# Congestion Control

COS 316: Principles of Computer System Design

Lecture 17

Amit Levy & Wyatt Lloyd

[Slides adapted from Michael J Freedman's]

# Concurrency

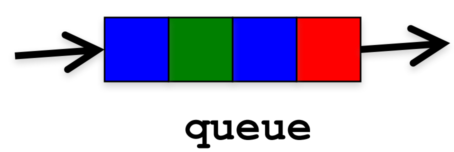
- Multiple things happening at the same time

- Primary benefit is better performance
  - Do more work in the same amount of time
  - Complete fixed amount work in less time
  - Better utilize resources

- Primary cost is complexity
  - Hard to reason about
  - Hard to get right
  - (Systems deal with it, not applications, … to some extent)

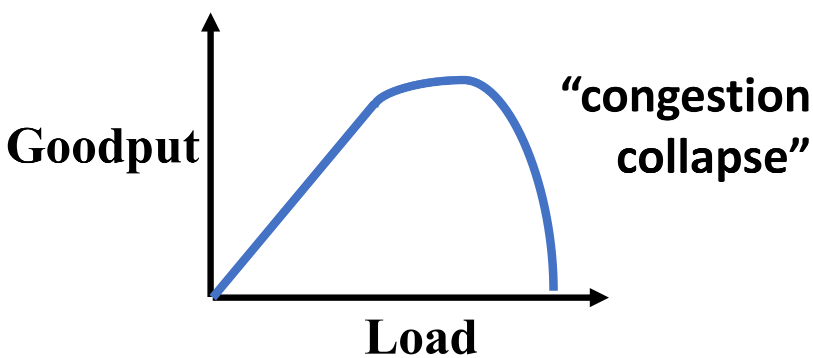
# Congestion

- Best-effort network does not "block" calls
  - So, they can easily become overloaded
  - Congestion == "Load higher than capacity"
- Examples of congestion
  - Link layer: Ethernet frame collisions
  - Network layer: full IP packet buffers
- Excess packets are simply dropped
  - And the sender can simply retransmit



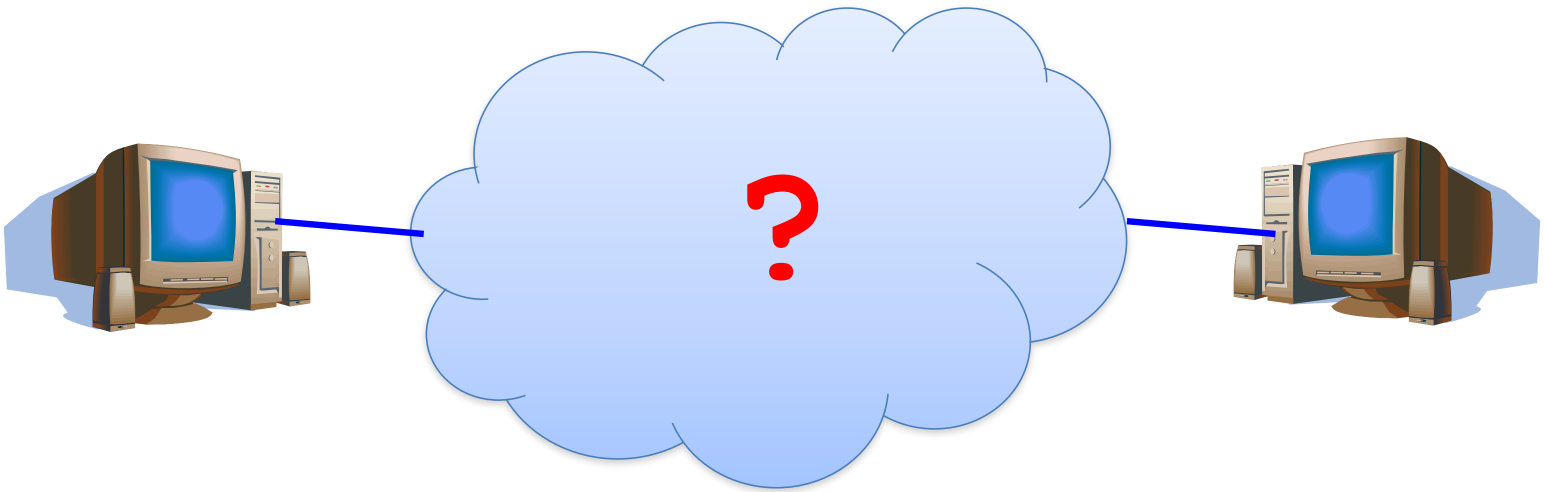
# Congestion Collapse

- Easily leads to *congestion collapse*
  - Senders retransmit the lost packets
  - Leading to even *greater* load
  - … and even *more* packet loss




**Increase in load that results in a *decrease* in useful work done.**

# Detect and Respond to Congestion




- What does the end host see?
- What can the end host change?

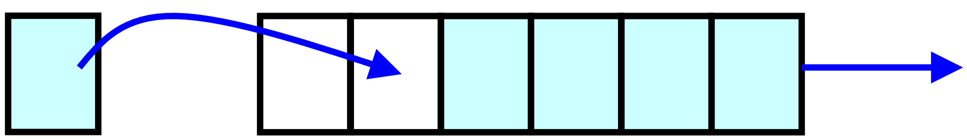
# Detecting Congestion

- Link layer
  - Carrier sense multiple access
  - Seeing your own frame collide with others
- Network layer
  - Observing end-to-end performance
  - Packet delay or loss over the path

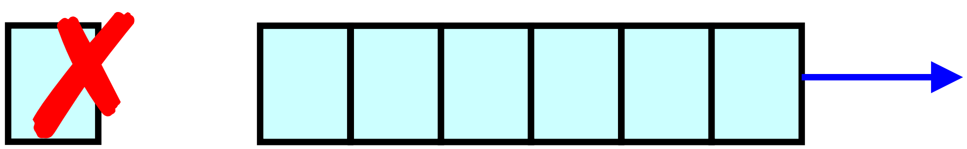
# TCP Congestion Control

# Congestion in a Drop-Tail FIFO Queue

- Access to the bandwidth: first-in first-out queue
  - Packets transmitted in the order they arrive



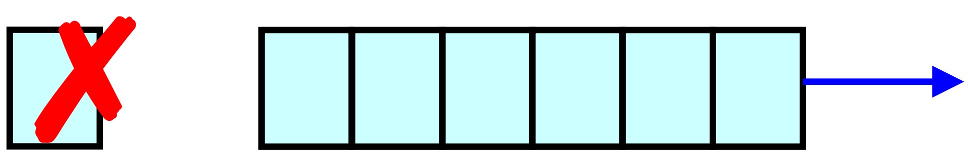
- Access to the buffer space: drop-tail queuing
  – If the queue is full, drop the incoming packet



