CS640: Introduction to Computer Networks

Aditya Akella

Lecture 15
TCP – III
Reliability and Implementation Issues

Reliability

- TCP provides a "reliable byte stream"
  - "Loss recovery" key to ensuring this abstraction
  - Sender must retransmit lost packets
- Challenges:
  - When is a packet lost?
    - Congestion related losses
    - Reordering of packets
      - How to tell the difference between a delayed packet and lost one?
  - Variable packet delays
    - What should the timeout be?
  - How to recover from losses?

Loss Recovery in a Sliding Window setup

- Sliding window with cumulative acks
  - Receiver can only return a single "ack" sequence number to the sender.
  - Acknowledges all bytes with a lower sequence number
  - Starting point for retransmission
  - Duplicate acks sent when out-of-order packet received
- Sender only retransmits a single packet.
  - Only one that it knows is lost
    - Sent after timeout
- Choice of timeout interval → crucial

Round-trip Time Estimation

- Reception success known only after one RTT
  - Wait at least one RTT before retransmitting
- Importance of accurate RTT estimators:
  - Low RTT estimate
    - unneeded retransmissions
  - High RTT estimate
    - poor throughput
- RTT estimator must adapt to change in RTT
  - But not too fast, or too slow!

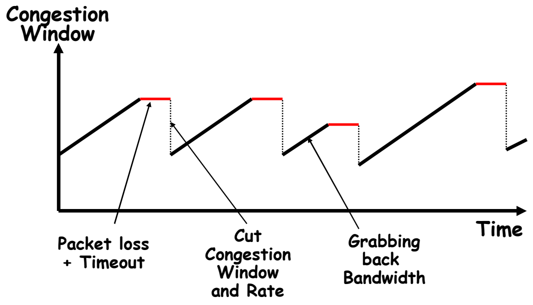
Jacobson's Retransmission Timeout (RTO)

- Original setting:
  - Round trip times exponentially averaged:
    - New RTT = α (old RTT) + (1 - α) (new sample)
    - Recommended value for α: 0.8 - 0.9
  - Retransmit timer set to (2 * RTT)
  - But this can lead to spurious retransmissions
- Key observation:
  - At high loads round trip variance is high
- Solution:
  - Base RTO on RTT and deviation
    - RTO = RTT + 4 * rttvar
  - new_rttvar = β * dev + (1- β) old_rttvar
    - Dev = linear deviation
    - Inappropriately named – actually smoothed linear deviation

AIMD Implementation

- If loss occurs when cwnd = W
  - Network can handle < W segments
  - Set cwnd to 0.5W (multiplicative decrease)
  - Known as "congestion control"
- Upon receiving ACK
  - Increase cwnd by (1 packet)/cwnd
    - What is 1 packet? → 1 MSS worth of bytes
    - MSS = maximum segment size
  - After cwnd packets have passed by → approximately increase of 1 MSS
  - Known as "congestion avoidance"
- Together these implement AIMD

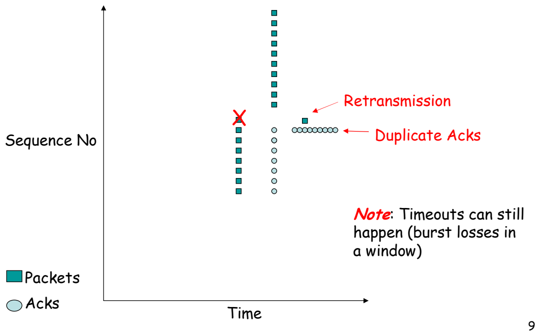
Control/Avoidance Behavior in the presence of timeouts



Improving Loss Recovery: Fast Retransmit

- Waiting for timeout to retransmit is inefficient
- Are there quicker recovery schemes?
  - Use duplicate acknowledgements as an indication
  - ***Fast retransmit***
- What are duplicate acks (dupacks)?
  - Repeated acks for the same sequence
- When can duplicate acks occur?
  - Loss
  - Packet re-ordering
- 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



Note: Timeouts can still happen (burst losses in a window)

