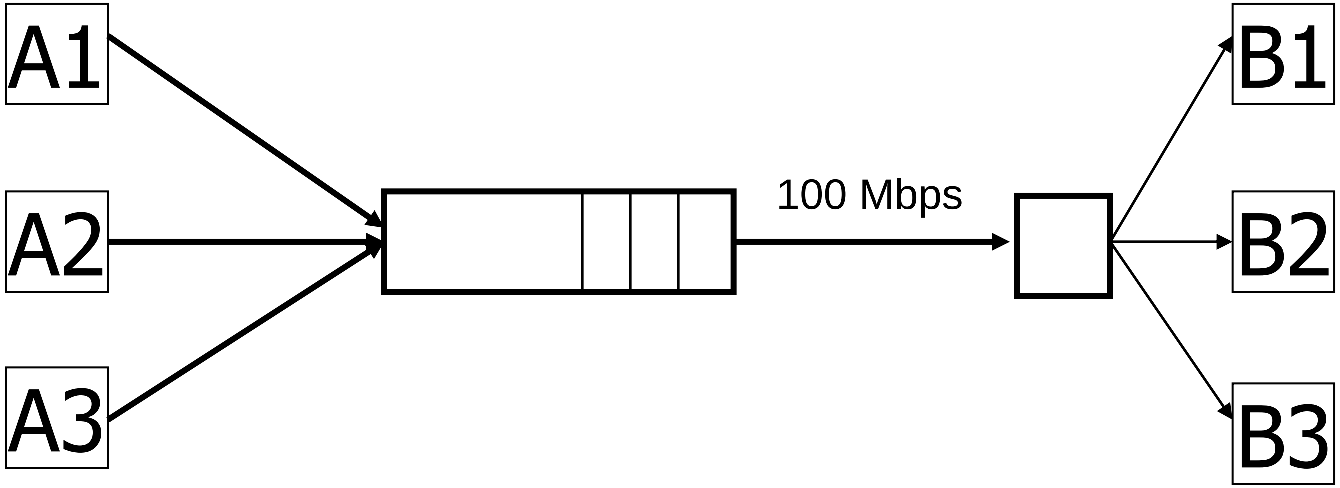
# ECE 463

Lecture: TCP CongestionControl

# Congestion Control

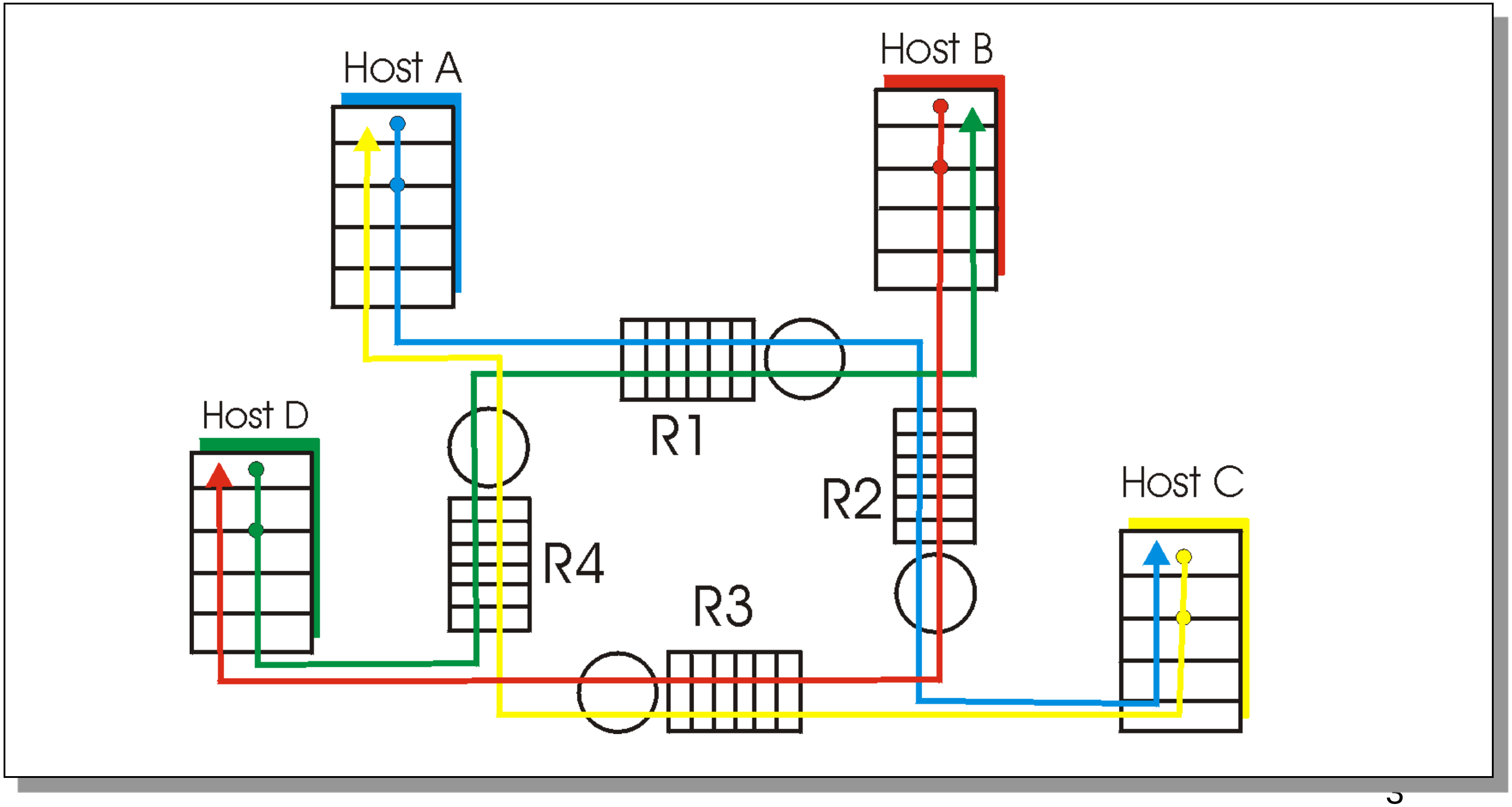
- What is congestion?
- Why does it occur?



