Network Programming
TCP Congestion Control

Minchen Yu

SDS@CUHK-SZ

Spring 2026

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

SCHOOL OF
DATA SCIENCE
數據科學學院

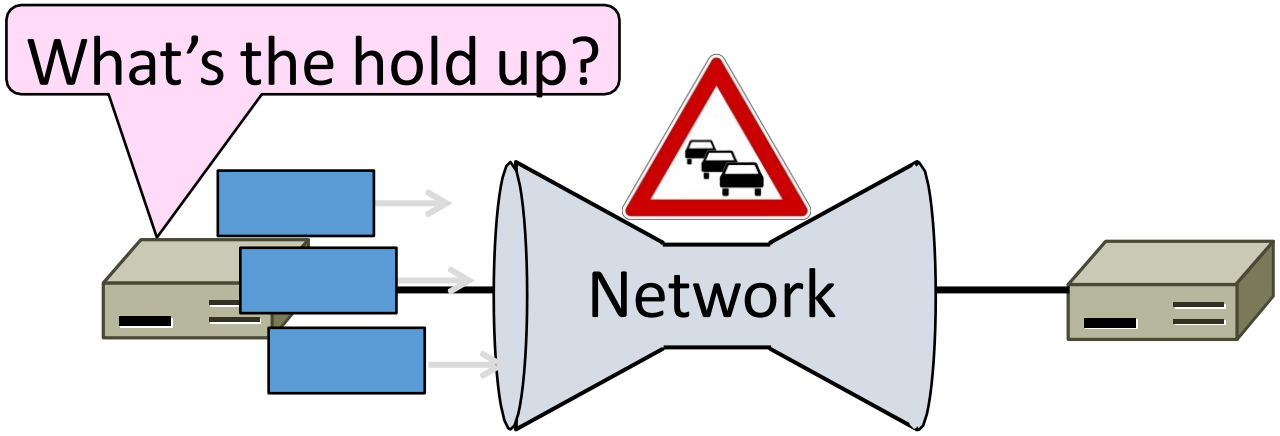
TCP to date

- We can set up and tear connections
  - Connection establishment and release handshakes
- Keep the sending and receiving buffers from overflowing (flow control)

What's missing?

Network congestion

- A “traffic jam” in the network
  - Later we will learn how to control it

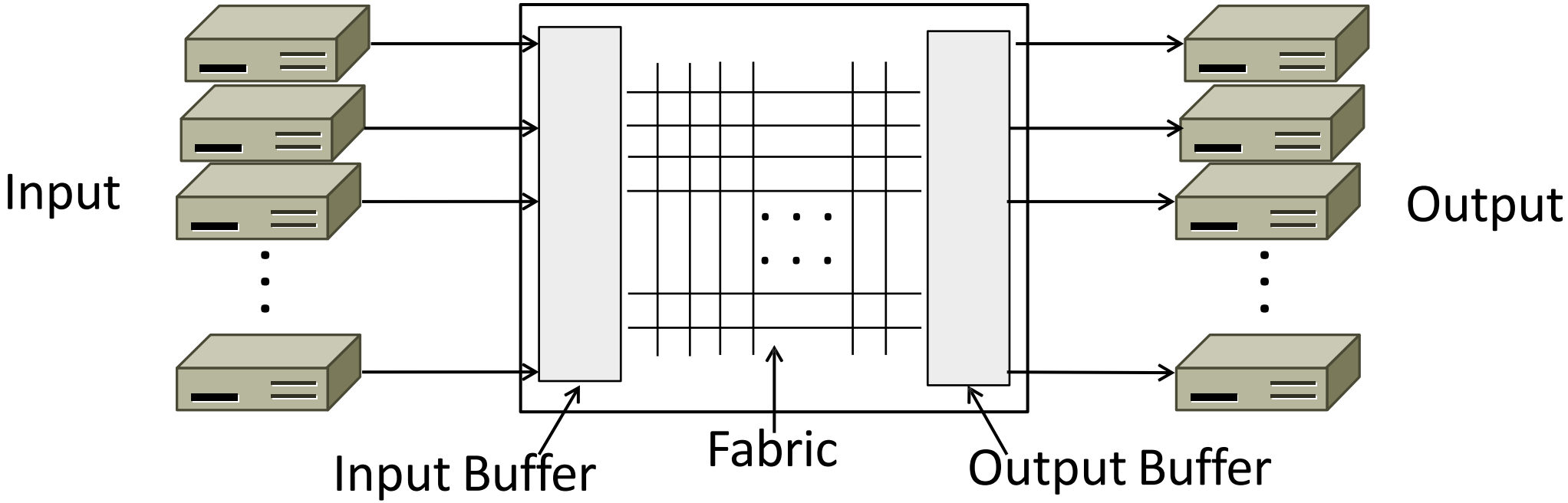


Congestion collapse in the 1980s

- Early TCP used fixed size window (e.g., 8 packets)
  - Initially fine for reliability
- But something happened as the network grew
  - Links stayed busy but transfer rates fell by orders of magnitude!

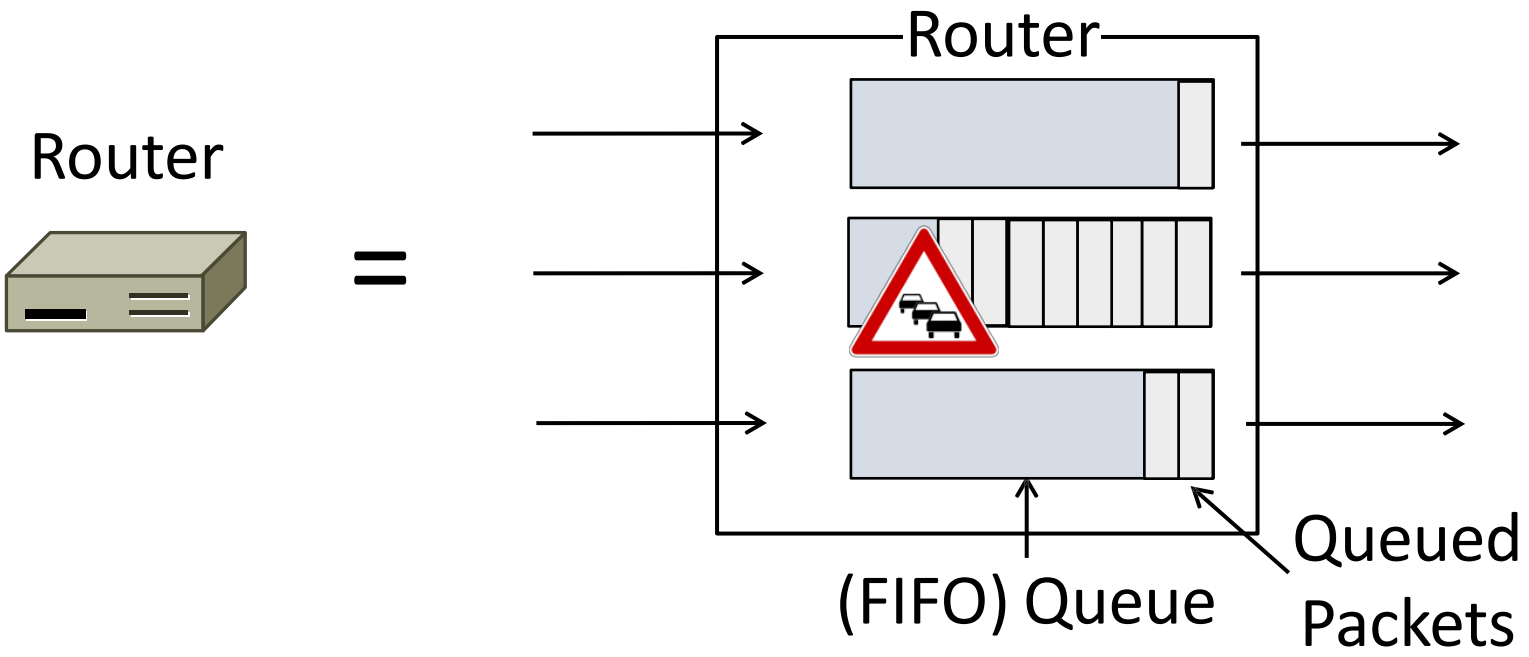
Nature of congestion

- Routers/switches have internal buffering



Nature of congestion

- Simplified view of per port output queues
  - Typically FIFO (First In First Out), discard when full



Nature of congestion

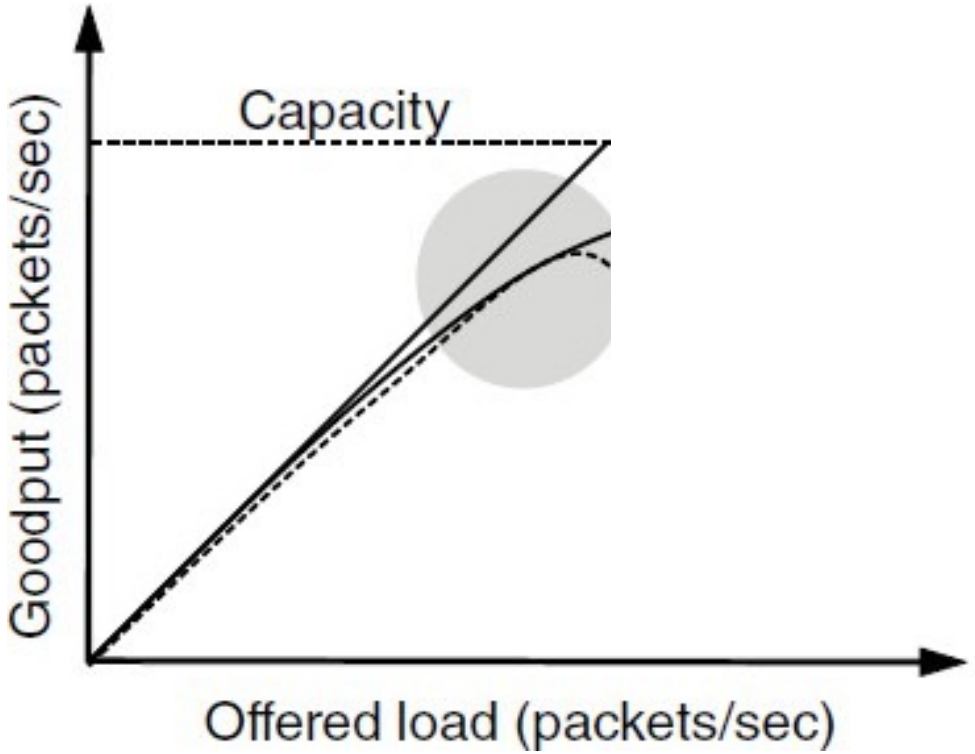
Queues absorb bursts when input > output rate

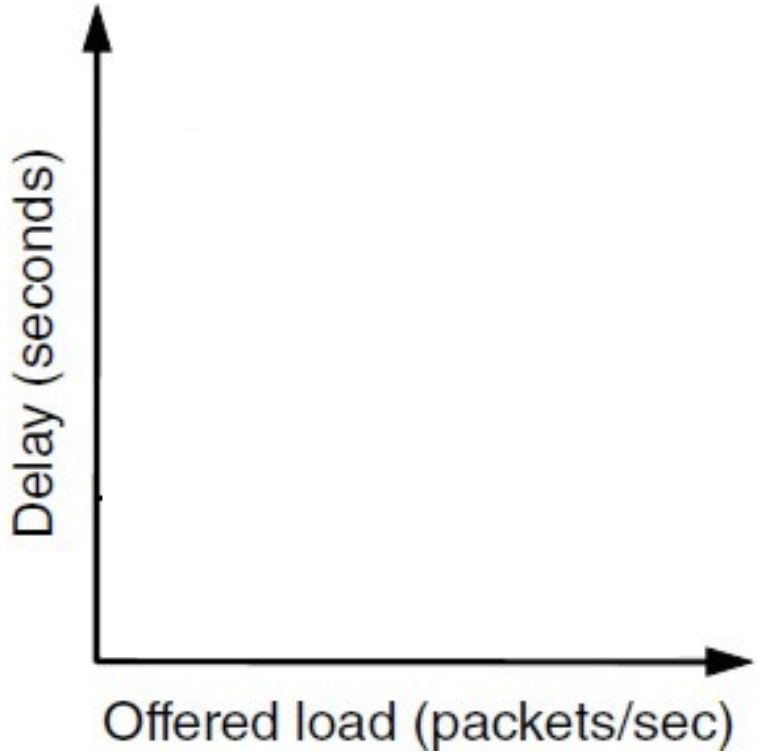
But if input > output rate persistently, queue will overflow -> congestion

