CSE/EE 461 Lecture 16
TCP Congestion Control

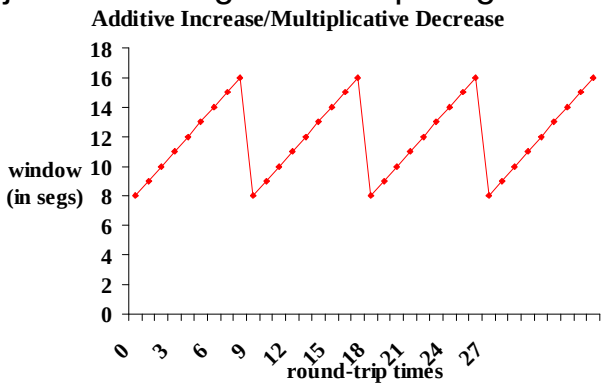
Tom Anderson
tom@cs.washington.edu
Peterson, Chapter 6

TCP Congestion Control

- Goal: efficiently and fairly allocate network bandwidth
  - Robust RTT estimation
  - Additive increase/multiplicative decrease
    - oscillate around bottleneck capacity
  - Slow start
    - quickly identify bottleneck capacity
  - Fast retransmit
  - Fast recovery

TCP “Sawtooth”

- Oscillates around bottleneck bandwidth
  - adjusts to changes in competing traffic

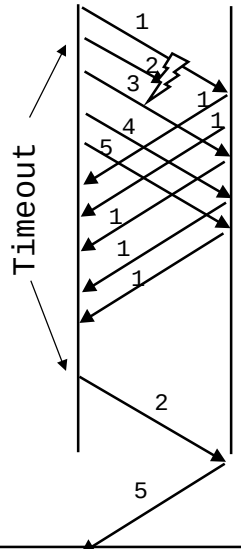


Slow start

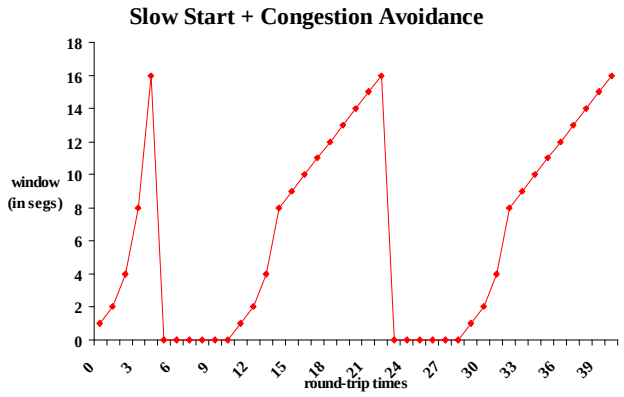
- How do we find bottleneck bandwidth?
  - Start by sending a single packet
    - start slow to avoid overwhelming network
  - Multiplicative increase until get packet loss
    - quickly find bottleneck
  - Remember previous max window size
    - shift into linear increase/multiplicative decrease when get close to previous max ~ bottleneck rate
    - called “congestion avoidance”

Ack Pacing After Timeout

- Packet loss causes timeout, disrupts ack pacing
  - slow start/additive increase are *designed* to cause packet loss
- After loss, use slow start to regain ack pacing
  - switch to linear increase at last successful rate
  - “congestion avoidance”



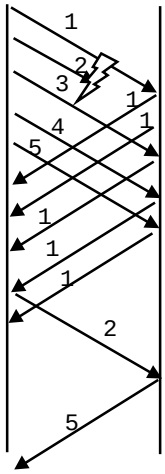
Putting It All Together



- Timeouts dominate performance!

Fast Retransmit

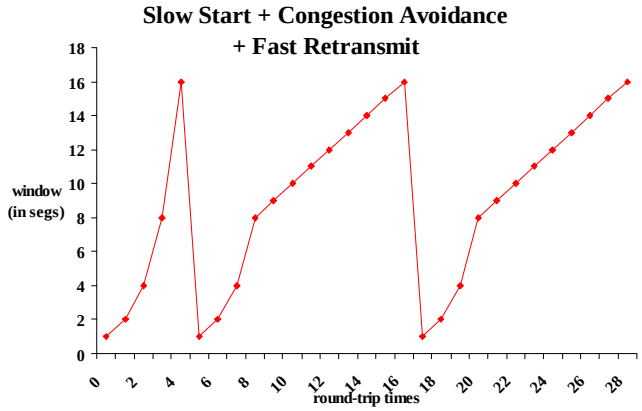
- Can we detect packet loss without a timeout?
  - Receiver will reply to each packet with an ack for last byte received in order
- Duplicate acks imply either
  - packet reordering (route change)
  - packet loss
- TCP Tahoe
  - resend if sender gets three duplicate acks, without waiting for timeout



