EN.601.414/614 Computer Networks

Flow and Congestion Control

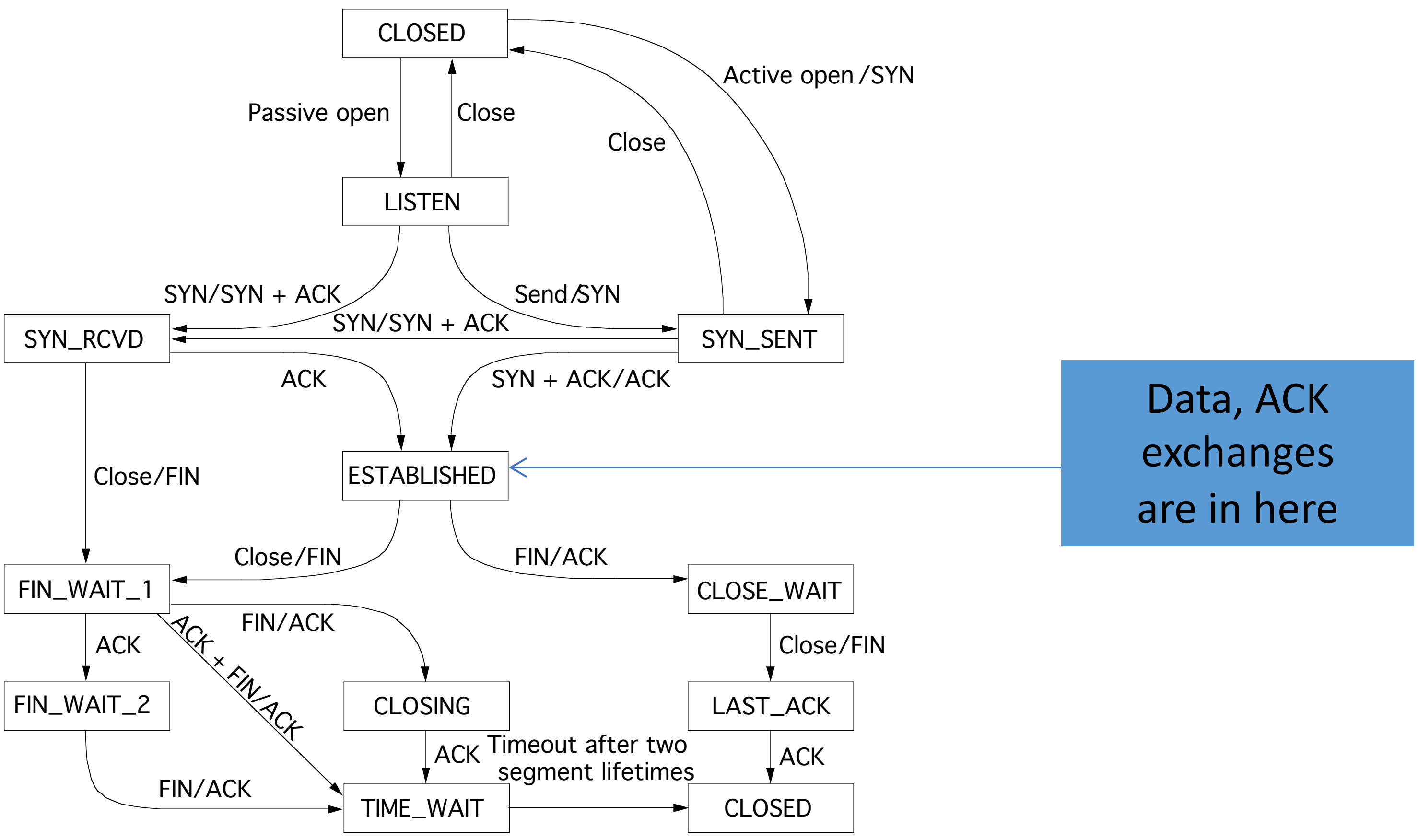
Xin Jin

Fall 2020 (TuTh 1:30-2:45pm on Zoom)

