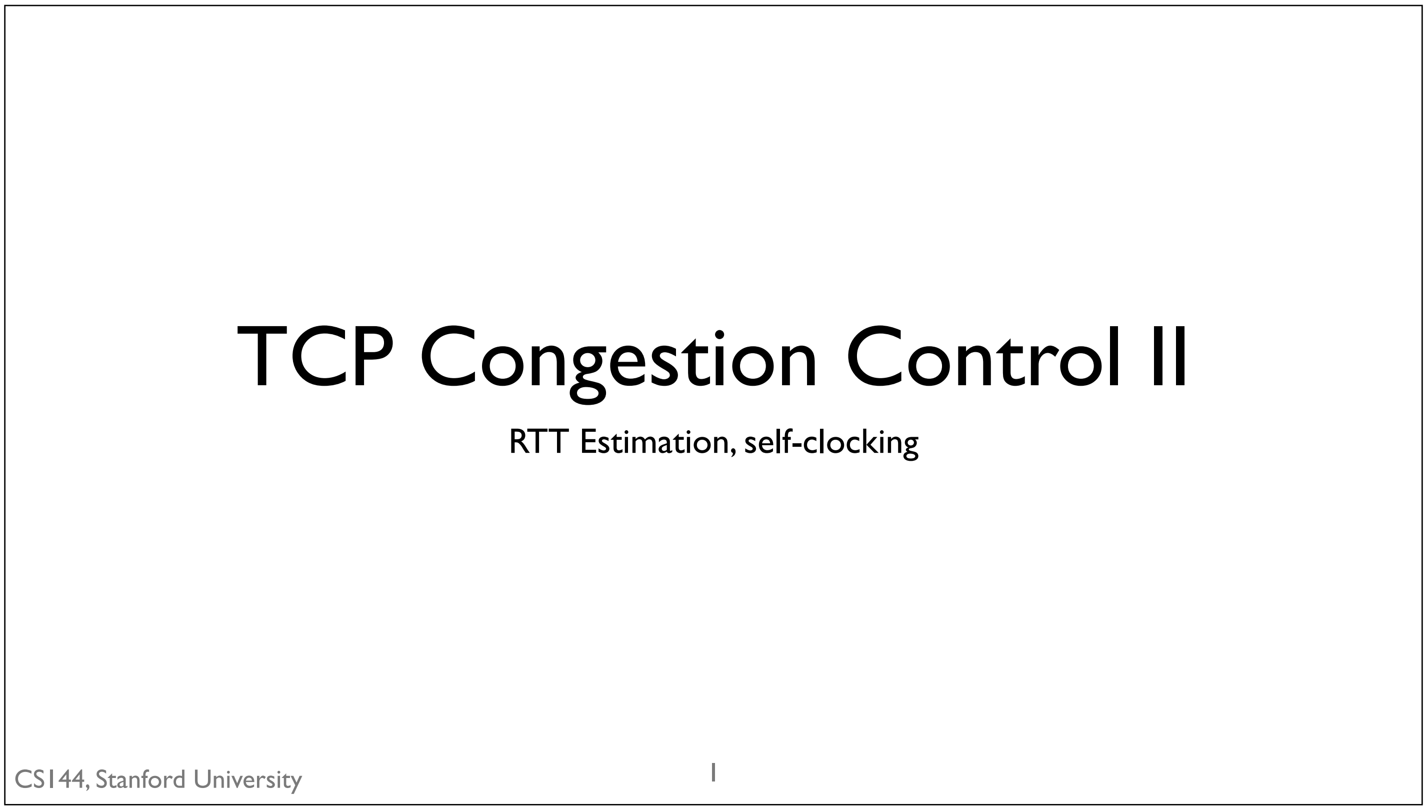


TCP Congestion Control II

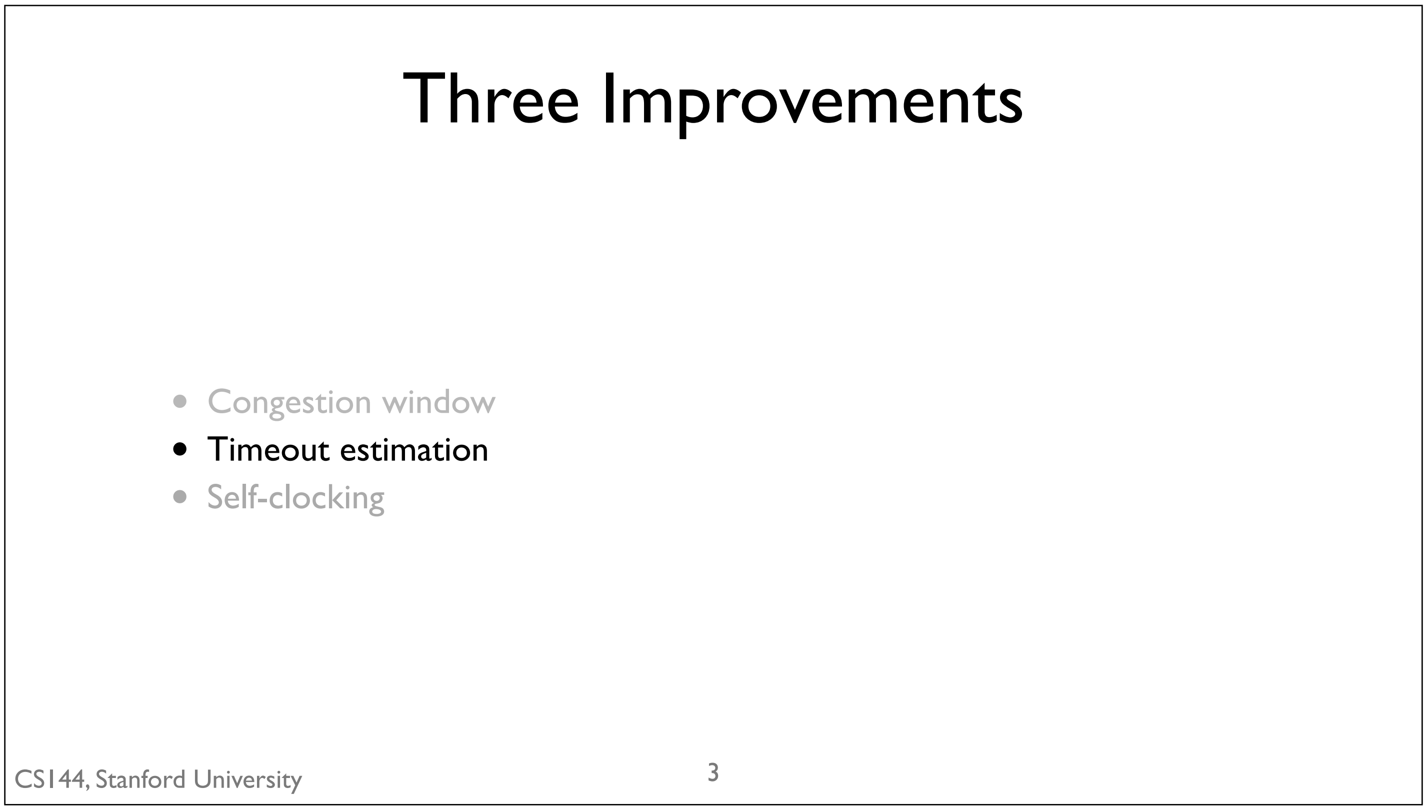
RTT Estimation, self-clocking

Recall that TCP Tahoe made three major improvements that solved the congestion collapse problem the Internet encountered. In the prior video, I talked about how it incorporated a congestion window its two states of operation, slow start and congestion avoidance. In this video I'll explain the other two mechanisms, RTT estimation and self-clocking.

