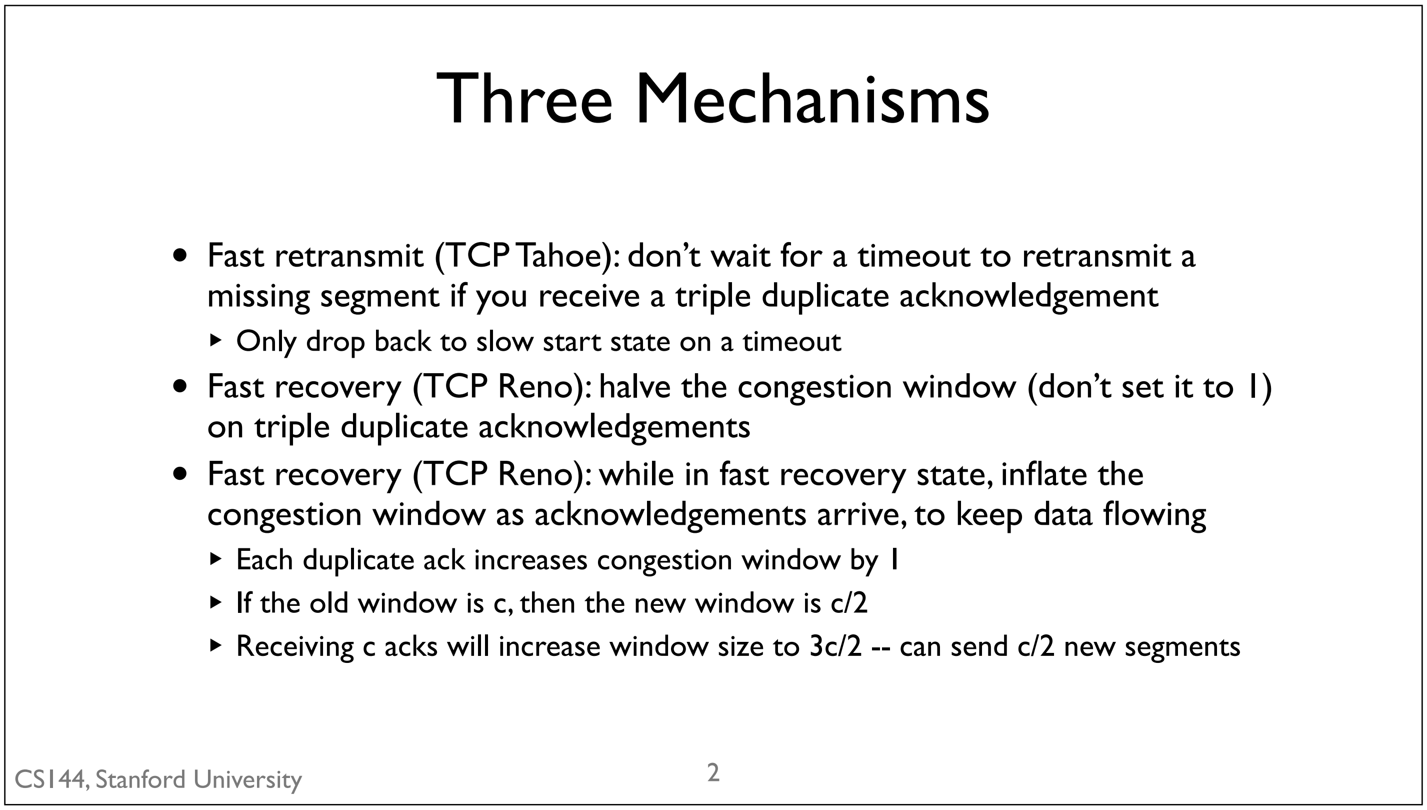
Congestion Control III

Performance improvements: fast retransmit and fast recovery

CS144, Stanford University

In this video I'm going to talk about two performance improvements in TCP: fast retransmit and fast recovery.



