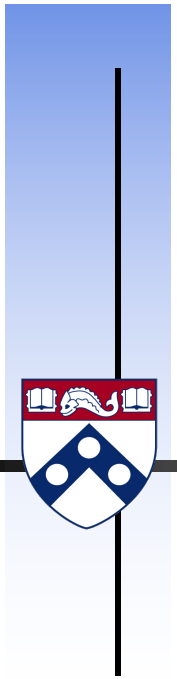


CIS 553: Networked Systems

Congestion Control

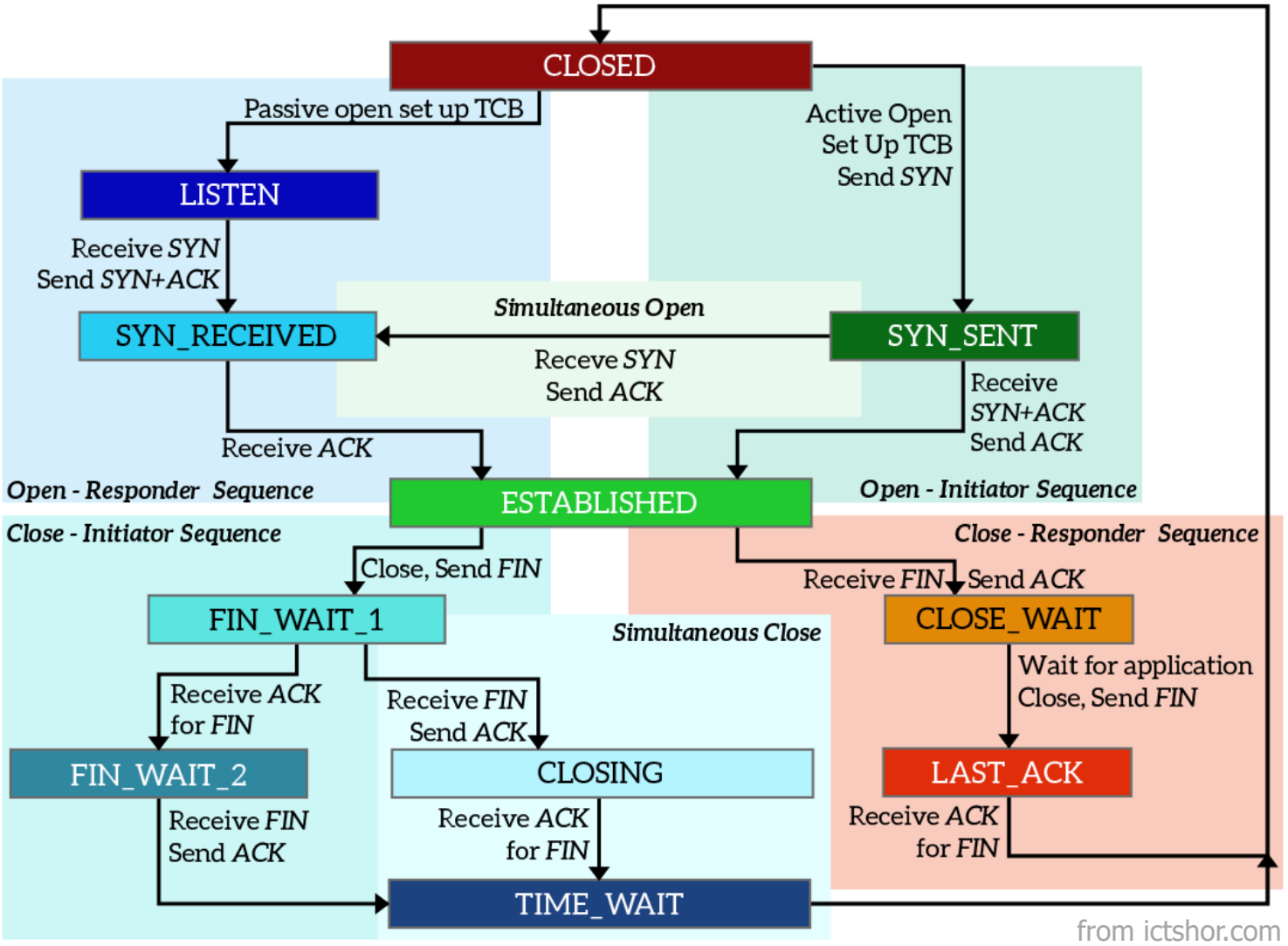
March 30, 2020

Agenda

- Transmission Control Protocol 
  - How to set timers 
  - Connection-oriented 
  - Flow control 
  - Congestion control 

