Lecture 6: Congestion Control

Kyle Jamieson

COS 461: Computer Networks

Today

Lecture 5
- Fundamentals, Data transmission
- Connection establishment

- **Pacing Transmissions**
- Slow Start and Self-clocking
- Congestion control
- Learning to Share: Chiu-Jain phase plots
- Modeling Throughput

TCP: Retransmit Timeouts

- Sender sets timer for each sent packet
  - when ACK returns, timer canceled
  - if timer expires before ACK returns, packet resent

- Expected time for ACK to return: RTT

- TCP estimates round-trip time using EWMA
  - measurements m_i from timed packet/ACK pairs
  - $RTT_i = ((1-\alpha) \times RTT_{i-1} + \alpha \times m_i)$
  - Original TCP retransmit timeout
    - $RTO_i = \beta \times RTT_i$
    - original TCP: $\beta = 2$

Mean and Variance: Jacobson's RTT Estimator

- Above link load of 30% at router, $\beta \times RTT_i$ will retransmit too early!
- Response to increasing load: waste bandwidth on duplicate packets
- Result: **congestion collapse!**

- [Jacobson 88]: estimate v_i, mean deviation (EWMA of $|m_i - RTT_i|$), stand-in for variance

  $$v_i = v_{i-1} \times (1-\gamma) + \gamma \times |m_i - RTT_i|$$

- **All modern TCPs:**

  use $RTO_i = RTT_i + 4v_i$

Connection Startup Behavior

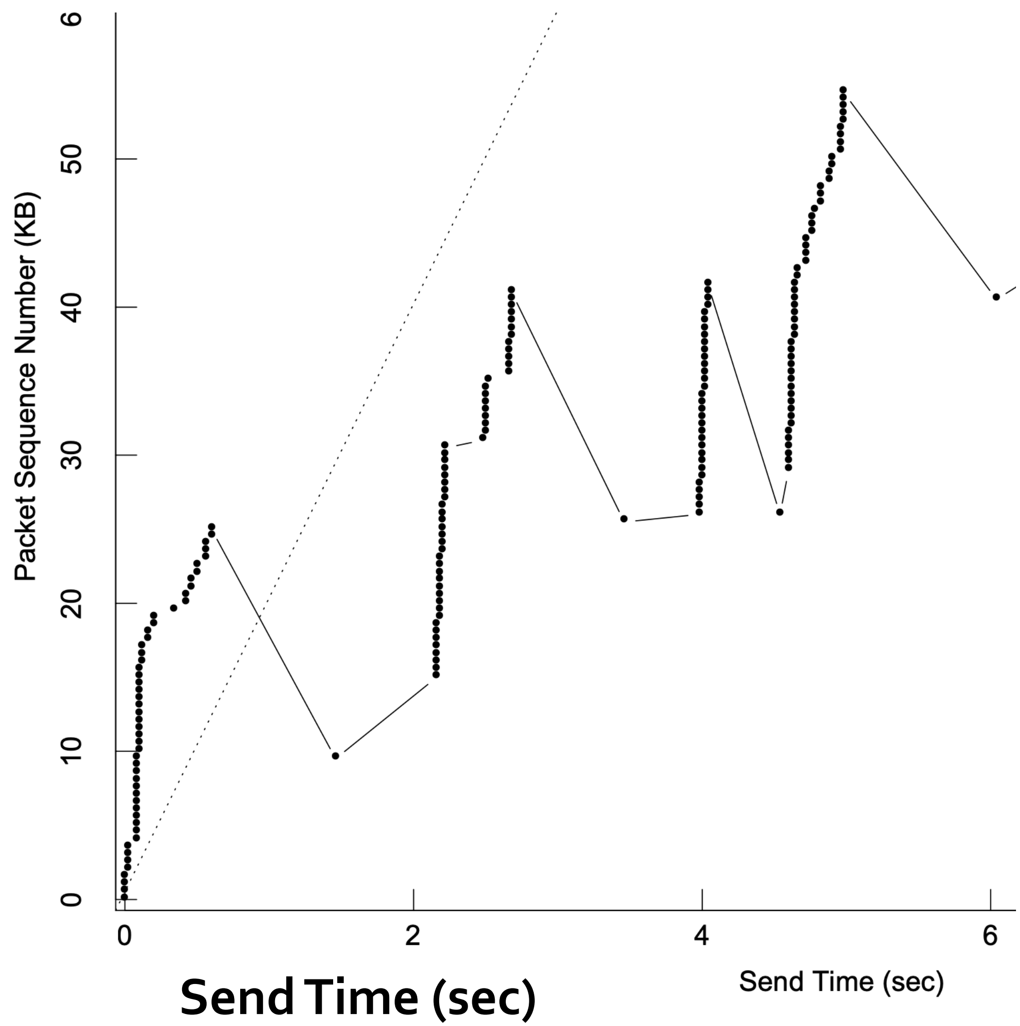
- TCP control of window size: *Slow Start*

- Original TCP, before [Jacobson 88]:
  - At connection start, send full window of packets
  - retransmit each packet just after its timer expires

  - **Result:** window-sized **bursts of packets** sent into network

Pre-Jacobson TCP (Obsolete!)