Congestion is a function of the traffic patterns – can occur even if every link has the same capacity

Effects of congestion

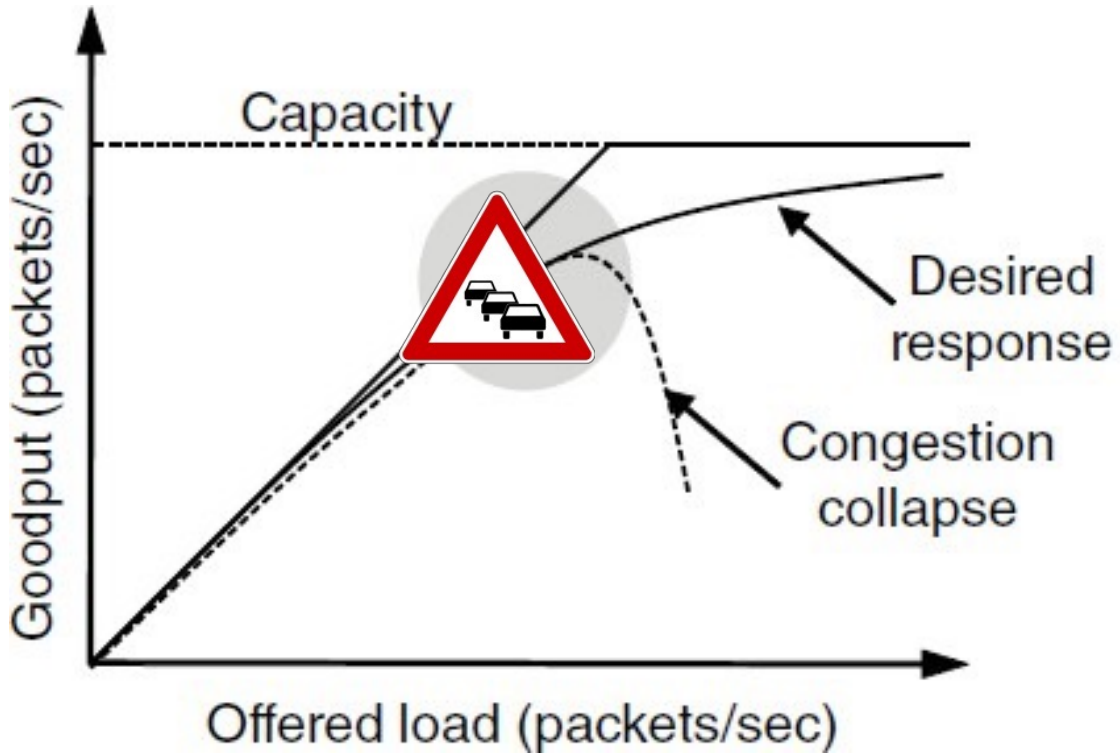
What happens to performance as we increase load?

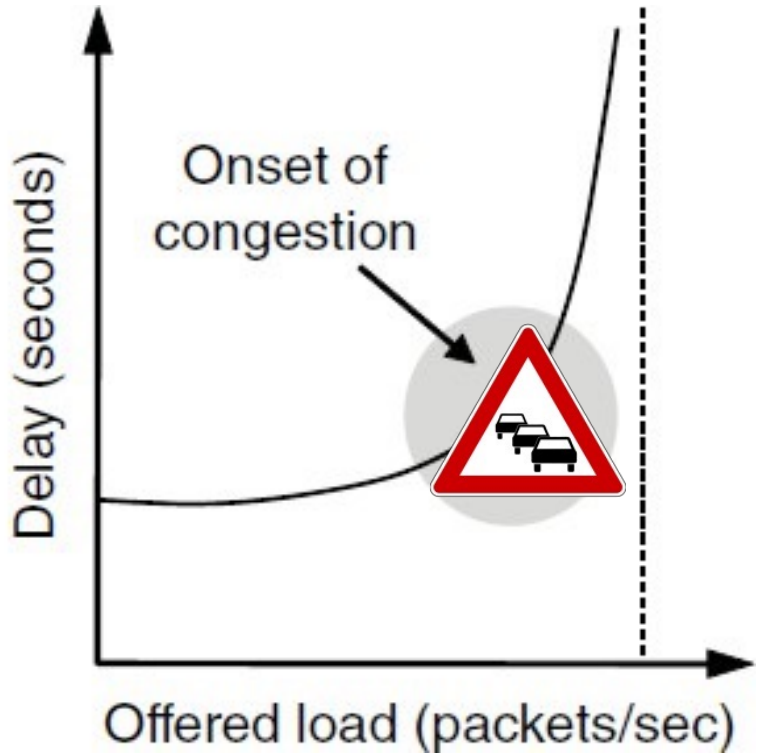




Effects of congestion

What happens to performance as we increase load?





Effects of congestion

- As offered load rises, congestion occurs as queues begin to fill:
  - Delay and loss rise sharply with load
  - Throughput < load (due to loss)
  - Goodput << throughput (due to spurious retransmissions)
- None of the above is good!
  - Want network performance just before congestion

TCP Tahoe/Reno

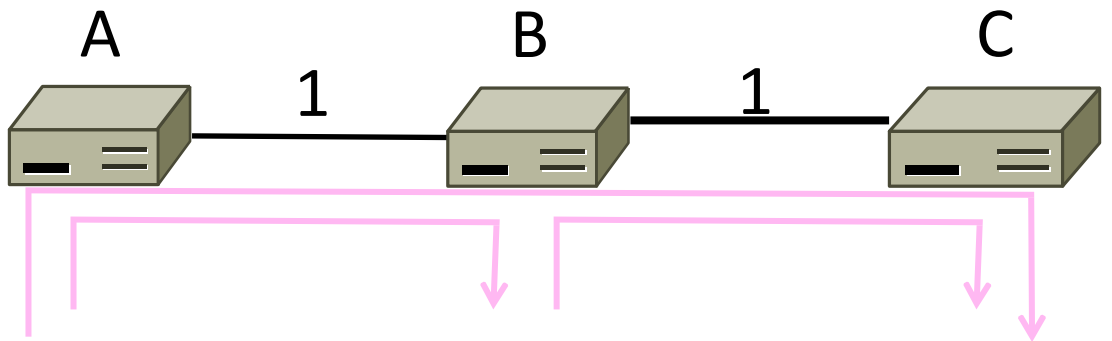
- TCP extensions and features we will study:
  - AIMD
  - Fair Queuing
  - Slow-start
  - Fast Retransmission
  - Fast Recovery

Bandwidth allocation

- Important task for network is to allocate its capacity to senders
  - Good allocation is both efficient and fair
- Efficient: most capacity is used but there is no congestion
- Fair: every sender gets a reasonable share of the network

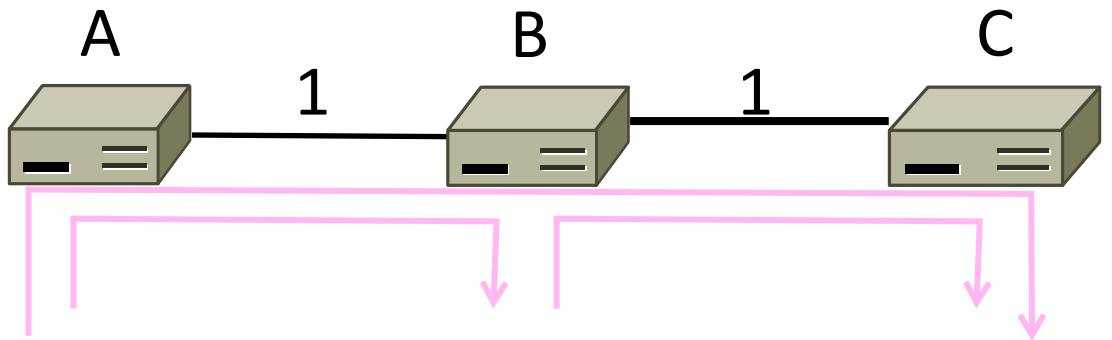
Efficiency vs. Fairness

- Cannot always have both!
  - Example network with traffic:
    - A->B, B->C and A->C
  - How much traffic can we carry?



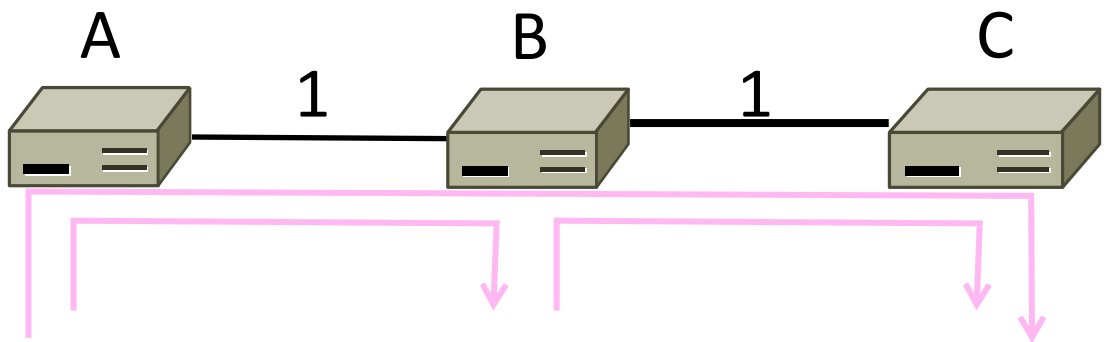
Efficiency vs. Fairness

- If we care about fairness:
  - Give equal bandwidth to each flow
  - A->B: ½ unit, B->C: ½, and A->C: ½
  - Total traffic carried is 1 ½ units



Efficiency vs. Fairness

- If we care about efficiency:
  - Maximize total traffic in network
  - A->B: 1 unit, B->C: 1, Total traffic rises to 2 units! and A->C: 0

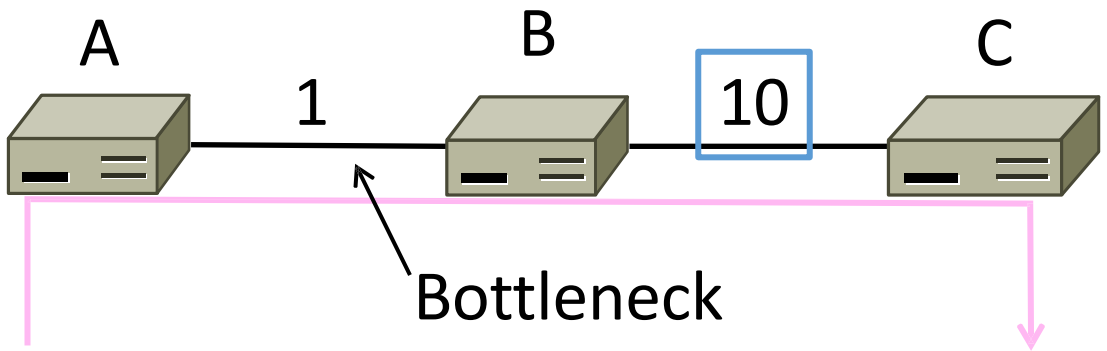


The slippery notion of fairness

- Why is “equal per flow” fair anyway?
  - A->C uses more network resources than A->B or B->C
  - Host A sends two flows, B sends one
- Not productive to seek exact fairness
  - More important to avoid starvation
    - A node that cannot use any bandwidth
  - “Equal per flow” is good enough

