

Reliable Transport II: TCP and Congestion Control

Brad Karp
UCL Computer Science



CS 3035/GZ01
31st October 2013

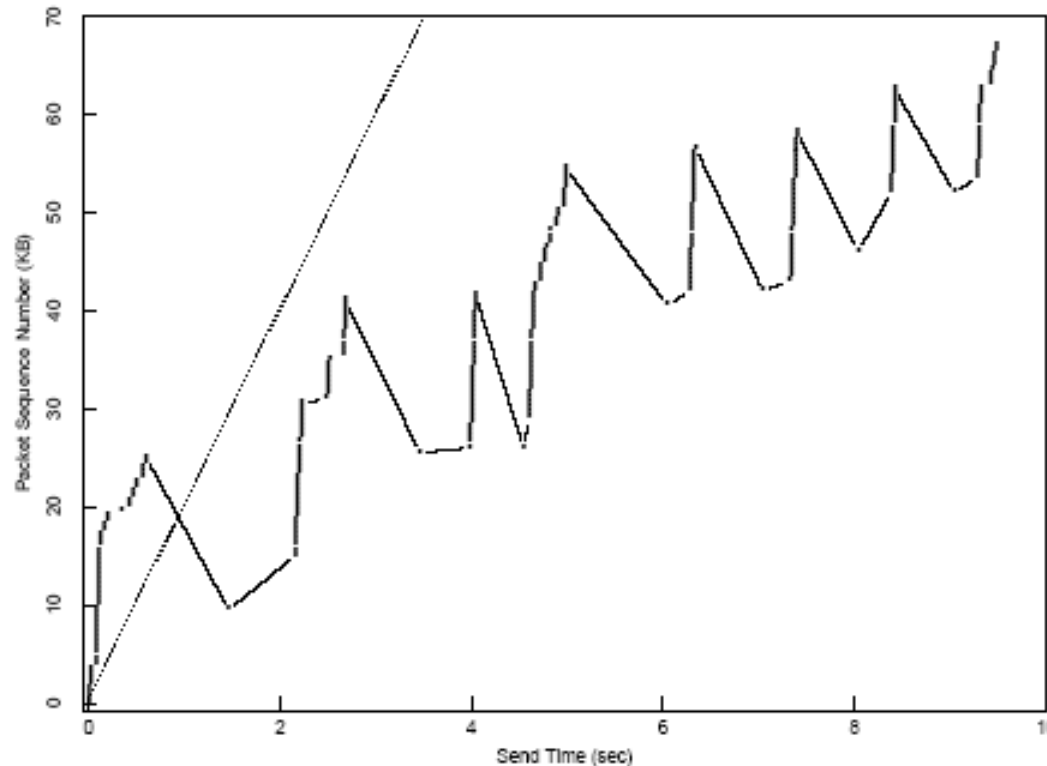
Outline

- Slow Start
- AIMD Congestion control
- Throughput, loss, and RTT equation
- Connection teardown
- Protocol state machine

Retransmit Behavior

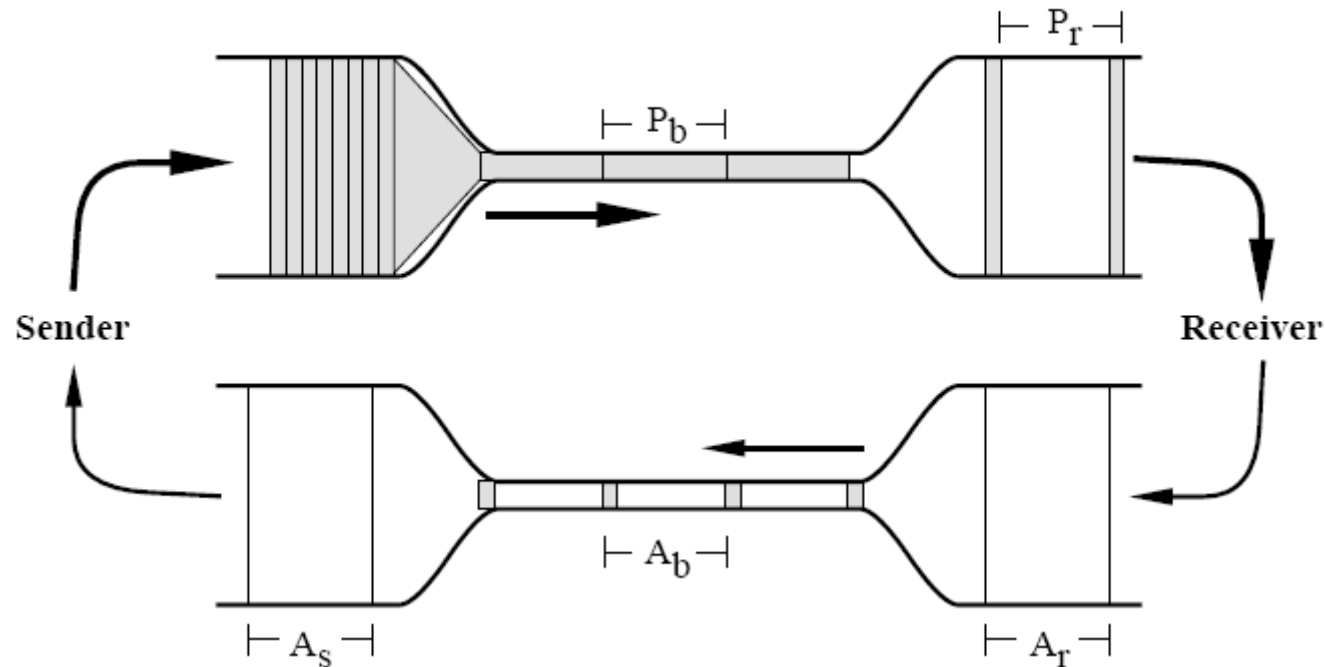
- Original TCP (pre-AIMD design), before [Jacobson 88]:
 - at start of connection, send full window of packets
 - retransmit each packet immediately after its timer expires
- Result: window-sized bursts of packets sent into network

Pre-Jacobson TCP (Obsolete!)



