# EECS 489

## Computer Networks

TCP – Congestion Control II

# Agenda

- TCP congestion control wrap-up
- TCP throughput equation
- Problems with congestion control

# Recap

- Flow Control
  - Restrict window to RWND to make sure that the receiver isn't overwhelmed
- Congestion Control
  - Restrict window to CWND to make sure that the network isn't overwhelmed
- Together
  - Restrict window to min{RWND, CWND} to make sure that neither the receiver nor the network are overwhelmed

# CC Implementation

- States at sender
  - CWND (initialized to a small constant)
  - ssthresh (initialized to a large constant)
  - dupACKcount and timer
- Events
  - ACK (new data)
  - dupACK (duplicate ACK for old data)
  - Timeout

# Event: ACK (new data)

- If CWND < ssthresh
  - CWND += 1

- *CWND packets per RTT*
- *Hence, after one RTT with no drops:*

  *CWND = 2xCWND*

# Event: ACK (new data)

- If CWND < ssthresh
  - CWND += 1

*Slow start phase*

# Slow Start phase

- Sender starts at a slow rate, but increases exponentially until first loss
- Start with a small congestion window
  - Initially, CWND = 1
  - So, initial sending rate is MSS/RTT
- Double the CWND for each RTT with no loss

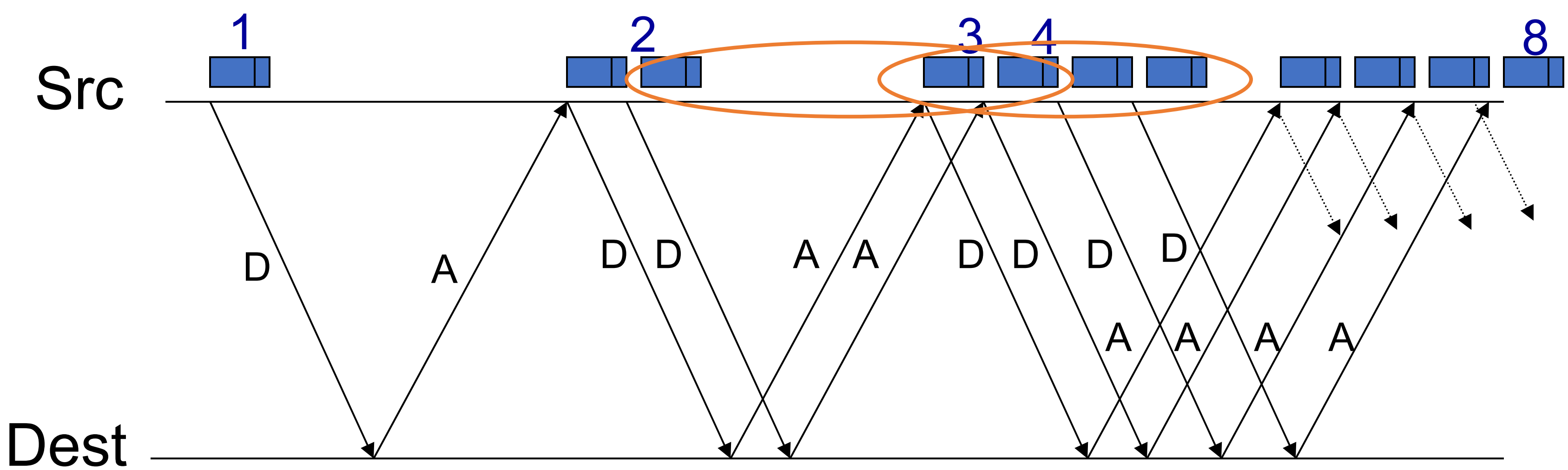
# Slow Start in action

- For each RTT: double CWND
  - i.e., for each ACK, CWND += 1

Linear increase per ACK(CWND+1) → exponential increase per RTT (2*CWND)

# Slow Start in action

- For each RTT: double CWND
  - i.e., for each ACK, CWND += 1



# When does Slow Start stop?

- Slow Start gives an estimate of available bandwidth
  - At some point, there will be loss
- Introduce a “slow start threshold” (ssthresh)
  - Initialized to a large value
- If CWND > ssthresh, stop Slow Start

# Event: ACK (new data)

- If CWND < ssthresh
  - CWND += 1

Slow start phase

- Else
  - CWND = CWND + 1/CWND

Congestion avoidance phase

- CWND packets per RTT
- Hence, after one RTT with no drops:
  CWND = CWND + 1

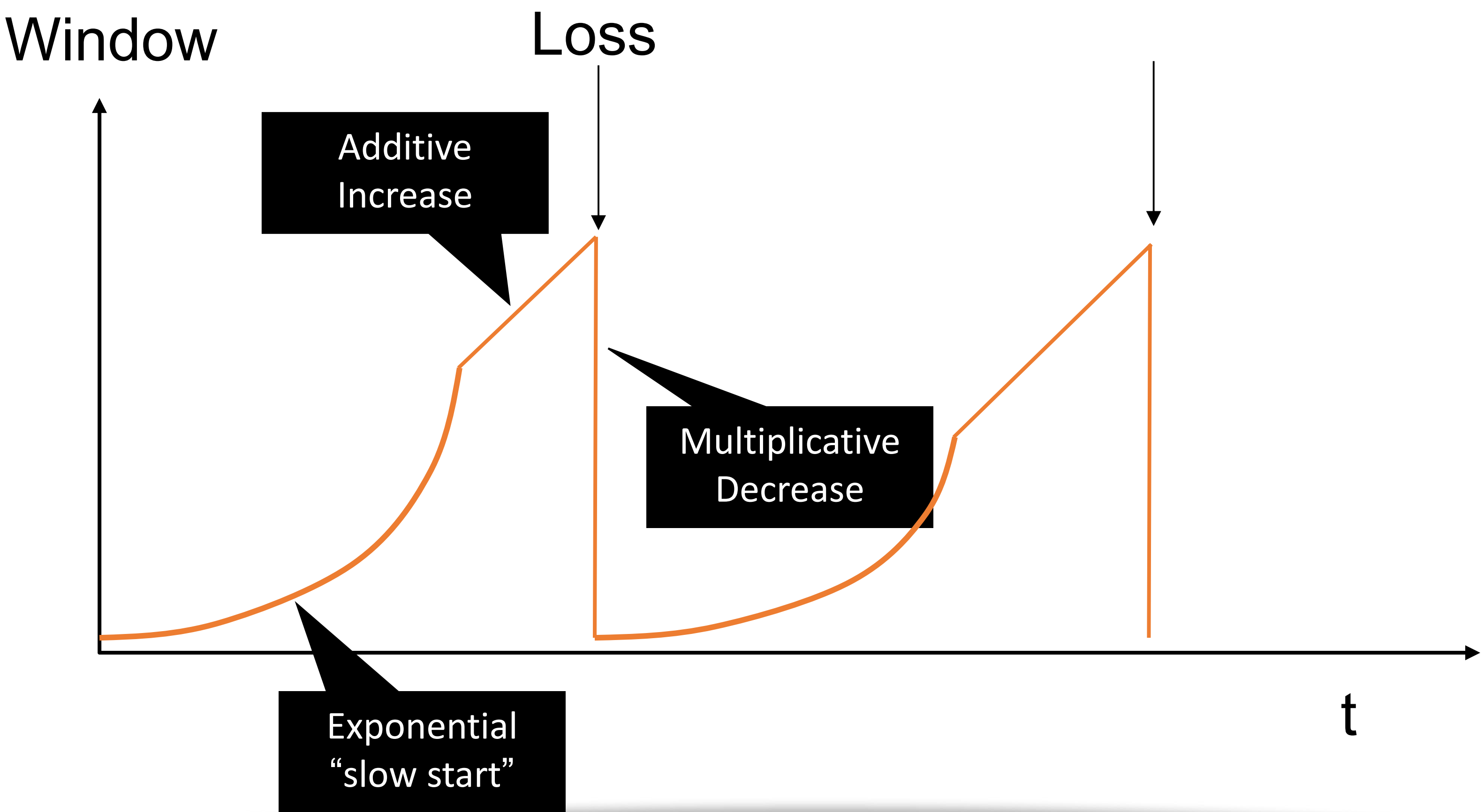
# Adjusting to varying bandwidth

- CWND > ssthresh
  - Stop rapid growth and focus on maintenance
- Now, want to track variations in this available bandwidth, oscillating around its current value
  - Repeated probing (rate increase) and backoff (decrease)
- TCP uses: “Additive Increase Multiplicative Decrease” (AIMD)

# AIMD

- Additive increase
  - For each ACK, CWND = CWND+ 1/CWND
  - CWND is increased by one only if all segments in a CWND have been acknowledged
- Multiplicative decrease
  - On packet loss, divide ssthresh in half and slow start
    - ssthresh = CWND/2
    - CWND = 1
    - Initiate Slow Start
  - Note that we're ignoring the "dupAck" fix for now

# AIMD leads to TCP sawtooth



# Why AIMD?

- Recall the three issues
  - Finding available bottleneck bandwidth
  - Adjusting to bandwidth variations
  - Sharing bandwidth

- Two goals for bandwidth sharing
  - Efficiency: High utilization of link bandwidth
  - Fairness: Each flow gets equal share