  - Fairness
  - ACK Clocking

TCP State Diagram

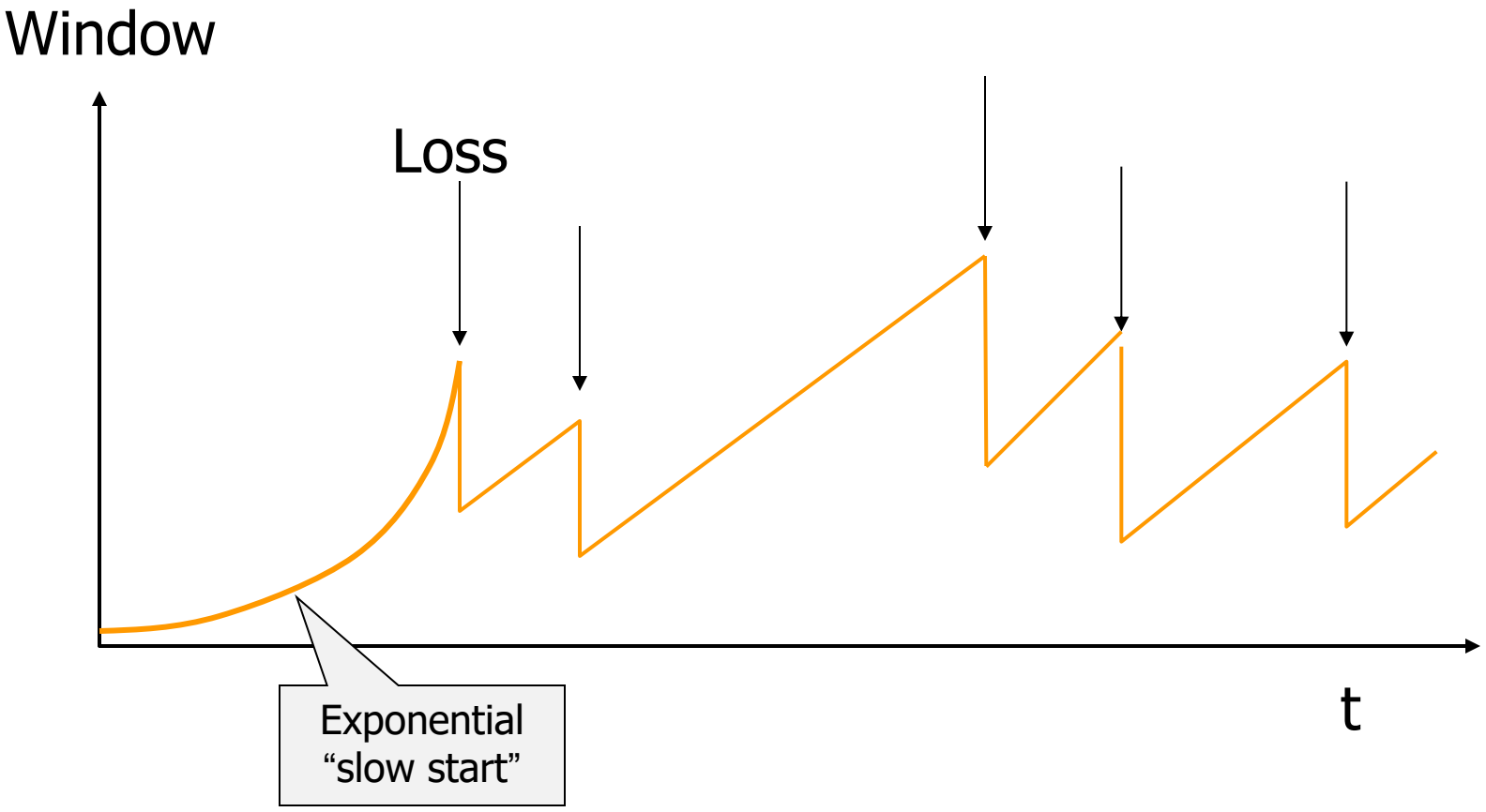


Agenda

- Transmission Control Protocol 
  - How to set timers 
  - Connection-oriented 
  - Flow control 
  - Congestion control 

  - Fairness
  - ACK Clocking

Leads to the TCP "Sawtooth"

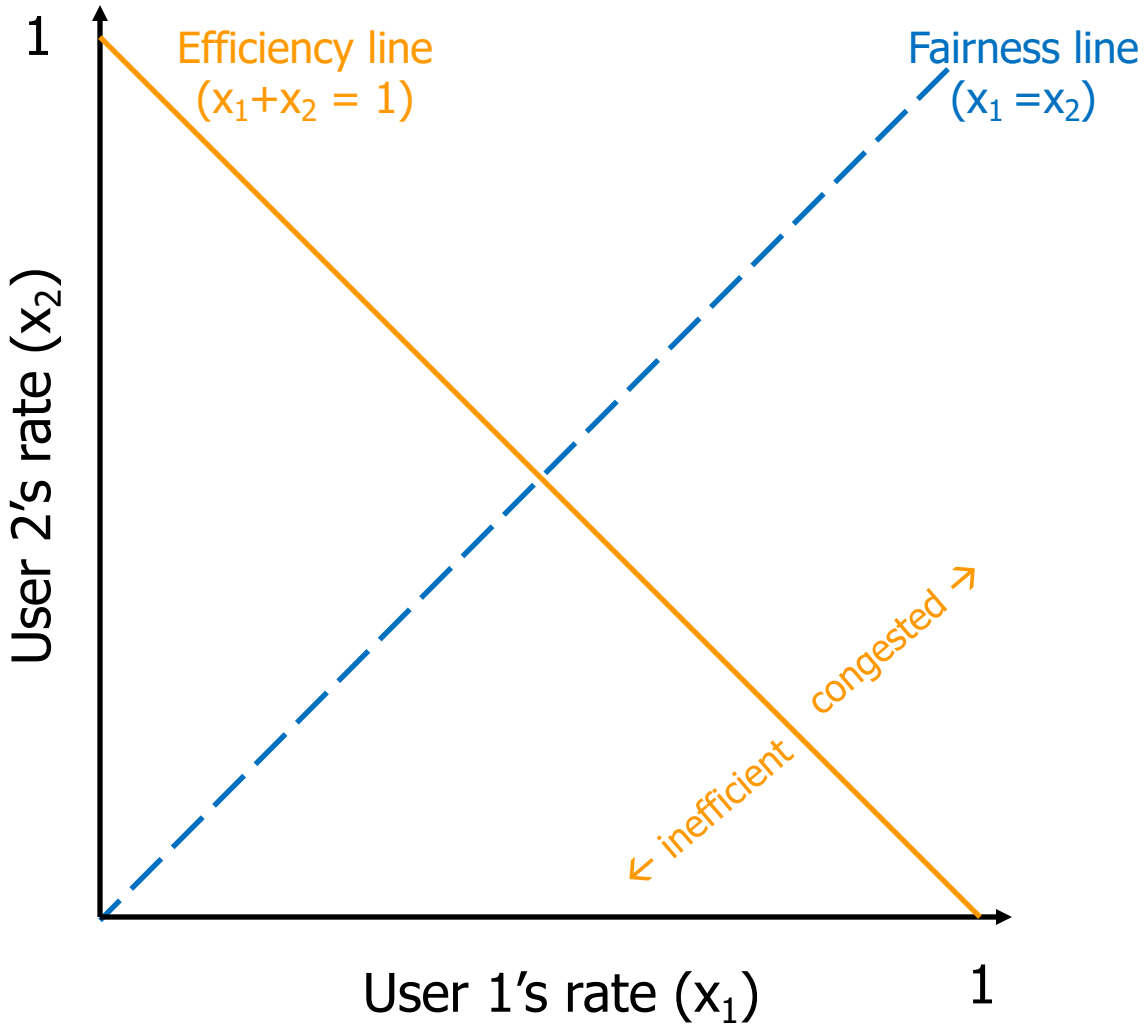


Why AIMD?