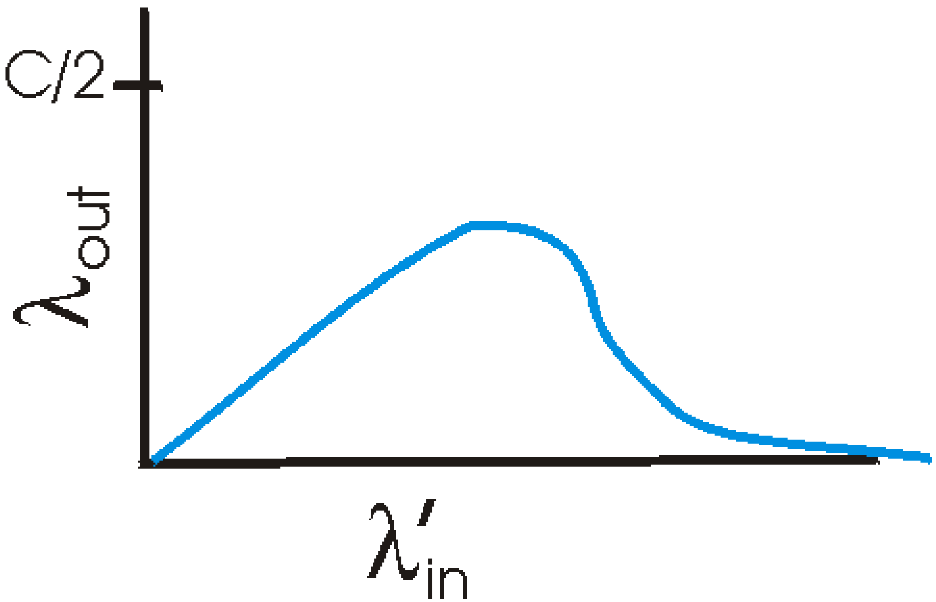
# Causes & Costs of Congestion

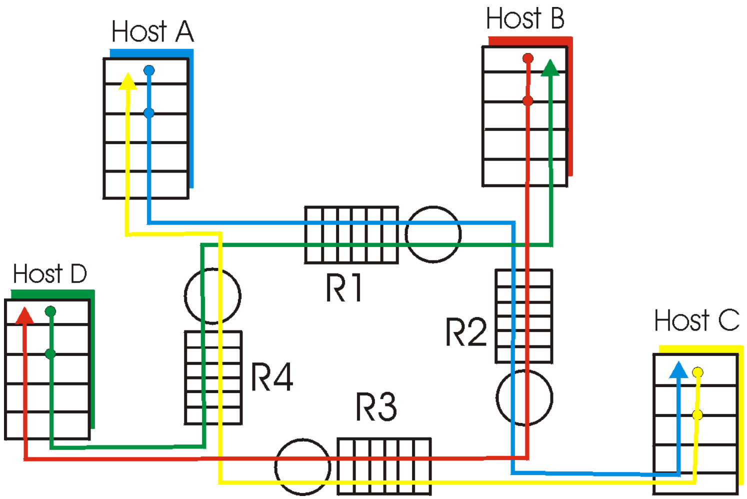
- Four senders – multihop paths
- Timeout/retransmit

Q: What happens as rate increases?



# Causes & Costs of Congestion




- When packet dropped, any "upstream transmission capacity used for that packet was wasted!
- Congestion Collapse: *Increase in network load results in decrease of useful work done*

# Congestion Control and Avoidance

- A mechanism which:
  - Uses network resources efficiently
  - Preserves fair network resource allocation
  - Prevents or avoids collapse
- Congestion collapse is not just a theory
  - Has been frequently observed in many networks

# General Approaches

- Send without care
  - many packet drops
  - could cause congestion collapse
- Reservations
  - pre-arrange bandwidth allocations
  - requires negotiation before sending packets
- Pricing
  - don’t drop packets for the high-bidders
  - requires payment model

# General Approaches (cont’d)

- Dynamic Adjustment
  - Every sender probe network to test level of congestion
  - speed up when no congestion
  - slow down when congestion

# Approaches Towards Congestion Control

- End-end congestion control:
  - No explicit feedback from network
  - Congestion inferred from end-system observed loss, delay
  - Approach taken by TCP
- Network-assisted congestion control:
  - Routers provide feedback to end systems
    - Single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
    - Explicit rate sender should send at
  - Problem: makes routers complicated

# Objectives

- Simple router behavior
- Distributedness
- Efficiency
- Fairness
- Convergence: control system must be stable

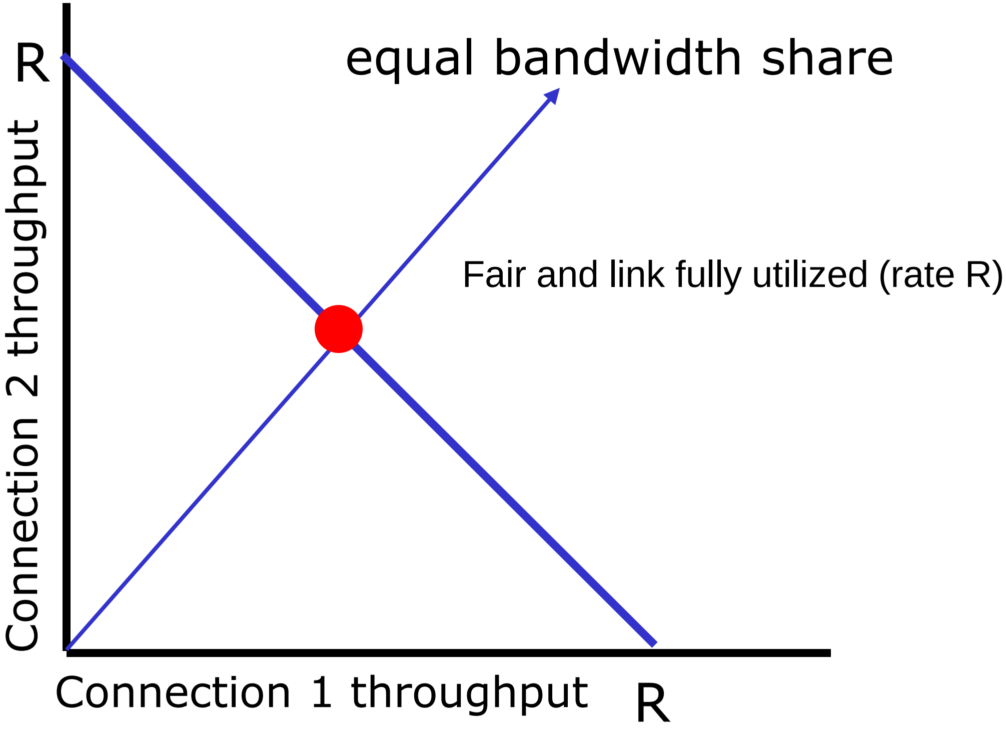
# Basic Control Model

- Reduce speed when congestion is perceived
  - How is congestion signaled?
    - Either mark or drop packets
  - How much to reduce?
- Increase speed otherwise
  - Probe for available bandwidth – how?