Three Questions

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

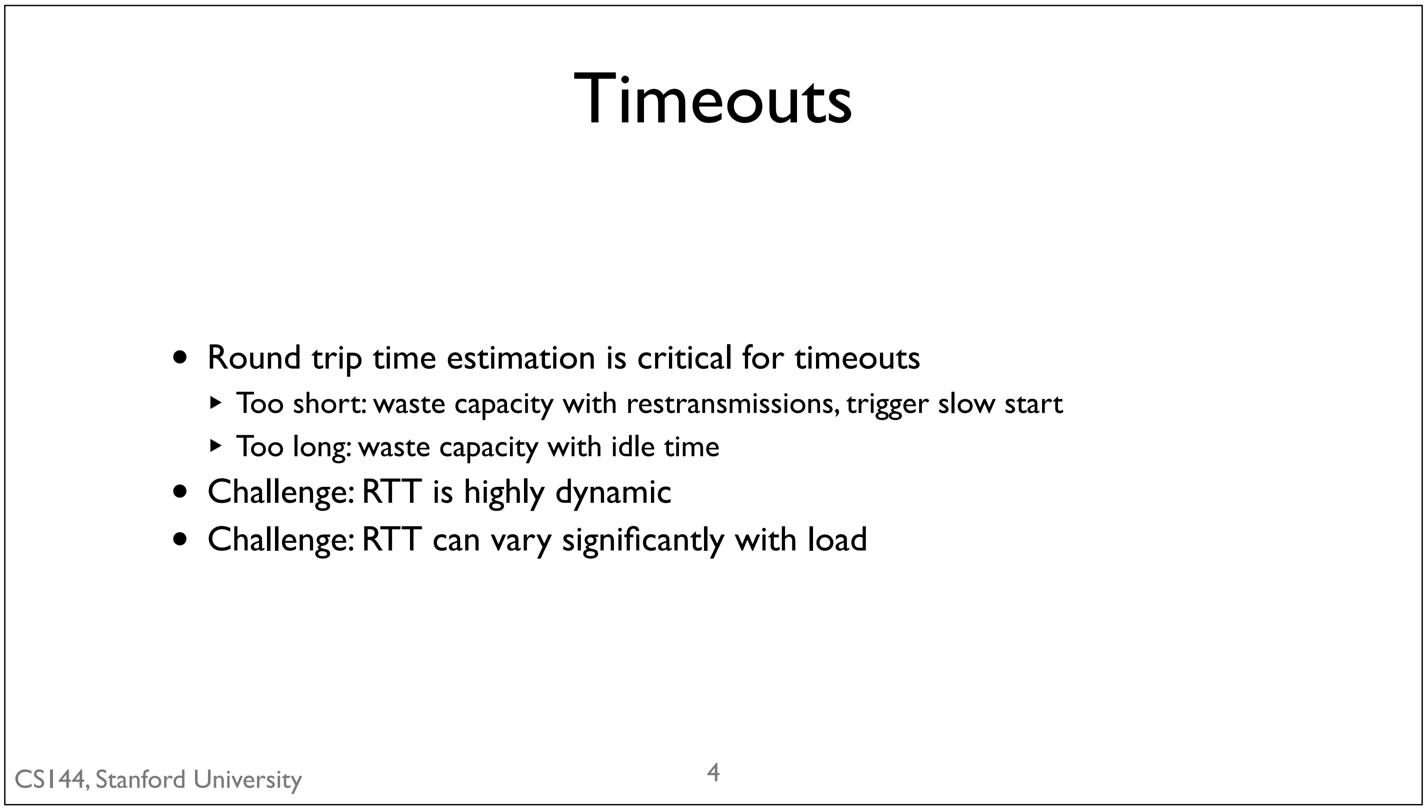
Recall that a TCP implementation boils down to three questions. The first is when you should send new data. TCP's answer is to use a congestion window to limit its transmissions when the network is congested. Next, let's answer the second question: when should TCP retransmit data?



