

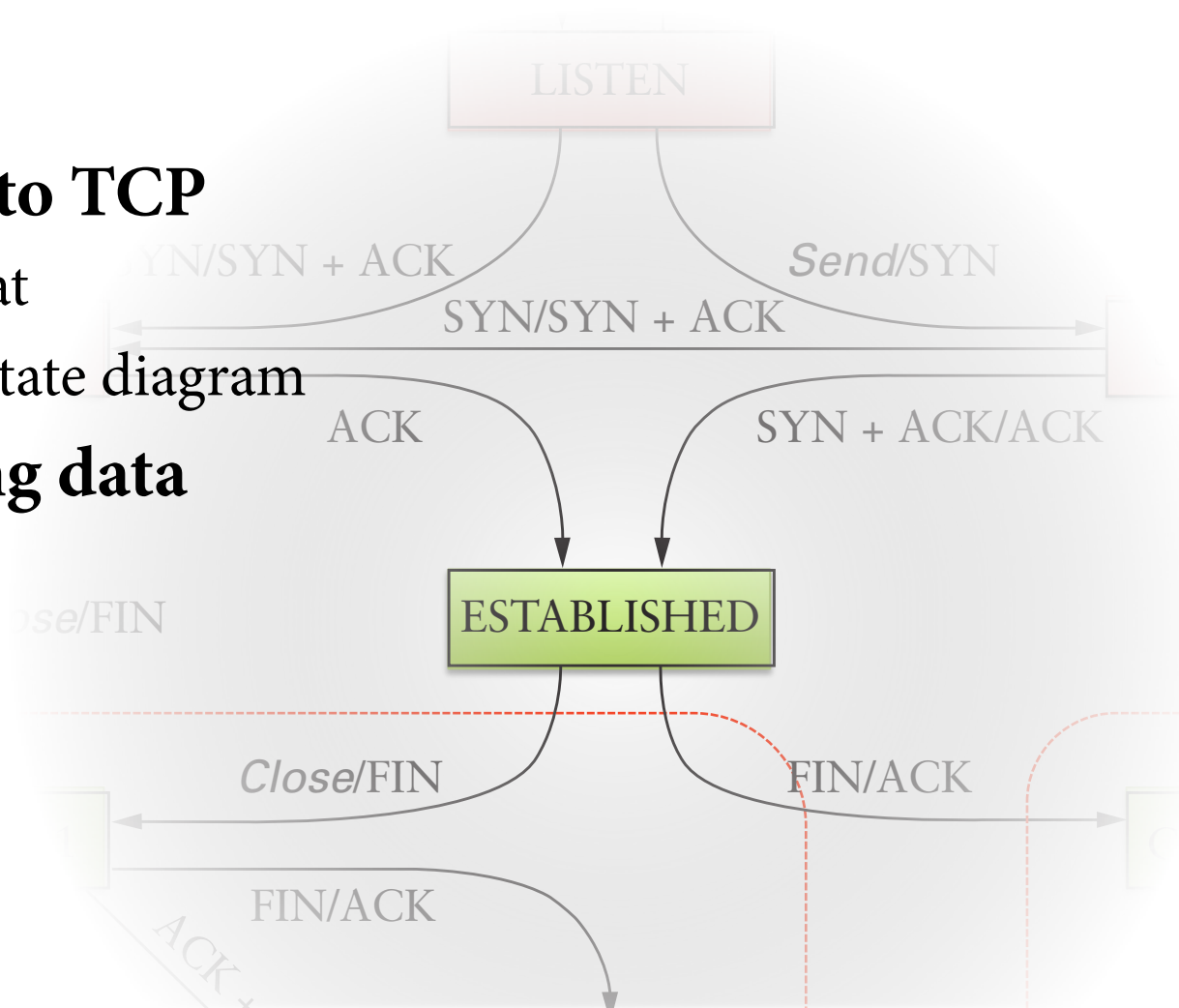
CSCI-1680
Transport Layer II
Data over TCP

Rodrigo Fonseca



Last Class

- **Introduction to TCP**
 - Header format
 - Connection state diagram
- **Today: sending data**



First Goal

- **We should not send more data than the receiver can take: *flow control***
- **When to send data?**
 - Sender can delay sends to get larger segments
- **How much data to send?**
 - Data is sent in MSS-sized segments
 - Chosen to avoid fragmentation

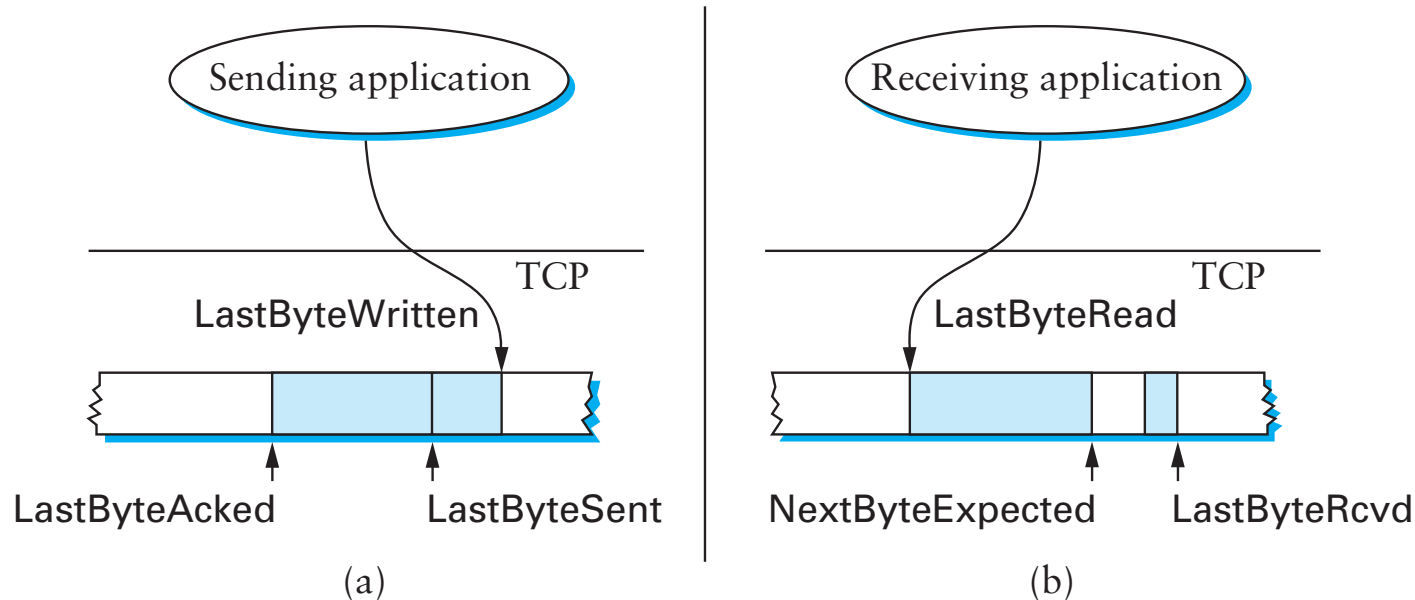


Flow Control

- **Part of TCP specification (even before 1988)**
- **Receiver uses window header field to tell sender how much space it has**



