15-441 Computer Networking

Lecture 18 – More TCP & Congestion Control

Overview

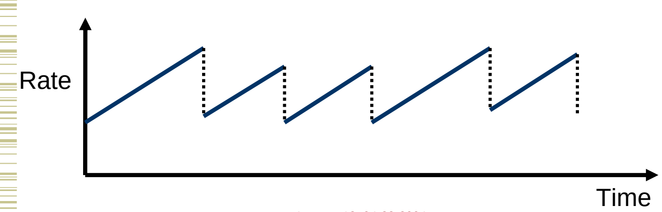
- TCP congestion control
- TCP modern loss recovery
- TCP modeling

TCP Congestion Control

- Changes to TCP motivated by ARPANET congestion collapse
- Basic principles
  - AIMD
  - Packet conservation
  - Reaching steady state quickly
  - ACK clocking

AIMD

- Distributed, fair and efficient
- 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



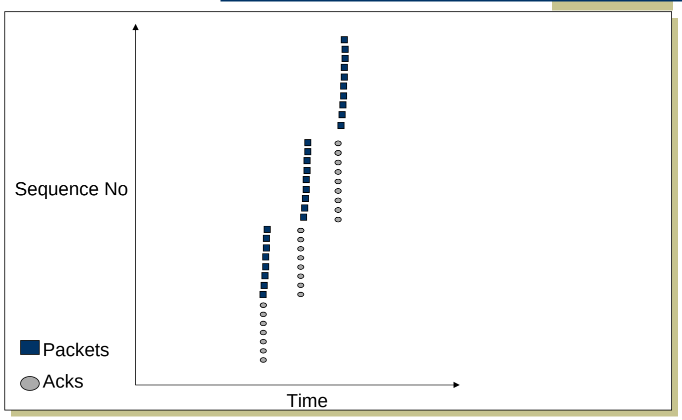
Implementation Issue

- Operating system timers are very coarse – how to pace packets out smoothly?
- Implemented using a congestion window that limits how much data can be in the network.
  - TCP also keeps track of how much data is in transit
- 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”
  - (last sent – last acked) < congestion window
- Window limited by both congestion and buffering
  - Sender’s maximum window = Min (advertised window, cwnd)

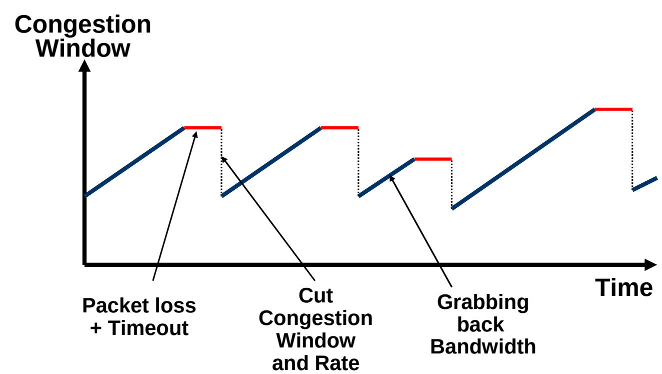
Congestion Avoidance

- If loss occurs when cwnd = W
  - Network can handle 0.5W ~ W segments
  - Set cwnd to 0.5W (multiplicative decrease)
- Upon receiving ACK
  - Increase cwnd by (1 packet)/cwnd
    - What is 1 packet? → 1 MSS worth of bytes
    - After cwnd packets have passed by → approximately increase of 1 MSS
- Implements AIMD

Congestion Avoidance Sequence Plot



Congestion Avoidance Behavior

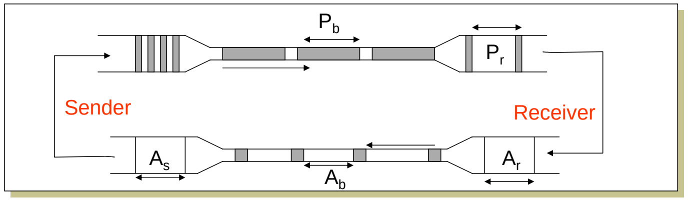


Packet Conservation

- At equilibrium, inject packet into network only when one is removed
  - Sliding window and not rate controlled
  - But still need to avoid sending burst of packets → would overflow links
    - Need to carefully pace out packets
    - Helps provide stability
- Need to eliminate spurious retransmissions
  - Accurate RTO estimation
  - Better loss recovery techniques (e.g. fast retransmit)

