Congestion Control

Soheil Abbasloo
Department of Computer Science
University of Toronto

Fall 2022

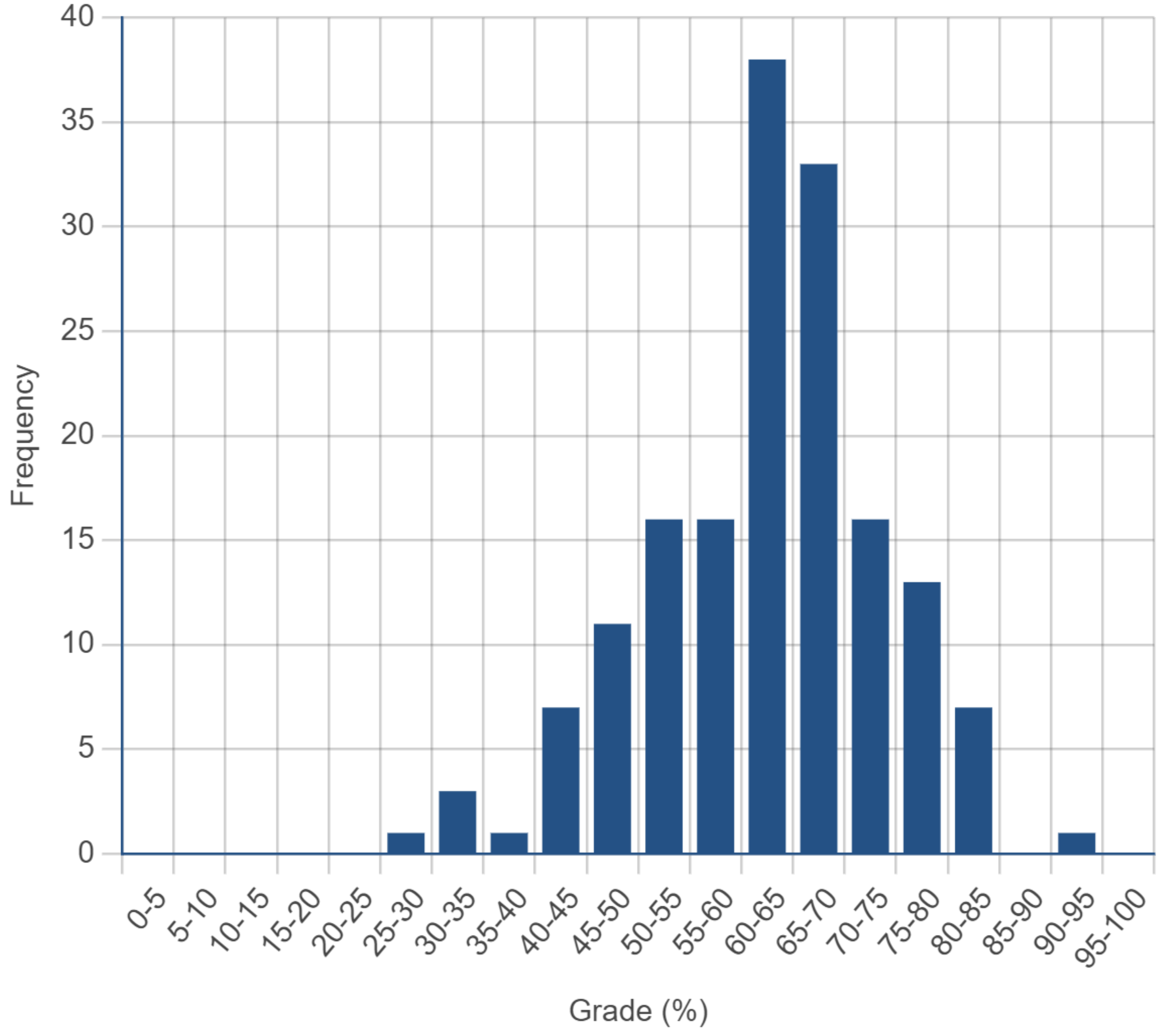
Outline

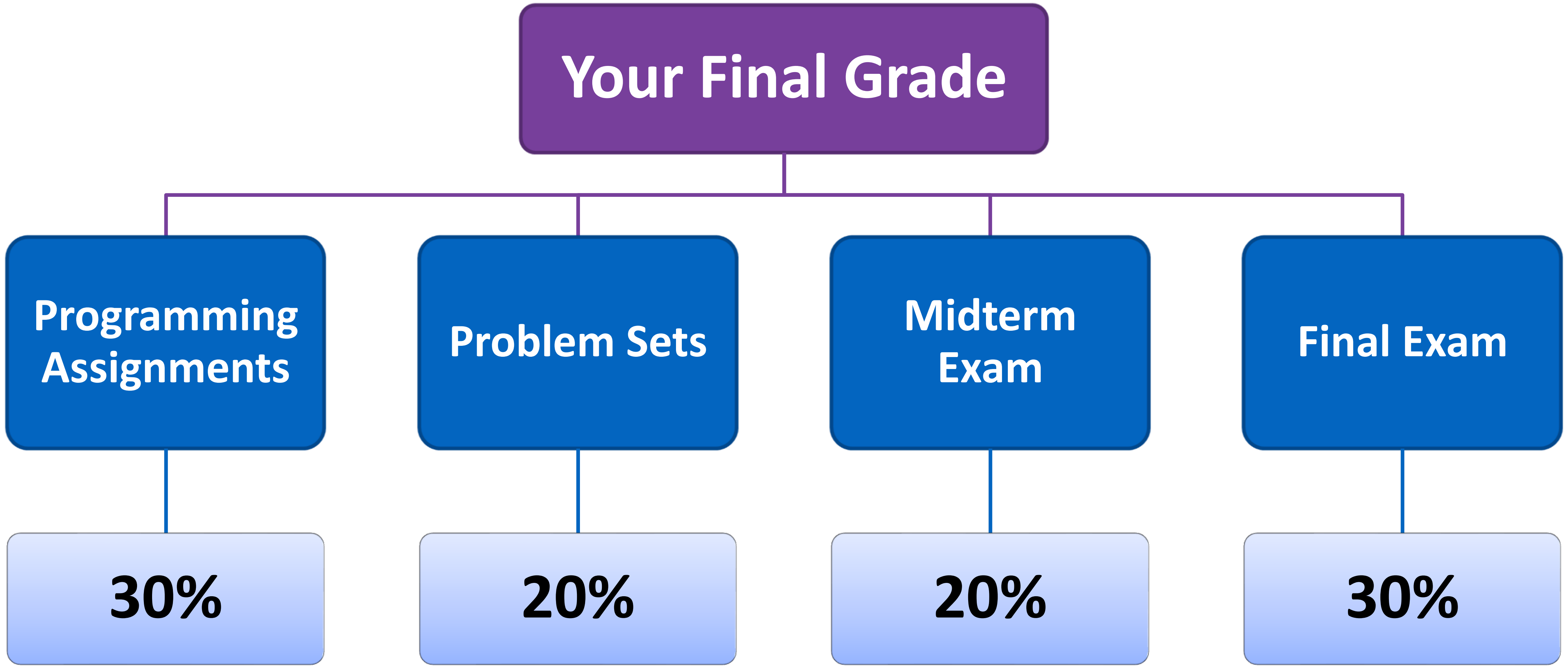
- Let’s Talk abt Midterm!
- Overview of TCP
- Congestion Control
  - What?
  - How?

Midterm Stats

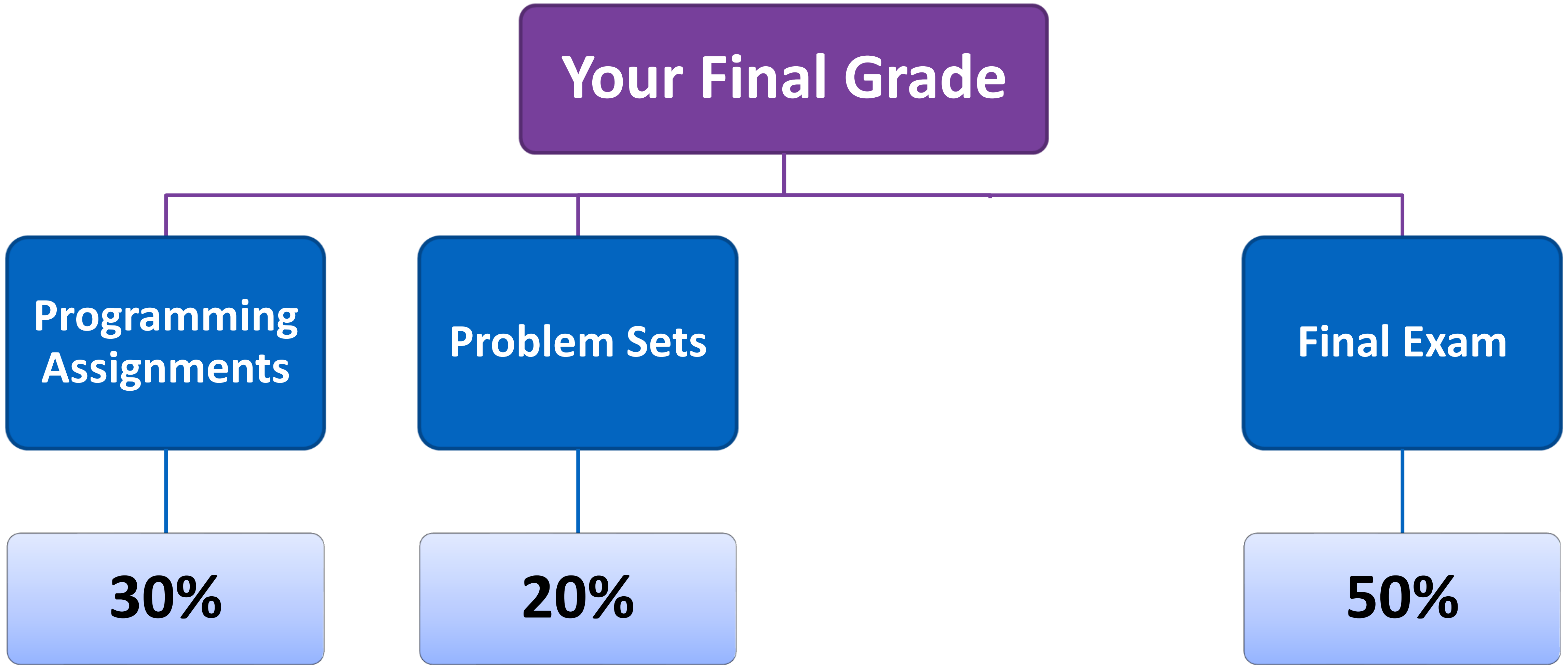
	Column	Average
▶	1- Multiple-Choice	9.76 / 18 (54.23%)
▶	2- D/Thr Measurements	9.52 / 10 (95.15%)
▶	3- Design Decisions	2.66 / 10 (26.63%)
▶	4- Discovering Path Properties	6.57 / 10 (65.71%)
▶	5- BGP	6.66 / 10 (66.56%)
▶	6- Routing	6.51 / 10 (65.09%)
▶	7- Feedback	1.96 / 2 (98.16%)
▶	Bonus Marks	1.96 / 2 (97.85%)

- Average
  - 45.6 / 72 (63.33%)
- Median
  - 46 / 72 (63.89%)
- Standard dev.
  - 8.16 (11.33%)