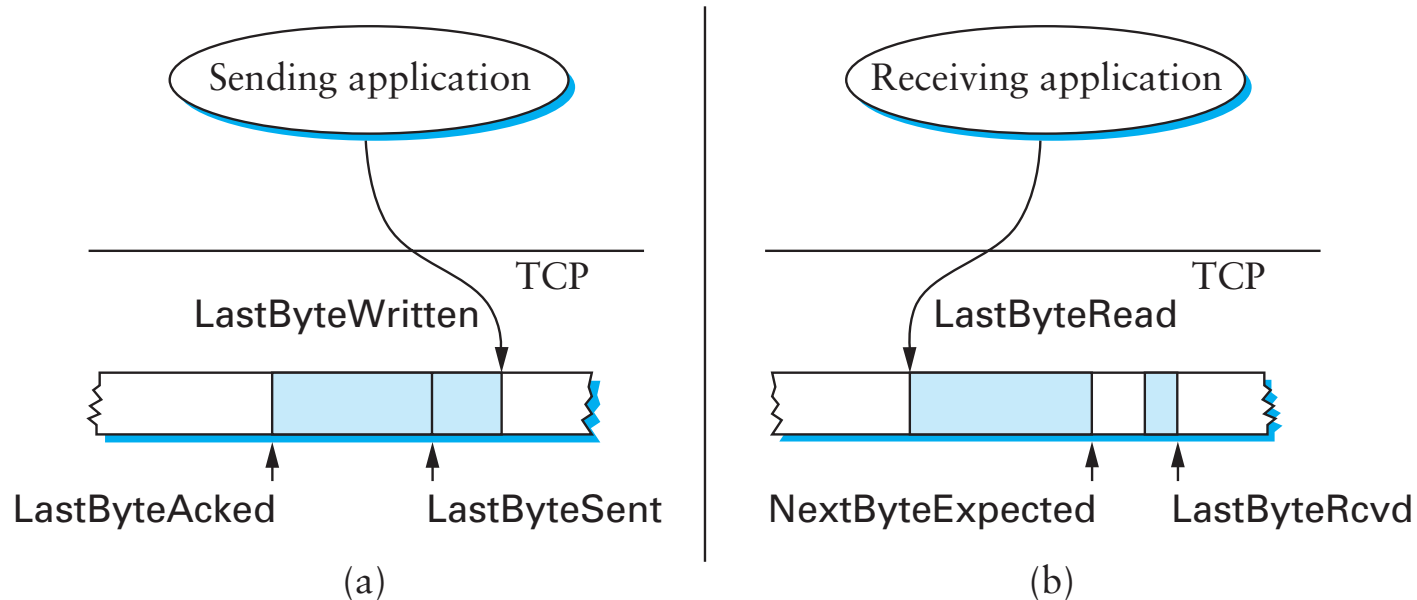
Flow Control



- **Receiver: AdvertisedWindow**
= $\text{MaxRcvBuffer} - ((\text{NextByteExpected} - 1) - \text{LastByteRead})$
- **Sender: LastByteSent - LastByteAacked \leq AdvertisedWindow**
EffectiveWindow = $\text{AdvertisedWindow} - (\text{BytesInFlight})$
LastByteWritten - LastByteAacked \leq MaxSendBuffer



Flow Control



- **Advertised window can fall to 0**
 - How?
 - Sender eventually stops sending, blocks application
- **Sender keeps sending 1-byte segments until window comes back > 0**



When to Transmit?

- **Nagle's algorithm**
- **Goal: reduce the overhead of small packets**
 - If available data and window \geq MSS
 - Send a MSS segment
 - else
 - If there is unAcked data in flight
 - buffer the new data until ACK arrives
 - else
 - send all the new data now
- **Receiver should avoid advertising a window \leq MSS after advertising a window of 0**



Delayed Acknowledgments

- **Goal: Piggy-back ACKs on data**
 - Delay ACK for 200ms in case application sends data
 - If more data received, immediately ACK second segment
 - Note: never delay duplicate ACKs (if missing a segment)
- **Warning: can interact *very* badly with Nagle**
 - Temporary deadlock
 - Can disable Nagle with `TCP_NODELAY`
 - Application can also avoid many small writes



Limitations of Flow Control

- **Network may be the bottleneck**
- **Signal from receiver not enough!**
- **Sending too fast will cause queue overflows, heavy packet loss**
- **Flow control provides *correctness***
- **Need more for performance: congestion control**



Second goal

- We should not send more data than the network can take: *congestion control*



A Short History of TCP

- 1974: 3-way handshake
- 1978: IP and TCP split
- 1983: January 1st, ARPAnet switches to TCP/IP
- 1984: Nagle predicts congestion collapses
- 1986: Internet begins to suffer **congestion collapses**
 - LBL to Berkeley drops from 32Kbps to 40bps
- 1987/8: Van Jacobson fixes TCP, publishes seminal paper*: (**TCP Tahoe**)
- 1990: Fast transmit and fast recovery added (**TCP Reno**)



* Van Jacobson. Congestion avoidance and control. SIGCOMM '88

Congestion Collapse

Nagle, rfc896, 1984

- **Mid 1980's. Problem with the protocol *implementations*, not the protocol!**
- **What was happening?**
 - Load on the network → buffers at routers fill up → round trip time increases
- **If close to capacity, and, e.g., a large flow arrives suddenly...**
 - RTT estimates become too short
 - Lots of retransmissions → increase in queue size
 - Eventually many drops happen (full queues)
 - Fraction of useful packets (not copies) decreases



TCP Congestion Control

- **3 Key Challenges**

- Determining the available capacity in the first place
- Adjusting to changes in the available capacity
- Sharing capacity between flows

- **Idea**

- Each source determines network capacity for itself
- Rate is determined by window size
- Uses implicit feedback (drops, delay)
- ACKs pace transmission (self-clocking)



Dealing with Congestion

- **TCP keeps *congestion* and *flow control* windows**
 - Max packets in flight is lesser of two
- **Sending rate: $\sim \text{Window}/\text{RTT}$**
- **The key here is how to set the congestion window to respond to congestion signals**



Dealing with Congestion

- **Assume losses are due to congestion**
- **After a loss, reduce congestion window**
 - How much to reduce?
- **Idea: conservation of packets at equilibrium**
 - Want to keep roughly the same number of packets in network
 - Analogy with water in fixed-size pipe
 - Put new packet into network when one exits



How much to reduce window?