Three Mechanisms

- Fast retransmit (TCP Tahoe): don’t wait for a timeout to retransmit a missing segment if you receive a triple duplicate acknowledgement
  - Only drop back to slow start state on a timeout
- Fast recovery (TCP Reno): halve the congestion window (don’t set it to 1) on triple duplicate acknowledgements
- Fast recovery (TCP Reno): while in fast recovery state, inflate the congestion window as acknowledgements arrive, to keep data flowing
  - Each duplicate ack increases congestion window by 1
  - If the old window is c, then the new window is c/2
  - Receiving c acks will increase window size to 3c/2 -- can send c/2 new segments

In addition to the congestion avoidance state, improved RTT estimation, and self-clocking, TCP uses three more mechanisms. These mechanisms improve TCP’s performance, by softening its response to a packet loss. Recall that the mechanisms for TCP Tahoe cause it to behave more conservatively than AIMD. If a packet is lost, TCP Tahoe drops to the slow start state, at which point it exponentially increases the congestion window up to ssthresh. At that point, it enters additive-increase.

The first mechanism, called fast retransmit, was part of TCP Tahoe. If a TCP Tahoe sender receives 3 duplicate acknowledgements (so 4 acks for the same sequence number), it assumes that the next segment was lost and retransmits it immediately without waiting for a timeout. Assuming this retransmission succeeds, this reduces the delay before the sender receives an acknowledgement and can move the send window forward to send new data. TCP Tahoe treats these triple duplicate acknowledgements as a loss and sets the congestion window to 1, dropping into the slow start state.

TCP Reno added a second algorithm, called fast recovery, which has two mechanisms. The first is that when a loss is detected by triple duplicate ack, TCP Reno does not set the congestion window to 1 and drop into the slow start state. Instead, it halves the congestion window. Since ssthresh is set to half the congestion window, this means TCP Reno does not exit the congestion avoidance state. Using this algorithm, TCP Reno, in a steady state with no timeouts, follows an AIMD policy. On a loss, it halves its congestion window, multiplicative decrease, and uses additive increase. On a timeout, TCP Reno behaves as in TCP Tahoe, setting the congestion window to 1.

The second mechanism TCP Reno added is that while in the fast recovery state, it “inflates” the congestion window by 1 for each duplicate acknowledgement. This is to prevent the problem that, in the case of a single loss, TCP cannot send data for a complete round trip time because it is waiting for an acknowledgement to advance its send window. Since each duplicate acknowledgement means a segment has left the network successfully, TCP could in theory send a new segment without congesting the network.

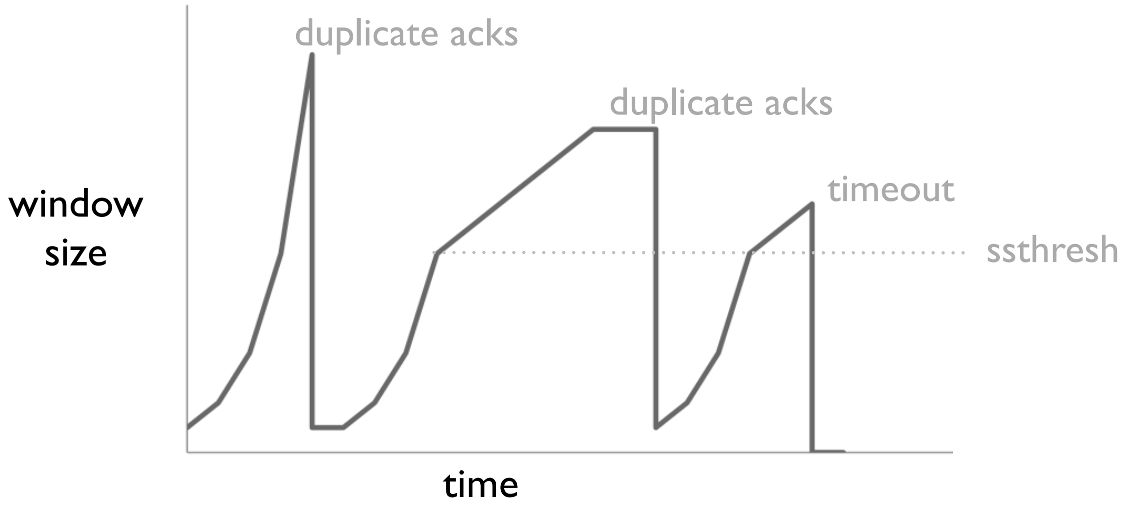
Suppose the old congestion window size was c. By increasing its congestion window by 1 for each duplicate ack, the congestion window grows from c/2 -- fast recovery -- to c + c over 2, or 3c over 2. This means that for acknowledgements c over 2 to c, TCP can send a new segment, as the congestion window has advanced further ahead. Once TCP receives a new acknowledgement, it resets its congestion window to the correct value, so c/2.