- Time-sequence plot taken at sender
- Bursts of packets: vertical lines
- Spurious retransmits: repeats at same y-value (enough buffer on path)
- Dashed line: available 20 Kbps capacity



Reaching Equilibrium: Slow Start

- At connection start: sender sets congestion window size, cwnd, to pktSize, not whole window

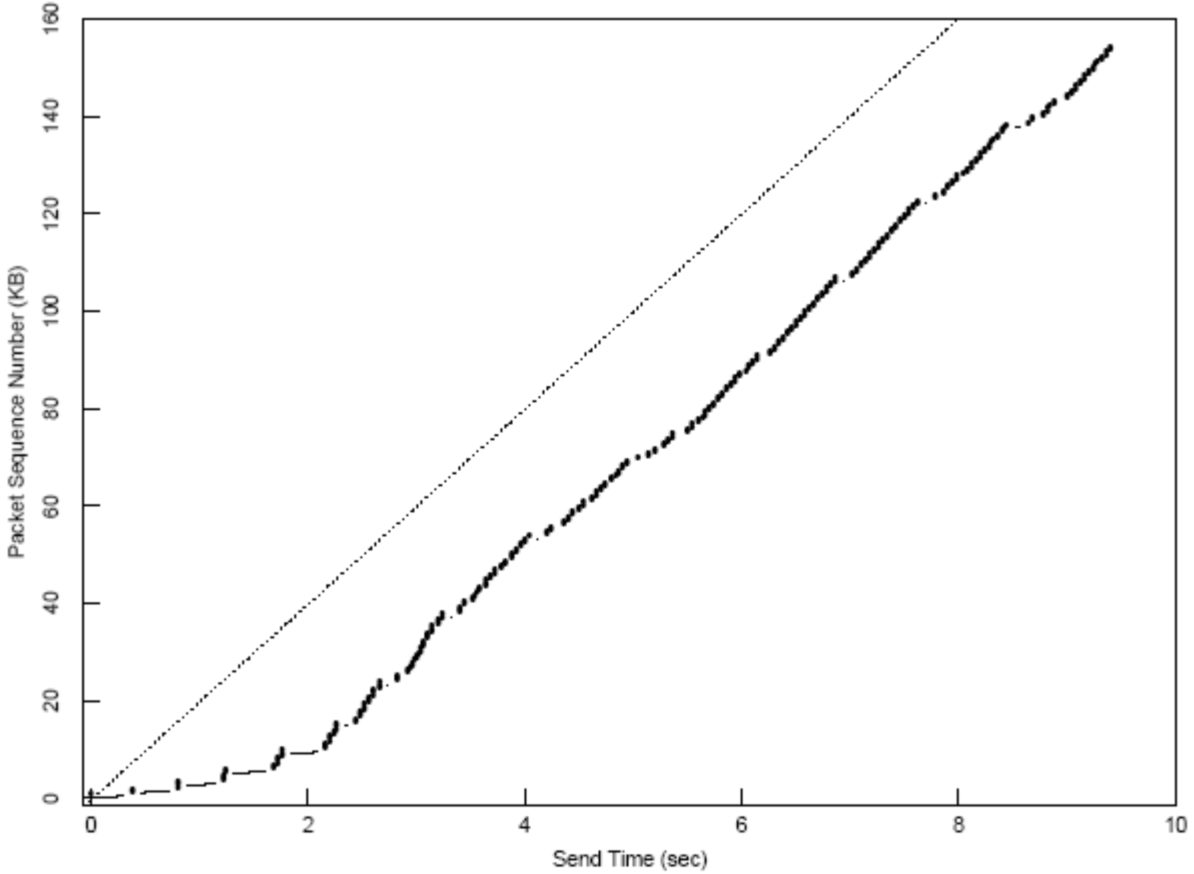
- Sender sends up to minimum of receiver's advertised window size W and cwnd

- Upon return of each ACK until receiver's advertised window size reached, increase cwnd by pktSize bytes

- "Slow" means exponential window increase!