- **Crude model of the network**
 - Let L_i be the load (# pkts) in the network at time i
 - If network uncongested, roughly constant $L_i = N$
- **What happens under congestion?**
 - Some fraction γ of packets can't exit the network
 - Now $L_i = N + \gamma L_{i-1}$, or $L_i \approx g^i L_0$
 - Exponential increase in congestion
- **Sources must decrease offered rate exponentially**
 - i.e, multiplicative decrease in window size
 - TCP chooses to cut window in half

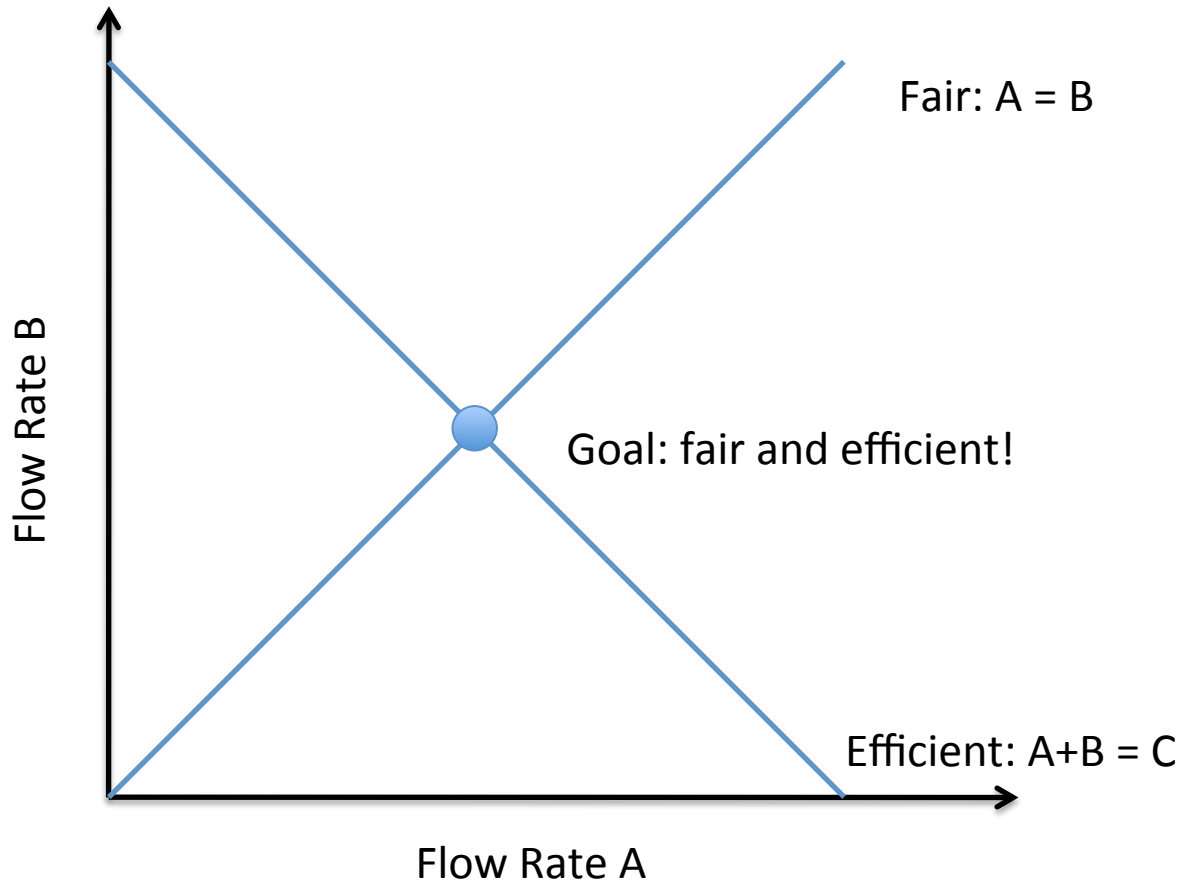


How to use extra capacity?

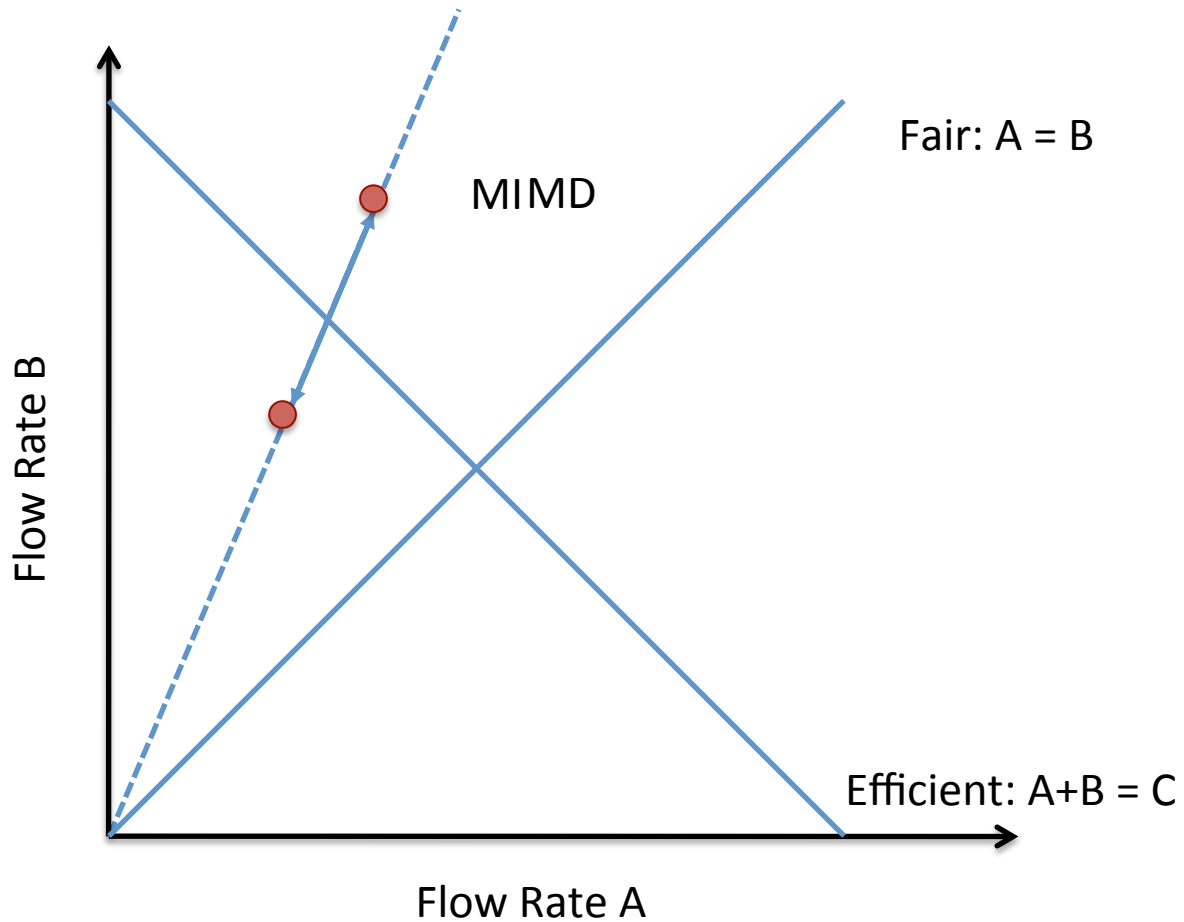
- **Network signals congestion, but says nothing of underutilization**
 - Senders constantly try to send faster, see if it works
 - So, increase window if no losses... By how much?
- **Multiplicative increase?**
 - Easier to saturate the network than to recover
 - Too fast, will lead to saturation, wild fluctuations
- **Additive increase?**
 - Won't saturate the network
 - Remember fairness (third challenge)?