# How it Looks to the End Host

- Delay: Packet experiences high delay
- Loss: Packet gets dropped along path

- How does TCP sender learn this?
  - Delay: Round-trip time estimate
  - Loss: Timeout and/or duplicate acknowledgments



# TCP Congestion Window

- Each TCP sender maintains a congestion window
  - Max number of bytes to have in transit (not yet ACK'd)
- Adapting the congestion window
  - Decrease upon losing a packet: backing off
  - Increase upon success: optimistically exploring
  - Always struggling to find right transfer rate
- Tradeoff
  - Pro: avoids needing explicit network feedback
  - Con: continually under- and over-shoots "right" rate

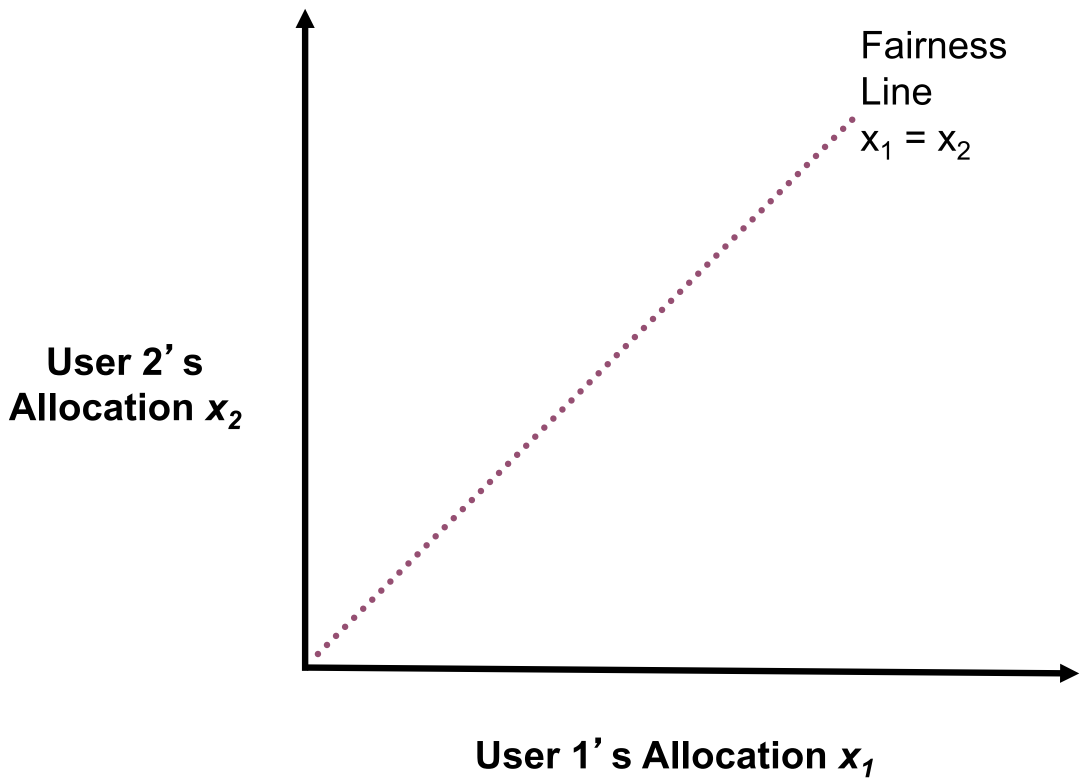
# Additive Increase, Multiplicative Decrease

- How much to adapt?
  - Additive increase: On success of last window of data, increase window by 1 Max Segment Size (MSS)
  - Multiplicative decrease: On loss of packet, divide congestion window in half

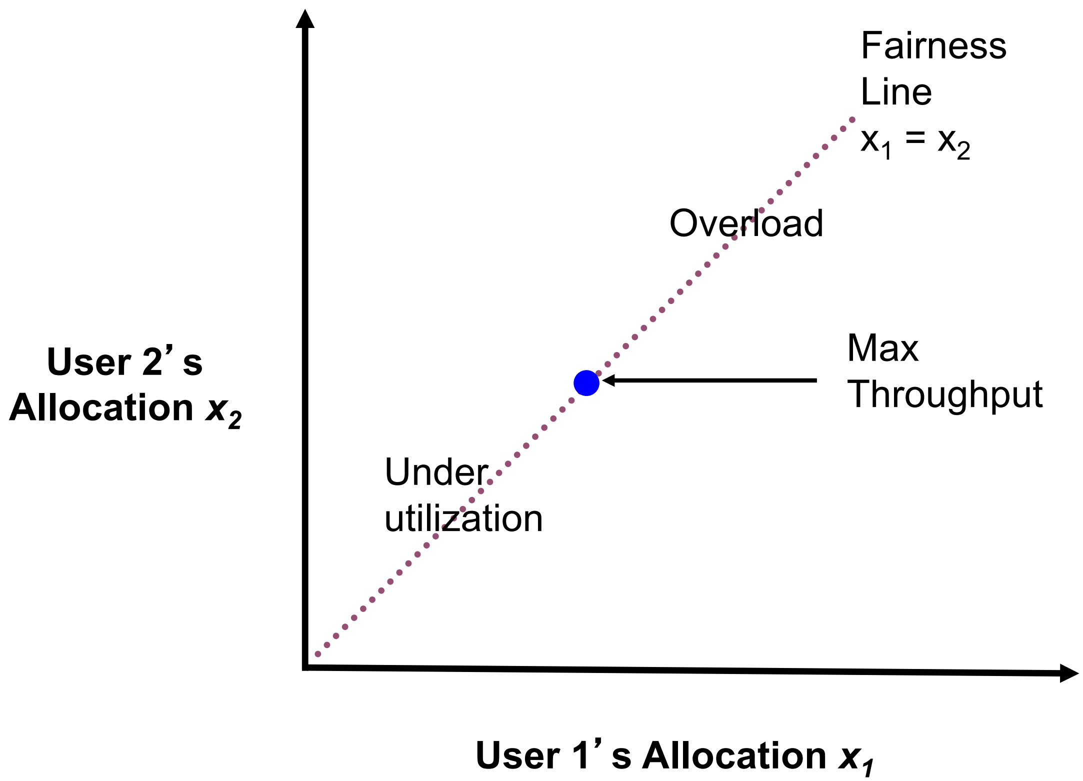
- Much quicker to slow down than speed up?
  - Over-sized windows (causing loss) are much worse than under-sized windows (causing lower throughput)

