Lecture 20
TCP Congestion Control

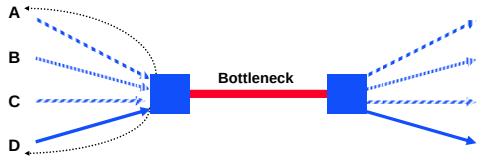
Peter Steenkiste
Department of Computer Science and
Electrical and Computer Engineering
Carnegie Mellon University

15-441 Networking, Spring 2006
http://www.cs.cmu.edu/~prs/15-441

Outline of the "Transport Lectures"

- Transport protocols introduction.
  - Functions, UDP
- Flow and error control.
  - Stop and go, go back N, sliding window, ...
- TCP.
  - Connections, flow control, error handling, extensions, ...
- Congestion control.
  - Congestion definition, congestion control strategies
- Congestion control in TCP.
  - More TCP, applied congestion control
- Other transports.
  - RPC, TCP conformance, multimedia

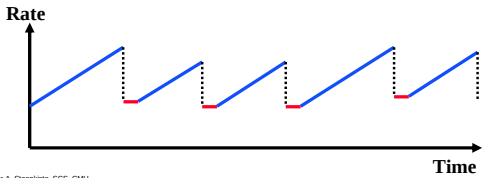
Summary: Closed Loop Congestion Control



- How does the network apply back pressure?
  - Dropped packets signal congestion
- How do the sources adapt?
  - Additive Increase Multiplicative Decrease (AIMD)
- How does the switch distribute link bandwidth?
  - FIFO queueing

TCP Congestion Control

- Packet loss is seen as sign of congestion and results in a multiplicative rate decrease
  - Factor of 2
- TCP periodically probes for available bandwidth by increasing its rate.



TCP Congestion Control Open Questions

- How can congestion control be implemented?
  - Operating system timers are very coarse – how do you accurately calculate the transmission rate?
- How does TCP know what is a good initial rate to start with?
  - Should work both for a CDPD (10s of Kbs or less) and for supercomputer links (2.4 Gbs and growing)
- Can we avoid the high overheads associated with the timeouts that detect packet loss?
  - Packet loss will be periodic
  - Timeouts are expensive!

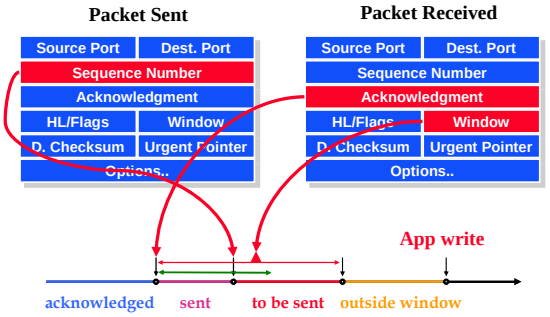
TCP Congestion Control Implementation

- Implemented using a congestion window that limits how much data can be in the network.
  - Rate = data in transit / roundtrip time
- Data can only be sent when the amount of outstanding data is less than the congestion window.
  - The amount of outstanding data is increased on a "send" and decreased on "ack" – packet conservation
  - (last sent – last acked) < congestion window

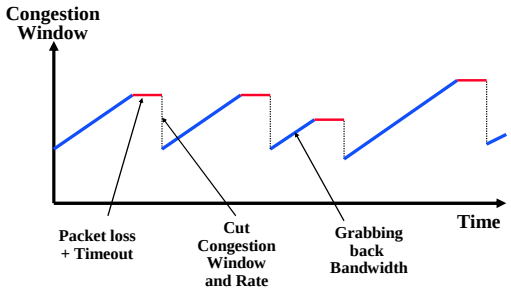
$$\text{Throughput} < \frac{\text{Cong Window Size}}{\text{Roundtrip Time}}$$

- Congestion window is similar to the end-end flow control window.
  - (last sent – last window update) < receiver window

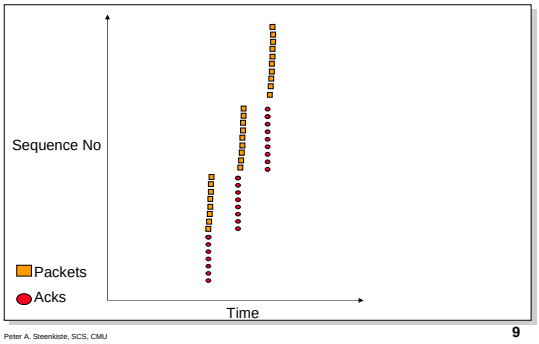
End-to-End Flow Control: View at the Sender