Three Improvements

- Congestion window
- Timeout estimation
- Self-clocking

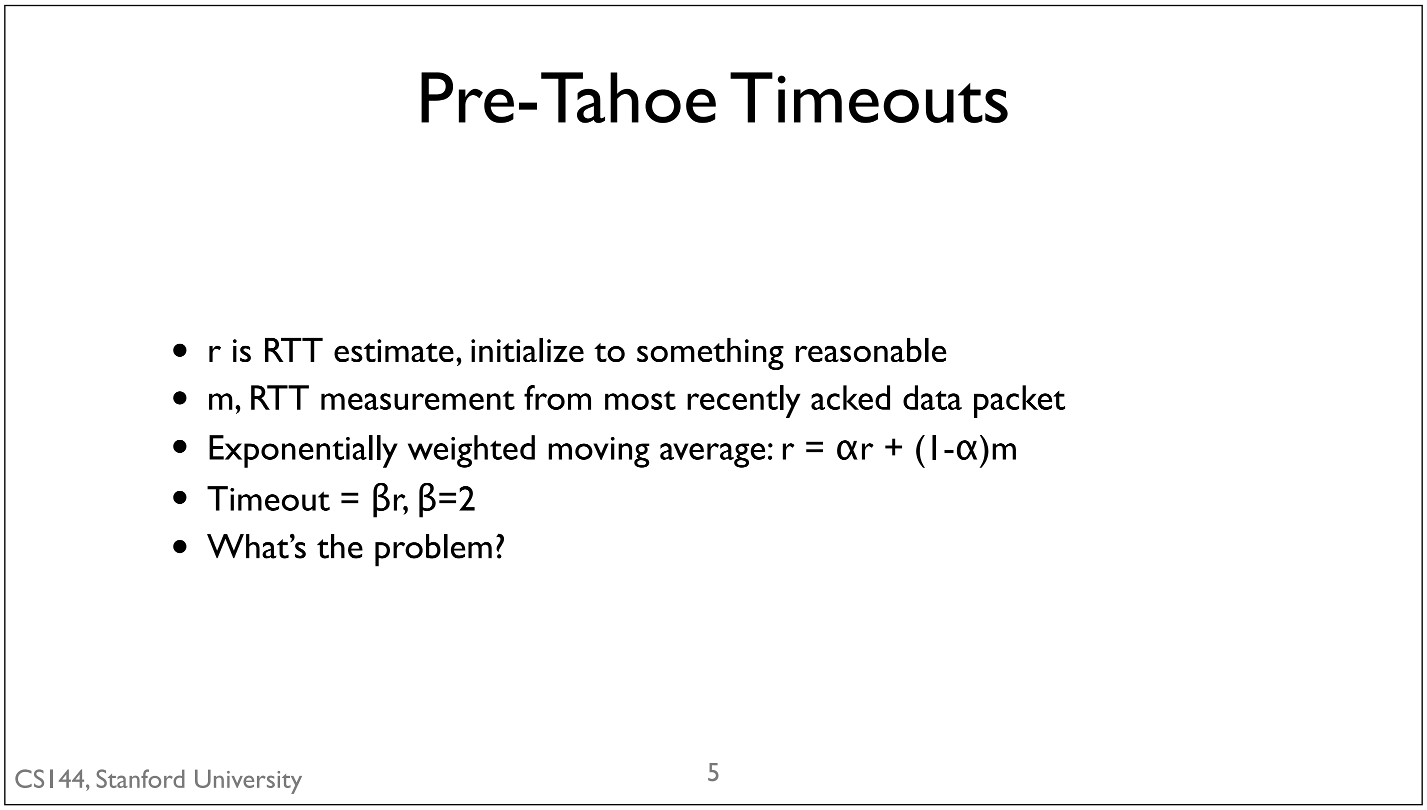
It turns out that estimating retransmission timeouts well can have a significant effect on TCP's behavior. Choosing timeouts that are too long will cause TCP to stall, waiting for acknowledgements. Choosing timeouts that are too short will cause TCP to back off too aggressively, dropping into the slow start state. So TCP Tahoe introduced some improvements to timeout estimation. These improvements have turned out to be a generally good way to estimate noisy signals in networked systems, so learning them can give guidance on designing other protocols.



Timeouts

- Round trip time estimation is critical for timeouts
  - Too short: waste capacity with restransmissions, trigger slow start
  - Too long: waste capacity with idle time
- Challenge: RTT is highly dynamic
- Challenge: RTT can vary significantly with load

Since TCP transitions to the slow start state and retransmits on a timeout, having a good timeout value is critical. If the round trip time is a constant, then the optimal retransmission timeout is just a tiny bit larger than the round trip time, because a successfully received packet's acknowledgement should take an RTT after the packet transmission. But round trip times are not constant. They can be highly dynamic. Furthermore, they can vary significantly with load. As a flow sends segments faster, it might start filling queues along the route, increasing the RTT. Bursts of traffic from other sources also vary the queueing delay. GIven all of this noise, we need a robust way to estimate how long before assuming a packet was lost.