# TCP seeks ”Fairness”

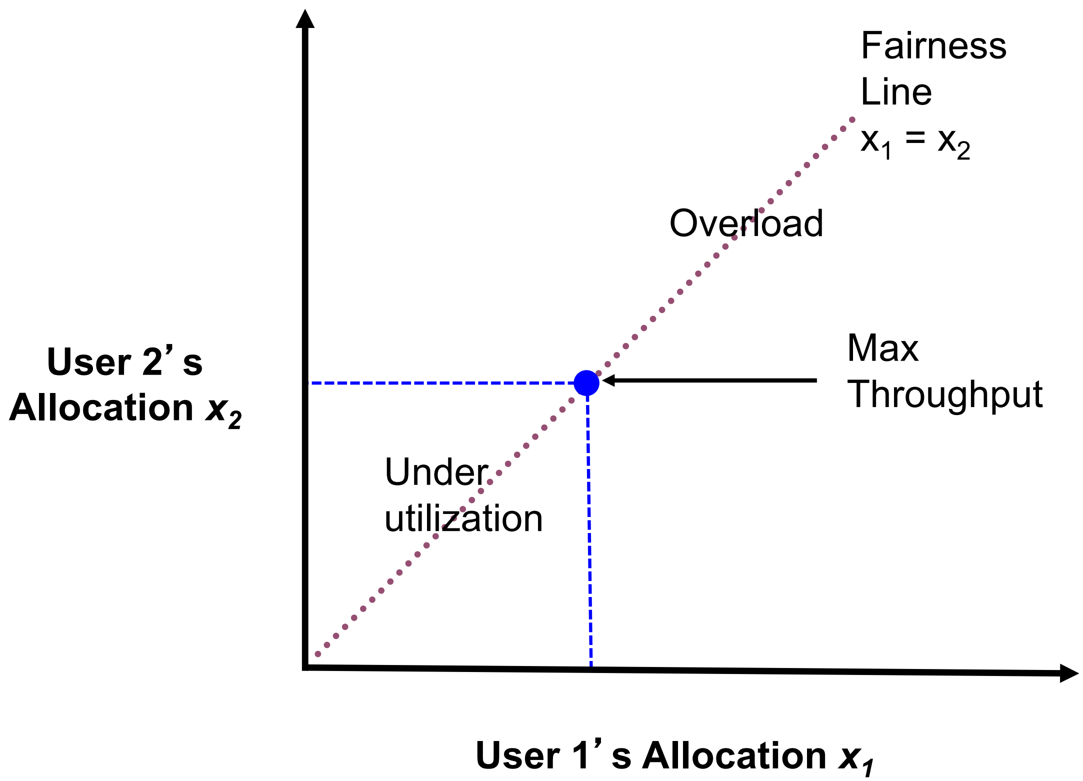
# Phase Plots


Fairness
Line
$x_1 = x_2$
**User 2's**
**Allocation $x_2$**
**User 1's Allocation $x_1$**

# Phase Plots

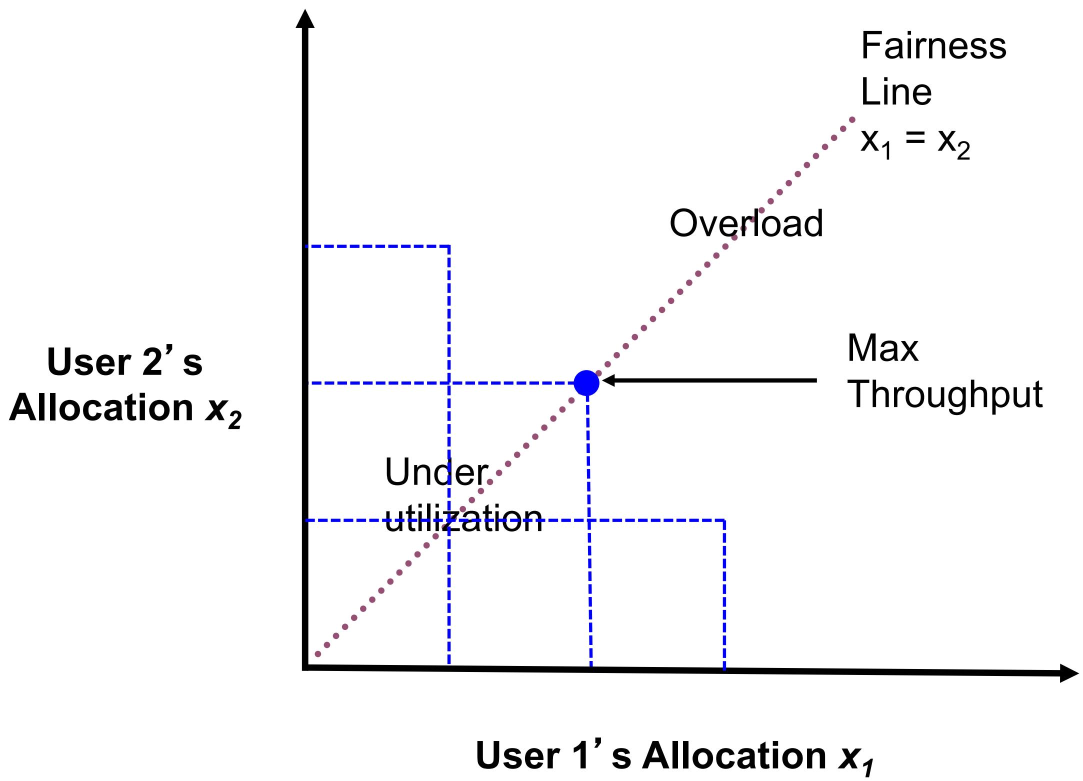

Fairness
Line
$x_1 = x_2$
Overload
Max
Throughput
Under
utilization
User 2's
Allocation $x_2$
User 1's Allocation $x_1$

