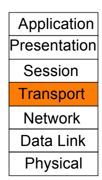
CSE/EE 461

TCP congestion control

Last Lecture

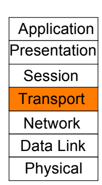
- Focus
  - How should senders pace themselves to avoid stressing the network?
- Topics
  - congestion collapse
  - congestion control



2

This Lecture

- Focus
  - Reaching and staying at equilibrium
- Topics
  - slow start (getting to equilibrium)
  - smarter retransmission timeouts (not violating equilibrium)
  - AIMD (restoring equilibrium after change)



3

Avoiding congestion collapse

- Achieve network stability by forcing the transport connection to obey a 'packet conservation' principle.
  - for connection in equilibrium (full window of data in transit), packet flow is conservative:

    a new packet is not put into the network until an old packet leaves.

4

Basic rules of TCP congestion control

1. The connection must reach equilibrium.
   - hurry up and stabilize!
   - when things get wobbly, put on the brakes and reconsider
2. Sender must not inject a new packet before an old packet has left
   - a packet leaves when the receiver picks it up,
   - or if it gets lost.
     - damaged in transit or dropped at congested point
     - (far fewer than 1% of packets get damaged in practice)
   - ACK or packet timeout signals that a packet has "exited."
     - ACK are easy to detect.
     - appropriate timeouts are harder…. all about estimating RTT.
3. Equilibrium is lost because of resource contention along the way.
   - new competing stream appears, must restabilize

5

1. The connection must reach equilibrium.

6

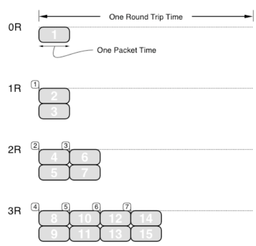
1. Getting to Equilibrium -- Slow Start

- Goal
  - Quickly determine the appropriate window size
- Strategy
  - Introduce *congestion_window (cwnd)*
  - When starting off, set cwnd to 1
  - For each ACK received, add 1 to cwnd
  - When sending, send the minimum of receiver's advertised window and cwnd
- Guaranteed to not transmit at more than twice the max BW, and for no more than RTT.
  - (bw delay product)

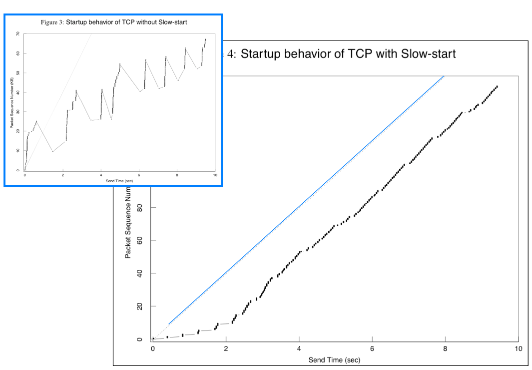
7

Cwnd doubles every RTT; Opening a window of size W takes time (RTT)$\log_2$W.

Figure 2: The Chronology of a Slow-start



The horizontal direction is time. The continuous time line has been chopped into one-round-trip-time pieces stacked vertically with increasing time going down the page. The grey, numbered boxes are packets. The white numbered boxes are the corresponding acks. As each ack arrives, two packets are generated: one for the ack (the ack says a packet has left the system so a new packet is added to take its place) and one because an ack opens the congestion window by one packet. It may be clear from the figure why an add-one-packet-to-window policy opens the window exponentially in time.



9

2. A sender must not inject a new packet before an old packet has exited.

10

2. Packet Injection. Estimating RTTs

- Do not inject a new packet until an old packet has left.
  - 1. ACK tells us that an old packet has left.
  - 2. Timeout expiration tells us as well.
    - *We must estimate RTT properly.*
- Strategy 1: pick some constant RTT.
  - simple, but probably wrong. (certainly not adaptive)
- Strategy 2: Estimate based on past behavior.