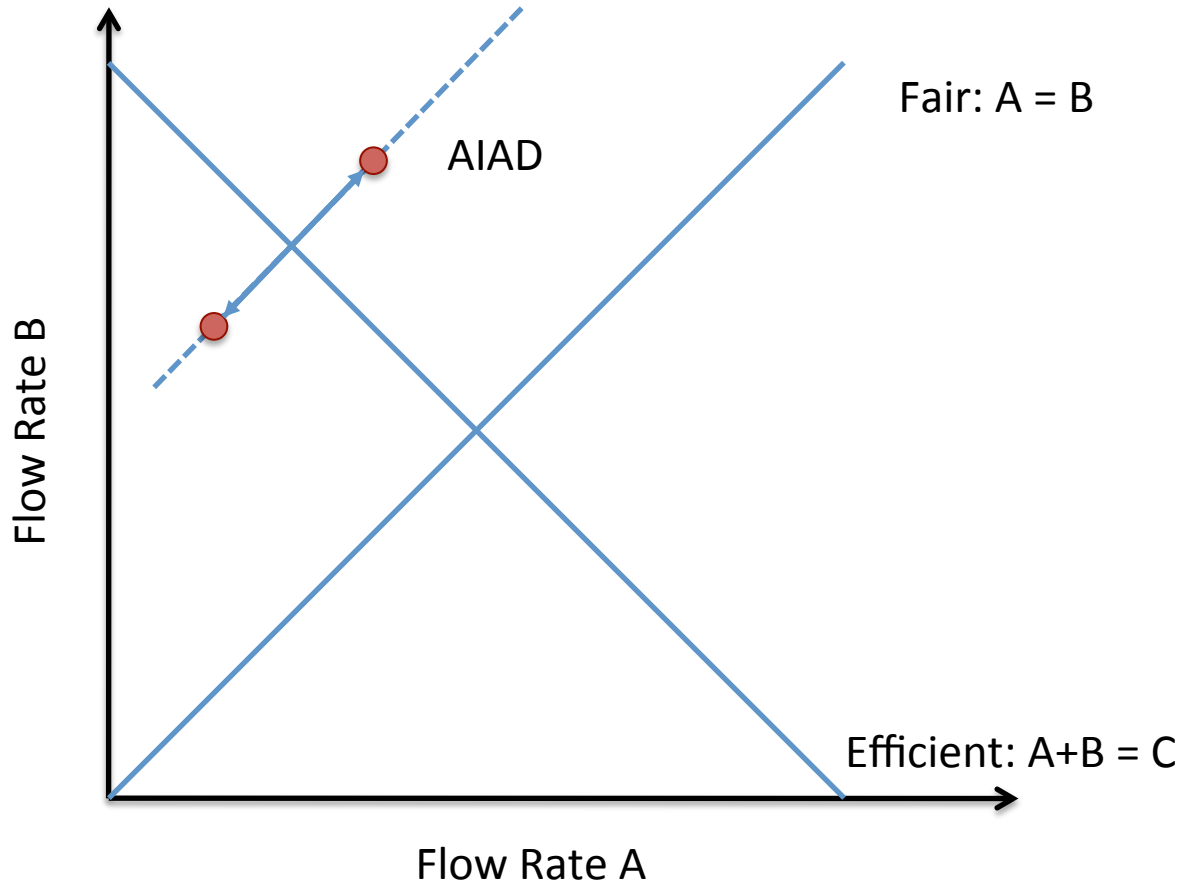
Chiu Jain Phase Plots



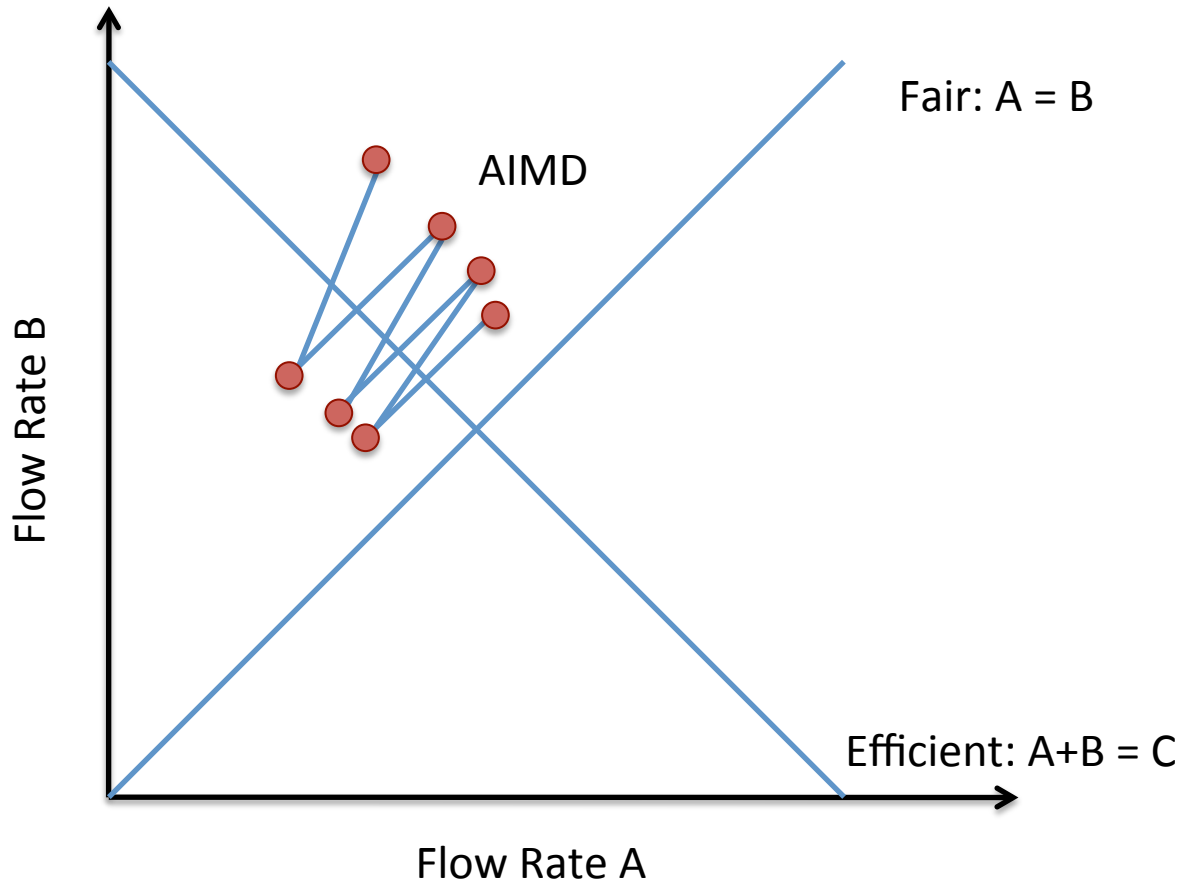
Chiu Jain Phase Plots



Chiu Jain Phase Plots



Chiu Jain Phase Plots



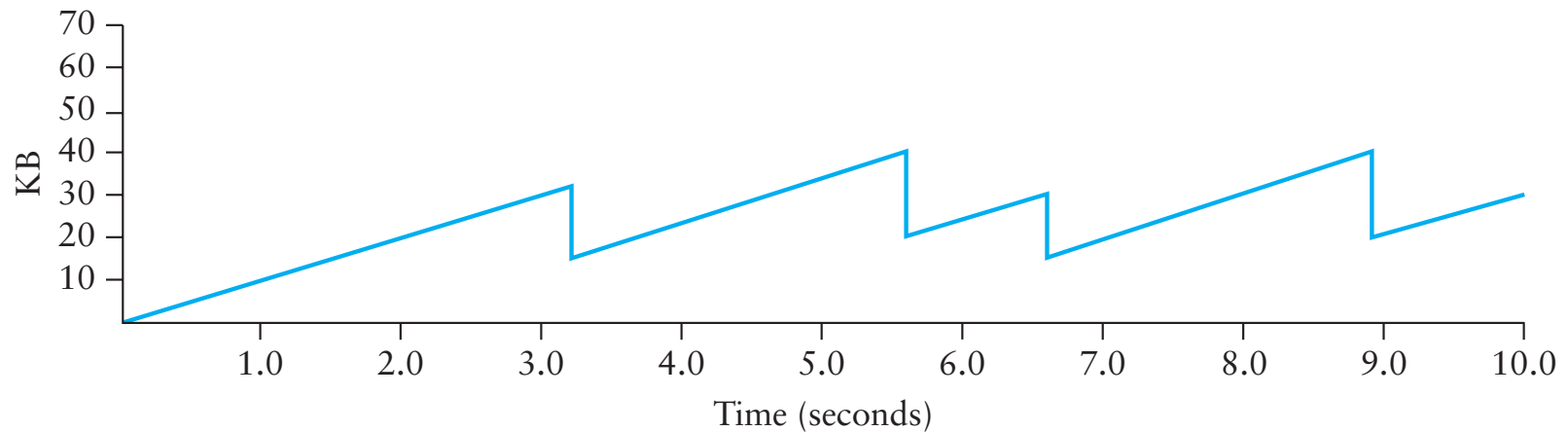
AIMD Implementation

- **In practice, send MSS-sized segments**
 - Let window size in bytes be w (a multiple of MSS)
- **Increase:**
 - After w bytes ACKed, could set $w = w + \text{MSS}$
 - Smoother to increment on each ACK
 - $w = w + \text{MSS} * \text{MSS}/w$
 - (receive w/MSS ACKs per RTT, increase by $\text{MSS}/(w/\text{MSS})$ for each)