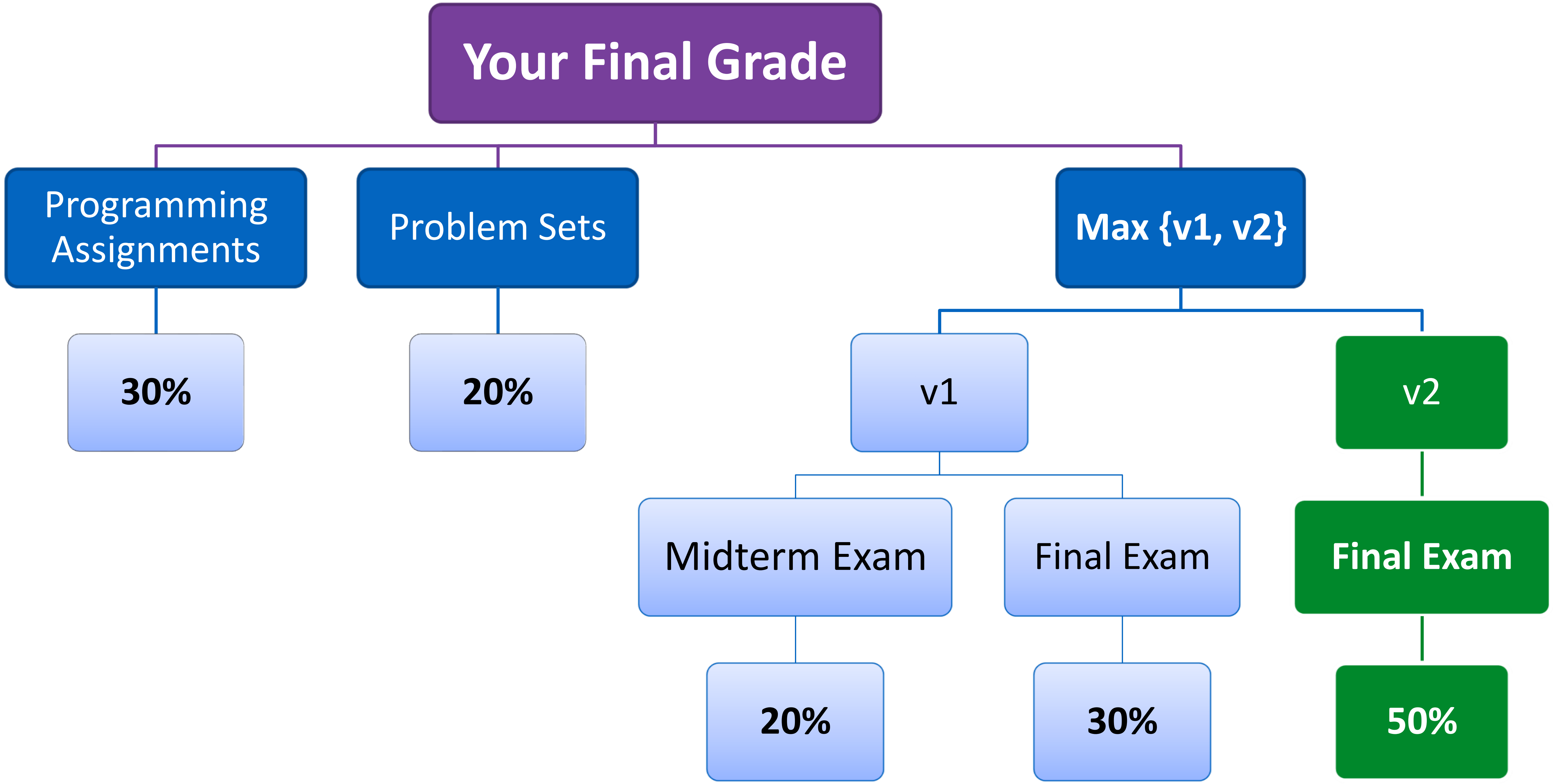




- Midterm:
  - About 10% of students were absent due to diff. issues (sickness, etc.)



- Midterm:
  - About 10% of students were absent due to diff. issues (sickness, etc.)
- **Might not be fair!**


Your Final Grade
Programming Assignments
Problem Sets
Max {v1, v2}
30%
20%
v1
v2
Midterm Exam
Final Exam
Final Exam
20%
30%
50%

Your Feedback!

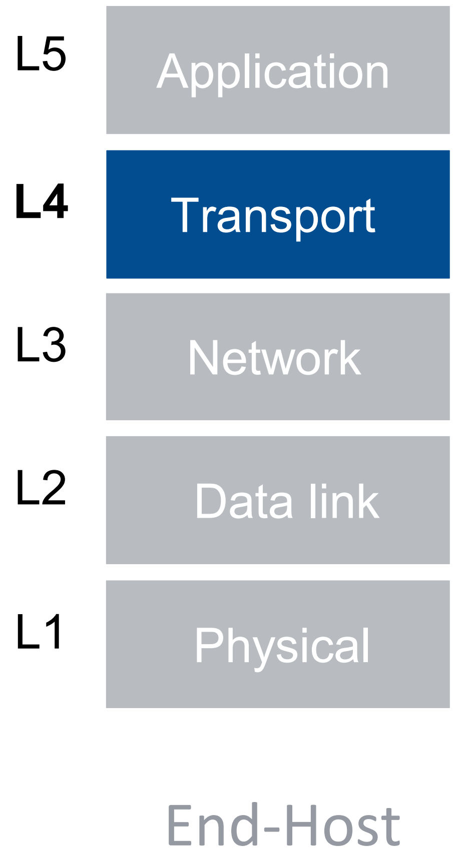
- Top ranked suggestions (based on #students):
  - Sample questions for final
  - A break after 1^{st} hour!

- Some other ones:
  - pptx plus pdf
  - Making slides easier to review later
  - Starting at XX:10
  - **Better Seats!**

Last time …

- TCP provides:
  - A stream-of-bytes service
  - A connection-oriented service

- To bring reliability, TCP:
  - Uses Sliding window algorithm (Flow Control)
  - Sets RTO values proportional to network RTT
  - Detects loss when it sees a timeout

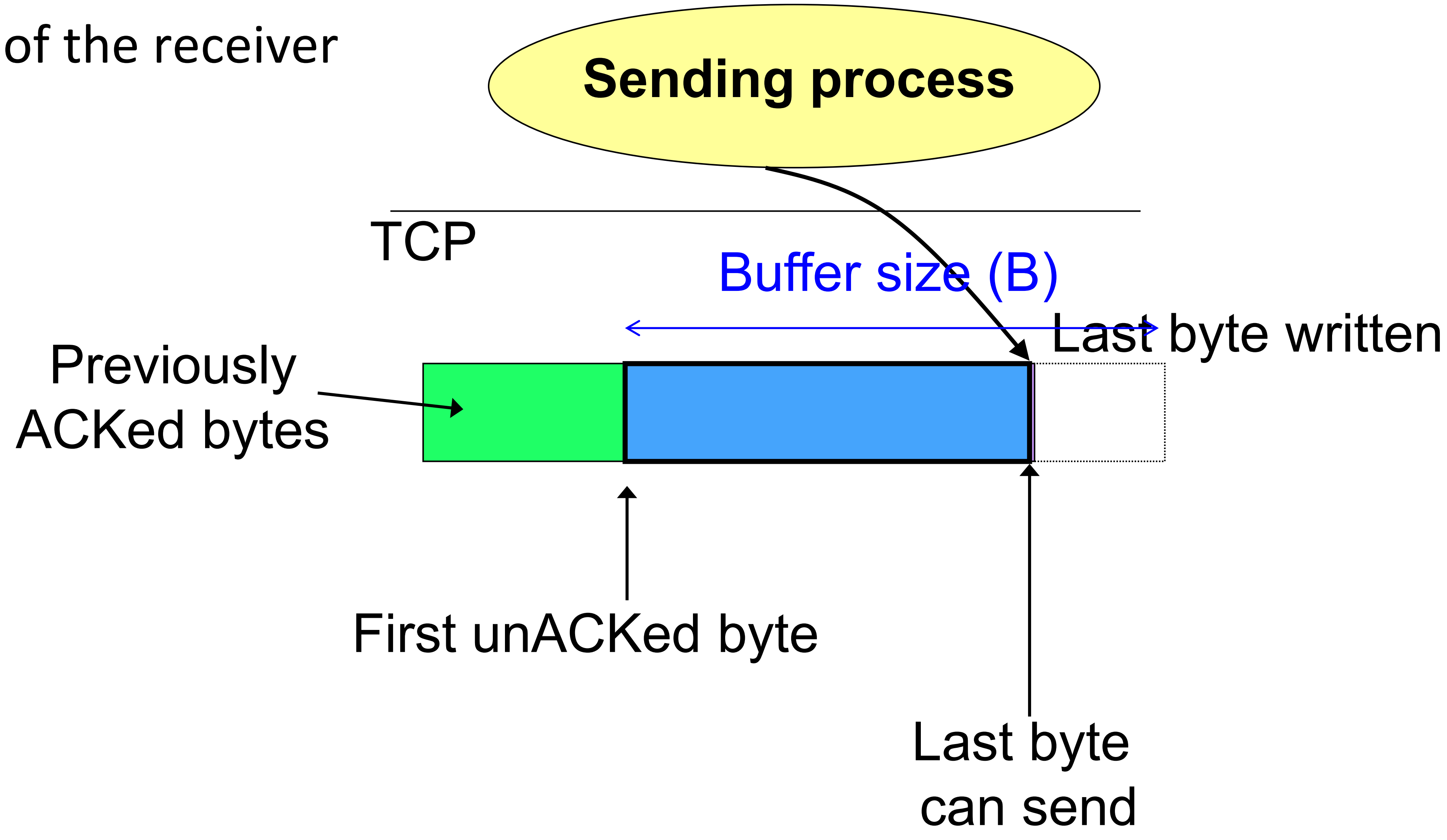


Last time: Sliding Window

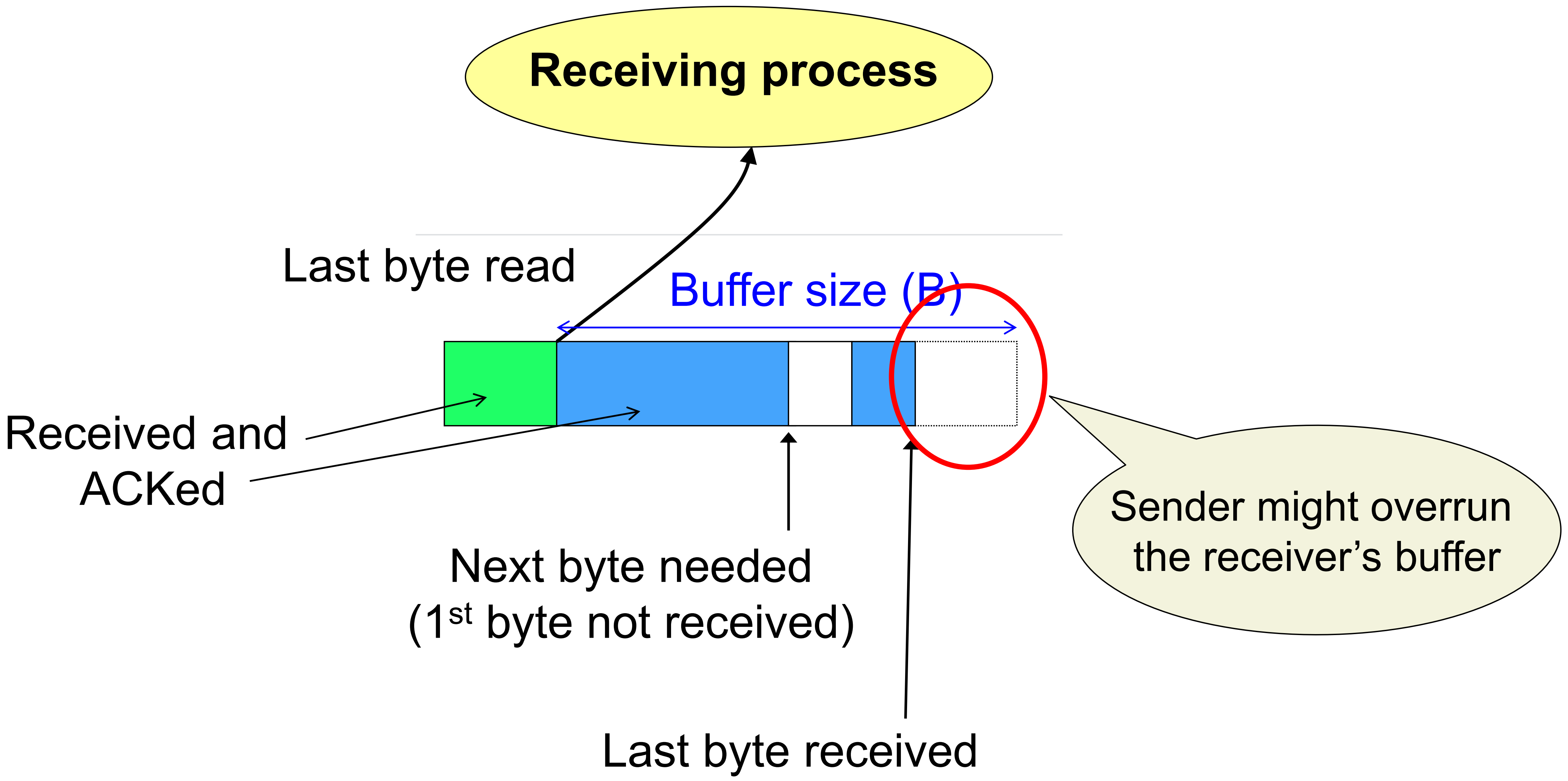
- Allow a larger amount of data “in flight”
  - Allow sender to get ahead of the receiver

Last time: Sliding Window at Sender

- Allow a larger amount of data “in flight”
  - Allow sender to get ahead of the receiver