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

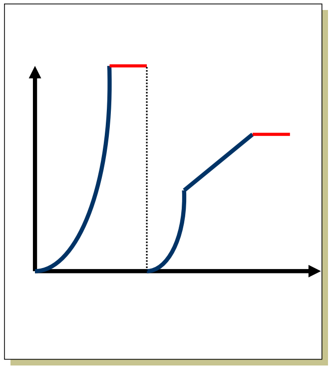


Reaching Steady State

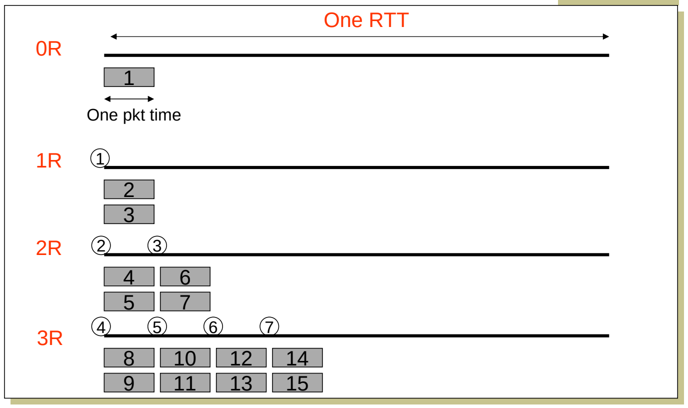
- Doing AIMD is fine in steady state but slow…
- How does TCP know what is a good initial rate to start with?
  - Should work both for a CDPD (10s of Kbps or less) and for supercomputer links (10 Gbps and growing)
- Quick initial phase to help get up to speed (slow start)

Slow Start Packet Pacing

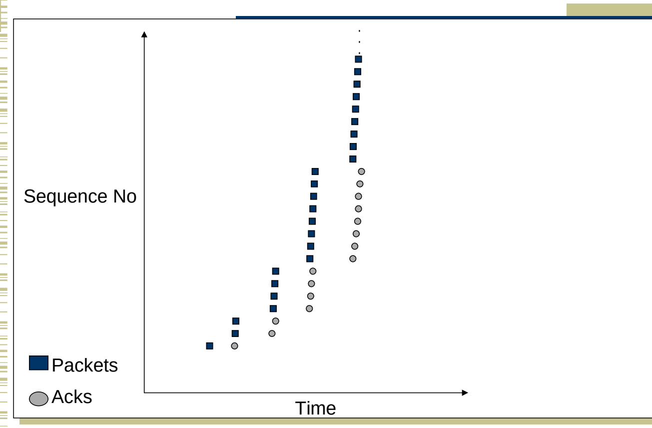
- How do we get this clocking behavior to start?
  - Initialize cwnd = 1
  - Upon receipt of every ack, cwnd = cwnd + 1
- Implications
  - Window actually increases to W in RTT * $\log_2(W)$
  - Can overshoot window and cause packet loss



Slow Start Example



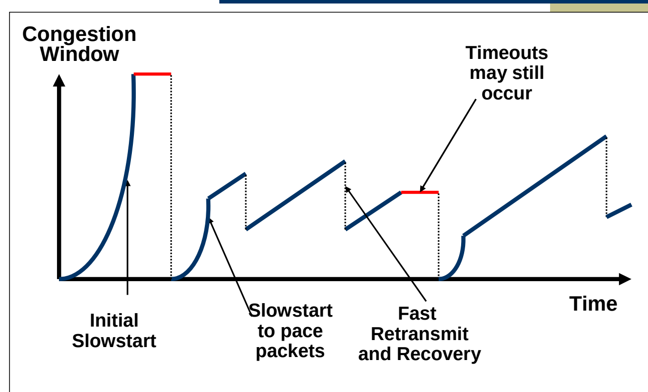
Slow Start Sequence Plot



Return to Slow Start

- If packet is lost we lose our self clocking as well
  - Need to implement slow-start and congestion avoidance together
- When timeout occurs set ssthresh to 0.5w
  - If cwnd < ssthresh, use slow start
  - Else use congestion avoidance

TCP Saw Tooth Behavior



Overview

- TCP congestion control
- TCP modern loss recovery
- TCP modeling

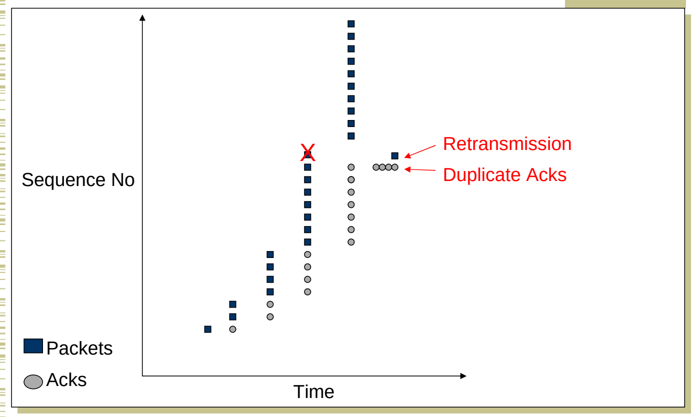
TCP Flavors

- Tahoe, Reno, Vegas
- TCP Tahoe (distributed with 4.3BSD Unix)
  - Original implementation of Van Jacobson's mechanisms (VJ paper)
  - Includes:
    - Slow start
    - Congestion avoidance
    - Fast retransmit