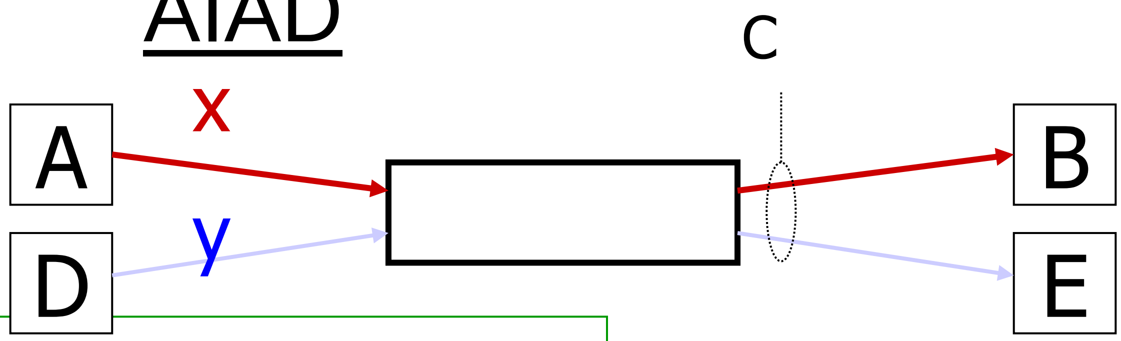
# Linear Control

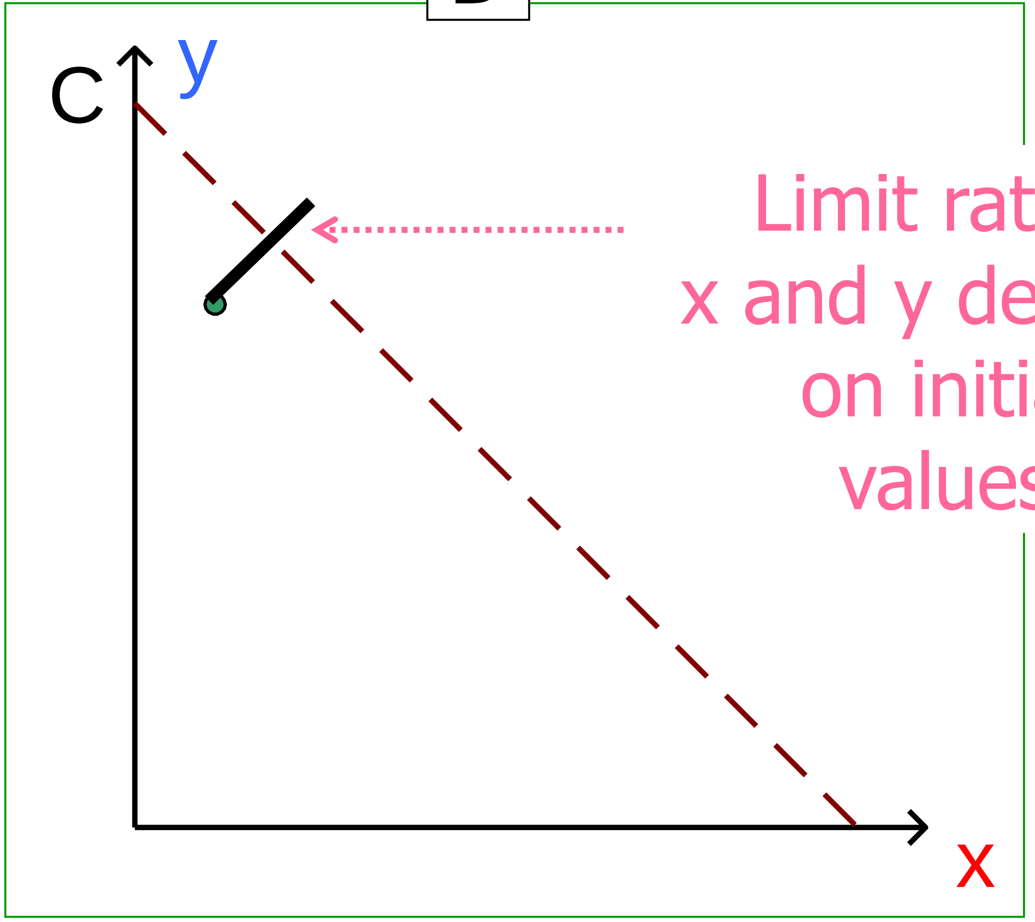
- Many different possibilities for reaction to congestion and probing
  - Examine simple linear controls
    - Window(t + 1) = a + b Window(t)
    - Different $a_i/b_i$ for increase and $a_d/b_d$ for decrease
- Supports various reaction to signals
  - Increase/decrease additively
  - Increased/decrease multiplicatively
  - Which of the four combinations is optimal?

