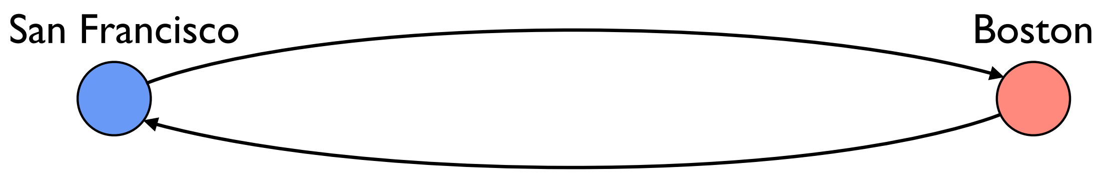
TCP Congestion Control I

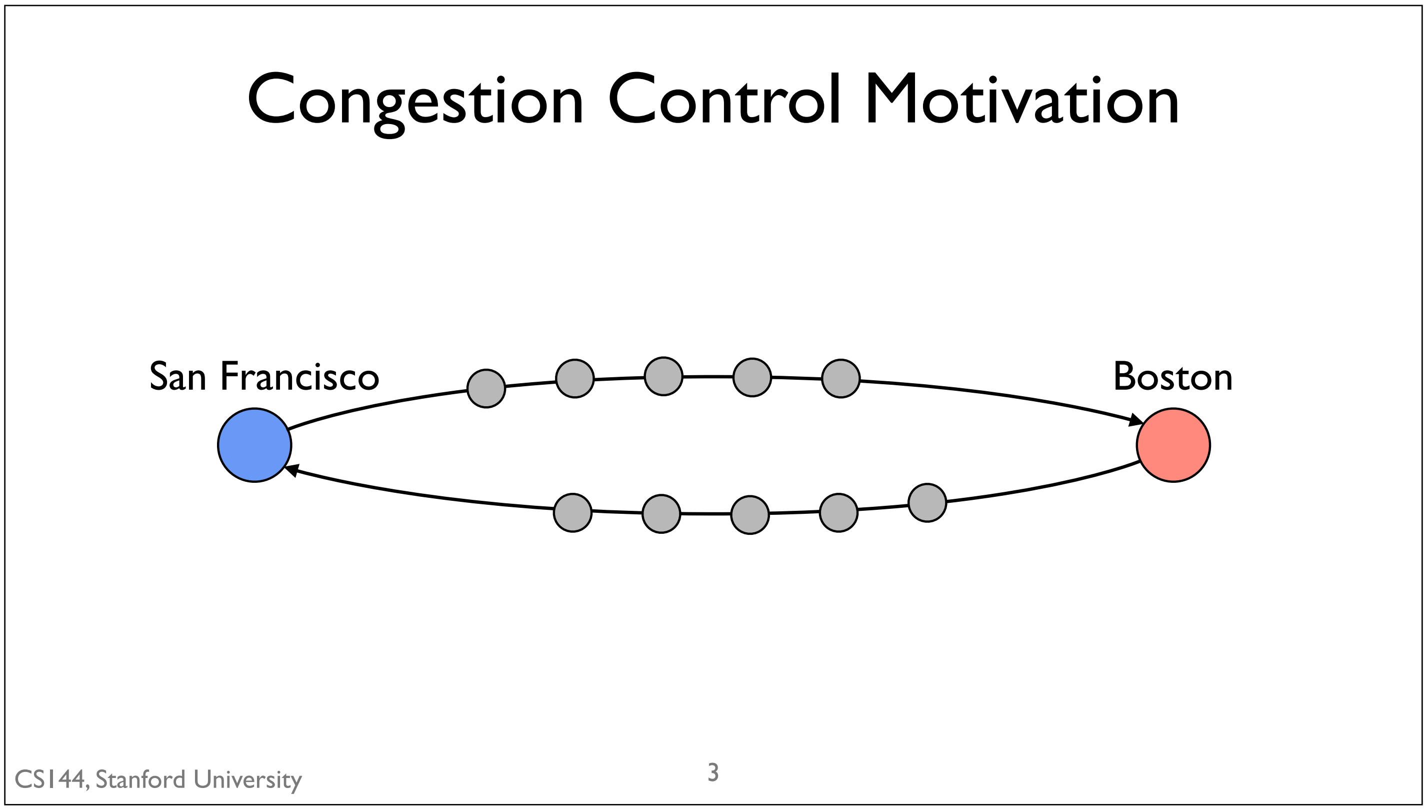
Slow start, congestion avoidance, triple duplicate acks

In the next three videos, I'm going to explain how TCP's congestion control works. The basic summary is that TCP uses additive increase, multiplicative decrease. But there are a lot of details on how this is achieved. TCP is considered a tremendous achievement in networking: it's a reliable, high performance transport layer that can operate well in a huge range of network environments. Of course it's not perfect, but you can see its strength in how many applications depend on it. In this video, I'm going to explain how TCP uses a simple finite state machine to control the number of packets it has outstanding in the network. This finite state machine implements AIMD as well as handling starting a connection and significant network disruptions.