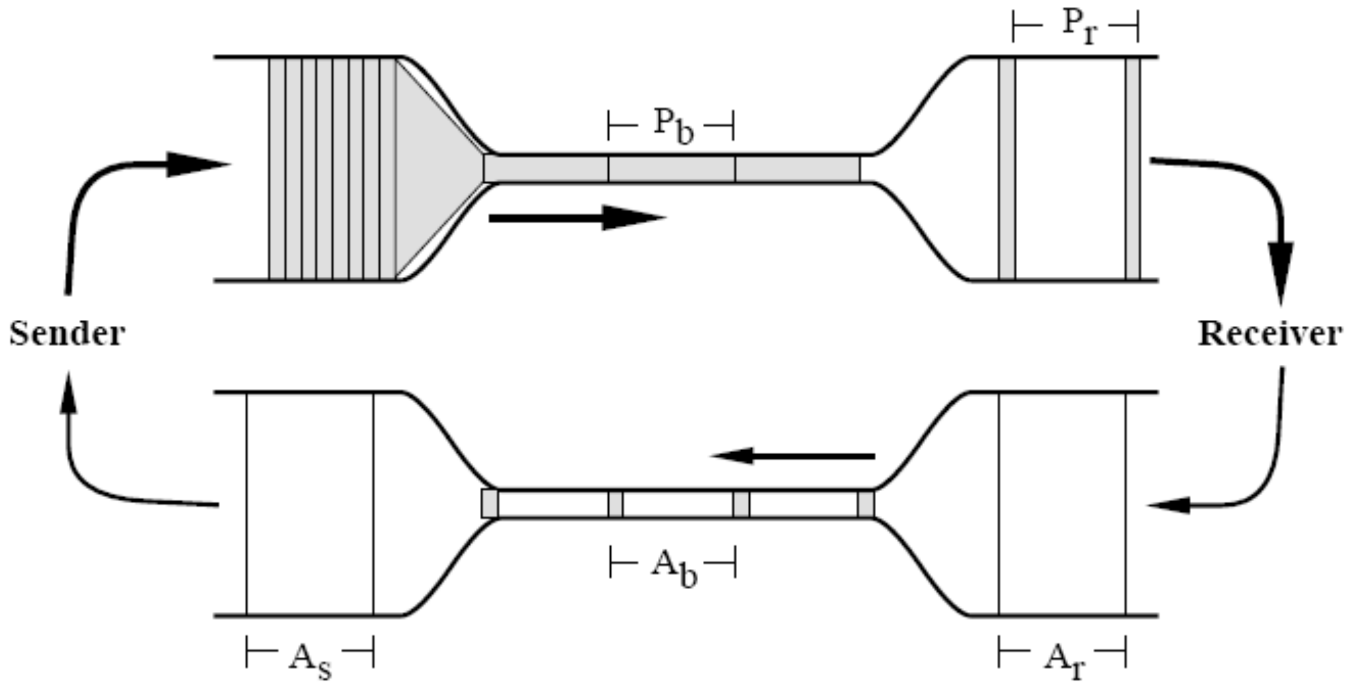
  – Takes $\log_2(W/pktSize)$ RTTs to reach receiver's advertised window size W

Post-Jacobson TCP: Slow Start and Mean+Variance RTT Estimator



- Time-sequence plot at sender; dashed line = available capacity
- "Slower" start
- No spurious retransmits

Self-Clocking: Conservation of Packets



- **Goal:** **self-clocking transmission**
  - each ACK returns, one data packet sent
  - spacing of returning ACKs: matches spacing of packets in time at slowest link on path P_b