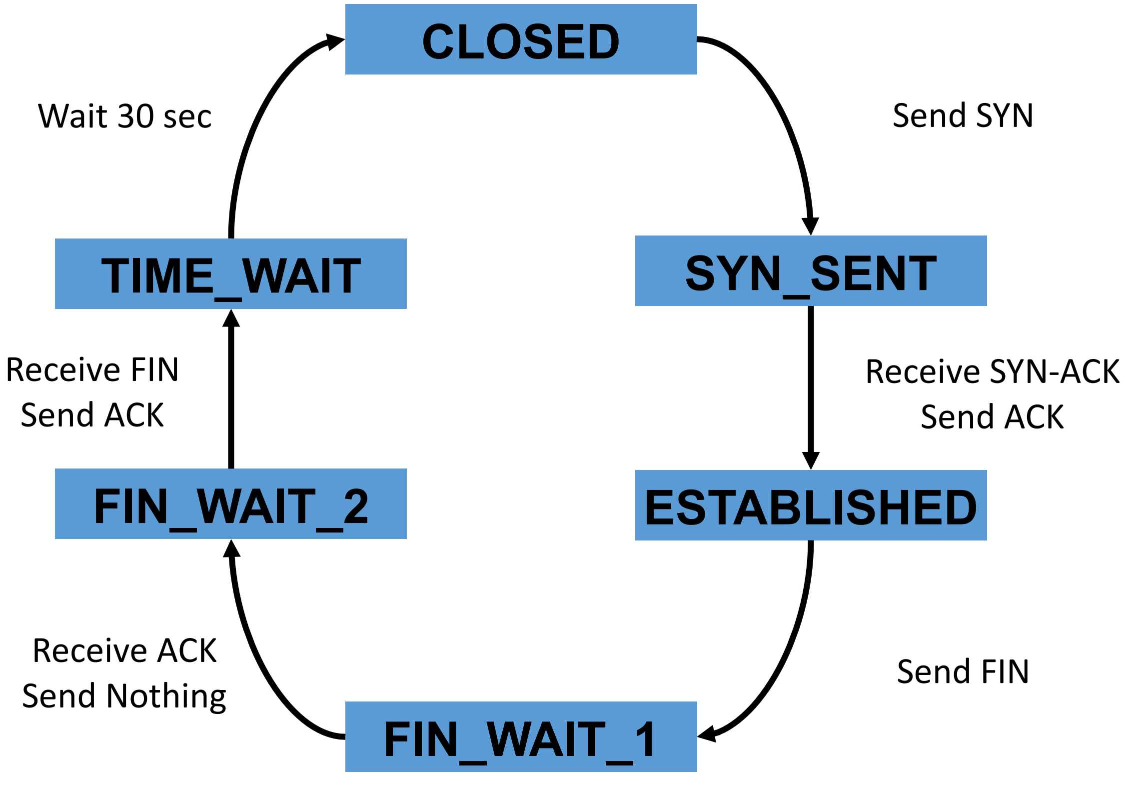
JOHNS HOPKINS UNIVERSITY

https://github.com/xinjin/course-net

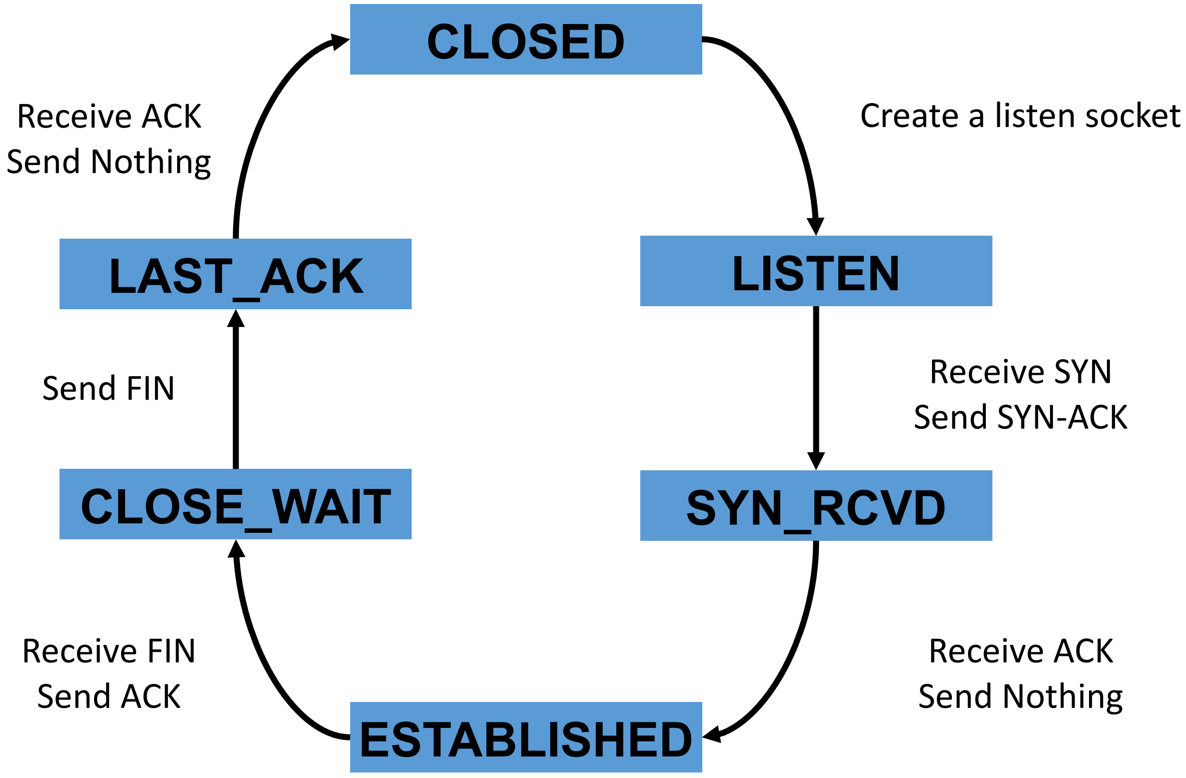
Recap: TCP state transitions



TCP client lifecycle



TCP server lifecycle

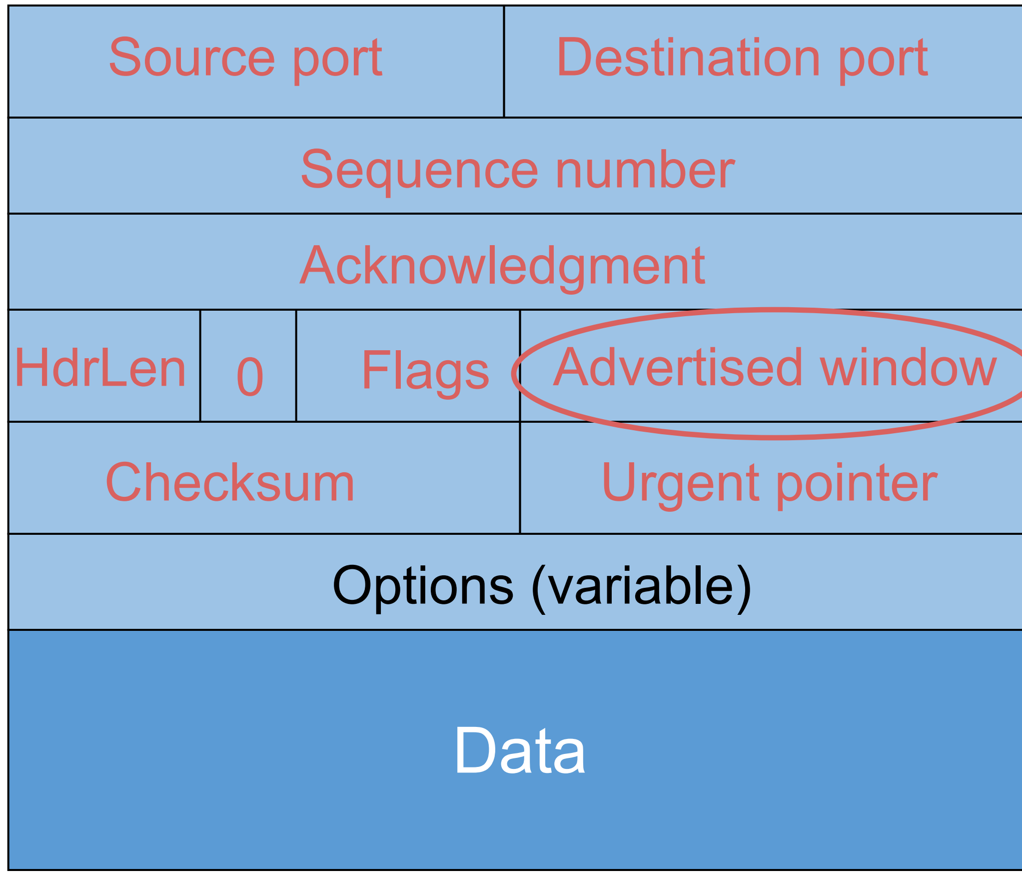


Agenda

- **TCP flow control**
- **TCP congestion control**

TCP Flow Control

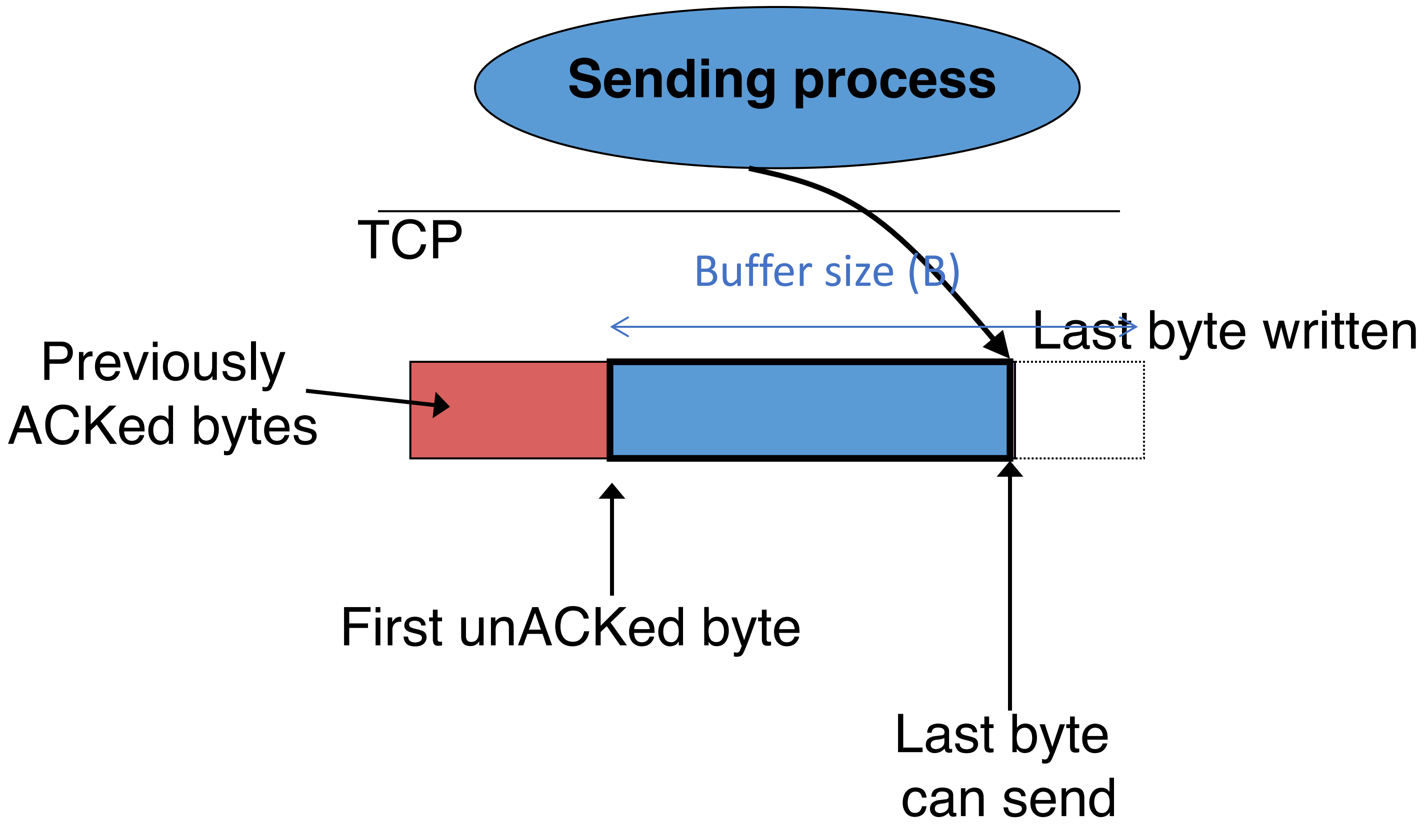
TCP header


Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

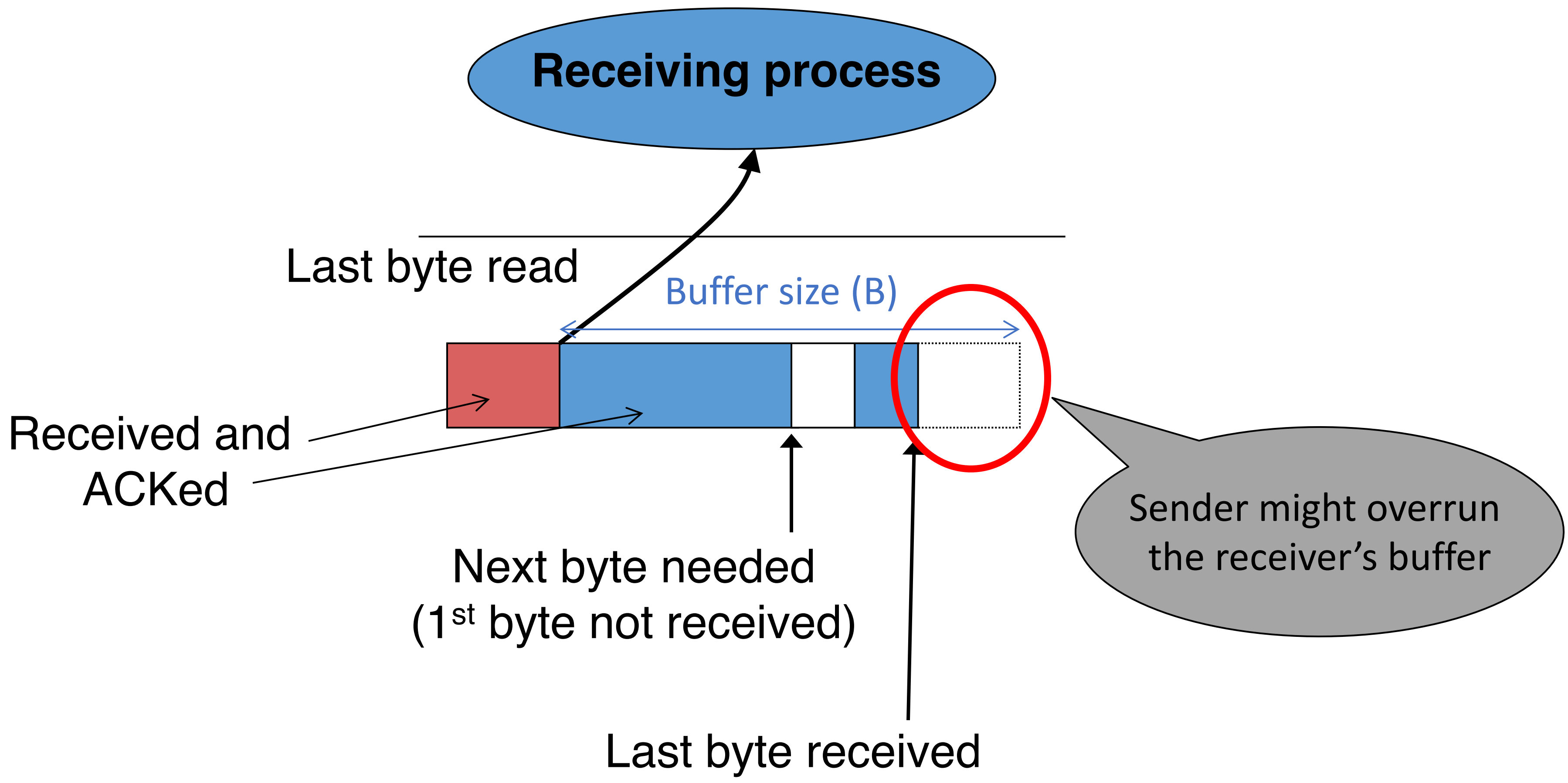
Recap: Sliding window

- **Both sender and receiver maintain a window**
- **Left edge of window:**
  - ➢ Sender: beginning of unacknowledged data
  - ➢ Receiver: beginning of undelivered/expected data
    - First "hole" in received data
    - When sender gets ack, knows that receiver's window has moved
- **Right edge: Left edge + constant**
  - ➢ The constant is only limited by buffer size in the transport layer

Sliding window at sender



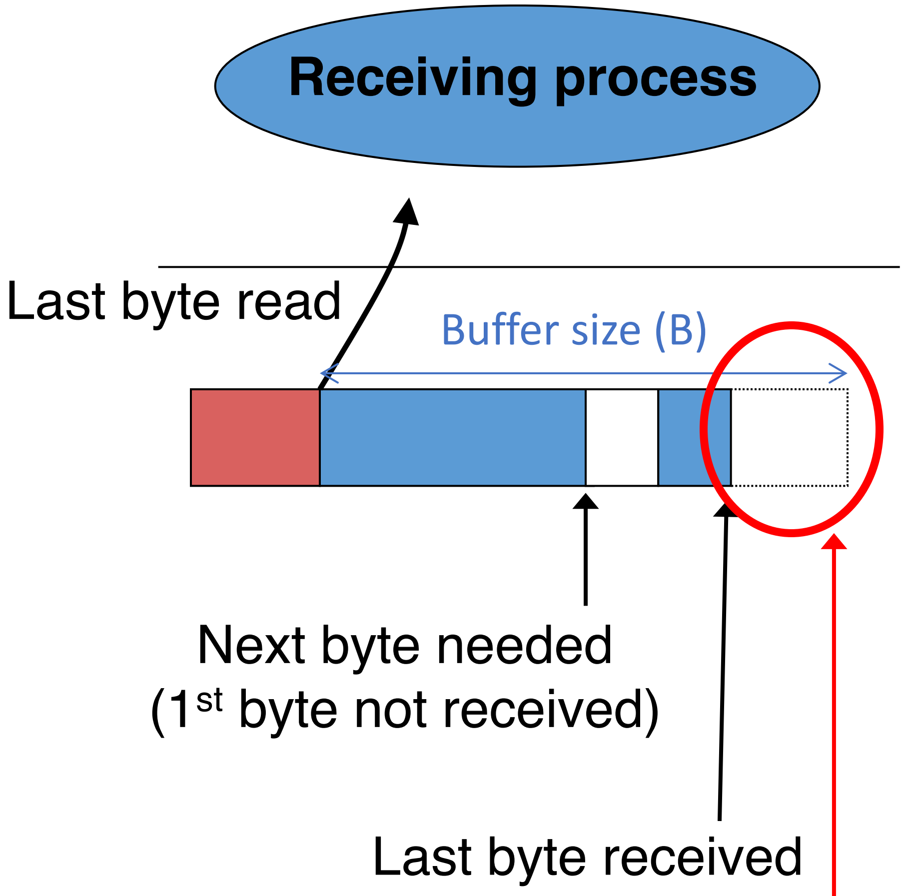
Sliding window at receiver



Solution: Advertised window (Flow Control)

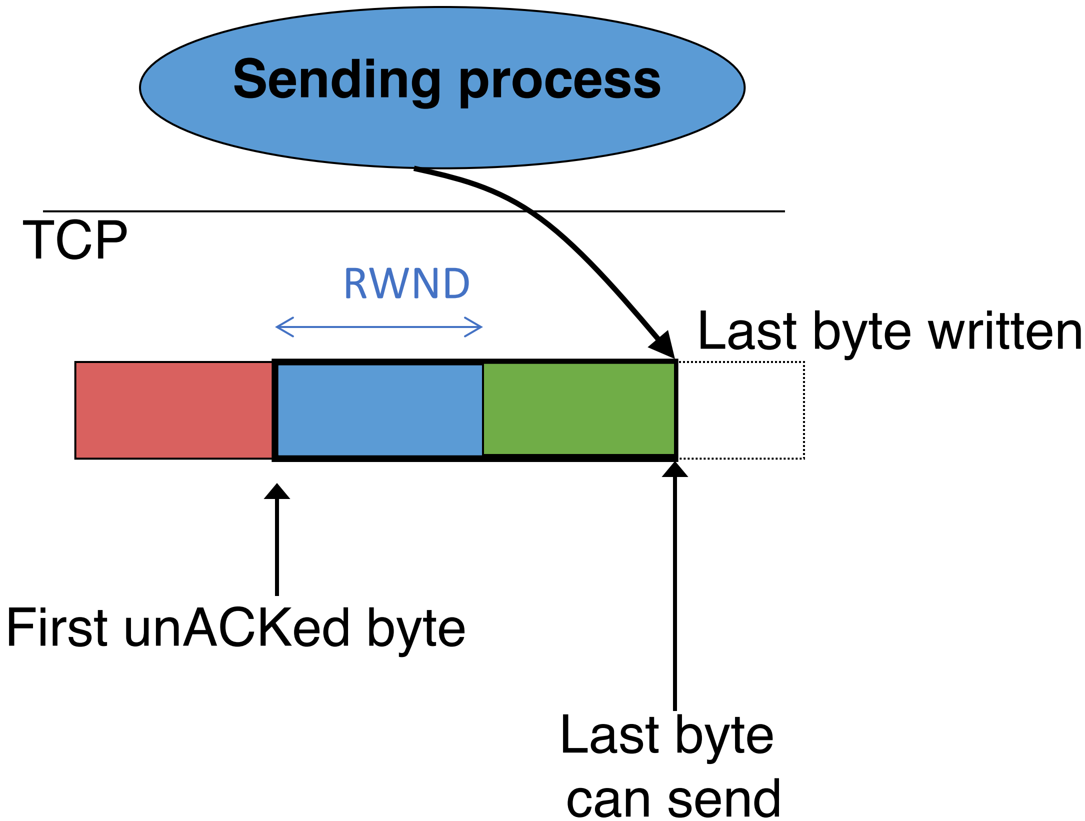
- **Receiver uses an "Advertised Window" (RWND) to prevent sender from overflowing its window**
  - Receiver indicates value of RWND in ACKs
  - Sender ensures that the total number of bytes in flight <= RWND

Sliding window at receiver



RWND = B - (LastByteReceived - LastByteRead)

Sliding window at sender



Sliding window with flow control

- **Sender: window advances when new data ACK'd**
- **Receiver: window advances as receiving process consumes data**
- **Receiver advertises to the sender where the receiver window currently ends ("righthand edge")**
  - Sender agrees not to exceed this amount
- **UDP does not have flow control**
  - Data can be lost due to buffer overflow

Advertised window limits rate

- **Sender can send no faster than RWND/RTT bytes/sec**
- **Receiver only advertises more space when it has consumed old arriving data**
- **What happens when RWND=0?**
  - Sender keeps probing with one data bytes

- **In original TCP design, that was the sole protocol mechanism controlling sender's rate**
  - What's missing?

TCP Congestion Control

What is congestion?

