EE 122: Advanced TCP

Ion Stoica (and Brighten Godfrey)

TAs: Lucian Popa, David Zats and Ganesh Ananthanarayanan

http://inst.eecs.berkeley.edu/~ee122/

(Materials with thanks to Vern Paxson, Jennifer Rexford, and colleagues at UC Berkeley)

1

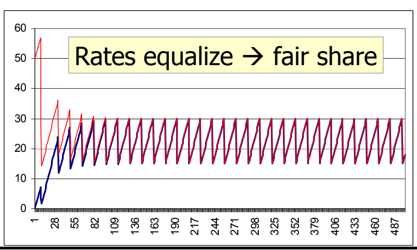
Goals of Today's Lecture

- Understanding AIMD/AIAD/MIAD/MIMD dynamics
- Improved TCP algorithms
- TCP Throughput Computation

2

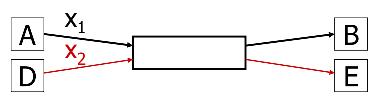
AIMD Sharing Dynamics

- No congestion → rate increases by one packet/RTT every RTT
- Congestion → decrease rate by factor 2

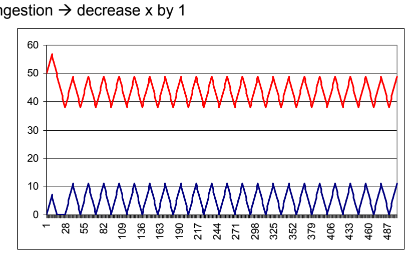


3

AIAD Sharing Dynamics



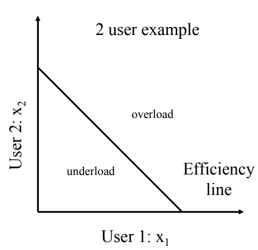
- No congestion → x increases by one packet/RTT every RTT
- Congestion → decrease x by 1



4

Efficient Allocation: Challenges of Congestion Control

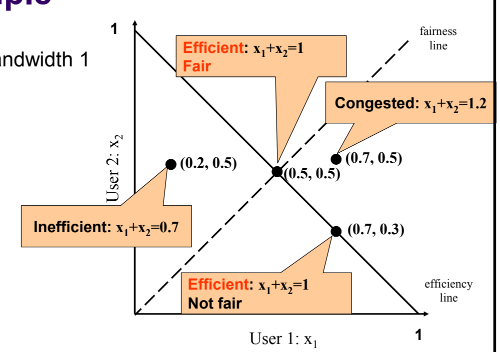
- Too slow
  - Fail to take advantage of available bandwidth → underload
- Too fast
  - Overshoot knee → overload, high delay, loss
- Everyone's doing it
  - May all under/over shoot → large oscillations
- Optimal:
  - $\Sigma x_i = X_{goal}$
- Efficiency = 1 - distance from efficiency line



5

Example

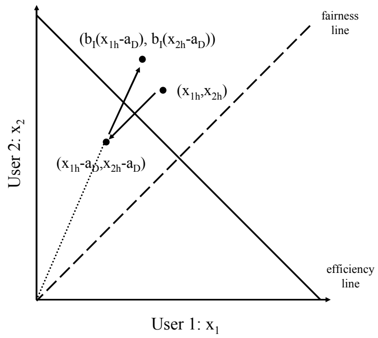
- Total bandwidth 1



6

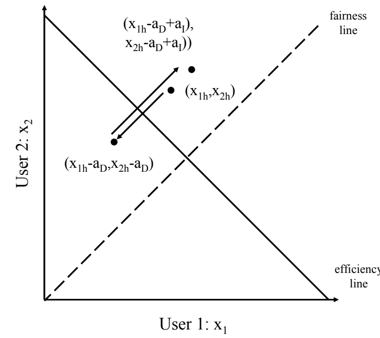
MIAD

- Increase: $x*b_I$
- Decrease: $x - a_D$
- Does not converge to fairness
- Does not converges to efficiency