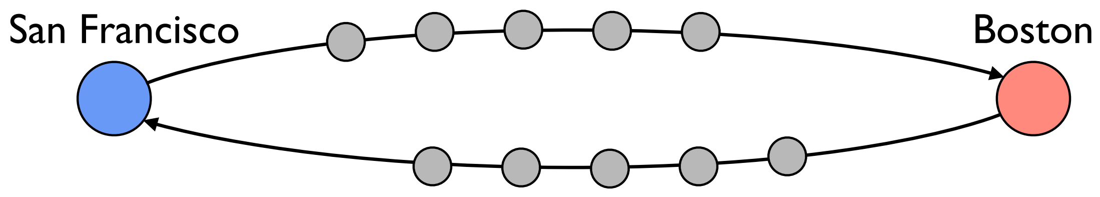
Congestion Control Motivation



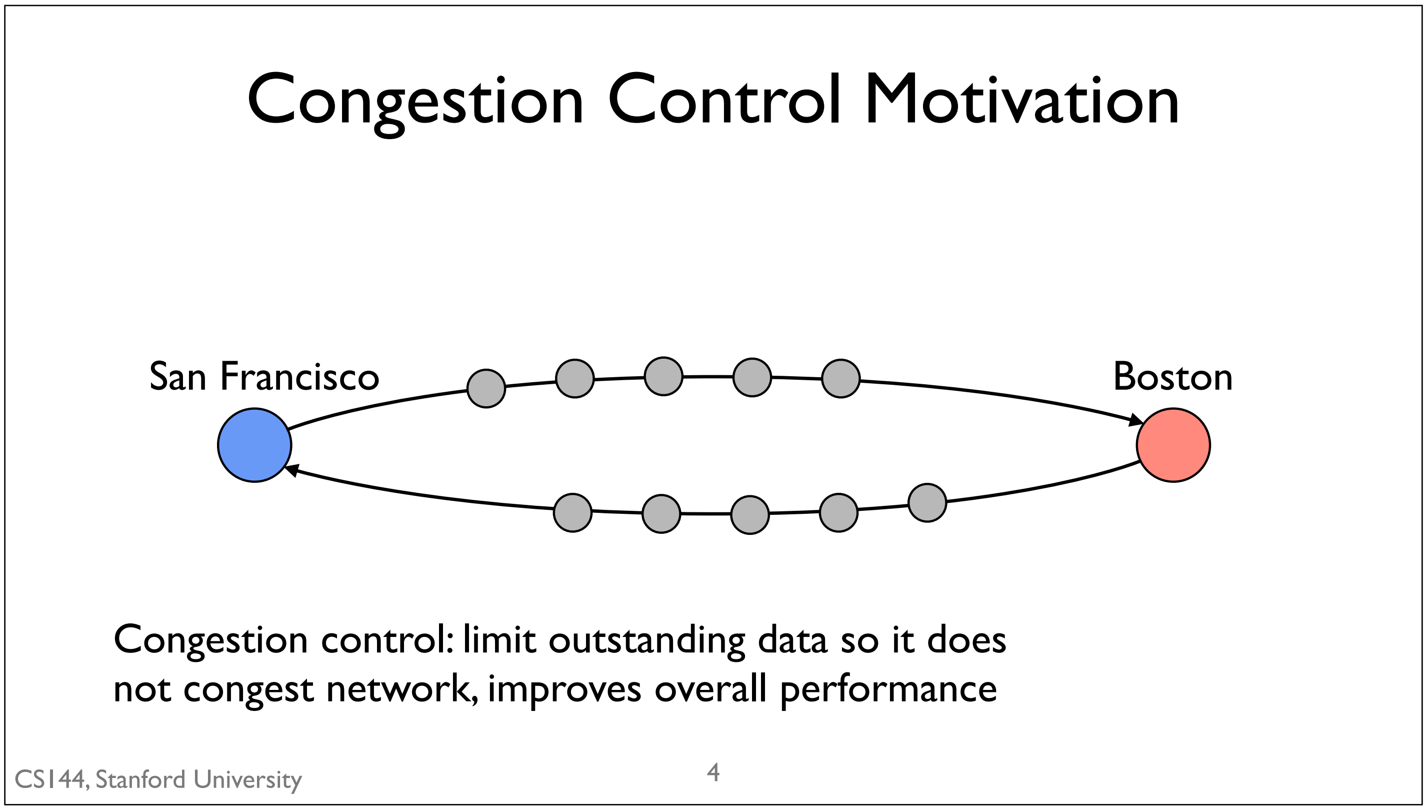
So, recall, congestion control is how a network protocol or system tries to not overload the network between two end points. Say that we’re transmitting data between San Francisco and Boston. There are many routers between these two end points, each with its own queues and traffic load.



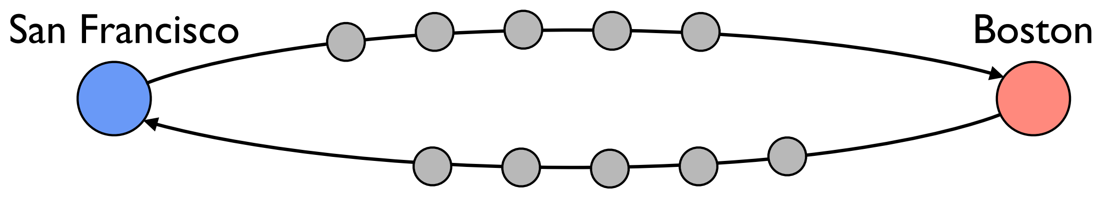
Congestion Control Motivation



If we send packets below the rate the route can support, we can expect to see reasonably low packet drop rates. Most of the packets from San Francisco to Boston will arrive and most of the packets from Boston to San Francisco will arrive. But if we send more packets than the route can support, then they'll be dropped from queues. The sender needs to somehow detect that this packet was lost and retransmit it. This takes time and reduces performance.

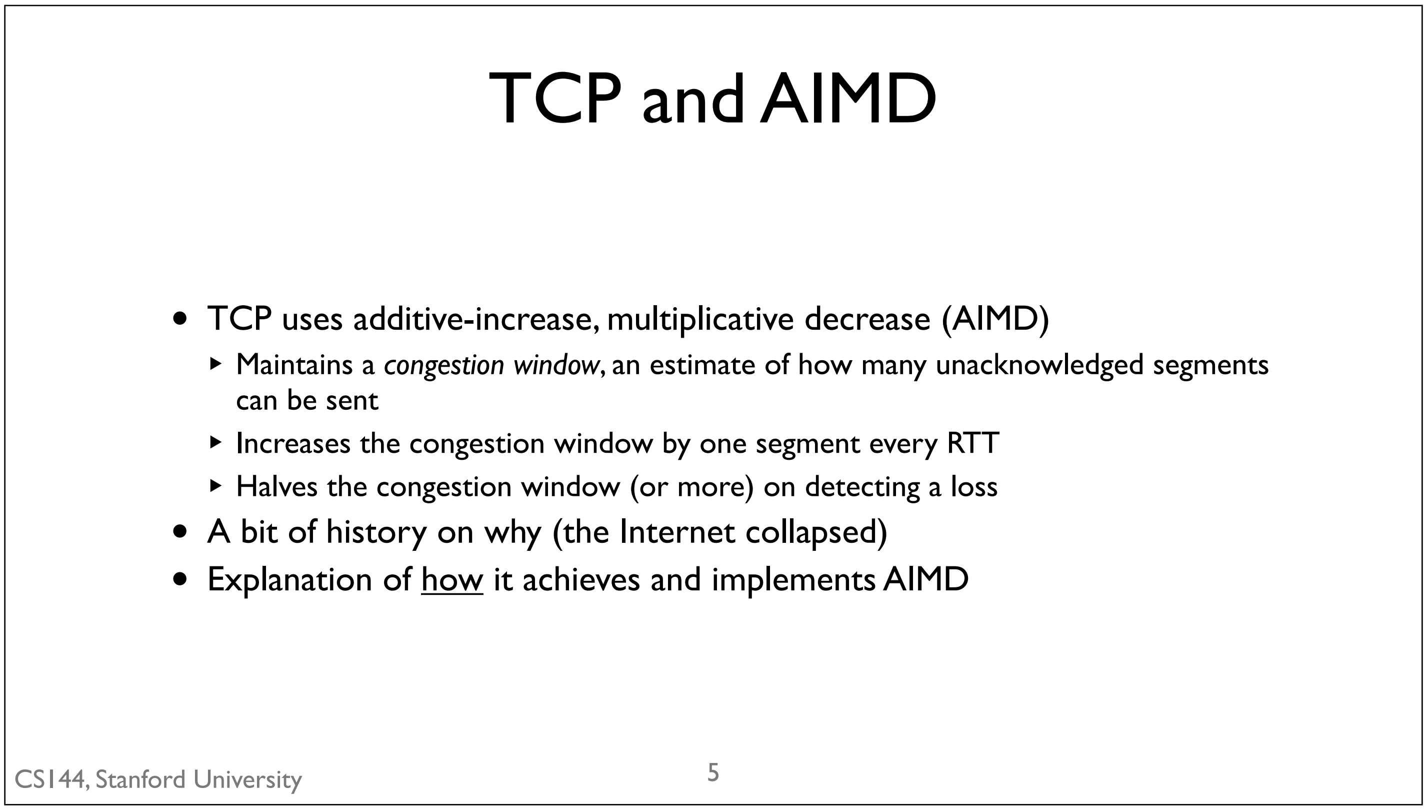


Congestion Control Motivation



Congestion control: limit outstanding data so it does not congest network, improves overall performance

So the purpose of congestion control is to control how many packets a sender has outstanding in the network. The goal is to send as many packets as the network can support but not more: sending more will cause queues to fill up and drop packets. At a high level, this seems simple. What makes it difficult, as we'll see, is that in TCP's case the senders have very limited information on the internal state of the network and so must infer losses from these limited signals.