# Why AIMD?

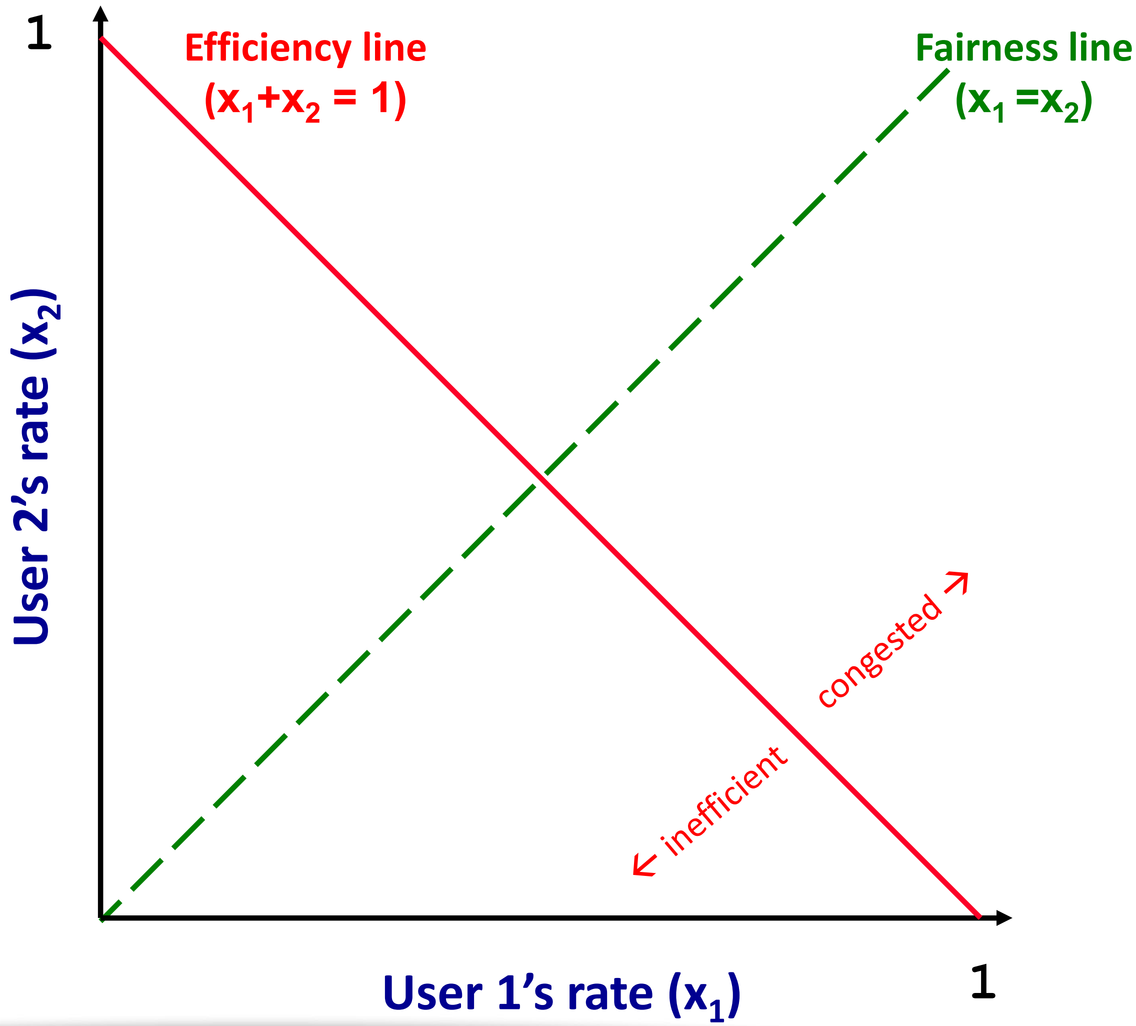
- Every RTT, we can do
  - Multiplicative increase or decrease: CWND$\rightarrow$ a*CWND
  - Additive increase or decrease: CWND$\rightarrow$ CWND + b
- Four alternatives:
  - AIAD: gentle increase, gentle decrease
  - AIMD: gentle increase, drastic decrease
  - MIAD: drastic increase, gentle decrease
  - MIMD: drastic increase and decrease