# Ideal Operating Point

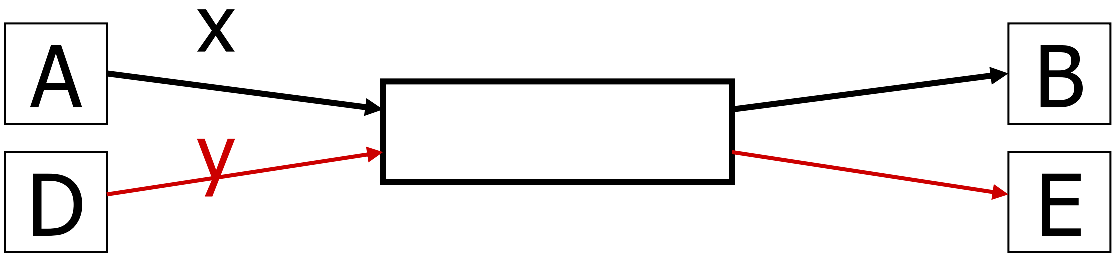


# AIAD

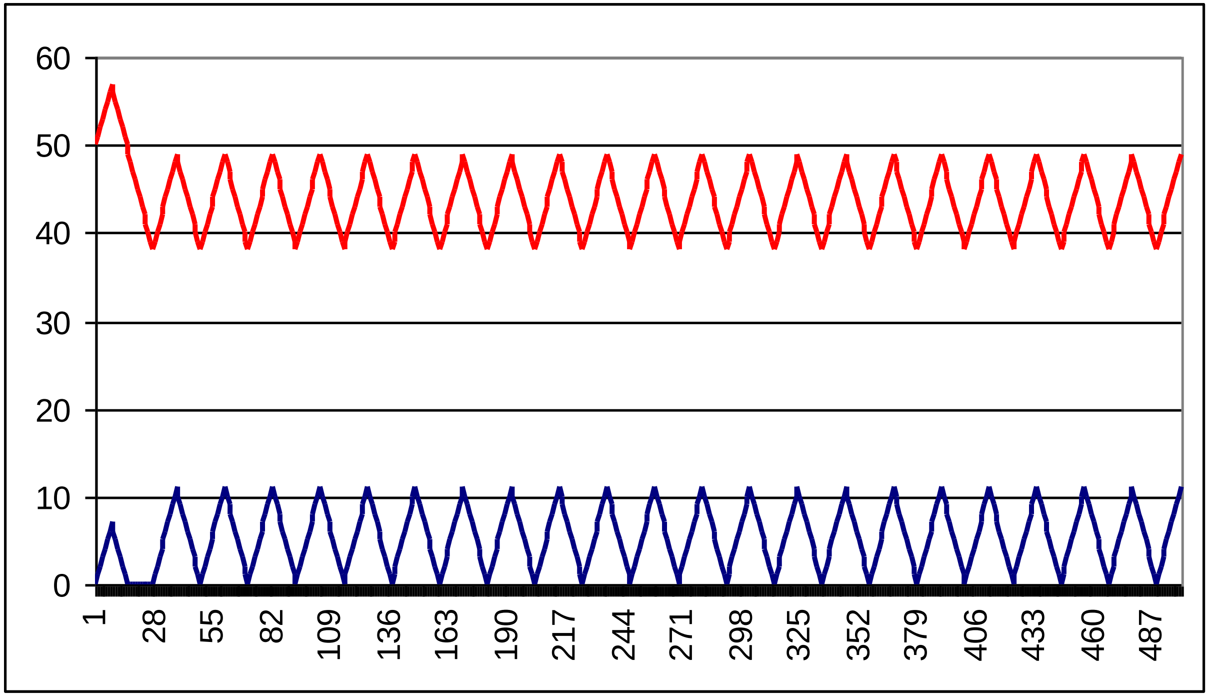


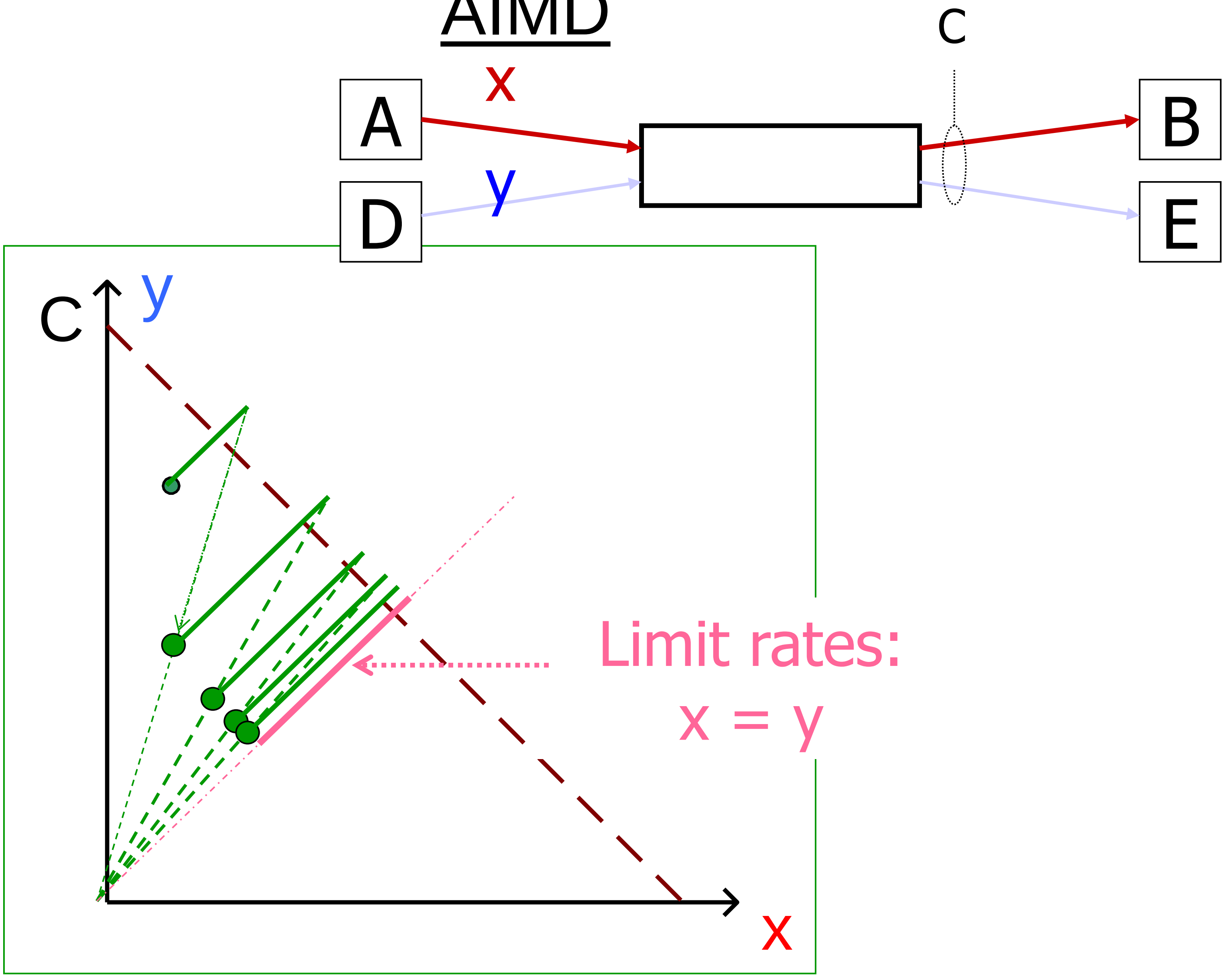
# AIAD Sharing Dynamics




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

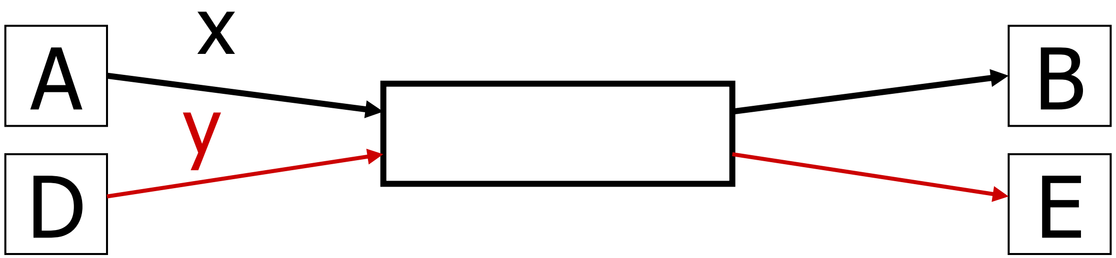


# AIMD

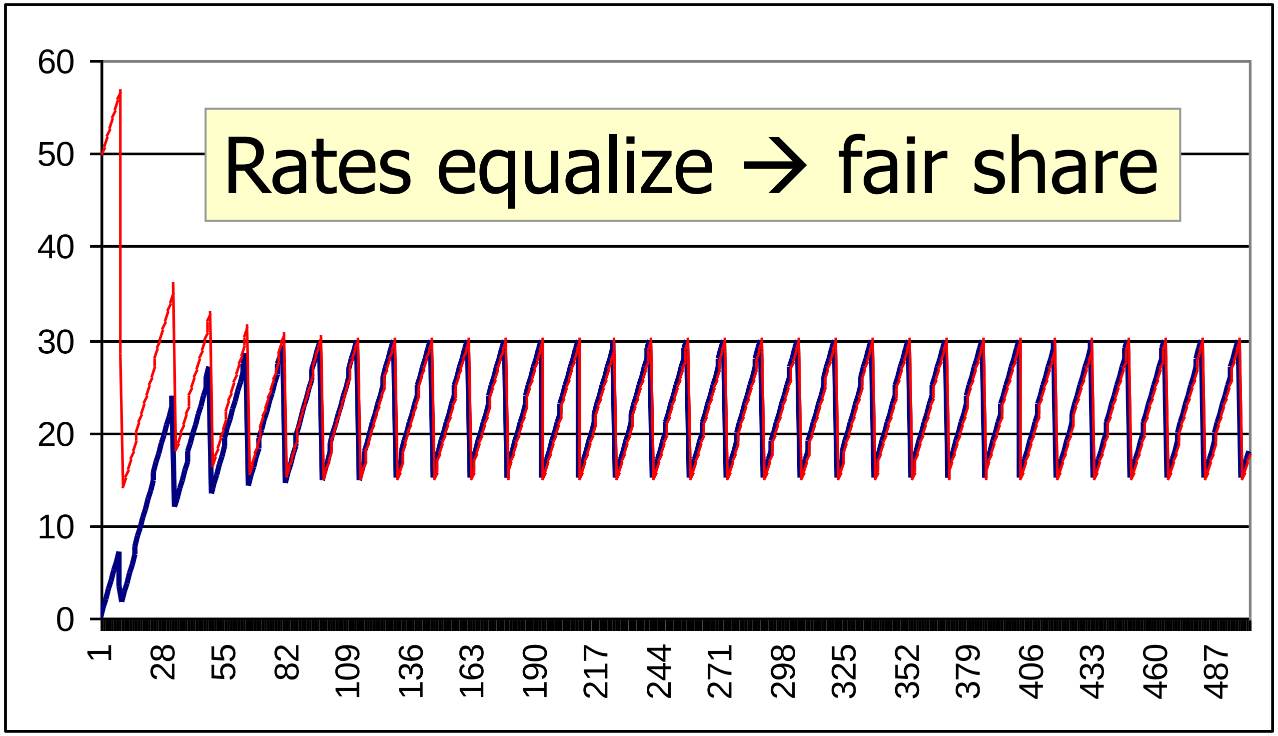



Limit rates:
$x = y$

# AIMD Sharing Dynamics




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



# TCP Congestion Control

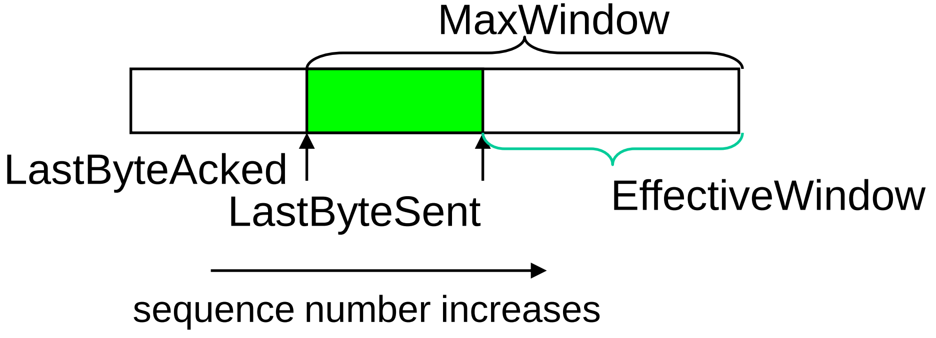
- TCP connection has window
  - controls number of unacknowledged packets
- Sending rate: ~Window/RTT
- Vary window size to control sending rate
- Introduce a new parameter called congestion window (cwnd) at the sender
  - Congestion control is mainly a sender-side operation

# Congestion Window (*cwnd*)

- Limits how much data can be in transit

MaxWindow = min(cwnd, AdvertisedWindow)

EffectiveWindow = MaxWindow – (LastByteSent – LastByteAcked)



# Two Basic Components

- Detecting congestion

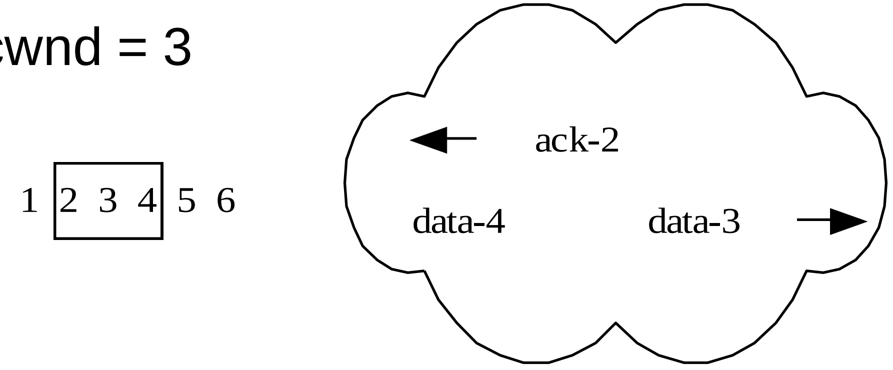
- Rate adjustment algorithm (change cwnd size)
  - depends on congestion or not

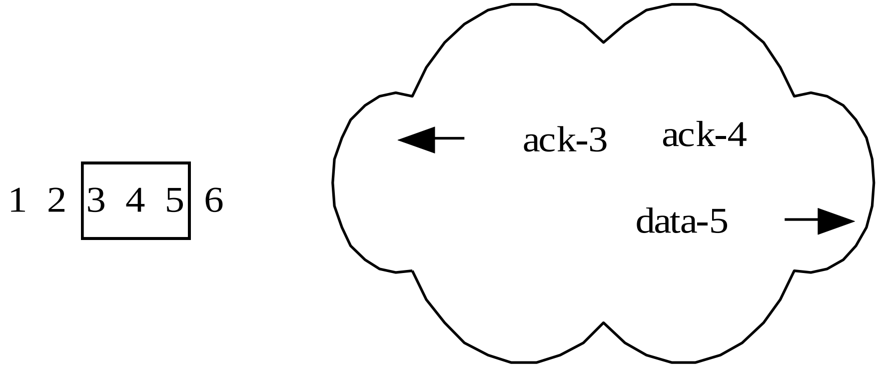
# Detecting Congestion

- Detected based on packet drops
  - Alternative: delay-based methods
- How do you detect packet drops?  ACKs
  - TCP uses ACKs to signal receipt of data
  - ACK denotes last contiguous byte received