- **If two packets arrive at a router at the same time**
  - Router will transmit one and buffer/drop the other
- **Internet traffic is bursty**
  - Many packets can arrive close in time
  - Causes packet delays and drops

- **Root cause: statistical multiplexing**

Congestion collapse in 1980s

- **Sending rate only limited by flow control**
  - Dropped packets → senders (repeatedly!) retransmit
- **Led to "congestion collapse" in Oct. 1986**
  - Throughput on the NSF network dropped from 32Kbits/s to 40bits/sec
- **"Fixed" by Van Jacobson's development of TCP's congestion control (CC) algorithms**

Jacobson's fix to TCP

- **Extend TCP's existing window-based protocol but adapt the window size in response to congestion**
- **A pragmatic and effective solution**
  - Required no upgrades to routers or applications!
  - Patch of a few lines of code to TCP implementations
- **Extensively researched and improved upon**
  - Especially now with datacenters and cloud services

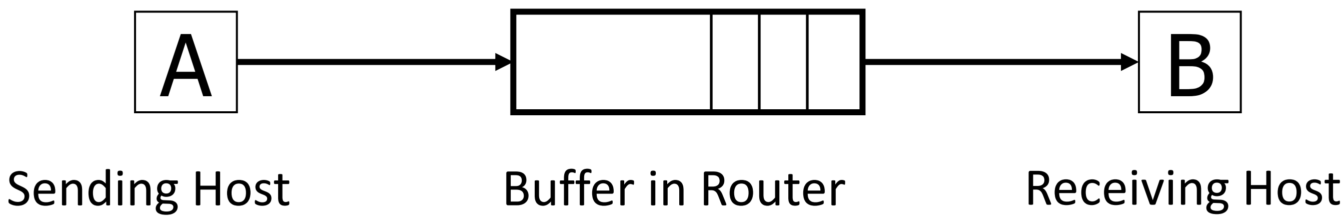
Key design considerations

- **How do we know the network is congested?**
  - ➢Implicit and/or explicit signals from the network
- **Who takes care of congestion?**
  - ➢End hosts (may receive some help from the network)
- **How do we handle congestion?**
  - ➢Continuous adaptation

Three issues to consider

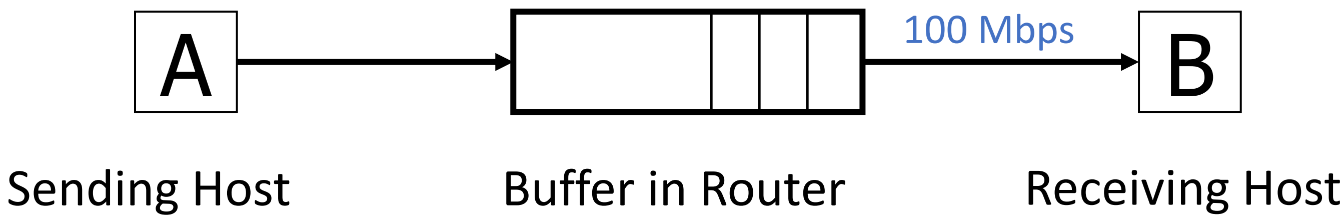
- **Discovering the available (bottleneck) bandwidth**
- **Adjusting to variations in bandwidth**
- **Sharing bandwidth between flows**

Abstract view



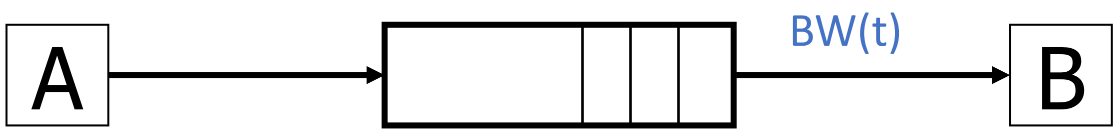
- **Ignore internal structure of router and model it as a single queue for a particular input-output pair**

Discovering available bandwidth



- **Pick sending rate to match bottleneck bandwidth**
    - ➢Without any a priori knowledge
    - ➢Could be gigabit link, could be a modem

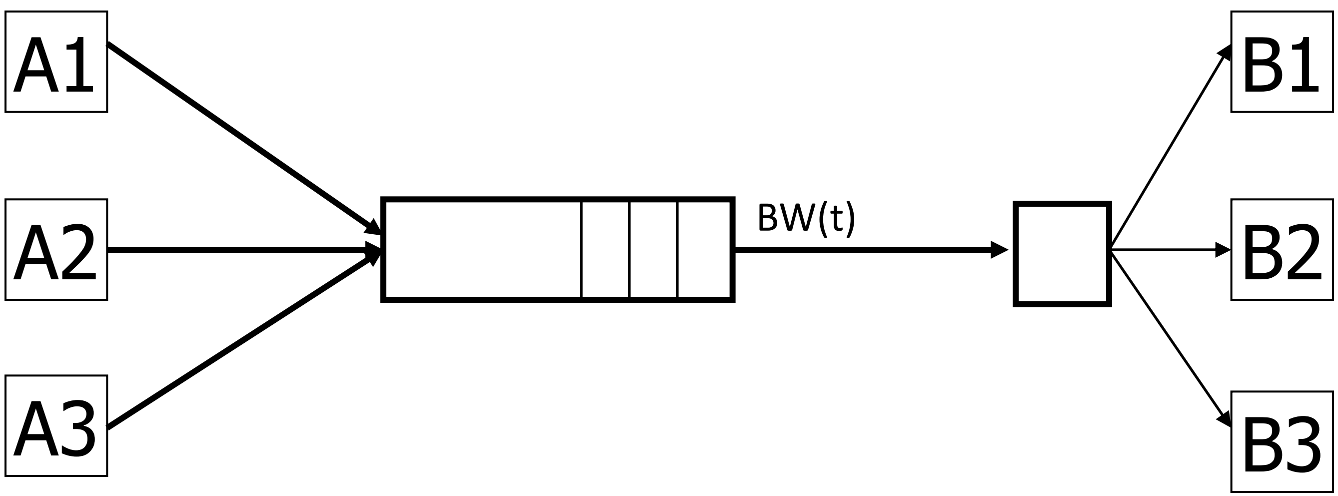
Adjusting to variations in bandwidth



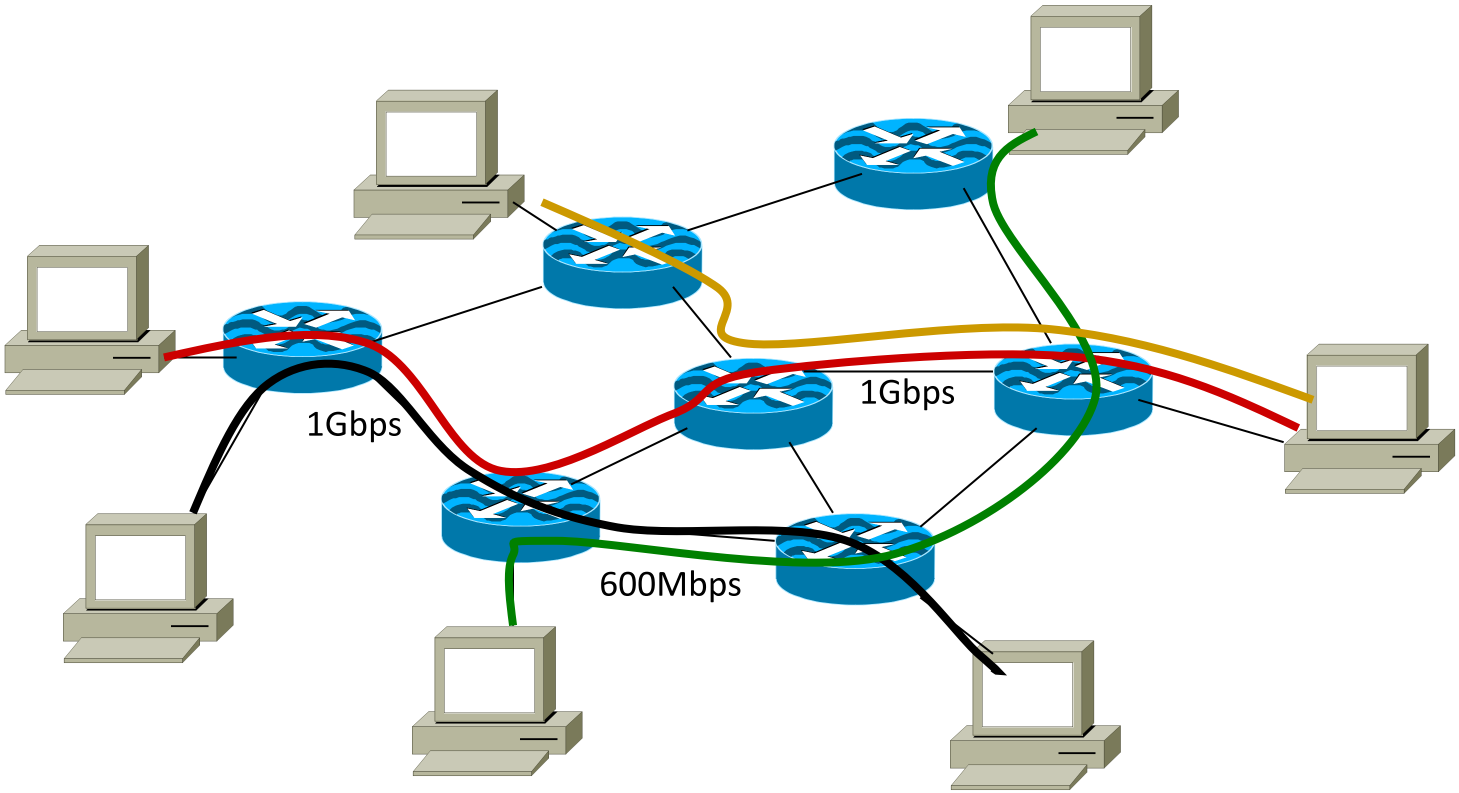
- **Adjust rate to match instantaneous bandwidth**
  - Assuming you have rough idea of bandwidth

Multiple flows and sharing bandwidth

- **Two Issues:**
  - Adjust total sending rate to match bandwidth
  - Allocation of bandwidth between flows



Reality



Congestion control is a resource allocation problem involving many flows, many links, and complicated global dynamics

Possible approaches

(0) Send without care

- Many packet drops

Possible approaches

(0) Send without care

(1) Reservations

- Pre-arrange bandwidth allocations
- Requires negotiation before sending packets
- Low utilization

Possible approaches

(0) Send without care

(1) Reservations

(2) Pricing

- Don’t drop packets for the high-bidders
- Requires payment model

Possible approaches

(0) Send without care

(1) Reservations

(2) Pricing

(3) Dynamic Adjustment

- Hosts infer level of congestion; adjust
- Network reports congestion level to hosts; hosts adjust
- Combinations of the above
- Simple to implement but suboptimal, messy dynamics

Possible approaches

(0) Send without care

(1) Reservations

(2) Pricing

(3) Dynamic Adjustment

- **Generality of dynamic adjustment has proven to be very powerful**
  - ➢ Doesn’t presume business model, traffic characteristics, application requirements
  - ➢ But does assume good citizenship!