How to Change Window

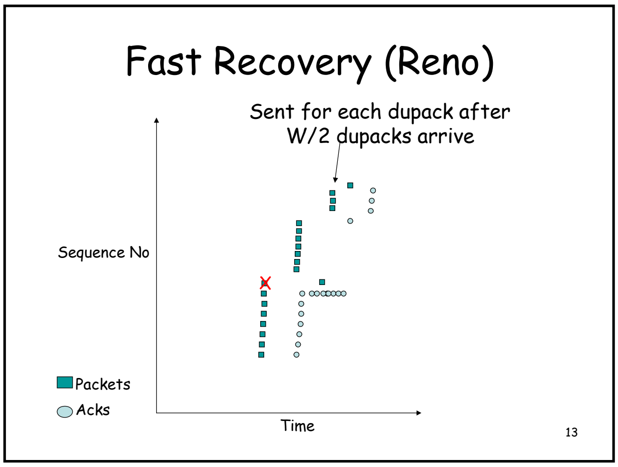
- When a loss occurs have W packets outstanding
  - A bunch of dupacks arrive
  - Rexmit on 3rd dupack
  - But dupacks keep arriving
  - Must wait for a new ack to send new packets
- New cwnd = 0.5 * cwnd
  - Send new cwnd packets in a burst when new ack arrives
  - Risk losing "self clocking" or "packet pacing"

Packet Pacing

- In steady state, a packet is sent when an ack is received
  - Data transmission remains smooth, once it is smooth (steady state)
  - "Self-clocking" behavior
  - When self clocking is lost → send packets in a burst → could momentarily overflow network capacity

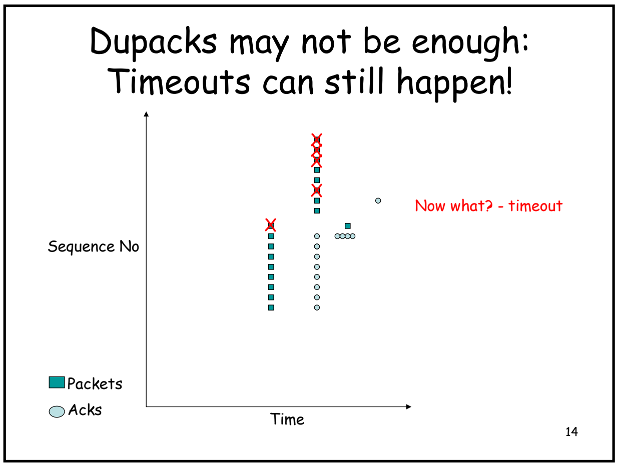
Preserving Clocking: Fast Recovery

- 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



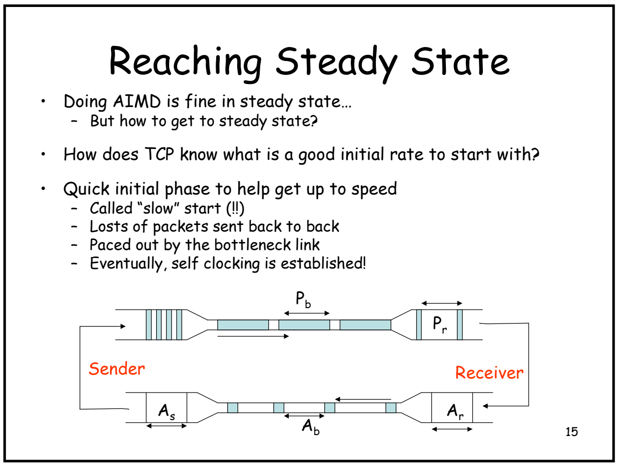
Fast Recovery (Reno)

Sent for each dupack after W/2 dupacks arrive



Dupacks may not be enough: Timeouts can still happen!