TCP Tahoe

- On timeout or triple duplicate ack (implies lost packet)
  - Set threshold to congestion window/2
  - Set congestion window to 1
  - Retransmit missing segment (fast retransmit for triple duplicate ack)
  - Enter slow start state

So let's put this together. TCP Tahoe, when it has a timeout or a triple duplicate acknowledgement, takes 3 steps. First, it sets ssthresh to half of the congestion window. Second it sets the congestion window to 1. Third, it retransmits the missing segment. These first two steps mean that it enters the slow start state and exponentially increases its congestion window until it experiences another loss or reaches ssthresh.

TCP Tahoe Behavior



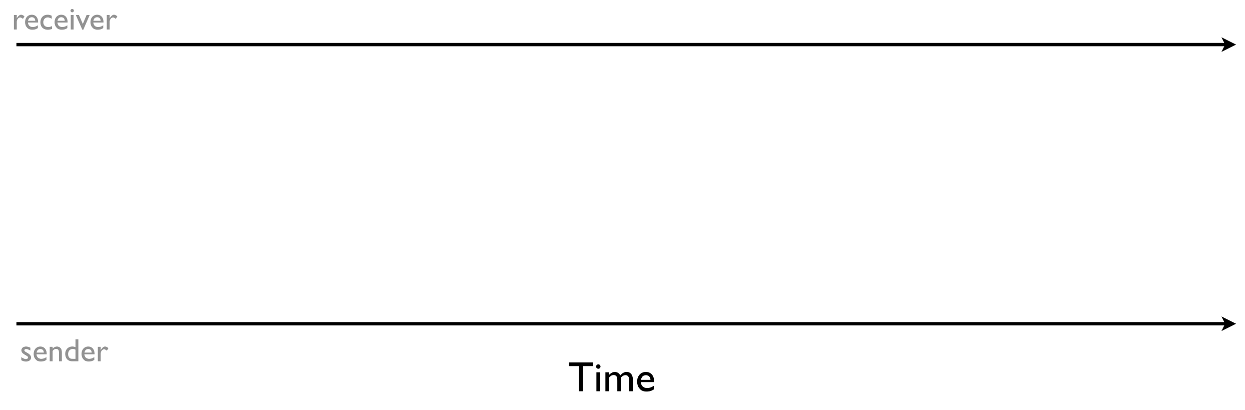
This figure shows an example of how TCP Tahoe's congestion window behaves over time. It starts with a size of one MSS, and increases exponentially. It observes a triple duplicate ack. So the window is returned to MSS, and begins to climb exponentially again, until it reaches half of its original value, 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.

TCP Tahoe Walkthrough



Now let’s walk through what TCP Tahoe does when it encounters a triple duplicate acknowledgement. Let’s say the congestion window is 8MSS, and a segment is lost. TCP will receive a total of 7 duplicate acknowledgements. After the third one, it retransmits the missing segment, sets its congestion window to 1MSS and ssthresh to 4. When it receives the ack for the retransmission, it sends one new segment. When it receives an ack for this segment, it increases its congestion window to 2. When it receives acks for these two segments, it increases its congestion window to 3 then 4. On reaching 4, it enters the congestion avoidance state. The next 4 acks will therefore increase the congestion window only by one MSS.