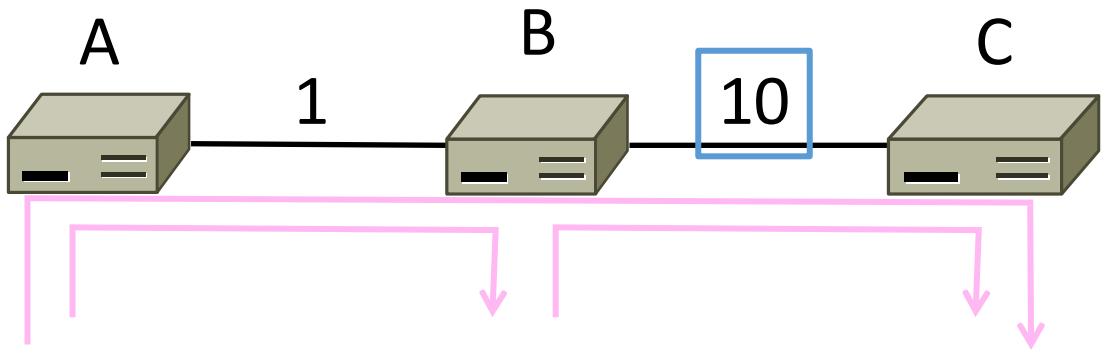
Generalizing “Equal per Flow”

- Bottleneck for a flow of traffic is the link that limits its bandwidth
  - Where congestion occurs for the flow
  - For A->C, link A–B is the bottleneck



Generalizing “Equal per Flow”

- Flows may have different bottlenecks
  - For A->C, link A–B is the bottleneck
  - For B->C, link B–C is the bottleneck
  - Can no longer divide links equally …



Max-Min fairness

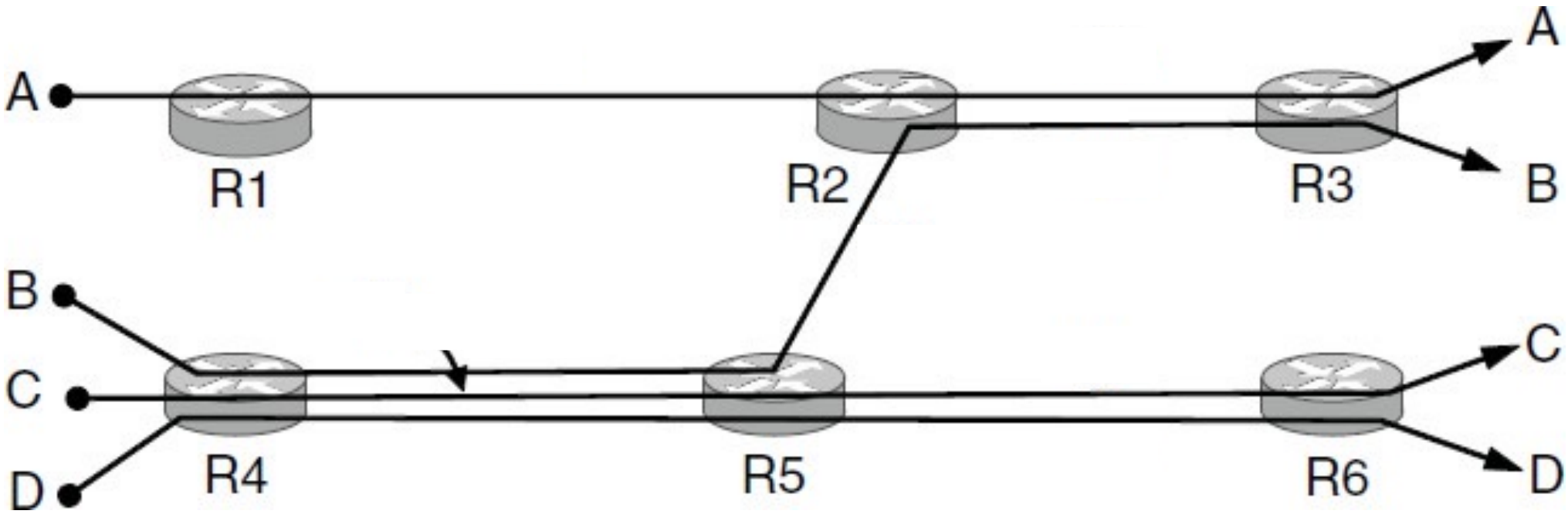
- Intuitively, flows bottlenecked on a link get an equal share of that link
- Max-min fair allocation is one that:
  - Increasing the rate of one flow will decrease the rate of a smaller flow
  - This “maximizes the minimum” flow

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

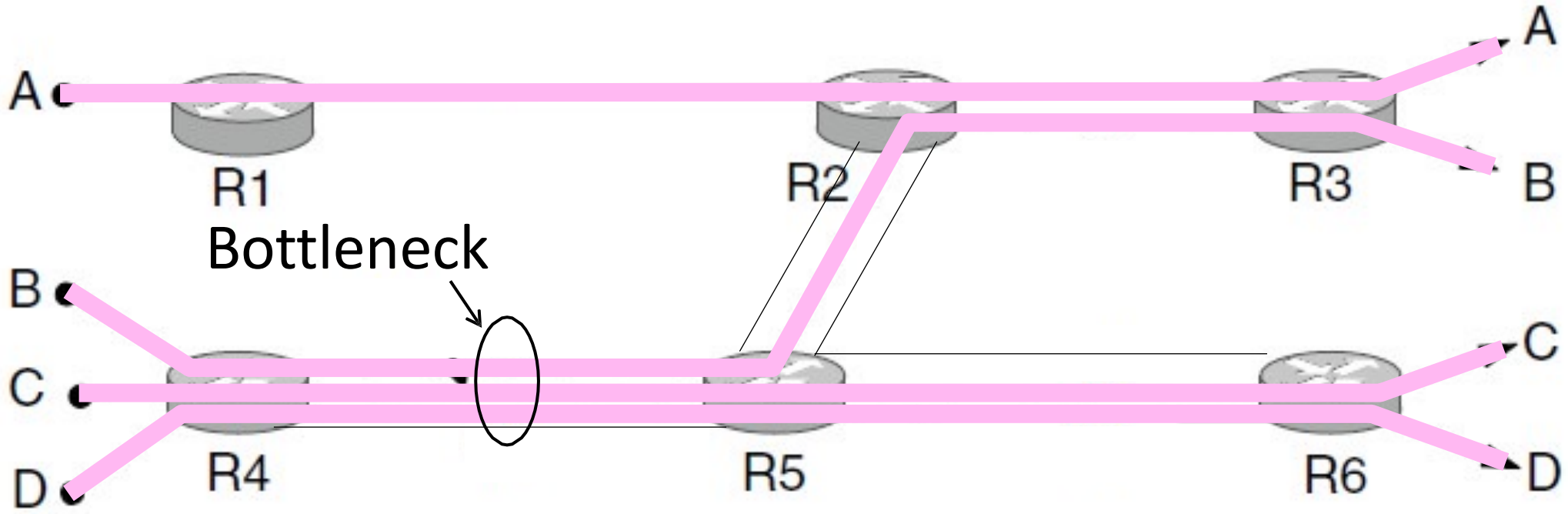
Max-Min example

- Example: network with 4 flows, link bandwidth = 1
  - What is the max-min fair allocation?



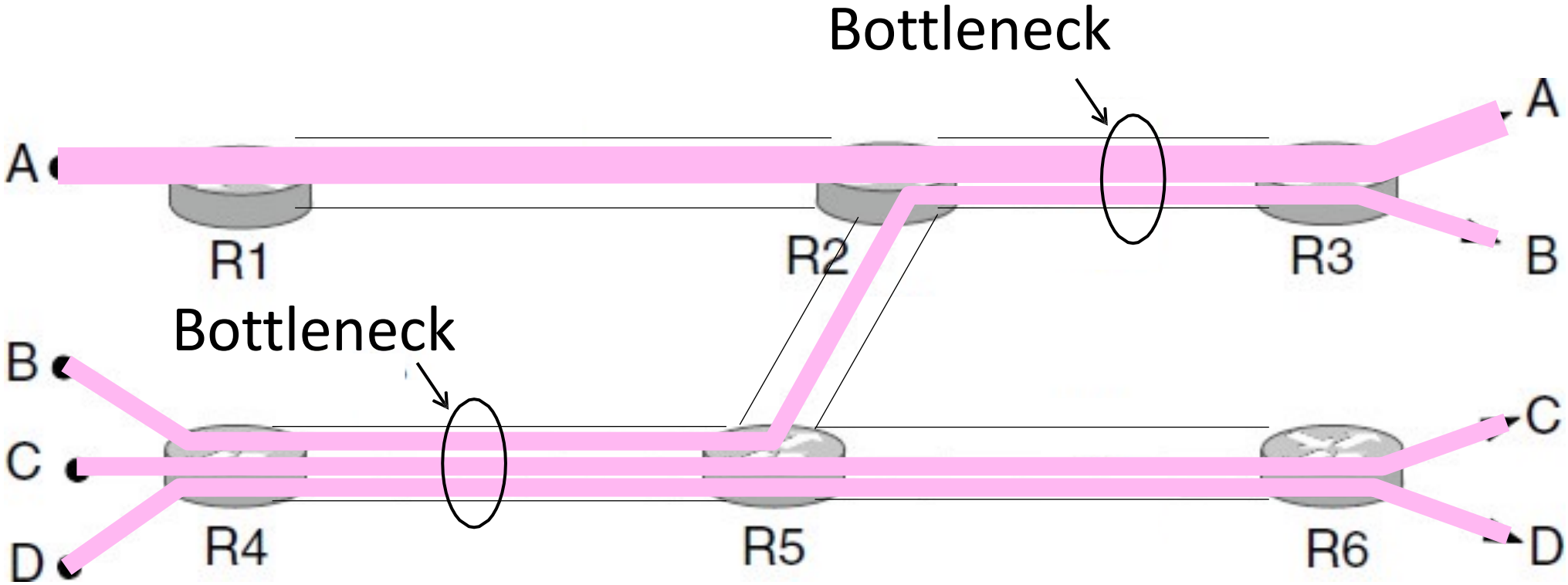
Max-Min example

- When rate=1/3, flows B, C, and D bottleneck R4—R5
  - Fix B, C, and D, continue to increase A