Pre-Tahoe Timeouts

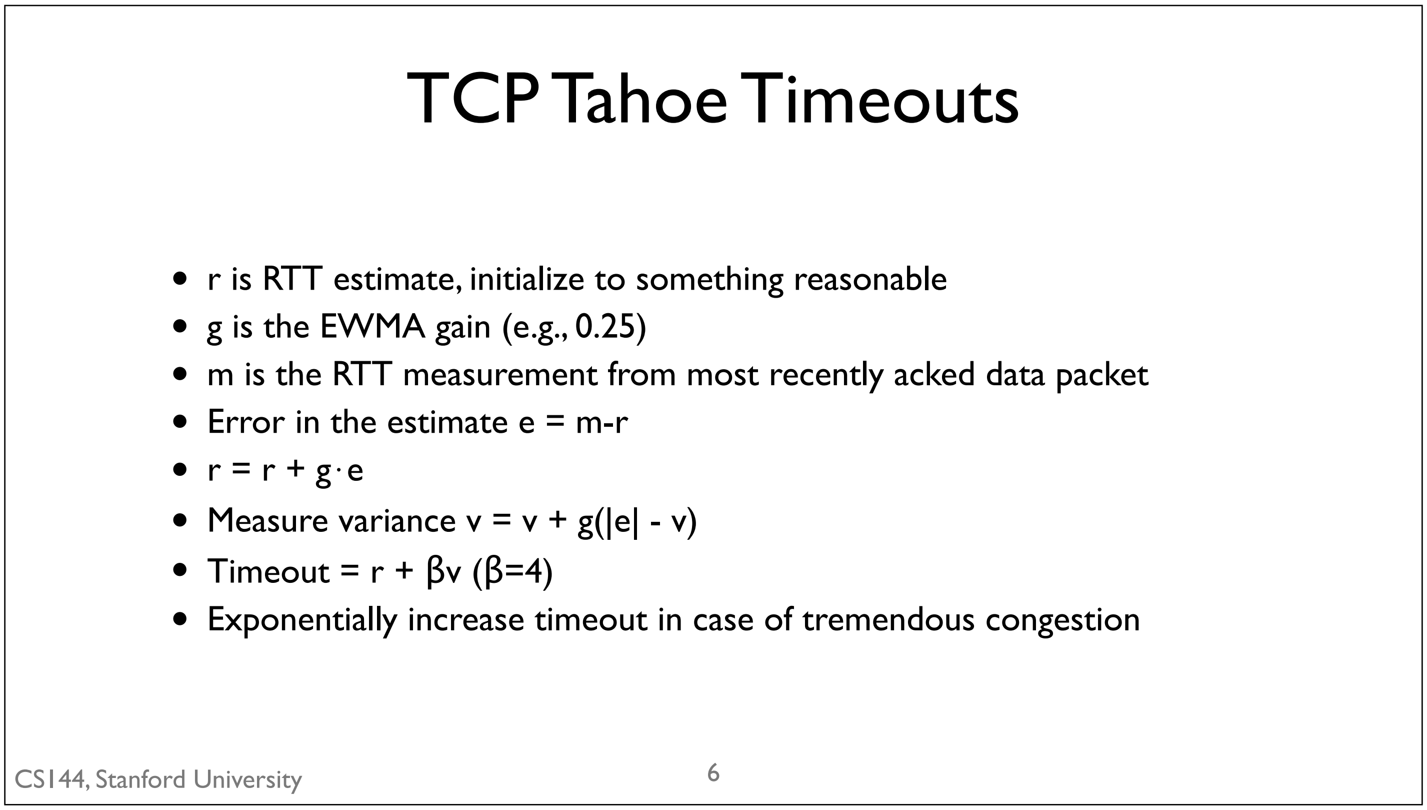
- r is RTT estimate, initialize to something reasonable
- m, RTT measurement from most recently acked data packet
- Exponentially weighted moving average: $r = \alpha r + (1-\alpha)m$
- Timeout = βr, $\beta=2$
- What's the problem?

Before TCP Tahoe, TCP kept track of a single variable r, its RTT estimate. Each time it received an new acknowledgement, it would calculate an RTT estimate m from the time between when the segment was sent and the acknowledgement was received. r was an exponentially weighted moving average of these measurements. If there was no acknowledgement for a segment after 2r, TCP assumed it was lost and triggers a retransmission.

So what's the problem with this approach?

The basic problem is that it assumes that the variance of the RTT measurements is a constant factor of its value. Imagine, for example, that you have a path with a high, low-variance delay. For example, you have an underutilized link across the ocean floor. Even if the RTT is 80ms, with 99.99% of RTTs being between 80 and 81ms, TCP will wait 160ms before triggering the retransmit timer. This is almost an entire wasted RTT.

Or imagine the opposite case, where the average RTT is 20ms but it has very high variance, such that RTTs sometimes as high as 80ms. Despite the fact that a significant fraction of packets have a high RTT, TCP will assume these packets are lost, shrink its congestion window to 1, and retransmit them.



TCP Tahoe Timeouts

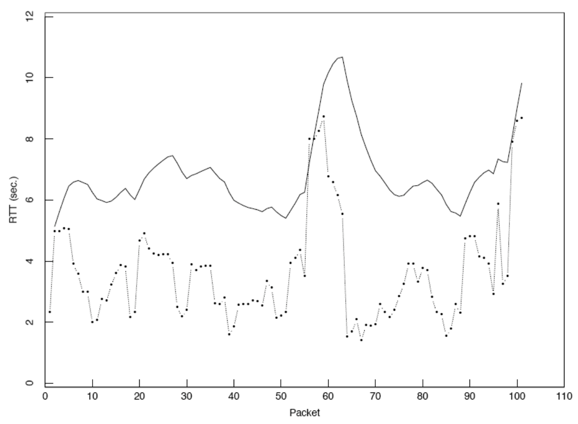
- r is RTT estimate, initialize to something reasonable
- g is the EWMA gain (e.g., 0.25)
- m is the RTT measurement from most recently acked data packet
- Error in the estimate e = m-r
- r = r + g·e
- Measure variance v = v + g(|e| - v)
- Timeout = r + βv (β=4)
- Exponentially increase timeout in case of tremendous congestion

So TCP Tahoe incorporated an estimate of the RTT variance in its retransmission timeout. It maintains an exponentially weighted moving average of the RTT estimates as before. It also measures how much the measurement differs from the estimate, that is, the error between the estimate and the most recent measurement. It applies an exponentially weighted average to this variance, v, as well. It calculates a timeout as the RTT estimate plus four times the variance. So if the connection has a very stable RTT, timeouts will be only slightly larger than the average RTT. But if there's a large variation in the RTT, TCP will choose a much larger timeout.