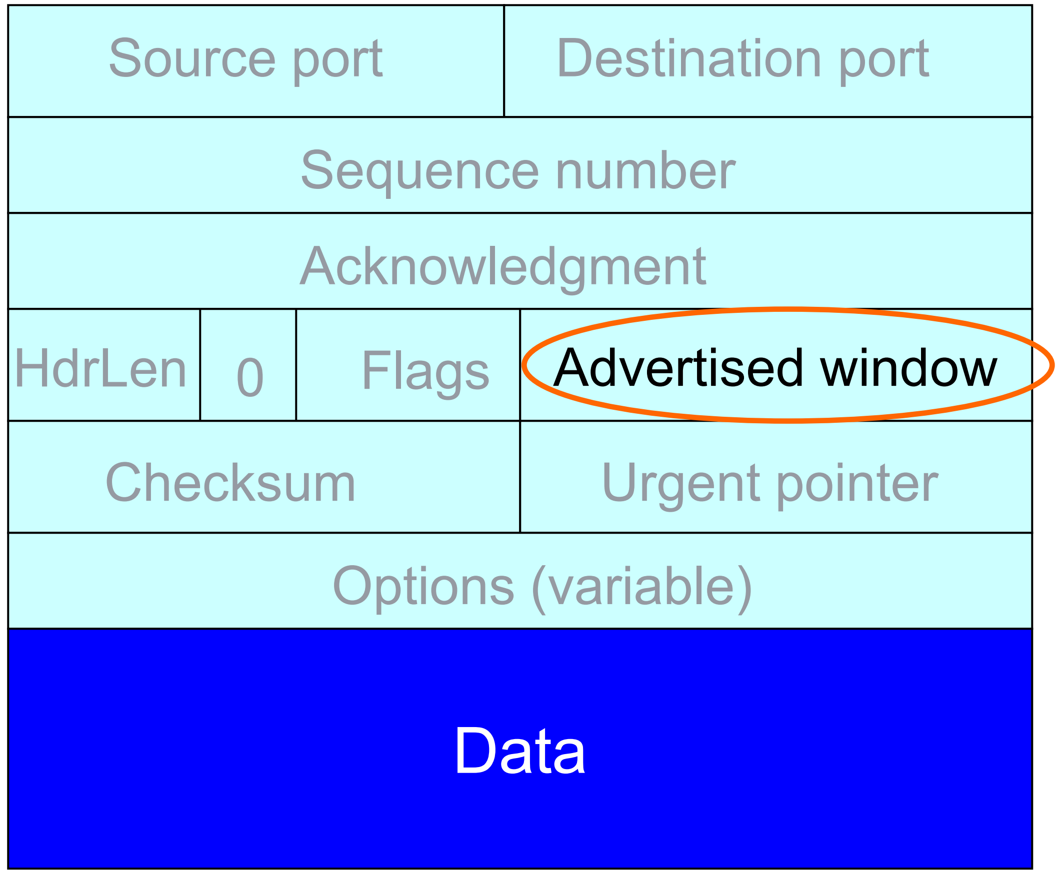
Last time: Sliding Window at Receiver



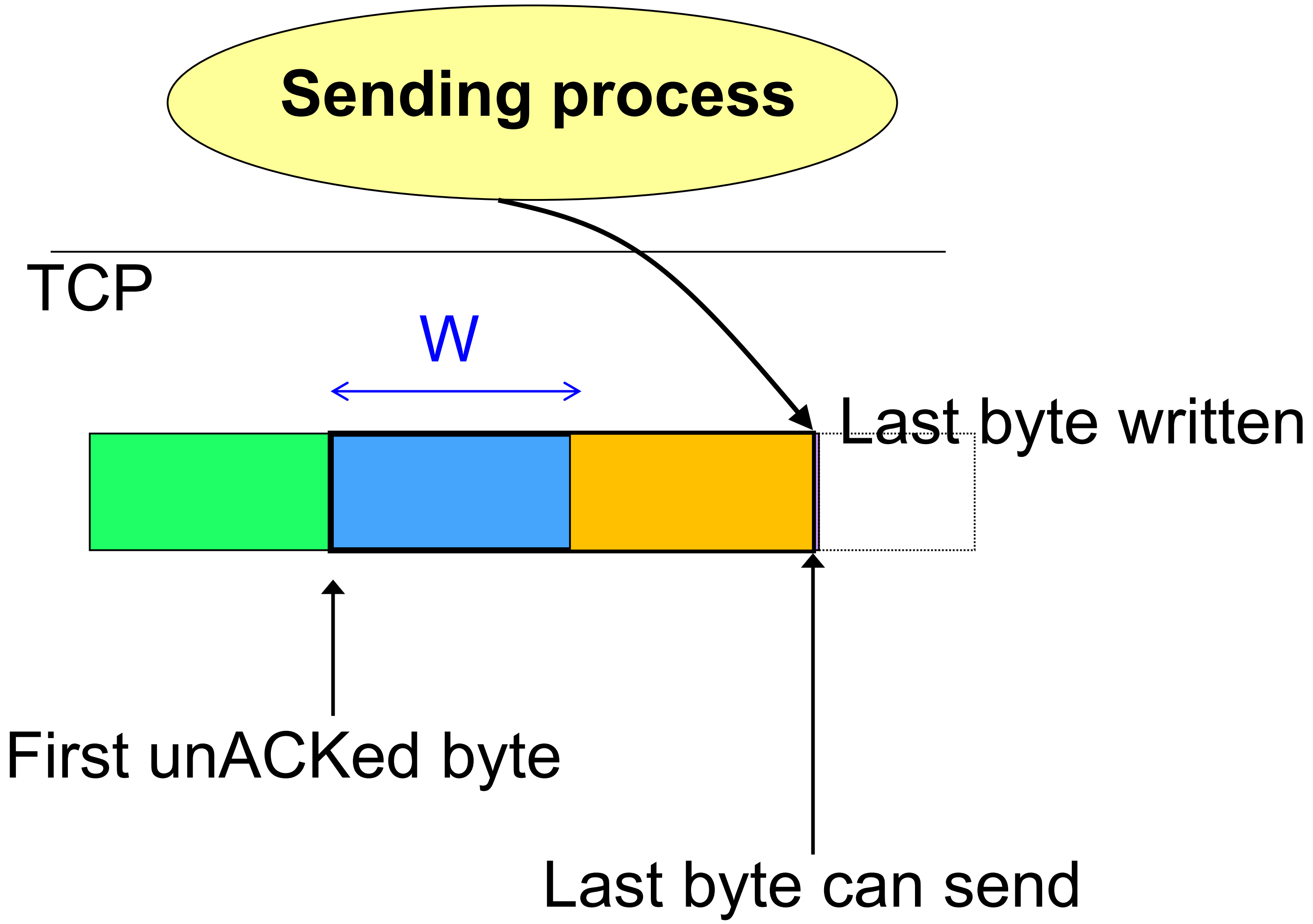
Last time: Advertised Window (Flow Control)

Receiver uses an "Advertised Window" (W) to prevent sender from overflowing its window

- Receiver indicates value of W in ACKs
- Sender limits number of bytes it can have in flight <= W



Sliding Window at Sender



Last time: Advertised Window Limits Sending Rate

- Sender can send no faster than $\frac{W}{RTT}$
- Receiver only advertises more space when it has consumed old arriving data
- In original TCP design, that was the **sole** protocol mechanism controlling sender's rate
- But it's just a very small part of the picture!
  - (Today!)

Congestion Control

What is Congestion?

In mid-1980s, Similar thing happened in Internet!

Network was fully utilized but no ***useful work*** was being done.

About **1000x** throughput reduction

Flow Control was **not** enough!

Statistical Multiplexing

Leads to Congestion

- Internet is based on packet switching
  - No BW reservation
  - Using buffers to absorb transient load
- If two packets arrive at a switch at the same time
  - Switch will transmit one and buffer/drop the other
- Internet traffic is bursty
- If many packets arrive close in time
  - the switch cannot keep up and it gets congested
  - causes packet **delays** and **drops**

Congestion Collapse

- **Definition**: Increase in network load results in a decrease of useful work done
- Many possible causes
  - Spurious retransmissions of packets still in flight
    - E.g., congestion collapses in 1980s
  - Undelivered packets
    - Packets consume resources and are dropped elsewhere in network

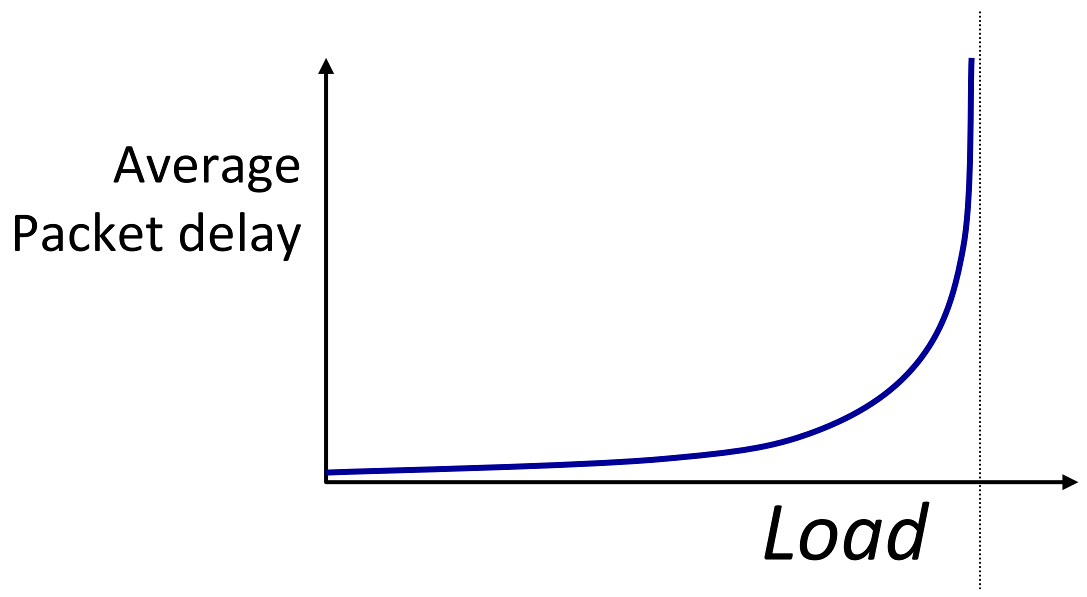
What Users Want?

- High throughput
  - Throughput: measured performance of a system
  - E.g., number of bits/second of data that get through
- Low delay
  - Delay: time required to deliver a packet or message
  - E.g., number of msec to deliver a packet