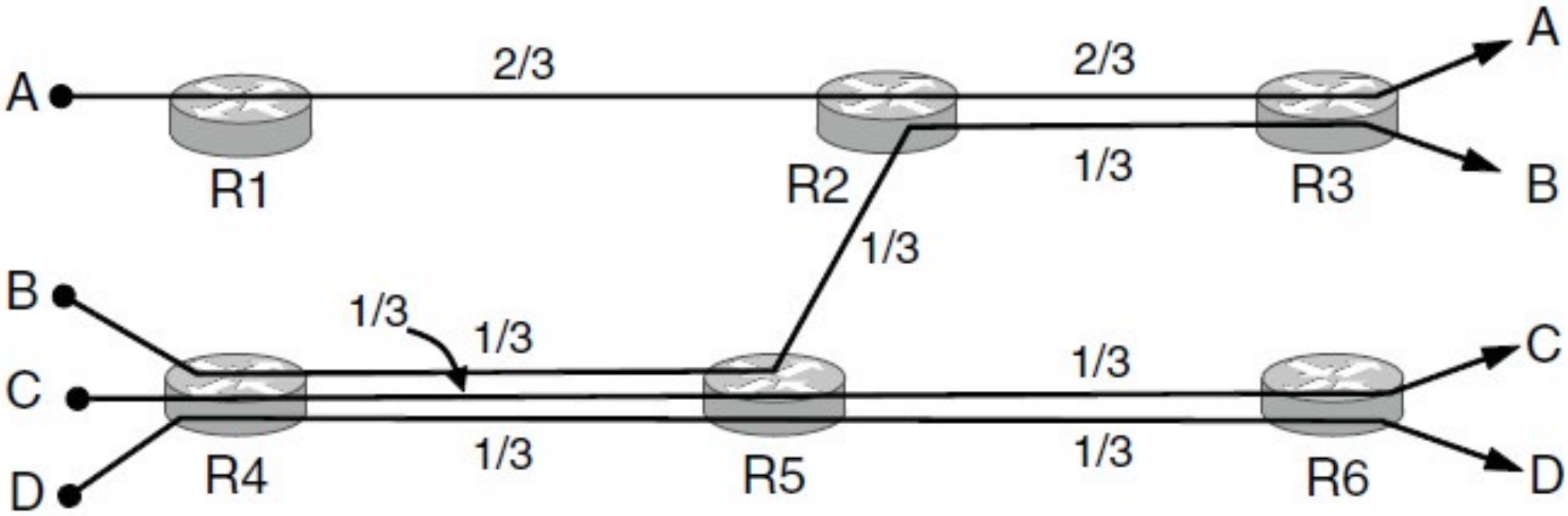
Max-Min example

- When rate=2/3, flow A bottlenecks R2—R3. Done.



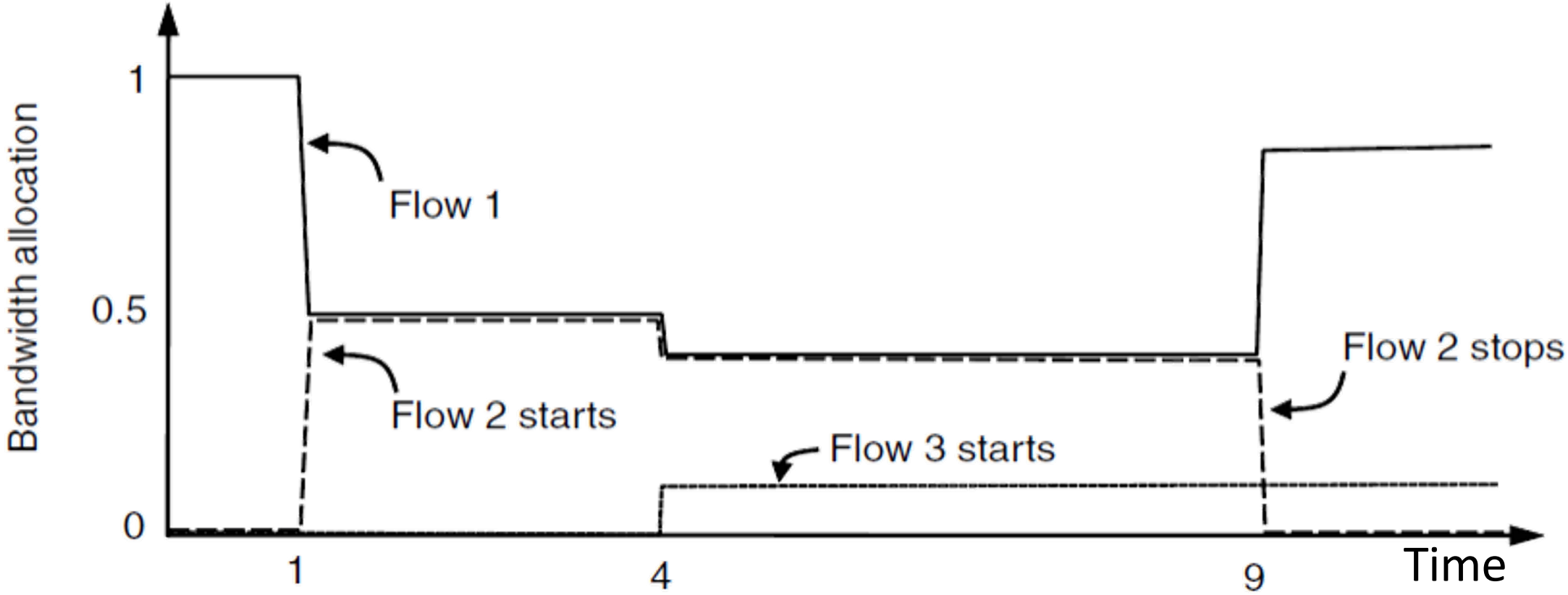
Max-Min example

- End with A=2/3, B, C, D=1/3, and R2—R3, R4—R5 full

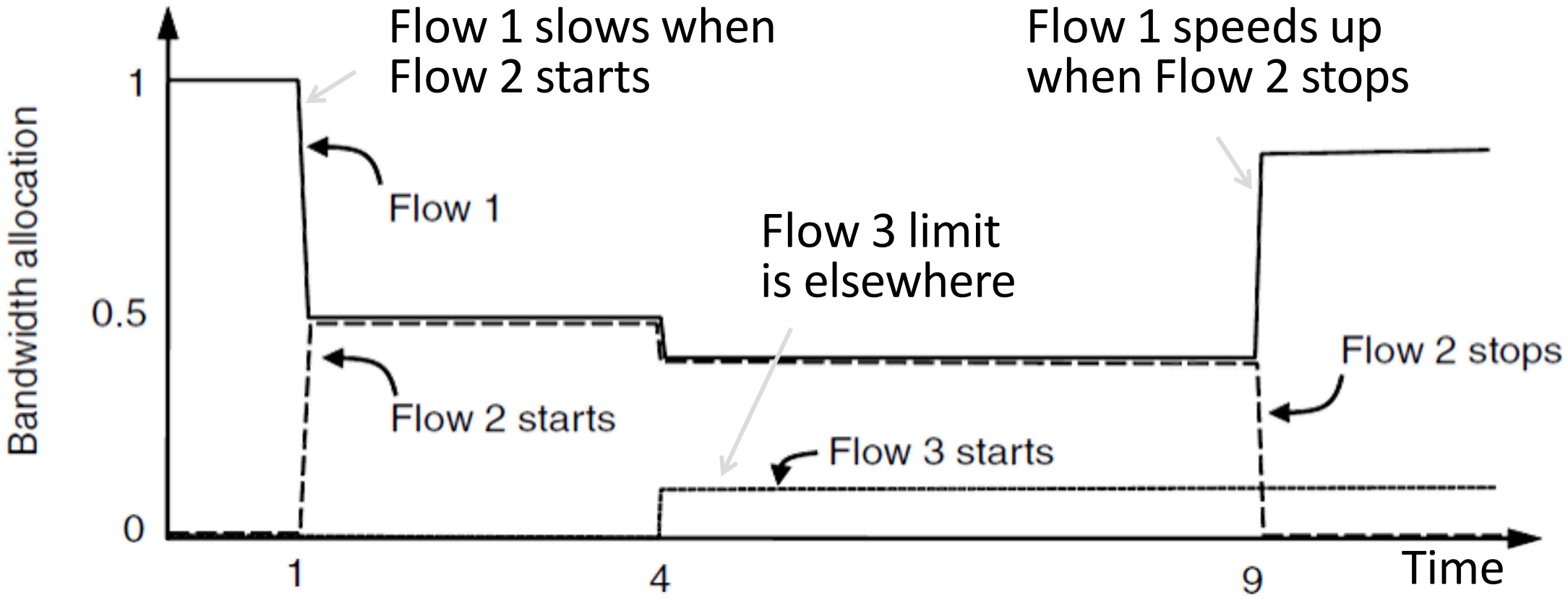


Adapting over Time

- Allocation changes as flows start and stop



Adapting over Time



Why is bandwidth allocation hard?

- Number of senders and their offered load changes
- Senders may be limited in other ways
  - Other parts of network or by applications
- Network is distributed; no single party has an overall picture of its state

Bandwidth allocation solution context

In networks without admission control (e.g., Internet) Transport and Network layers must work together

- Network layer sees congestion
  - Only it can provide direct feedback
- Transport layer causes congestion
  - Only it can reduce load

Bandwidth allocation solution overview

- Senders adapt concurrently based on their own view of the network
- Design this adaptation so the network usage as a whole is efficient and fair
  - In practice, efficiency is more important than fairness
- Adaptation is continuous since offered loads continue to change over time

Bandwidth allocation models

- Open loop versus closed loop
  - Open: reserve bandwidth before use
  - Closed: use feedback to adjust rates
- Host versus Network support
  - Who is sets/enforces allocations?
- Window versus Rate based
  - How is allocation expressed?

TCP is a closed loop, host-driven, and window-based

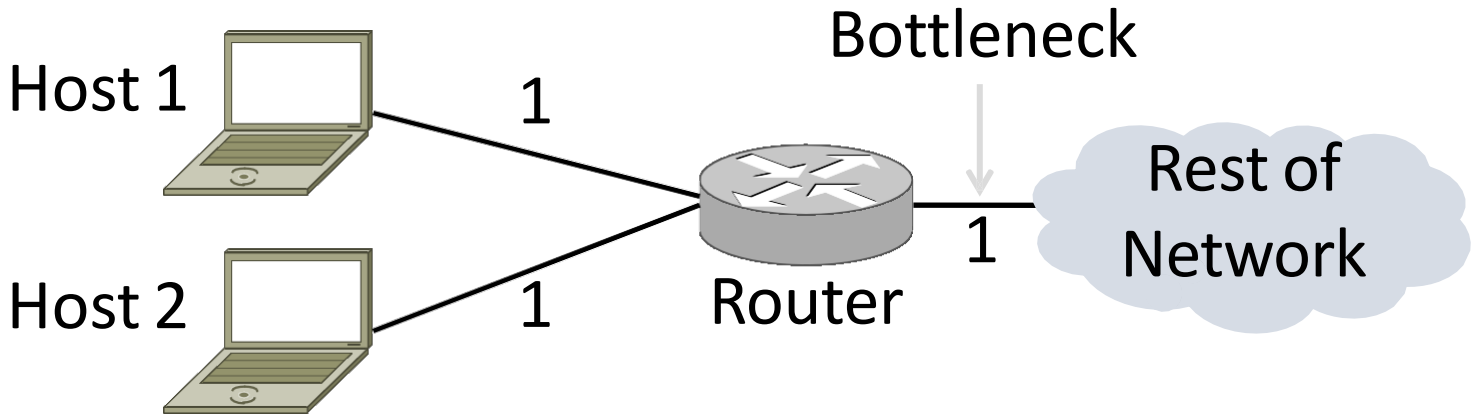
Additive Increase Multiplicative Decrease

- AIMD is a control law hosts can use to reach a good allocation
  - Hosts additively increase rate while network not congested
  - Hosts multiplicatively decrease rate when congested
  - Used by TCP