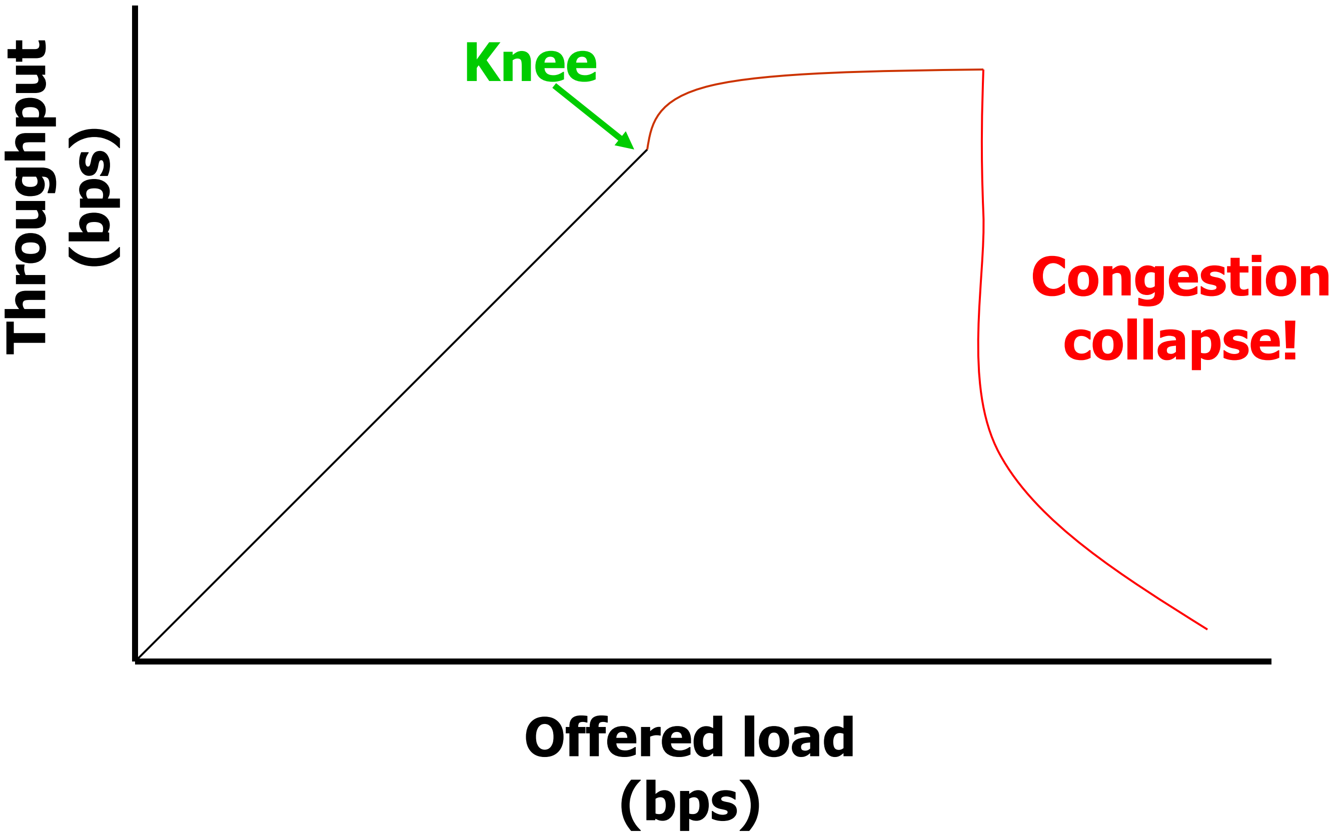
Today

- Pacing Transmissions
- Slow Start and Self-clocking
- **Congestion control**
- Learning to Share: Chiu-Jain phase plots
- Modeling Throughput

Goals in Congestion Control

- Achieve high link utilization; don't waste capacity!
- Divide bottleneck link capacity fairly among users
- Be stable: converge to steady allocation among users
- Avoid congestion collapse

Congestion Collapse



- Cliff behavior observed in [Jacobson 88]

Congestion Requires Slowing Senders

- Recall: big buffers can't prevent congestion collapse
  - Senders must **slow down** to alleviate congestion. How?
  - **Absence of ACKs** implicitly indicates congestion

- TCP sender's window size determines sending rate

- *How can sender learn the right cwnd?*
  - **Search** for it, by adapting window size
  - **Feedback** from network: ACKs
    return (window OK) or do not
    return (window too big)

Avoiding Congestion: Multiplicative Decrease

- Upon **timeout** for sent packet, sender presumes packet lost to congestion, and:
  - sets ssthresh = cwnd / 2
  - sets cwnd = pktSize
  - uses slow start to grow cwnd up to ssthresh
- End result: cwnd = cwnd / 2, via slow start
- Sender sends one window per RTT
  - Halving cwnd halves transmit rate

Avoiding Congestion: Additive Increase

- No feedback to indicate TCP using **less** than its fair share of bottleneck

- Solution: speculatively increase window size as ACKs return
  - Additive increase: for each returning ACK,

    cwnd = cwnd + (pktSize × pktSize)/cwnd

    - Increases cwnd by ~pktSize bytes per RTT

Combined algorithm: Additive Increase, Multiplicative Decrease (AIMD)

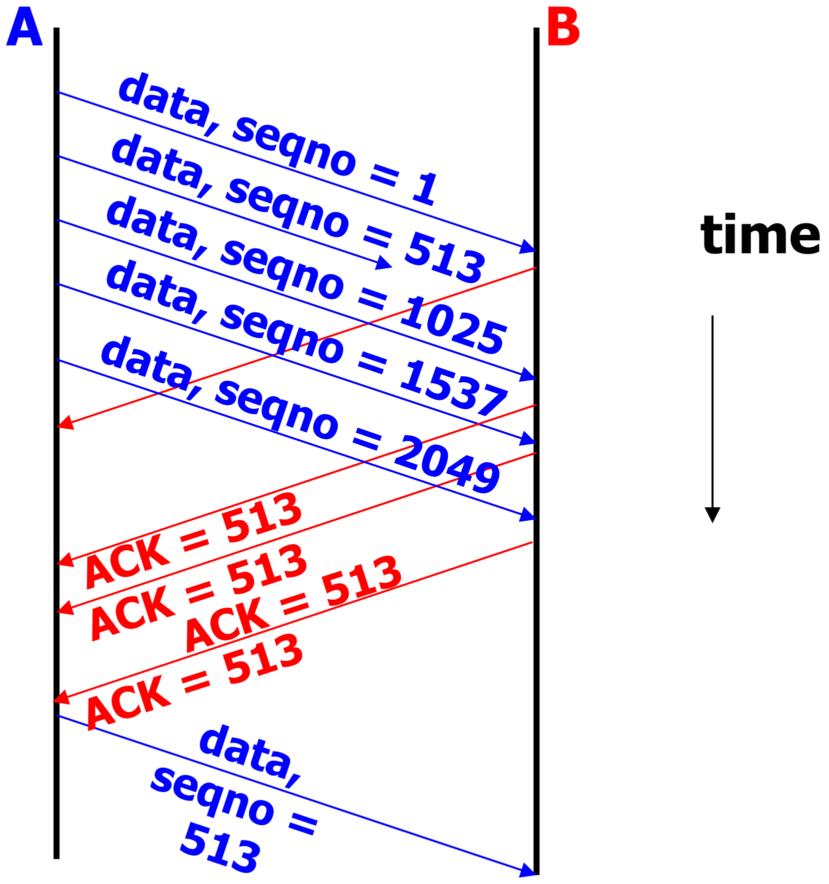
Refinement: Fast Retransmit (I)