AIAD

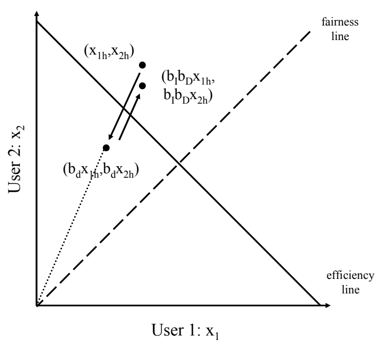
- Increase: $x + a_I$
- Decrease: $x - a_D$
- Does not converge to fairness
- Does not converge to efficiency



MIMD

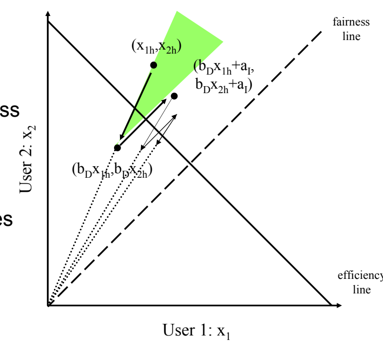
- Increase: $x*b_I$
- Decrease: $x*b_D$
- Does not converge to fairness
- Converges to efficiency iff

$$b_I \geq 1$$
$$0 \leq b_D < 1$$



AIMD

- Increase: $x+a_D$
- Decrease: $x*b_D$
- Converges to fairness
- Converges to efficiency
- Increments smaller as fairness increases



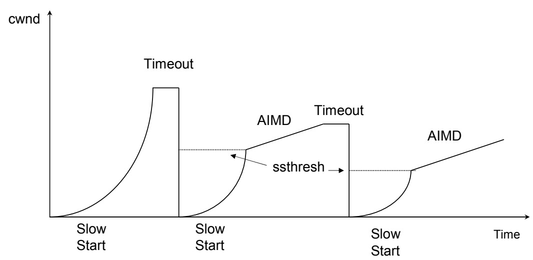
Implementing AIMD

- After each ACK
  - Increment *cwnd* by 1/*cwnd* (*cwnd* += 1/*cwnd*)
  - As a result, *cwnd* is increased by one only if all segments in a *cwnd* have been acknowledged
- But need to decide when to leave slow-start and enter AIMD
  - Use ssthresh variable

Slow Start/AIMD Pseudocode

```
Initially:
    cwnd = 1;
    ssthresh = infinite;
New ack received:
    if (cwnd < ssthresh)
         /* Slow Start*/
         cwnd = cwnd + 1;
    else
         /* Congestion Avoidance */
         cwnd = cwnd + 1/cwnd;
Timeout:
    /* Multiplicative decrease */
    ssthresh = cwnd/2;
    cwnd = 1;
```

The big picture (with timeouts)



5 Minute Break

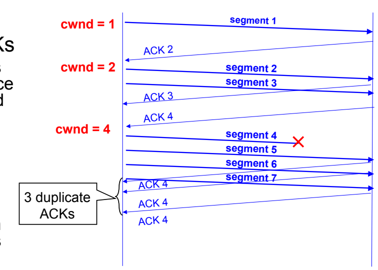
Questions Before We Proceed?

Congestion Detection Revisited

- Wait for Retransmission Time Out (RTO)
  - RTO kills throughput
- In BSD TCP implementations, RTO is usually more than 500ms
  - The granularity of RTT estimate is 500 ms
  - Retransmission timeout is RTT + 4 * mean_deviation
- Solution: Don't wait for RTO to expire

Fast Retransmits

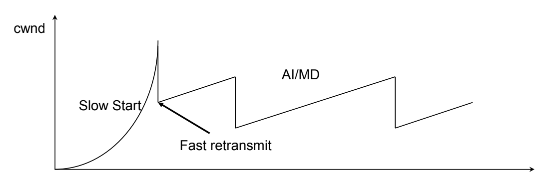
- Resend a segment after 3 duplicate ACKs
  - Duplicate ACK means that an out-of sequence segment was received
- Notes:
  - ACKs are for next expected packet
  - Packet reordering can cause duplicate ACKs
  - Window may be too small to get enough duplicate ACKs