# Simple model of congestion control

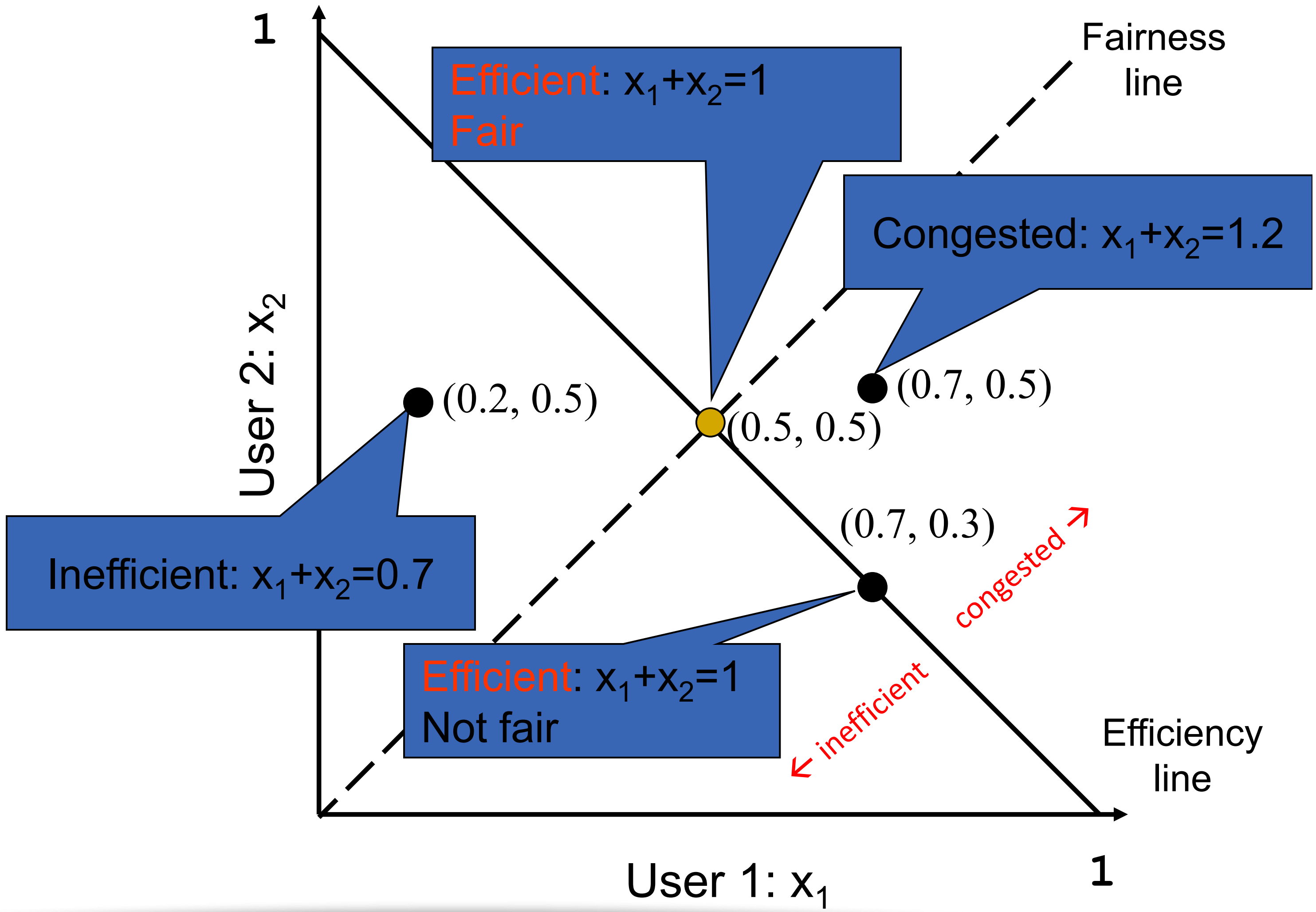
- Two users
  - rates x1 and x2

- Congestion when x1+x2 > 1
- Unused capacity when x1+x2 < 1

- Fair when x1 =x2



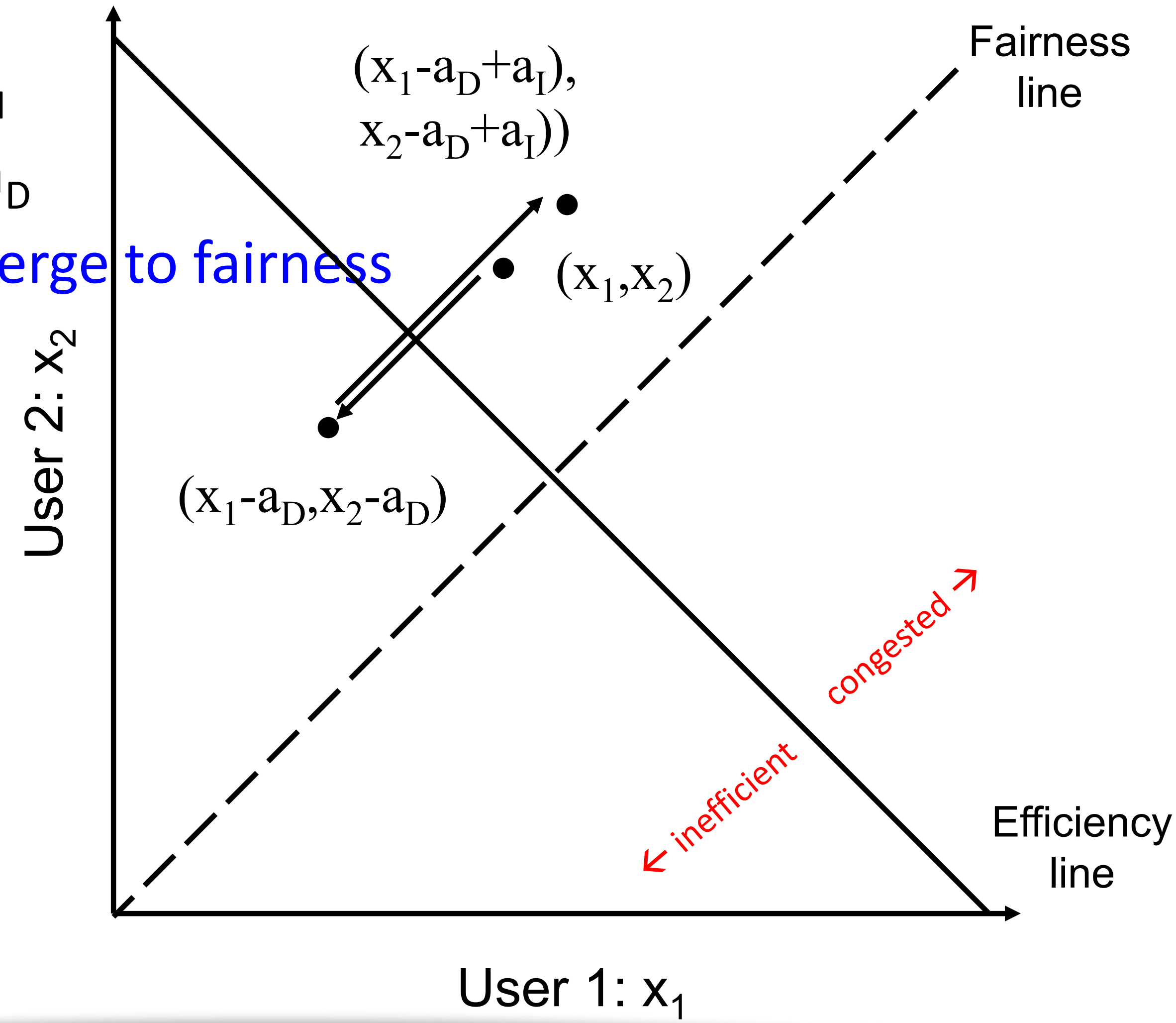
# Example



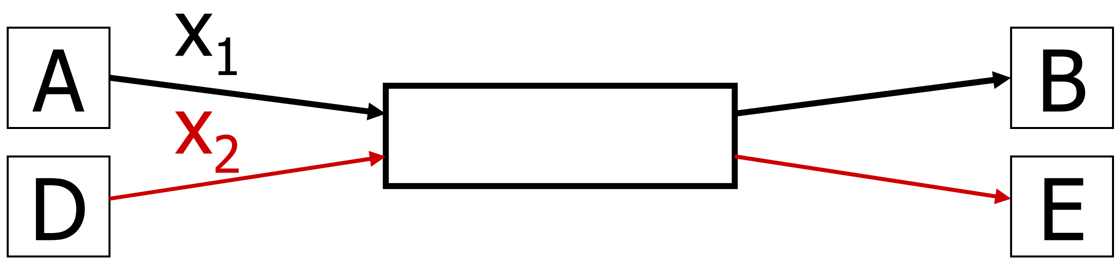
1
Efficient: $x_1+x_2=1$
Fair
Fairness line
Congested: $x_1+x_2=1.2$
User 2: $x_2$
(0.2, 0.5)
(0.7, 0.5)
(0.5, 0.5)
(0.7, 0.3)
congested →
Inefficient: $x_1+x_2=0.7$
Efficient: $x_1+x_2=1$
Not fair
← inefficient
Efficiency line
User 1: $x_1$
1