So that’s TCP Tahoe.

TCP Reno

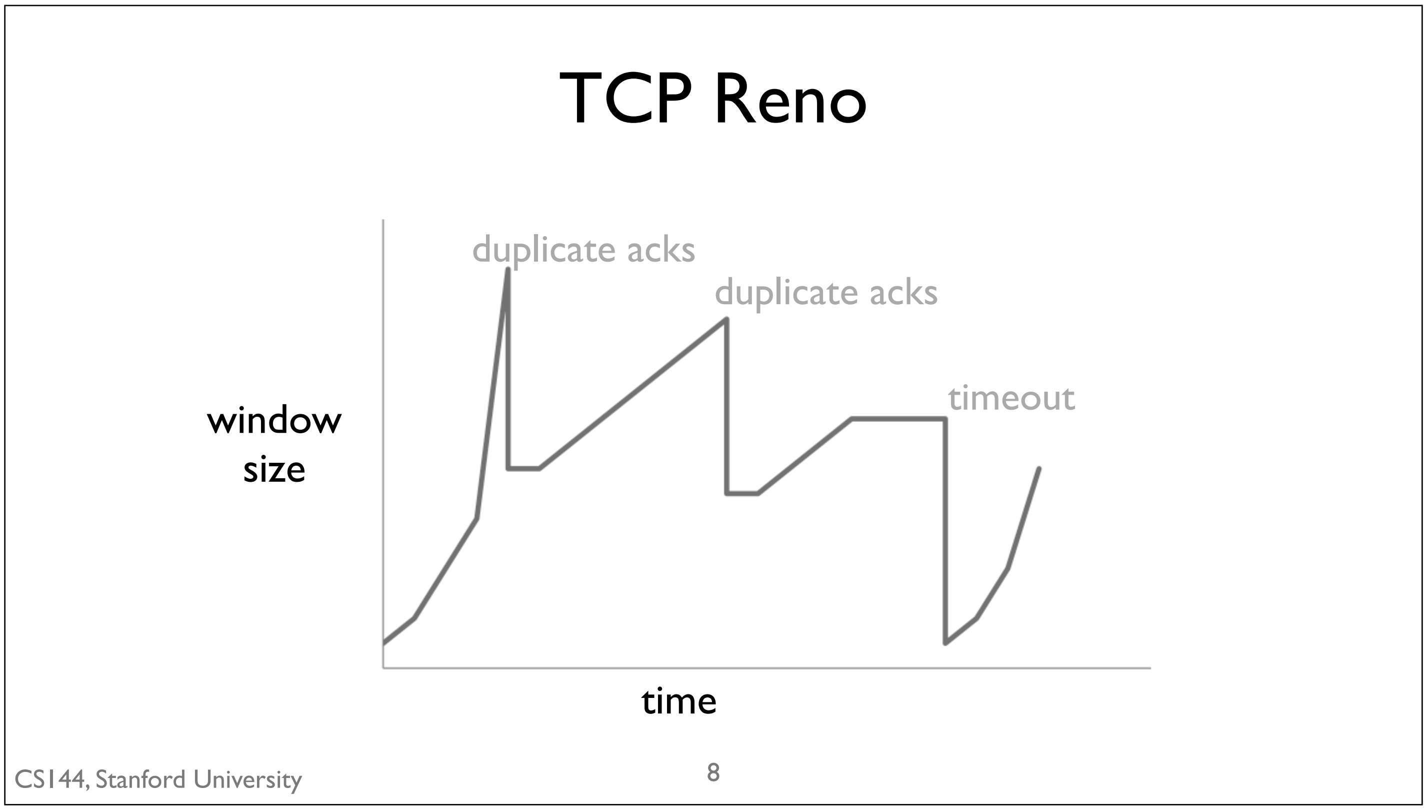
- Same as Tahoe on timeout
- On triple duplicate ack
  - Set threshold to congestion window/2
  - Set congestion window to congestion window/2 (fast recovery)
  - Inflate congestion window size while in fast recovery (fast recovery)
  - Retransmit missing segment (fast retransmit)
  - Stay in congestion avoidance state

TCP Reno behaves identically to TCP Tahoe on a timeout. On a triple duplicate ACK, it performs fast retransmit, sending the segment immediately. Instead of setting the congestion window size to 1, it halves it, thereby staying in the congestion avoidance state. For each duplicate acknowledgement, it inflates the congestion window by 1, such that it sends new segments before the retransmitted segment is acknowledged.