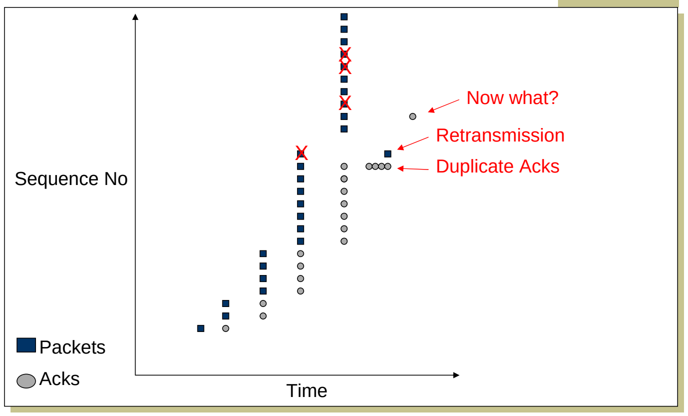
Fast Retransmit

- What are duplicate acks (dupacks)?
  - Repeated acks for the same sequence
- When can duplicate acks occur?
  - Loss
  - Packet re-ordering
  - Window update – advertisement of new flow control window
- Assume re-ordering is infrequent and not of large magnitude
  - Use receipt of 3 or more duplicate acks as indication of loss
  - Don't wait for timeout to retransmit packet

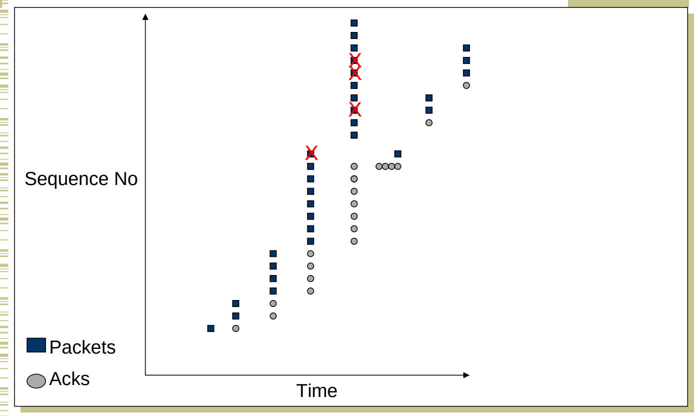
Fast Retransmit



Multiple Losses



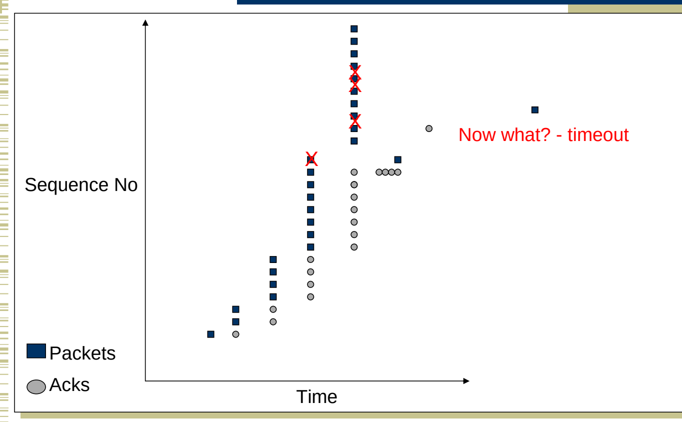
Tahoe



TCP Reno (1990)

- All mechanisms in Tahoe
- Addition of fast-recovery
  - Opening up congestion window after fast retransmit
- Delayed acks
- Header prediction
  - Implementation designed to improve performance
  - Has common case code inlined
- With multiple losses, Reno typically timeouts because it does not see duplicate acknowledgements

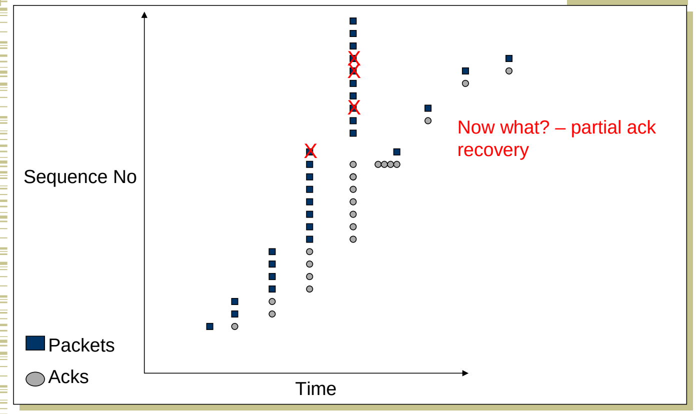
Reno



NewReno

- The ack that arrives after retransmission (partial ack) could indicate that a second loss occurred
- When does NewReno timeout?
  - When there are fewer than three dupacks for first loss
  - When partial ack is lost
- How fast does it recover losses?
  - One per RTT

