot 0:47:44.418 of congestion going on. You get nothing back. 0:47:46.942 At that point, you've lost this ability for 0:47:49.352 acknowledgments to strobe new data packets. 0:47:51.762 OK, so the acts have acted as a clock allowing you to strobe 0:47:55.147 packets through. And you've lost that. 0:47:57.27 At this point, your best plan is just to stop. 0:48:01 And you halt for a while. It's also called a timeout 0:48:03.563 where you just be quiet for awhile, often on the order of a 0:48:06.478 second or more. And then you try to start over 0:48:08.74 again. Of course, now we have to 0:48:10.298 figure out how you start over again. 0:48:12.057 And that's the same problem you have when you start a connection 0:48:15.223 in the beginning. When a connection starts up, 0:48:17.485 how does it know how fast to send? 0:48:19.143 It's the same problem as when many, many packet drops happen, 0:48:22.159 and you timeout. And, in TCP, 0:48:23.566 that startup is done using something called, 0:48:25.727 a technique called slow start. And I'll describe the 0:48:28.291 algorithm. It's a really elegant 0:48:31.038 algorithm, and it's very, very simple to implement. 0:48:33.511 So you have a sender and receiver. 0:48:35.143 And the plan is initially, the sender wants to initiate a 0:48:37.913 connection to the receiver. It sends a request. 0:48:40.188 It gets the response back with the flow control window. 0:48:42.859 It just says how much buffer space the receiver currently 0:48:45.629 has. OK, we're not going to use that 0:48:47.36 anymore in this picture. But it's just there saying, 0:48:49.882 which means the sender can never send more than eight 0:48:52.454 packets within any given roundtrip. 0:48:54.136 Then what happens is the sender sends the first segment, 0:48:56.856 and it gets an act. So, initially this congestion 0:48:59.23 window value starts off at one. OK, so you send one packet. 0:49:03.637 One segment, you get an acknowledgment back. 0:49:06.246 And now the plan is that the algorithm is as follows. 0:49:09.4 For every acknowledgment you get back, you increase the 0:49:12.676 congestion window by one. OK, so what this means is when 0:49:16.012 you get this acknowledgment back, you send out two packets. 0:49:19.531 And each of those two packets and sends you back two 0:49:22.625 acknowledgements. Now, for each of those two 0:49:25.233 acknowledgements, you increase the window by one, 0:49:28.145 which means that after you got both those acknowledgements, 0:49:31.663 you've increased your window by two. 0:49:35 In previously, the window was two. 0:49:36.605 So, now you can send four packets. 0:49:38.21 So what happens is in response to the rate acknowledgment, 0:49:40.982 you send out two packets. In response to the light blue 0:49:43.608 acknowledgement, you send out two more packets. 0:49:45.846 And now, those get through to the other side, 0:49:47.986 they send back acknowledgments to you. 0:49:49.785 In response to each one of those, you increase the window 0:49:52.509 by one. So increasing the window by one 0:49:54.357 on each acknowledgment means you can send two packets. 0:49:56.935 The reason is that one packet went through. 0:50:00 And you got an acknowledgment. So in response to the 0:50:02.819 acknowledgment, you could send one packet to 0:50:05.195 fill in the space occupied by the packet that just got 0:50:08.125 acknowledged. In addition, 0:50:09.507 you increase the window by one, which means you can send one 0:50:12.768 more packet. So really, what's going on with 0:50:15.145 this algorithm is if on each act, you send two segments, 0:50:18.185 which means you increase the window by one, 0:50:20.507 the congestion window by one, really what happens is the rate 0:50:23.824 at which this window opens is exponential in time because in 0:50:27.085 every roundtrip, from one roundtrip to the next, 0:50:29.683 you're doubling the entire window, OK? 0:50:33 So although it's called slow start, it's really quite fast. 0:50:36.678 And, it's an exponential increase. 0:50:38.771 So if you put it all together, it will turn out that the way 0:50:42.513 in which algorithms like the TCP algorithm works is it starts off 0:50:46.572 at a value of the congestion window. 0:50:48.792 If this is the congestion window, and this is time, 0:50:51.964 it starts off at some small value like one. 0:50:54.627 And, it increases exponentially. 0:50:56.594 And then, if you continually increase exponentially, 0:50:59.828 this is a recipe for congestion. 0:51:03 It's surely going to overflow some buffer and slowdown. 0:51:05.877 So what really happens in practice is after a while, 0:51:08.595 based on some threshold, it just automatically decides 0:51:11.419 it's probably not worth going exponentially fast. 0:51:13.976 And, it turns out slows down to go at a linear increase. 0:51:16.907 So every round-trip time it's increasing the window by a 0:51:19.838 constant amount. And then at some point, 0:51:21.916 congestion might happen. And when congestion happens, 0:51:24.687 we drop the window by a factor of two. 0:51:26.658 So, we cut down to a factor of two of the current window. 0:51:29.642 And then, we continue to increase linearly. 0:51:33 And then you might have congestion coming again. 0:51:35.362 For instance, we might have lost all of the 0:51:37.473 acknowledgements, not got any of the 0:51:39.232 acknowledgements, losing a lot of packets, 0:51:41.292 which means we go down to this thing where we've lost the act 0:51:44.308 clock. So, we have to time out. 0:51:45.816 And, it's like a new connection. 0:51:47.374 So, we remain silent for a while. 0:51:48.982 And this period is the time out period that's governed by the 0:51:51.998 equations that we talked about the last time based on the 0:51:54.812 roundtrip time and the standard deviation. 0:51:56.873 We remain quiet for a while, and then we start over as if 0:51:59.688 it's a new connection. And we increase exponentially. 0:52:03.787 And this time, we don't exponentially increase 0:52:06.468 as we did the very first time. We actually exponentially 0:52:09.744 increase only for a little bit because we knew that congestion 0:52:13.378 happened here. So, we go up to some fraction

0:52:15.94 of that, and then increase linearly. 0:52:18.025 So the exact details are not important. 0:52:20.289 The thing that's important is just understanding that when 0:52:23.685 congestion occurs, you drop your window by a 0:52:26.544 factor of some multiplicative degrees. 0:52:30 And at the beginning of connection, you're trying to 0:52:32.318 quickly figure out what the sustainable capacity is. 0:52:34.636 So he you are going in this increase phase here. 0:52:36.772 But after awhile, you don't really want to be 0:52:38.772 exponentially increasing all the time because that's almost 0:52:41.409 guaranteed to overrun buffers. So you want to go into a more 0:52:44.09 ginger mode. And, people have proposed 0:52:45.909 congestion control schemes that do all sorts of things, 0:52:48.363 all sorts of functions. But this is the thing that's 0:52:50.681 just out there on all of the computers. 0:52:52.409 And it works remarkably well. So, there's two basic points 0:52:55 you have to take home from here. The main goal of congestion 0:52:57.681 control is to avoid congestion collapse, which is the picture 0:53:00.409 that's being hidden behind that, which is that as often load 0:53:03.09 increases, you don't want throughput to drop like a cliff. 0:53:07 And this is a cross layer problem. 0:53:08.503 And the essence of all solutions is to use a 0:53:10.461 combination of network layer feedback and end system control, 0:53:13.194 end to end control. And, good solutions work across 0:53:15.471 eight orders of magnitude of bandwidth, and four orders of 0:53:18.067 magnitude of delay. So, with that, 0:53:19.57 we'll start. DP1 is due tomorrow. 0:53:21.028 Have a great spring break, and see you after spring break.