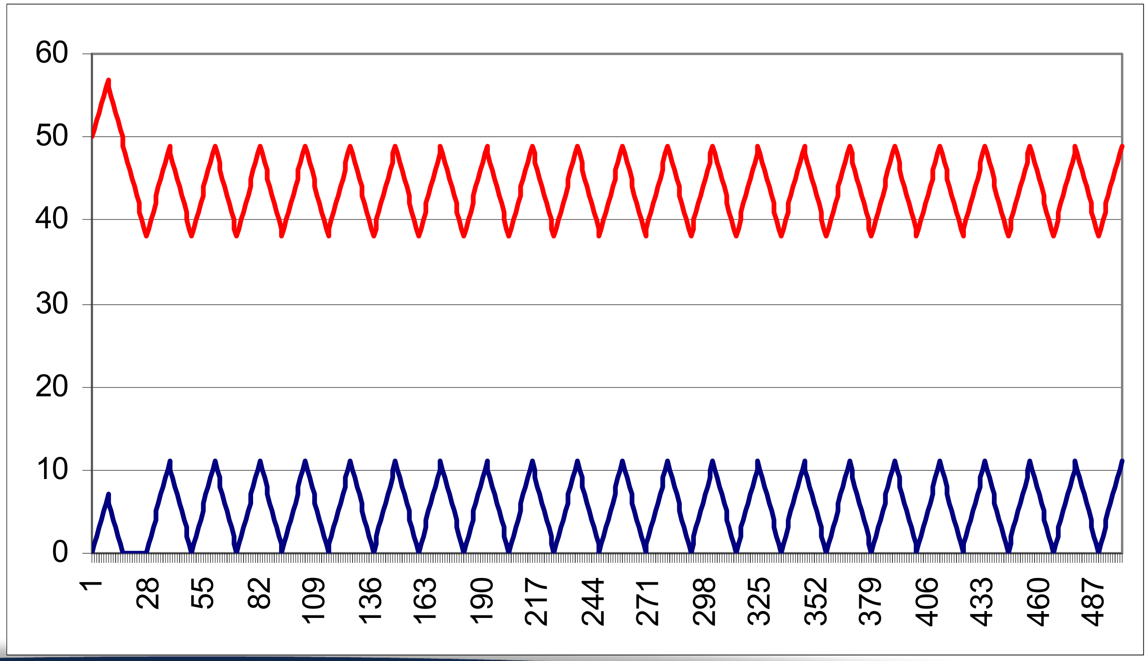
# AIAD

- Increase: $x + a_I$
- Decrease: $x - a_D$
- Does not converge to fairness



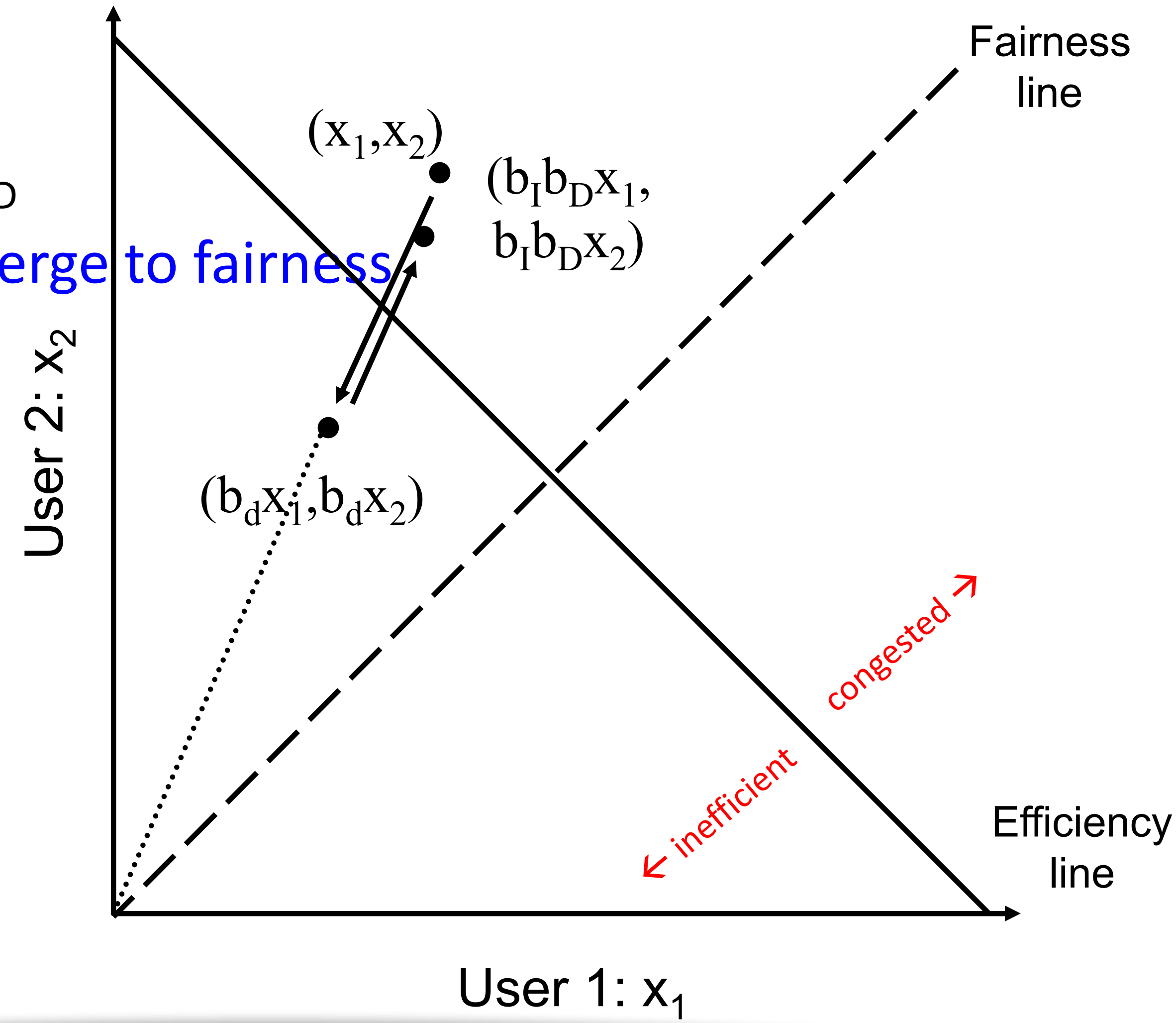
# AIAD Sharing Dynamics



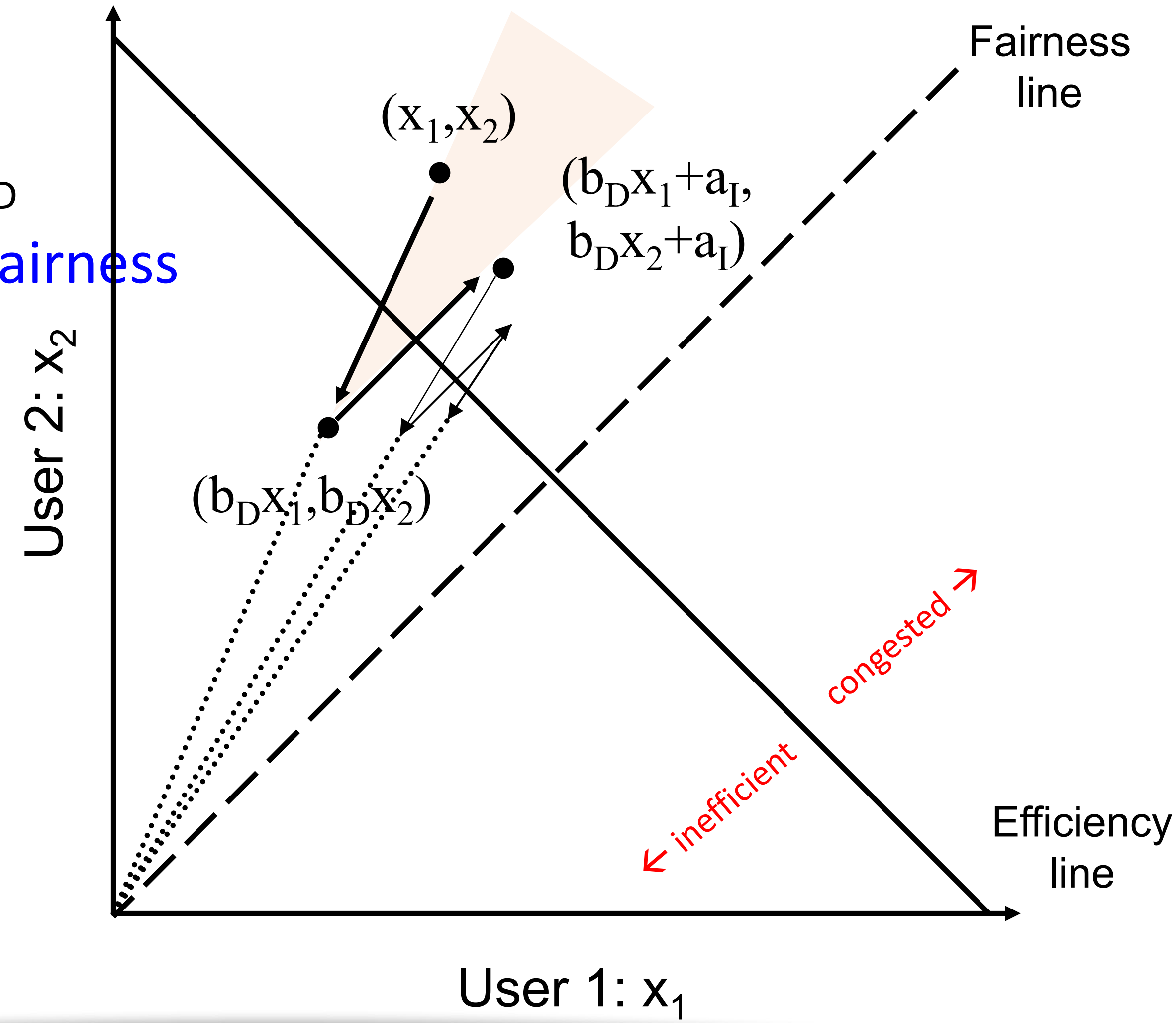
# MIMD

- Increase: $x*b_I$
- Decrease: $x*b_D$
- Does not converge to fairness

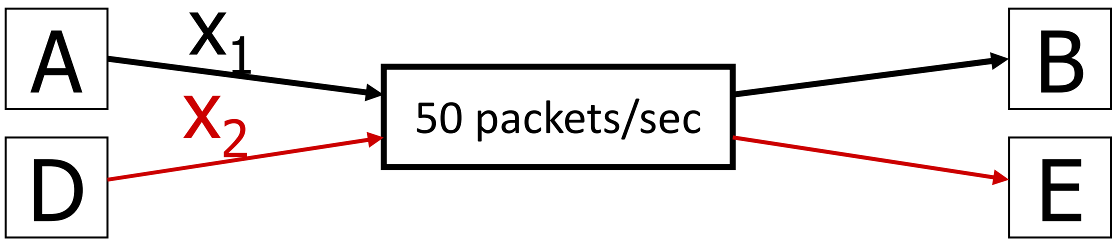


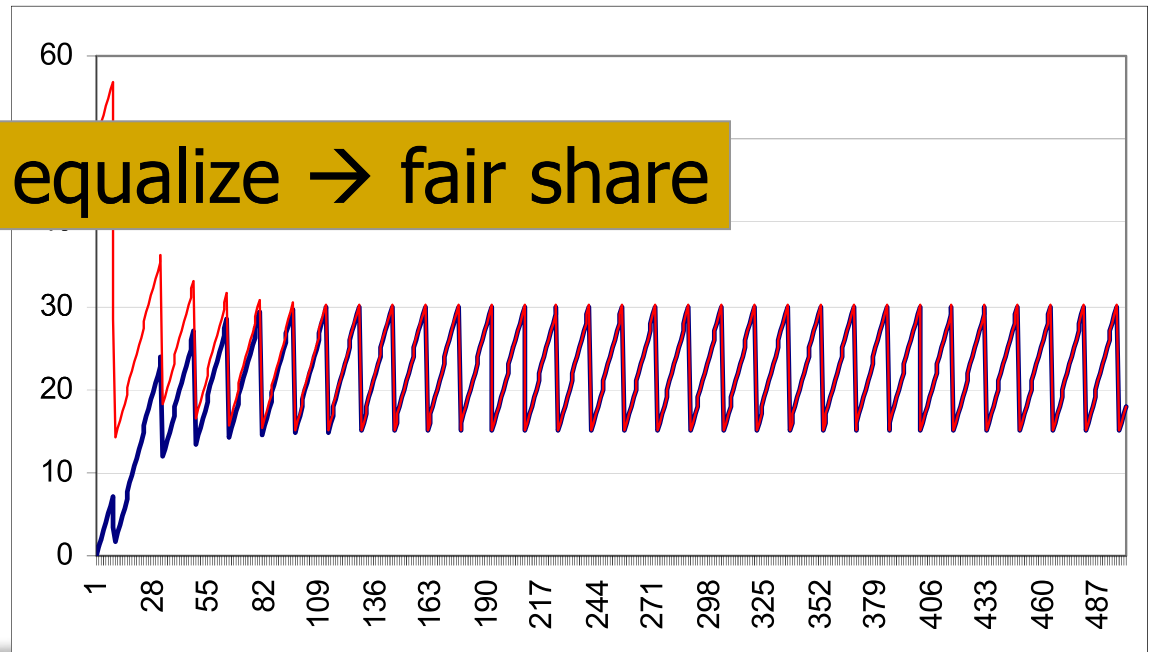
# AIMD

- Increase: $x+a_I$
- Decrease: $x*b_D$
- Converges to fairness



# AIMD Sharing Dynamics



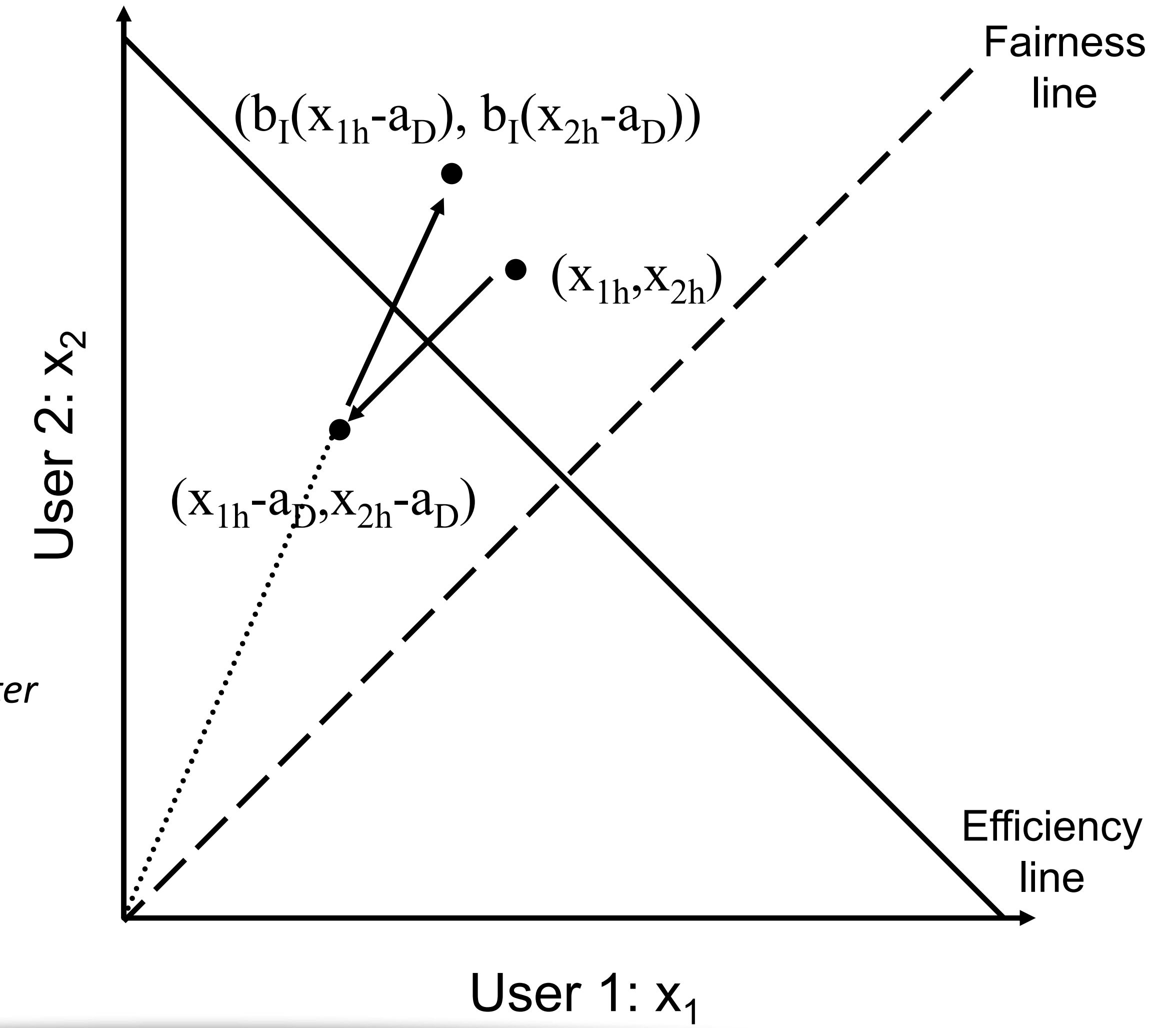

**Rates equalize → fair share**

# MIAD

- Increase: $x*b_I$
- Decrease: $x - a_D$
- Does not converge to fairness
- Does not converge to efficiency

- *"Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks"*

  *-- Chiu and Jain*



# Event: TimeOut

- On Timeout
  - ssthresh ← CWND/2
  - CWND ← 1

# Event: dupACK

- dupACKcount ++
- If dupACKcount = 3 /* fast retransmit */
  - ssthresh = CWND/2
  - CWND = CWND/2 - More on this in a bit

# Not done yet!

- Problem: congestion avoidance too slow in recovering from an isolated loss

# Example

- Consider a TCP connection with:
  - CWND=10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq. no. 101
- 10 packets [101, 102, 103,..., 110] are in flight
  - Packet 101 is dropped

# Timeline: [~~101~~, 102, ..., 110]

- If CWND < ssthresh
  - CWND += 1

  *Slow start phase*
- Else
  - CWND = CWND + 1/CWND