Fast Retransmit Caveats

- Assumes in order packet delivery
  - Recent proposal: measure rate of out of order delivery; dynamically adjust number of dup acks needed for retransmit
- Doesn't work with small windows (e.g. modems)
  - what if window size <= 3
- Doesn't work if many packets are lost
  - example: at peak of slow start, might lose many packets

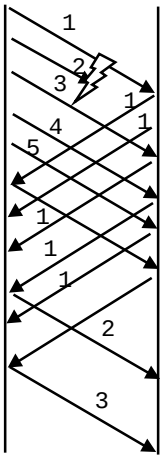
Fast Retransmit



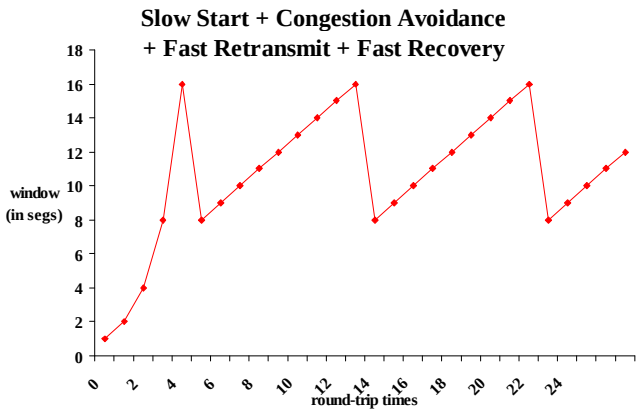
- Regaining ack pacing limits performance

Fast Recovery

- Use duplicate acks to maintain ack pacing
  - duplicate ack => packet left network
  - after loss, send packet after every other acknowledgement
- Doesn't work if lose many packets in a row
  - fall back on timeout and slow start to reestablish ack pacing



Fast Recovery



Delayed ACKS

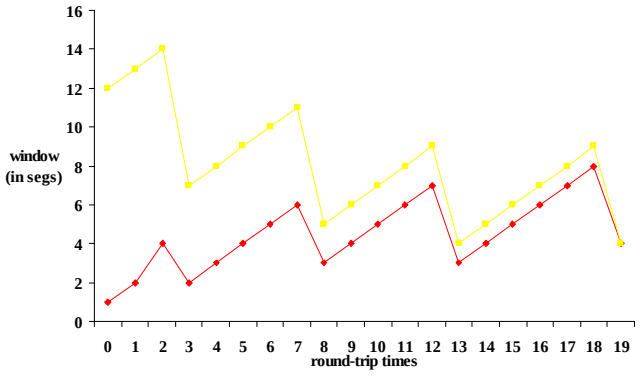
- Problem:
  - In request/response programs, server will send separate ACK and response packets
    - computing the response can take time
- TCP solution:
  - Don’t ACK data immediately
  - Wait 200ms (must be less than 500ms)
  - Must ACK every other packet
  - Must not delay duplicate ACKs

Delayed Ack Impact

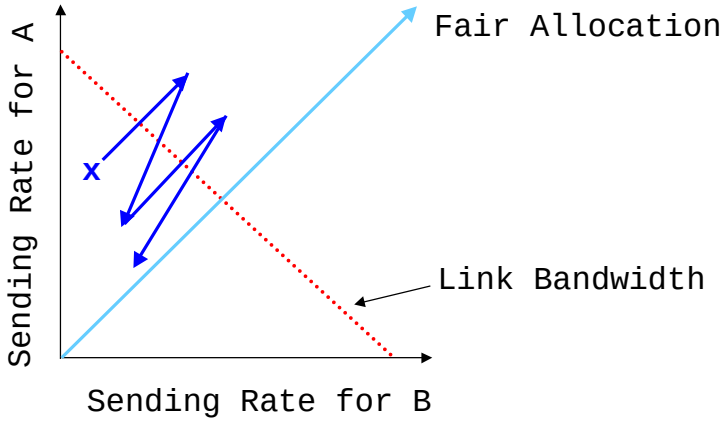
- TCP congestion control triggered by acks
  - if receive half as many acks => window grows half as fast
- Slow start with window = 1
  - ack will be delayed, even though sender is waiting for ack to expand window

What if two TCP connections share link?