    - actually, ACKs indicate next segment expected
- Two signs of packet drops
  - No ACK after certain time interval: time-out
  - Several duplicate ACKs (ignore for now)
- May not work well for wireless networks, why?

# Sliding (Congestion) Window

- Sliding window: each ACK = permission to send a new packet
  - Ex. cwnd = 3



# Rate Adjustment

- Basic structure:
  - Upon receipt of ACK (of new data): increase rate
    - Data successfully delivered, perhaps can send faster
  - Upon detection of loss: decrease rate

# Adapting cwin

- So far: sliding window + self-clocking of ACKs
- How to know the best cwnd (and best transmission rate)?

- Phases of TCP congestion control

1. Slow start (getting to equilibrium)
    1. Want to find this very very fast and not waste time
2. Congestion Avoidance
    - Additive increase - gradually probing for additional bandwidth
    - Multiplicative decrease - decreasing cwnd upon loss/timeout

# Phases of Congestion Control

- **Congestion Window** (**cwnd**)
  Initial value is 1 MSS (=maximum segment size) counted as bytes
- **Slow-start threshold Value** (**CongestionThreshold = congthresh)**
  Initial value is the advertised window size

- **slow start** (cwnd < congthresh)
- **congestion avoidance** (cwnd >= congthresh)

# TCP: Slow Start

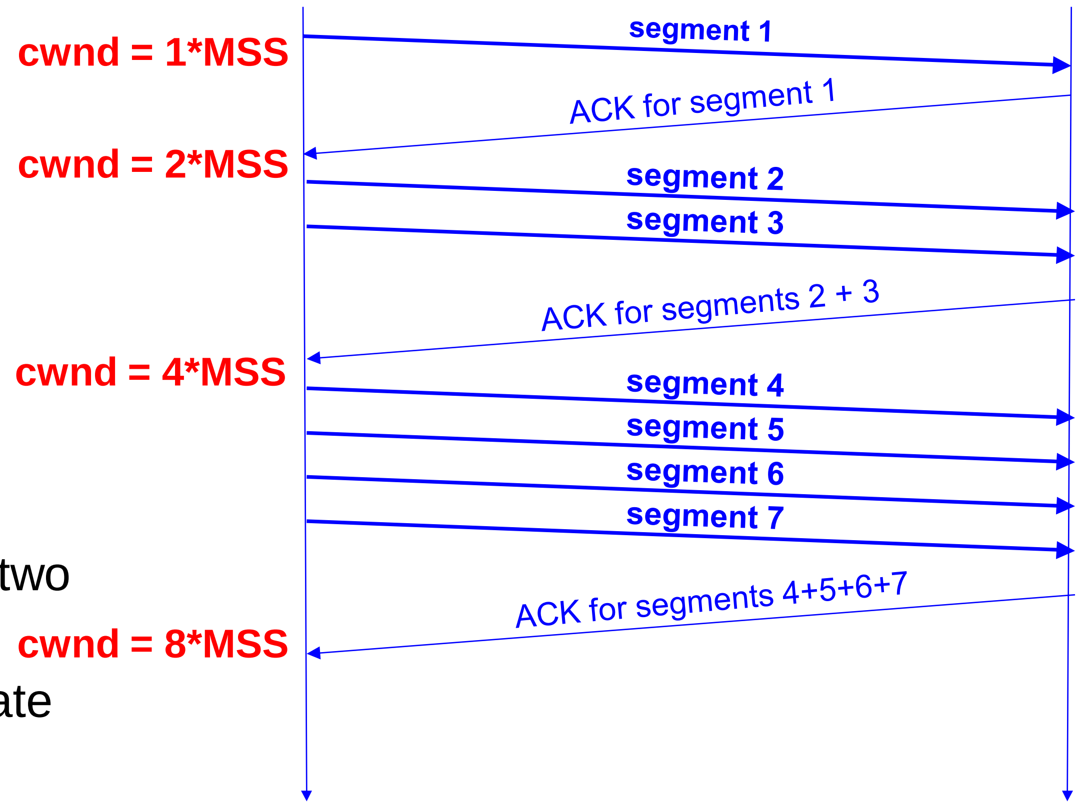
- Goal: discover roughly the proper sending rate quickly