  *Congestion avoidance phase*

- ACK 101 (due to 102) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 103) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 104) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 105) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 106) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 107) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 108) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 109) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 110) cwnd=6 + 1/6 (no xmit)
- ACK 111 (due to 101) 🡐 only now can we transmit new packets
- Plus no packets in flight so ACK "clocking" (to increase CWND) stalls for another RTT

# Solution: Fast recovery

- Idea: Grant the sender temporary “credit” for each dupACK so as to keep packets in flight
- If dupACKcount = 3
  - ssthresh = CWND/2
  - CWND = ssthresh + 3
- While in fast recovery
  - CWND = CWND + 1 for each additional dupACK
- Exit fast recovery after receiving new ACK
  - set CWND = ssthresh

# Example

- Consider a TCP connection with:
  - CWND=10 packets
  - Last ACK was for packet # 101
    - i.e., receiver expecting next packet to have seq. no. 101
- 10 packets [101, 102, 103,..., 110] are in flight
  - Packet 101 is dropped

# Timeline: [~~101~~, 102, ..., 110]

If dupACKcount = 3

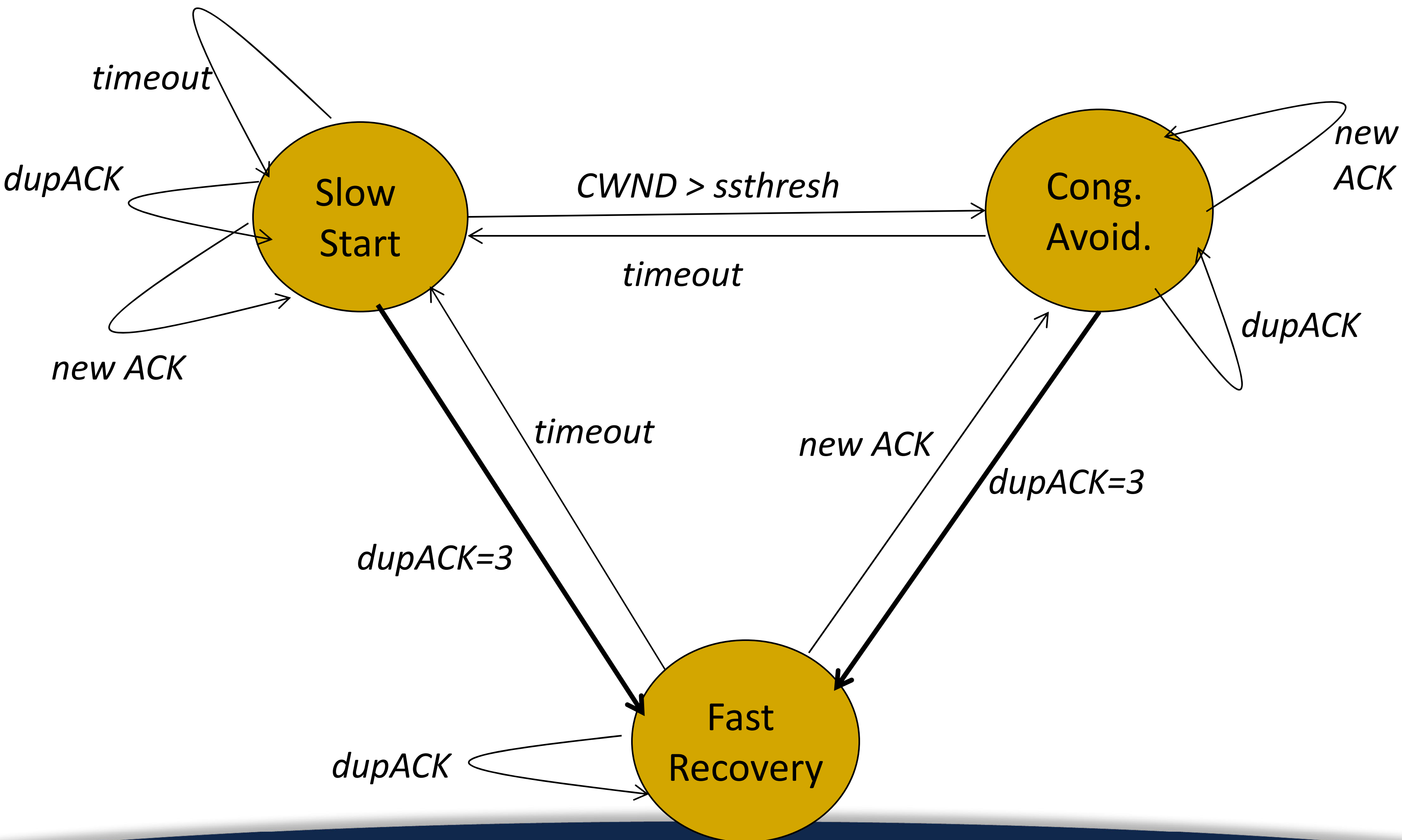
- ssthresh = CWND/2
- CWND = ssthresh + 3