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

Self-Clocking: Conservation of Packets

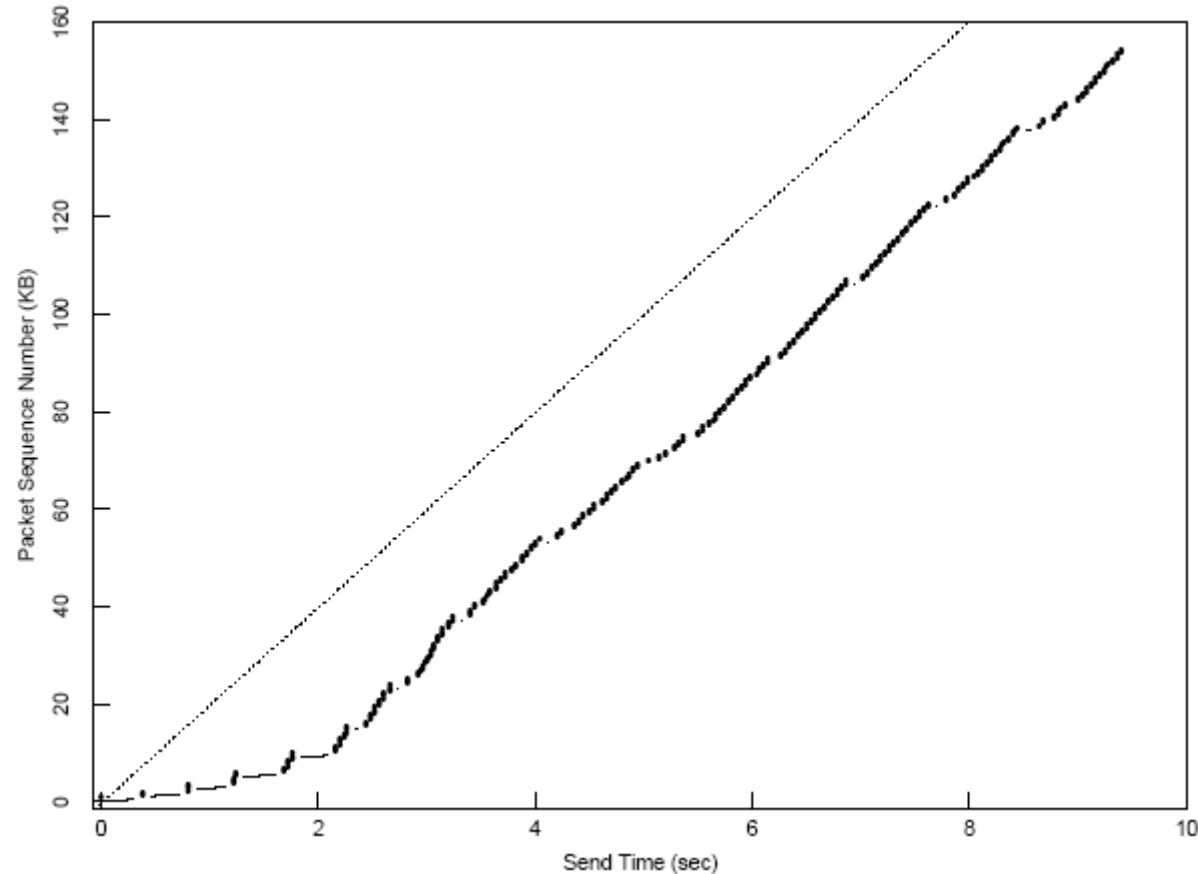


- Goal: **self-clocking transmission**
 - each ACK returns, one data packet sent
 - spacing of returning ACKs: matches spacing of packets in time at slowest link on path

Reaching Equilibrium: Slow Start

- At connection start, sender sets **congestion window size, cwnd**, to **pktSize** (one packet's worth of bytes), 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 **$\log_2(W/\text{pktSize})$ RTTs** to reach receiver's advertised window size W