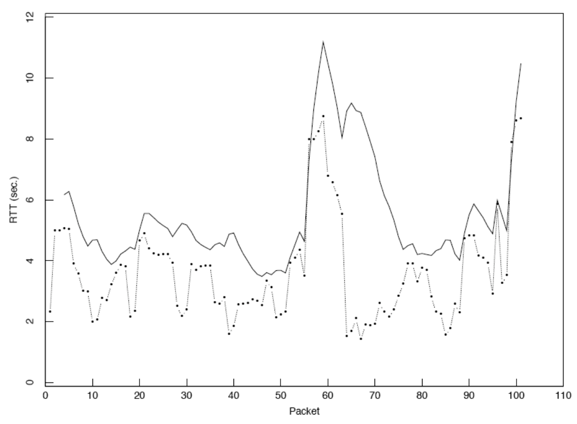
If a retransmission fails, that is, it isn't acknowledged, then retransmit again with an exponentially increasing timer. TCP assumes that this means there's tremendous congestion so continues its multiplicative decrease by increasing the interval between segments.

The two values, g and Beta, were selected after a bit of experimentation. This approach is reasonably robust to slight changes in them. There isn't any special magic behind them, they just tend to work well in practice.

RTT Estimation Improvement



Pre-Tahoe



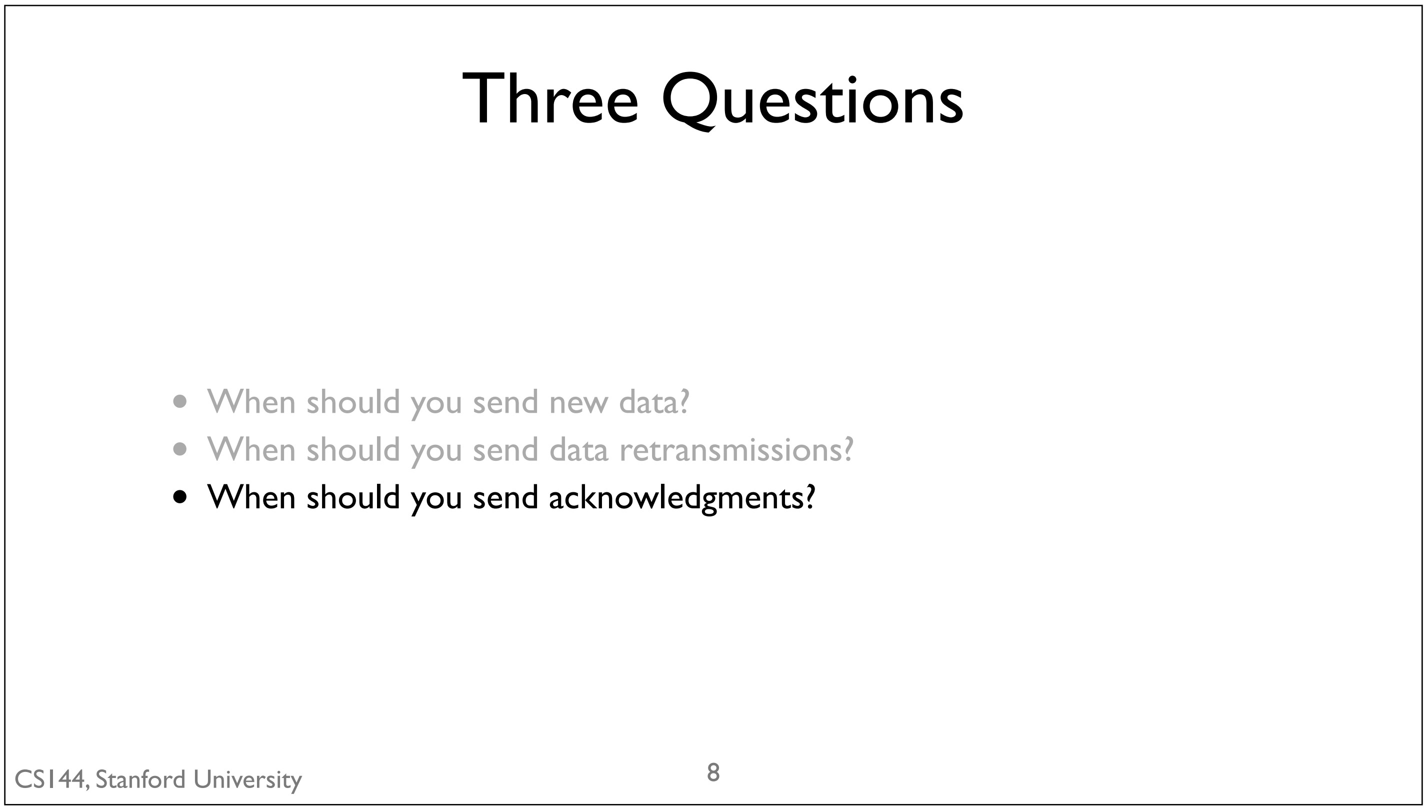
Tahoe

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

Here are two plots taken from the original TCP congestion control paper. The both show the RTT that TCP observes, the bottom, light line with data points, and the timeout estimate TCP maintains, the dark line on top. TCP pre-Tahoe was very conservative: the large gap between the measurements and the timeout value represents wasted