Tactic 0: Mean
Tactic 1: Mean with exponential decay
Tactic 2: Tactic 1 + *safety margin*
safety margin based on current estimate of error in Tactic 1

11

Estimating RTT

- for each packet, note time sent and time ACK received (RTT sample)
  - compute RTT samples and average recent samples for timeout

$$EstimatedRTT = (1-g)(EstimatedRTT) + g(SampleRTT)$$
$$0 \le g \le 1$$

  - this is an **exponentially-weighted moving average** (low pass filter) that smoothes the samples with a gain of *g*
    - big *g* can be jerky, but adapts quickly to change
    - small g can be smooth, but slow to respond
    - typically, g = .1 or .2, --> stable is better than precise
    - (lousy estimate right now does more damage than so-so estimate right now, followed by better one a little later)

12

Original TCP (RFC793) retransmission timeout algorithm

- Use EWMA to estimate RTT:

  $EstimatedRTT = (1-g)(EstimatedRTT) + g(SampleRTT)$
  $0 \le g \le 1$, *usually* $g = .1$ *or* $.2$

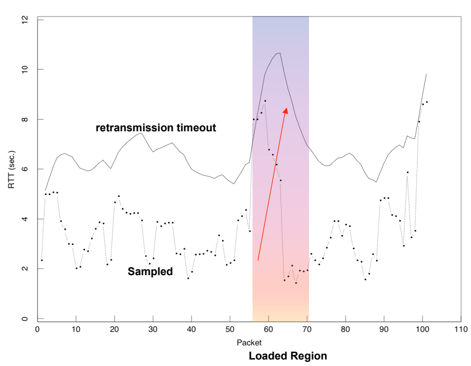
- Conservatively set timeout to small multiple (2x) of the estimate
  - in order to account for variance

  $Retransmission\ Timeout = 2 \times EstimatedRTT$

- Reason?
  - Better to wait “too long” than not long enough. (How come?)

13

Figure 5: Performance of an RFC793 retransmit timer



14

Bad Estimators and the Bad Things They Do

- Problem:
  - Variance in RTTs gets large as network gets loaded
  - So an average RTT isn’t a good predictor when we need it most
    - Time out too soon, unnecessarily drop another packet onto the network.
    - Timing out too soon occurs during load increase
      - if we time out when load increases but packet not yet lost, then we’ll inject another packet onto the network which will increase load, which will cause more timeouts, which will increase load, until we actually starting dropping packets!

15

Jacobson/Karels Algorithm

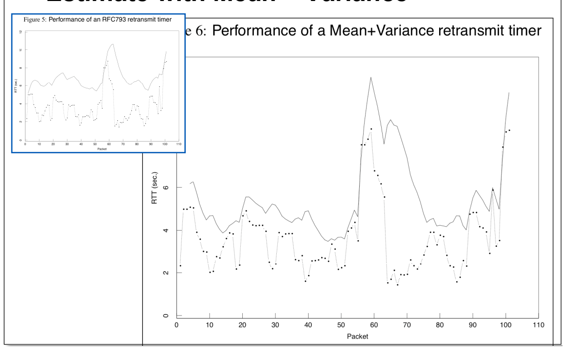
- EstimatedRTT + "safety margin"
  - large variation in EstimatedRTT --> larger safety margin
  - safety margin based on estimate of variance

1. Estimate how much SampledRTT deviates from EstimatedRTT
   - DevRTT = (1-b) * DevRTT + b * |SampledRTT - EstimatedRTT|
     - typically, b = .25
2. Set timeout interval as:
   - retransmission timeout = EstimatedRTT + k * DevRTT
   - k is generally set to 4

- timeout =~ EstimatedRTT when variance is low (estimate is good)
  - timeout quickly moves away from EstimatedRTT (4x!) when the variance is high (estimate is bad)

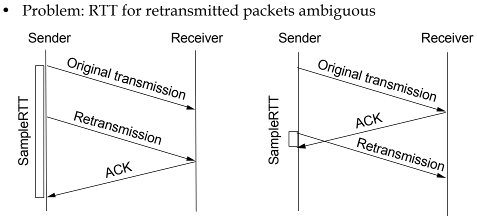
16

Estimate with Mean + Variance



Karn/Partridge Algorithm

- Problem: RTT for retransmitted packets ambiguous



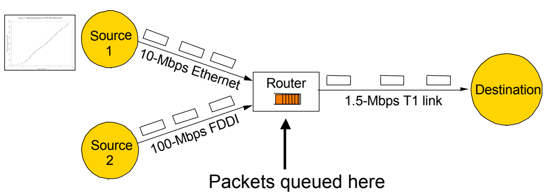
- Solution: Don't measure RTT for retransmitted packets and do not relax backed off timeout until valid RTT measurements