While in fast recovery

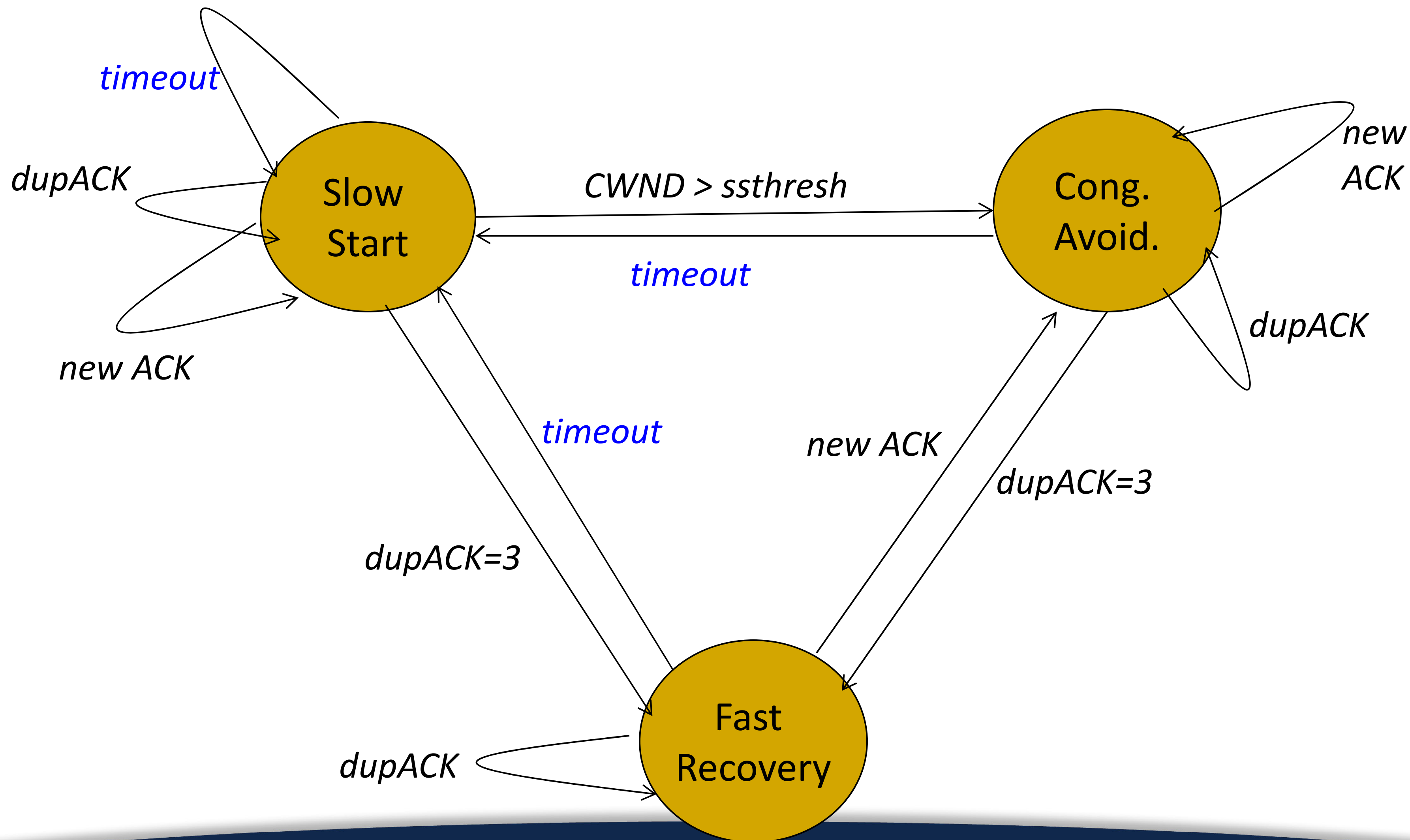
- CWND = CWND + 1 for each additional dupACK

- ACK 101 (due to 102) cwnd=10 dup#1
- ACK 101 (due to 103) cwnd=10 dup#2
- ACK 101 (due to 104) cwnd=10 dup#3
- RETRANSMIT 101 ssthresh=5 cwnd= 8 (5+3)
- ACK 101 (due to 105) cwnd= 9 (no xmit)
- ACK 101 (due to 106) cwnd=10 (no xmit)
- ACK 101 (due to 107) cwnd=11 (xmit 111)
- ACK 101 (due to 108) cwnd=12 (xmit 112)
- ACK 101 (due to 109) cwnd=13 (xmit 113)
- ACK 101 (due to 110) cwnd=14 (xmit 114)
- ACK 111 (due to 101) cwnd = 5 (xmit 115) ← exiting fast recovery
- Packets 111-114 already in flight
- ACK 112 (due to 111) cwnd = 5 + 1/5 ← back in cong. avoidance