- These two metrics are sometimes at odds

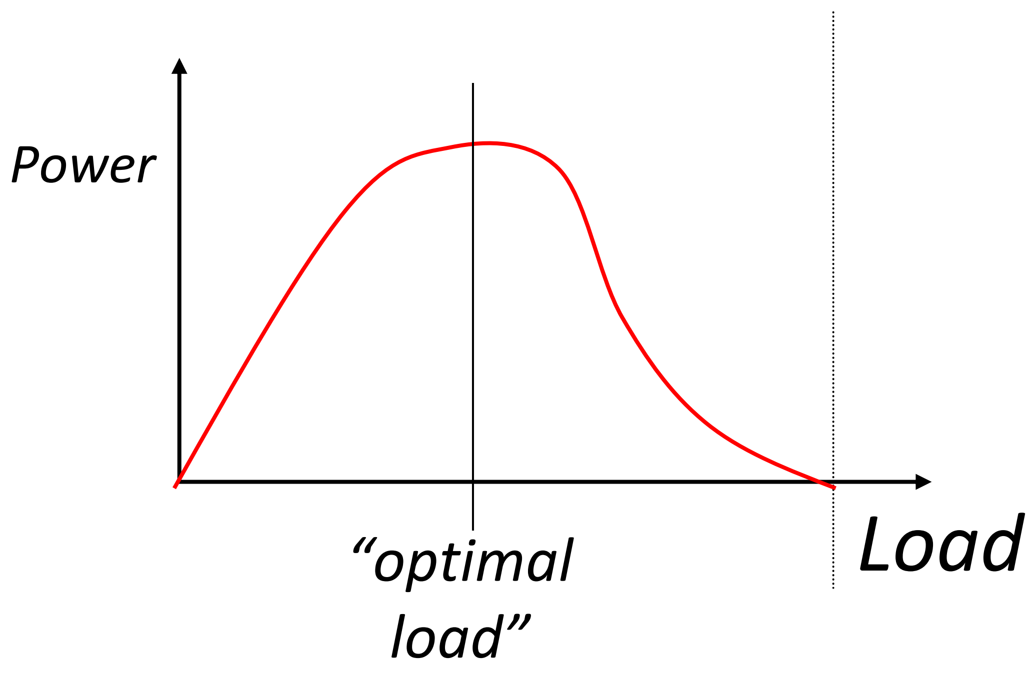
Load, Delay, and Power

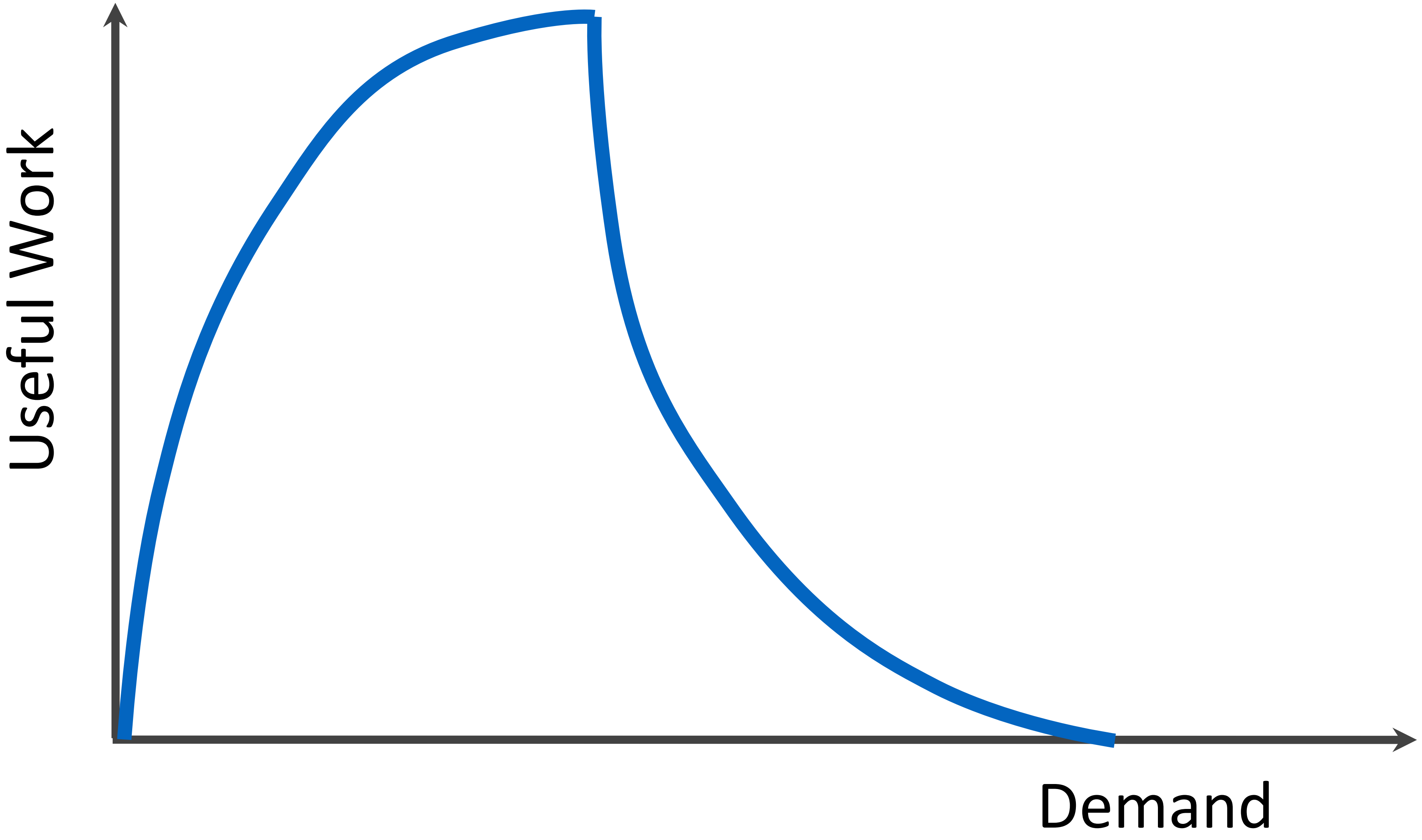
Typical behavior of queuing systems with random arrivals:

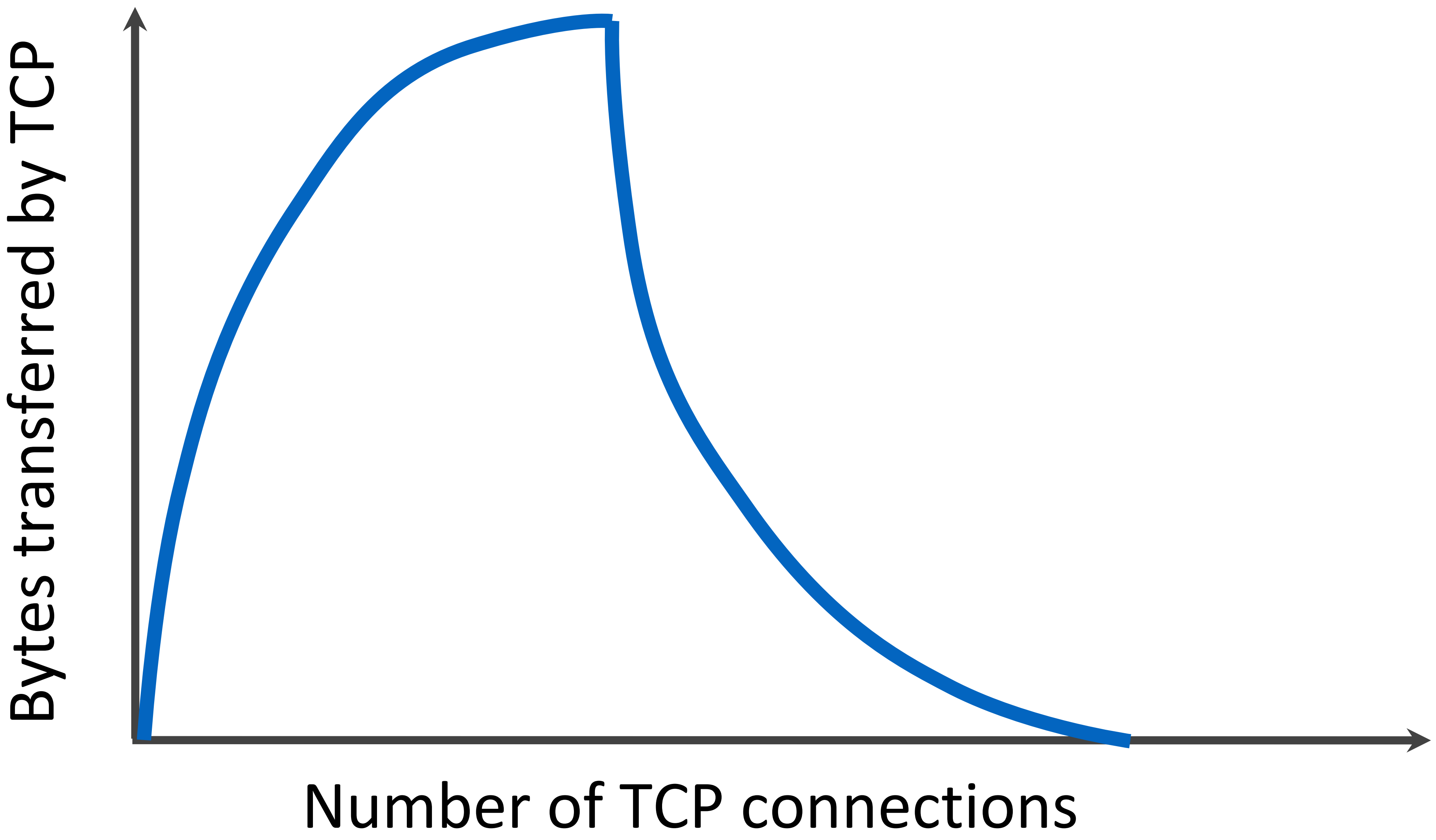


A simple old metric of how well the network is performing:

$$Power = \frac{Load}{Delay}$$

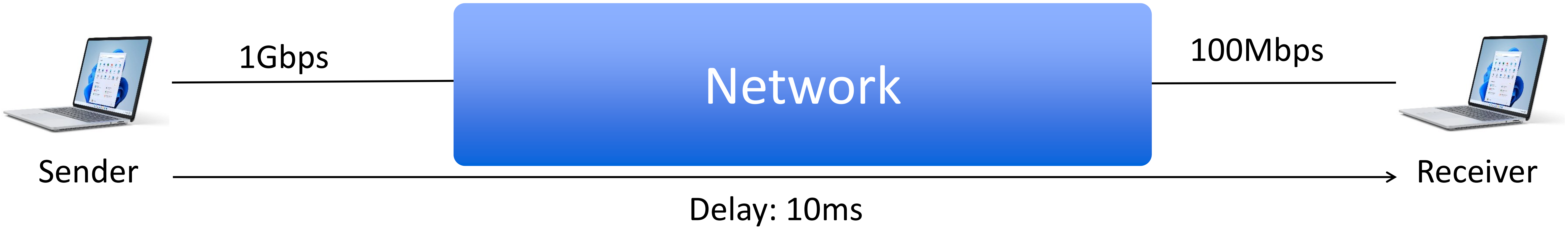



Useful Work
Demand

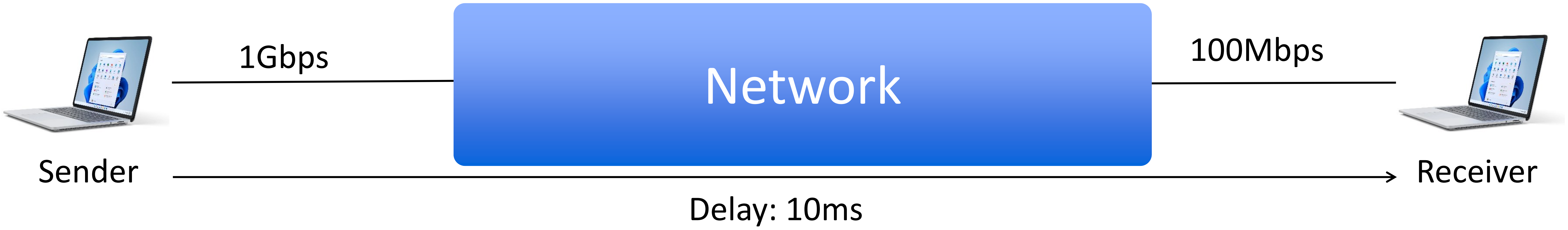


Bytes transferred by TCP
Number of TCP connections

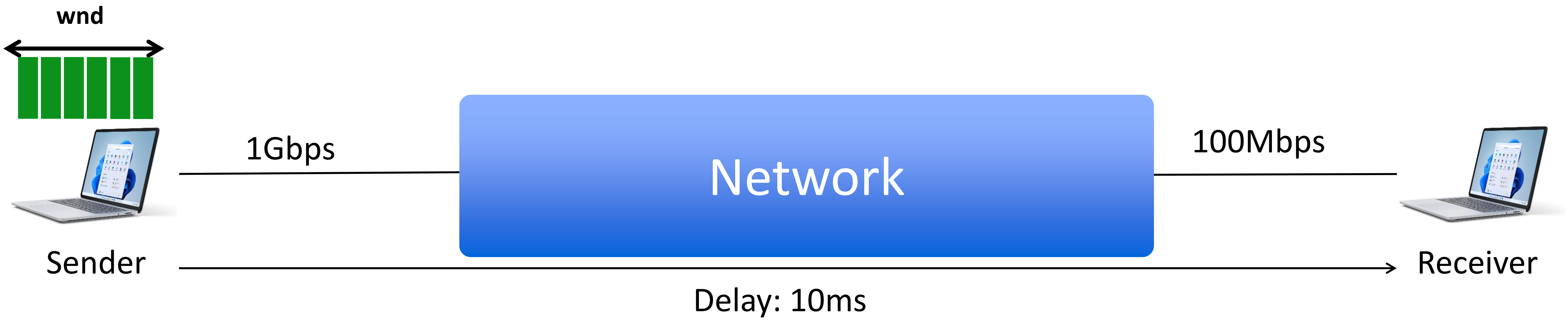
Experiment



Let’s forget abt *reliability*, what is the best sending rate?

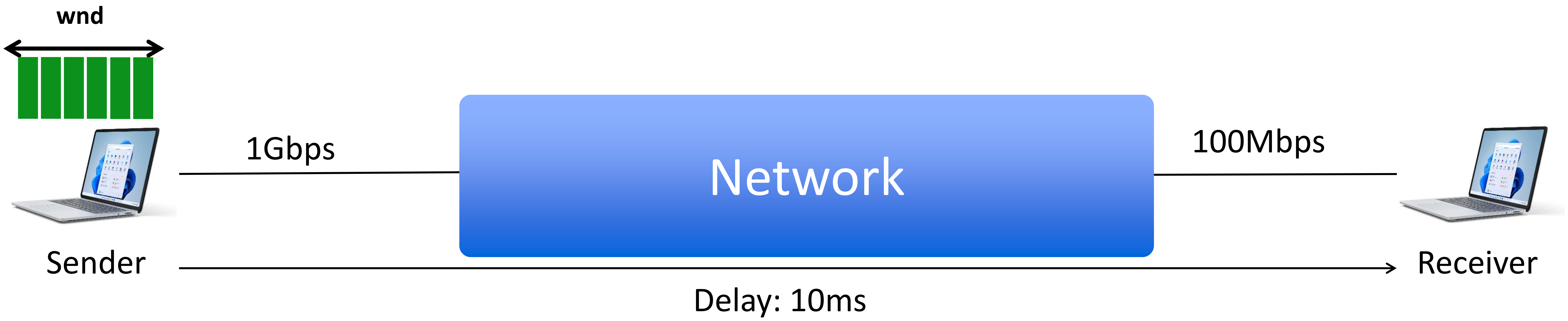


How abt when we want reliability? What is the best sending rate?