TCP Saw Tooth Behavior



Congestion Avoidance Sequence Plot



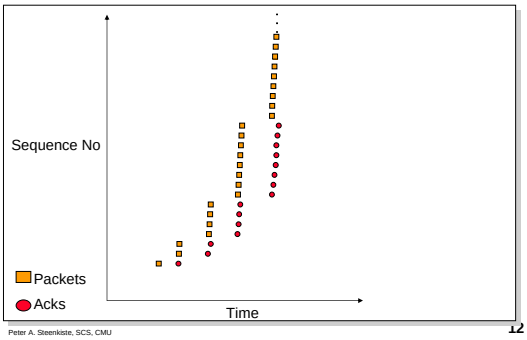
TCP Packet Pacing

- Congestion window helps to “pace” the transmission of data packets.
- In steady state, a packet is sent when an ack is received.
  - » Data transmission remains smooth, once it is smooth
  - » Self-clocking behavior

Slow start

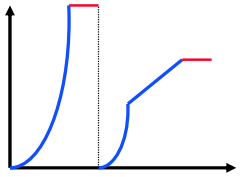
- At startup, no information is available about the congestion state of the network.
  - » Sending the full flow control window would flood the switches and result in severe packet loss
- Slow start quickly probes for a good window.
  - » Additional counter that limits number of packets in network
  - » Initialized to 1
  - » Incremented on each ack: results in exponential increase in traffic - not so slow at all!
- After packet losses and a timeout, TCP drops into (oscillating) steady state.
  - » During slow start, TCP remembers the congestion window size before losses occurred and uses that as an initial congestion window

Slow Start Sequence Plot

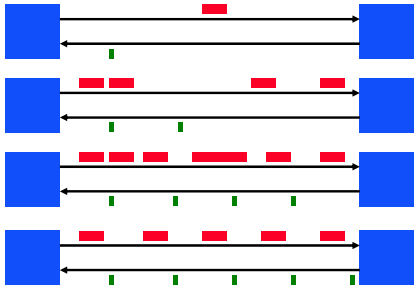


Slow Start also Starts Packet Pacing

- Slow start initiates the self-clocking behavior of TCP.
  - » Sender sends one packet
  - » Receives an ack and sends two packets
  - » Packets will be separated on the bottleneck link, resulting in separate acks
  - » Sender sends two packets in response to each ack, ...
- Slow start also used after timeout when pipeline has drained.
  - » but only up to old congestion window



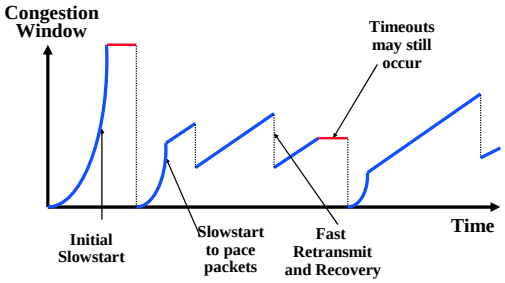
Starting of Packet Pacing



Fast Retransmit and Fast Recovery

- Timeouts are very expensive.
  - » Timeout itself is long
  - » Pipe drains - have to go through slow start again
  - » Congestion window is cut in half
- Lost packets can sometimes be detected without timeout by checking for duplicate acks.
  - » Receiver sends ack for every packet it receives
  - » A packet loss results in the same packet being acked again
- After 3 duplicate acks sender retransmits packet without waiting for a timeout.
  - » Fast retransmit: retransmit the missing packet immediately
  - » Fast recovery: skip slow start
    - – Transmission pipeline has not drained so there is no need to start up the self-clocking process
    - – But congestion window is still cut in half – why?

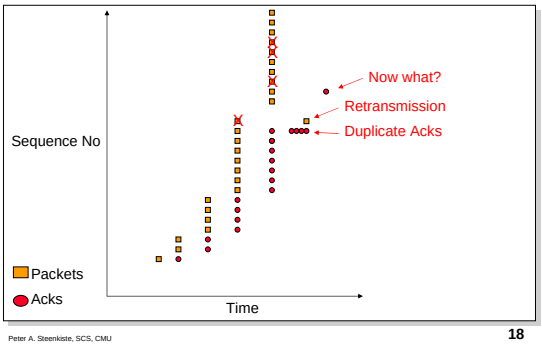
TCP Saw Tooth Behavior



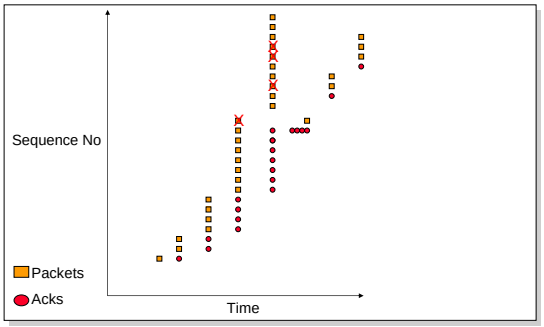
TCP Variants

- TCP Tahoe (distributed with 4.3BSD Unix)
  - » Original implementation of Van Jacobson's mechanisms
  - » Includes: Slow start, Congestion avoidance, Fast retransmit
- TCP Reno (1990): Tahoe + more
  - » Addition of fast-recovery
  - » Delayed acks
  - » Header prediction: inline common case
  - » With multiple losses, Reno typically timeouts because it does not see duplicate acknowledgements
- TCP NewReno: Reno + modified fast recovery
  - » Helps with "long fat pipes" – multiple packet losses per window are more likely
  - » Even better: selective acknowledgements