- **Decrease:**
 - After a packet loss, $w = w/2$
 - But don't want $w < \text{MSS}$
 - So react differently to multiple consecutive losses
 - Back off exponentially (pause with no packets in flight)



AIMD Trace

- **AIMD produces sawtooth pattern of window size**
 - Always probing available bandwidth

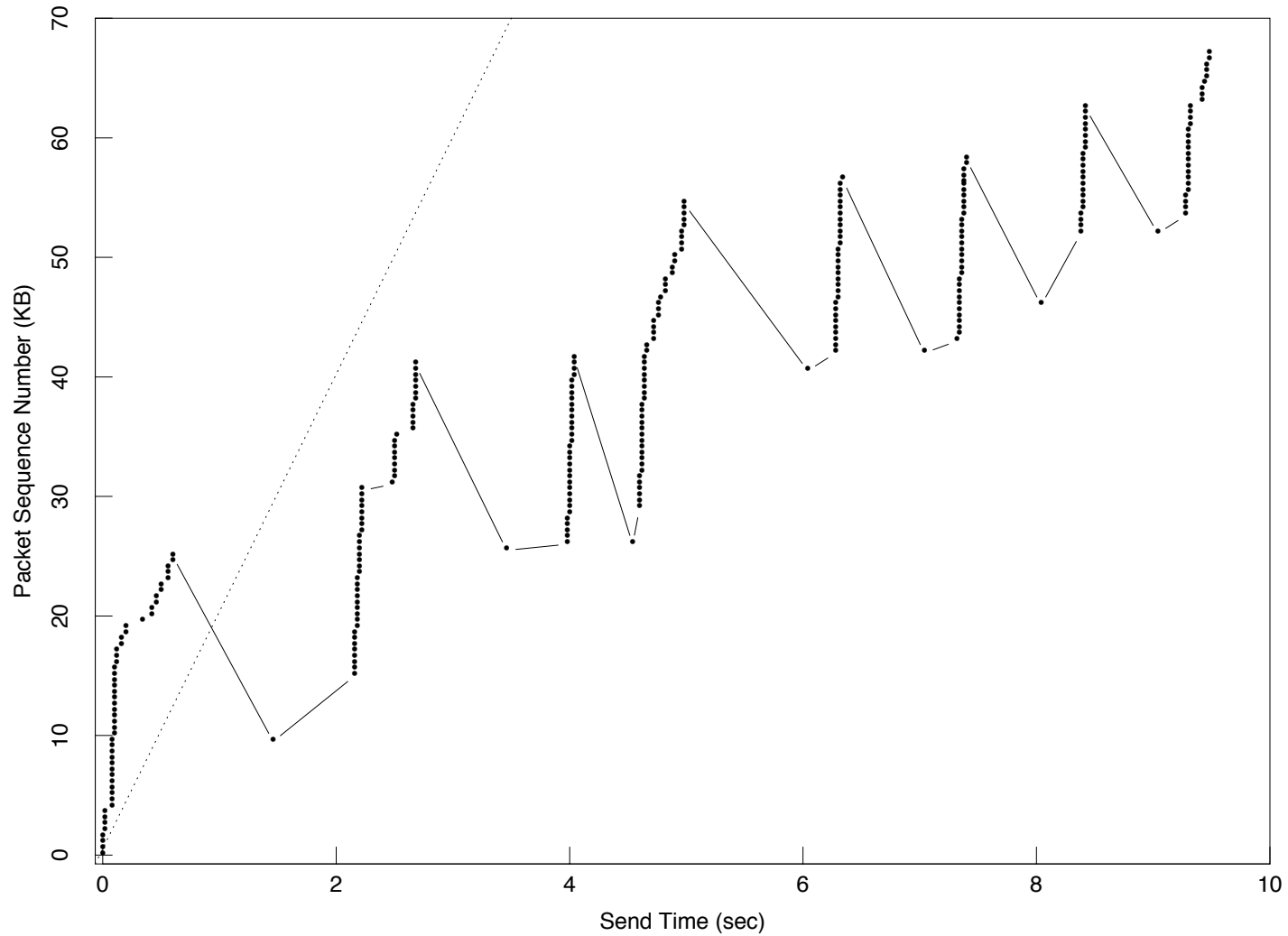


Starting Up

- **Before TCP Tahoe**
 - On connection, nodes send full (rcv) window of packets
 - Retransmit packet immediately after its timer expires
- **Result: window-sized bursts of packets in network**