time when TCP would take too long to retransmit. Also, around packet 60, when the RTT spikes upwards, pre-Tahoe would retransmit and enter slow start unnecessarily.

In contrast, RTT estimation for TCP Tahoe, shown on the right, much more closely tracks the RTT values. So now we've answered the second question: when does TCP retransmit data?

The lesson from this is that when one needs to estimate a retransmission or retry timer in a network protocol, consider not only the observed round trip time, but also its variance.



Three Questions

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

CS144, Stanford University

So now we’ve come to the third question: when should TCP send acknowledgements?

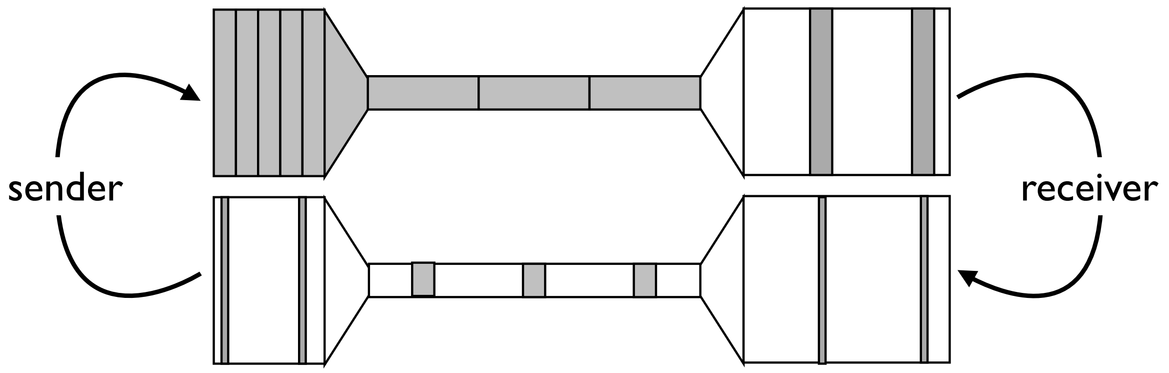
Three Improvements

- Congestion window
- Timeout estimation
- Self-clocking

It turns out the answer is generally “with as little delay as possible.” If TCP follows this policy, it leads to a very important and powerful behavior, called self-clocking.