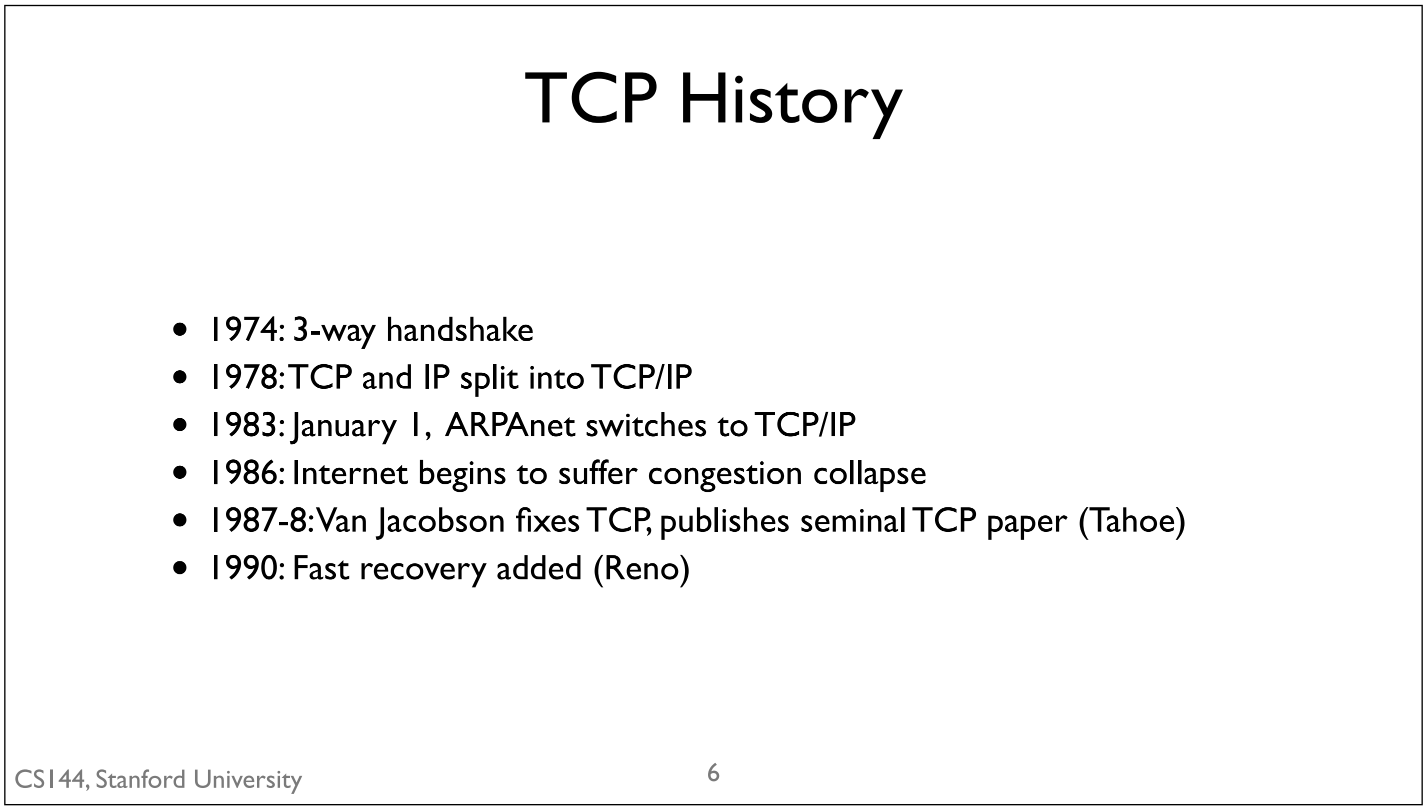


TCP and AIMD

- TCP uses additive-increase, multiplicative decrease (AIMD)
  - Maintains a *congestion window*, an estimate of how many unacknowledged segments can be sent
  - Increases the congestion window by one segment every RTT
  - Halves the congestion window (or more) on detecting a loss
- A bit of history on why (the Internet collapsed)
- Explanation of <u>how</u> it achieves and implements AIMD

The basic summary is that, in its steady state, TCP uses AIMD. It maintains a variable called its congestion window. The congestion window specifies how many unacknowledged segments the connection can have outstanding in the network. As the sender receives acknowledgements, these indicate that segments have left the network and so it can send more. Every round-trip-time (RTT), TCP increases the size of the congestion window by 1. This is the additive increase of AIMD. When TCP detects a packet loss, it halves the congestion window, or in some cases sets it to be 1 segment.

So far we've explained congestion control as a clear and obvious issue that arises, and AIMD as a simple and highly effective algorithm to manage congestion. But this wasn't realized from the beginning. I'm going to give a bit of history on how TCP evolved to use AIMD. This is a great story -- the Internet collapsed and became unusable! In response to this meltdown some researchers came up with AIMD. I'll explain exactly how it works.



TCP History

- 1974: 3-way handshake
- 1978: TCP and IP split into TCP/IP
- 1983: January 1, ARPAnet switches to TCP/IP
- 1986: Internet begins to suffer congestion collapse
- 1987-8: Van Jacobson fixes TCP, publishes seminal TCP paper (Tahoe)
- 1990: Fast recovery added (Reno)