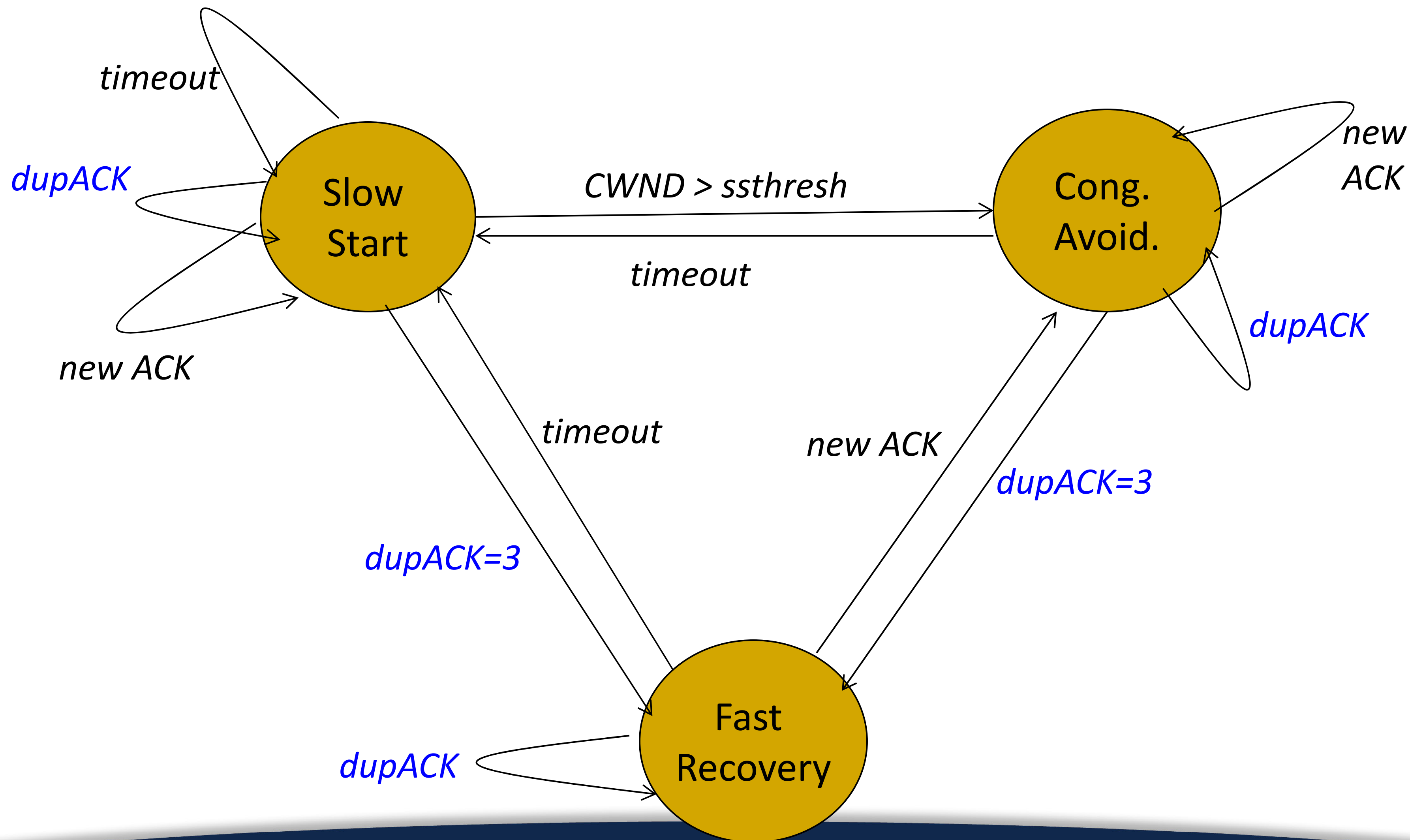
# TCP state machine



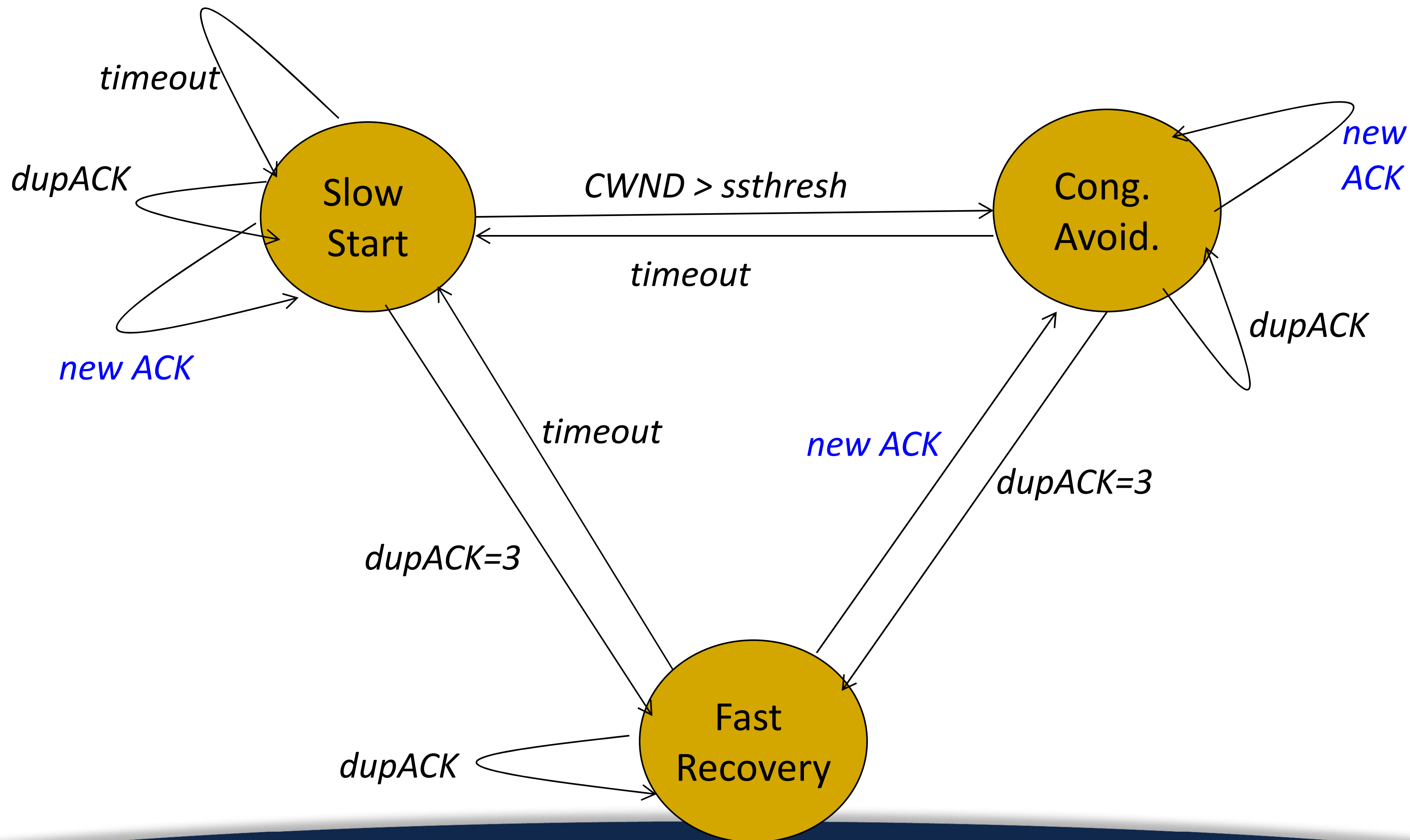
# Timeouts ➔ Slow Start



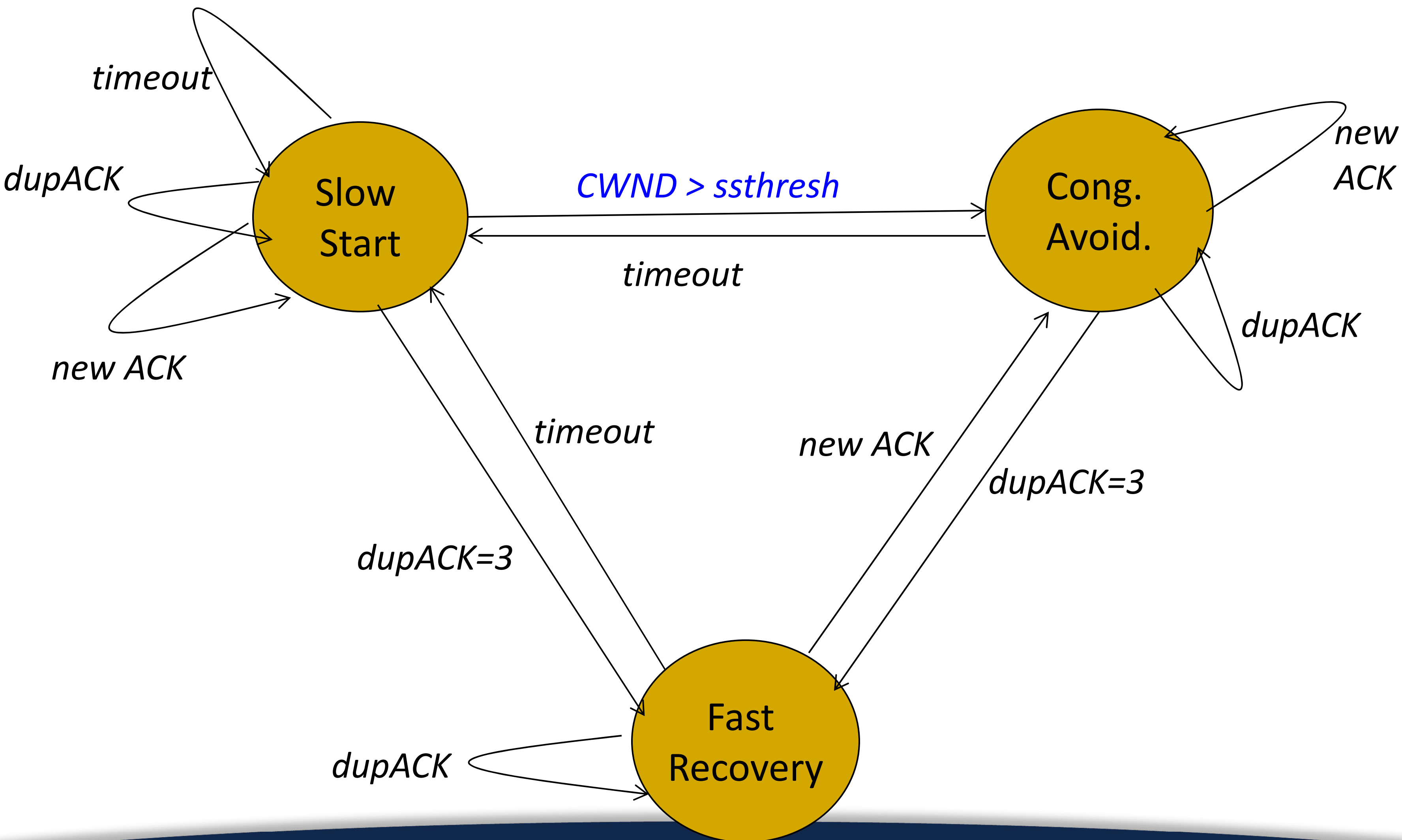
# dupACKs ➔ Fast Recovery



# New ACK changes state ONLY from Fast Recovery



# TCP state machine



# TCP flavors

- TCP-Tahoe
  - CWND =1 on 3 dupACKs
- TCP-Reno
  - CWND =1 on timeout
  - CWND = CWND/2 on 3 dupACKs
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK
  - Incorporates selective acknowledgements