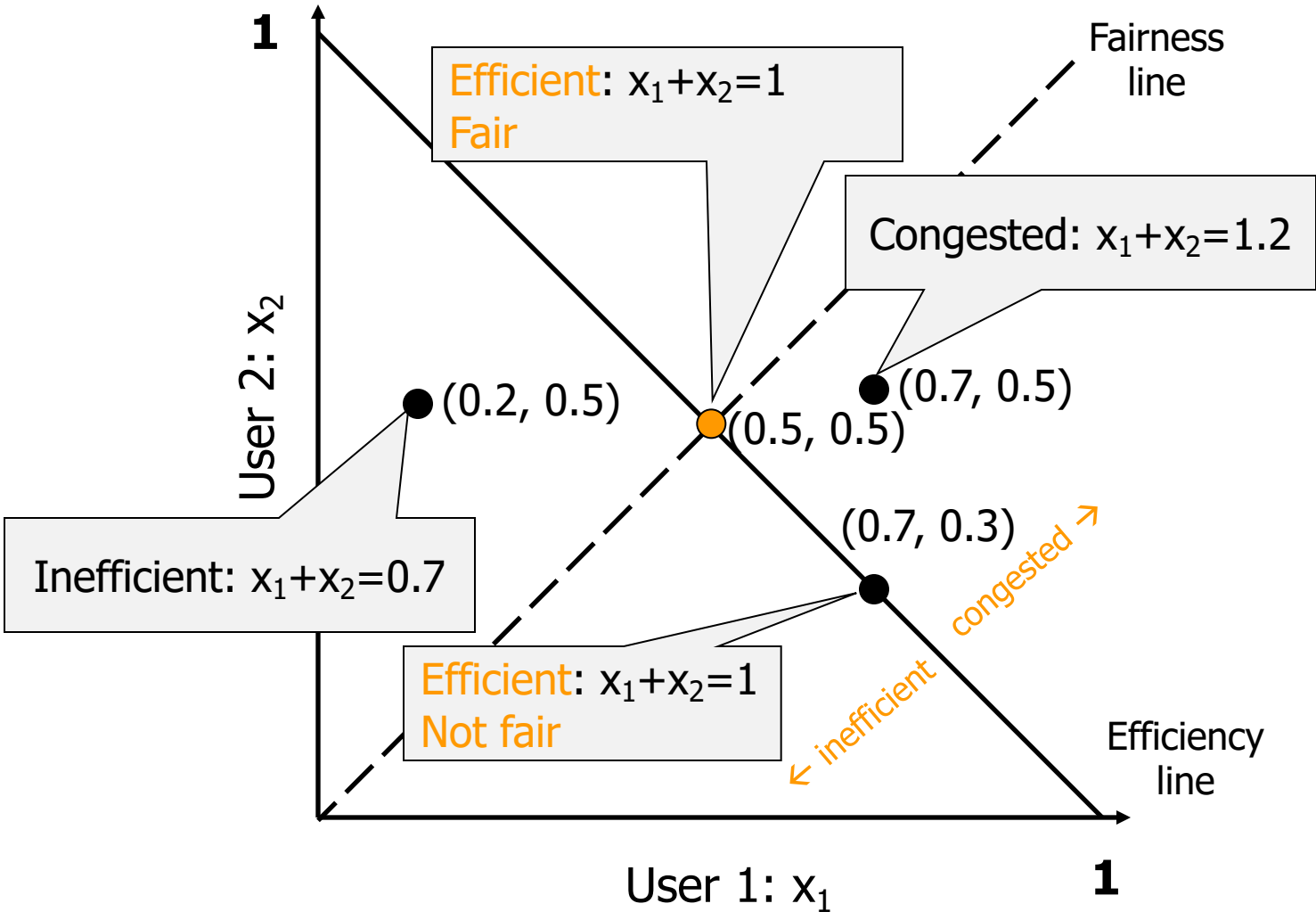
- Every RTT, we can do
  - Multiplicative increase or decrease: CWND→ a*CWND
  - Additive increase or decrease: CWND→ CWND + b
- Four alternatives:
  - AIAD: gentle increase, gentle decrease
  - AIMD: gentle increase, drastic decrease
  - MIAD: drastic increase, gentle decrease
  - MIMD: drastic increase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

Simple model of congestion control

- Two users
  - rates x_1 and x_2
- Congestion when $x_1+x_2 > 1$
- Unused capacity when $x_1+x_2 < 1$
- Fair when $x_1 = x_2$