- Whenever starting traffic on a new connection, or whenever increasing traffic after congestion was experienced:
    - Intialize *cwnd* =1 MSS
    - Each time a segment is acknowledged, increment *cwnd* by one MSS (*cwnd* += 1 * MSS).

- Continue until
  – Reach congthresh
  – Packet loss

# Slow Start Illustration

- The congestion window size grows very rapidly
- TCP slows down the increase of *cwnd* when ***cwnd >= congthresh***
- Observe:
  - Each ACK generates two packets
  - slow start increases rate exponentially fast (doubled every RTT)!



# Congestion Avoidance (After Slow Start)

- Slow Start figures out roughly the rate at which the network starts getting congested
- Congestion Avoidance continues to react to network condition
  - Probes for more bandwidth, increase cwnd if more bandwidth available
  - If congestion detected, aggressive cut back cwnd

# Congestion Avoidance: Additive Increase

- After exiting slow start, slowly increase cwnd to probe for additional available bandwidth
  - Competing flows may end transmission
  - May have been "unlucky" with an early drop

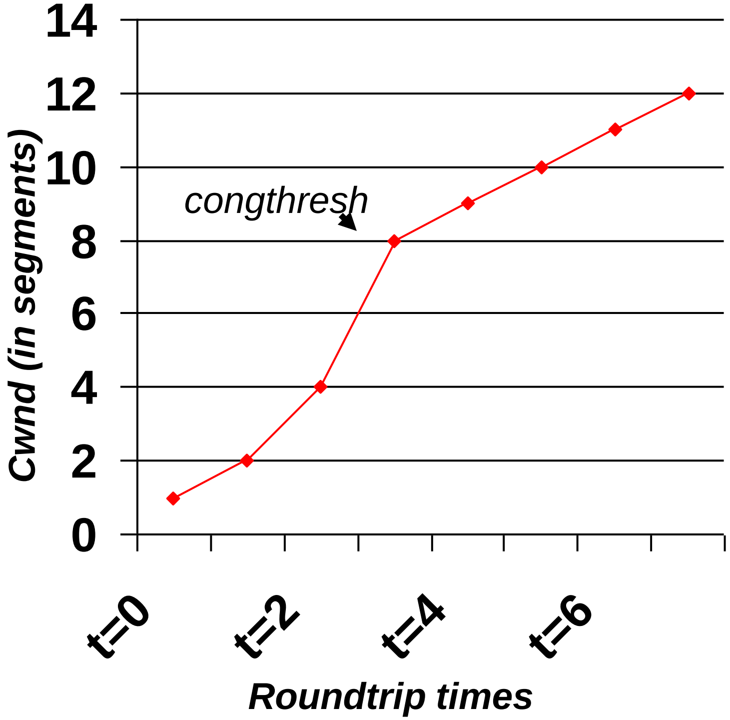
- **If** *cwnd > congthresh* **then**
  each time a segment is acknowledged
  increment *cwnd* by *MSS * (MSS/cwnd*)

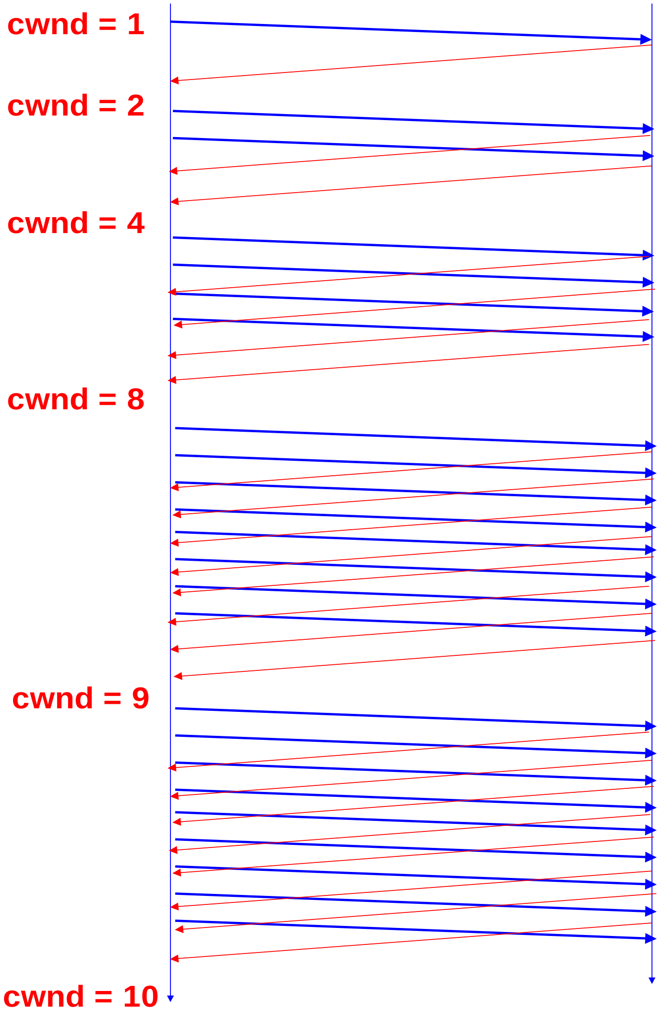
- *cwnd* is increased by one MSS only if all segments have been acknowledged
  - Increases by 1 MSS per RTT, vs. doubling per RTT