Fast Recovery: After a Fast Retransmit

- *ssthresh* = *cwnd* / 2
- *cwnd* = *ssthresh*
  - Instead of setting *cwnd* to 1, cut cwnd in half (multiplicative decrease)
- For each dup ack arrival
  - *dupack*++
  - Indicates packet left network, so we may be able to send more
  - MaxWindow = min(*cwnd* + *dupack*, AdvWin)
- Receive ack for new data (beyond initial dup ack)
  - *dupack* = 0
  - Exit fast recovery
- But when RTO expires still do *cwnd* = 1

Fast Retransmit and Fast Recovery



- Retransmit after 3 duplicated acks
  - Prevent expensive timeouts
- Reduce slow starts
- At steady state, *cwnd* oscillates around the optimal window size

TCP Congestion Control Summary

- Measure available bandwidth
  - Slow start: fast, hard on network
  - AIMD: slow, gentle on network
- Detecting congestion
  - Timeout based on RTT
    - Robust, causes low throughput
  - Fast Retransmit: avoids timeouts when few packets lost
    - Can be fooled, maintains high throughput
- Recovering from loss
  - Fast recovery: don't set cwnd=1 with fast retransmits

TCP Flavors

- TCP-Tahoe
  - cwnd =1 whenever drop is detected
- TCP-Reno
  - cwnd =1 on timeout
  - cwnd = cwnd/2 on dupack
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK

TCP-SACK

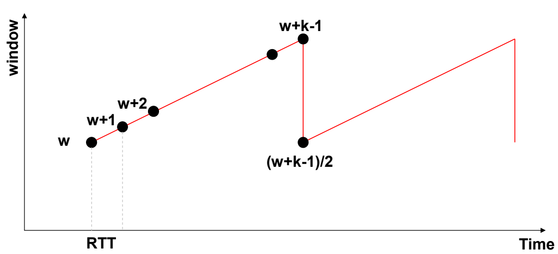
- SACK = Selective Acknowledgements
- ACK packets identify exactly which packets have arrived
- Makes recovery from multiple losses much easier

Standards?

- How can all these algorithms coexist?
- Don't we need a single, uniform standard?
- What happens if I'm using Reno and you are using Tahoe, and we try to communicate?

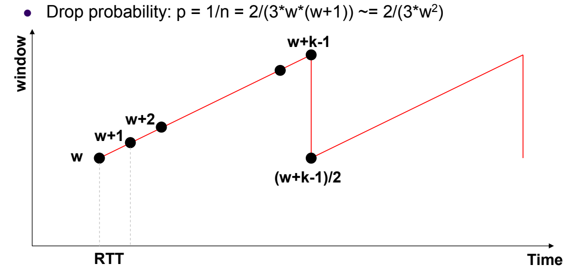
TCP Throughput

- Assume a drop every k'th RTT (for some large k)
- w, w+1, w+2, ...w+k-1 DROP (w+k-1)/2, (w+k-1)/2+1,...



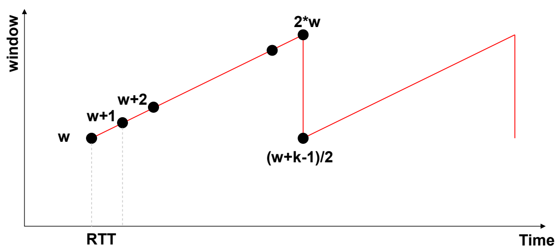
TCP Throughput (cont'd)

- In steady state: w= (w+k-1)/2 → w=k-1
- Average window: (w + w + k -1)/2 = 3*w/2
- Total packets between drops: n = w+(w+1)+… +2*w = 3*w*(w+1)/2
- Drop probability: p = 1/n = 2/(3*w*(w+1)) ~= 2/(3*w^2)



TCP Throughput (cont'd)

- Throughput = average_window/RTT = (3*w/2)/RTT
- Drop probability: p ~= 2/(3*w^2) → w = sqrt(2/3p)
- Throughput ~= (1/RTT)*sqrt(3/2p)