# Phase Plots

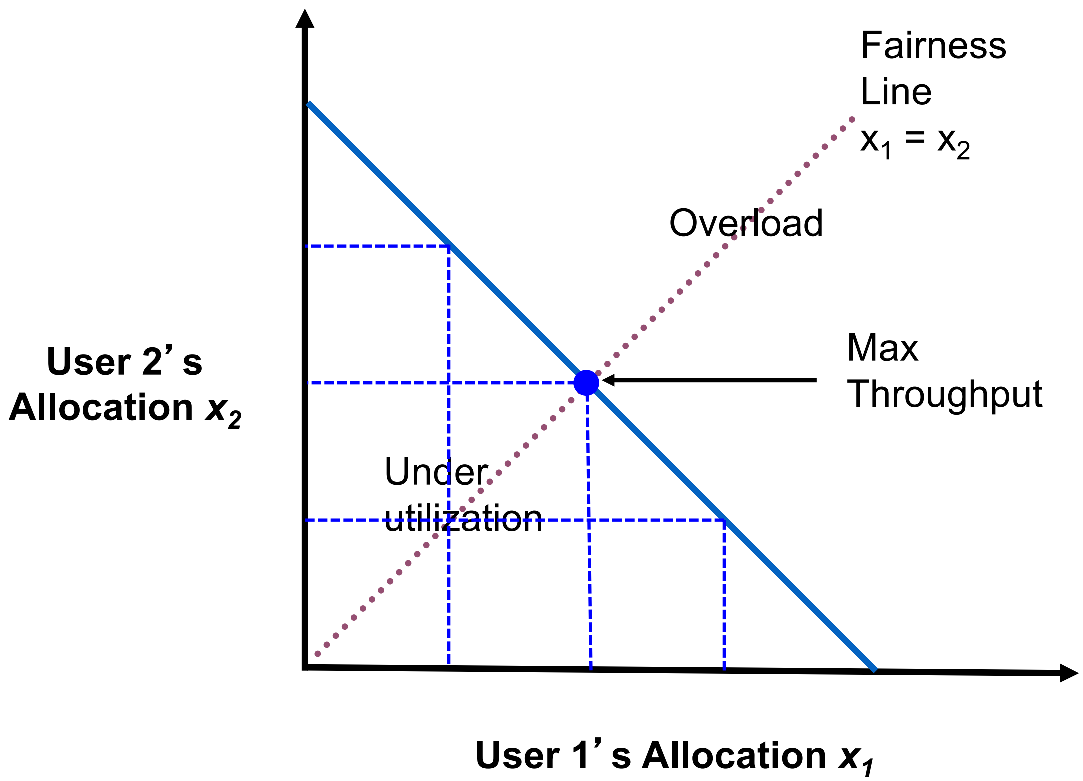


Fairness
Line
$x_1 = x_2$
Overload
Max
Throughput
User 2's
Allocation $x_2$
Under
utilization
User 1's Allocation $x_1$

# Phase Plots

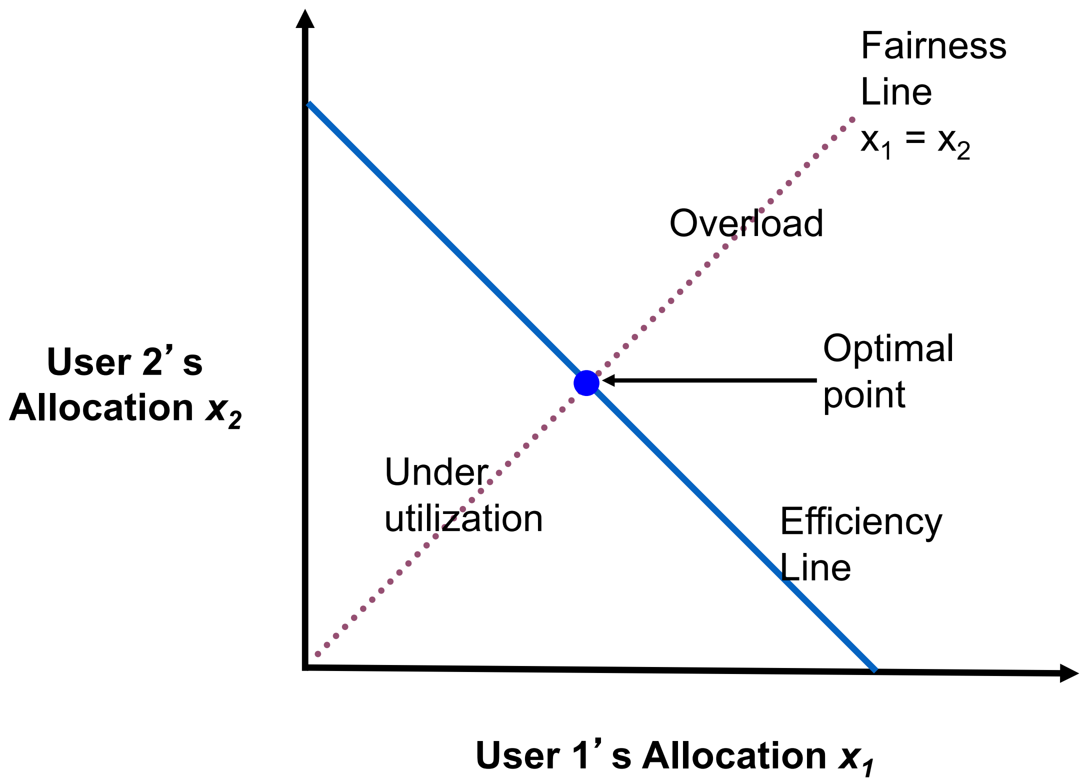

Fairness
Line
$x_1 = x_2$
Overload
Max
Throughput
User 2's
Allocation $x_2$
Under
utilization
User 1's Allocation $x_1$

# Phase Plots



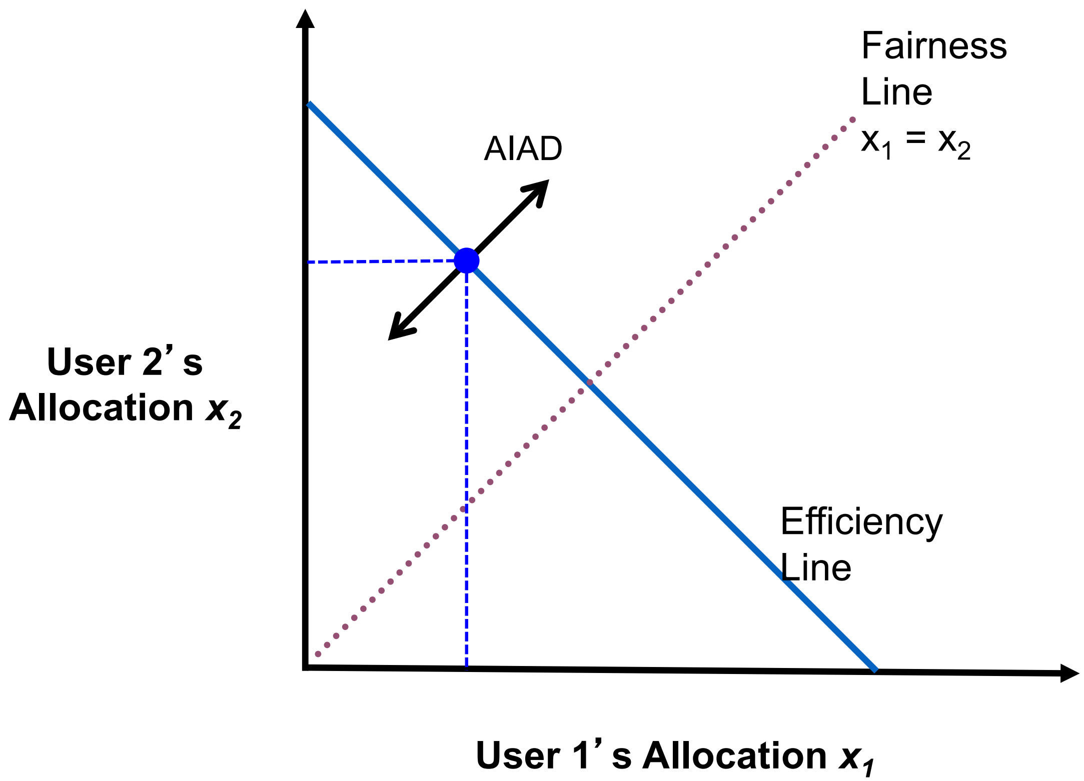
Fairness Line $x_1 = x_2$
Overload
Max Throughput
Under utilization
User 2's Allocation $x_2$
User 1's Allocation $x_1$