TCP’s approach in a nutshell

- **Each TCP connection has a window**
  - Controls number of packets in flight
- **Sending rate ~Window/RTT**
- **Vary window size to control sending rate**

Windows to keep in mind

- **Congestion Window: CWND**
  - ➢Bytes that can be sent without overflowing routers
  - ➢Computed by sender using congestion control algorithm
- **Flow control window: RWND**
  - ➢Bytes that can be sent without overflowing receiver
  - ➢Determined by the receiver and reported to the sender
- **Sender-side window = min {CWND, RWND}**
  - ➢Assume for this lecture that RWND >> CWND

Note

- **This lecture talks about CWND in units of MSS**
    - ➢ MSS (Maximum Segment Size): the amount of payload data in a TCP packet
    - ➢ This is only for the simplicity of presentation
- **Real implementations maintain CWND in bytes**

Two basic questions

- **How does the sender detect congestion?**
- **How does the sender adjust its sending rate?**
  - ➢ To address three issues
    - Finding available bottleneck bandwidth
    - Adjusting to bandwidth variations
    - Sharing bandwidth

Detecting congestion

- **Packet delays**
  - Tricky: noisy signal (delay often varies considerably)
- **Routers tell end hosts when they're congested**
- **Packet loss**
  - Fail-safe signal that TCP already has to detect
  - Complication: non-congestive loss (e.g., checksum errors)

Not all losses are the same

- **Duplicate ACKs: isolated loss**
  - Still getting ACKs
- **Timeout: much more serious**
  - Not enough duplicate acks
  - Must have suffered several losses
- **Will adjust rate differently for each case**

Rate adjustment

- **Basic structure**
  - ➢ Upon receipt of ACK (of new data): increase rate
  - ➢ Upon detection of loss: decrease rate
- **How we increase/decrease the rate depends on the phase of congestion control we're in:**
  - ➢ Discovering available bottleneck bandwidth vs.
  - ➢ Adjusting to bandwidth variations

Bandwidth discovery with "Slow Start"

- **Goal: estimate available bandwidth**
  - Start slow (for safety)
  - Ramp up quickly (for efficiency)
- **Consider**
  - RTT = 100ms, MSS=1000bytes
  - Window size to fill 1Mbps of BW = 12.5 packets
  - Window size to fill 1Gbps = 12,500 packets
  - Either is possible!

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**

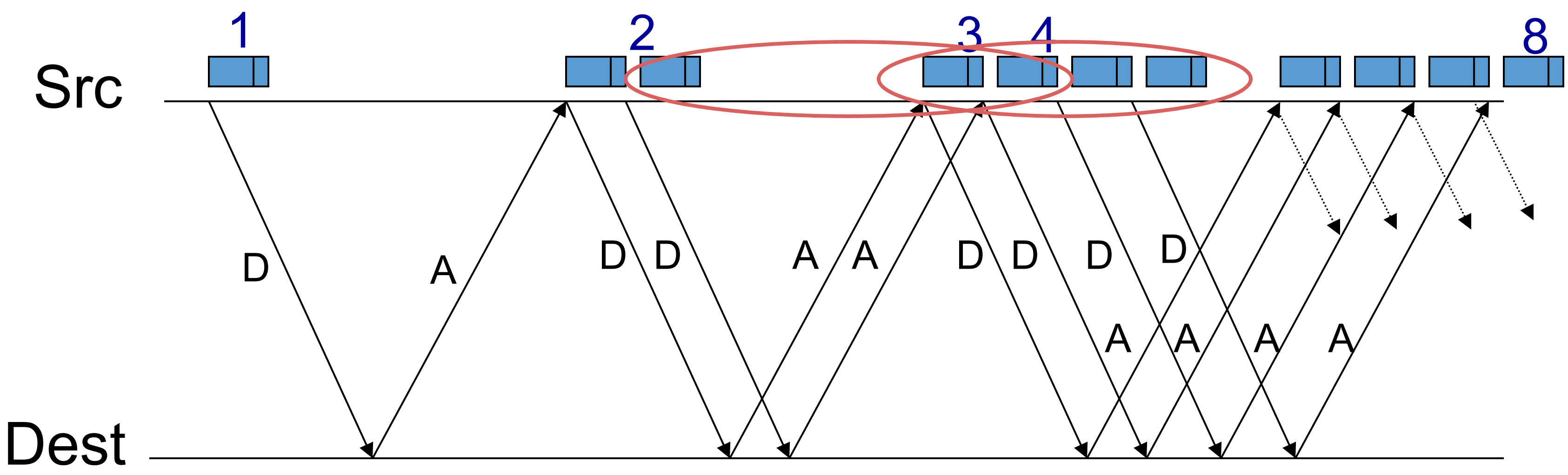
Slow Start in action

- **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**

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

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?

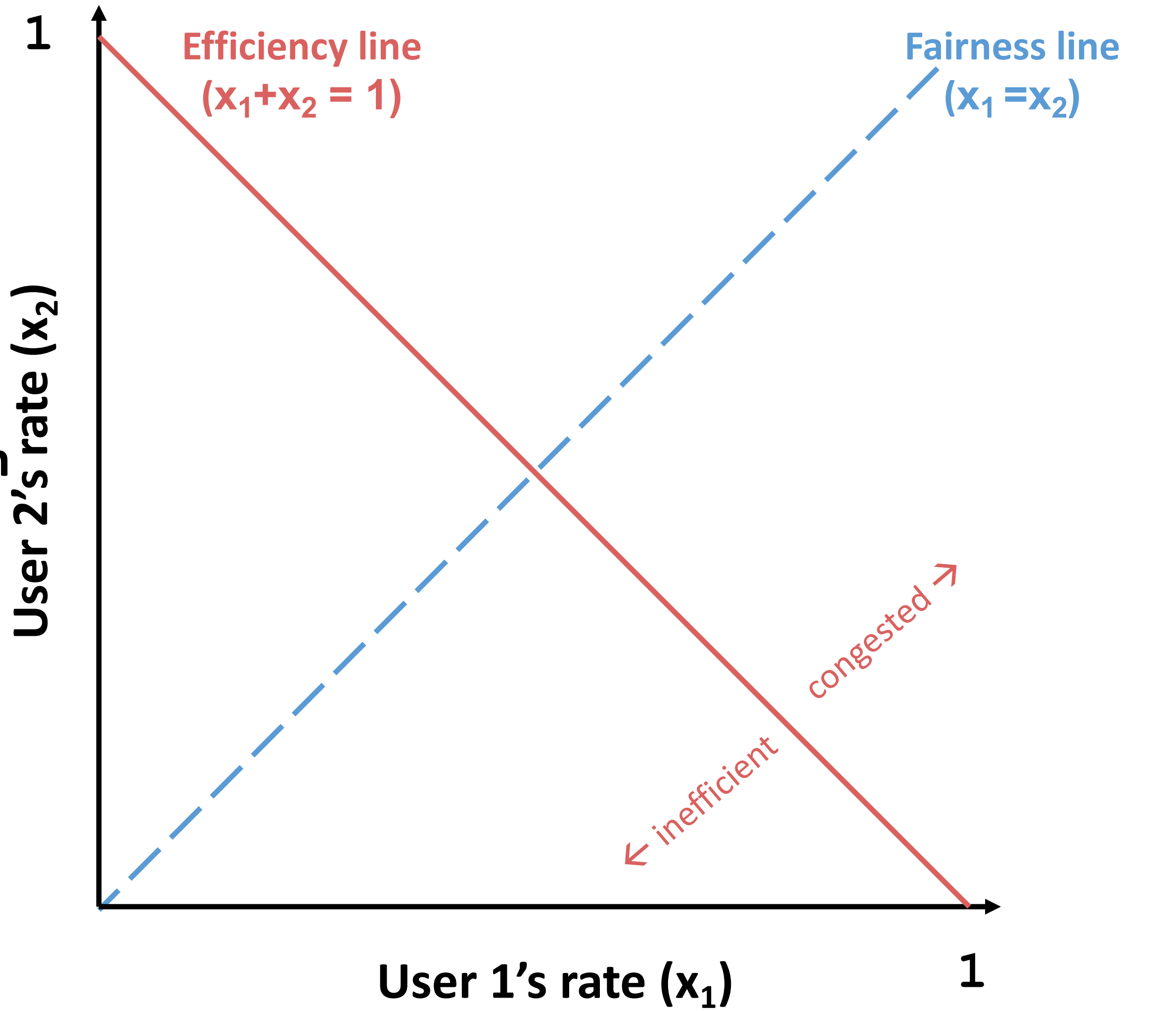
- **Every RTT, we can do**
  - Multiplicative increase or decrease: CWND$\rightarrow$ a*CWND
  - Additive increase or decrease: CWND$\rightarrow$ CWND + b
- **Four alternatives:**
  - AIAD: gentle increase, gentle decrease
  - AIMD: gentle increase, drastic decrease
  - MIAD: drastic increase, gentle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