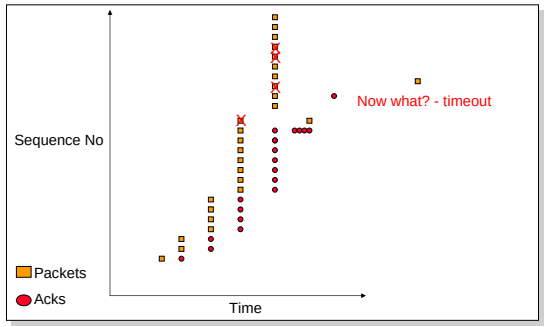
Multiple Losses



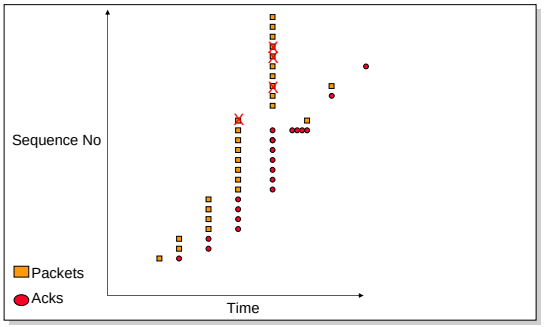
Tahoe



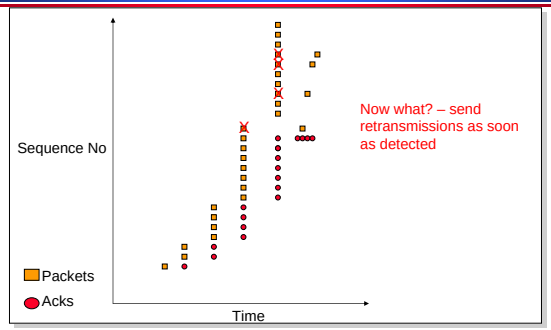
Reno



New Reno



SACK



When to Send a Packet?

- **End-to-end flow control.**
  - avoid buffer overflow on receiver
  - only send data if there is room in the receiver window
- **Congestion control.**
  - estimates amount of data that can be in the network
  - only send if data fits in the congestion window
  - during slow start: additional counter counts packets in the pipe
- **Efficiency considerations.**
  - only send data if there is sufficient data, or if there is no unacknowledged data
    - unless Nagle is disabled
  - piggybacking of acks (on the receiver)

Why Does TCP Work?

- **Cooperating users.**
  - As opposed to (bad) competing users
  - End-points that respond correctly to network feedback
  - No need for enforcement
  - But, what happens if a user does not cooperate?
- **Common end points.**
  - As opposed to diverse applications
  - Results in predictable response (i.e. as in TCP)
  - But, there is a fair bit of diversity in practice
- **Informal service definition.**
  - As opposed to precise service specifications demanded by paying customers
  - Does not require quantitative commitment (best effort!)
  - But, some users would prefer more predictable service

TCP Fairness Issues

- Multiple TCP flows sharing the same bottleneck link do not necessarily get the same bandwidth.
  - » Factors such as roundtrip time, small differences in timeouts, and start time, … affect how bandwidth is shared
  - » The bandwidth ratio typically does stabilize
- Modifying the congestion control implementation changes the aggressiveness of TCP and will change how much bandwidth a source gets.
  - » Affects "fairness" relative to other flows
  - » Changing timeouts, dropping or adding features, ..
- Users can grab more bandwidth by using parallel flows.
  - » Each flow gets a share of the bandwidth to the user gets more bandwidth than users who use only a single flow

TCP Performance Issues

- Consistently full queues can degrade TCP performance.
  - » Lock out some of the sessions
  - » Synchronization of TCP sessions due to the dropping of bursts of packets
  - » Increased queueing delay
- Not all users sharing a bottleneck link get the same bandwidth.
  - » Can be considered to be unfair
- Malicious users can grab more bandwidth by modifying TCP or by using UDP.
  - » Again a fairness issue

Random Early Detection (RED)

- Start randomly dropping packets before queue is full.
  - » Some flows will observe a single packet loss and slow down, hopefully avoiding queue overflow
  - » High bandwidth users are more likely to have a packet dropped than low bandwidth users
  - » Queue can still accommodate bursts of packets
- Improves overall network performance by avoiding that queues stay full.
  - » Congestion avoidance
  - » How do you set the thresholds?