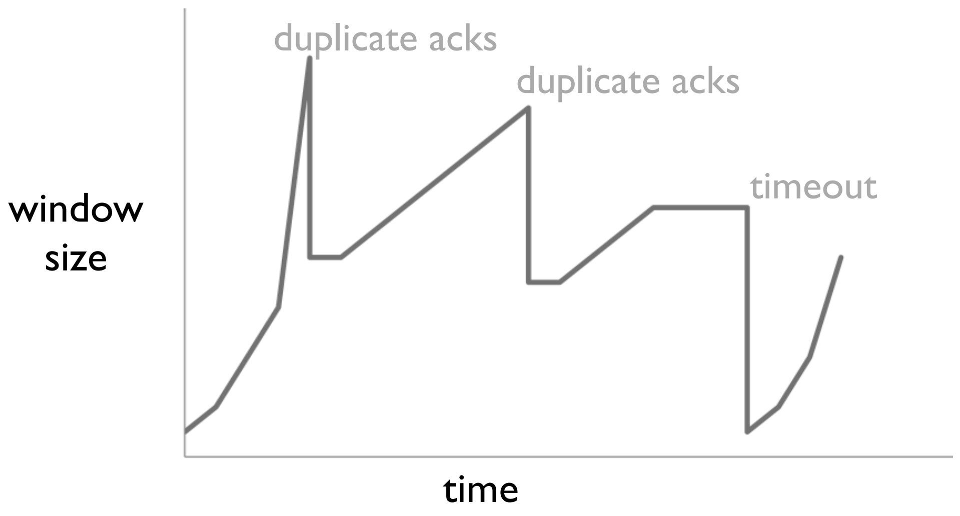
TCP Reno

- Same as Tahoe on timeout
- On triple duplicate ack
  - Set threshold to congestion window/2
  - Set congestion window to congestion window/2 (fast recovery)
  - Inflate congestion window size while in fast recovery (fast recovery)
  - Retransmit missing segment (fast retransmit)
  - Stay in congestion avoidance state