- Reach equilibrium independent of initial bandwidth
  - assuming equal RTTs, “fair” drops at the router

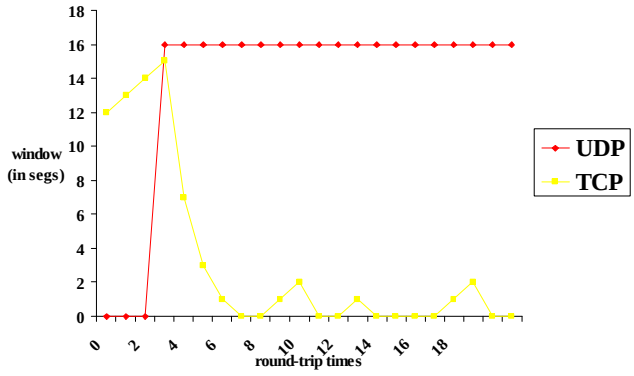


Equilibrium Proof



What if TCP and UDP share link?

- Independent of initial rates, UDP will get priority! TCP will take what's left.



What if two different TCP implementations share link?

- If cut back more slowly after drops => will grab bigger share
- If add more quickly after acks => will grab bigger share
- Incentive to cause congestion collapse!
  - Many TCP “accelerators”
  - Easy to improve perf at expense of network
- Solution: enforce good behavior at router

What if TCP connection is short?

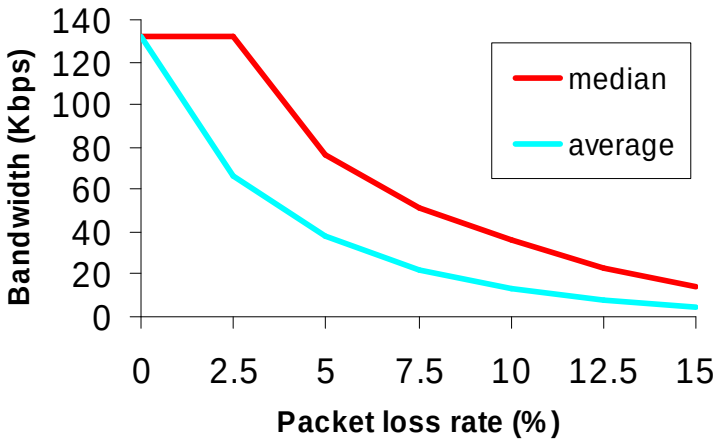
- Slow start dominates performance
  - What if network is unloaded?
  - Burstiness causes extra drops
- Packet losses unreliable indicator
  - can lose connection setup packet
  - can get drop when connection near done
  - signal unrelated to sending rate
- In limit, have to signal every connection
  - 50% loss rate as increase # of connections

Example: 10KB document

10Mb/s Ethernet,70ms RTT, 536 MSS

- Ethernet ~ 10 Mb/s
- 64KB window, 70ms RTT ~ 7.5 Mb/s
- can only use 10KB window ~ 1.2 Mb/s
- 5% drop rate ~ 275 Kb/s (steady state)
- model timeouts ~ 228 Kb/s
- slow start, no losses ~ 140 Kb/s
- slow start, with 5% drop ~ 75 Kb/s

Short flow bandwidth

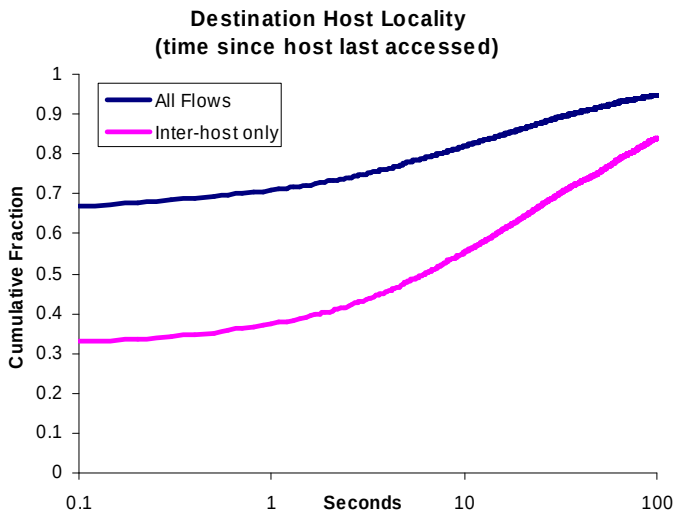


Flow length=10Kbytes, RTT=70ms

Improving Short Flow Performance

- Start with a larger initial window
  - Proposed standard to start with 4 packets
- Persistent connections
  - HTTP: reuse TCP connection for multiple objects on same page
  - Share congestion state between connections on same host
- Dynamic initial window
  - Share congestion state between different hosts
  - Within a large server farm or a large client population

Destination locality (UW)



Sharing Congestion Information