- Let's explore the AIMD game …

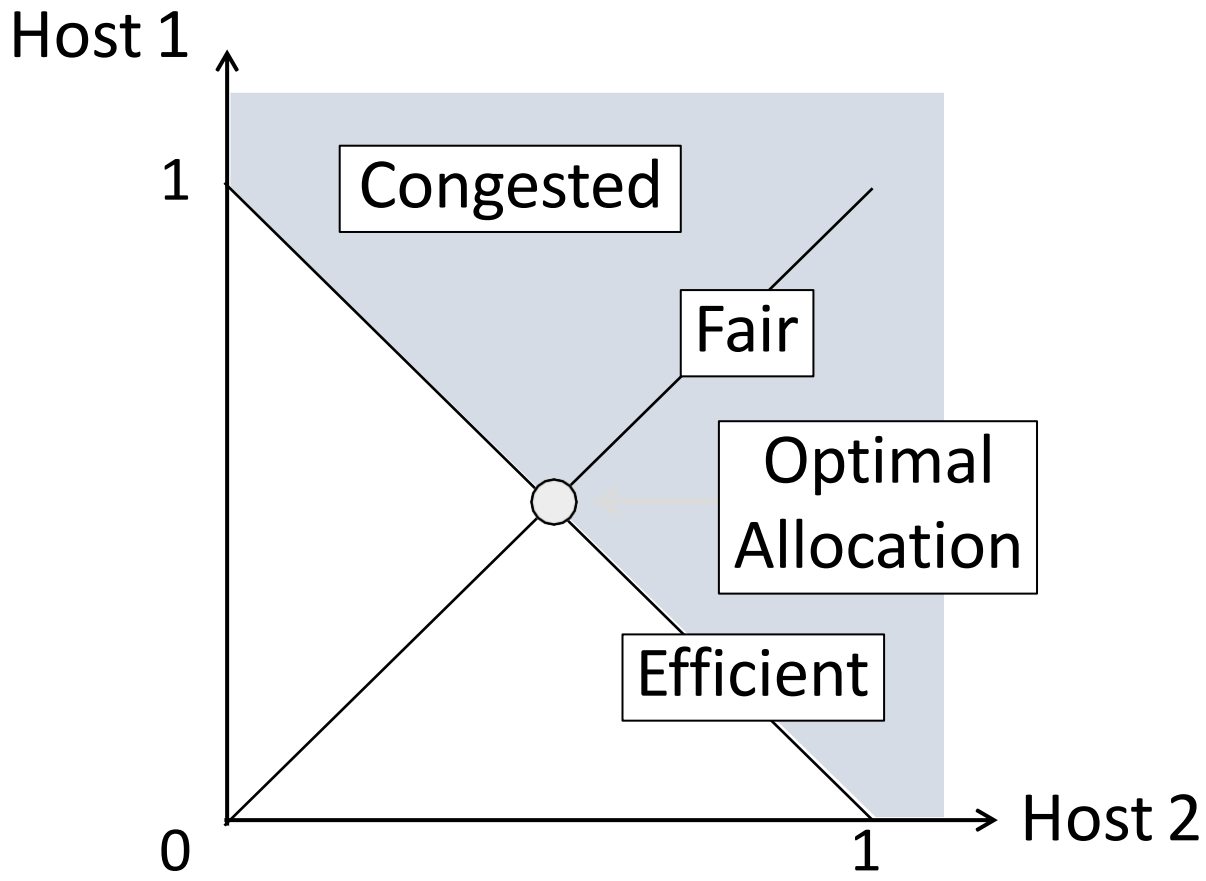
AIMD Game

- Hosts 1 and 2 share a bottleneck
  - But do not talk to each other directly
- Router provides binary feedback
  - Tells hosts if network is congested