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



- Time-sequence plot at sender
- “Slower” start
- No spurious retransmits

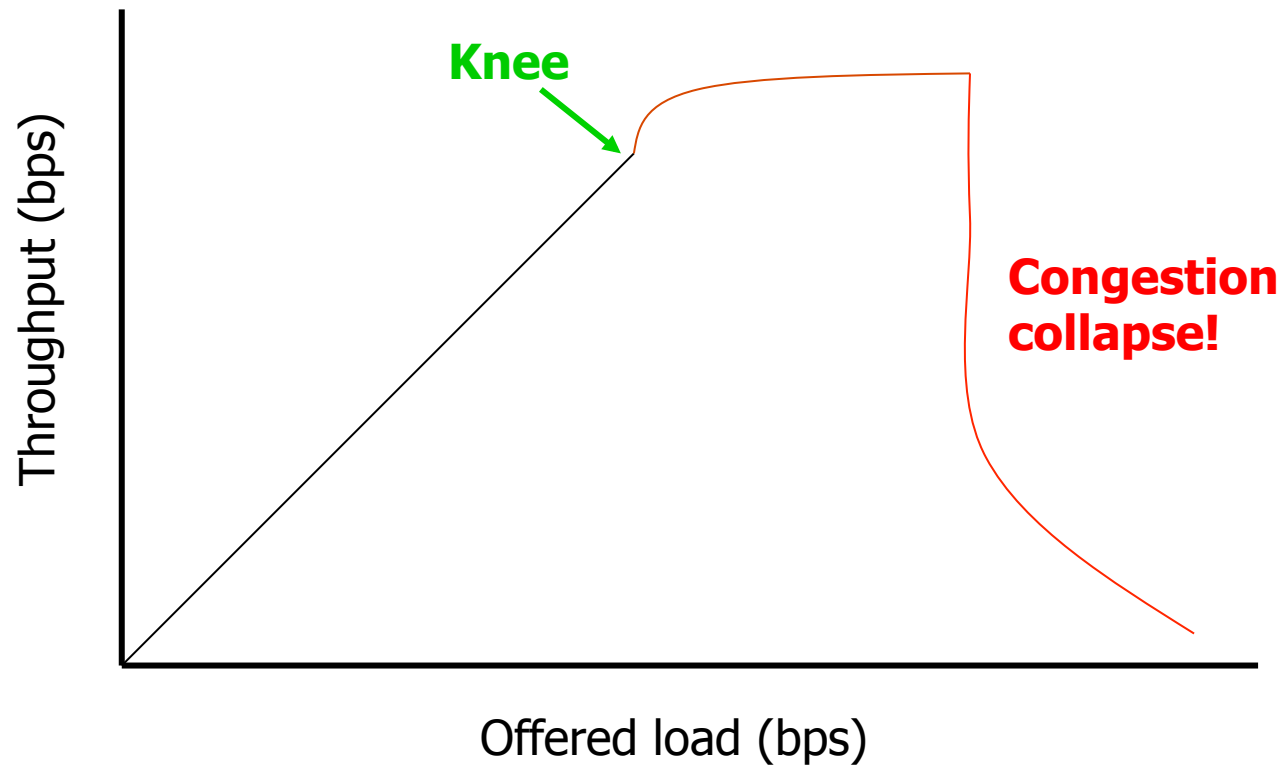
Outline

- Slow Start
- **AIMD Congestion control**
- Throughput, loss, and RTT equation
- Connection teardown
- Protocol state machine

Goals in Congestion Control

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

Congestion Collapse



- Cliff behavior observed in [Jacobson 88]

Congestion Requires Slowing Senders

- Recall: bigger buffers cannot prevent congestion
- Senders must slow to alleviate congestion
- Absence of ACKs implicitly indicates congestion
- TCP sender's window size determines sending rate
- Recall: correct window size is bottleneck bandwidth-delay product
- **How can sender learn this value?**
 - **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

- Recall that sender uses sending window of size $\min(\text{cwnd}, \text{rwnd})$, where rwnd is receiver's advertised window
- Upon timeout for sent packet, sender presumes packet lost to congestion, and:
 - sets $\text{ssthresh} = \text{cwnd} / 2$
 - sets $\text{cwnd} = \text{pktSize}$
 - uses slow start to grow cwnd up to ssthresh
- End result: $\text{cwnd} = \text{cwnd} / 2$, via slow start
- Sender sends one window per RTT; halving cwnd halves transmit rate

Avoiding Congestion: Additive Increase

- Drops indicate TCP sending more than its fair share of bottleneck
- 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,
$$cwnd = cwnd + (pktSize \times pktSize) / cwnd$$
 - Increases cwnd by $\sim pktSize$ bytes per RTT

Avoiding Congestion: Additive Increase

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

Combined algorithm:

Additive Increase, Multiplicative Decrease (AIMD)

$\text{cwnd} = \text{cwnd} + (\text{pktSize} \times \text{pktSize}) / \text{cwnd}$

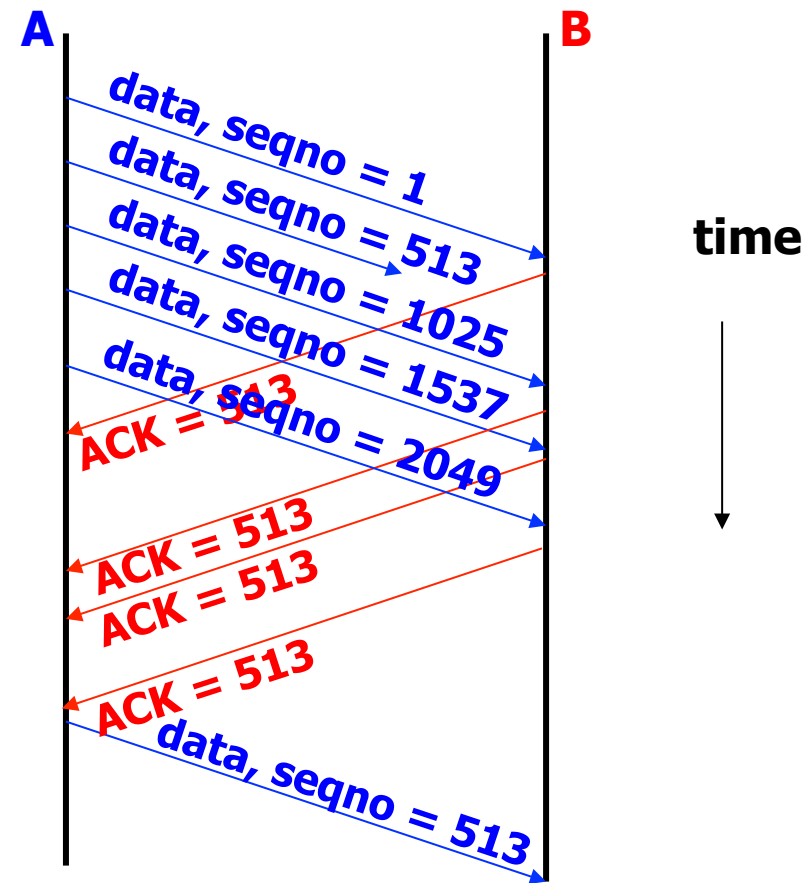
– Increases cwnd by $\sim \text{pktSize}$ bytes per RTT

Refinement: Fast Retransmit (I)