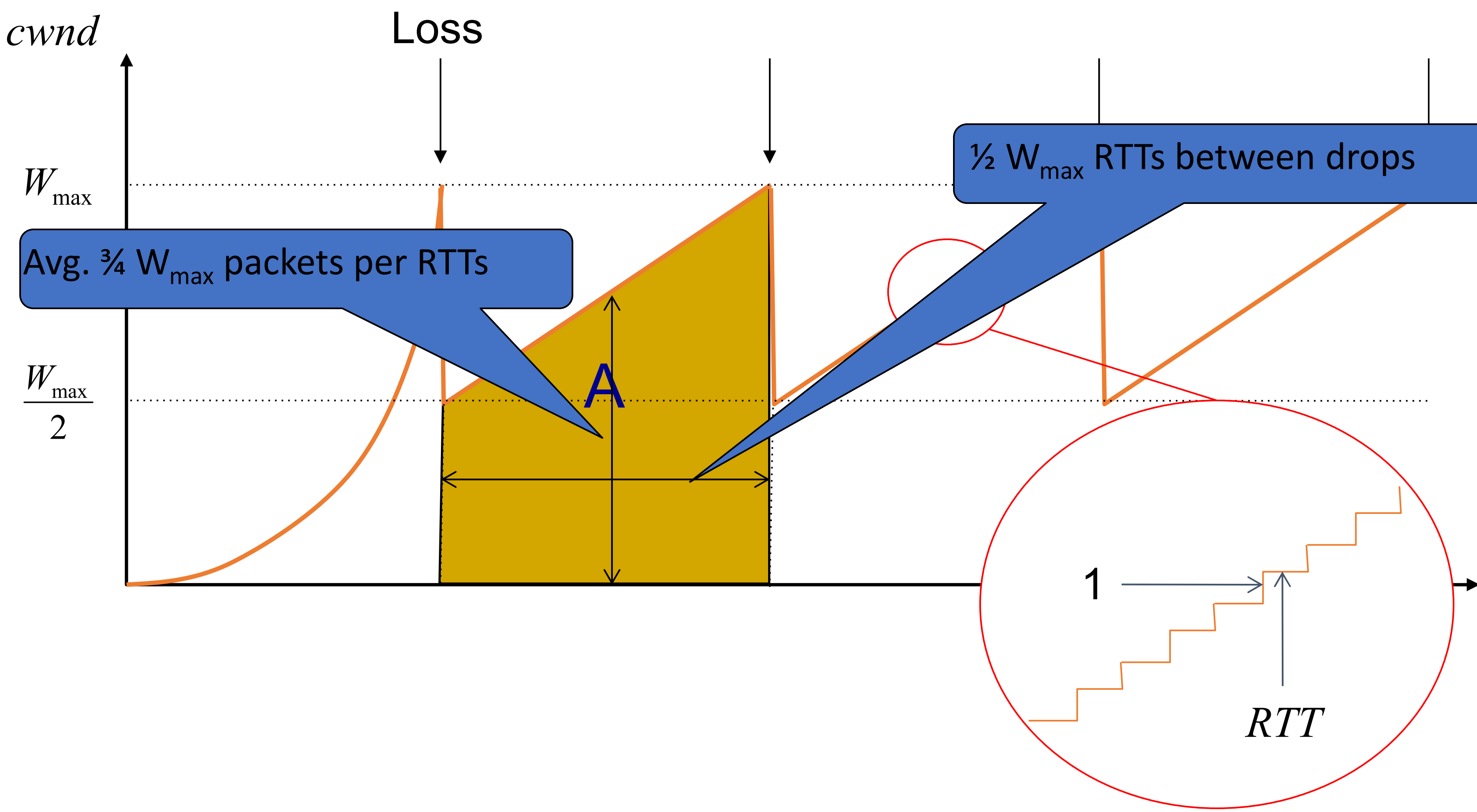
Our default assumption

# How can they coexist?

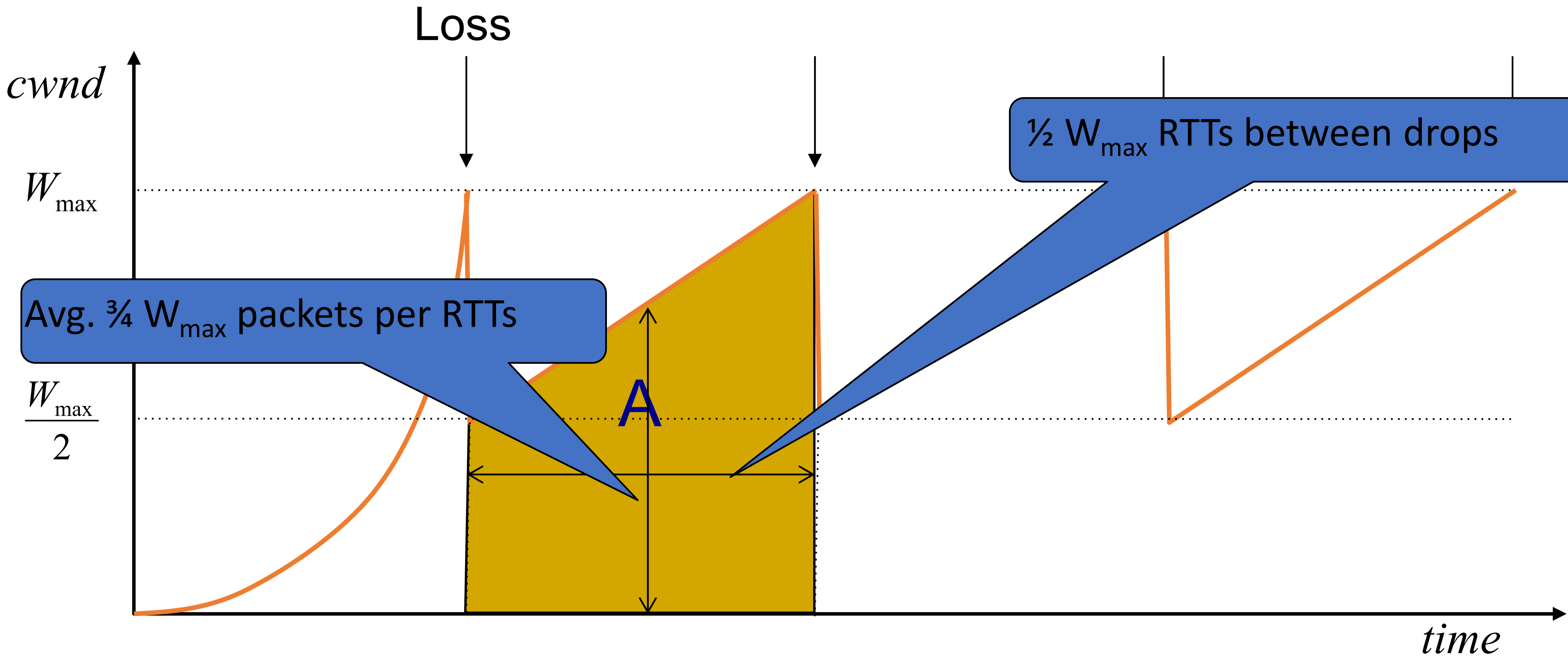
- All follow the same principle
  - Increase CWND on good news
  - Decrease CWND on bad news

# TCP Throughput Equation

# A simple model for TCP throughput



# A simple model for TCP throughput

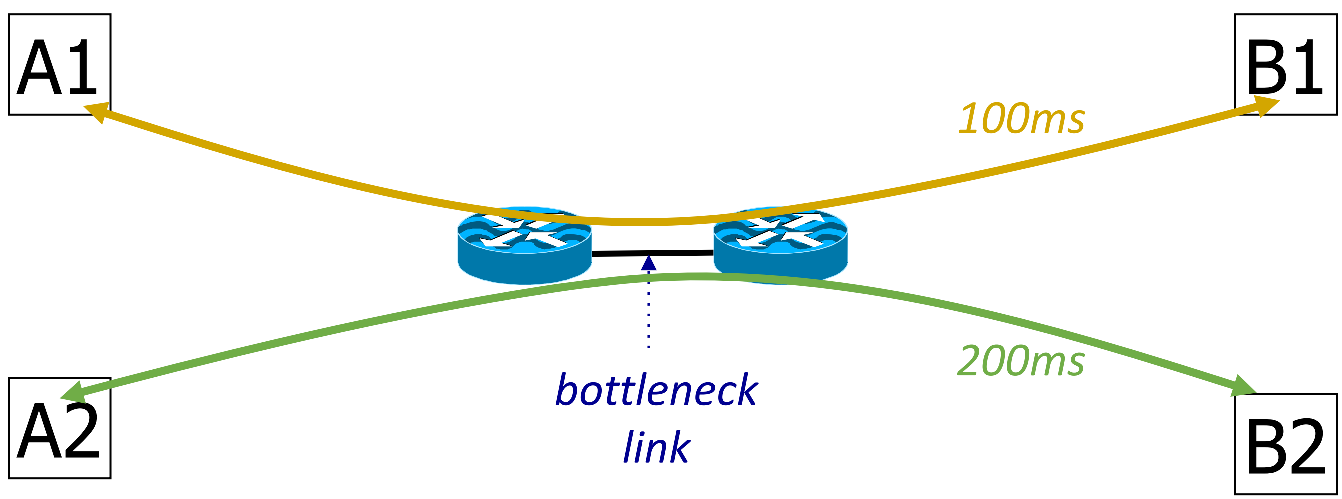



in MSS

# Implications (1): Different RTTs

$$\text{Throughput} = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

- Flows get throughput inversely proportional to RTT
- TCP unfair in the face of heterogeneous RTTs!



# Implications (2): High-speed TCP

$$\text{Throughput} = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

- Assume RTT = 100ms, MSS=1500bytes, BW=100Gbps
- What value of p is required to reach 100Gbps throughput?
  - ~ 2 x $10^{-12}$
- How long between drops?
  - ~ 16.6 hours
- How much data has been sent in this time?
  - ~ 6 petabits

# Adapting TCP to high speed

- Once past a threshold speed, increase CWND faster
  - A proposed standard [Floyd'03]: once speed is past some threshold, change equation to $p^{-.8}$ rather than $p^{-.5}$
  - Let the additive constant in AIMD depend on CWND
- Other approaches?
  - Multiple simultaneous connections (hack but works today)
  - Router-assisted approaches

# Implications (3): Rate-based CC

$$\text{Throughput} = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

- TCP throughput swings between W/2 to W
- Apps may prefer steady rates (e.g., streaming)
- "Equation-Based Congestion Control"
  - Ignore TCP's increase/decrease rules and just follow the equation
  - Measure drop percentage p, and set rate accordingly
- Following the TCP equation ensures "TCP friendliness"
  - i.e., use no more than TCP does in similar setting

# Implications (4):
# Loss not due to congestion?

- TCP will confuse corruption with congestion
- Flow will cut its rate
  - Throughput ~ 1/sqrt(p) where p is loss prob.
  - Applies even for non-congestion losses!

# Implications (5): Short flows cannot ramp up

- 50% of flows have < 1500B to send; 80% < 100KB
- Implications
  - Short flows never leave slow start!
    - They never attain their fair share
  - Too few packets to trigger dupACKs
    - Isolated loss may lead to timeouts
    - At  typical timeout values of ~500ms, might severely impact flow completion time

# Implications (6): Short flows share long delays

- A flow deliberately overshoots capacity, until it experiences a drop
- Means that delays are large, and are large for everyone
  - Consider a flow transferring a 10GB file sharing a bottleneck link with 10 flows transferring 100B
  - Larger flows dominate smaller ones

# Implications (7): Cheating

- Three easy ways to cheat
  - Increasing CWND faster than +1 MSS per RTT

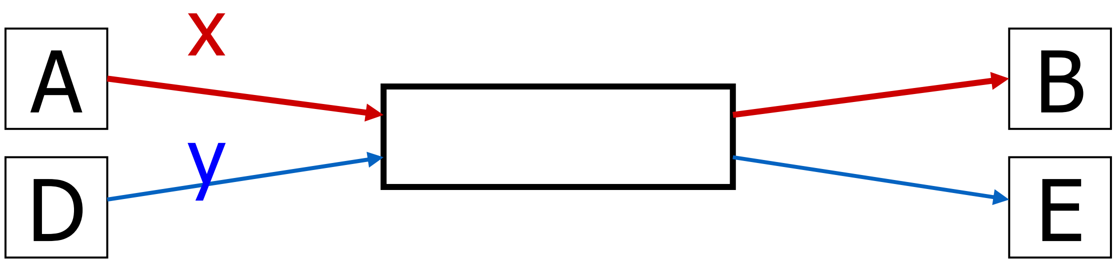
# Implications (7): Cheating

- Three easy ways to cheat
  - Increasing CWND faster than +1 MSS per RTT
  - Using large initial CWND
    - Common practice by many companies

# Implications (7): Cheating

- Three easy ways to cheat
  - Increasing CWND faster than +1 MSS per RTT
  - Using large initial CWND
    - Common practice by many companies
  - Opening many connections

# Open many connections




- Assume
  - A starts 10 connections to B
  - D starts 1 connection to E
  - Each connection gets about the same throughput

- Then A gets 10 times more throughput than D

# Implications (8):
# CC intertwined with reliability

- CWND adjusted based on ACKs and timeouts
- Cumulative ACKs and fast retransmit/recovery rules
- Complicates evolution
  - Changing from cumulative to selective ACKs is hard
- Sometimes we want CC but not reliability
  - e.g., real-time applications
- We may also want reliability without CC

# Recap: TCP problems

- Misled by non-congestion losses
- Fills up queues leading to high delays
- Short flows complete before discovering available capacity
- AIMD impractical for high speed links
- Saw tooth discovery too choppy for some apps
- Unfair under heterogeneous RTTs
- Tight coupling with reliability mechanisms
- End hosts can cheat

Routers tell endpoints if they're congested

Routers tell endpoints what rate to send at

Routers enforce fair sharing

Could fix many of these with some help from routers!

# Quiz 8 – Congestion Control

- https://forms.gle/QiNmR7DUFBaFCZMFA

# Summary

- TCP works even though it has many flaws
- Many of them can be fixed via assistance from the network

- Next: The Network Layer