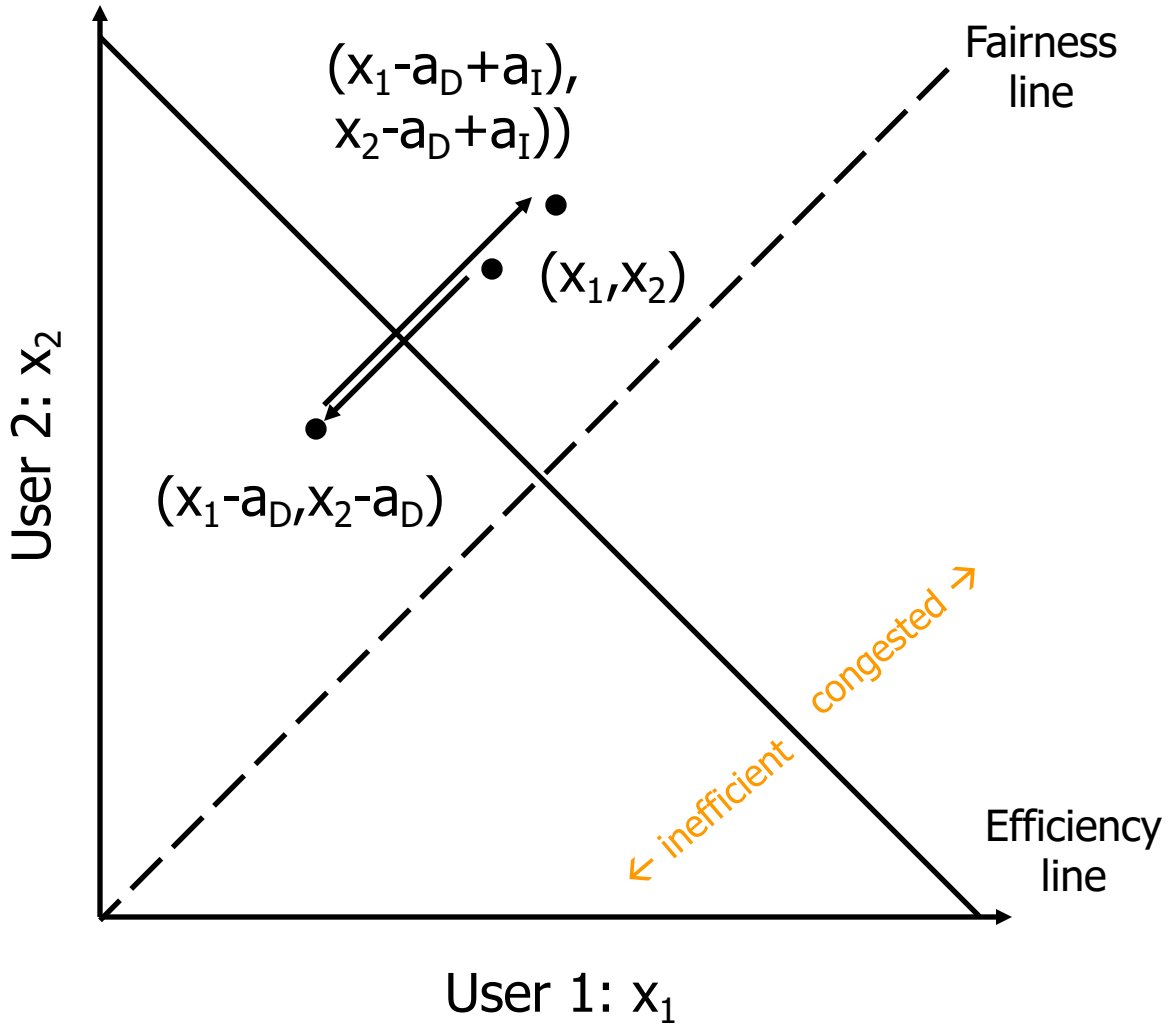


Example

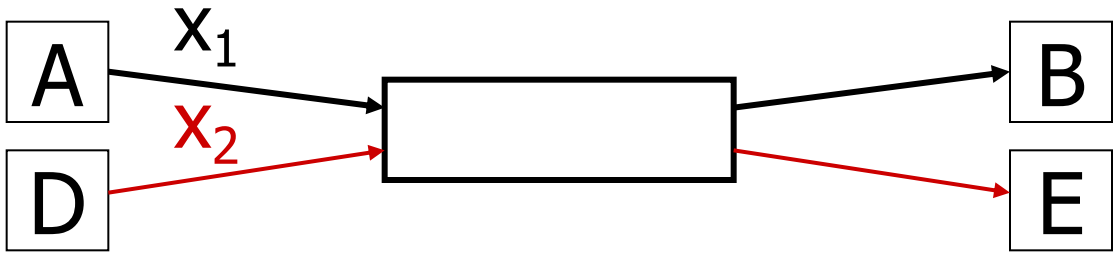

1
Efficient: $x_1+x_2=1$
Fair
Fairness line
Congested: $x_1+x_2=1.2$
User 2: x_2
(0.2, 0.5)
(0.5, 0.5)
(0.7, 0.5)
(0.7, 0.3)
congested
Inefficient: $x_1+x_2=0.7$
Efficient: $x_1+x_2=1$
Not fair
inefficient
Efficiency line
User 1: x_1
1