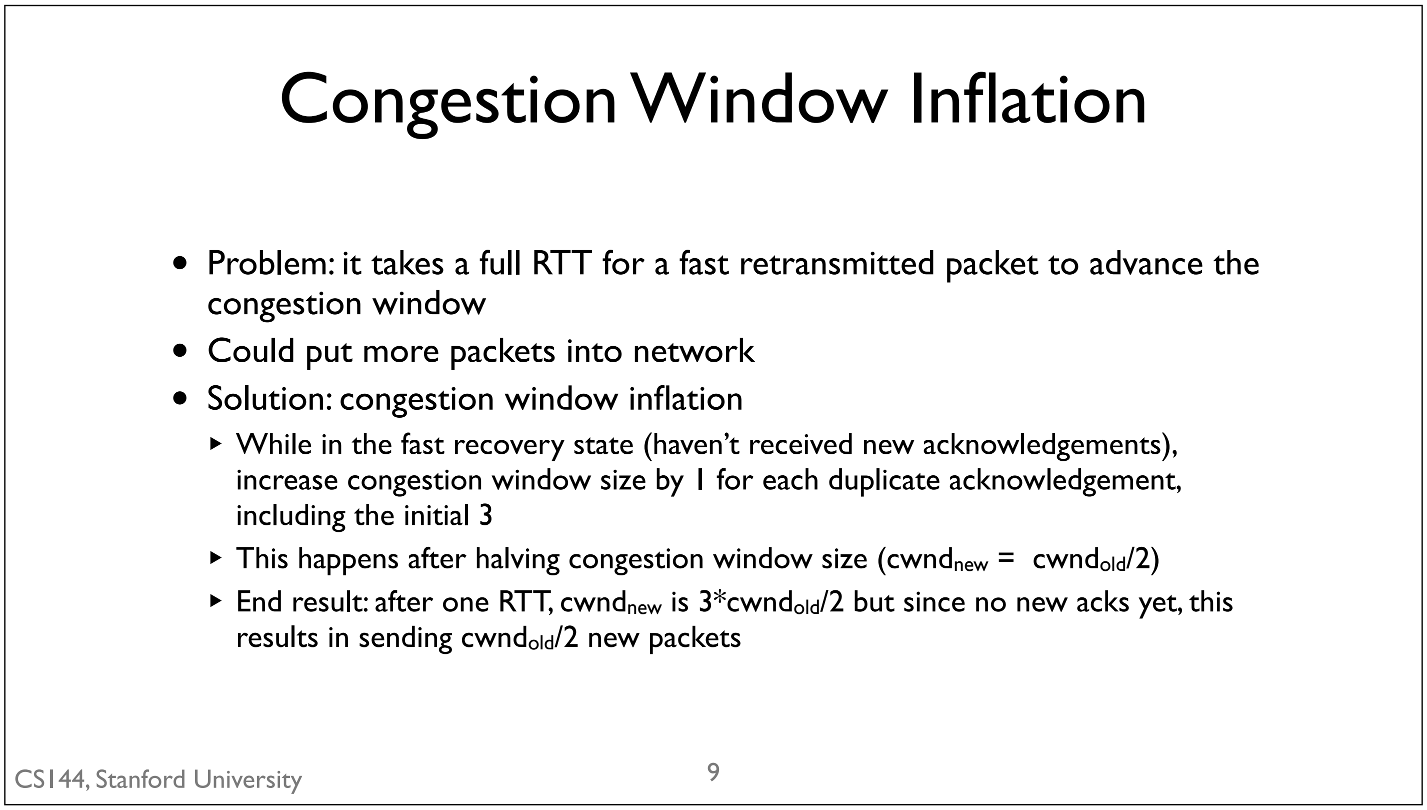
So the major difference between Reno and Tahoe is fast recovery.



TCP Reno



So this is a figure showing TCP Reno's behavior. It starts in slow start state, until it sees a triple duplicate acks. It halves the congestion window, staying in the congestion avoidance state, and performs fast retransmit. It then starts increasing the congestion window using AIMD. In the flat spot before AIMD, it uses congestion window inflation to send new segments -- I don't show that inflation in the window size. On a second triple duplicate ack, the same occurs. It performs a fast retransmit, halves the window, and stays in congestion avoidance. On a timeout, it sets the congestion window to 1 and re-enters the slow start state.