- Sender must wait well over RTT for timer to expire before loss detected
  - TCP's minimum retransmit timeout: 1 sec

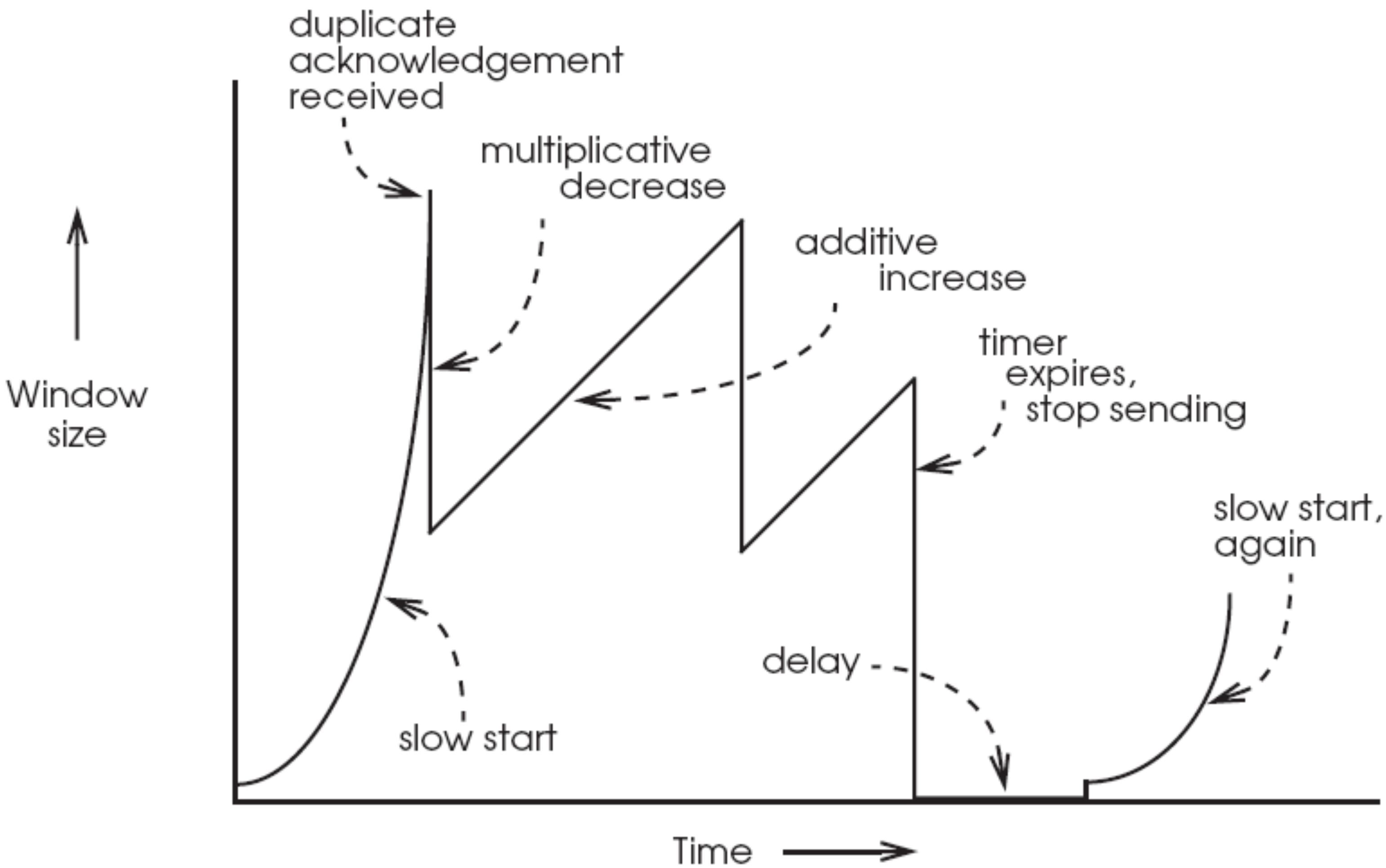
- Another loss indication: duplicate ACKs
  - Suppose sender sends 1, 2, 3, 4, 5, but 2 lost
  - Receiver receives 1, 3, 4, 5
  - Receiver sends cumulative ACKs 2, 2, 2, 2
  - Loss causes duplicate ACKs!