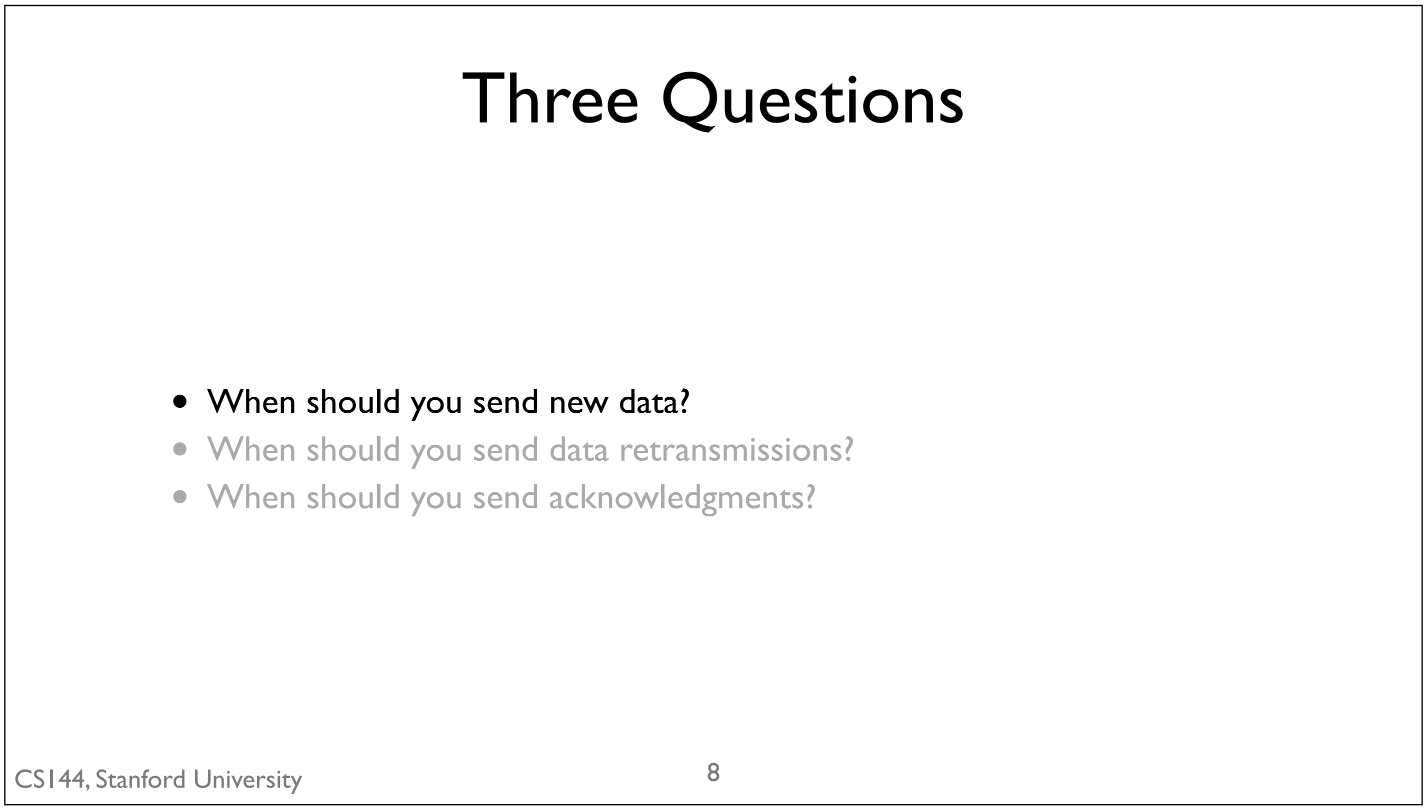
This is the brief, early history of TCP. In the mid-to-late 70s, Vint Cerf and others developed the 3-way handshake for connection establishment and split TCP into the two layers we know today: IP at the network layer and TCP at the transport layer. On January 1st, 1983, the entire APRAnet switched to TCP/IP. Three years later, in 1986, the Internet began to suffer from “congestion collapse.” As TCP streams started saturating links, routers dropped packets. But TCP didn’t respond well to these dropped packets. TCP spent most of its time retransmitting packets that had arrived successfully, continuing to waste the capacity of these saturated links. This kept packet losses high, such that new segments were dropped often and so applications saw very low data rates. The network was working furiously hard sending wasted segments. This coined the term “congestion collapse.” The network was tremendously congested but applications saw no useful work being done.

In 1987-8, Van Jacobson dug into what was happening and fixed TCP, publishing the seminal TCP paper whose algorithms underlie all TCP implementations today. This version of TCP was called TCP Tahoe, named after the particular release of BSD UNIX that it was packaged with. Two years later, some further improvements were added to TCP, in a version called TCP Reno. Modern TCP layers add a bit more complexity on top of TCP Reno for modern network speeds, but have TCP Reno at their core.

Three Questions

- When should you send new data?
- When should you send data retransmissions?
- When should you send acknowledgments?

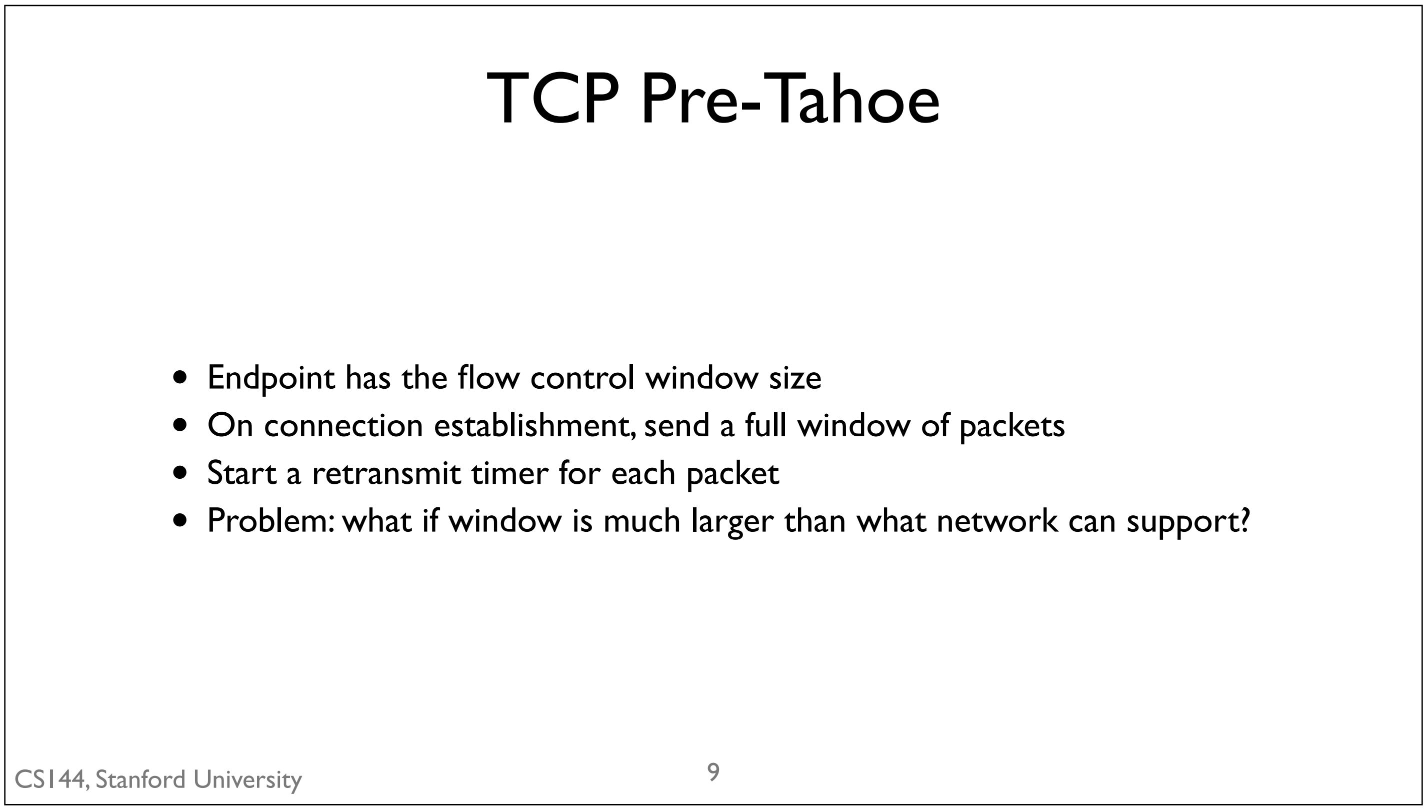
We can boil down TCP to three questions. When should you send new data? When should you retransmit data? And when should you send acknowledgements?



Three Questions

- When should you send new data?
- When should you send data retransmissions?
- When should you send acknowledgments?