Bursts of Packets

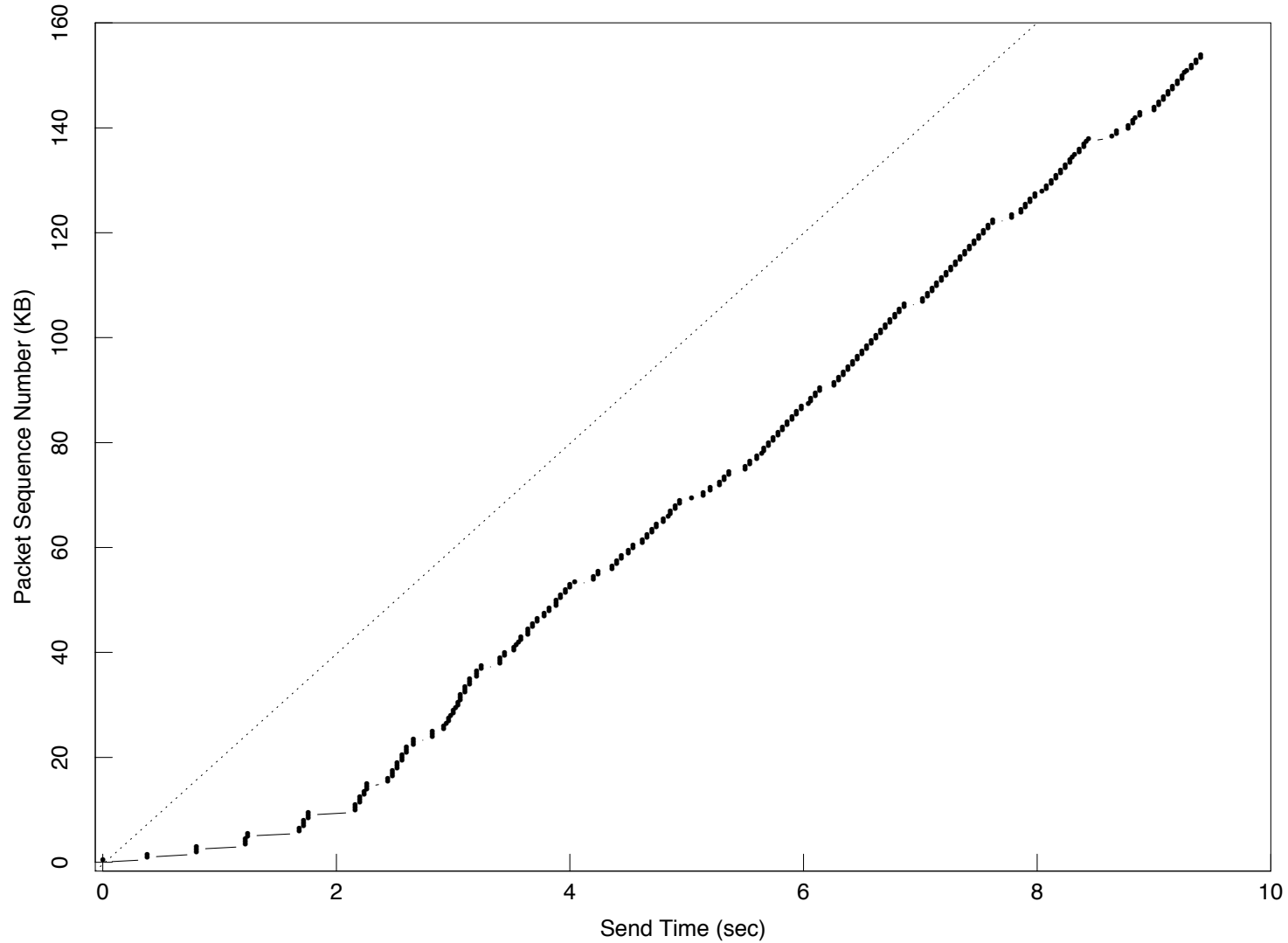


Determining Initial Capacity

- **Question: how do we set w initially?**
 - Should start at 1MSS (to avoid overloading the network)
 - Could increase additively until we hit congestion
 - May be too slow on fast network
- **Start by doubling w each RTT**
 - Then will dump at most one extra window into network
 - This is called *slow start*
- ***Slow start*, this sounds quite fast!**
 - In contrast to initial algorithm: sender would dump entire *flow control* window at once



Startup behavior with Slow Start



Slow start implementation

- **Let w be the size of the window in *bytes***
 - We have w/MSS segments per RTT
- **We are doubling w after each RTT**
 - We receive w/MSS ACKs each RTT
 - So we can set $w = w + \text{MSS}$ on every ACK
- **At some point we hit the network limit.**
 - Experience loss
 - We are at most one window size above the limit
 - Remember window size (ssthresh) and reduce window



Putting it together

- **TCP has two states: Slow Start (SS) and Congestion Avoidance (CA)**
- **A window size threshold governs the state transition**
 - Window \leq threshold: SS
 - Window $>$ threshold: congestion avoidance
- **States differ in how they respond to ACKs**
 - Slow start: $w = w + \text{MSS}$
 - Congestion Avoidance: $w = w + \text{MSS}^2/w$ (1 MSS per RTT)
- **On loss event: set $w = 1$, slow start**