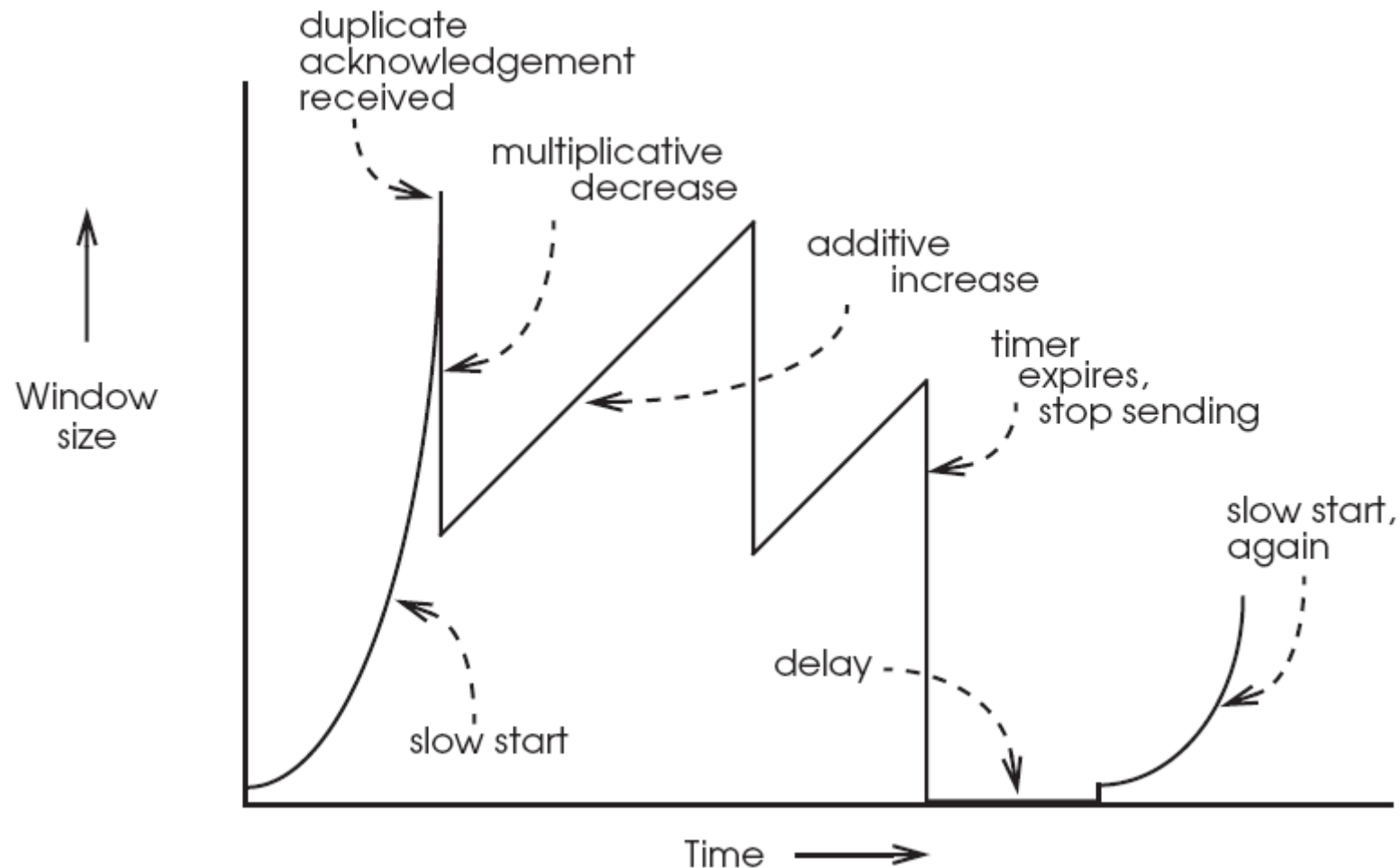
- Sender must wait **well over RTT** for timer to expire before loss detected
- TCP's minimum retransmit timeout: **1 second**
- Another loss indication: **duplicate ACKs**
 - Suppose sender sends 1, 2, 3, 4, 5, but 2 lost
 - Receiver receives 1, 3, 4, 5
 - Receiver sends cumulative ACKs 2, 2, 2, 2
 - **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



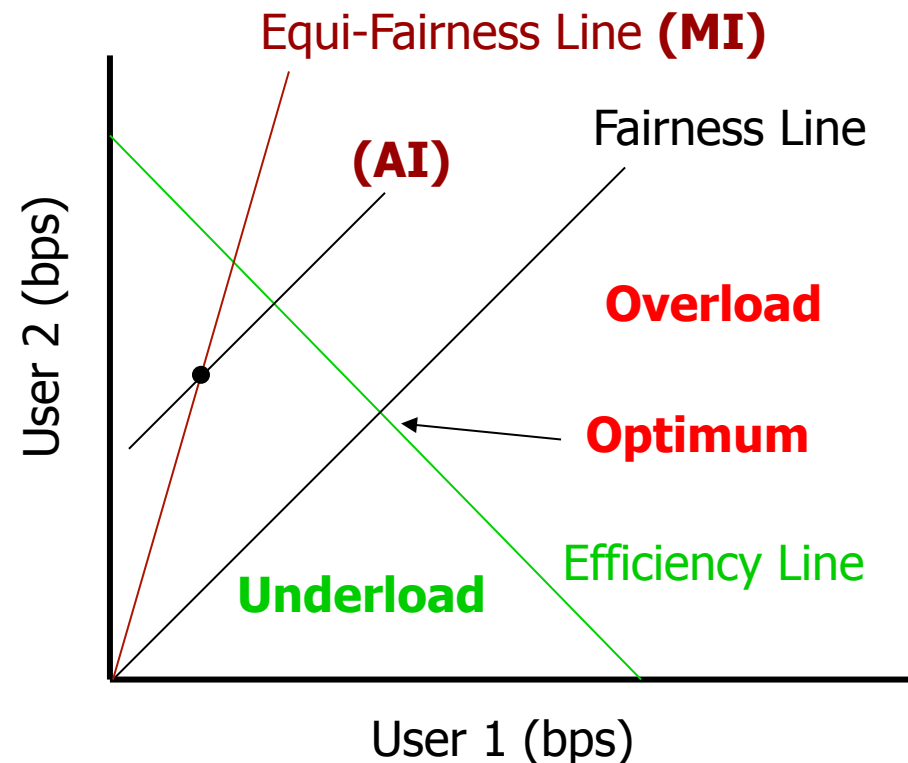
- Sender **searches** for correct window size

Why AIMD?