Self-Clocking

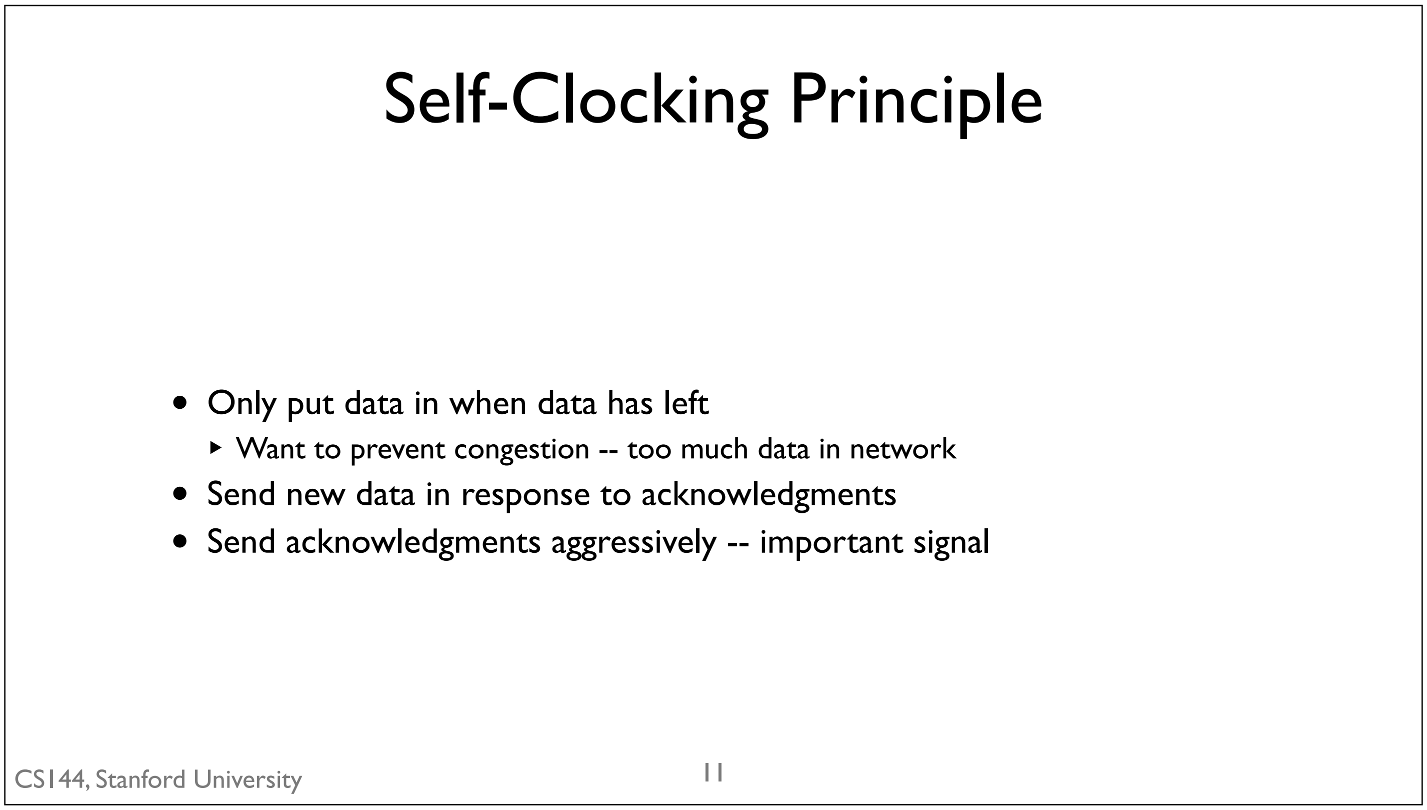
- In case of a bottleneck link, sender receives acks properly spaced in time



Self clocking means that if TCP sends acknowledgements aggressively, then it turns out they will space out in time according to the throughput of the bottleneck link. The sender will receive acknowledgements spaced out over time. Since the sender will send new data as soon as its sender window advances, this means that it will send segments at the rate that the bottleneck link can support.

This figure shows that behavior visually. The throughput of the network is shown as the width of the path, and time is shown as the length of the path in time. Packets sent on fat, high bandwidth links take a small amount of time. These same packets, sent over a lower bandwidth link, have a longer transmission and queueing delay and so take more time. When they emerge out of the bottleneck link, they will be fast again, but spaced out in time according to the rate at which they exit the bottleneck. Since acknowledgement packets are small, and assuming the bottleneck is not on the reverse path, this means that acknowledgements will arrive at the sender at approximately the rate at which they traverse the bottleneck link.

This policy has an additional benefit: TCP only puts new packets into the network when it receives an acknowledgement, that is, when one of its existing packets has left the network. From a congestion standpoint, this means TCP is keeping the number of outstanding packets, that is, its utilization of queues and capacity in the network, stable.



Self-Clocking Principle

- Only put data in when data has left
  - Want to prevent congestion -- too much data in network
- Send new data in response to acknowledgments
- Send acknowledgments aggressively -- important signal

So the self-clocking principle means that TCP sends acknowledgements aggressively, as soon as it receives data packets. This is to signal to the sender that data has left the network, as well as to give it RTT estimates and allow it to self-clock its transmissions.

So in summary, TCP Tahoe introduced three major mechanisms that make it effectively manage congestion and obtain good performance even in busy networks. The first is that it introduces a congestion window and maintains an AIMD-like state machine that transitions between the slow start and congestion avoidance states. This state machine controls how the congestion window updates. The second is that it calculates retransmission timers using both an exponentially weighted moving average as well as a variance, thereby reducing both false positives as well as false negatives. Finally, by sending acknowledgements aggressively, it self-clocks data transmissions to match the speed of the bottleneck link.