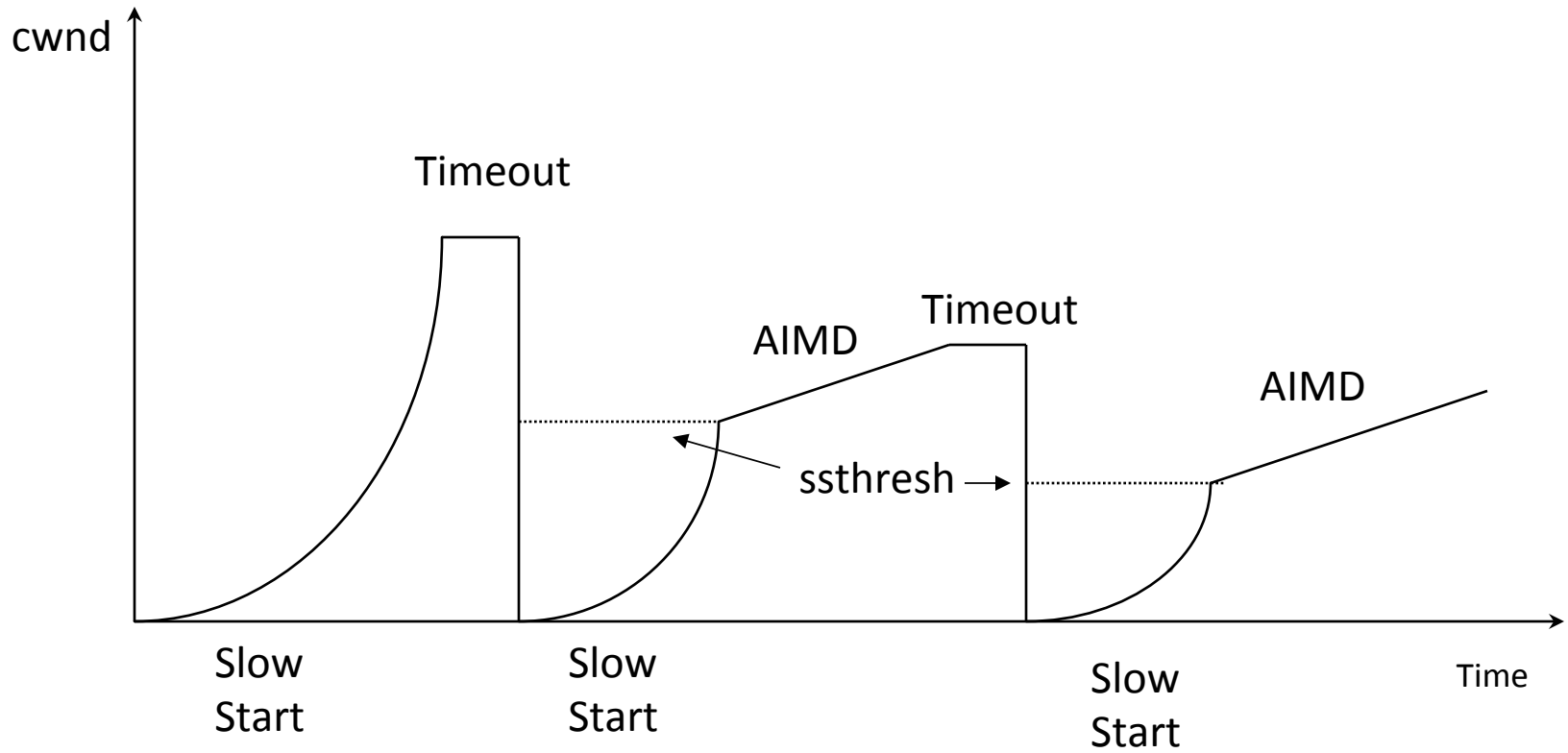


How to Detect Loss

- **Timeout**
- **Any other way?**
 - Gap in sequence numbers at receiver
 - Receiver uses cumulative ACKs: drops => duplicate ACKs
- **3 Duplicate ACKs considered loss**
- **Which one is worse?**



Putting it all together



RTT

- We want an estimate of RTT so we can know a packet was likely lost, and not just delayed
- **Key for correct operation**
- **Challenge: RTT can be highly variable**
 - Both at long and short time scales!
- **Both average and variance increase a lot with load**
- **Solution**
 - Use exponentially weighted moving average (EWMA)
 - Estimate deviation as well as expected value
 - Assume packet is lost when time is well beyond reasonable deviation



Originally

- **EstRTT = $(1 - \alpha) \times \text{EstRTT} + \alpha \times \text{SampleRTT}$**
- **Timeout = $2 \times \text{EstRTT}$**
- **Problem 1:**
 - in case of retransmission, ack corresponds to which send?
 - Solution: only sample for segments with no retransmission
- **Problem 2:**
 - does not take variance into account: too aggressive when there is more load!

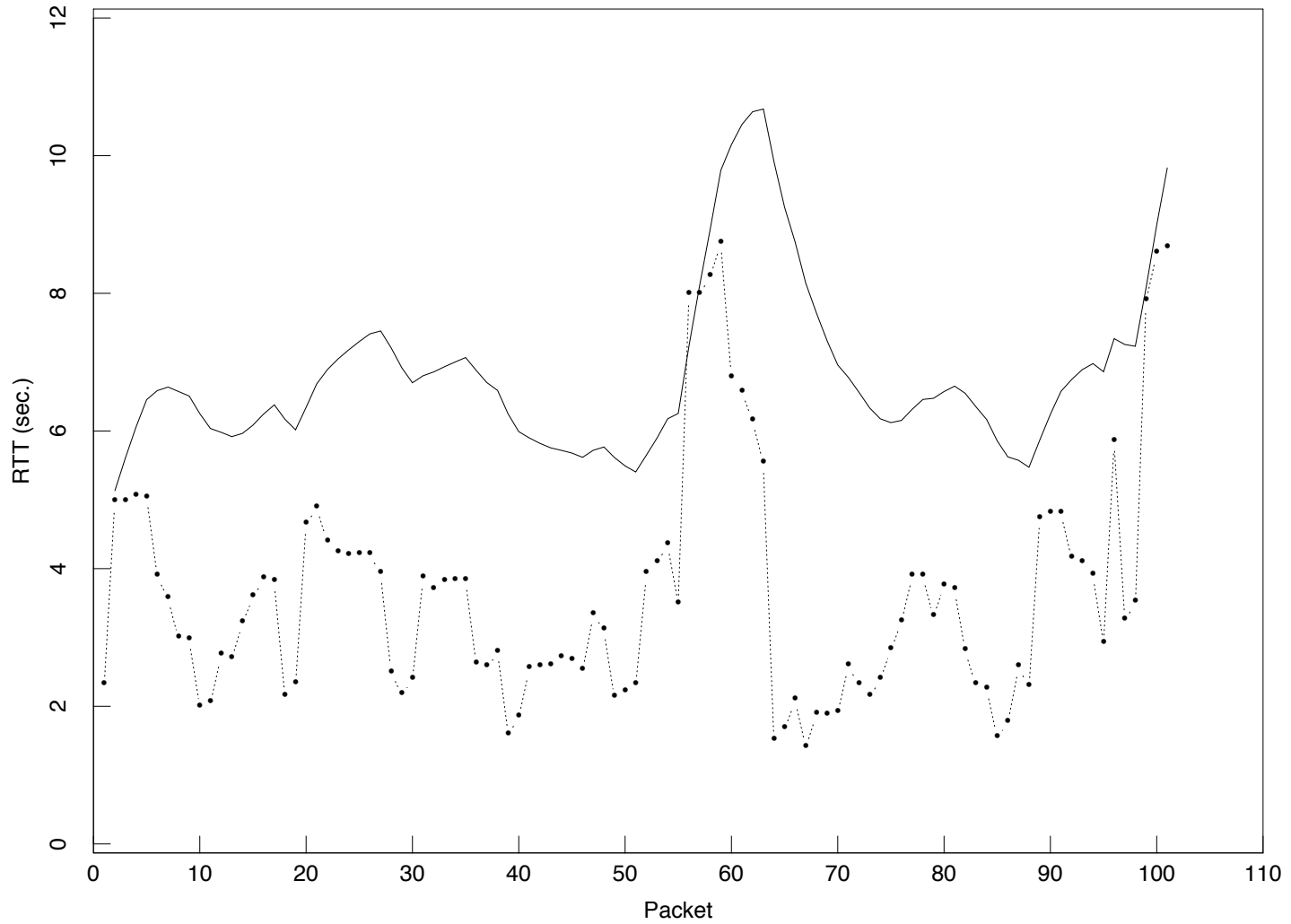


Jacobson/Karels Algorithm (Tahoe)