# Phase Plots

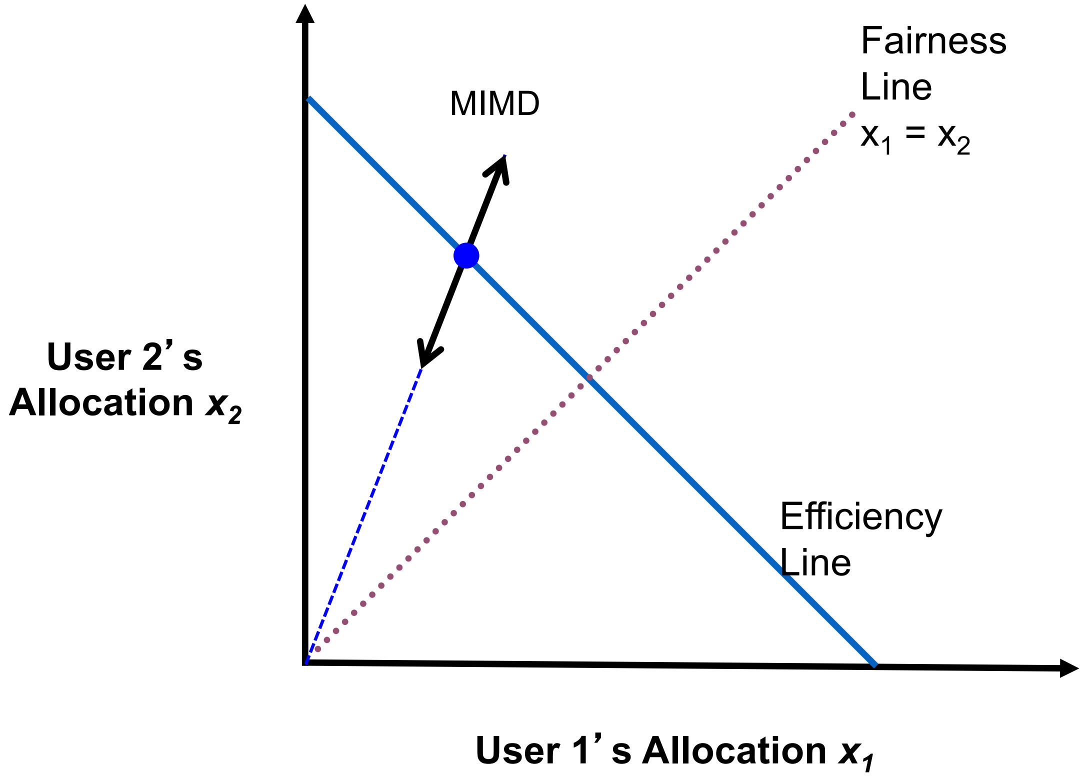


Fairness
Line
$x_1 = x_2$
Overload
Optimal
point
User 2's
Allocation $x_2$
Under
utilization
Efficiency
Line
User 1's Allocation $x_1$