# Example of Slow Start + Congestion Avoidance

Assume that *congthresh = 8*

*(cwnd in units of MSS)*



# Detecting Congestion via Timeout

- If there is a packet loss, the ACK for that packet will not be received

- The packet will eventually timeout
  - No ack is seen as a sign of congestion

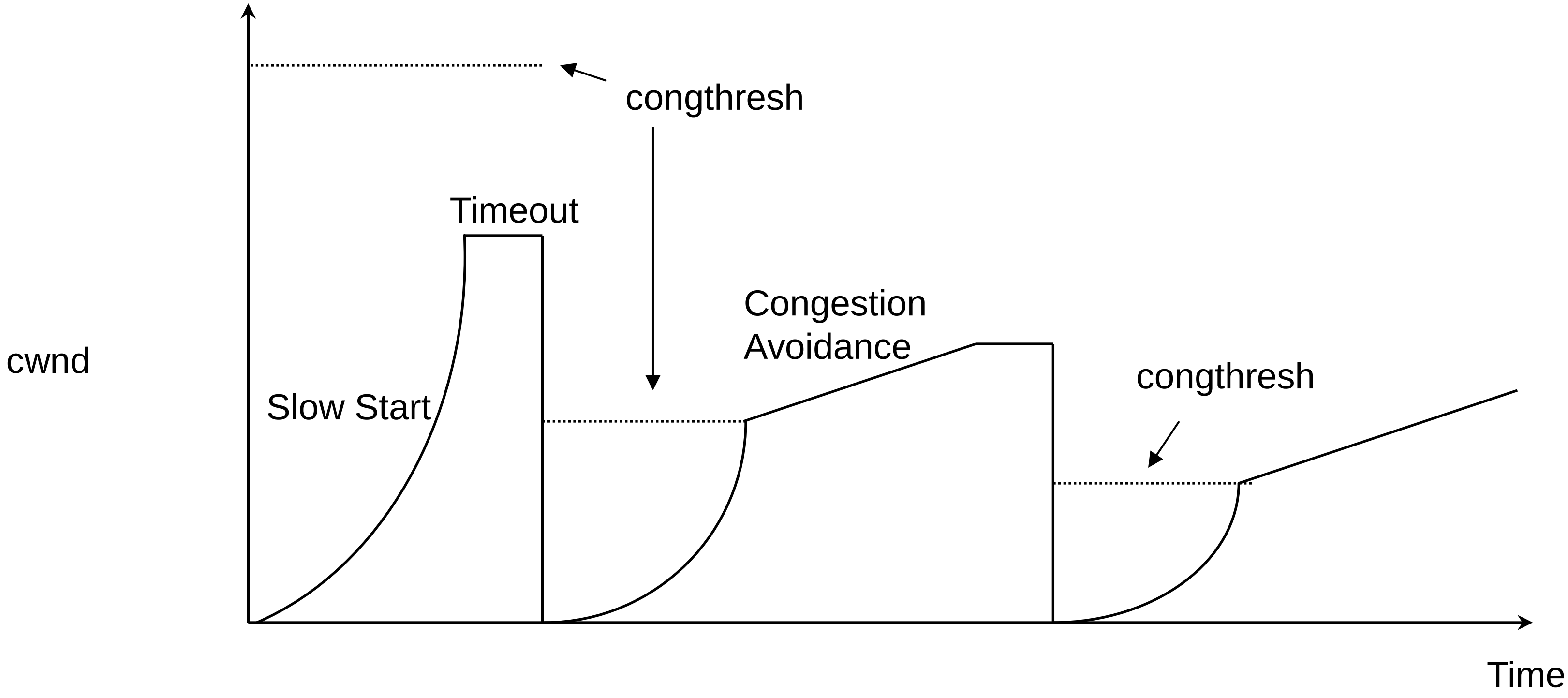
# Congestion Avoidance: Multiplicative Decrease

- Timeout = congestion
- Each time when congestion occurs,
  - congthresh is set to half the current size of the congestion window:

    congthresh = cwnd / 2
  - cwnd is reset to one MSS:

    cwnd = 1 * MSS
  - and slow-start is entered

# TCP illustration



# Responses to Congestion (Loss)

- There are algorithms developed for TCP to respond to congestion

  - **TCP Tahoe** - the basic algorithm (discussed previously)
  - **TCP Reno** - Tahoe + fast retransmit & fast recovery
    - Most end hosts today implement TCP Reno

- and many more:
  - TCP Vegas (use timing of ACKs to avoid loss)
  - TCP SACK (selective ACK)

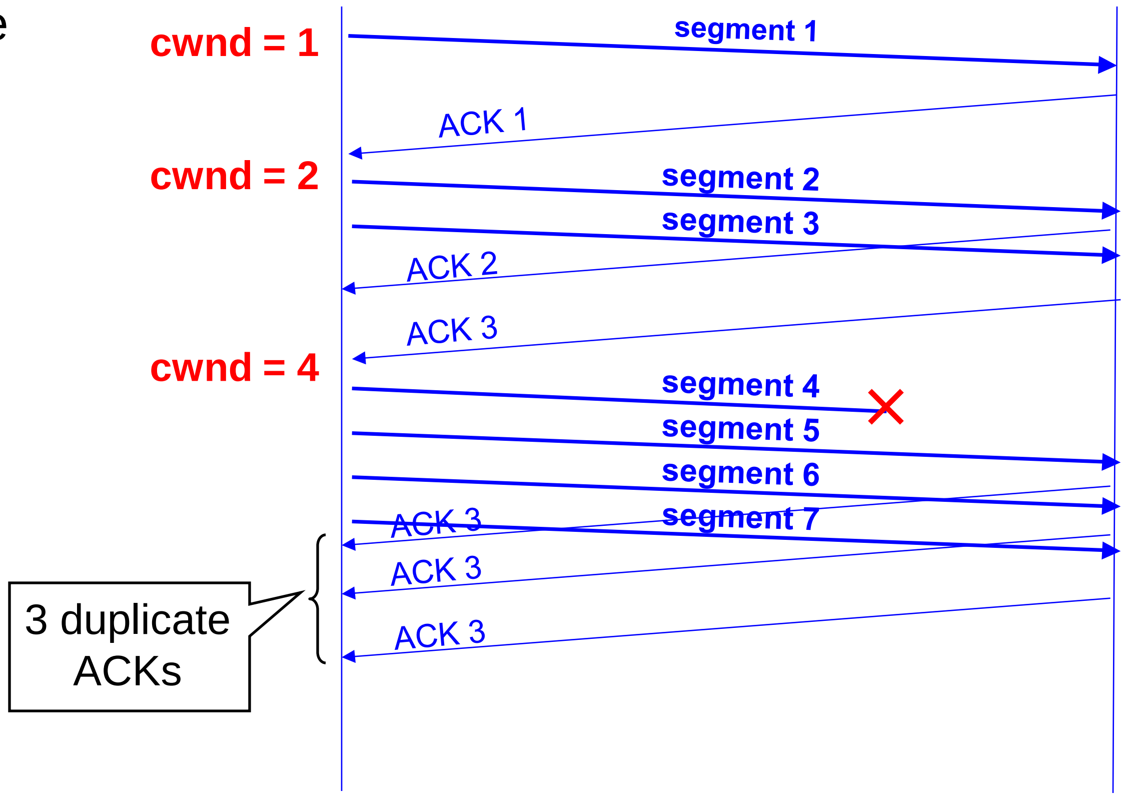
# TCP Reno

- Problem with Tahoe: If a segment is lost, there is a long wait until timeout
- Reno adds a **fast retransmit** and **fast recovery mechanism**

- Upon receiving 3 duplicate ACKs**,** retransmit the presumed lost segment ("fast retransmit")
- But do not enter slow-start. Instead enter congestion avoidance ("fast recovery")

# Fast Retransmit

- Resend a segment after 3 duplicate ACKs
  - remember a duplicate ACK means that an out-of sequence segment was received
  - ACK-n means packets 1, …, n all received