Simple model of congestion control

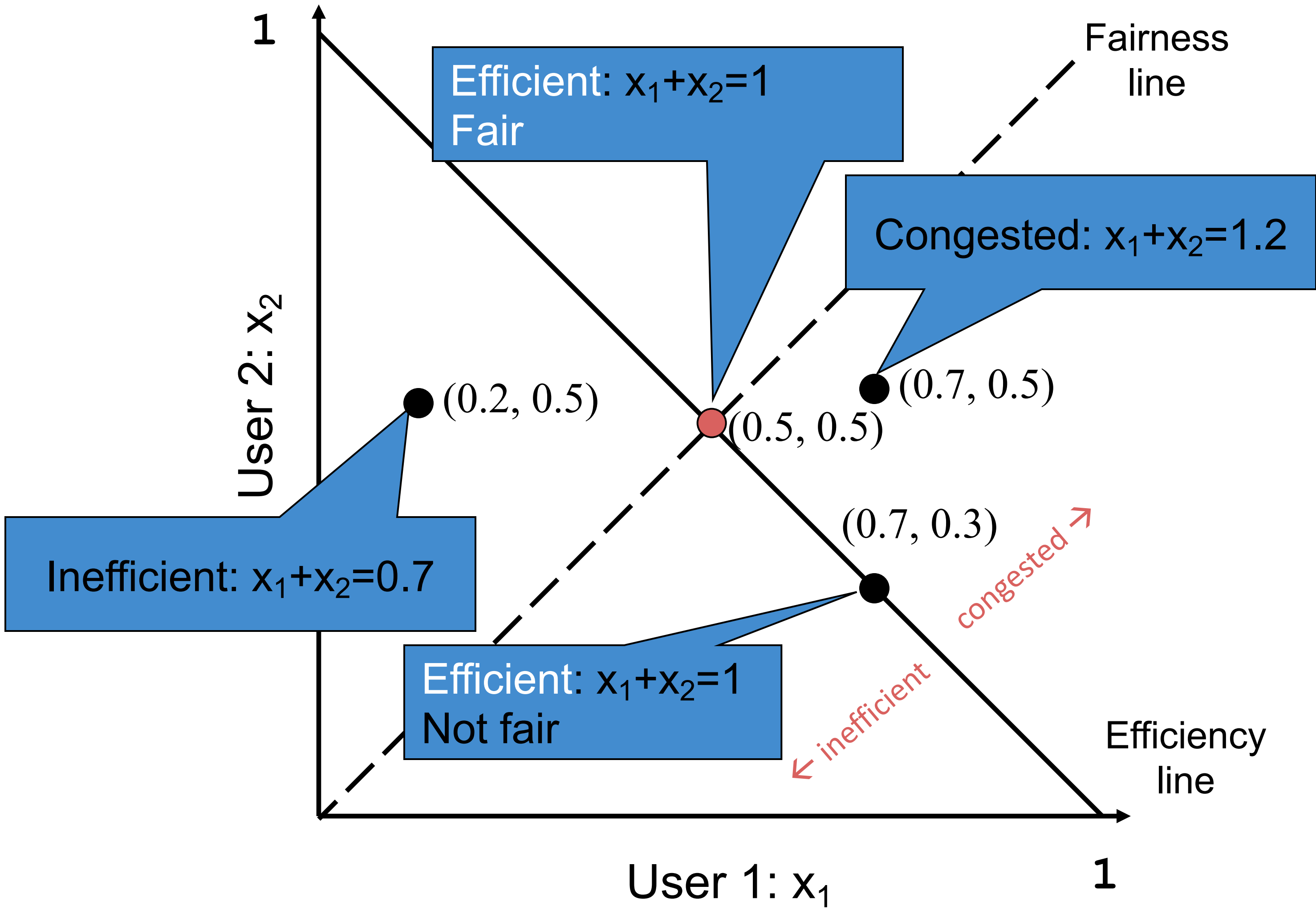
- **Two users**
  - rates x_1 and x_2

- **Congestion when $x_1+x_2 > 1$**
- **Unused capacity when $x_1+x_2 < 1$**

- **Fair when $x_1 = x_2$**



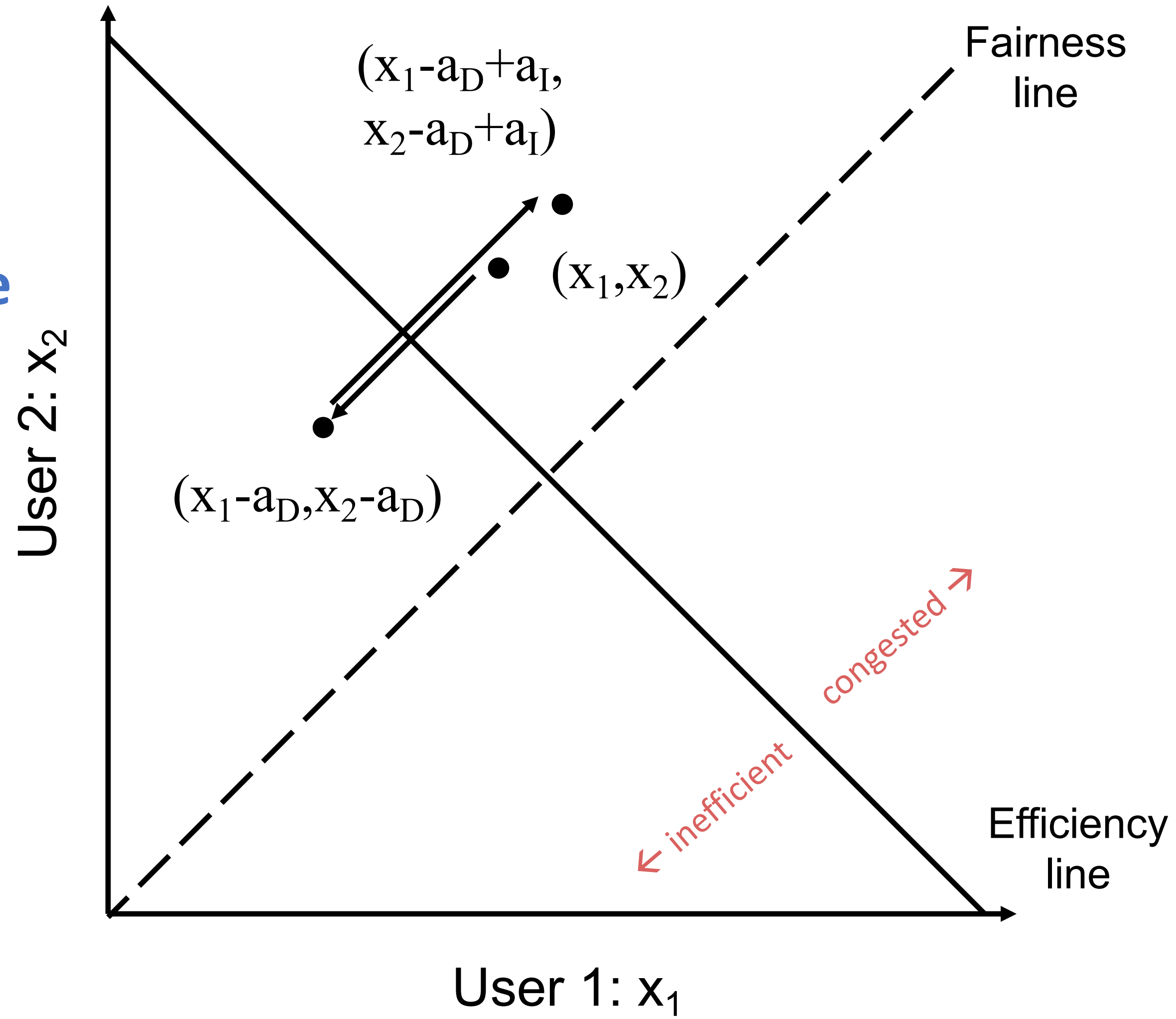
Example



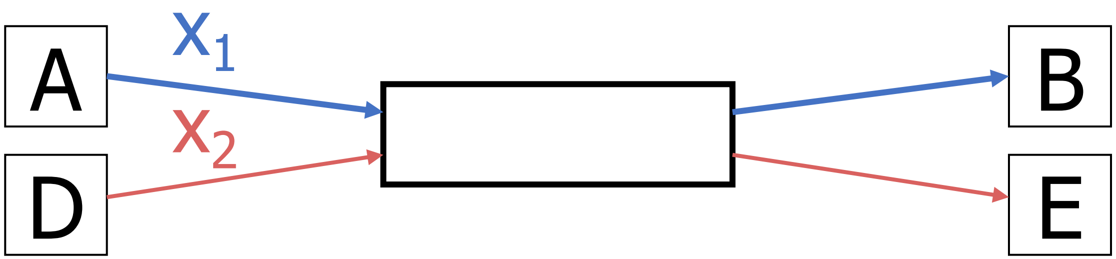
1
Fairness
line
Efficient: $x_1+x_2=1$
Fair
Congested: $x_1+x_2=1.2$
User 2: x_2
(0.2, 0.5)
(0.5, 0.5)
(0.7, 0.5)
(0.7, 0.3)
Inefficient: $x_1+x_2=0.7$
congested →
← inefficient
Efficient: $x_1+x_2=1$
Not fair
Efficiency
line
User 1: x_1
1