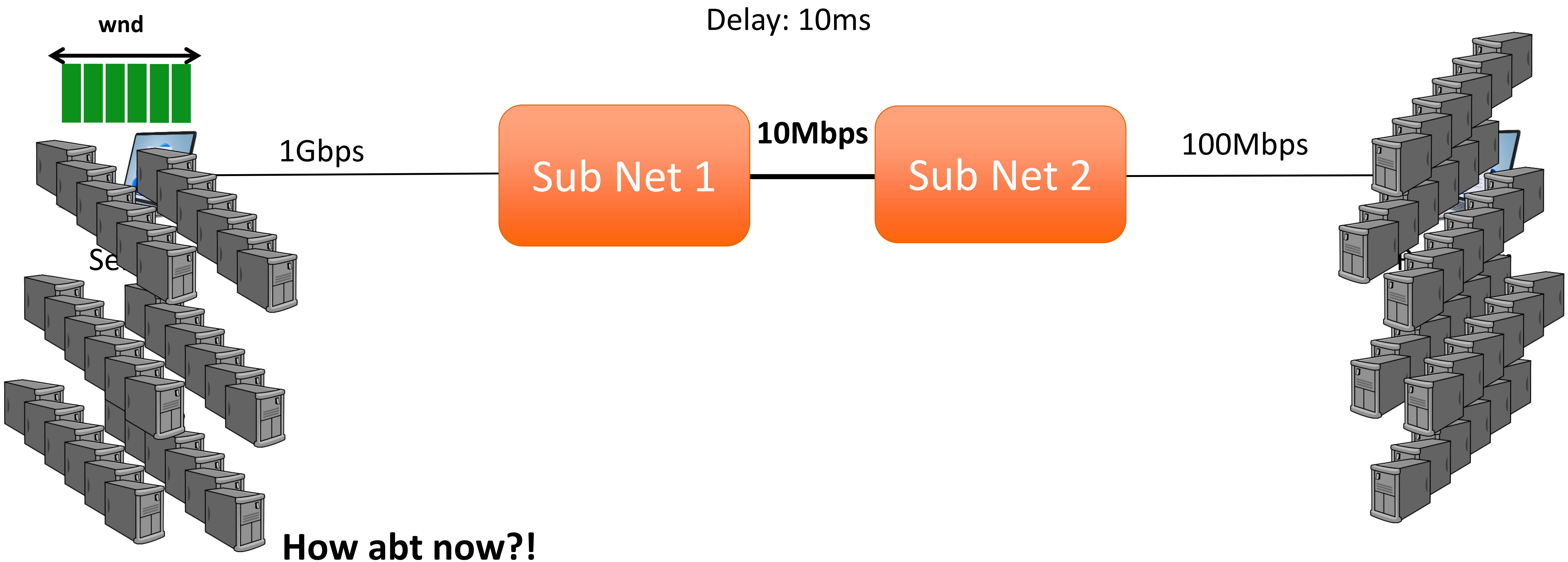
How abt when we want reliability? What is the best sending rate?

- (recall from last lecture) TCP uses a sliding window
  - It controls the number of inflight packets (pkts sent but not Acked yet)
- What is throughput in terms of *wnd*?
  - $Thr = \frac{\#\, packets\ sent}{Time} = \frac{wnd}{RTT}$
- So, desired *wnd* = BW × RTT



How abt when we want reliability? What is the best sending rate? ✓

But what if ?



How abt now?!

We don't know the bottleneck BW

We don't know how many other users are there

But we should come up with an acceptable sending rate!

Congestion Control (CC) on the Internet is about . . .

- Maximizing **your** performance objective
  - **I** want Higher throughput
  - **I** want Lower delay
  - **I** want no lost packets
- While being a **good citizen**
  - Be fair to others and respect them!
- All that without having any prior knowledge of the network or other users!

Let’s design a CC scheme

A Simple One!

(1) Probe

Do Some Basic Measurements

- We need at least to know two things: RTT and bottleneck BW
- How to find RTT?
  - Use sRTT estimations from older (like SYN) packets
- How to find bottleneck link bandwidth?
  - **Try and Error!**
  - Start from a small number, If everything is fine, increase your guess & If something is wrong, decrease your guess
  - How much increase at each step?
    - Linearly? Exponentially?
  - How much decrease at each step?
    - Linearly? Exponentially?

(2) Use your best guess and update it

- After knowing **congestion window** (cwnd = BW×RTT), keep using it!
- Do we need to update value of *cwnd?*
  - Bottleneck link BW may change!
  - So, oscillate around your best guess

(3) Don't hug the links forever!

Fairness

- What does it mean if you detect a packet loss?
  - Someone else is in the network?
  - Backoff and let others probe/use the network

First TCP CC schemes were born more than 30 years ago

- **TCP Tahoe** & **TCP Reno** (∼1989) use the same three-step algorithm
  - Take a good guess about the bottleneck bandwidth
  - Conservatively increase your sending rate
  - Backoff in case of loss and let others use the network too

Van Jacobson

In more technical terms . . .

Step 1) **Slow Start**: take a good guess

- Each RTT, Double your cwnd, until you detect loss of packets

Step 2) **Congestion Avoidance**: see if bandwidth is changed (but cautiously)

- Each RTT, Increase your cwnd by only one packet

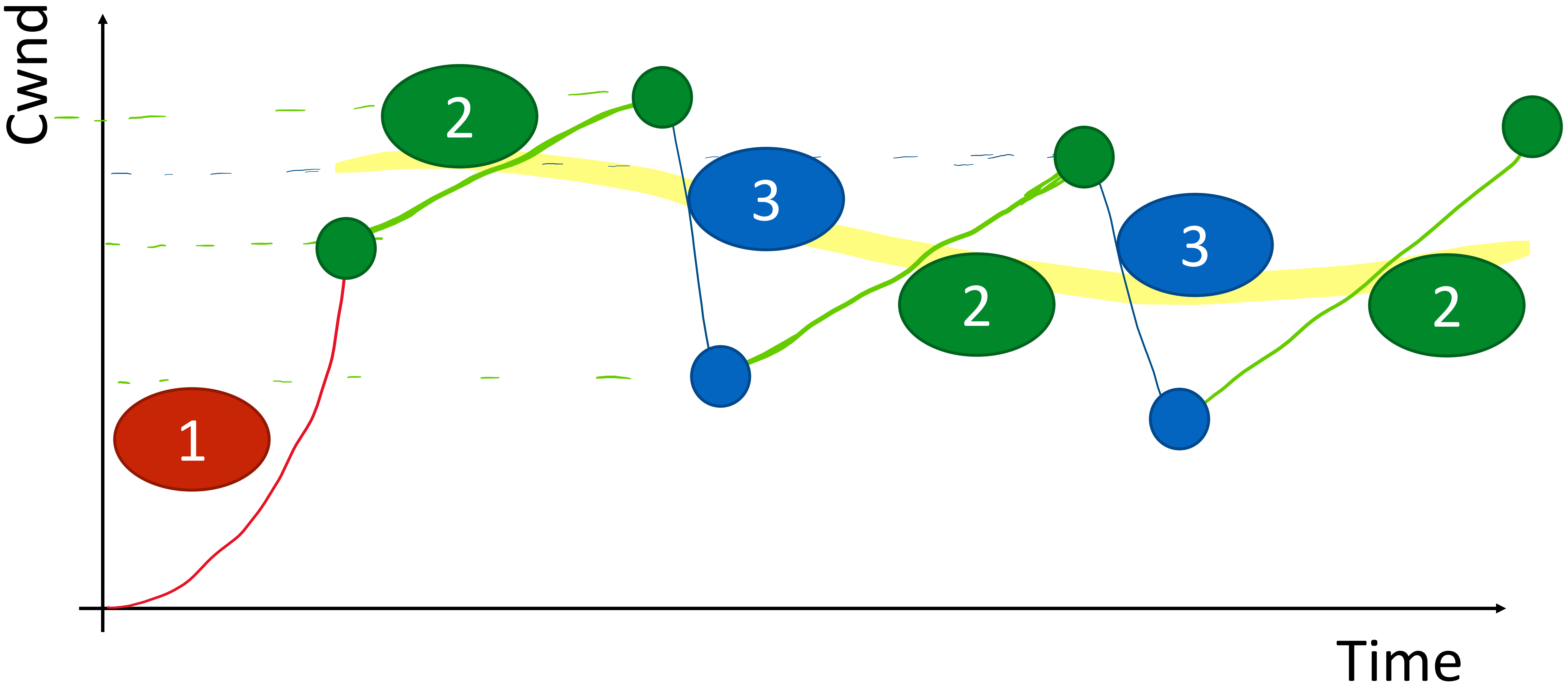
Step 3) **Loss Detection**: be a good citizen and backoff!

- E.g., If you detect packet loss, divide your 1st guess by 2

AIMD:

Additive Increase Multiplicative Decrease

A classic set of solutions {Reno, Tahoe, New Reno, etc.}



More Details . . .

Not All Losses Are the Same

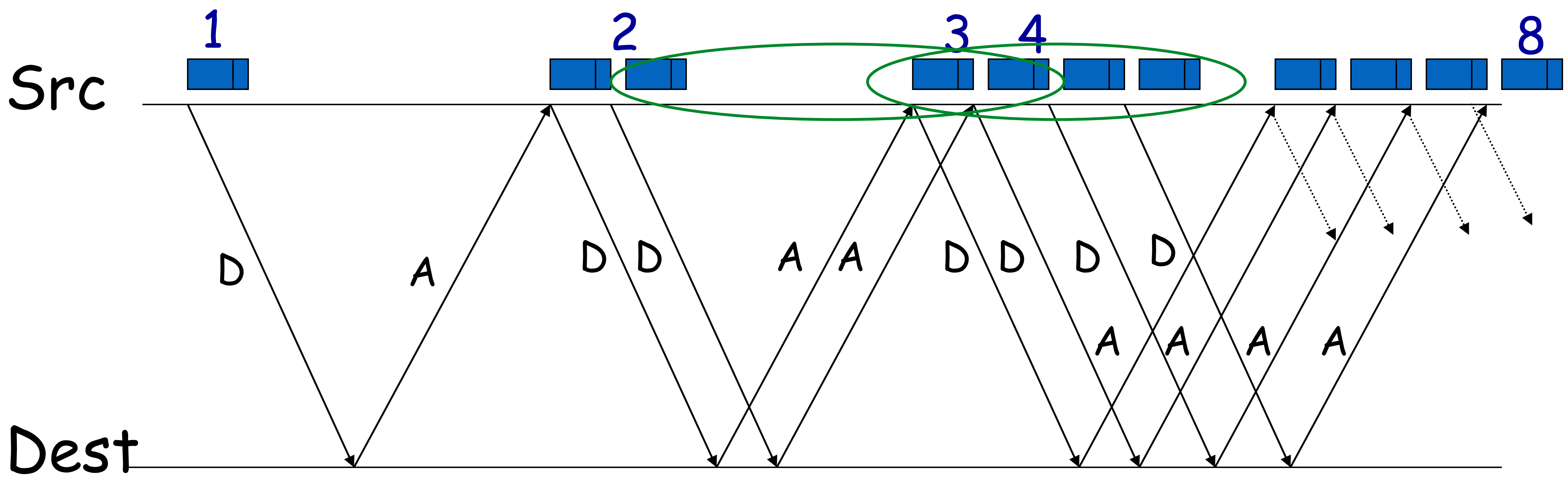
- Duplicate ACKs: isolated loss
  - Still getting ACKs
- Timeout: much more serious
  - Not enough dupacks
  - Must have suffered several losses

- Will adjust rate differently for each case

- Duplicate ACK: cwnd = cwnd/2
- Timeout: cwnd = 1

Slow Start in Action

- For each RTT: double CWND
- We have a sliding window in practice not a jumping window!
- So, how to implement this?
  - for each ACK, cwnd += 1
  - Linear increase per ACK = exponential increase per RTT

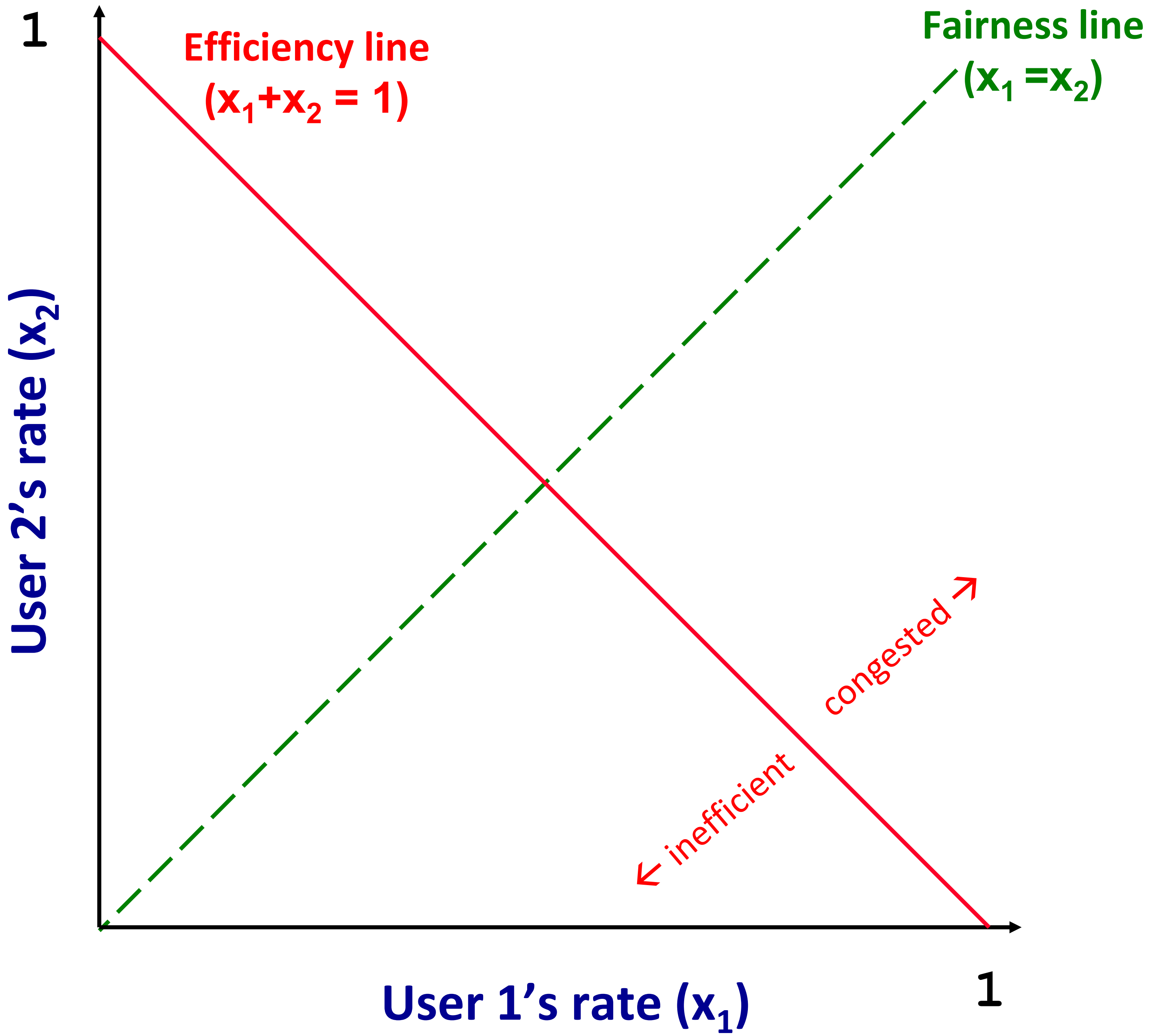


Why AIMD?