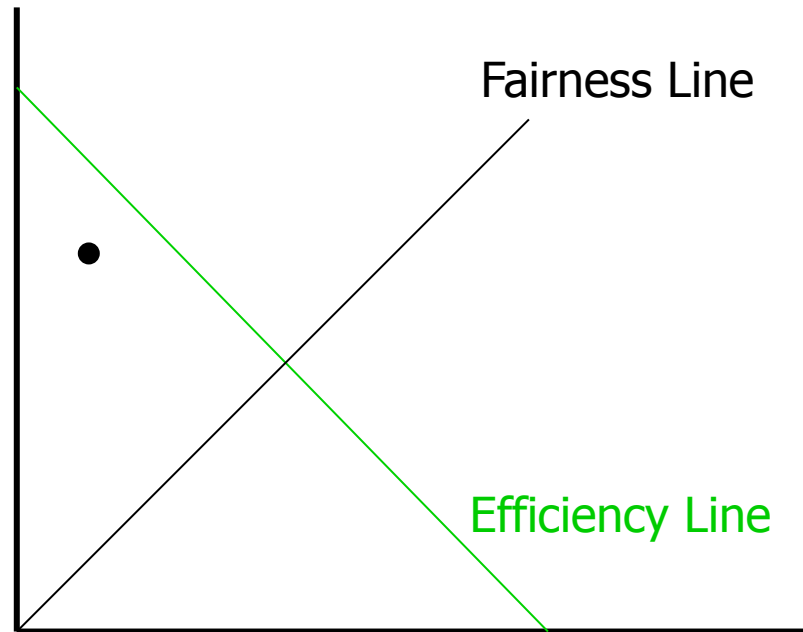
- Other control rules possible
 - E.g., MIMD, AIAD, ...
- Recall goals:
 - Links fully utilized (efficient)
 - Users share resources fairly
- TCP adapts all flows' window sizes independently
- Must choose a control that will always converge to an efficient and fair allocation of windows

Chiu-Jain Phase Plots

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

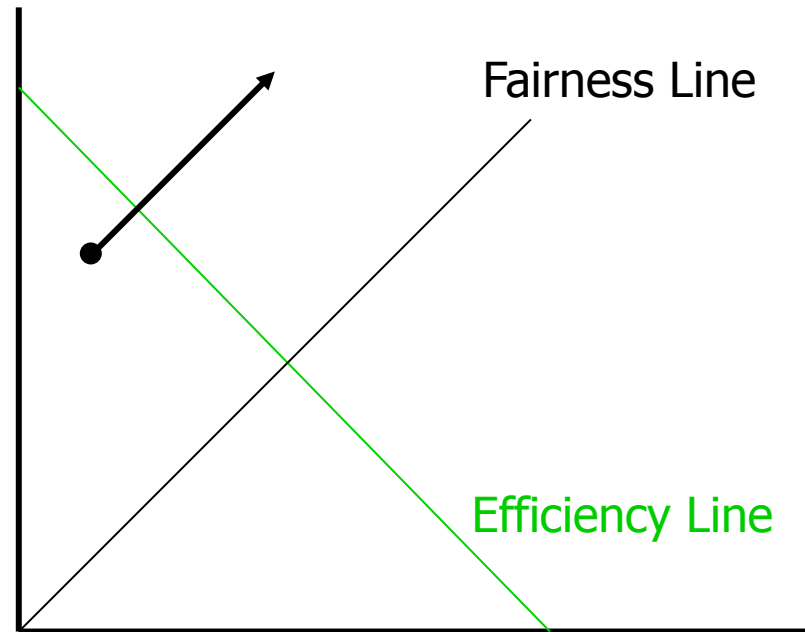


Chiu Jain: AIMD



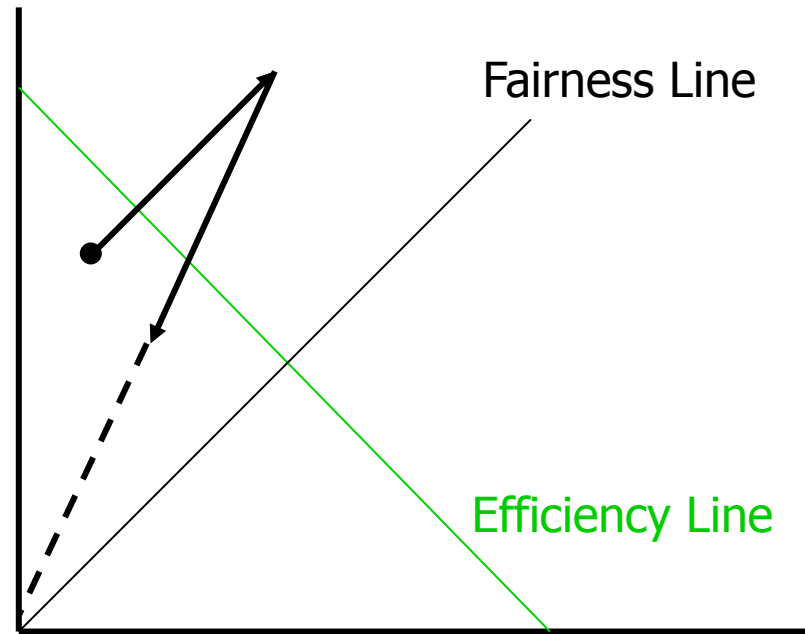
- AIMD converges to optimum efficiency and fairness

Chiu Jain: AIMD



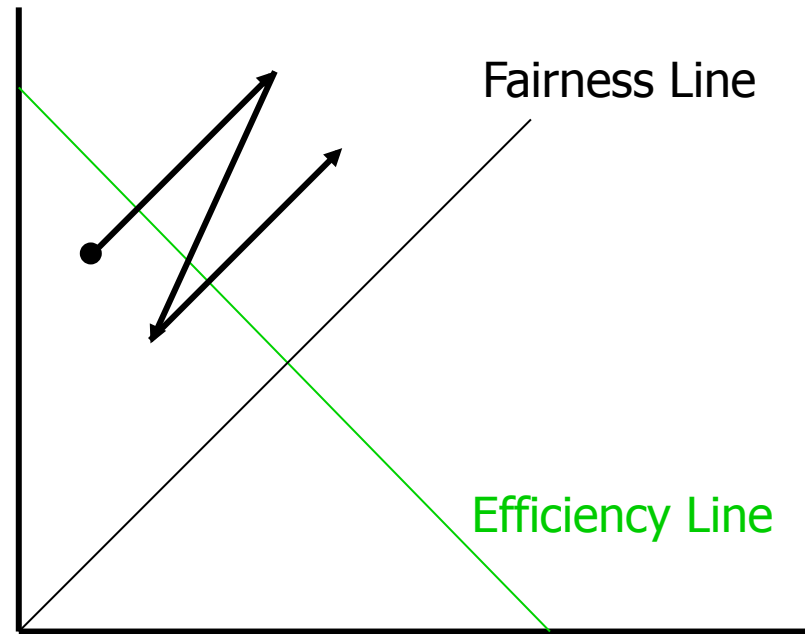
- AIMD converges to optimum efficiency and fairness