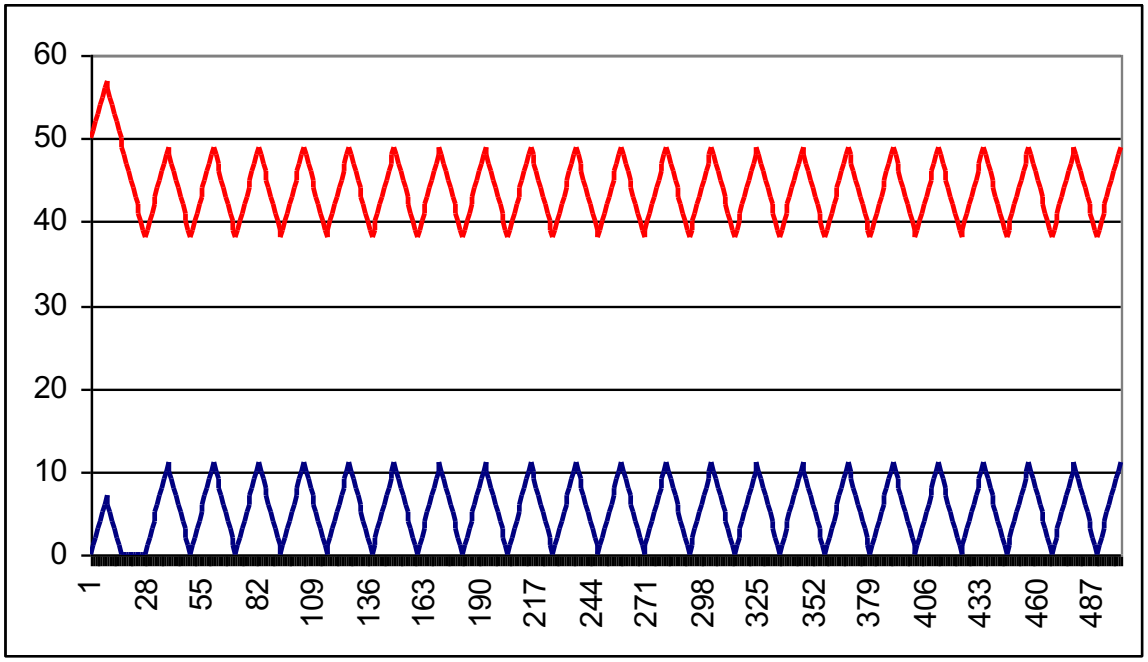
AIAD

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



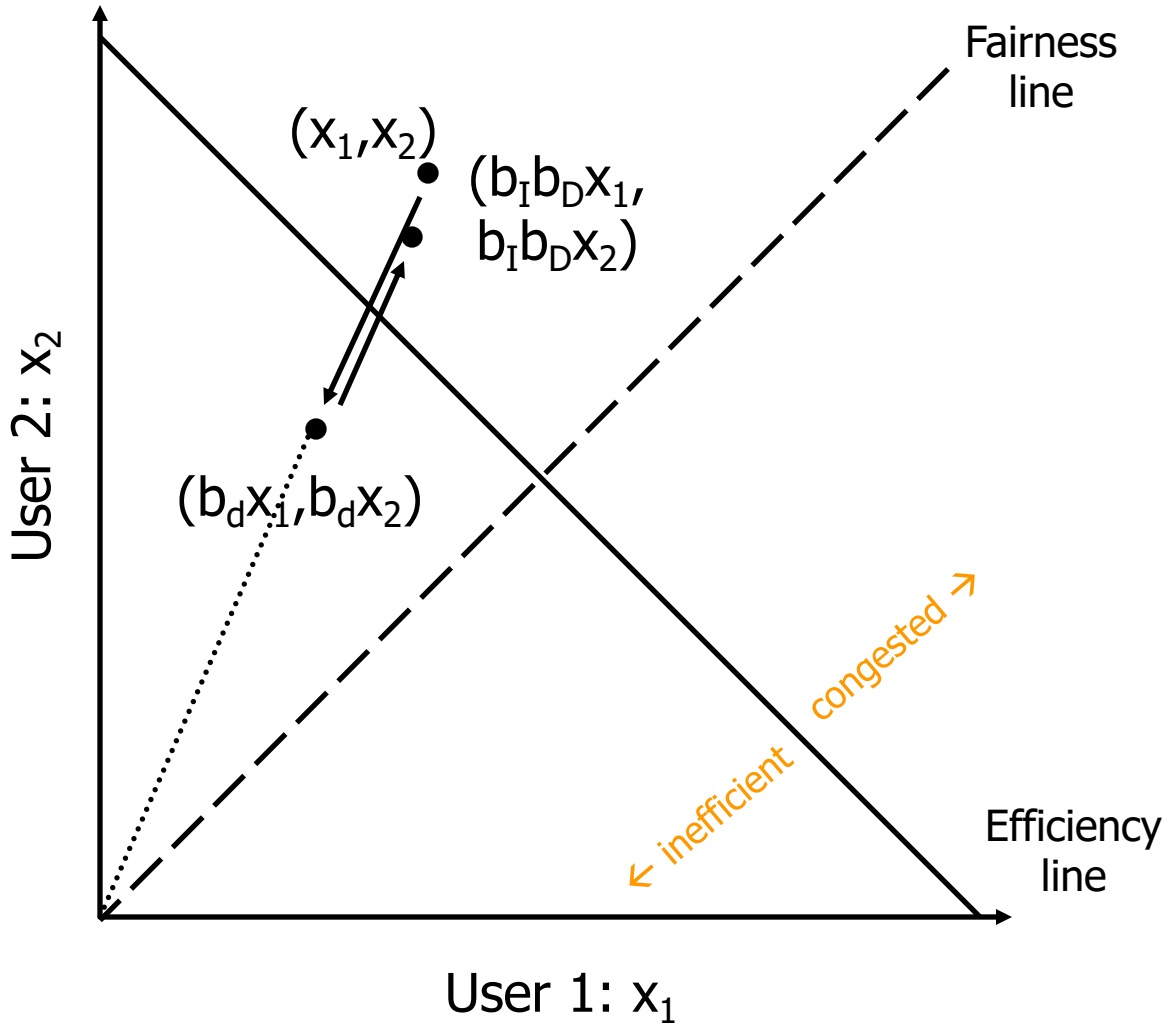
AIAD Sharing Dynamics





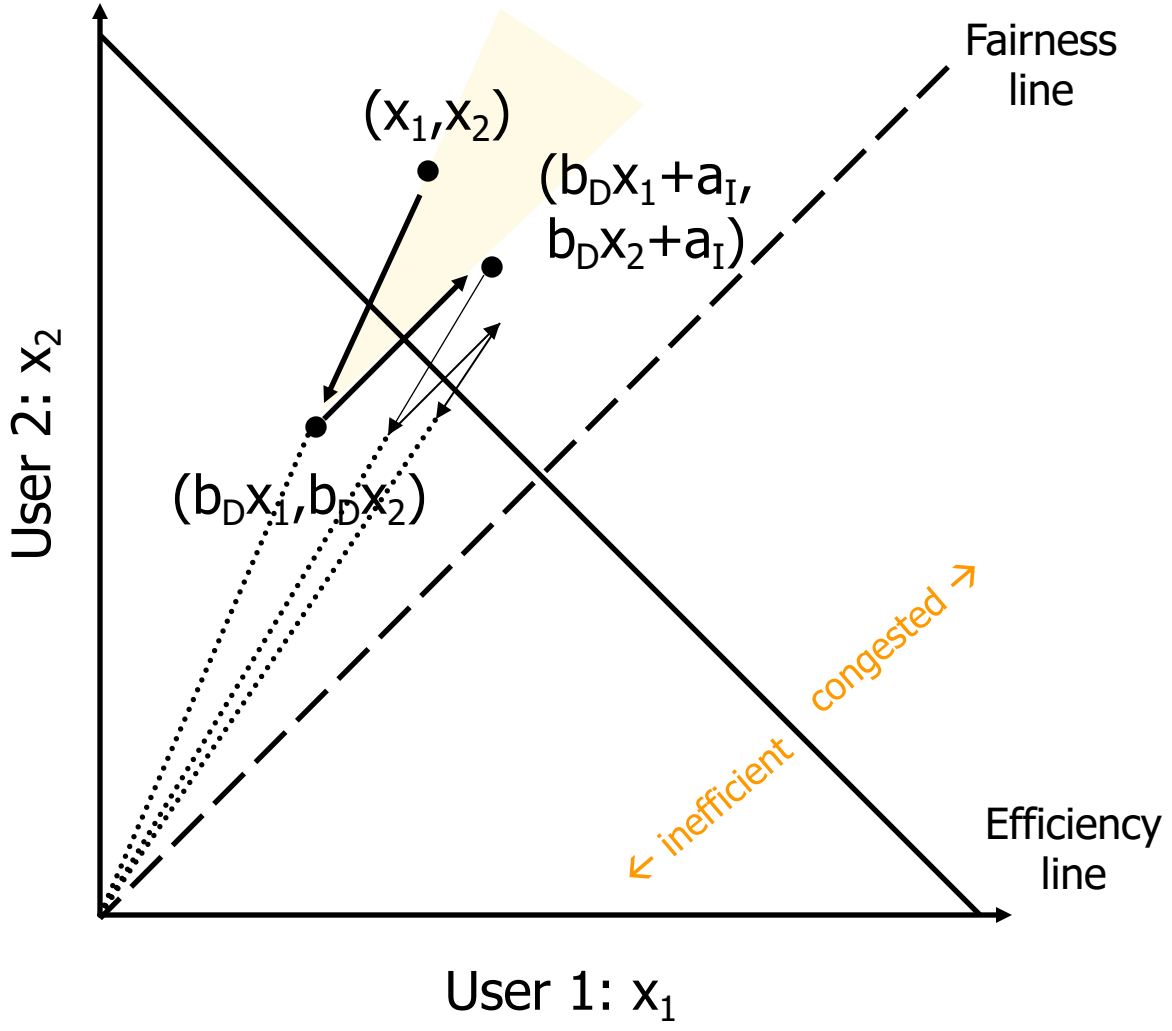
MIMD

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

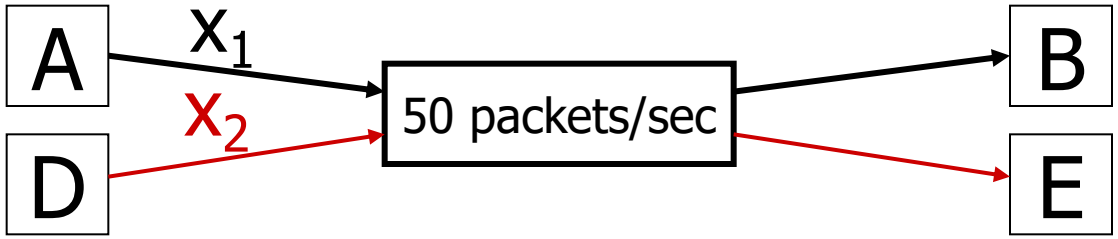


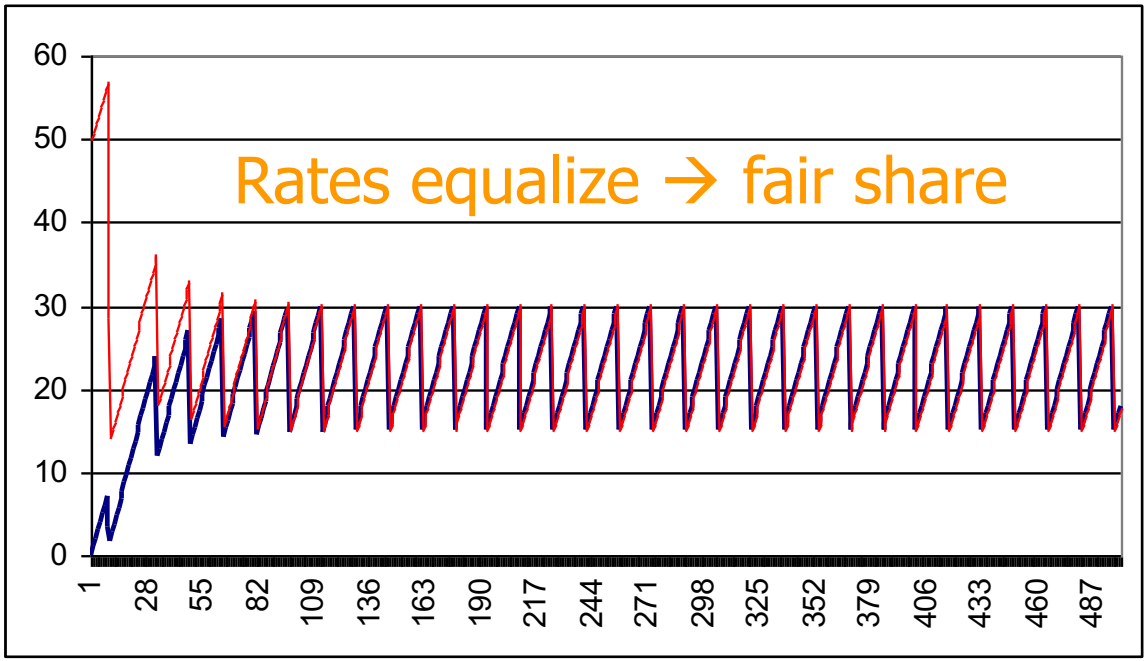