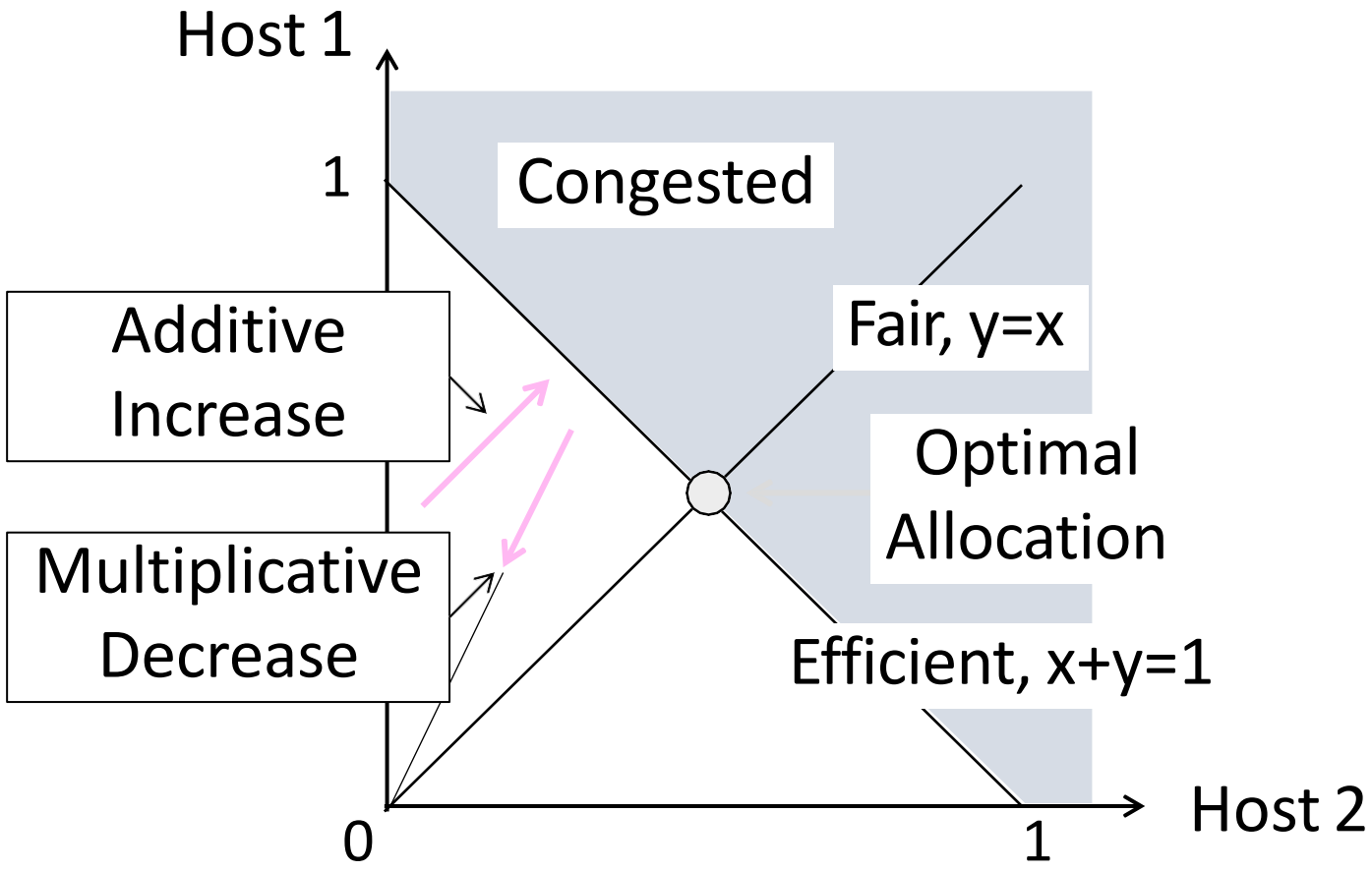
AIMD Game

- Each point is a possible allocation



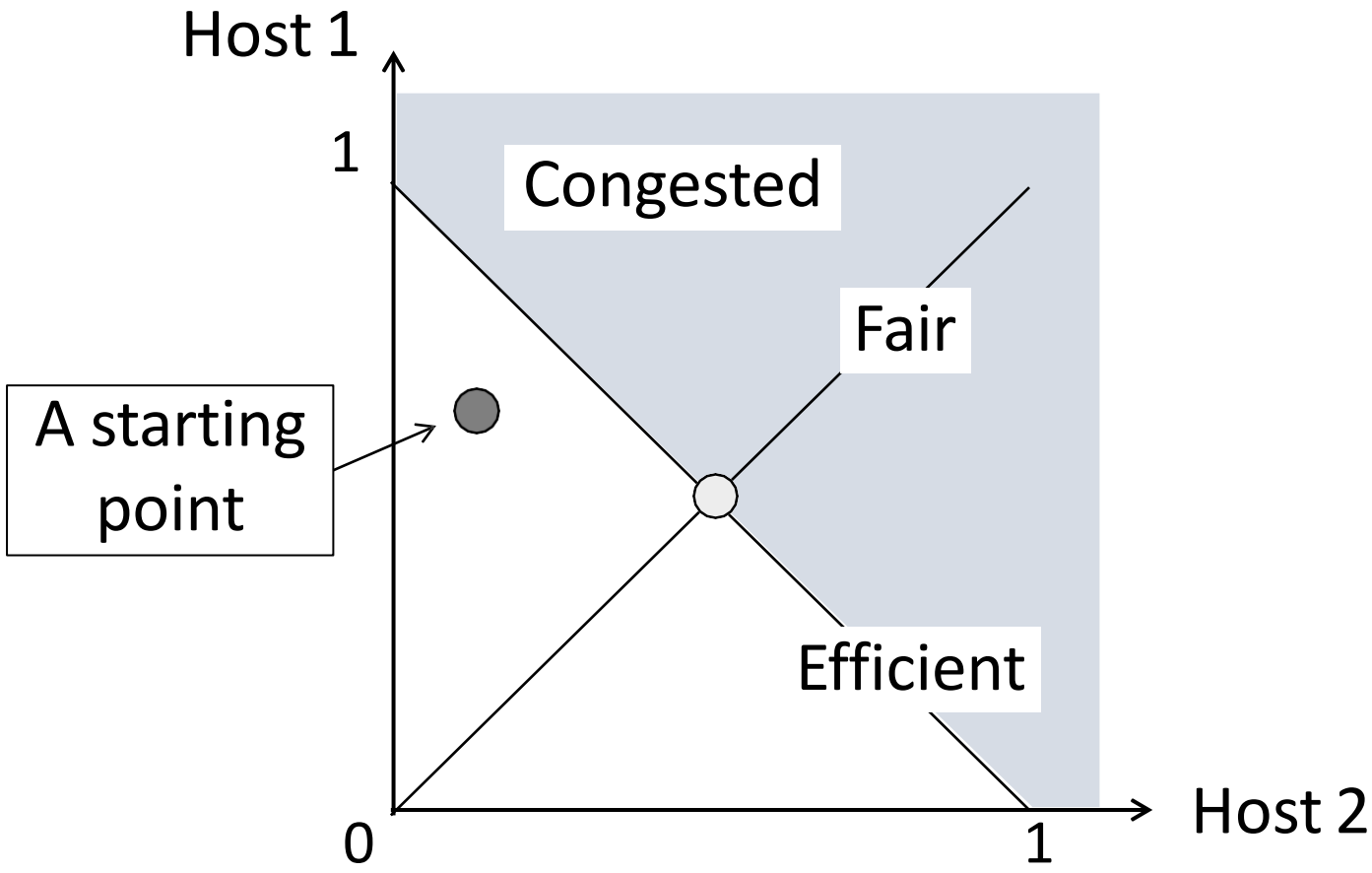
AIMD Game

- AI and MD move the allocation



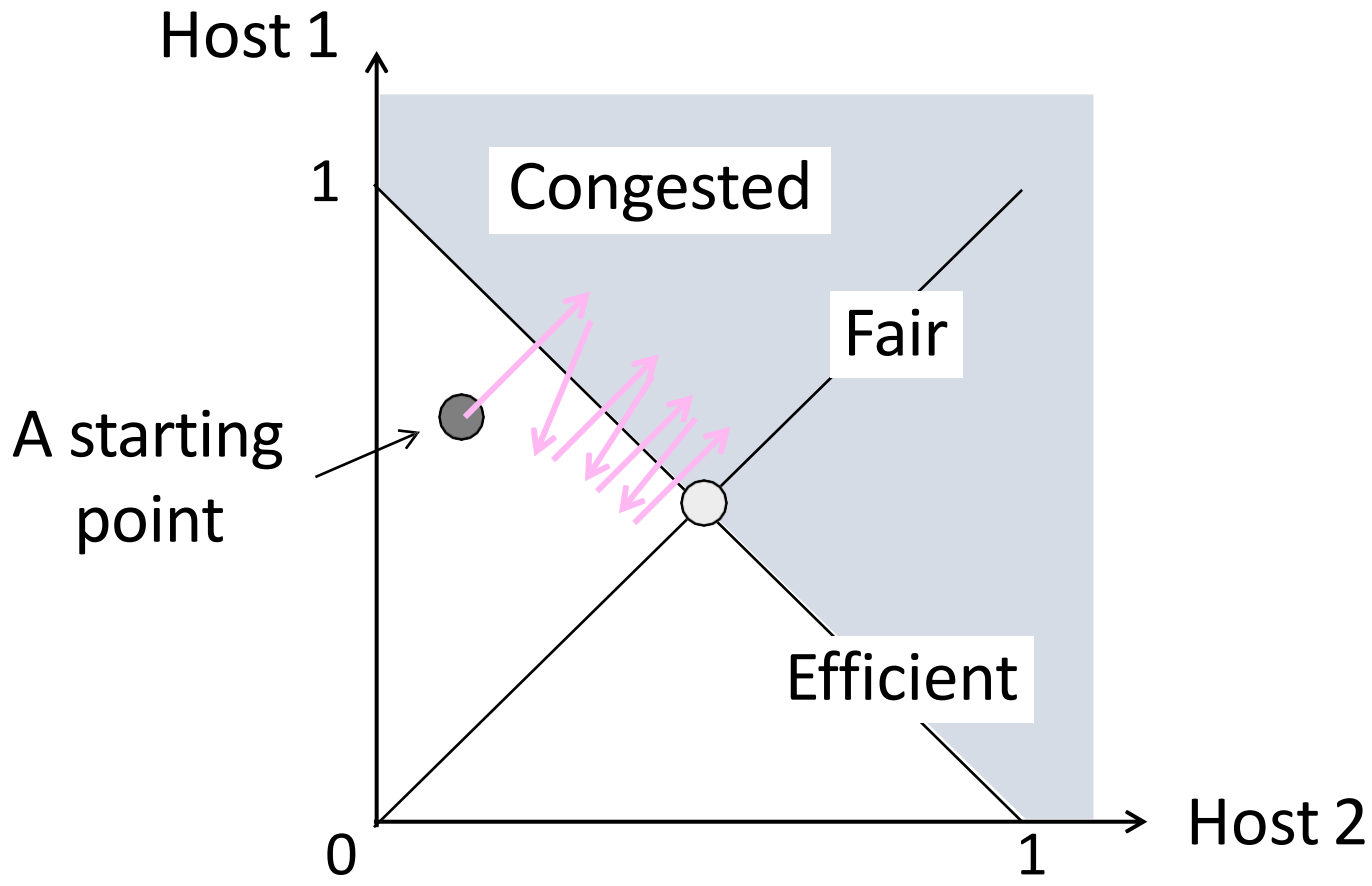
AIMD Game

- Play the game!



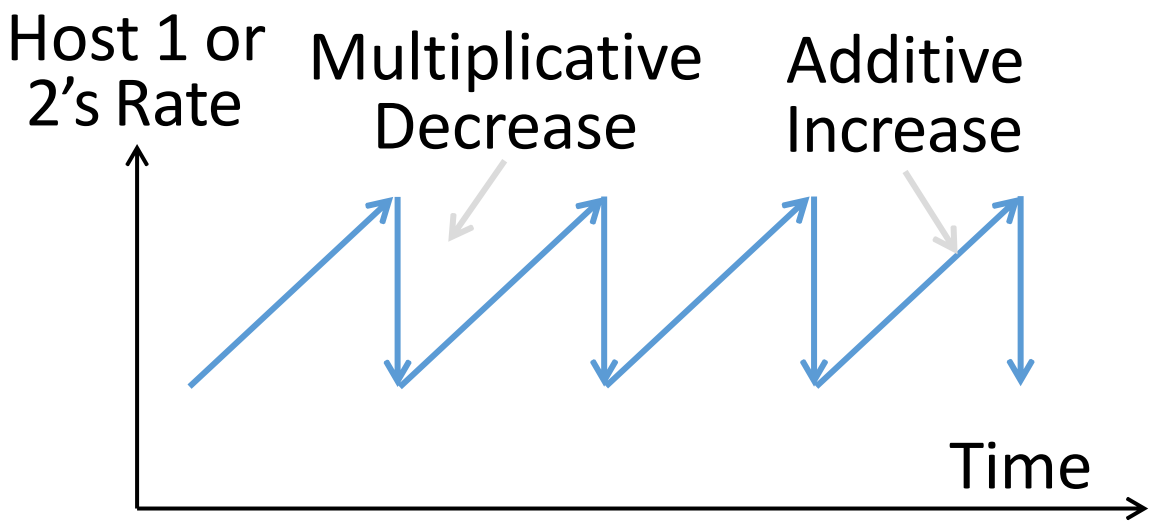
AIMD Game

- Always converge to good allocation!



AIMD sawtooth

- Produces a “sawtooth” pattern over time for rate of each host
  - This is the TCP sawtooth (later)



AIMD Properties

- Converges to an allocation that is efficient and fair when hosts run it
  - Holds for more general topologies
- Other increase/decrease control laws do not! (Try MIAD, MIMD, MIAD)
- Requires only binary feedback from the network

Feedback signals

- Several possible signals, with different pros/cons
- We'll look at classic TCP that uses packet loss as a signal

Signal	Example Protocol	Pros / Cons
Packet loss	TCP NewReno Cubic TCP (Linux)	Hard to get wrong Hear about congestion late Other events can cause loss
Packet delay	BBR (Google)	Hear about congestion early Need to infer congestion
Router indication	TCPs with Explicit Congestion Notification	Hear about congestion early Require router support

TCP congestion control overview

Sender uses congestion window (cwnd)

- Sending rate (≈cwnd/RTT)

Sender uses loss as network congestion signal

Follow AIMD control law for a good allocation

- Goal is efficient and (roughly) fair allocation

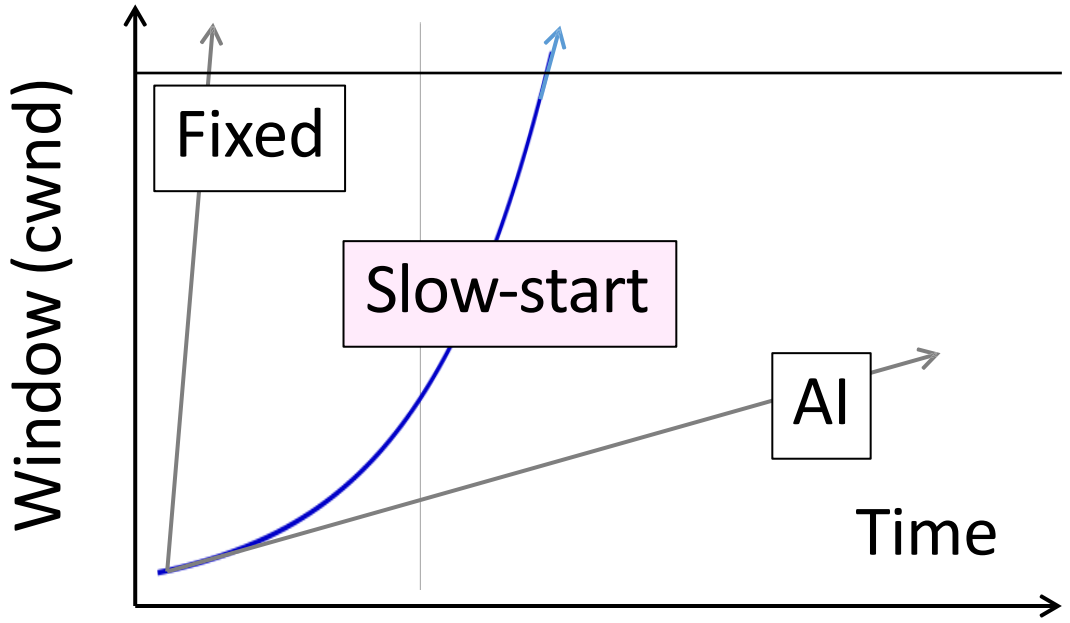
TCP “slow-start” problem

- We want to quickly get to the right cwnd but it varies
  - Fixed window can be too inefficient or too aggressive
  - Additive Increase adapts cwnd gently, but might take a long time to become efficient

Slow-start solution

Start by doubling cwnd every RTT

- Exponential growth (1, 2, 4, 8, 16, …)
- Start slow, quickly reach large values