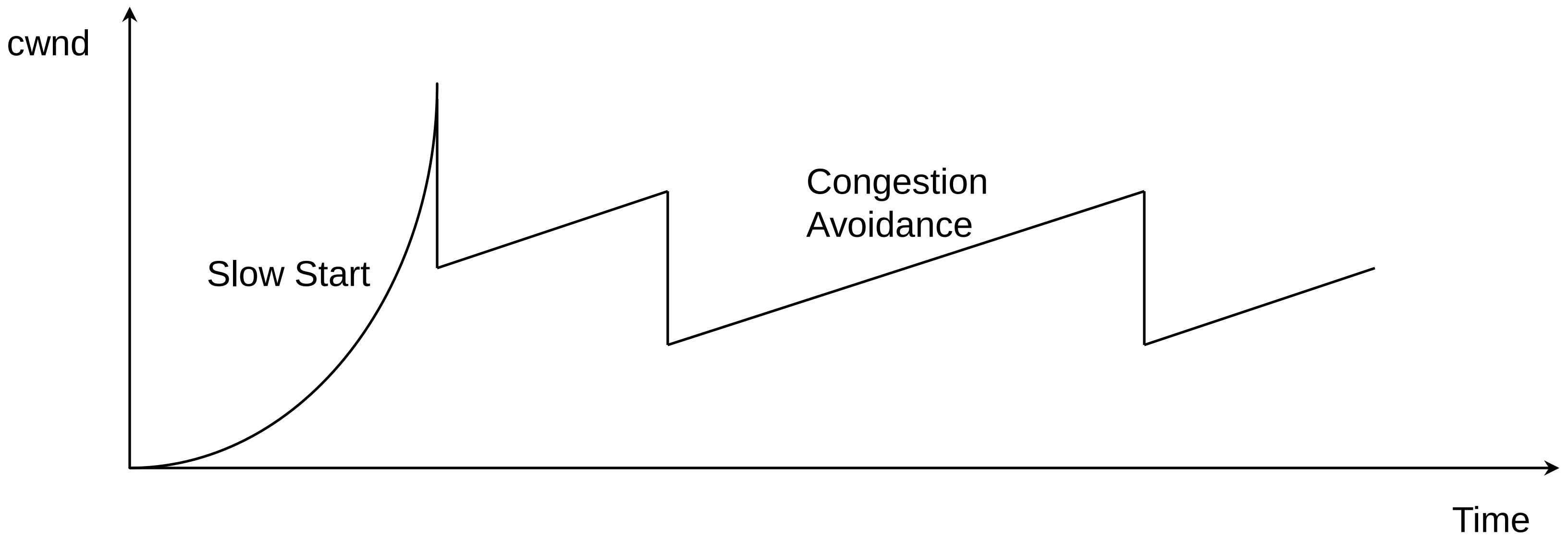

- Notes:
  - duplicate ACKs due to packet reordering!
  - if window is small don't get duplicate ACKs!

# Fast Recovery

- After a fast-retransmit
  - cwnd = cwnd*/2*  (vs. 1 in Tahoe)
  - congthresh = cwnd
  - i.e. starts congestion avoidance at new cwnd
    - Not slow start from cwnd = 1 * MSS

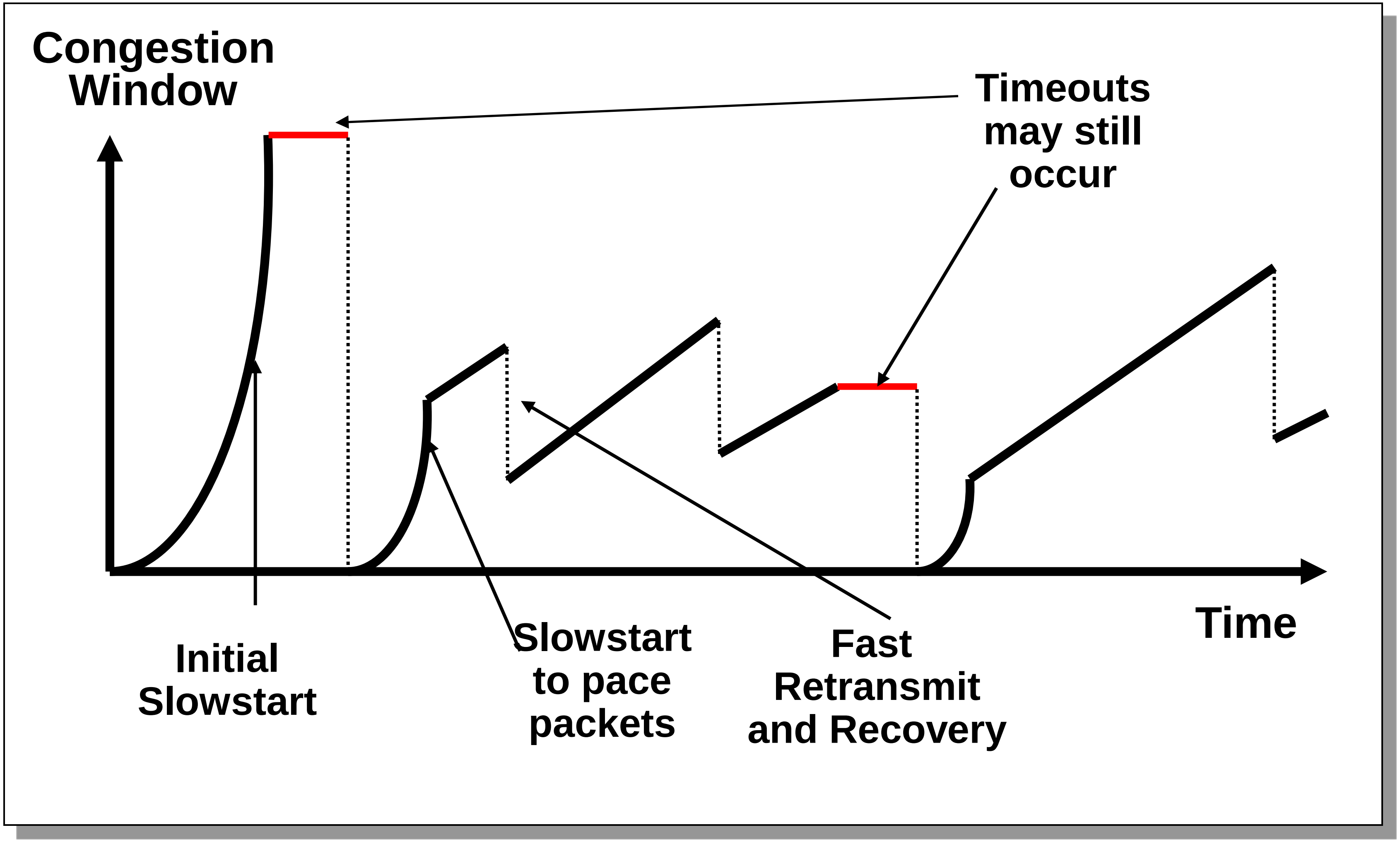
- After a timeout
  - congthresh = *cwnd/2*
  - cwnd = 1 * MSS
  - Do slow start
  - Same as Tahoe

# Fast Retransmit and Fast Recovery




- Retransmit after 3 duplicate ACKs
  - prevent expensive timeouts
- Slow start only once per session (if no timeouts)
- In steady state, *cwnd* oscillates around the ideal window size.

# TCP Reno Saw Tooth Behavior



# TCP Reno Quick Review

- Slow-Start if cwnd < congthresh
  - cwnd+= 1 MSS upon every new ACK (exponential growth)
  - Timeout: congthresh = cwnd/2 and cwnd = 1 MSS

- Congestion avoidance if cwnd >= congthresh
  - Additive Increase Multiplicative Decrease (AIMD)
  - ACK: cwnd = cwnd + MSS * MSS/cwnd
  - Timeout: congthresh = cwnd/2 and cwnd = 1 * MSS

- Fast Retransmit & Recovery
  - 3 duplicate ACKS (interpret as packet loss)
  - Retransmit lost packet
  - cwnd=cwnd/2, congthresh = cwnd

# 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