18

3. Equilibrium is lost because of resource contention along the way.

19

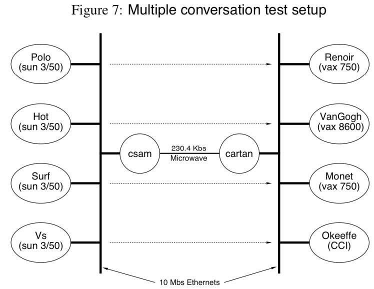
Congestion from Multiple Sources



Packets Lost Here

In Real Life

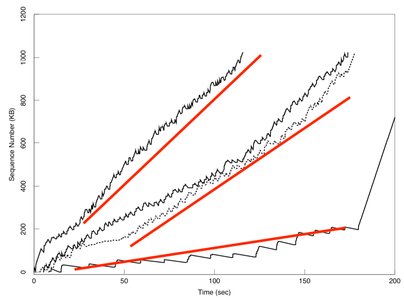
Figure 7: Multiple conversation test setup



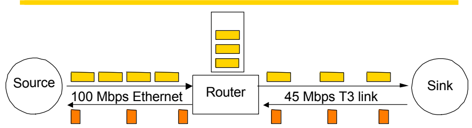
21

Four Simultaneous Streams

Figure 8: Multiple, simultaneous TCPs with no congestion avoidance



TCP is “Self-Clocking”



- Neat observation: acks pace transmissions at approximately the botteneck rate
- So just by sending packets we can discern the “right” sending rate (called the packet-pair technique)

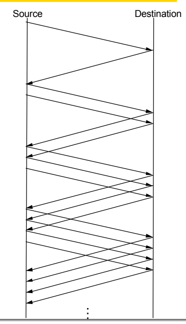
Congestion Control Relies on Signals from the Network

- The network is not saturated: *Send even more*
- The network is saturated: *Send less*

- *ACK* signals that the network is not saturated.
- A Lost packet (no ACK) signals that the network is saturated
  - Assumption here??
- Leads to a simple strategy:
  - On each ack, increase *congestion window* (***additive increase)***
  - On each lost packet, decrease congestion window (***multiplicative decrease)***
- Why increase slowly and decrease quickly?
  - *Respond to good news conservatively, but bad news aggressively*

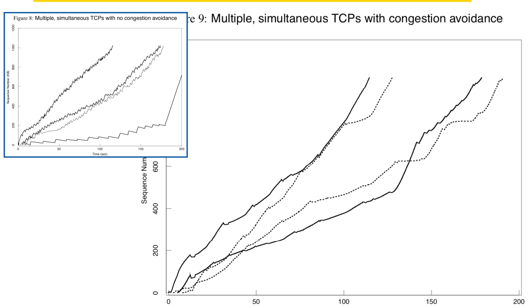
AIMD (Additive Increase/Multiplicative Decrease)

- How to adjust probe rate?
- Increase slowly while we believe there is bandwidth
  - Additive increase per RTT
  - Cwnd += 1 packet / RTT
- Decrease quickly when there is loss (went too far!)
  - Multiplicative decrease
  - Cwnd /= 2

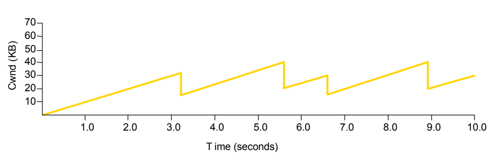


25

With Additive Increase/Multiplicative Decrease



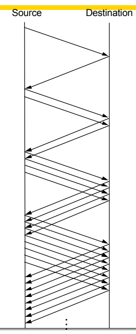
TCP Sawtooth Pattern



27

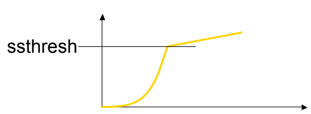
Comparing to "Slow Start"