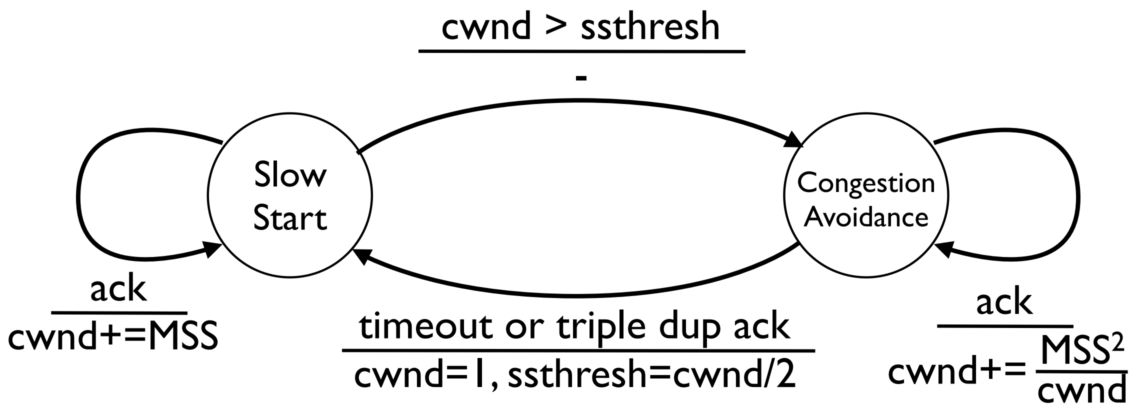


Congestion Window Inflation

- Problem: it takes a full RTT for a fast retransmitted packet to advance the congestion window
- Could put more packets into network
- Solution: congestion window inflation
  - While in the fast recovery state (haven't received new acknowledgements), increase congestion window size by 1 for each duplicate acknowledgement, including the initial 3
  - This happens after halving congestion window size ($cwnd_{new} = cwnd_{old}/2$)
  - End result: after one RTT, $cwnd_{new}$ is $3*cwnd_{old}/2$ but since no new acks yet, this results in sending $cwnd_{old}/2$ new packets

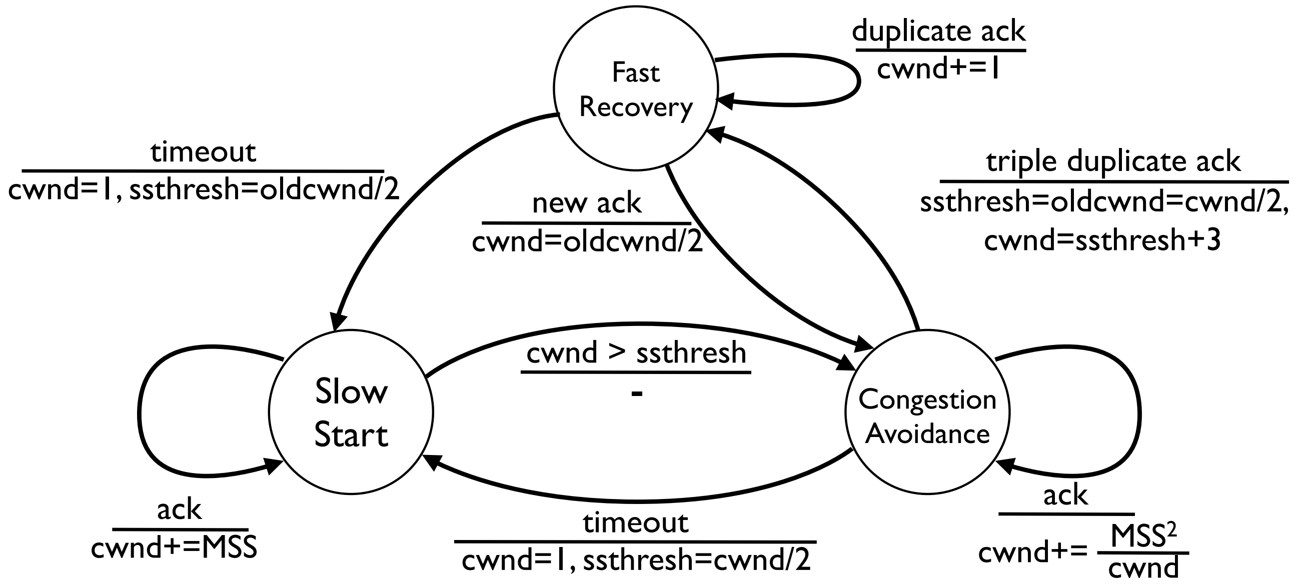
Congestion window inflation is part of fast recovery. These are the details of how it works. When TCP Reno enters fast recovery, it inflates its congestion window by 1 for each duplicate acknowledgement. Since the congestion window was halved, this means that the congestion window can grow larger than it was originally, up to 3 times the old value divided by 2. This means that the sender will send old congestion window divided by 2 new segments, minus one for the lost segment, almost exactly the amount that obeying AIMD requires.

