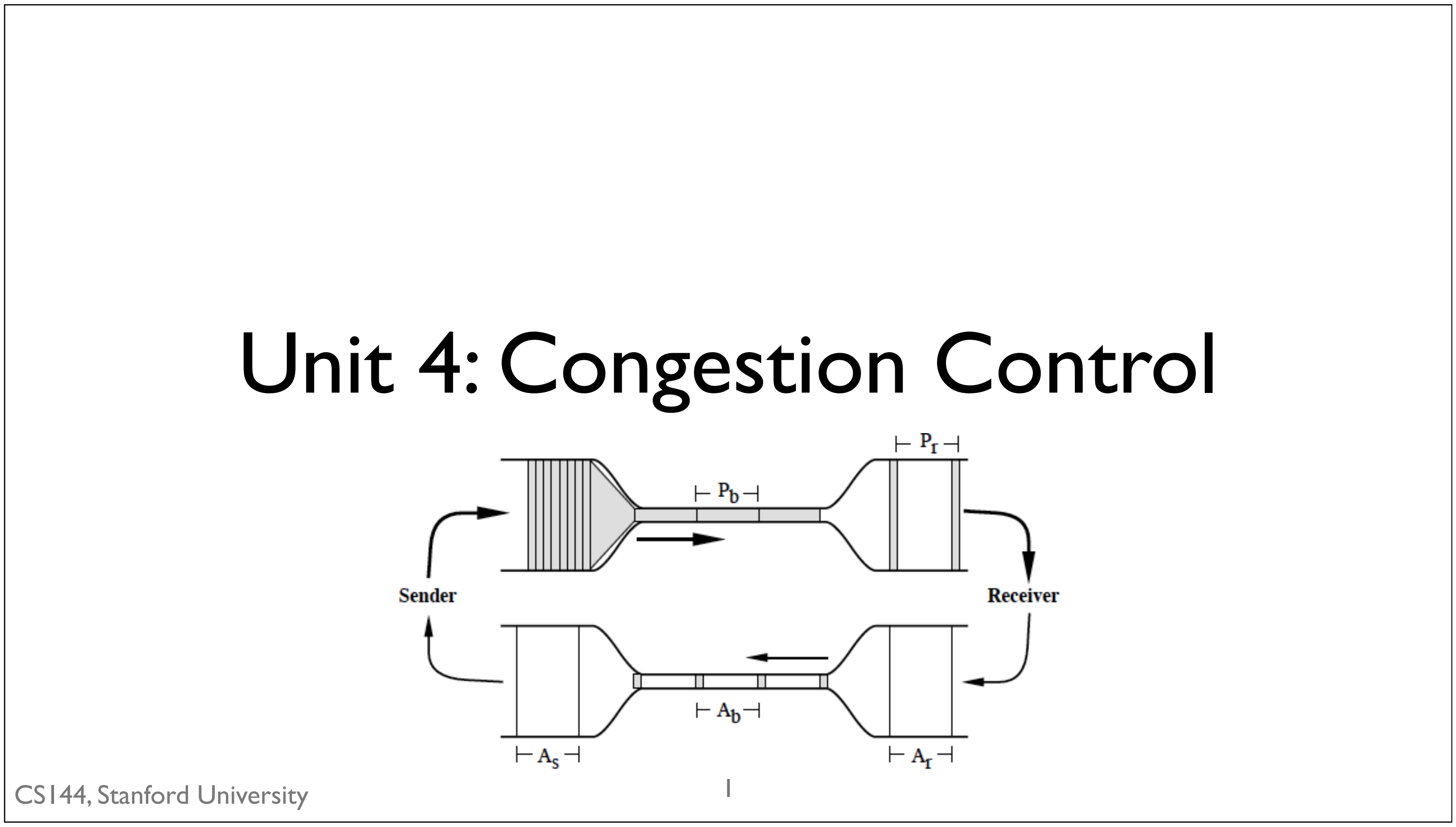
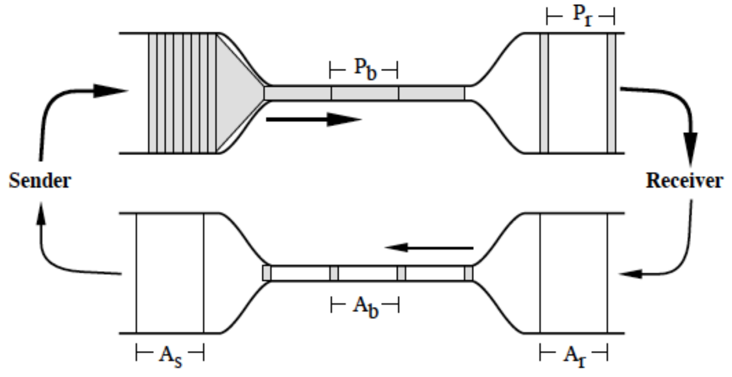


Unit 4: Congestion Control



Nick: In this unit, you've seen how transport and packet switching interact through congestion control. Flow control ensures that a sender doesn't send more than a receiver can receive, but there are typically many routers between the two, each with their own time-varying queues. Congestion control is how endpoints control their sending rate so that they don't overly congest the network by filling up queues. This reduces packet drop rates, which makes it easier for protocols like TCP to be fast. It also keeps the network working well, using links capacity but not dropping too many packets.