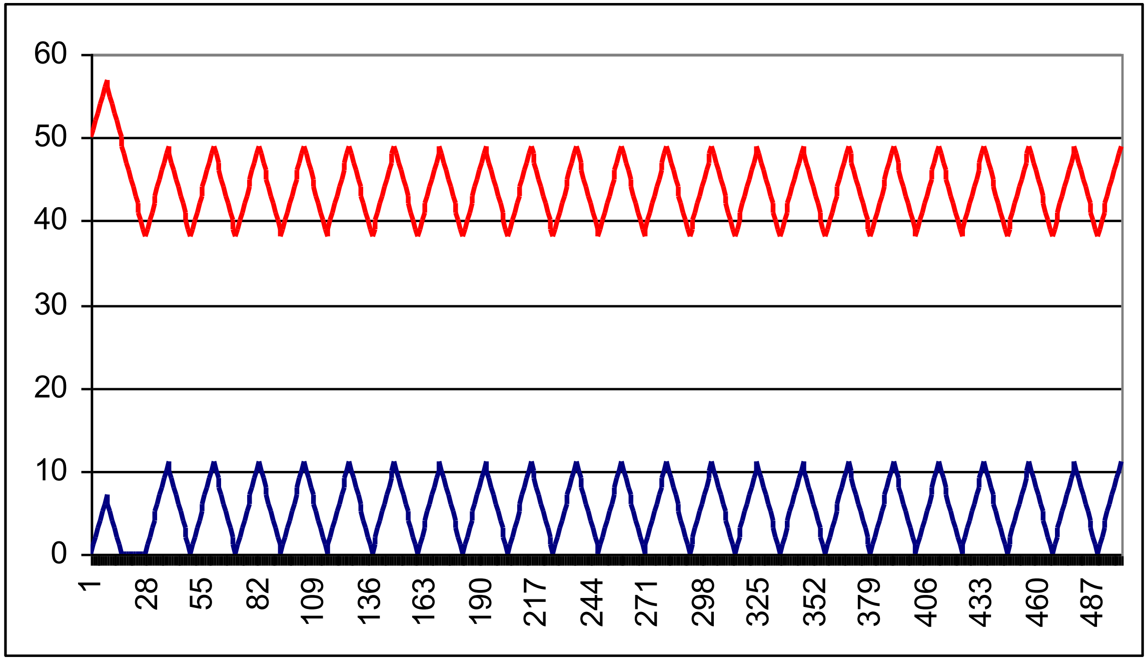
AIAD

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



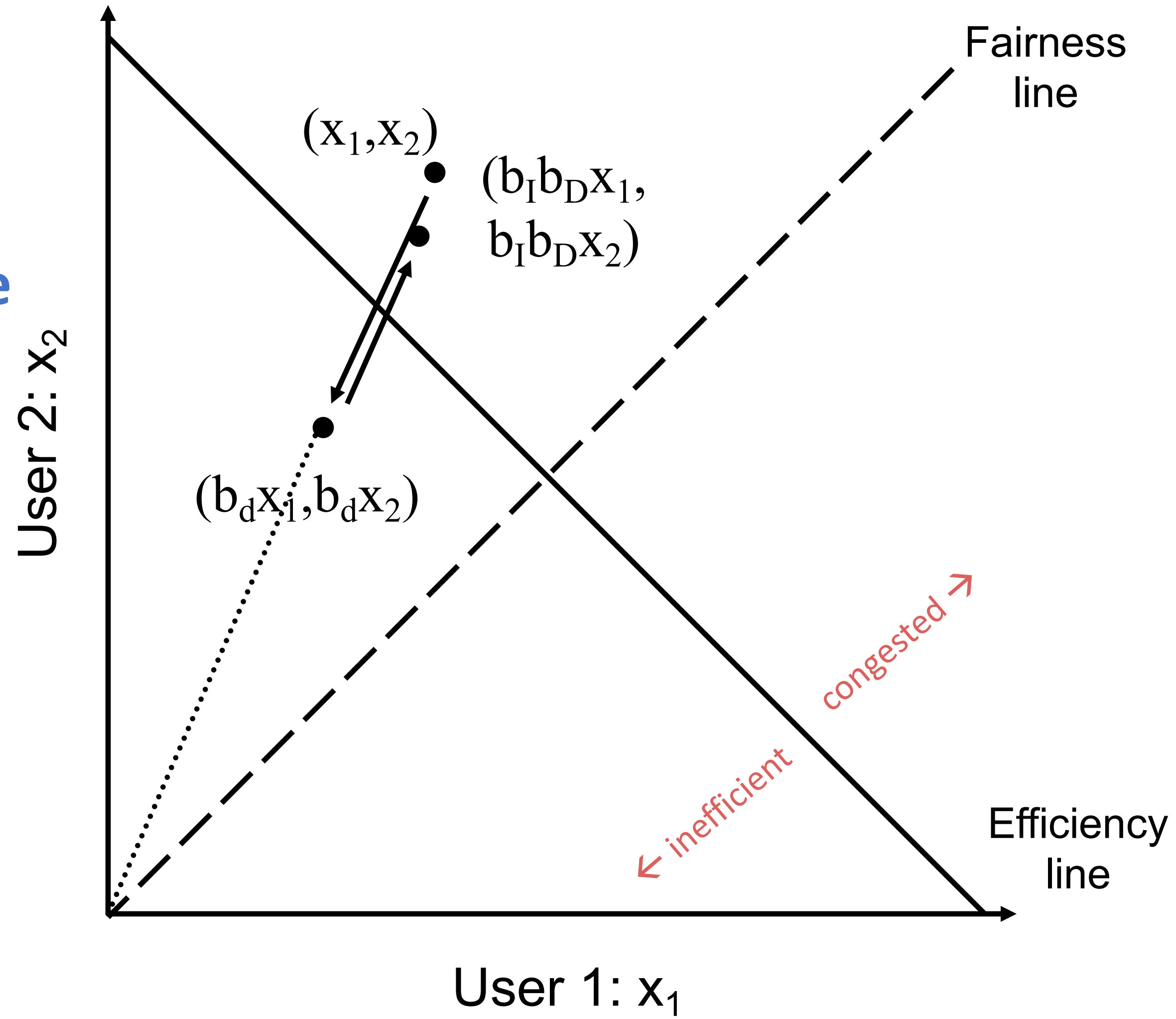
AIAD Sharing Dynamics





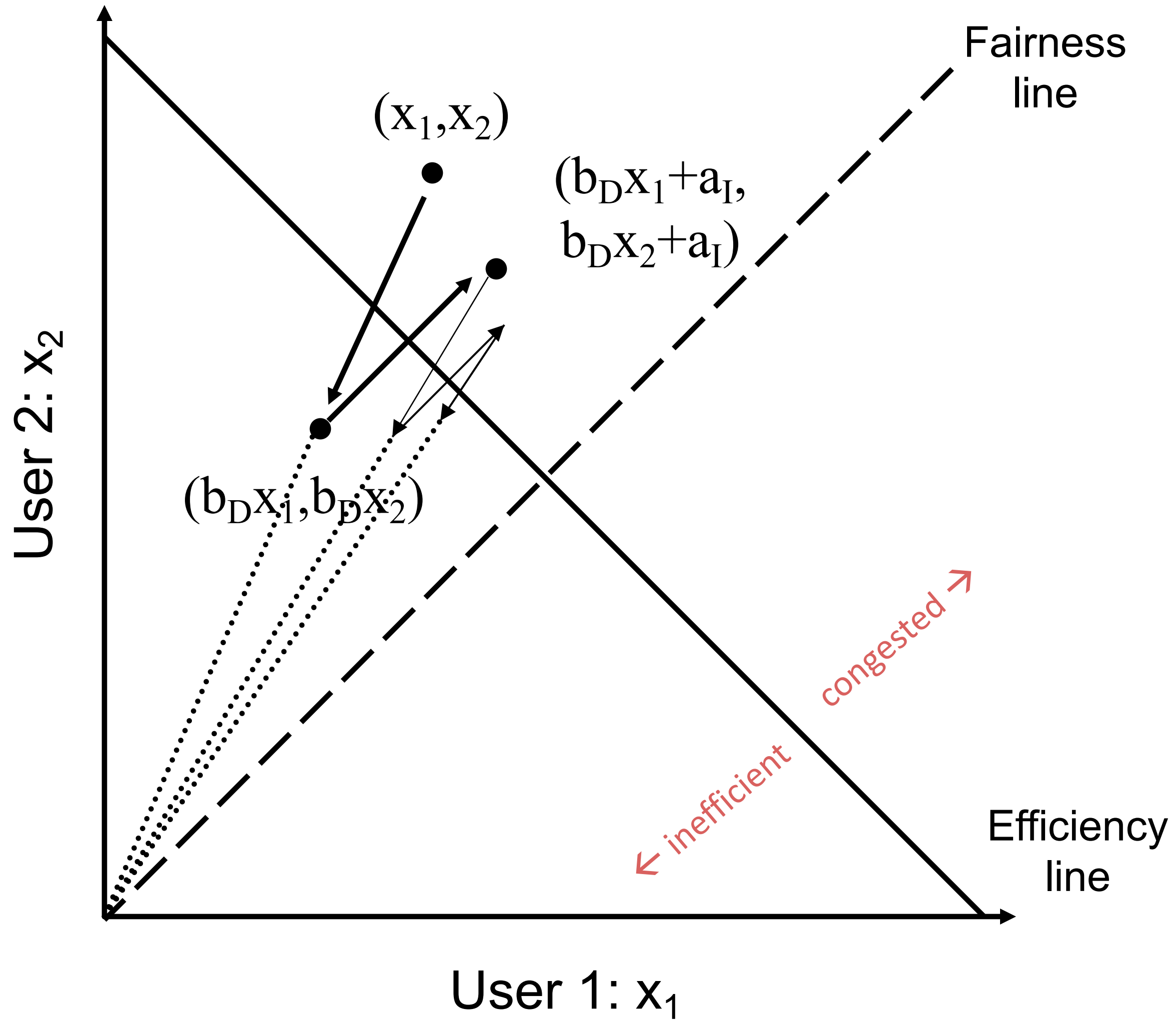
MIMD

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

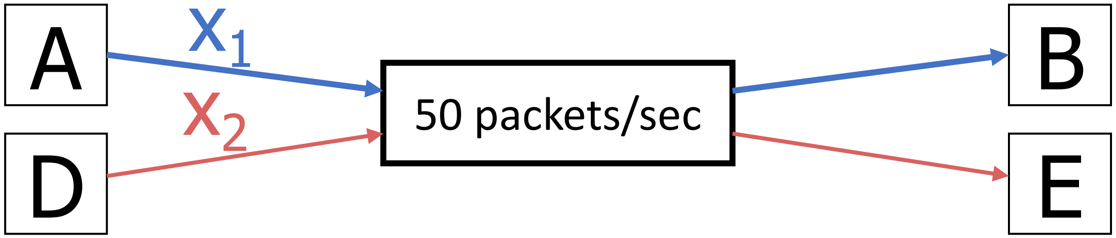


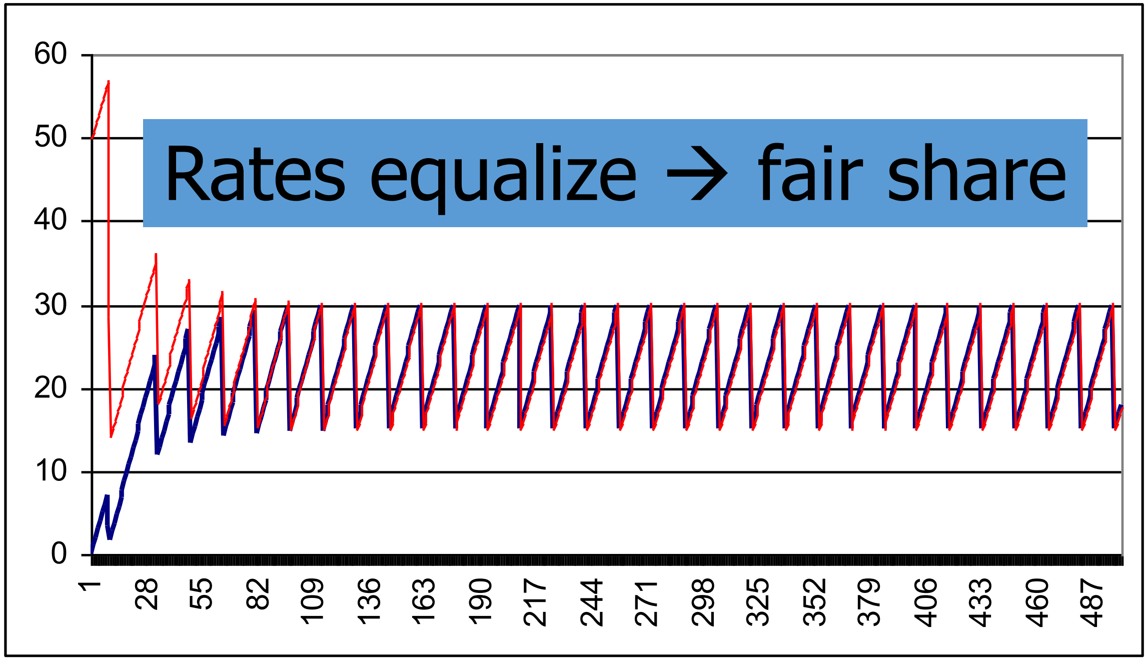
AIMD

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



AIMD Sharing Dynamics



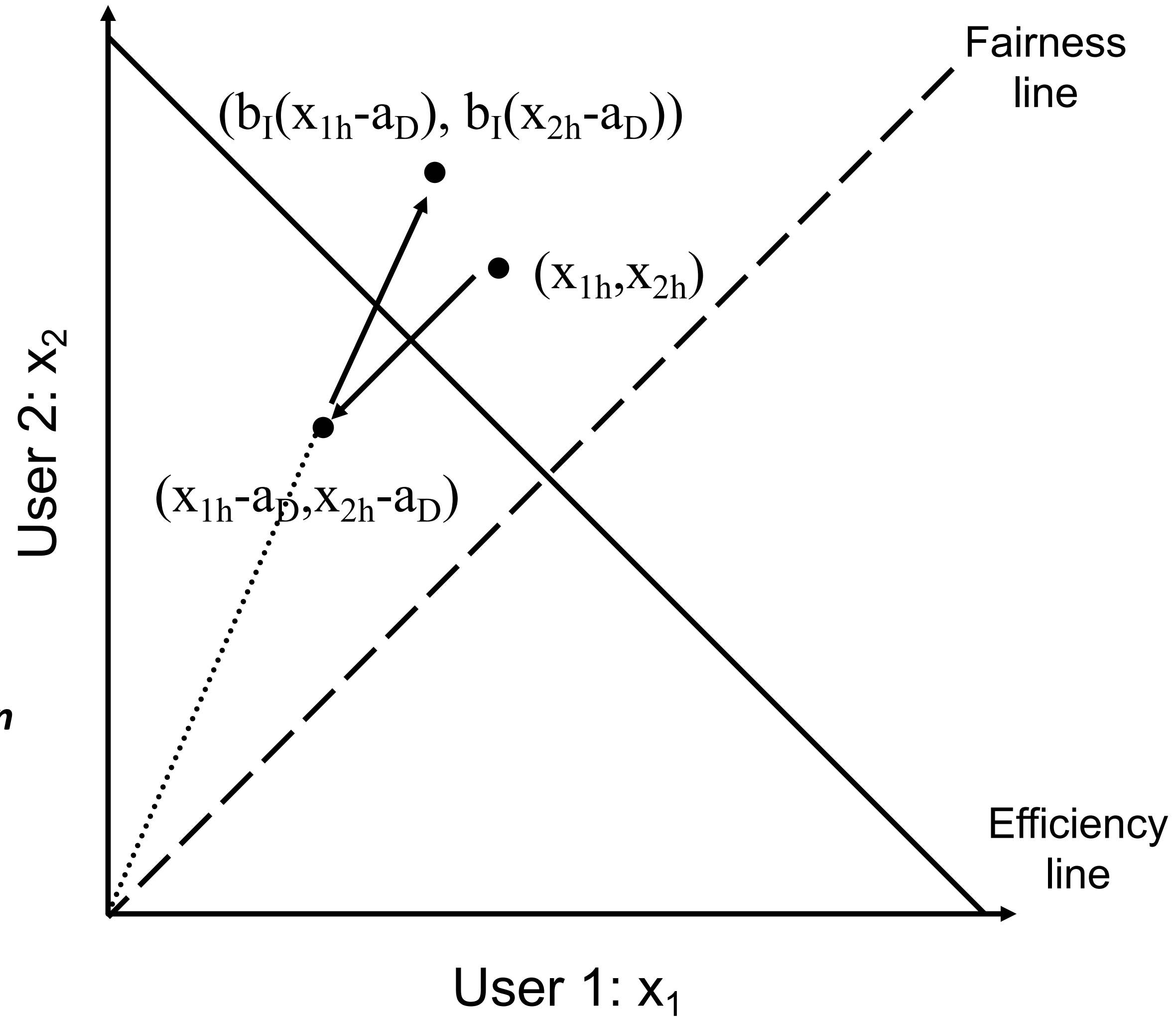


MIAD

- **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



Group Discussion

- **Topic: TCP congestion control**
  - We mentioned four possible approaches for congestion control: (1) send without care, (2) reservations, (3) pricing, and (4) dynamic adjustments. What are the pros and cons of each approach?

- **Discuss in groups, and each group chooses a leader to summarize the discussion**
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak
  - Turn on your video if you can

Summary

- **Flow control ensures that the sender does not overflow the receiver**
- **Congestion control ensures that the sender does not overflow the network**
  - ➢Discover bandwidth
  - ➢Adjust to conditions
  - ➢Share bandwidth with others

Thanks!
Q&A