In this video, I’m going to explain the answer to the first question: when should TCP send new data? I’ll explain the answer to the second two in the next video, on RTT estimation and self-clocking.



TCP Pre-Tahoe

- Endpoint has the flow control window size
- On connection establishment, send a full window of packets
- Start a retransmit timer for each packet
- Problem: what if window is much larger than what network can support?

Remember that TCP has a “window” field in its header, which one side of a connection uses to tell the other the size of its flow control window. The TCP specification says that a TCP sender shouldn’t send data past the last acknowledged byte plus the size of the flow control window. Flow control ensures that a sender doesn’t send data that a receiver can’t handle.

The original version of TCP would, once the three way handshake completed, send a full window of segments. So if a sender received a flow control window of 40 kilobytes and had 40 kilobytes to send, it would send 40 kilobytes worth of segments immediately. It would then start a retransmit timer for each packet. If the timer fired and the segment hadn’t been acknowledged, TCP would retransmit it. As acknowledgements come in, they can advance the sender’s window when the sum of the acknowledgement number and window fields indicate the receiver can handle more data.

This turns out to be a problem if the window is much larger than what the network can support. Suppose, for example, that the bottleneck link between the two endpoints can only queue a few packets. As soon as the handshake completes, the sender sends 30 or more packets. After the first few fill up the bottleneck queue, the rest will be dropped.

TCP in 1986

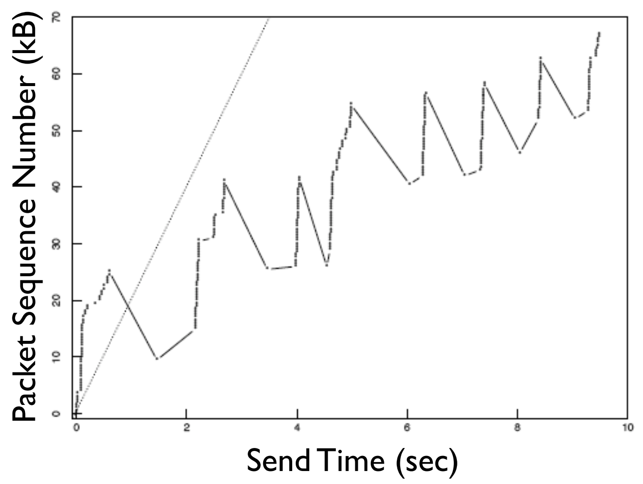


Figure from "Congestion Avoidance and Control", Van Jacobson and Karels. Used with permission.

And this is exactly what was observed. This plot is from the seminal paper that established TCP's congestion control mechanisms. The X axis shows time, in seconds. The Y axis shows, in terms of segment sequence numbers, what segments TCP transmits. A line up and to the right means TCP is sending more data, while a line that jumps down and to the right means there's a retransmission, because TCP is sending an older sequence number. Two points with the same Y but different X values show retransmissions. The straight line shows the available bandwidth, about 20kilobits per second. Networks were much slower then! If TCP were behaving well, we'd see the dark line track this light line, filling the available bandwidth.

But that's not what we see. Instead, what we can see in this plot is that TCP immediately sends a lot of segments, much more than the network can handle. It then waits for nearly a second, until a timeout causes it to retransmit a packet, and another packet after that. Then, with the window advanced, it sends a flurry of more segments, some of which, again are lost. This jagged pattern means that TCP is losing packets from almost every burst it sends, and the overall slope of the line is well below the capacity of 20kbps.

Three Improvements

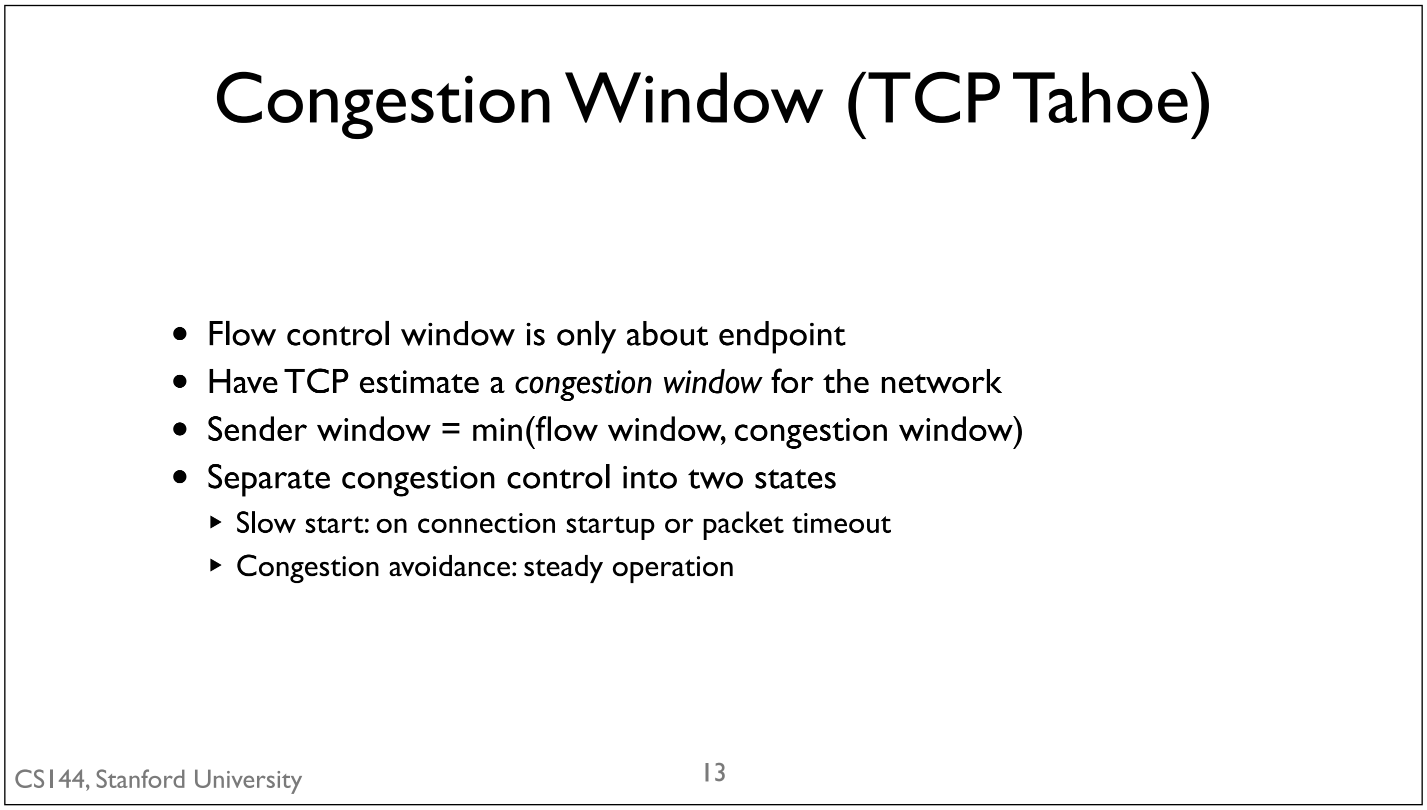
- Congestion window
- Timeout estimation
- Self-clocking

So TCP Tahoe added three improvements to properly control congestion: a congestion window, better timeout estimation, and self-clocking. In the rest of this video, I'll present each one.