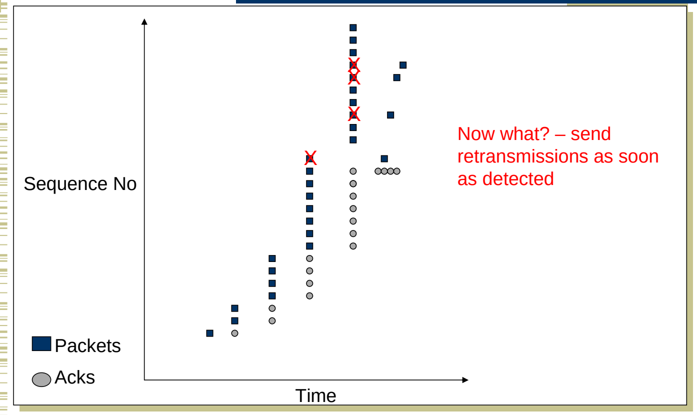
NewReno



SACK

- Basic problem is that cumulative acks provide little information
  - Ack for just the packet received
    - What if acks are lost? → carry cumulative also
    - This technique is not used
  - Bitmask of packets received
    - Selective acknowledgement (SACK)
- When to retransmit?
  - Still need to deal with reordering → wait for out of order by 3pkts

SACK



Performance Issues

- Timeout >> fast rexmit
  - Need 3 dupacks/sacks
  - Not great for small transfers
    - Don't have 3 packets outstanding
  - What are real loss patterns like?
- How to deal with reordering?

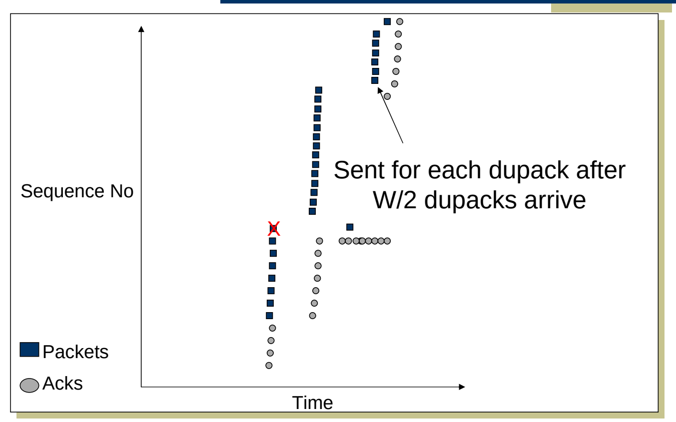
How to Change Window

- When a loss occurs have W packets outstanding
- New cwnd = 0.5 * cwnd
  - How to get to new state?

Fast Recovery

- Each duplicate ack notifies sender that single packet has cleared network
- When < cwnd packets are outstanding
  - Allow new packets out with each new duplicate acknowledgement
- Behavior
  - Sender is idle for some time – waiting for ½ cwnd worth of dupacks
  - Transmits at original rate after wait
    - Ack clocking rate is same as before loss

Fast Recovery



Overview

- TCP congestion control
- TCP modern loss recovery
- TCP modeling

TCP Performance

- Can TCP saturate a link?
- Congestion control
  - Increase utilization until… link becomes congested
  - React by decreasing window by 50%
  - Window is proportional to rate * RTT
- Doesn't this mean that the network oscillates between 50 and 100% utilization?
  - Average utilization = 75%??
  - No…this is *not* right!

TCP Performance

- In the real world, router queues play important role
  - Window is proportional to rate * RTT
    - But, RTT changes as well the window
  - Window to fill links = propagation RTT * bottleneck bandwidth
    - If window is larger, packets sit in queue on bottleneck link

TCP Performance

- If we have a large router queue → can get 100% utilization
  - But, router queues can cause large delays
- How big does the queue need to be?
  - Windows vary from W → W/2
    - Must make sure that link is always full
    - W/2 > RTT * BW
    - W = RTT * BW + Qsize
    - Therefore, Qsize > RTT * BW
  - Ensures 100% utilization
  - Delay?
    - Varies between RTT and 2 * RTT

Important Lessons

- How does TCP implement AIMD?
  - Sliding window, slow start & ack clocking
  - How to maintain ack clocking during loss recovery → fast recovery
- Modern TCP loss recovery
  - Why are timeouts bad?
  - How to avoid them? → fast retransmit, SACK
- How does TCP fully utilize a link?
  - Role of router buffers