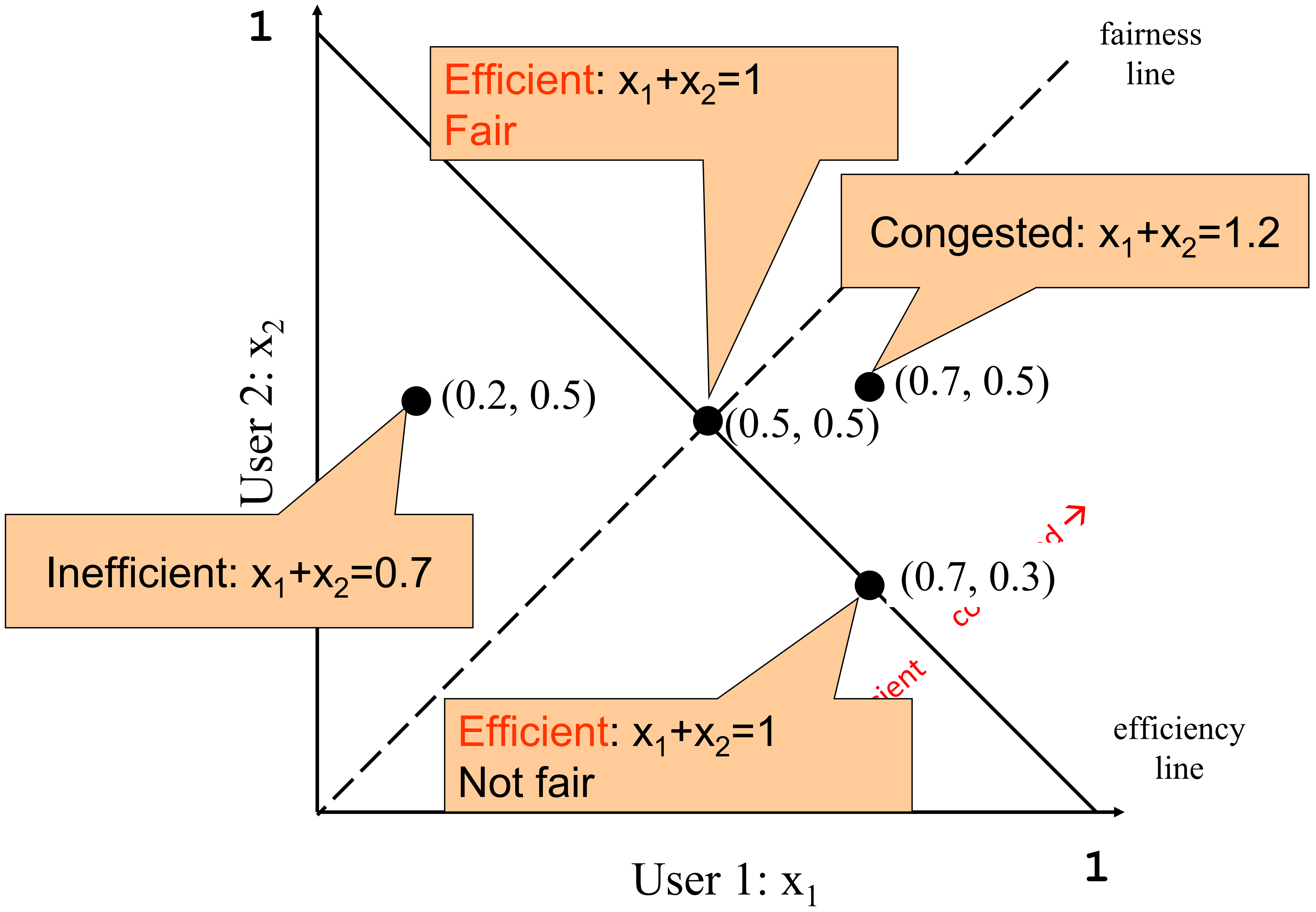
- Some rate adjustment options: Every RTT, we can
  - Multiplicative increase or decrease: CWND$\rightarrow$ a*CWND
  - Additive increase or decrease: CWND$\rightarrow$ CWND + b
- Four alternatives:
  - AIAD: gentle increase, gentle decrease
  - AIMD: gentle increase, drastic decrease
  - MIAD: drastic 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 $x_1 = x_2$



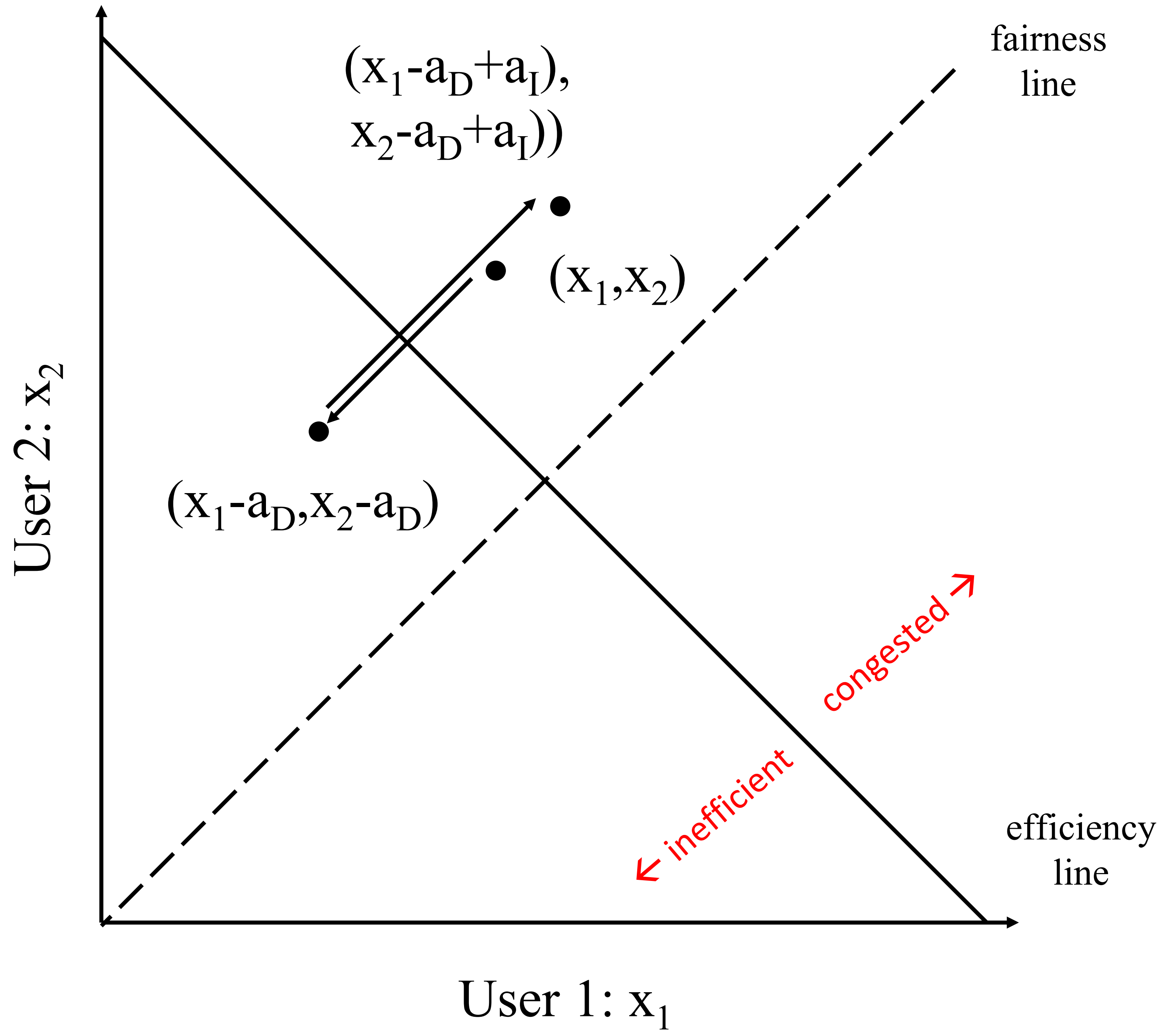
Example



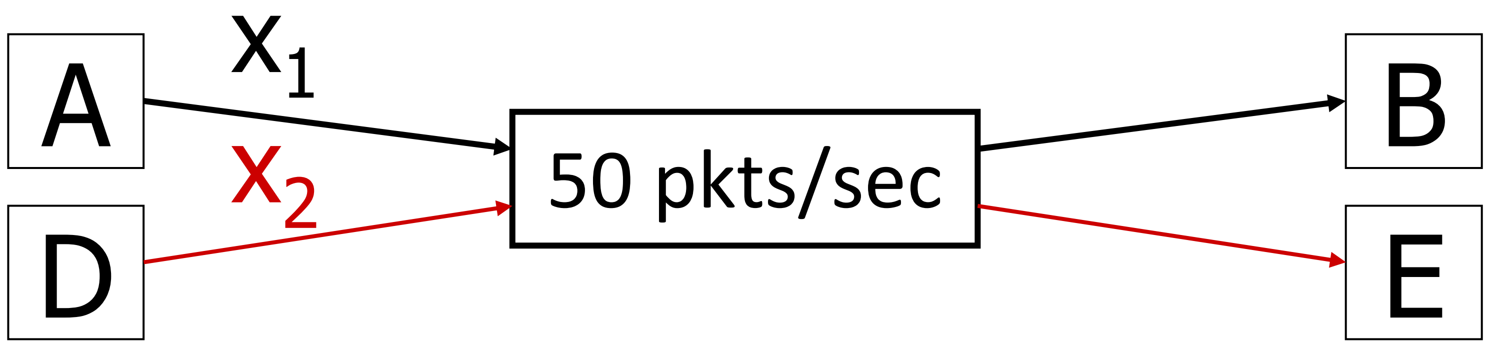
1
fairness line
Efficient: x1+x2=1
Fair
Congested: x1+x2=1.2
User 2: x2
(0.2, 0.5)
(0.5, 0.5)
(0.7, 0.5)
(0.7, 0.3)
Inefficient: x1+x2=0.7
Efficient: x1+x2=1
Not fair
efficiency line
User 1: x1
1