25

Equation-Based CC

- Idea:
  - Forget complicated increase/decrease algorithms
  - Use this equation T(p) directly!
- Approach:
  - Measure drop rate (don't need ACKs for this)
  - Send drop rate p to source
  - Source sends at rate T(p)
- Good for streaming audio/video that can't tolerate the high variability of TCP's sending rate

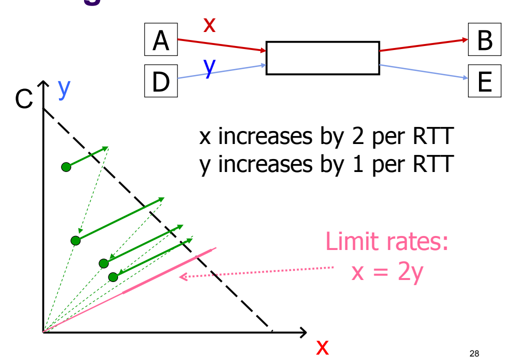
26

Cheating

- Three main ways to cheat:
  - Increasing cwnd faster than 1 per RTT
  - Using large initial cwnd
  - Opening many connections

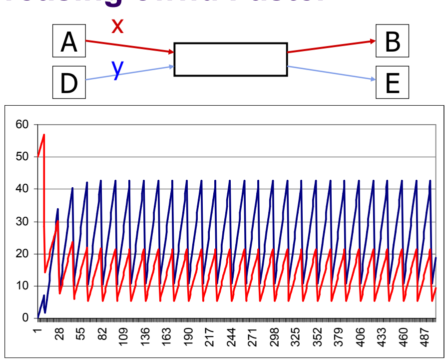
27

Increasing cwnd Faster



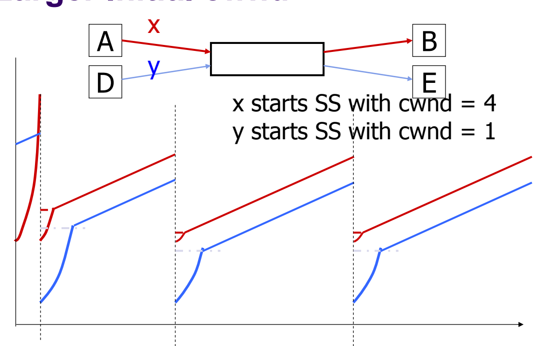
28

Increasing cwnd Faster



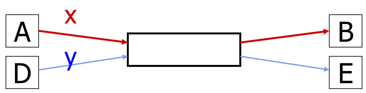
29

Larger Initial cwnd



30

Open Many Connections



Assume

- A starts 10 connections to B
- D starts 1 connection to E
- Each connection gets about the same throughput

Then A gets 10 times more throughput than D

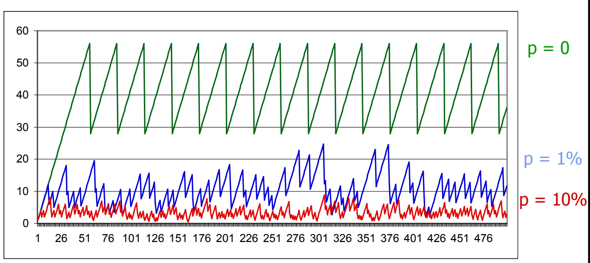
31

Lossy Links

- TCP assumes that all losses are due to congestion
- What happens when the link is lossy?
- Recall that Tput ~ 1/sqrt(p) where p is loss prob.
- This applies even for non-congestion losses

32

Example



33

Summary

- Congestion control critical for avoiding collapse
  - **AIMD**: Additive Increase, Multiplicative Decrease
  - Congestion detected via packet loss (fail-safe)
  - Slow start to find initial sending rate & to restart after timeout
- Spectrum of TCP mechanisms to improve TCP performance
  - Fast Retransmit (avoid RTO stall)
  - Fast Recovery (full AIMD)

34