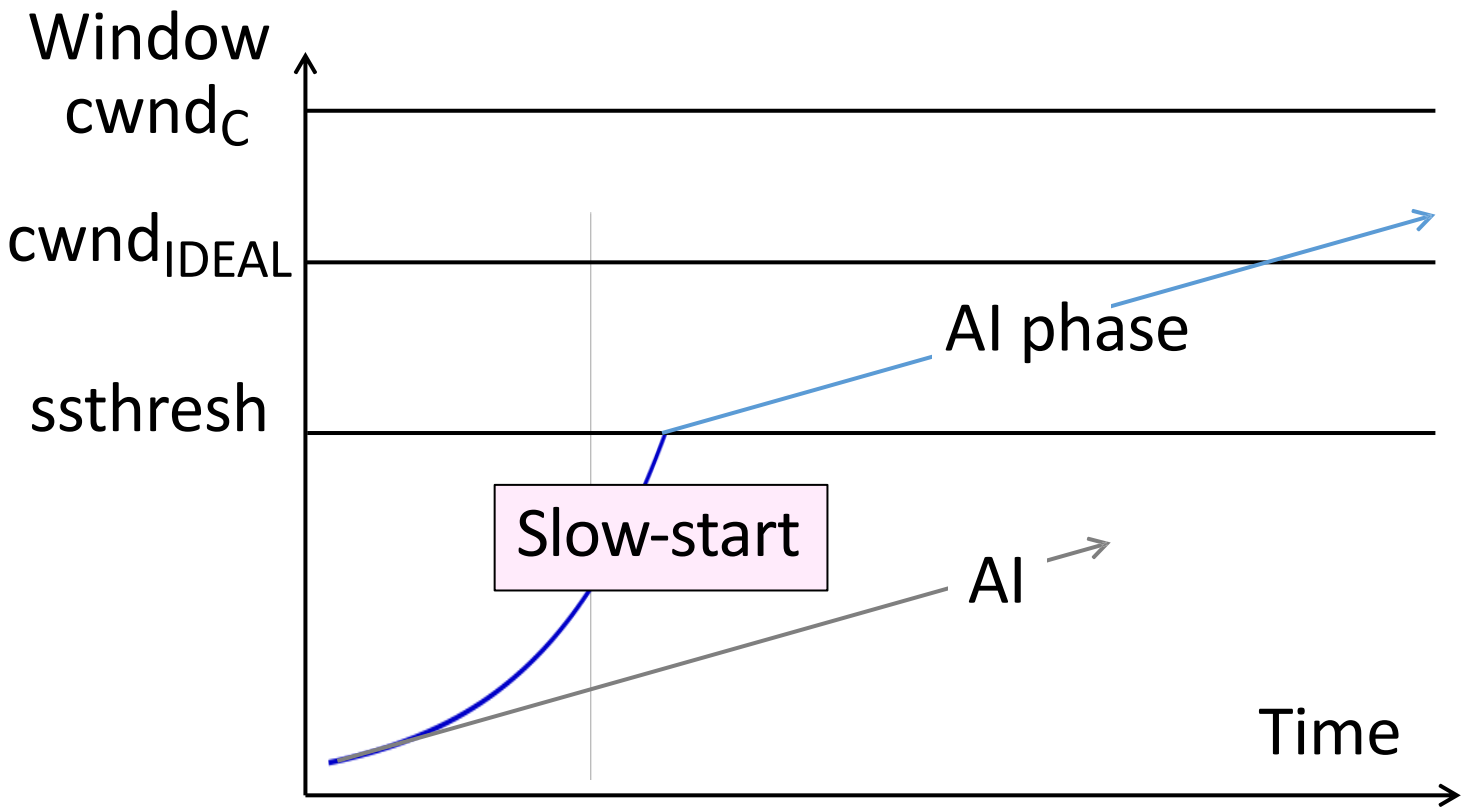
Slow-start solution

Eventually packet loss will occur when the network is congested

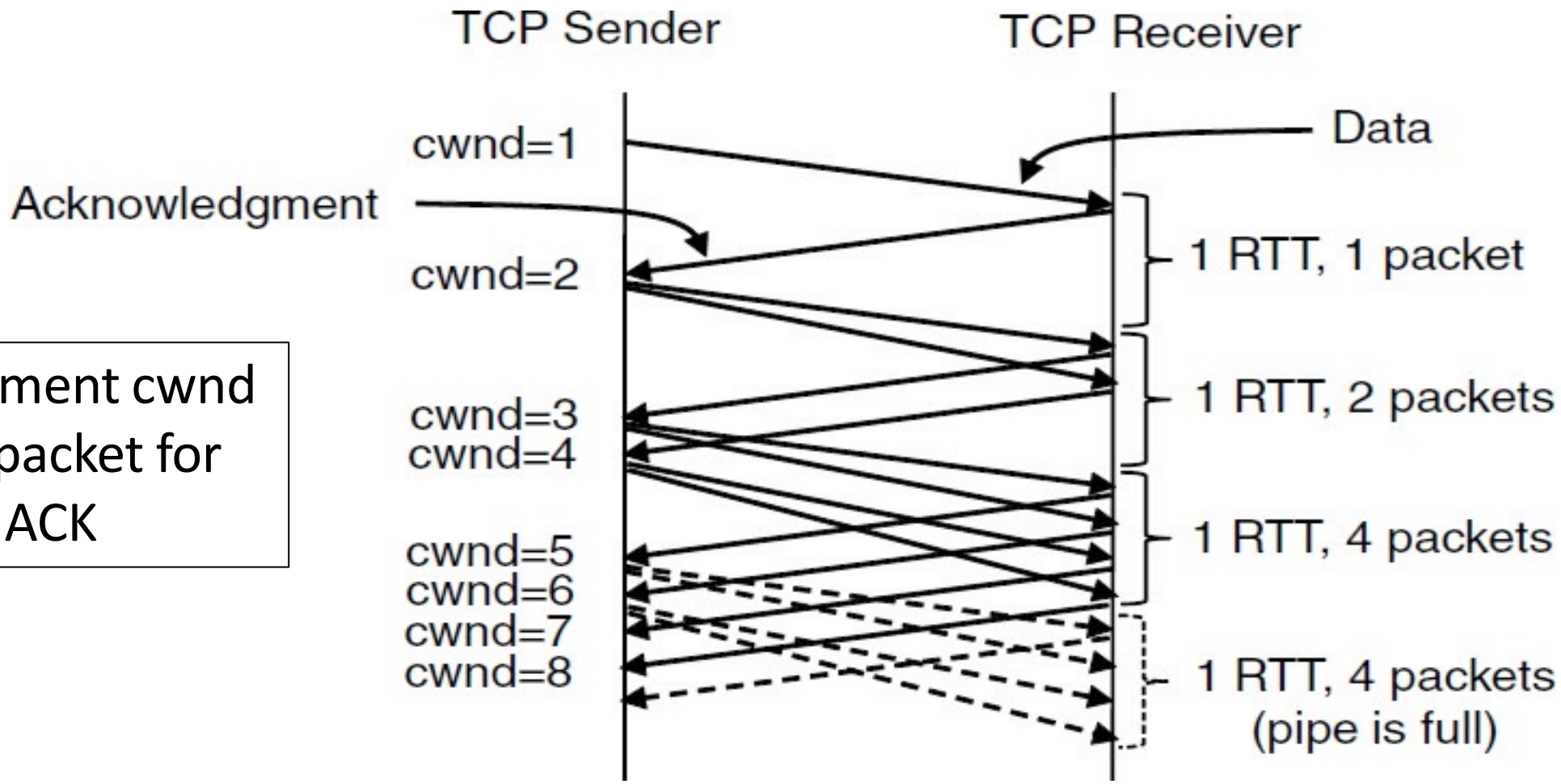
- Loss timeout tells us cwnd is too large
- Next time, switch to AI beforehand
- Slowly adapt cwnd near right value

Slow-start solution

- Combined behavior, after first time
  - Most time spent near right value

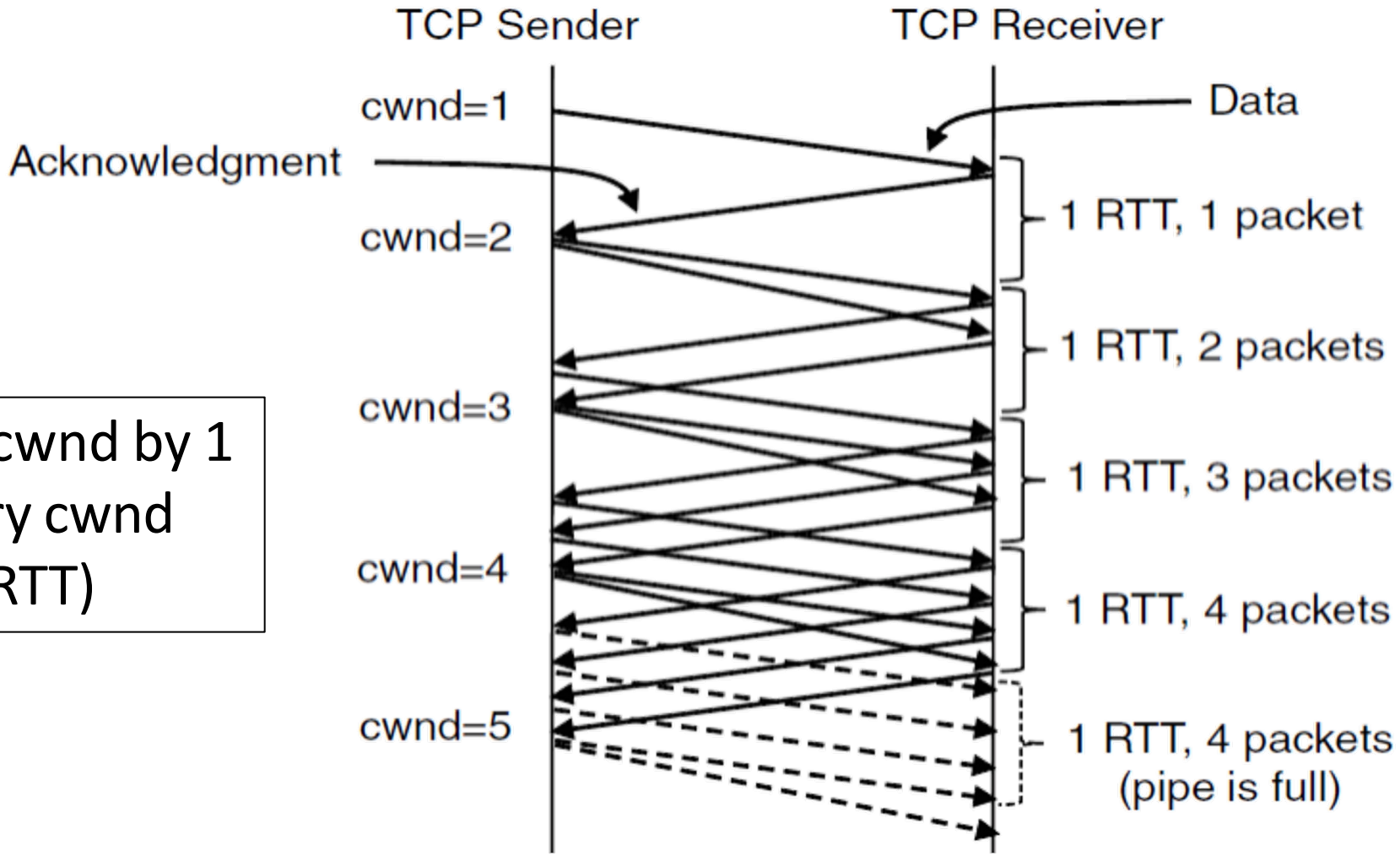


Slow-start (doubling) timeline



Increment cwnd by 1 packet for each ACK

Additive increase timeline



Increment cwnd by 1 packet every cwnd ACKs (or 1 RTT)

TCP Tahoe (Implementation)

Initial slow-start (doubling) phase

- Start with cwnd = 1 (or small value)
- cwnd += 1 packet per ACK

Later Additive Increase phase

- cwnd += 1/cwnd packets per ACK
- Roughly adds 1 packet per RTT

Switching threshold (initially infinity)

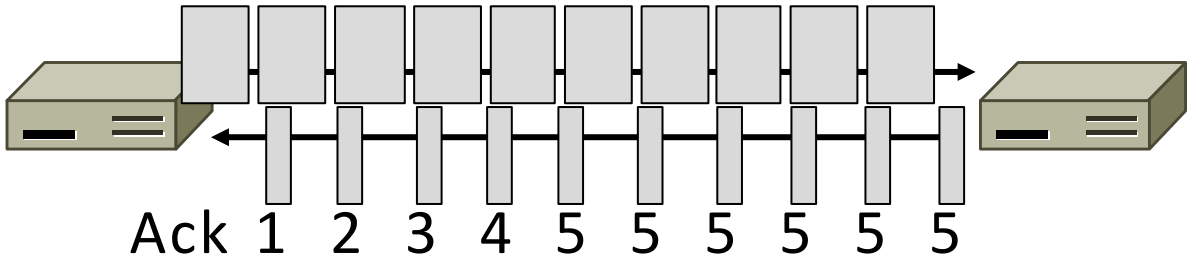
- Switch to AI when cwnd > ssthresh
- Set ssthresh = cwnd/2 after loss
- Begin with slow-start after timeout