What You Learned: Principles

- What network congestion is, what causes it, and what happens in routers
- How we want the network to behave when congested: max-min fairness
- AIMD: additive increase, multiplicative decrease
- Behavior of multiple AIMD flows and TCP throughput equation

$$R = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

Phil: First, Nick explained the principles of network congestion. He showed you what happens when a router starts receiving packets faster than it can send them. If the congestion is short lived, then a router can absorb this extra traffic into a queue and drain the queue. If the congestion is long lived, long lived enough that the queue overflows, then the router has to drop some packets.

Nick introduced a very valuable way to think about this. Rather than come up with a scheme for dropping packets, think about what you want the overall network behavior to be. We want the network to be fair and explained what that means, introducing the concept of max-min fairness. Max-min fairness says that the network is fair if you can't increase the rate of a flow without decreasing the rate of flow with a lower rate.

There are a lot of ways to achieve this goal, and networks today have many different mechanisms. But we focused on one in particular, how TCP can control its sending rate. We introduced the basic algorithm TCP uses, called additive increase, multiplicative decrease, or AIMD. When running smoothly, TCP increases the number of bytes it can have outstanding by one segment size per round trip time. When TCP detects a packet is dropped, it halves the number of bytes it can have outstanding.

We looked at what this behavior looks like using a TCP sawtooth diagram. While each individual flow has a sawtooth, over link that many flows share these all average out to a consistently high use of the link. Using the sawtooth, we derived TCP's throughput using simple AIMD. If you assume that a network drops packets at a uniform rate p, then the throughput of a TCP flow is the square root of three halves times the inverse of RTT times the square root of p. If you increase the round trip time, throughput goes down. This equation makes a lot of simplifying assumptions, but it turns out to be generally pretty accurate and so a very valuable tool when thinking about how a network might behave.

What You Learned: Practice

- TCP Tahoe, Reno, and New Reno
- Congestion window
- Slow start, congestion avoidance
- RTT estimation
- Self-clocking
- Fast retransmit, fast recovery, congestion window inflation

Nick: You then learned how TCP realizes these principles in practice. Phil told you about how the Internet collapsed in the late 1980s due to congestion, and the fixes made to TCP, which are still in use today. You learned about three versions of TCP. TCP Tahoe, TCP Reno, and TCP New Reno.

The first important idea we covered is that a TCP endpoint maintains a congestion window. A TCP flow can have N unacknowledged bytes outstanding in the network, where N is the minimum of its flow control window and its congestion control window. You don't send packets faster than the other end can receive or faster than the network can handle. You learned how TCP controls the size of this congestion control window using two states: slow start and congestion avoidance. Slow start lets TCP quickly find something close to the right congestion window size, while congestion avoidance uses AIMD. TCP starts in slow start and transitions to congestion avoidance when it first detects a

You learned how TCP estimates the round trip time of its connection. It needs this estimate to figure out when an acknowledgment times out. By keeping track of both the average as well as the variance of how long it takes to receive an ack for a segment, TCP can avoid unnecessary retransmissions as well as not wait too long.

You learned how TCP controls the rate at which it puts packets in the network using a technique called “self clocking” You first saw self-clocking when I showed you an animation of TCP's behavior. Phil then walked you through some examples of this. With self clocking, TCP only puts a new packet into the network when it receives an acknowledgment or when there's a timeout. This is really helpful in preventing congestion, as it means TCP only puts packets in the network when packets have left the network.

Finally, we covered three optimizations added in TCP Reno and TCP New Reno. Fast retransmit lets TCP keep on making progress when only one packet has been dropped. Rather than wait for a timeout, TCP retransmits a segment when it detects three duplicate acknowledgments for the previous segment. This is a sign that TCP is continuing to receive segments but hasn't received that particular one. Using fast recovery, TCP Reno doesn't drop back into slow start on three duplicate acks, it just cuts the congestion window in half and stays in congestion avoidance. Finally, TCP New Reno adds an additional optimization, window inflation, such that three duplicate acks don't cause TCP to lose an RTT worth of transmissions as it waits for the missing segment to be acked.

What You Learned: Practice

- TCP Tahoe, Reno, and New Reno
- Congestion window
- Slow start, congestion avoidance
- RTT estimation
- Self-clocking
- Fast retransmit, fast recovery, congestion window inflation

Phil: What's really fascinating about congestion was that it's something we discovered as the Internet evolved. Nobody had really thought something like this might happen, or how to control it. It was an emergent behavior once the network became large and heavily used enough. Nowadays it's a basic concept in networking, seen as critical to building robust networks that have high performance.

Nick: Modern versions of TCP are a bit more advanced than what we've talked about, but mostly they've evolved to handle much, much faster networks. The TCP versions shipped in operating systems have TCP Reno or TCP NewReno in their algorithms, adding additional features and modes of operation to handle very fast networks. Take a look at the Linux source code, you'll see these algorithms in there.

Phil: But what's also neat is that these nitty-gritty algorithms have a sound conceptual basis and theory behind them. On one hand, we can talk about RTT variance estimation, fast recovery, and self-clocking. On the other, we're also talking about AIMD flows that can converge to max-min fairness.