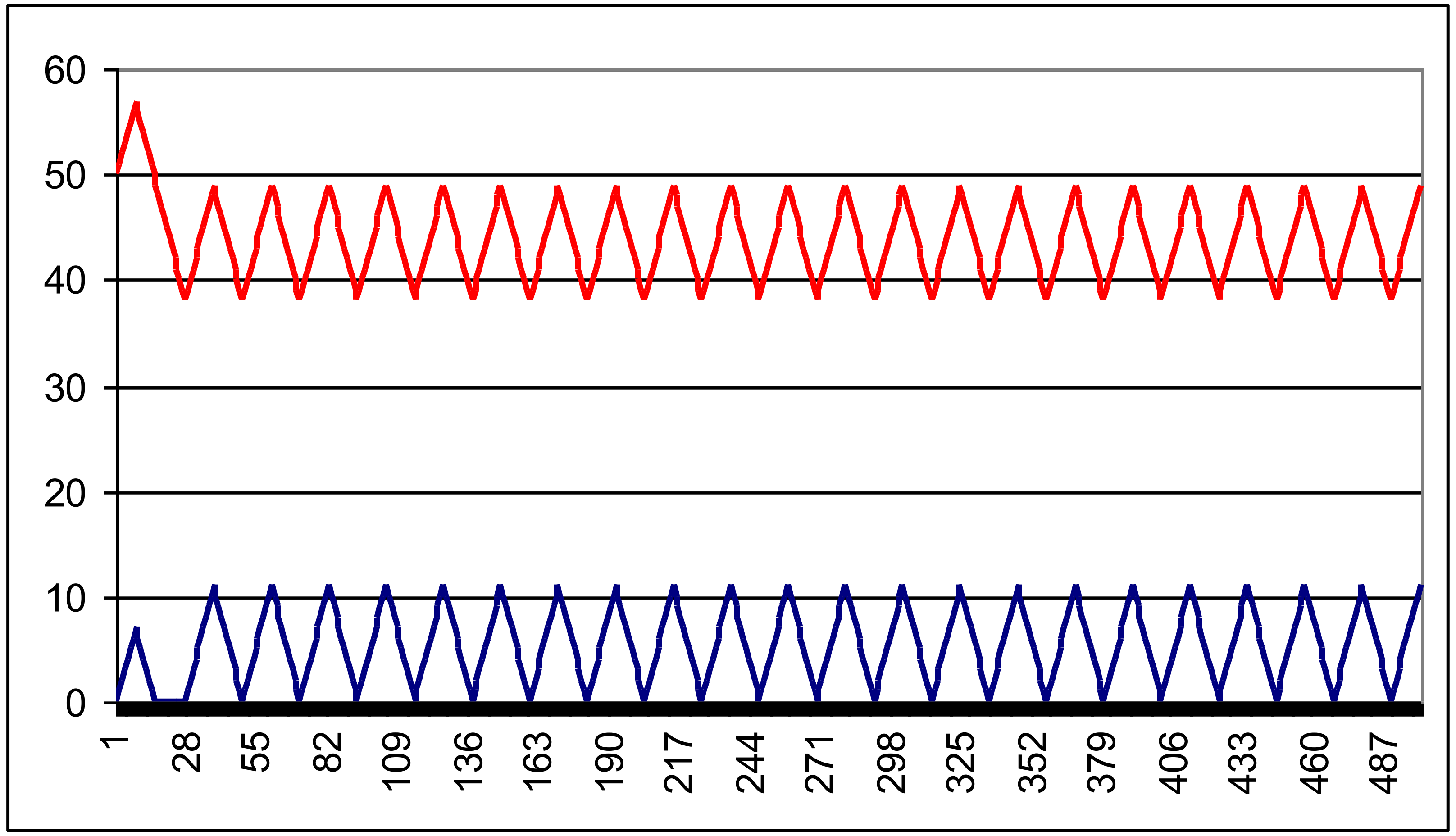
AIAD

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



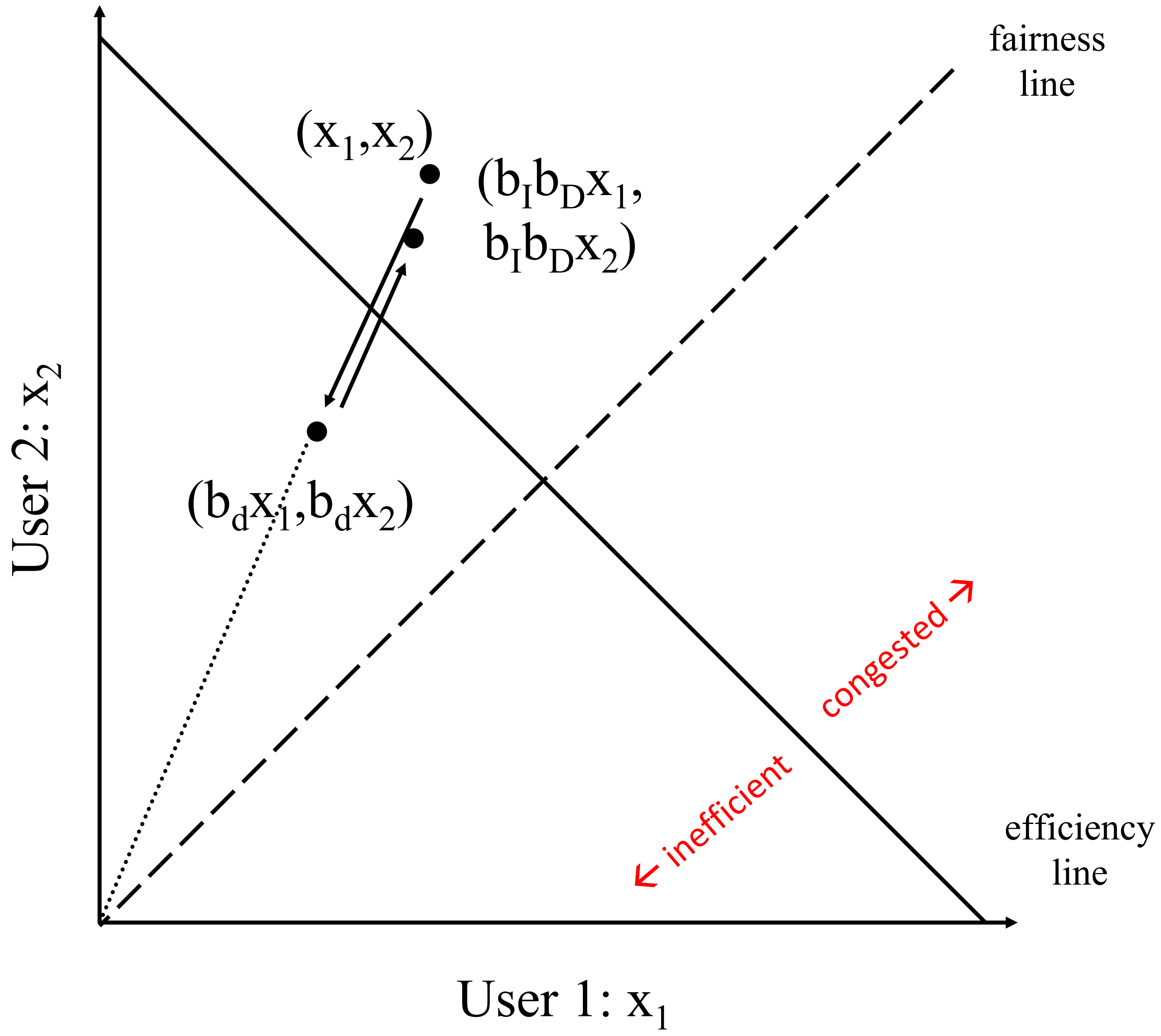
AIAD Sharing Dynamics





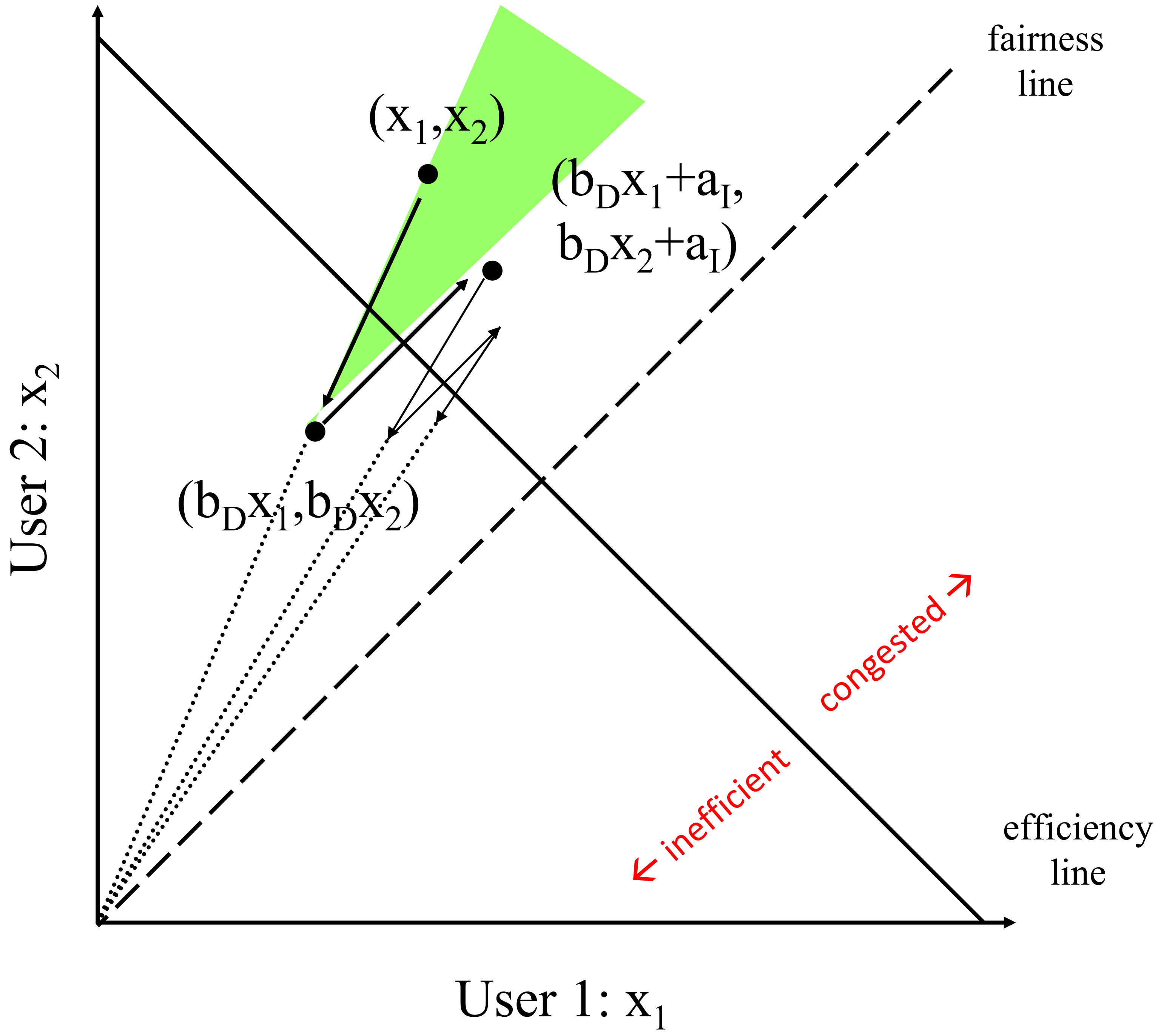
MIMD

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

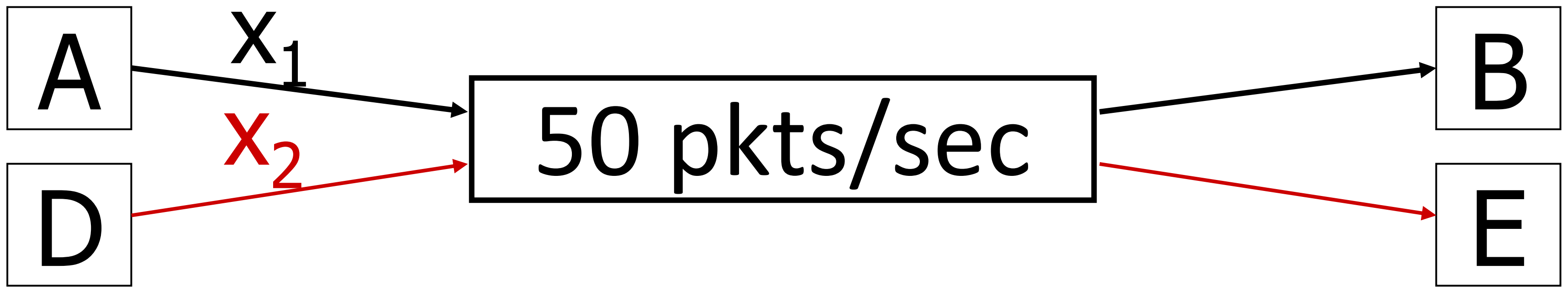


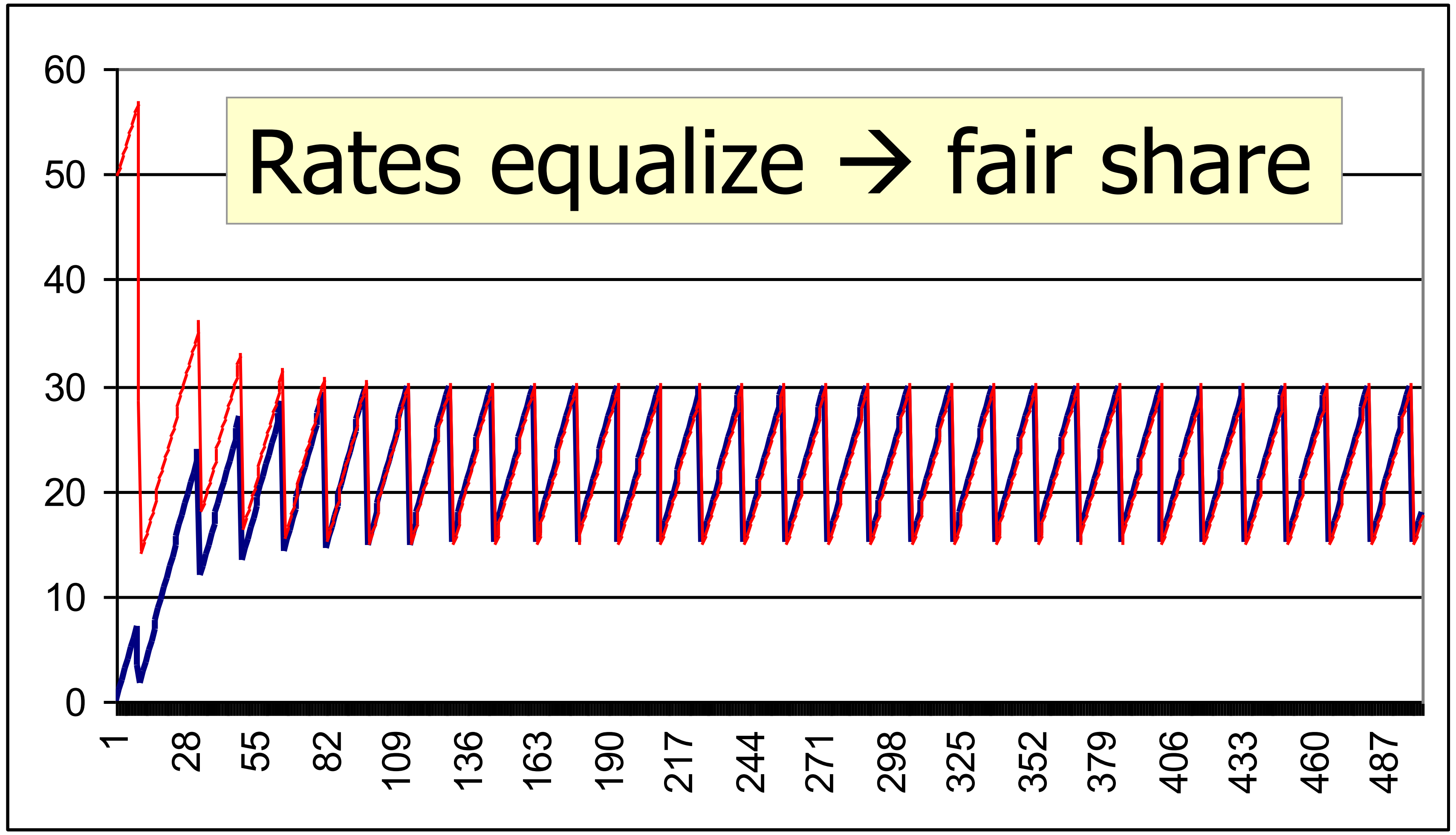
AIMD

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



AIMD Sharing Dynamics

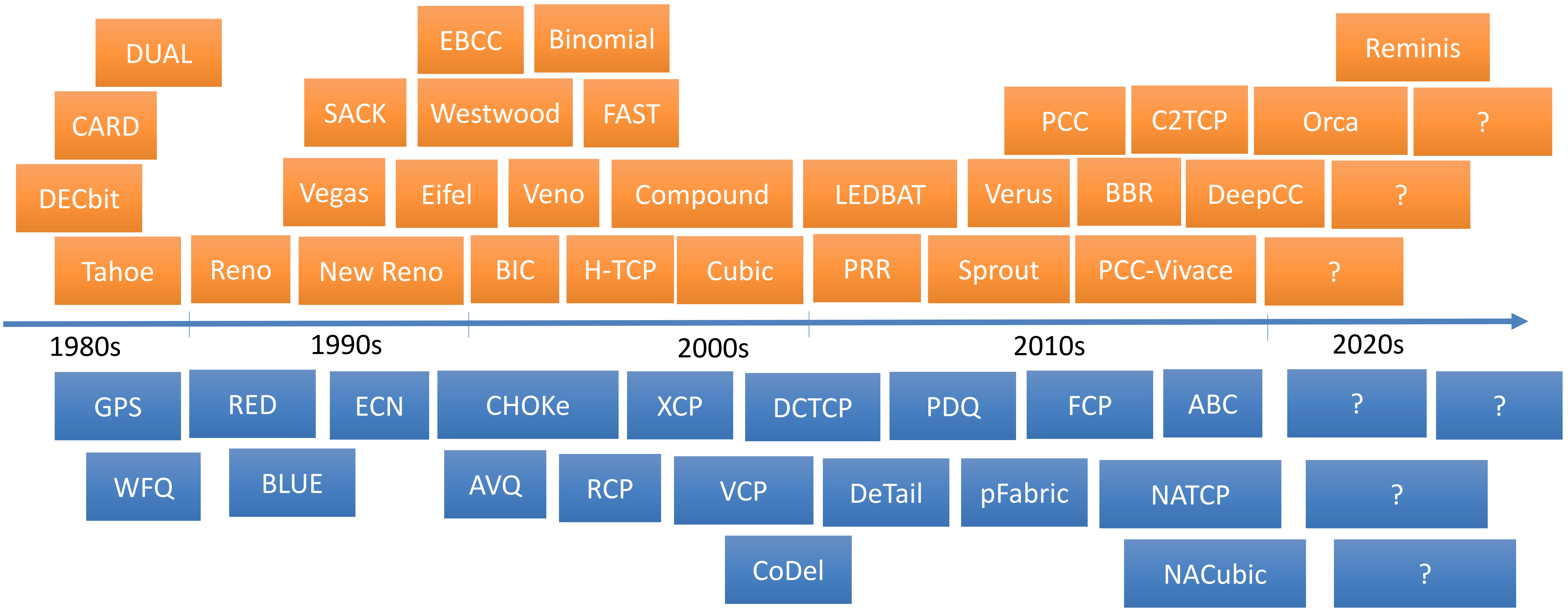




Do we still use a CC protocol from 1989?!

Short answer: **No!**

The march of congestion control mechanisms



Why is that and why it matters?

Deployed TCP CC schemes . . .

- The latest Linux Kernel alone includes more than 15 different TCP CC flavors!
  - **TCP Cubic (the default TCP)**
  - Vegas
  - BBR
  - TCP Reno
  - BIC
  - CDG
  - DCTCP
  - Westwood
  - Highspeed TCP
  - Illinois
  - Veno
  - …

You can change your CC algorithm. e.g., in Linux:
sudo sysctl -w net.ipv4.tcp_congestion_control=“vegas"

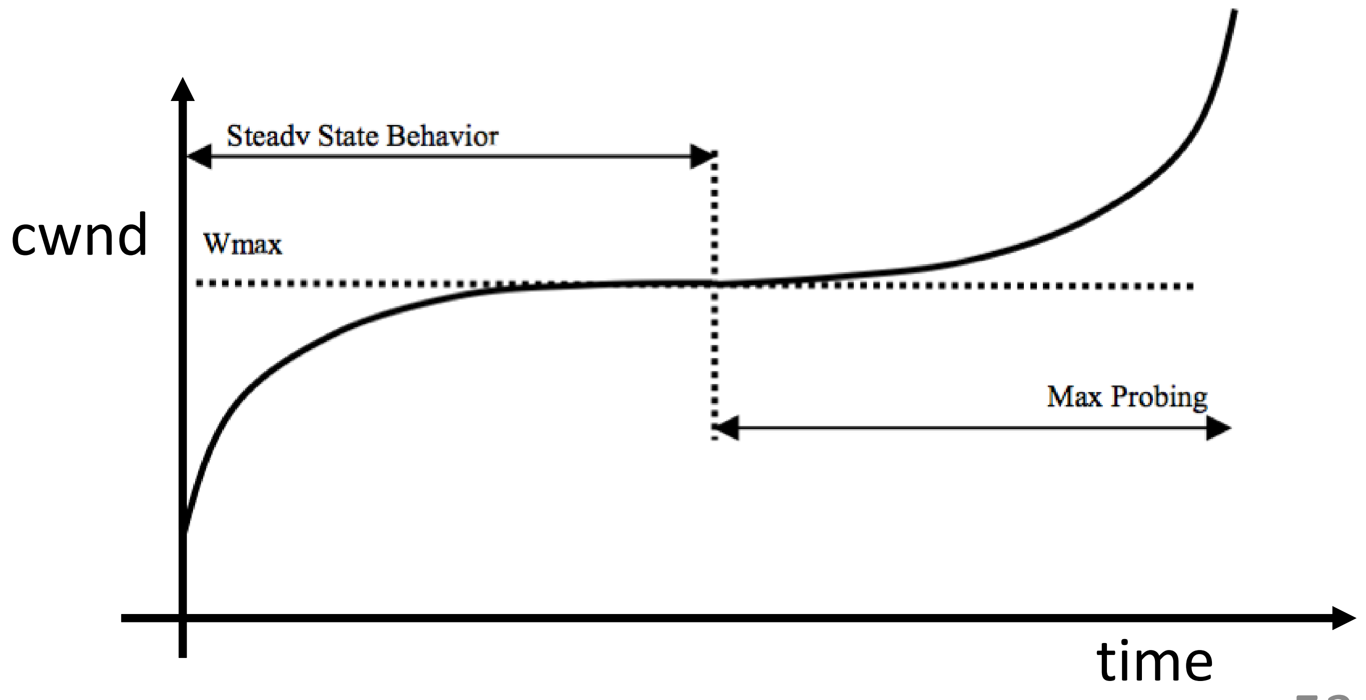