Now what? - timeout

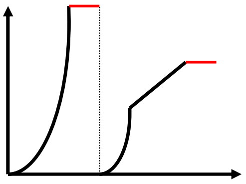


Reaching Steady State

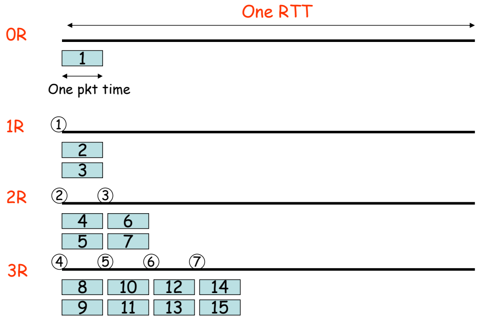
- Doing AIMD is fine in steady state…
  - But how to get to steady state?
- How does TCP know what is a good initial rate to start with?
- Quick initial phase to help get up to speed
  - Called "slow" start (!!)
  - Losts of packets sent back to back
  - Paced out by the bottleneck link
  - Eventually, self clocking is established!

Slow Start

- Slow 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



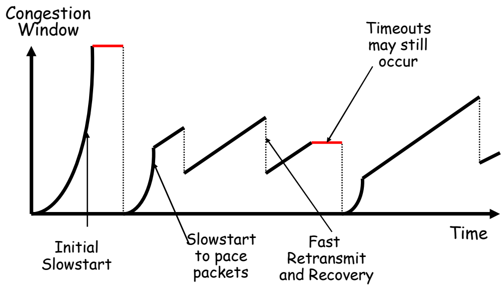
Slow Start Example



Return to Slow Start

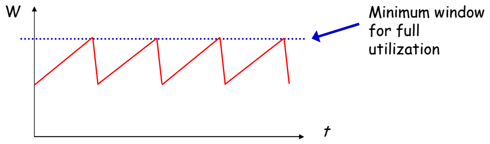
- If too many packets are lost self clocking is lost as well
  - Need to implement slow-start and congestion avoidance together
- When timeout occurs set ssthresh to 0.5w
  - If cwnd < ssthresh, use slow start
  - Else use congestion avoidance

The Whole TCP “Saw Tooth”



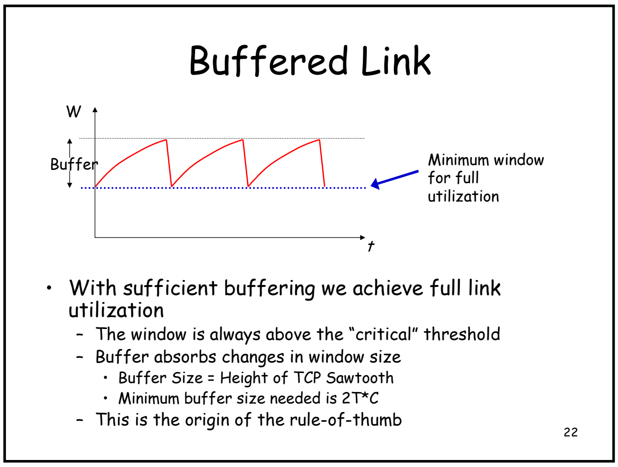
TCP Performance and Role of Buffering

- Can TCP saturate a link? Depends on buffering in network
- Congestion control
  - Increase utilization until… link becomes congested
  - React by decreasing window by 50%
  - Window is proportional to rate * RTT
- Unbuffered link
  - The router can’t fully utilize the link
    - If the window is too small, link is not full
    - If the link is full, next window increase causes drop
    - With no buffer TCP achieves 75% utilization



TCP Performance

- In the real world, router queues play important role
  - Role of Buffers → If window is larger, packets sit in queue on bottleneck link
- 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
    - To make sure that link is always full
W/2 > RTT * BW
    - W = RTT * BW + Qsize
➔ Qsize > RTT * BW
  - Ensures 100% utilization
  - Delay?
    - Varies between RTT and 2 * RTT
    - Queuing between 0 and RTT



Buffered Link

Minimum window for full utilization

- With sufficient buffering we achieve full link utilization
  - The window is always above the "critical" threshold
  - Buffer absorbs changes in window size
    - Buffer Size = Height of TCP Sawtooth
    - Minimum buffer size needed is 2T*C
  - This is the origin of the rule-of-thumb