Three Improvements

- Congestion window
- Timeout estimation
- Self-clocking

The first improvement TCP Tahoe added was something called a congestion window.



Congestion Window (TCP Tahoe)

- Flow control window is only about endpoint
- Have TCP estimate a *congestion window* for the network
- Sender window = min(flow window, congestion window)
- Separate congestion control into two states
  - Slow start: on connection startup or packet timeout
  - Congestion avoidance: steady operation

Recall that flow control is about what an endpoint can handle. TCP won't send data past what the flow control window specifies. But what if the endpoint can handle more data than the network can? The flow control window is only an upper bound on how much the sender should send. It could be, for good performance, it should send much less.

So TCP Tahoe estimates something called a congestion window for the network. Its sending window is the minimum of the flow window and congestion window. Don't send more than the other side can handle, and don't send more than the network can handle. To manage this congestion window, TCP separates congestion control into two states, called slow start and congestion avoidance. In the steady state, TCP is in the congestion avoidance state, in which if follows an AIMD policy. When a connection starts up or there is a packet timeout, TCP is in the slow start state, in which it does not follow an AIMD policy.

Slow Start Benefits

- Slow start
  - Window starts at Maximum Segment Size (MSS)
  - Increase window by MSS for each acknowledged packet
- Exponentially grow congestion window to sense network capacity
- "Slow" compared to prior approach

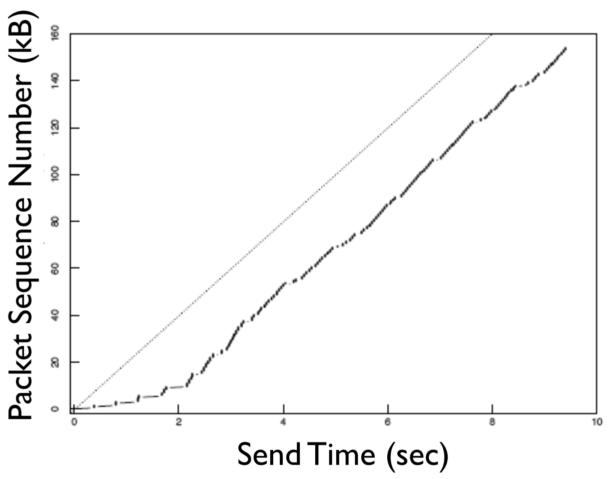
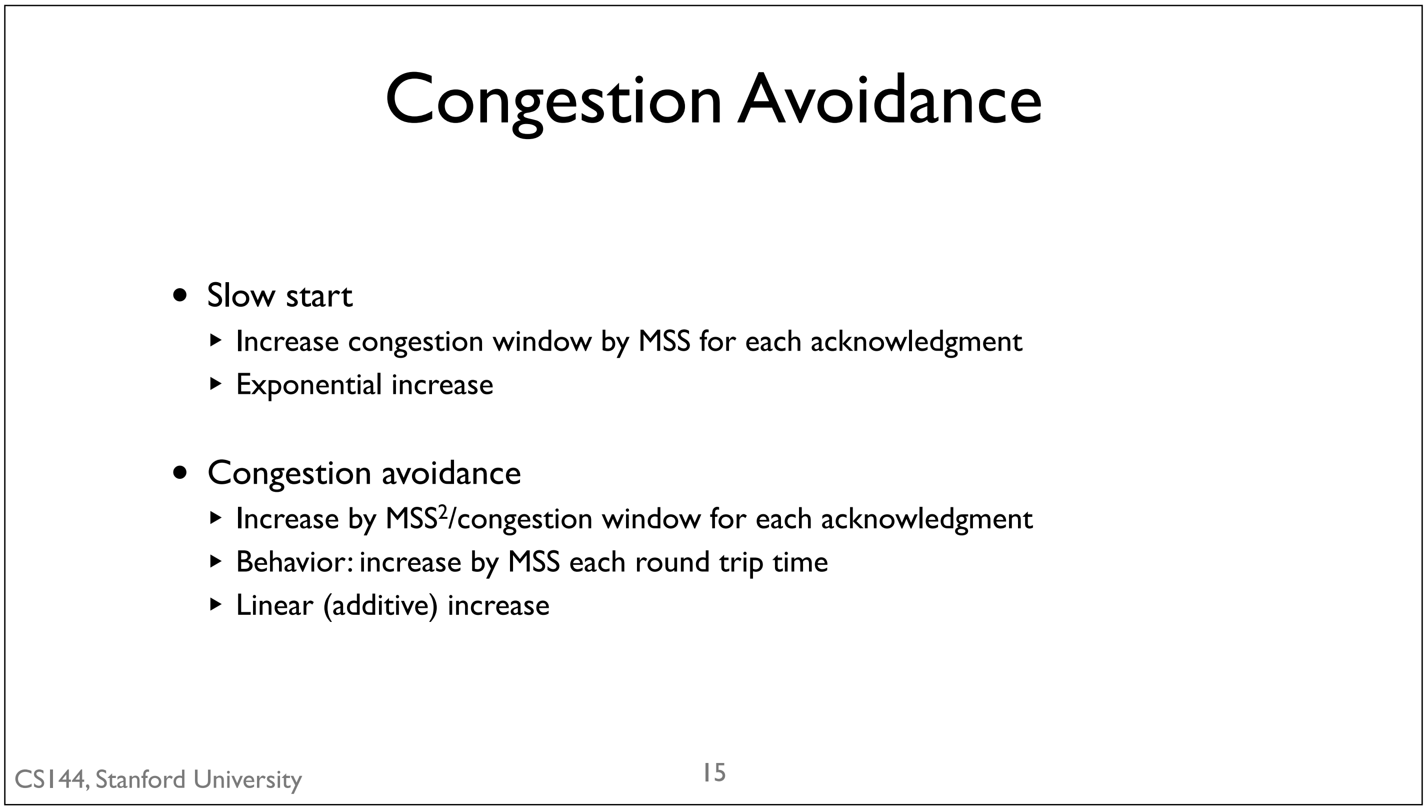


Figure from "Congestion Avoidance and Control", Van Jacobson and Karels. Used with permission.

The way to think about how the congestion avoidance and slow start works is in terms of how much they increase their congestion window. When TCP enters the slow start state, its congestion window is the maximum segment size, MSS, or one packet. Every time it receives a new acknowledgement, that is, an acknowledgement segment that acknowledges data that hadn't been acknowledged before, TCP increases the congestion window by one MSS.

This policy in slow start means that the congestion window grows *exponentially.* The sender starts with a window of size MSS. It sends a segment. When it receives an acknowledgement, it increases the congestion window to two MSS and sends two new segments. When it receives acknowledgements for these segments, it increases the congestion window to four MSS (one for each acknowledgement) and sends four new segments.

The name "slow start" might seem a bit misleading -- exponential increase is much faster than additive increase. But it's called slow start because it's slow in comparison to the old approach TCP used, of sending the whole window immediately.