AIMD

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



AIMD Sharing Dynamics

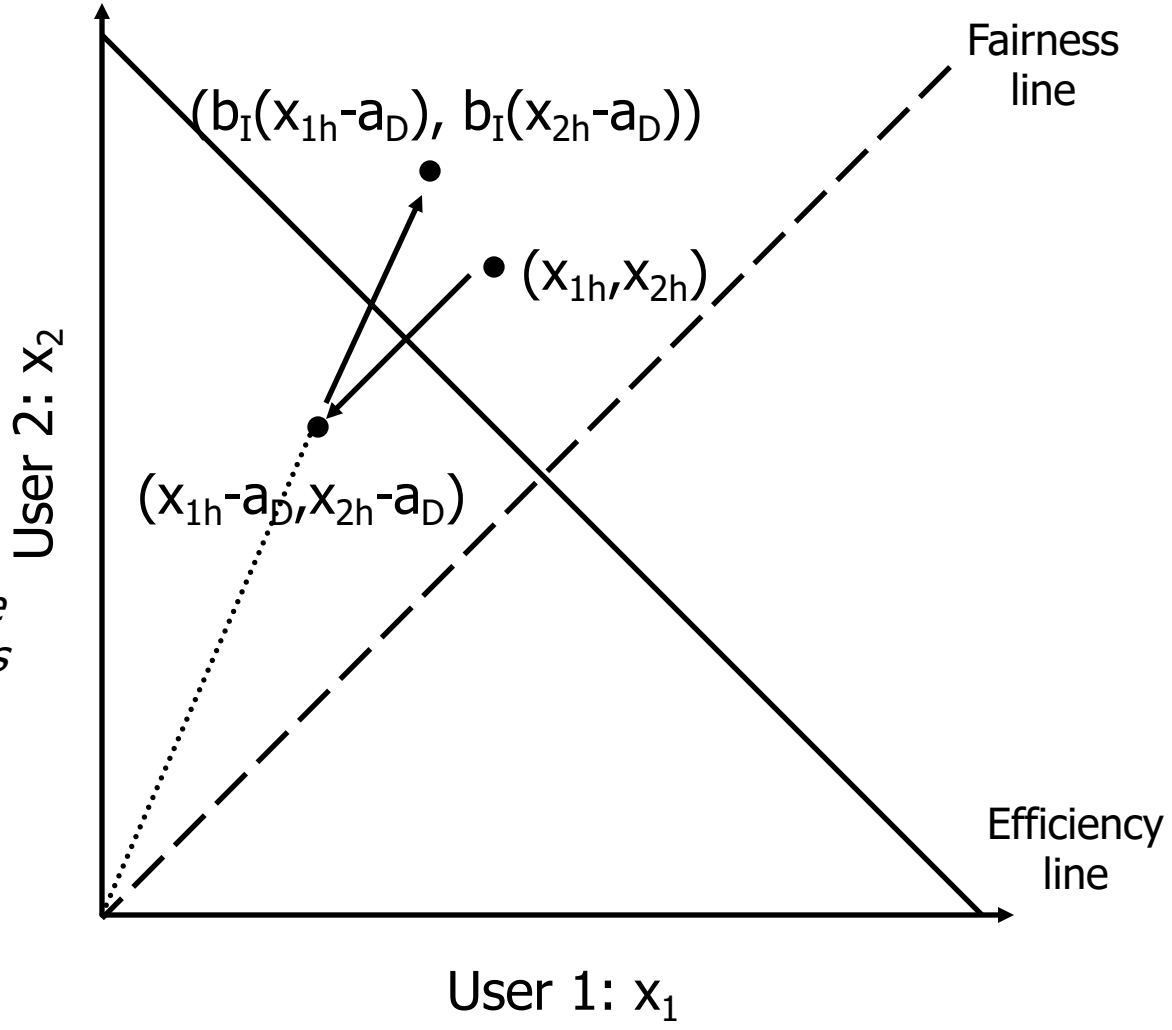




MIAD

- 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

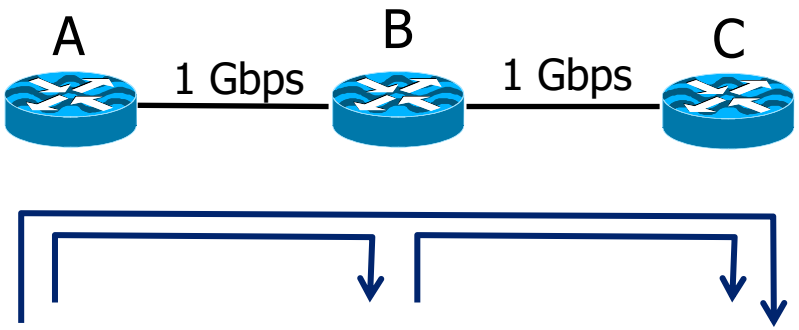


Agenda

- Transmission Control Protocol
  - Congestion control
  - Fairness NEXT
  - ACK Clocking