Fast Retransmit (II)

- Upon arrival of 3 duplicate ACKs, sender:
  - sets cwnd = cwnd/2
  - retransmits "missing" packet
  - no slow start
- Not only loss causes dup ACKs, reordering, too



AIMD in Action



- Sender **searches** for correct window size

Why AIMD?

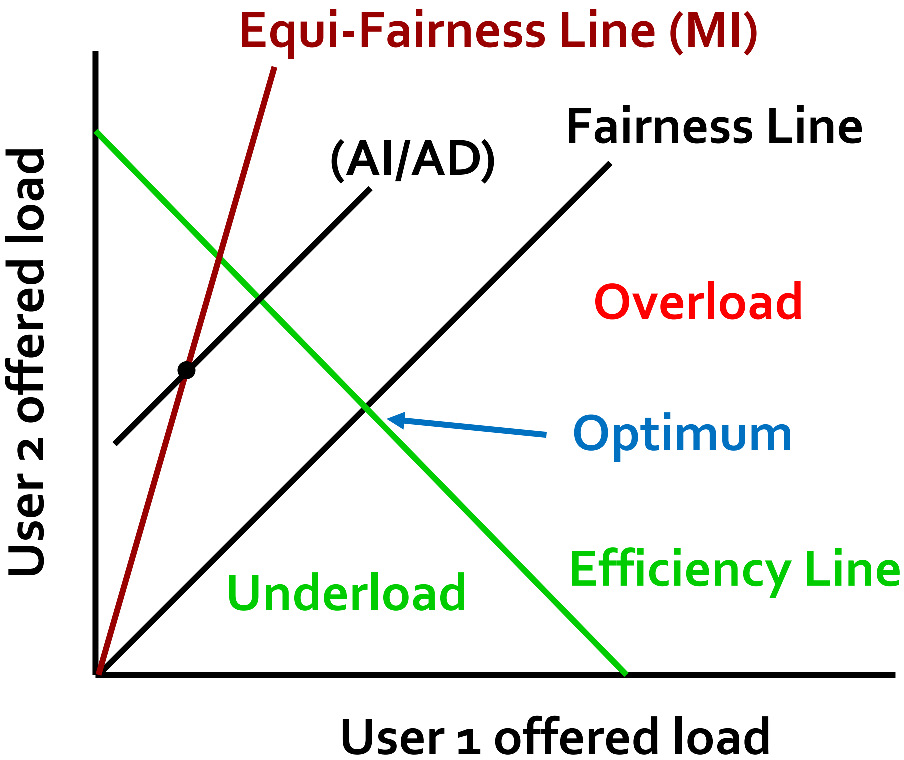
- Other control rules possible
  - E.g., MIMD, AIAD, …
- Recall goals:
  - Links fully utilized (efficient)
  - Users share resources fairly
- TCP adapts all flows' window sizes independently
- Must choose a control that will always converge to an efficient and fair allocation of windows

Today

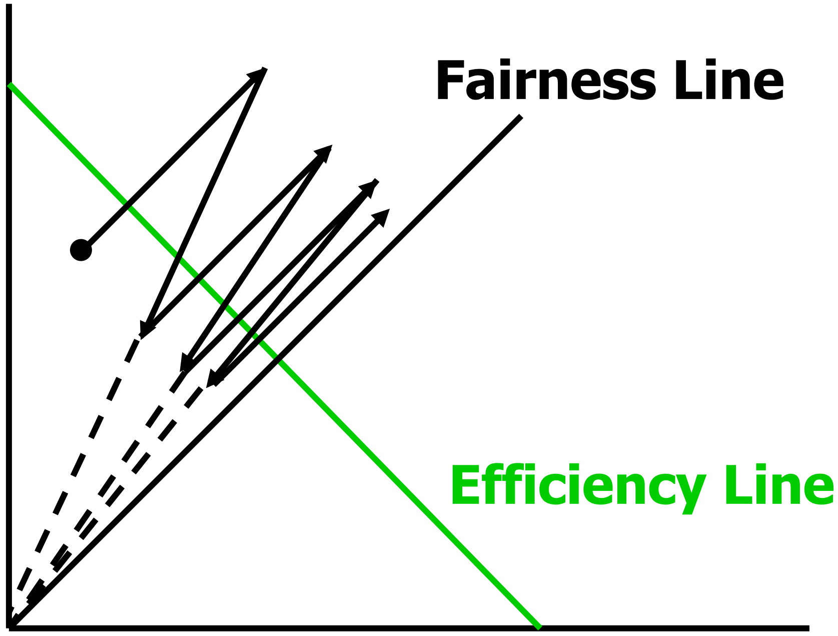
- Pacing Transmissions
- Slow Start and Self-clocking
- Congestion control
- **Learning to Share: Chiu-Jain phase plots**
- Modeling Throughput