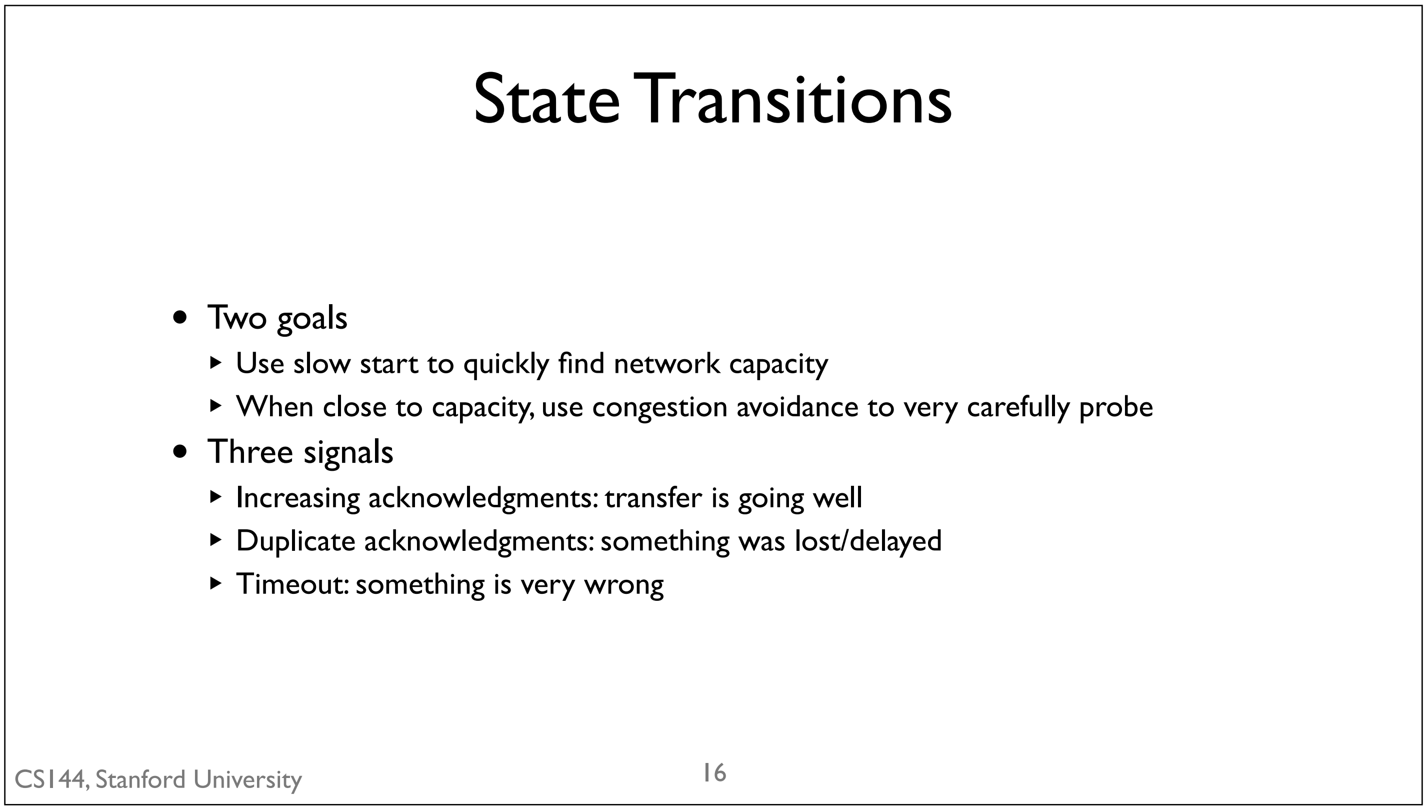


Congestion Avoidance

- Slow start
  - Increase congestion window by MSS for each acknowledgment
  - Exponential increase
- Congestion avoidance
  - Increase by MSS^2/congestion window for each acknowledgment
  - Behavior: increase by MSS each round trip time
  - Linear (additive) increase

In the congestion avoidance state, TCP increases the window much more slowly and resembled AIMD. It increases the window by MSS squared divided by the congestion window for each acknowledgement. Assuming no packets are dropped, this causes TCP to increase the congestion window by one MSS per round-trip-time. We want to increase the window by one MSS per round-trip-time. There are congestion window divided by MSS segments outstanding. If there are N outstanding segments, each acknowledgement should add 1/Nth of our desired increase to the congestion window. Put in terms of bytes, this means MSS/congestion window. Since our desired increase is MSS, this means each acknowledgement increases the window by MSS times (MSS divided by congestion window).

So this is the additive increase part of AIMD.



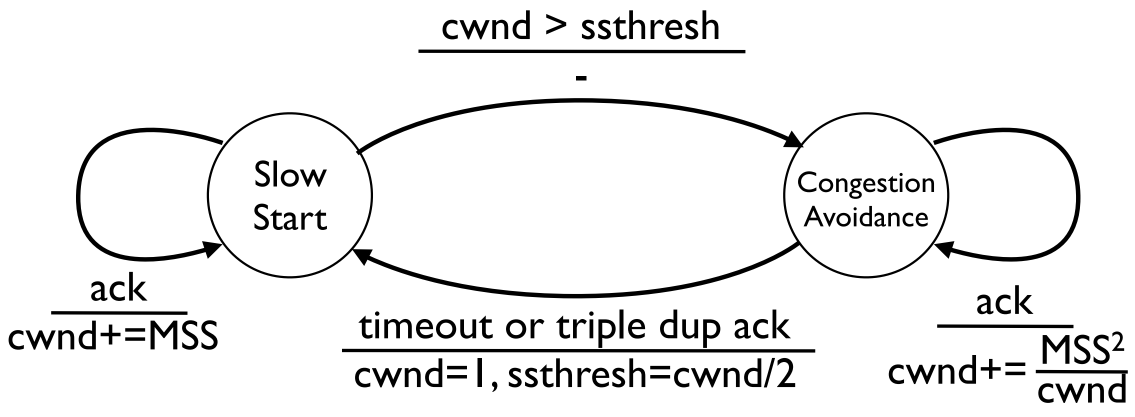
State Transitions

- Two goals
  - Use slow start to quickly find network capacity
  - When close to capacity, use congestion avoidance to very carefully probe
- Three signals
  - Increasing acknowledgments: transfer is going well
  - Duplicate acknowledgments: something was lost/delayed
  - Timeout: something is very wrong

So we have these two states. The first, slow start, allows TCP to quickly find the available network capacity. For example, suppose that the network can support a congestion window of 40 packets: waiting 40 round trip times to reach this value would take too long. But when we're close to the network capacity, we want to use congestion avoidance to more carefully probe using additive increase.