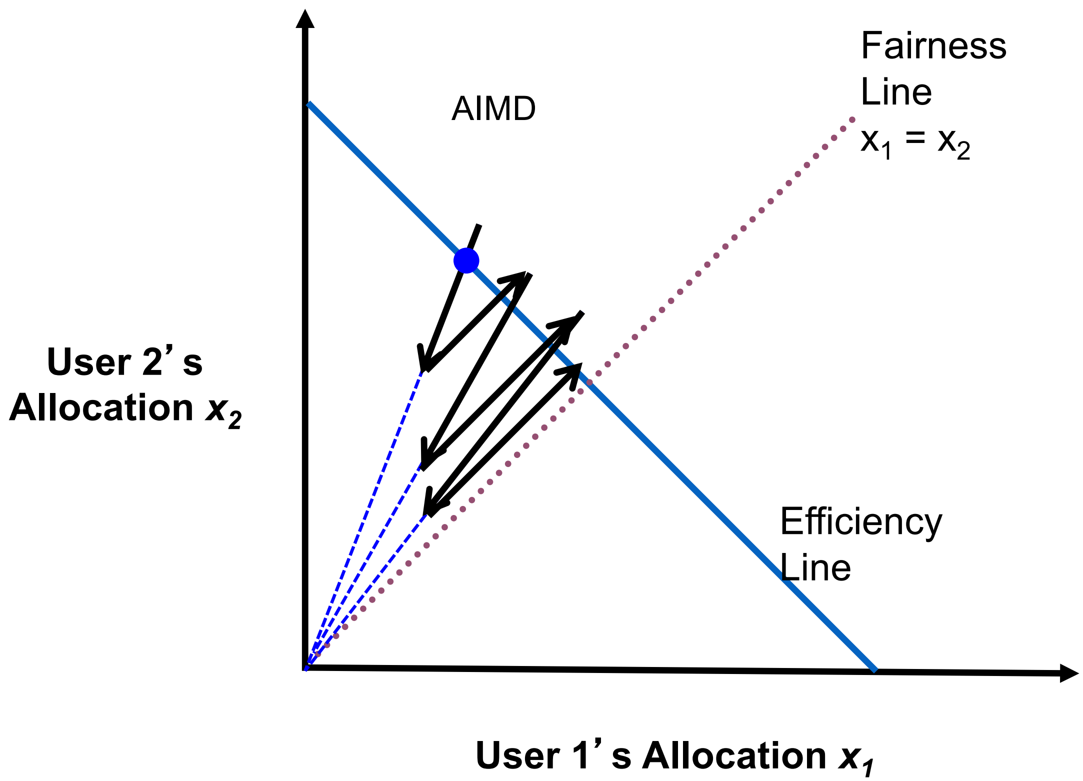
# Additive Increase/Decrease



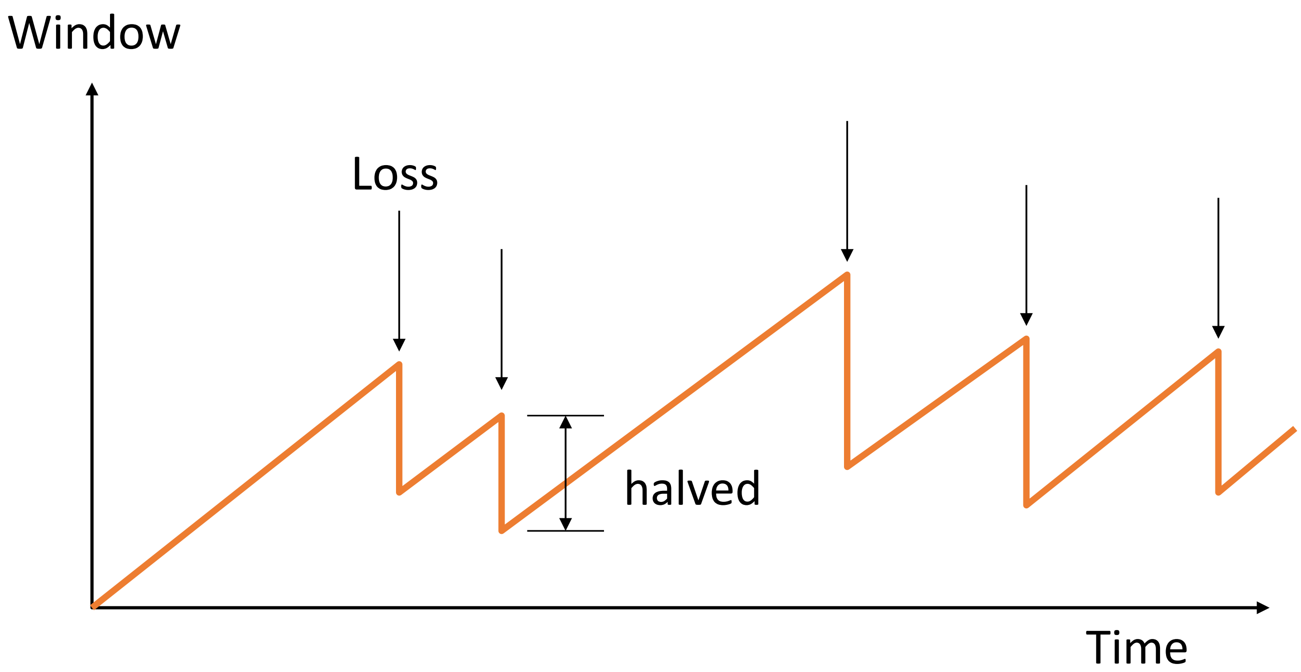
# Multiplicative Increase/Decrease



# Additive Increase / Multiplicative Decrease



# Leads to the TCP “Sawtooth”



# Receiver Window vs. Congestion Window

- Flow control
  - Keep a *fast sender* from overwhelming *a slow receiver*
- Congestion control
  - Keep a *set of senders* from overloading the *network*

- Different concepts, but similar mechanisms
  - TCP flow control: receiver window
  - TCP congestion control: congestion window
  - Sender TCP window =

    min { congestion window, receiver window }

# Sources of Poor TCP Performance

- The below conditions *may* primarily result in:

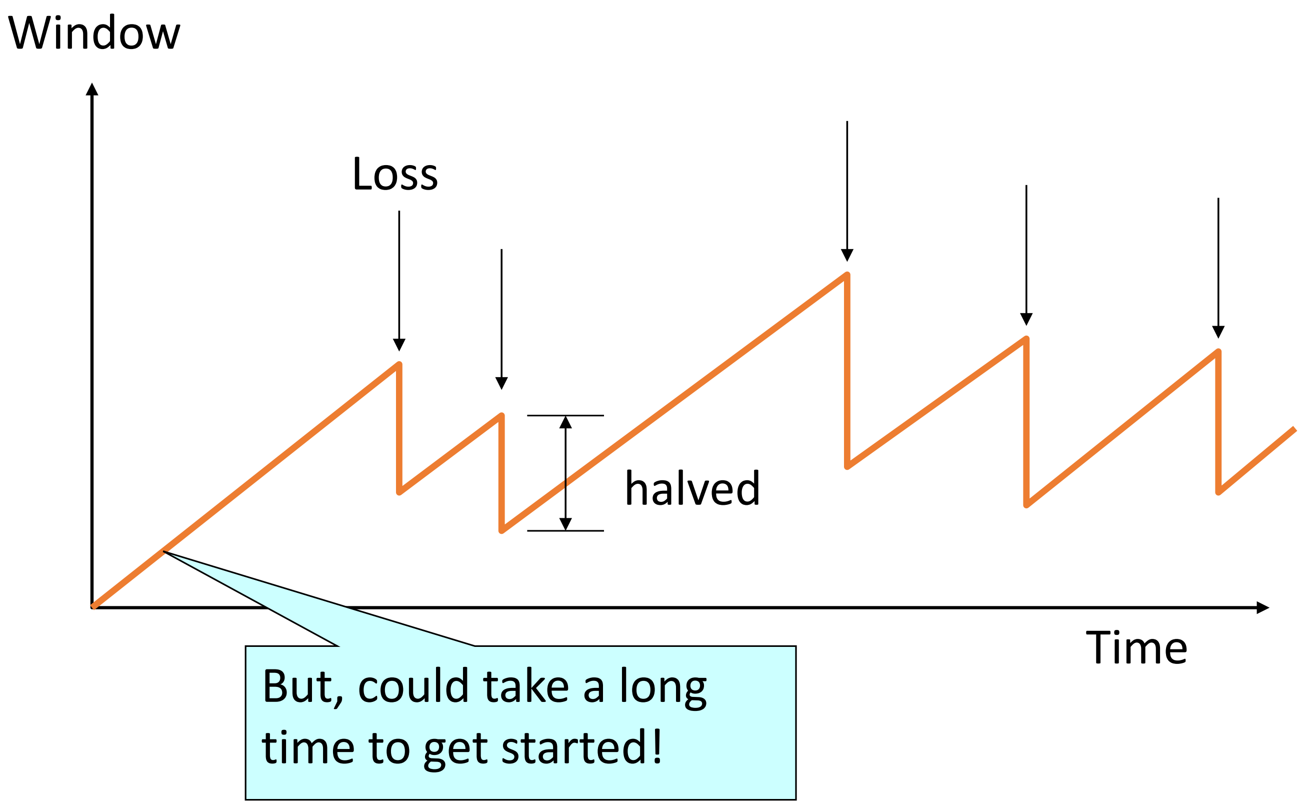
(A) Higher packet latency (B) Greater loss (C) Lower throughput