Chiu-Jain Phase Plots

- Consider two users sharing a bottleneck link
  - Plot bandwidths allocated to each
- **Efficiency Line:** sum of two users' rates = bottleneck capacity
- **Fairness Line:** two users' rates equal
- **Equi-Fairness Line:** ratio of two users' rates fixed

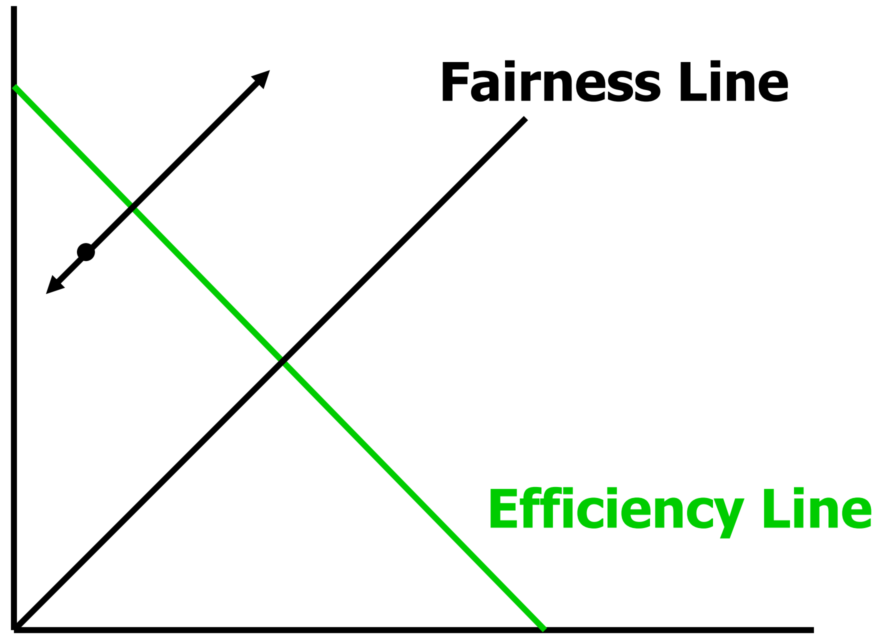


Chiu Jain: AIMD



- AIMD converges to optimum efficiency and fairness

Chiu Jain: AIAD



- AIAD doesn't converge to optimum point!
- Similar oscillations for MIMD

Today

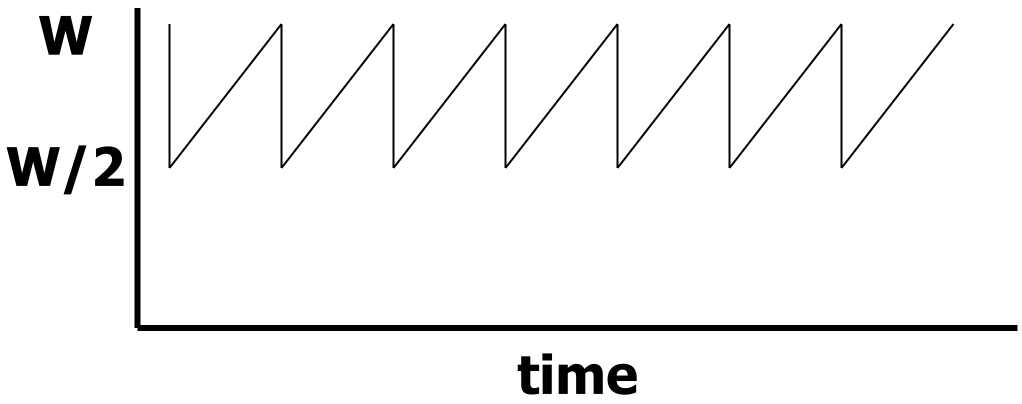
- Pacing Transmissions
- Slow Start and Self-clocking
- Congestion control
- Learning to Share: Chiu-Jain phase plots
- **Modeling Throughput**
- Ending a TCP Connection

Modeling Throughput, Loss, and RTT

- *How do packet loss rate and RTT affect throughput TCP achieves?*

- Assume:
  – only fast retransmits
  – no timeouts (so no slow starts in steady-state)

Evolution of Window Over Time



- Average window size: 3W/4
- One window sent per RTT
- Bandwidth:
  - 3W/4 packets per RTT
  - (3W/4 x packet size) / RTT bytes per second
  - W depends on loss rate…

Loss and Window Size

- Assume no delayed ACKs, fixed RTT
- cwnd grows by one packet per RTT
- So it takes W/2 RTTs to go from window size W/2 to window size W; this period is one cycle
- How many packets sent in total?
  - $((3W/4) / RTT) \times (W/2 \times RTT) = 3W^2/8$
- One loss per cycle (as window reaches W)
  - loss rate: $p = 8/3W^2$
  - $W = sqrt(8/3p)$