Chiu Jain: AIMD



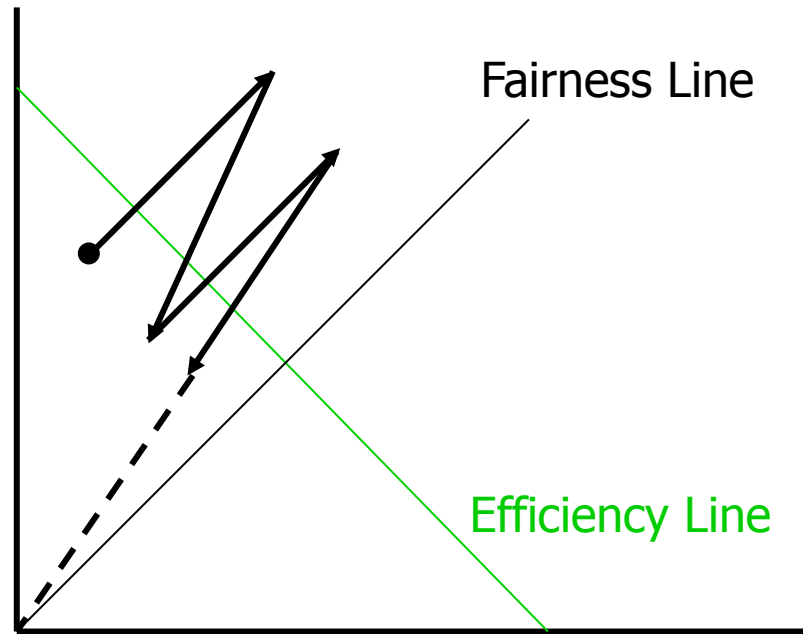
- AIMD converges to optimum efficiency and fairness

Chiu Jain: AIMD



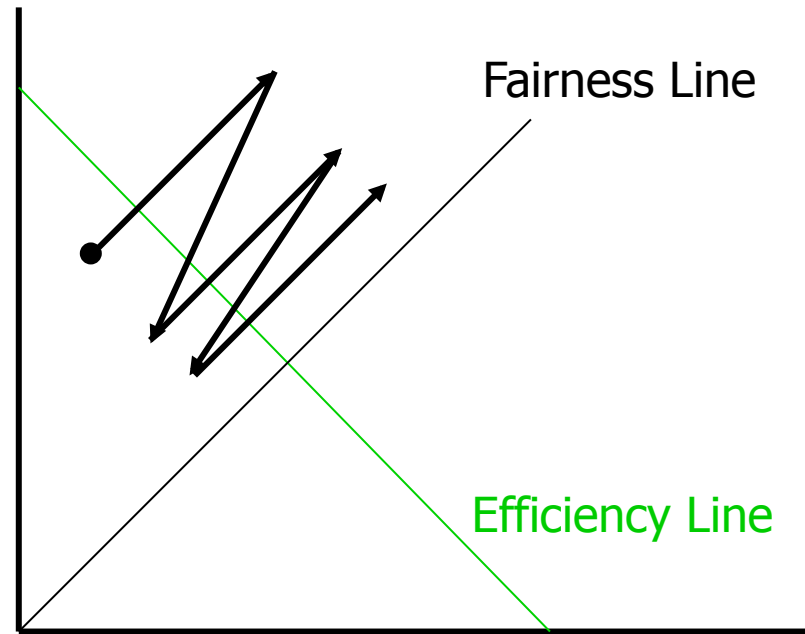
- AIMD converges to optimum efficiency and fairness

Chiu Jain: AIMD



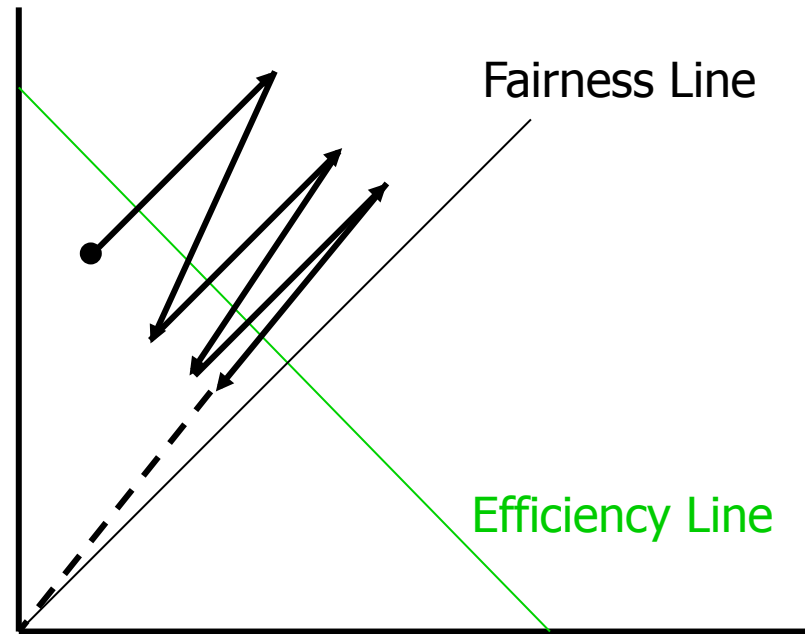
- AIMD converges to optimum efficiency and fairness