1. Larger buffers in routers
2. Smaller buffers in routers
3. Smaller buffers on end-hosts
4. Slow application receivers

# Starting a New Flow

# How Should a New Flow Start?

**Start slow (a small CWND) to avoid overloading network**

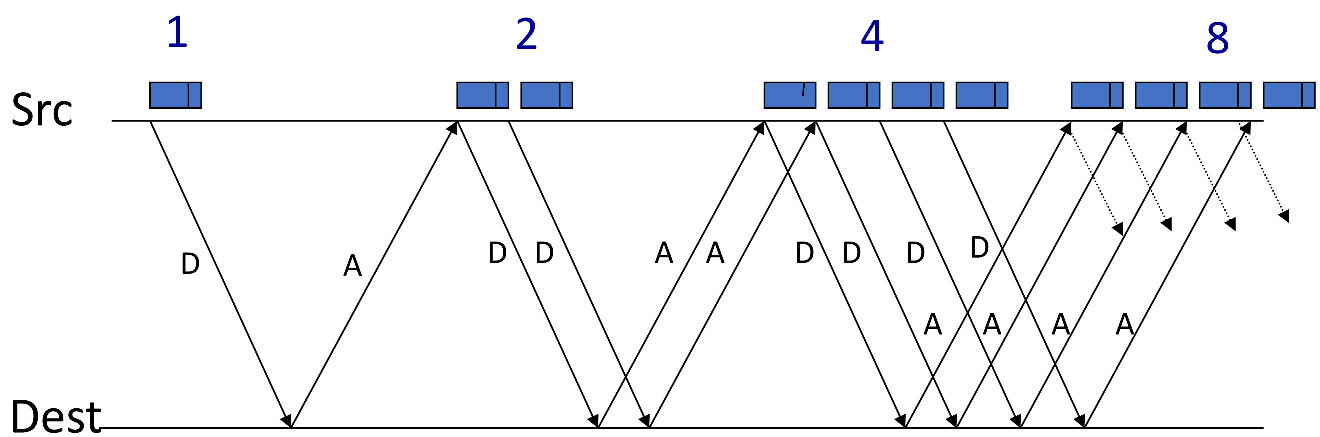


# “Slow Start” Phase

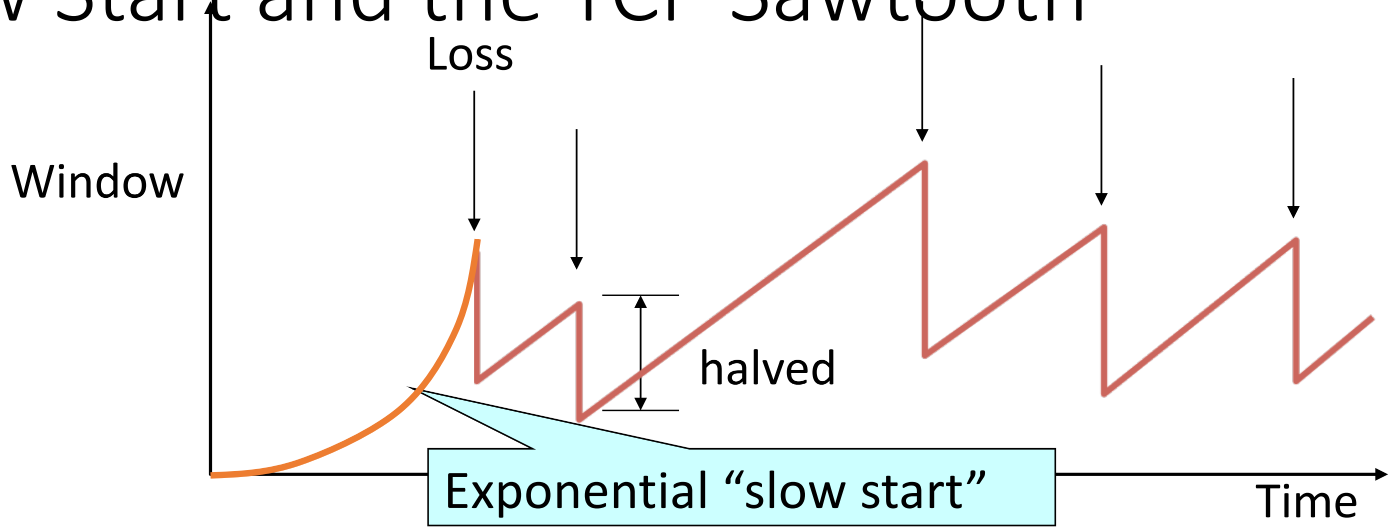
- Start with a small congestion window
  - Initially, CWND is 1 MSS
  - So, initial sending rate is MSS / RTT
- Could be pretty wasteful
  - Might be much less than actual bandwidth
  - Linear increase takes a long time to accelerate
- Slow-start phase (really “fast start”)
  - Sender starts at a slow rate (hence the name)
  - … but increases rate exponentially until the first loss