- Q: What is the ideal value of cwnd? How long will AIMD take to get there?
- Use a different strategy to get close to ideal value
  - Slow start:
    - Double cwnd every RTT
      - cwnd *= 2 per RTT
      - i.e., cwnd += 1 per ACK
  - AIMD:
    - add one to cwnd per RTT
      - cwnd +=1 per RTT
      - i.e., cwnd += (1/cwnd) per ACK



28

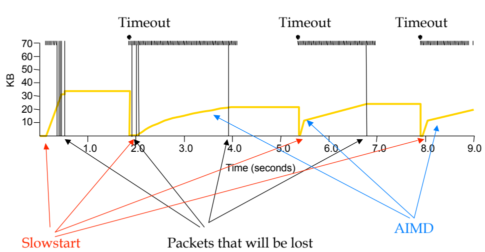
Combining Slow Start and AIMD



- Slow start is used whenever the connection is not running with packets outstanding
  - initially, and after timeouts indicating that there's no data on the wire
- But we don't want to overshoot our ideal cwnd on next slow start, so remember the last cwnd that worked with no loss
  - ssthresh = cwnd after cwnd /= 2 on loss
  - switch to AIMD once cwnd passes ssthresh

29

Example (Slow Start +AIMD)



30

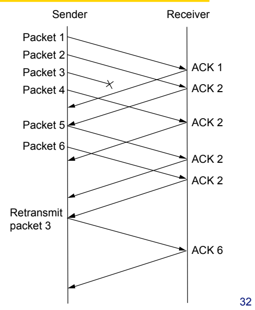
The Long Timeout Problem

- Would like to "signal" a lost packet earlier than timeout
  - enable retransmit sooner
- Can we infer that a packet has been lost?
  - Receiver receives an "out of order packet"
  - Good indicator that the one(s) before have been misplaced
- Receiver generates a duplicate ack on receipt of a misordered packet
- Sender interprets sequence of duplicate acks as a signal that the as-yet-unacked packet has not arrived

31

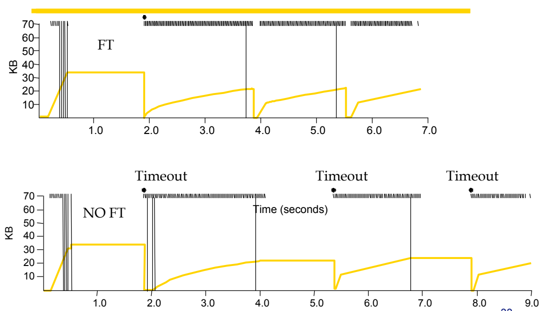
Fast Retransmit

- TCP uses cumulative acks, so duplicate acks start arriving after a packet is lost.
- We can use this fact to infer which packet was lost, instead of waiting for a timeout.
- 3 duplicate acks are used in practice



32

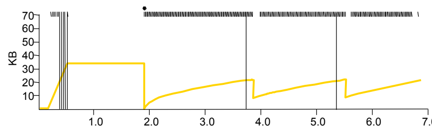
Example (with Fast Retransmit)



33

Fast Recovery

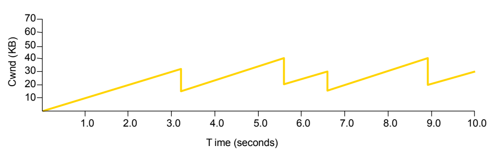
- After Fast Retransmit, use further duplicate acks to grow cwnd and clock out new packets, since these acks represent packets that have left the network.
- End result: Can achieve AIMD when there are single packet losses. Only slow start the first time and on a real timeout.



34

Example (with Fast Recovery)

(Not the same trace as before)



The Familiar Saw Tooth Pattern

35

Key Concepts

- Packet conservation is a fundamental concept in TCP's congestion management
  - Get to equilibrium
    - *Slow Start*
  - Do nothing to get out of equilibrium
    - *Good RTT Estimate*
  - Adapt when equilibrium has been lost due to other's attempts to get to/stay in equilibrium
    - *Additive Increase/Multiplicative Decrease*
- The Network Reveals Its Own Behavior

36

Key Concepts (next level down)

- TCP probes the network for bandwidth, assuming that loss signals congestion
- The congestion window is managed to be additive increase / multiplicative decrease
  - It took fast retransmit and fast recovery to get there
- Slow start is used to avoid lengthy initial delays
  - Ramp up to near target rate and then switch to AIMD