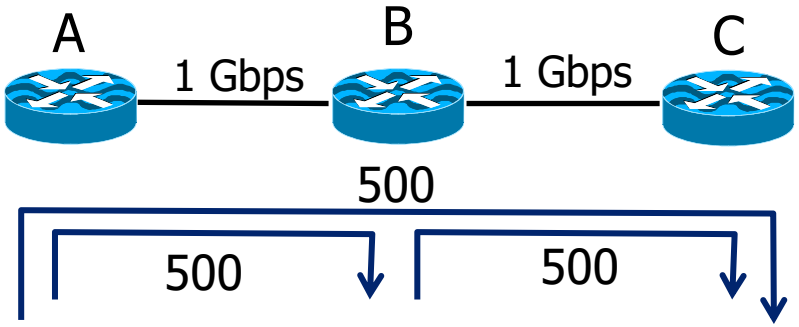
Efficiency vs. Fairness

- Cannot always have both!
  - Example network with traffic A→B, B→C and A→C
  - All three flows want to use 1 Gbps
  - How much traffic can we carry?



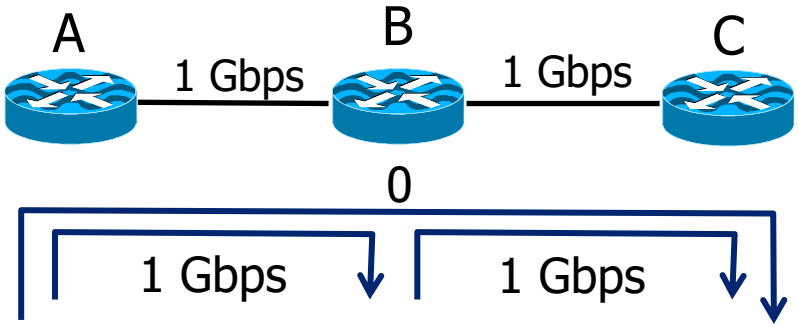
Fairness?

- How would you allocate if you only cared about fairness?
  - Give equal bandwidth to each flow
  - A→B: 500 Mbps unit, B→C: 500 Mbps, and A→C, 500 Mbps
  - Total traffic carried is 1.5 Gbps



Efficiency?

- How would you allocate if you only cared about efficiency?
  - Maximize total traffic in network
  - A→B: 1 Gbps, B→C: 1 Gbps, and A→C: 0
  - Total traffic rises to 2 Gbps!



Max-Min fairness

- For a single link, given set of bandwidth demands r_i and total bandwidth C
  - Allocation $a_i = \min(f, r_i)$
  - where f is the unique value such that $\text{Sum}(a_i) = C$
- If you don't get full demand, no one gets more than you
- This is what round-robin service gives if all packets are the same size

Computing Max-Min Fairness

- To find it given a network, imagine "pouring water into the network"
  1. Start with all flows at rate 0
  2. Increase the flows until there is a new bottleneck in the network
  3. Hold fixed the rate of the flows that are bottlenecked
  4. Go to step 2 for any remaining flows

Agenda

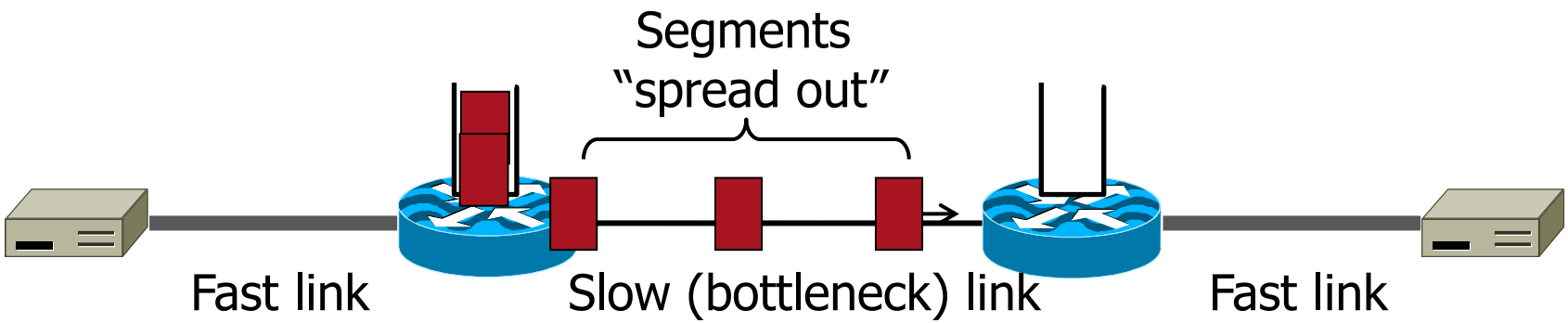
- Transmission Control Protocol ✓
  - Congestion control ✓
  - Fairness ✓
  - ACK Clocking ← NEXT