# Slow Start in Action

Double CWND per round-trip time



# Slow Start and the TCP Sawtooth

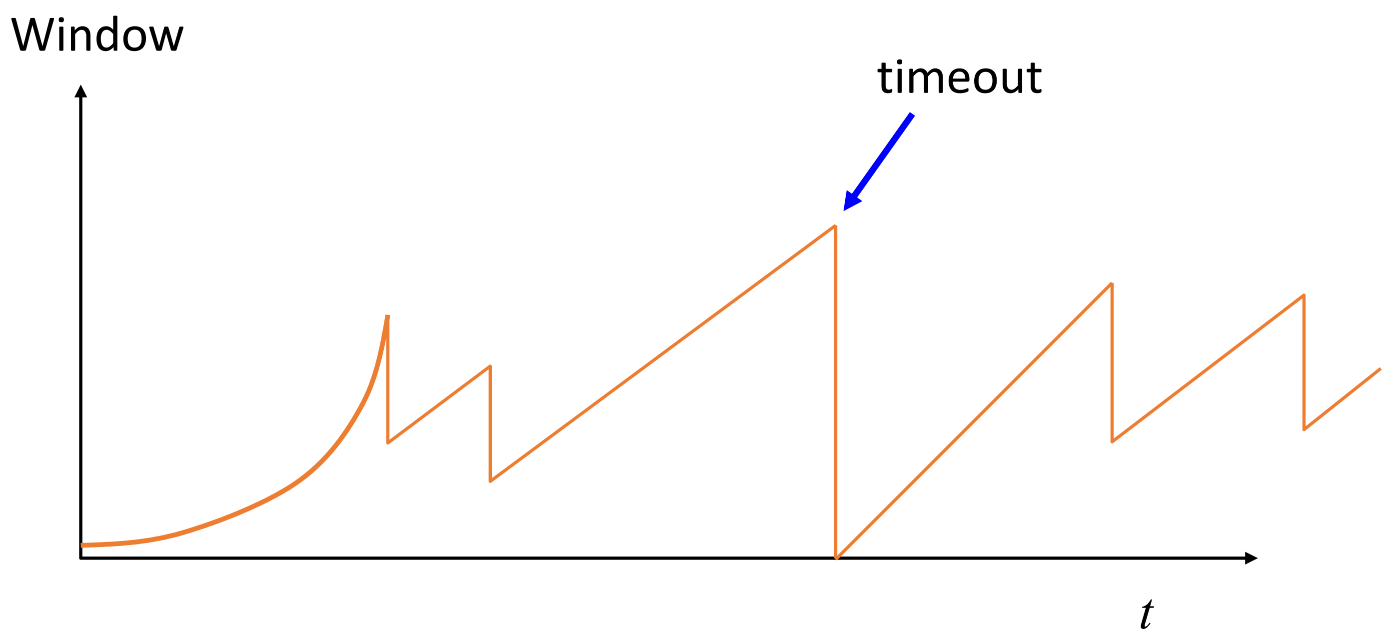



- TCP originally had *no* congestion control
  - Source would start by sending entire receiver window
  - Led to congestion collapse!
  - "Slow start" is, comparatively, slower

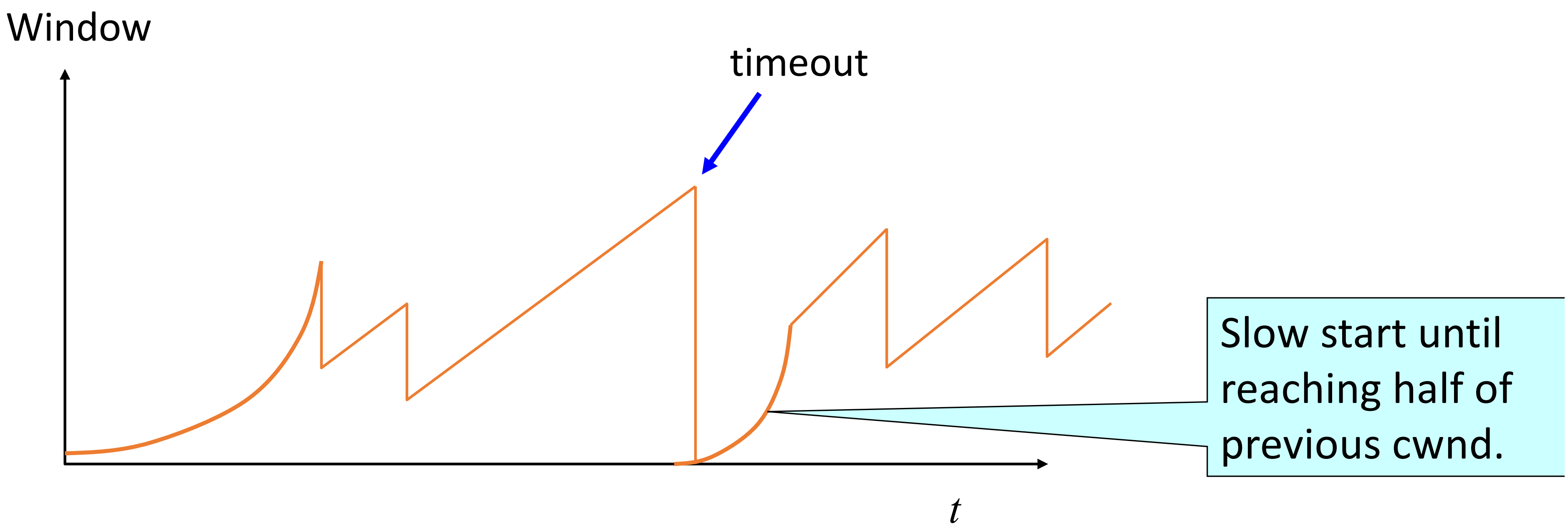
# Two Kinds of Loss in TCP

- Timeout vs. Triple Duplicate ACK
  - Which suggests network is in worse shape?
- Timeout
  - If entire window was lost, buffers may be full
  - ...blasting entire CWND would cause another burst
  - ...be aggressive: start over with a low CWND
- Triple duplicate ACK
  - Might be do to bit errors, or “micro” congestion
  - ...react less aggressively (halve CWND)

# Repeating Slow Start After Timeout



# Repeating Slow Start After Timeout




Slow-start restart: Go back to CWND of 1, but take advantage of knowing the previous value of CWND.

# Repeating Slow Start After Idle Period

- Suppose a TCP connection goes idle for a while
- Eventually, the network conditions change
  - Maybe many more flows are traversing the link
- Dangerous to start transmitting at the old rate
  - Previously-idle TCP sender might blast network
  - … causing excessive congestion and packet loss
- So, some TCP implementations repeat slow start
  - Slow-start restart after an idle period

# TCP Problem

- 1 MSS = 1KB
- Max capacity of link: 200 KBps
- RTT = 100ms
- New TCP flow starting, no other traffic in network, assume no queues in network

1. About what is cwnd at time of first packet loss?
   (A) 16 pkts (B) 32 KB (C) 100 KB (D) 200 KB

2. About how long until sender discovers first loss?
   (A) 400 ms (B) 600 ms (C) 1s (D) 1.6s

# Conclusions

- Congestion is inevitable
  - Internet does not reserve resources in advance
  - TCP actively tries to push the envelope
- Congestion can be handled
  - Additive increase, multiplicative decrease
  - Slow start and slow-start restart
- Concurrency in a different form
  - Many concurrent users of the network working together to maximize utilization and fairness
  - Coordinate through signals observable through the network: delay, loss, …