Inferring loss from ACKs

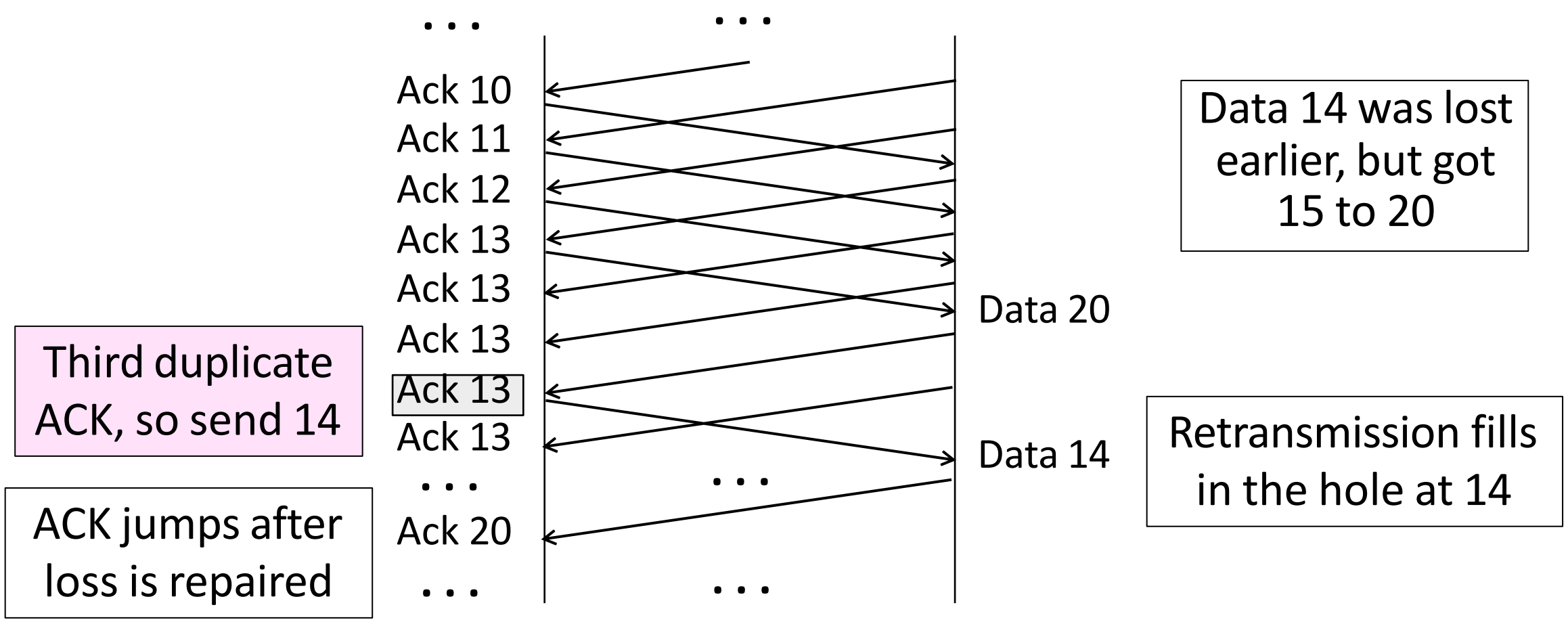
- TCP uses a cumulative ACK
  - Carries highest in-order seq. number
  - Normally a steady advance
- Duplicate ACKs give us hints about what data hasn't arrived
  - Tell us some new data did arrive, but it was not next segment
  - Thus the next segment may be lost

Fast retransmit

- Treat three duplicate ACKs as a loss
  - Retransmit next expected segment
  - Some repetition allows for reordering, but still detects loss quickly



Fast retransmit



Fast retransmit

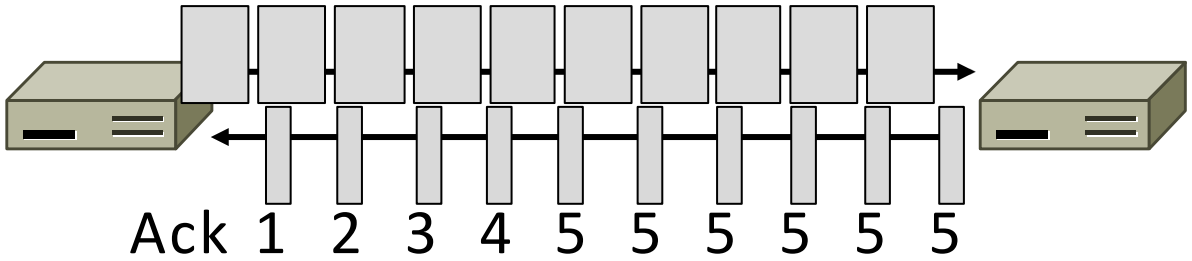
- It can repair single segment loss quickly, typically before a timeout
- However, we have quiet time at the sender/receiver while waiting for the ACK to jump
- And we still need to MD (Multiplicative Decrease) cwnd …

Inferring non-loss from ACKs

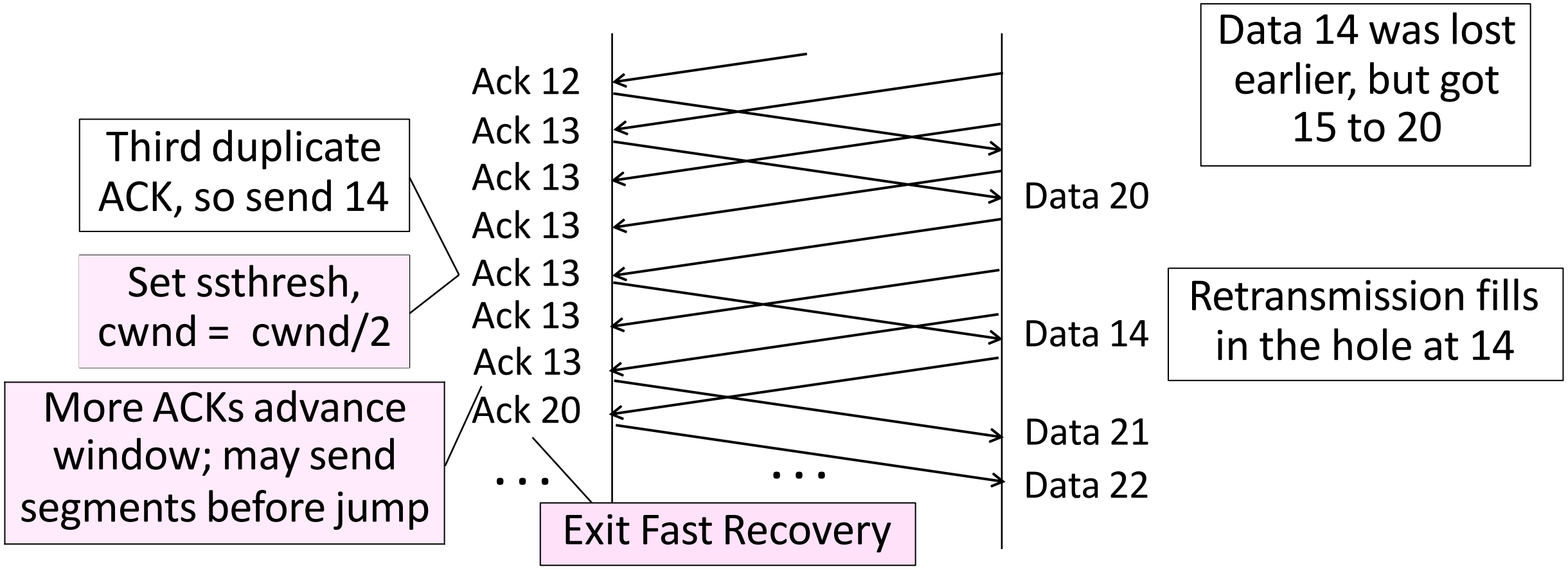
- Duplicate ACKs also give us hints about what data has arrived
  - Each new duplicate ACK means that some new segment has arrived
  - It will be the segments after the loss
  - Thus advancing the sliding window will not increase the number of segments stored in the network

Fast recovery (TCP Reno)

- First fast retransmit, and MD cwnd
- Then pretend further duplicate ACKs are the expected ACKs
  - Lets new segments be sent for ACKs
  - Reconcile views when the ACK jumps



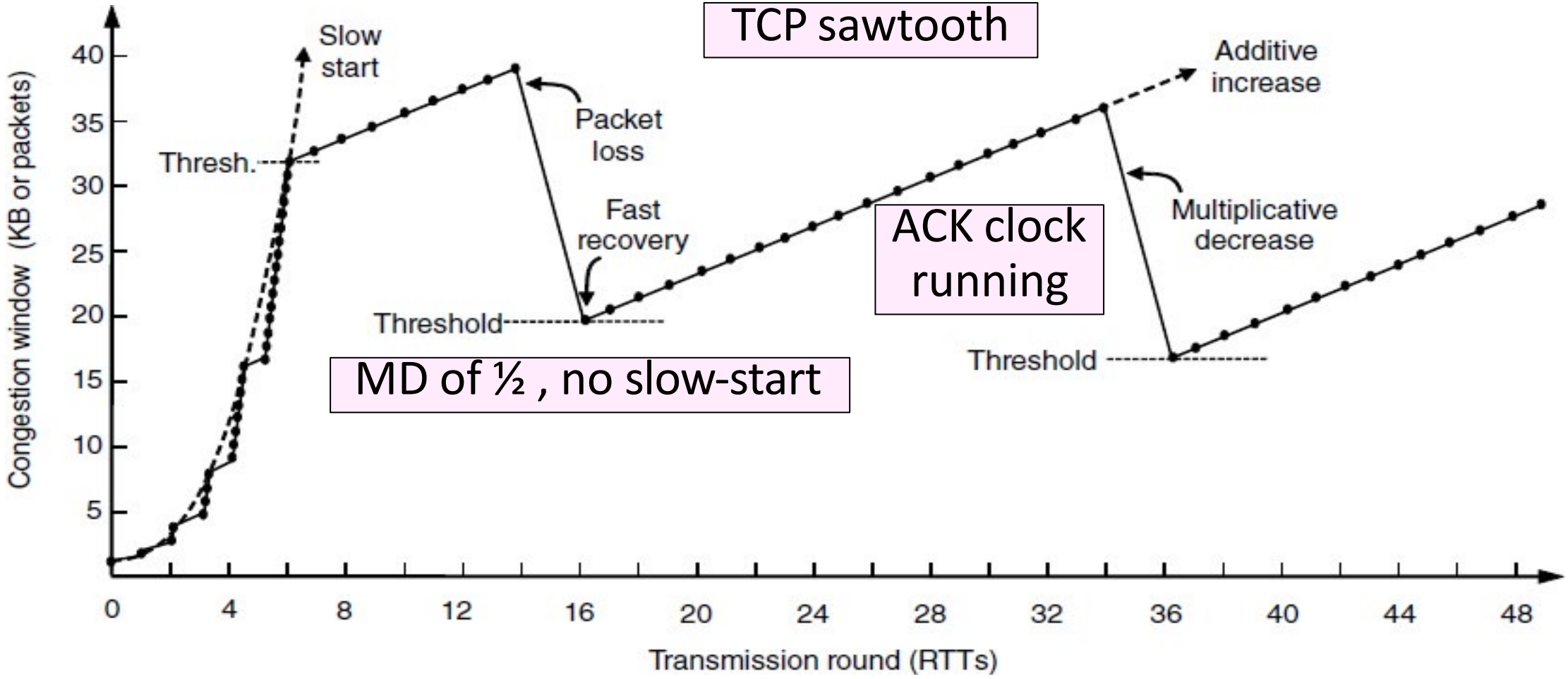
Fast recovery



Fast recovery

- With fast retransmit, it repairs a single segment loss quickly and keeps the ACK clock running
- This allows us to realize AIMD
  - No timeouts or slow-start after loss, just continue with a smaller cwnd
- TCP Reno combines slow-start, fast retransmit and fast recovery
  - Multiplicative Decrease is ½

TCP Reno


TCP sawtooth
Slow start
Thresh.
Packet loss
Fast recovery
Threshold
MD of ½ , no slow-start
Additive increase
ACK clock running
Multiplicative decrease
Threshold
Congestion window (KB or packets)
Transmission round (RTTs)
0
4
8
12
16
20
24
28
32
36
40
44
48
5
10
15
20
25
30
35
40

Recap: transport protocols

Goal: Provide end-to-end message delivery to applications

- Reliable or not; messages or streams

Challenges:

- Dealing with packet losses
- Dealing with slow receivers (flow control) and network (congestion control)
- Adapting to network conditions
  - Determine the right sending rate for yourself
  - Individual behaviors resulting in efficient and fair resource use

Toolbox

- Timeouts/retransmissions, sliding windows, max-min fairness, AIMD, ….

Credits

- Some slides are adapted from course slides of CSE 461 in UW