ACK Clocking

- Consider what happens when sender injects a burst of segments into the network

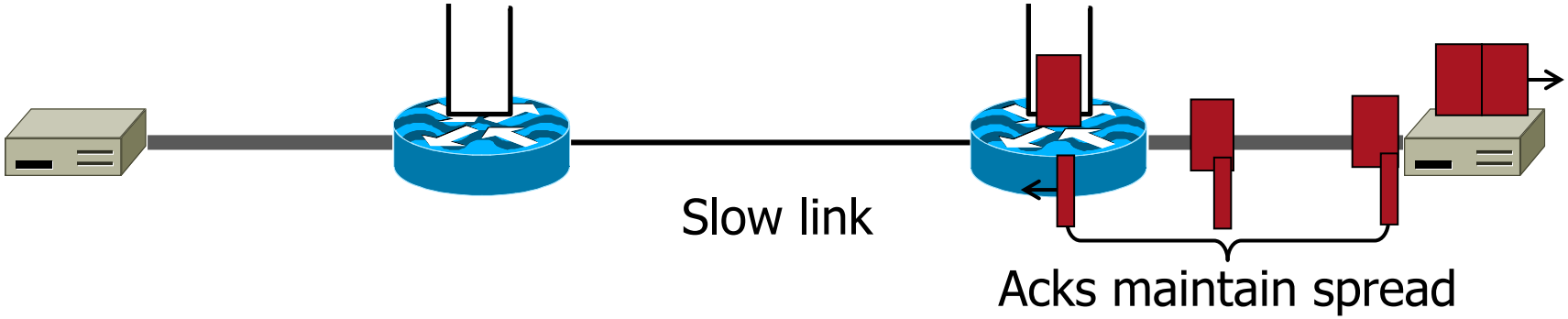
ACK Clocking

- Segments are buffered and spread out on slow link



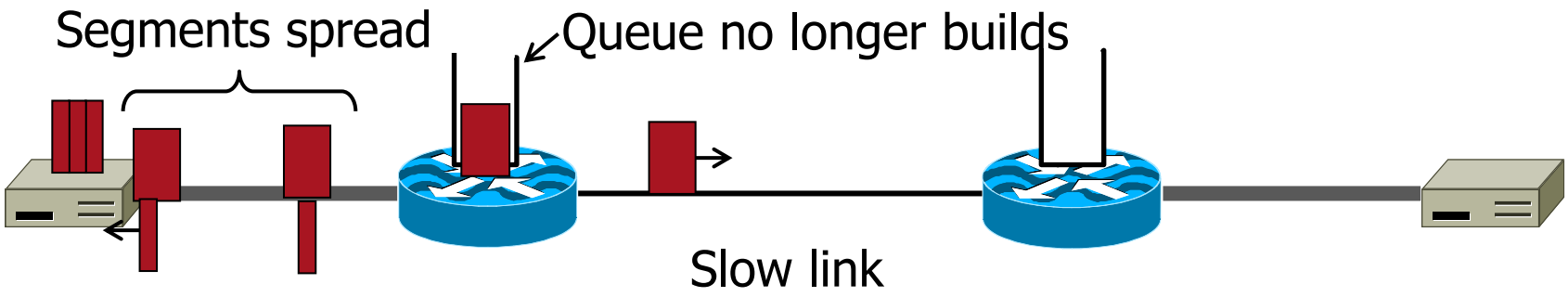
ACK Clocking

- ACKs maintain the spread back to the original sender



ACK Clocking

- Sender clocks new segments with the spread
  - Now sending at the bottleneck link without queuing!



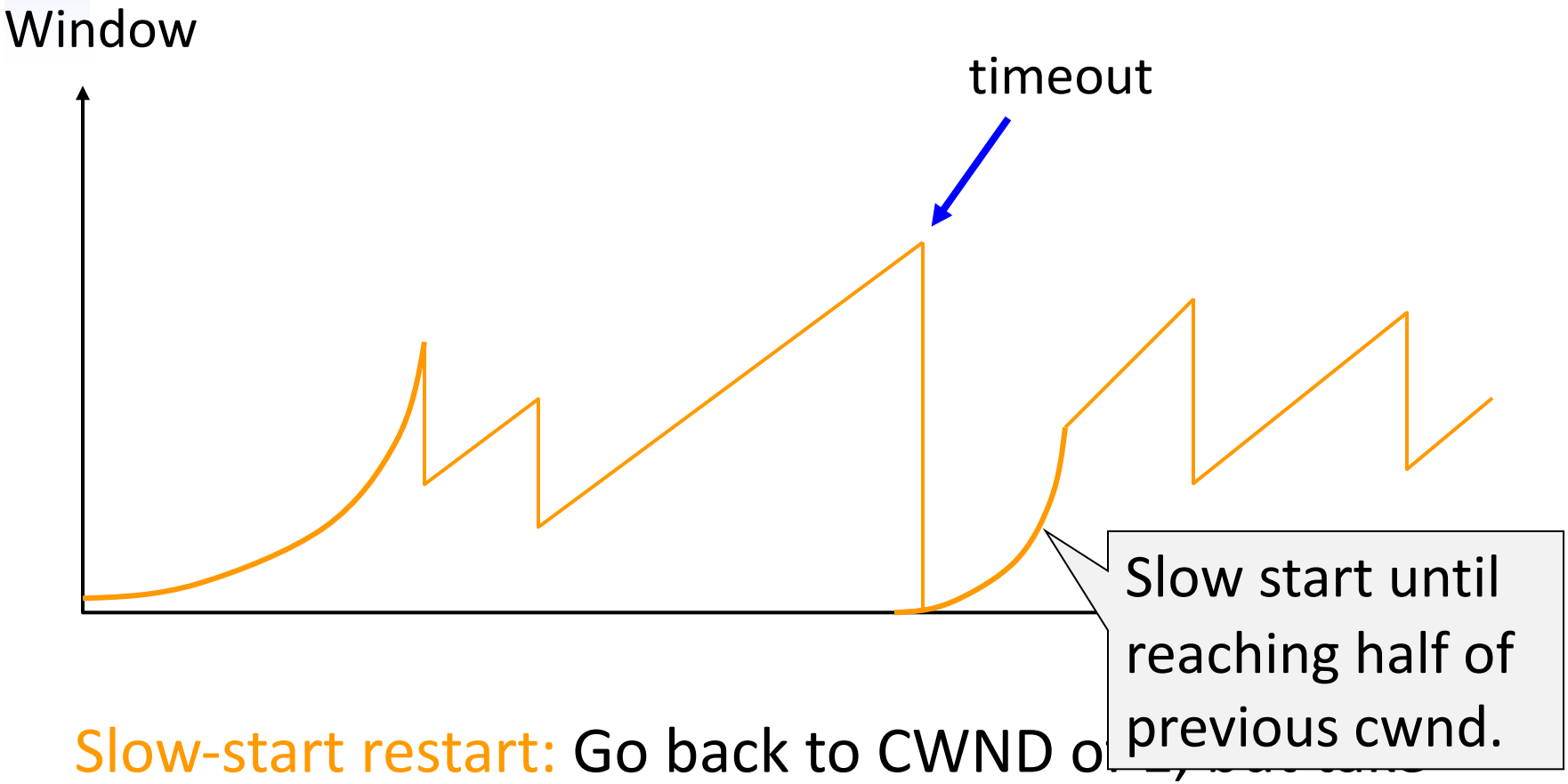
ACK Clocking

- Helps the network run with low levels of loss and delay!
- The network has smoothed out the burst of data segments
- ACK clock transfers this smooth timing back to the sender
- Subsequent data segments are not sent in bursts so do not queue up in the network

Timeouts and Idle Periods

- After a timeout or idle period:
  - We lose ACK clocking!
  - Also, 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

Repeating Slow Start After Idleness



Slow-start restart: Go back to CWND o[illegible] advantage of knowing the previous value of CWND.

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 + 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
- Notion of TCP-friendliness