Chiu Jain: AIMD



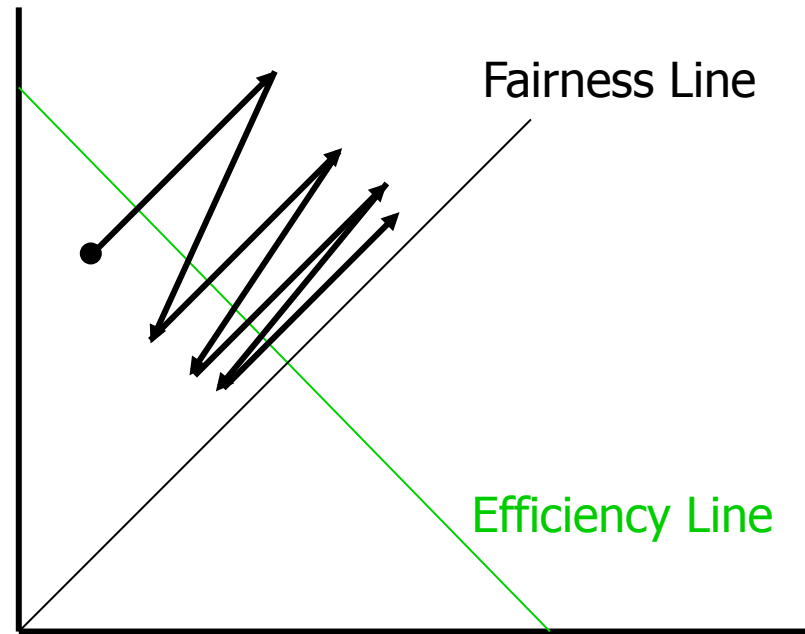
- AIMD converges to optimum efficiency and fairness

Chiu Jain: AIMD



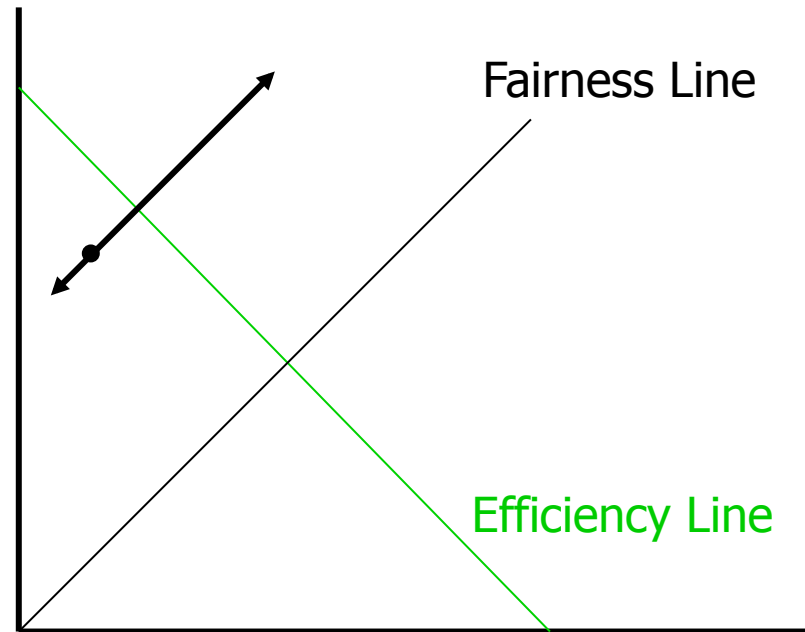
- AIMD converges to optimum efficiency and fairness

Chiu Jain: AIMD



- AIMD converges to optimum efficiency and fairness

Chiu Jain: AIAD



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

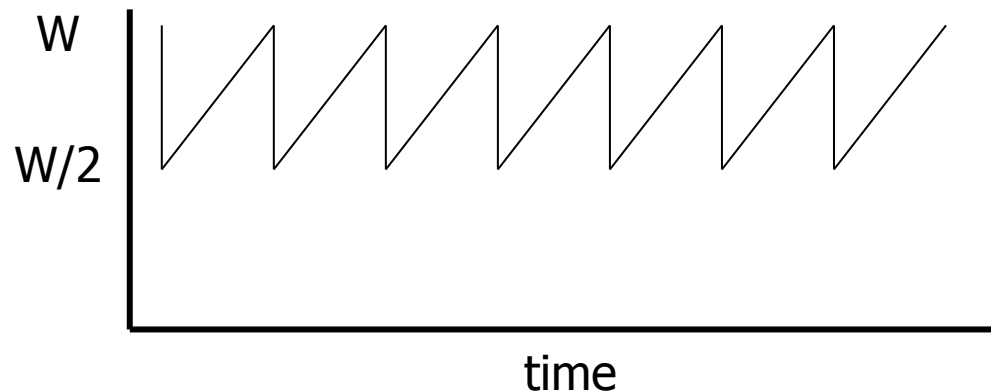
Outline

- Packet header format
- Connection establishment
- Data transmission
- Retransmit timeouts
- RTT estimator
- AIMD Congestion control
- Throughput, loss, and RTT equation
- Connection teardown
- Protocol state machine

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
 - $(3W/4 \times \text{packet size}) / \text{RTT}$ 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