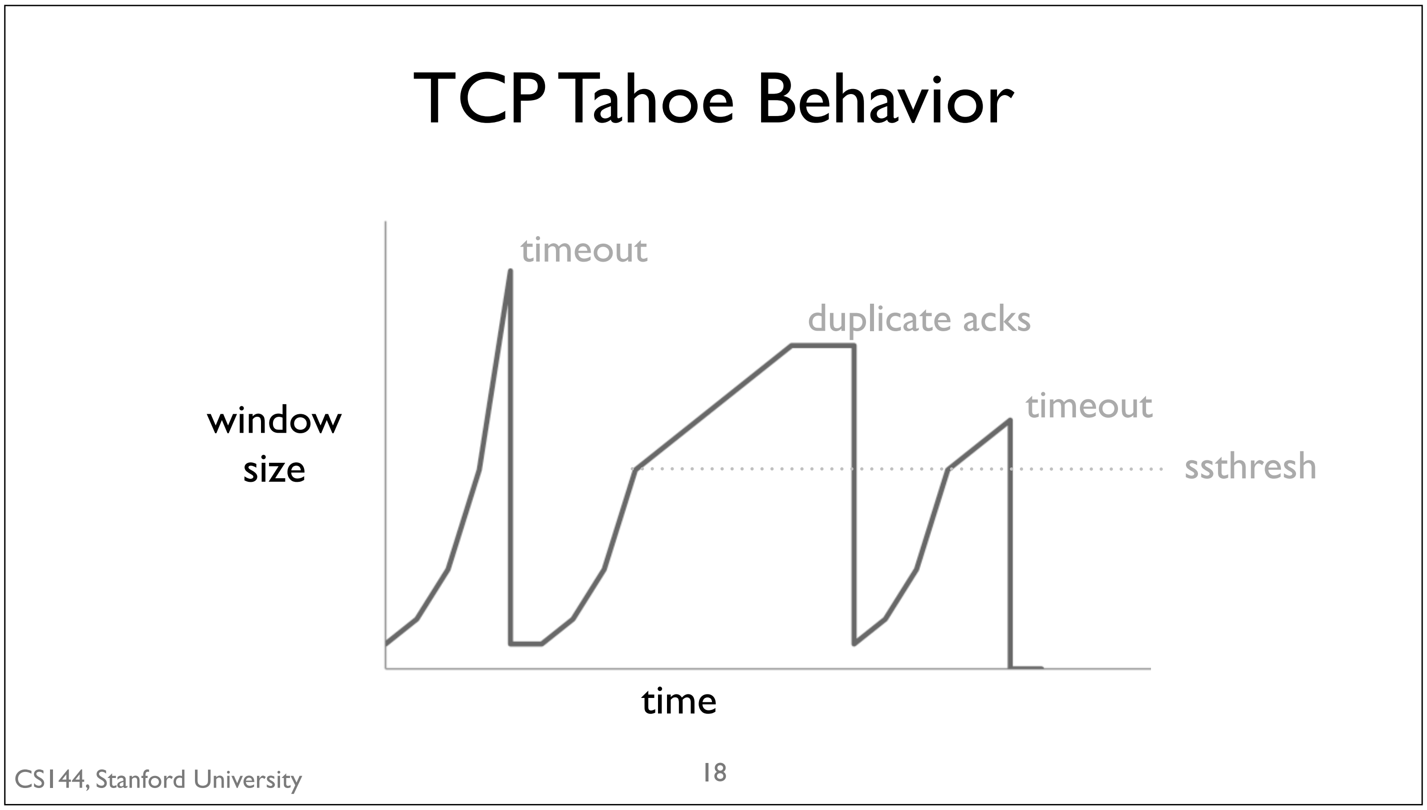
So how do we choose to transition between these two states? TCP has three signals available to it. Increasing acknowledgements mean the transfer is going well. Since TCP uses cumulative acknowledgements, duplicate acknowledgements mean that a segment was lost or delayed but other segments are arriving successfuly. Finally, if there's a timeout, then something is very wrong.

TCP Tahoe FSM

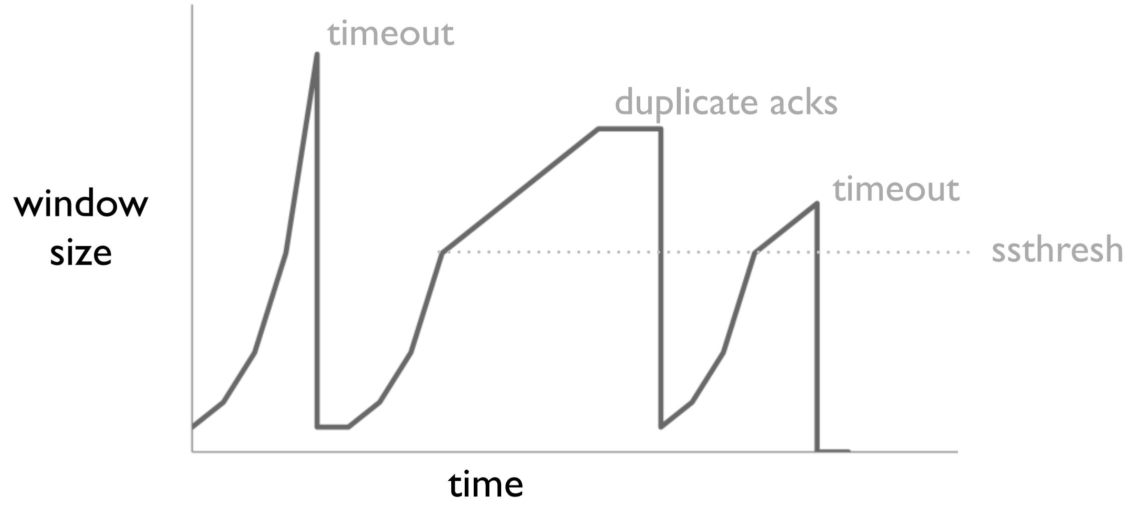


So this is what the TCP state machine looks like. A connection starts in the slow start state. From there, every time it receives an acknowledgement, it increases the congestion window, cwnd, by one MSSS. It increases the congestion window so until it passes a threshold called ssthresh, or slow start threshold. When the congestion window grows larger than this threshold, TCP transitions into the congestion avoidance state. While in the congestion avoidance state, it increases the congestion window more conservatively, one MSS per round trip time.

On a timeout or a triple duplicate acknowledgement, TCP transitions back to the slow start state. TCP infers that three duplicate acknowledgements (so 4 of the same ack) are good evidence that the next segment was lost. The receiver is continuing to receive segments, but can't forward the acknowledgement number because it's missing one. When TCP transitions back to the slow start state, it sets the ssthresh variable to be half of the congestion window. This value sets the cutoff after which TCP will follow AIMD. So on a packet loss, TCP enter slow start, then AIMD.



TCP Tahoe Behavior



This figure shows an example of how TCP Tahoe's congestion window behaves over time. It starts with a size of one MSS, and increases exponentially. This first drop is in response to a timeout. So the window is returned to MSS, and begins to climb exponentially again, until it reaches half of its original value, at which point it begins growing additively. It grows until a segment is lost and there are three duplicate acknowledgements (the window does not increase in response to these duplicate acks). It then drops again to one MSS, increases exponentially following slow start until it reaches ssthresh, then starts increasing additively again.

If you look carefully, ssthresh is the same value both times TCP returns to slow start, in that it transitions to congestion avoidance at the same window size. For this simple plot, this occurs because I calculated ssthresh in terms of integer numbers of MSS, and in both cases it rounded down to the same value.

Note that TCP Tahoe doesn't strictly manage congestion using AIMD. While AIMD is an excellent algorithm for managing the steady state, or a stable network, in practice TCP has to deal with a much wider range of conditions, such as startup, transient network failures that lose bursts of packets, and sudden changes in the available bandwidth.

So recall those three questions: when does TCP send new data, when does it retransmit data, and when does it acknowledge data. Tis answers the first question: TCP sends new data when its sender window, defined as the minimum of its congestion window and flow control window, allows it to do so. The congestion window is a value a sender maintains based on the acknowledgements and timeouts it observes.