TCP Cubic (2008)

- The default TCP CC algorithm in macOS, Linux, Windows, and Android.
- Cubic follows the three-step algorithm **but** changes each part a little bit!
- E.g., it replaces the AI (additive increase) part with a cubic function of time

$$cwnd = C \cdot (t - K)^3 + W_{\max}$$

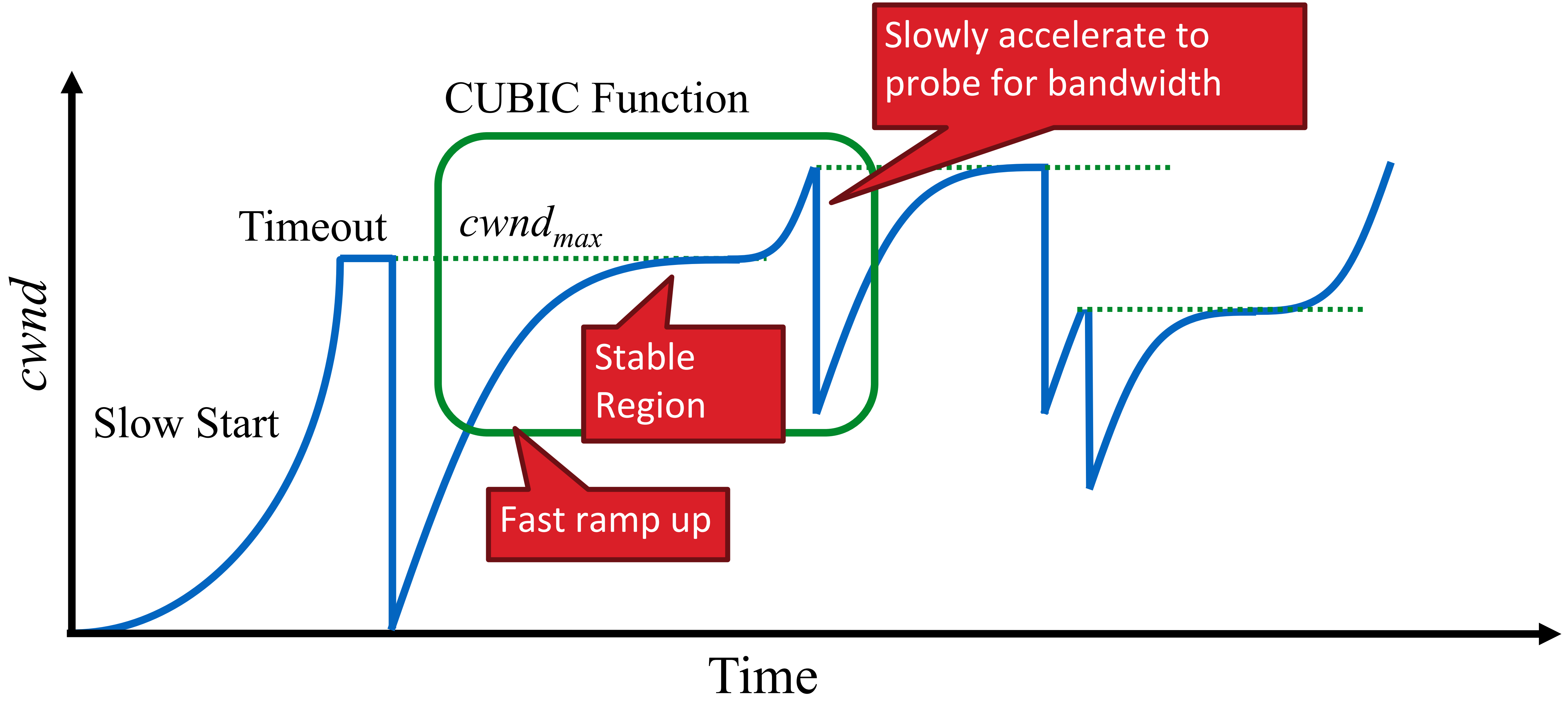
- K is updated when a packet is lost

$$K = \sqrt[3]{W_{max} \times \beta / C}$$



TCP Cubic

- Less wasted bandwidth due to fast ramp up
- Stable region and slow acceleration help maintain fairness



Putting all together

- In a packet switching network, congestion is inevitable
  - Internet does not reserve resources
- TCP handles congestion control with a simple three-step algorithm
  - Exponentially explore the available link BW
  - Cautiously (~linearly) probe for more BW
  - Backoff and let other use the network
- Congestion control is a hot active research topic
  - New environments require new designs
  - New demands lead to new designs
  - We are still far from universally solving this!