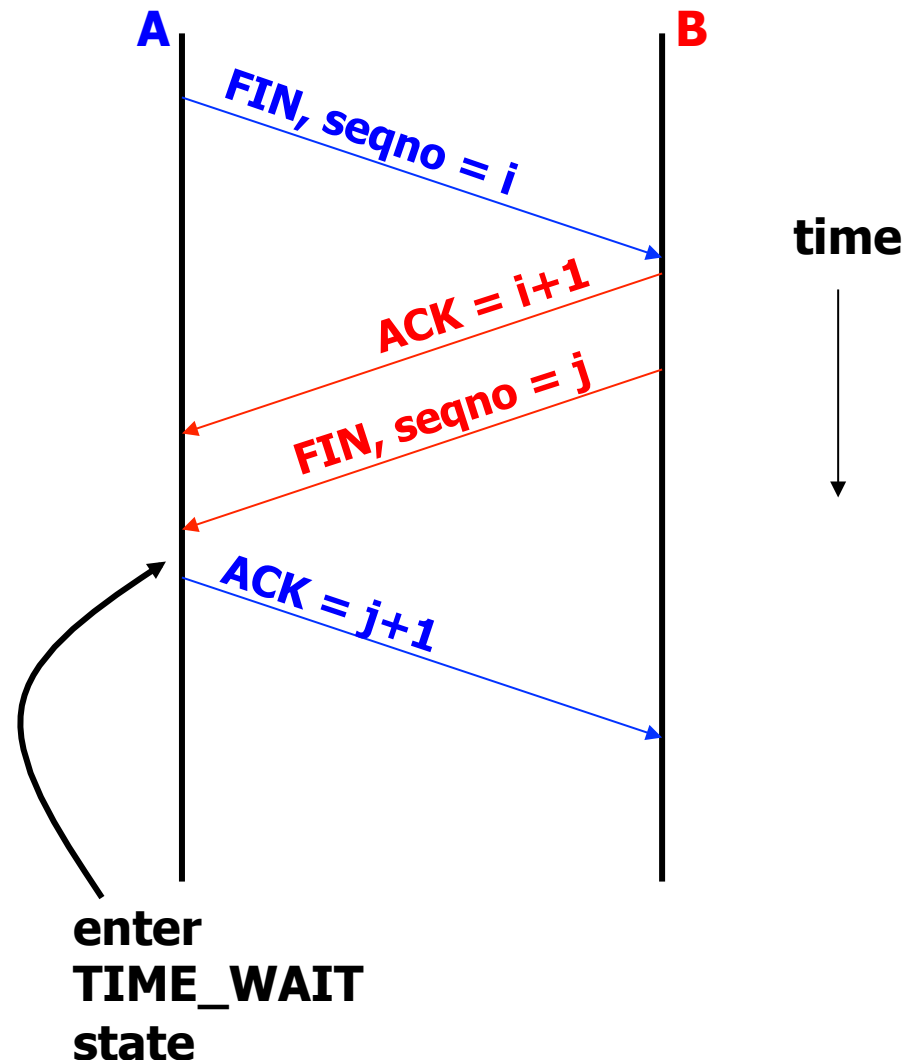
- $W = \sqrt{8/3p} = (4/3) \times \sqrt{3/2p}$
- Recall:
 - Bandwidth: $B = (3W/4 \times \text{packet size}) / \text{RTT}$
- $B = \text{packet size} / (\text{RTT} \times \sqrt{2p/3})$
- Consequences:
 - Increased loss quickly reduces throughput
 - At same bottleneck, flow with longer RTT achieves less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

Outline

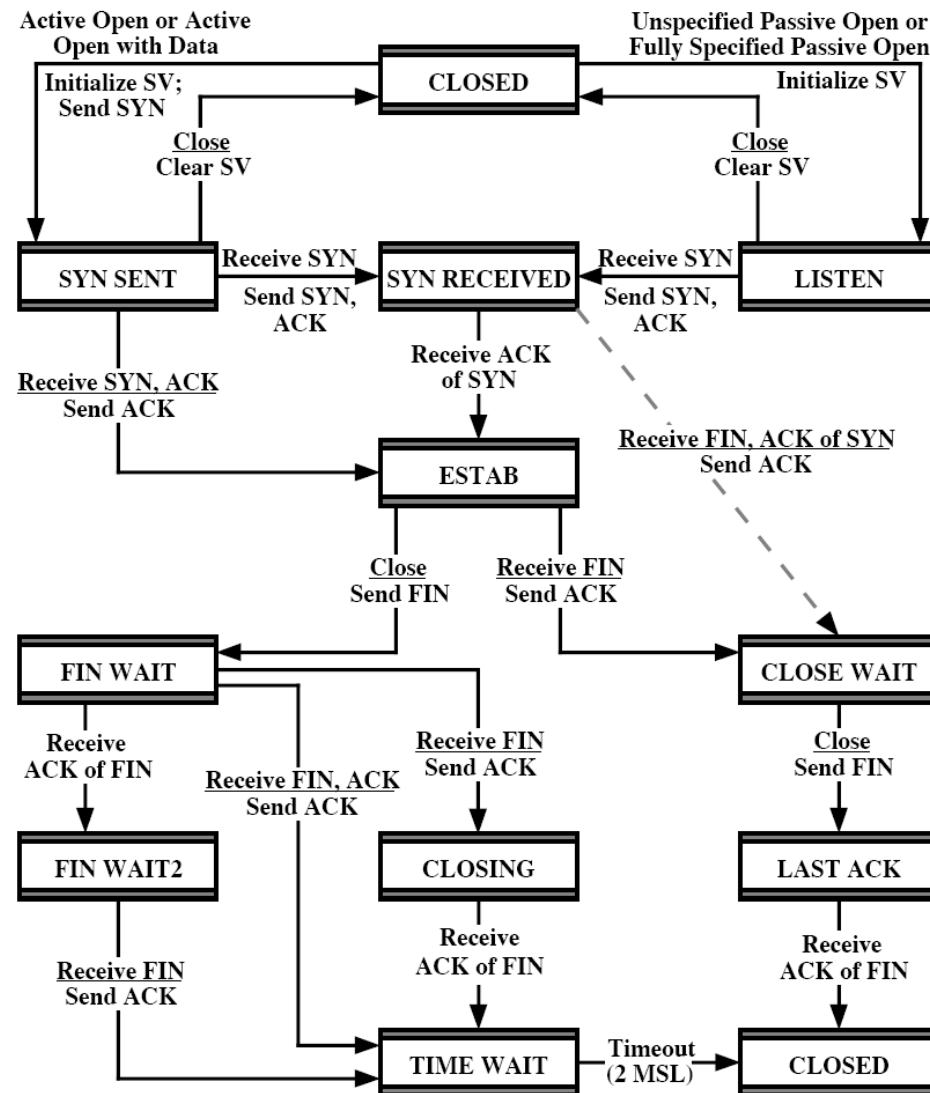
- Packet header format
- Connection establishment
- Data transmission
- Retransmit timeouts
- RTT estimator
- AIMD Congestion control
- Throughput, loss, and RTT equation
- Connection teardown
- Protocol state machine

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**



TCP: Protocol State Machine



Summary: TCP and Congestion Control

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