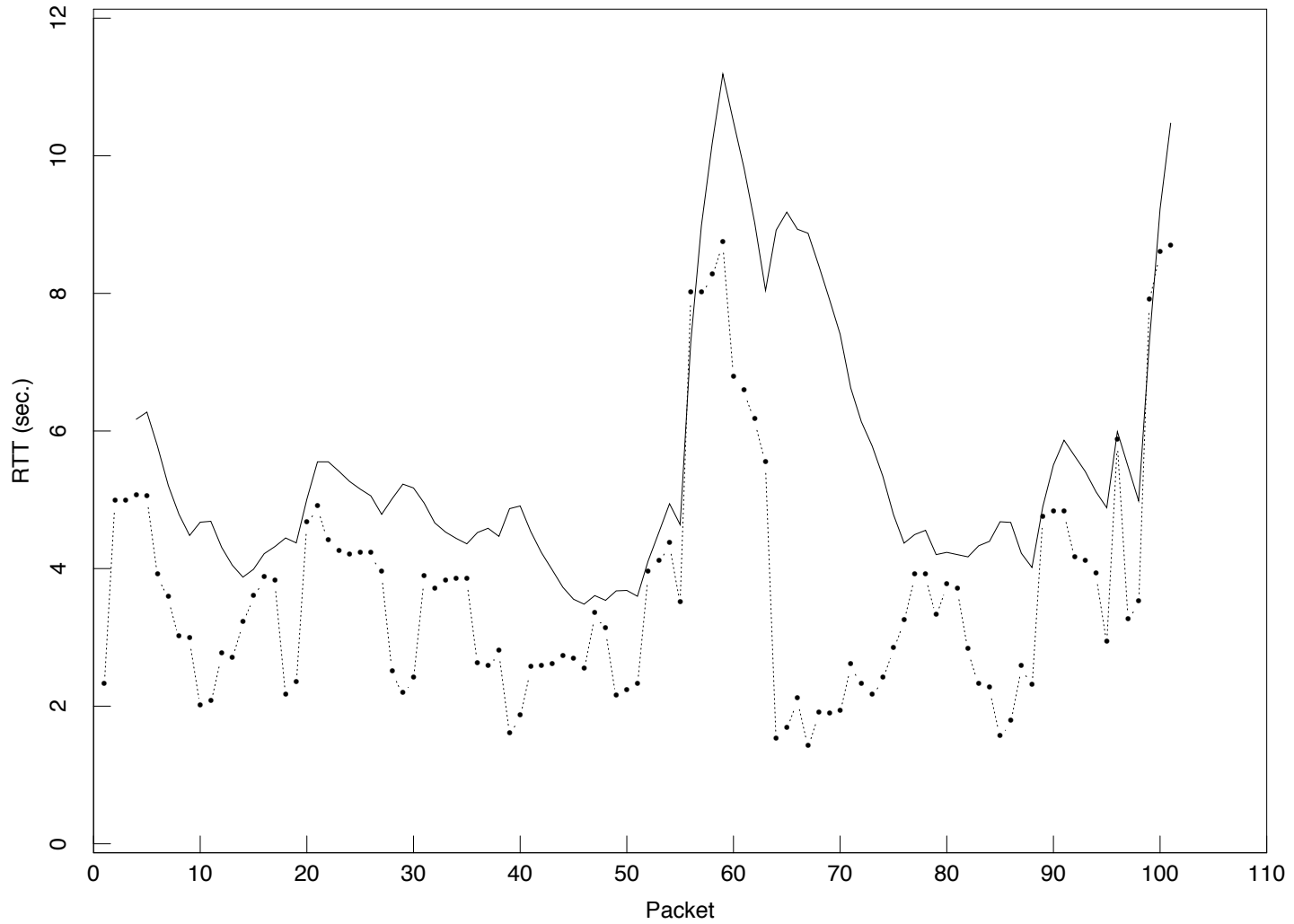
- **$\text{EstRTT} = (1 - \alpha) \times \text{EstRTT} + \alpha \times \text{SampleRTT}$**
 - Recommended α is 0.125
- **$\text{DevRTT} = (1 - \beta) \times \text{DevRTT} + \beta | \text{SampleRTT} - \text{EstRTT} |$**
 - Recommended β is 0.25
- **$\text{Timeout} = \text{EstRTT} + 4 \text{DevRTT}$**
- **For successive retransmissions: use exponential backoff**



Old RTT Estimation



Tahoe RTT Estimation

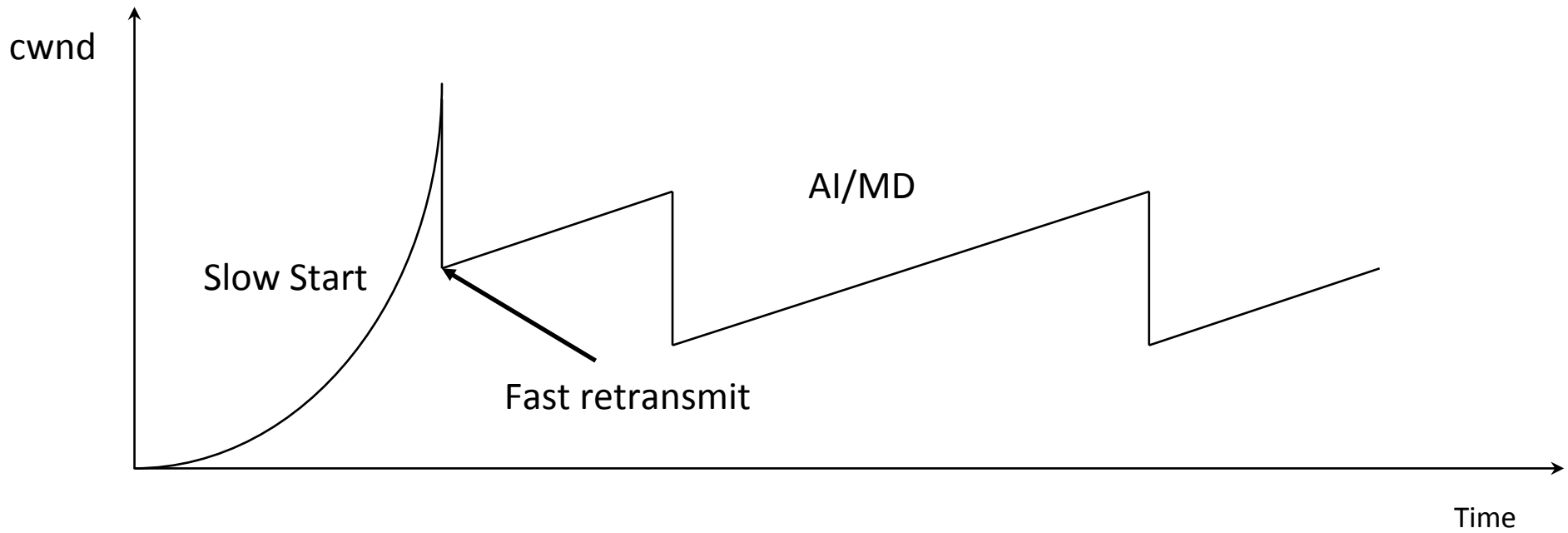


Slow start every time?!

- **Losses have large effect on throughput**
- **Fast Recovery (TCP Reno)**
 - Same as TCP Tahoe on Timeout: $w = 1$, slow start
 - On triple duplicate ACKs: $w = w/2$
 - Retransmit missing segment (fast retransmit)
 - Stay in Congestion Avoidance mode



Fast Recovery and Fast Retransmit



3 Challenges Revisited

- **Determining the available capacity in the first place**
 - Exponential increase in congestion window
- **Adjusting to changes in the available capacity**
 - Slow probing, AIMD
- **Sharing capacity between flows**
 - AIMD
- **Detecting Congestion**
 - Timeout based on RTT
 - Triple duplicate acknowledgments
- **Fast retransmit/Fast recovery**
 - Reduces slow starts, timeouts



Next Class

- **More Congestion Control fun**
- **Cheating on TCP**
- **TCP on extreme conditions**
- **TCP Friendliness**
- **TCP Future**