Throughput, Loss, and RTT Model

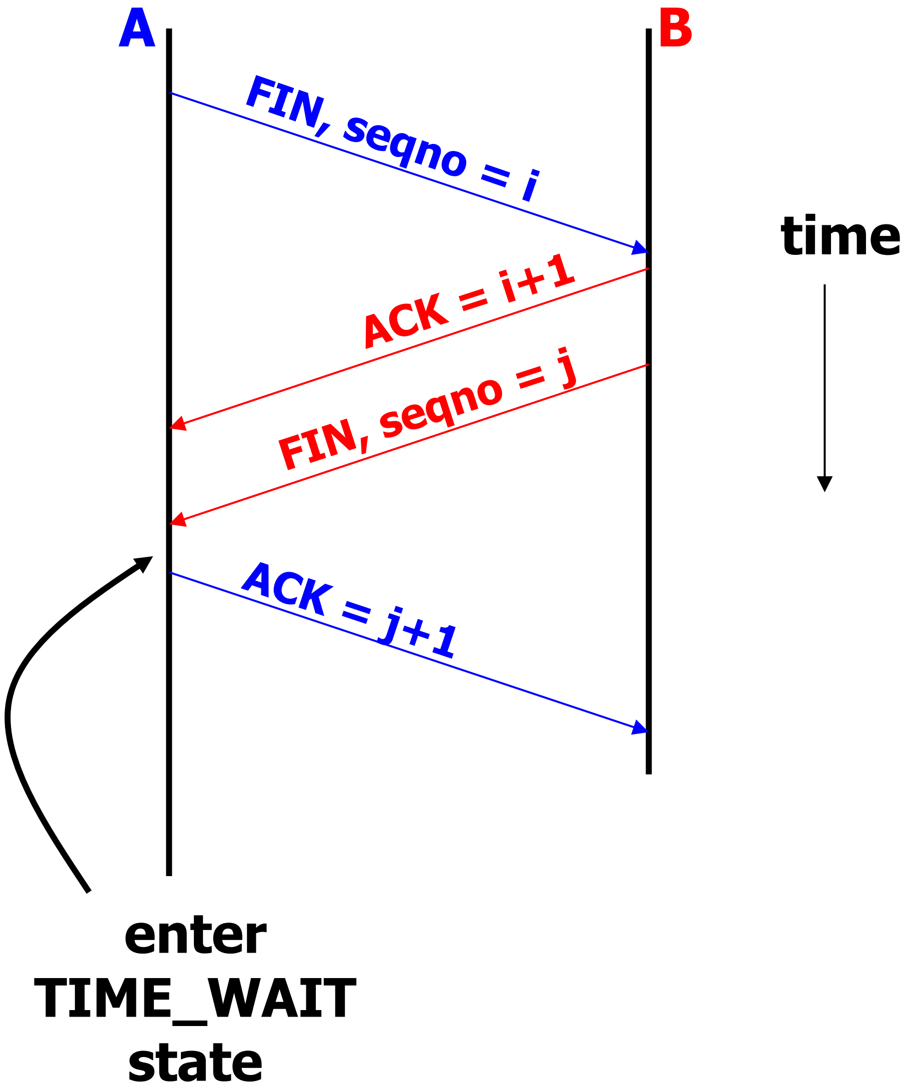
- W = sqrt(8/3p) = (4/3) x sqrt(3/2p)
- Recall:
  - Bandwidth: B = (3W/4 x packet size) / RTT
- B = packet size / (RTT x sqrt(2p/3))
- Consequences:
  - Increased loss quickly reduces throughput
  - At same bottleneck, flow with longer RTT achieves less throughput than flow with shorter RTT!

Today

- Pacing Transmissions
- Slow Start and Self-clocking
- Congestion control
- Learning to Share: Chiu-Jain phase plots
- Modeling Throughput
- **Ending a TCP Connection**

TCP: Connection Teardown

- Data may flow bidirectionally
- Each side independently decides when to close connection
- In each direction, FIN answered by ACK
- Must reliably terminate connection for both sides
  - During TIME_WAIT state at first side to send FIN, ACK valid FINs that arrive
- Must avoid mixing data from old connection with new one
  - During TIME_WAIT state, disallow all new connections for 2 x max segment lifetime



Summary: TCP and Congestion Control

- Connection establishment and teardown
  - Robustness against delayed packets crucial
- Round-trip time estimation
  - EWMAs estimate both RTT mean and deviation
- Congestion detection at sender
  - Timeout: half window, slow start from one packet
  - Fast retx: three dup ACKs, half window, no slow start
- Search for optimal sending window size
  - Additive increase, multiplicative decrease (AIMD)
  - AIMD converges to high utilization, fair sharing