TCP Tahoe FSM



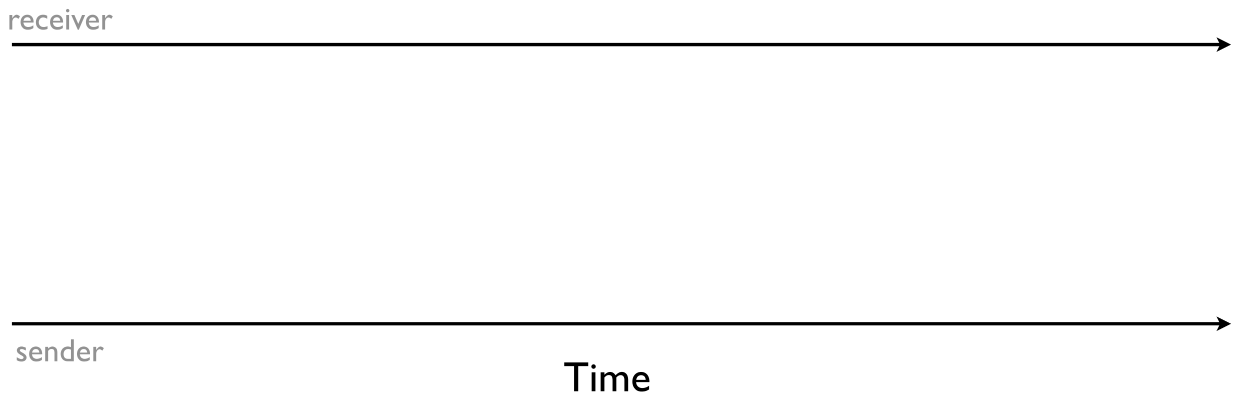
Recall that this is the TCP Tahoe FSM, with two states, slow start and congestion avoidance.

TCP Reno FSM



And this is the full TCP Reno FSM. It adds the third state, Fast Recovery. On a triple duplicate ack, rather than transition to slow start, it transitions to fast recovery. In the fast recovery state, it transitions back to congestion avoidance when it receives a new acknowledgement, resetting the congestion window to be half of the congestion window size when it transitioned to the fast recovery state. On a timeout, it returns to slow start just as in congestion avoidance. On receiving a duplicate ack, it increments the congestion window by 1.

TCP Reno Walkthrough



Let’s walk through TCP Reno’s behavior. Suppose we start with a congestion window size of 8MSS and a segment is dropped. The sender will receive 7 duplicate acks. After the first three, it will shrink the congestion window to be 4, remaining in the congestion avoidance state. It will the inflate the congestion window by 3, to 7MSS. On the next, fourth, duplicate acknowledgement, the congestion window grows to 8MSS. The next three acknowledgements increase it to 9, 10, then 11MSS, such that the sender can send three new segments. At around this time, the sender will acknowledge the retransmission, whose ack number moves the send window up to include all of the segments that triggered the duplicate acknowledgements.

At this point, TCP Reno deflates the congestion window to the correct value, half of its old value. This allows TCP Reno to send one new segment, at which point it will wait for acknowledgements from the segments sent during fast recovery.

Congestion Control

- One of the hardest problems in robust networked systems
- Basic approach: additive increase, multiplicative decrease
- Tricks to keep pipe full, improve throughput
  - Fast retransmit (don't wait for timeout to send lost data)
  - Congestion window inflation (don't wait an RTT before sending more data)

Congestion control is a very hard problem. Recall that it wasn't something anyone expected: it was an emerging behavior that the early Internet developers and users observed and had to tackle in order to make the Internet work again. The basic approach that TCP uses today is additive increase, multiplicative decrease, or AIMD. But there are a lot of details on exactly how that works, exactly how and when TCP sends data, retransmits data, and sends acknowledgements. Making AIMD work well and stably in practice requires tackling a bunch of edge cases.

Almost all TCP variants used today have TCP Reno at their core. There have been some additions to better deal with modern network speeds, but when you open a connection to your favorite website, your OS is using TCP Reno, with slow start, congestion avoidance, fast retransmit, and fast recovery.