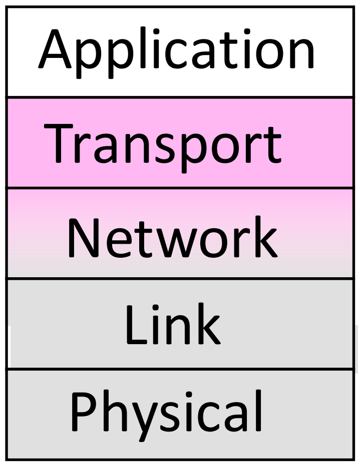
Operating Systems and Networks

Network Lecture 10: Congestion Control

Adrian Perrig
Network Security Group
ETH Zürich

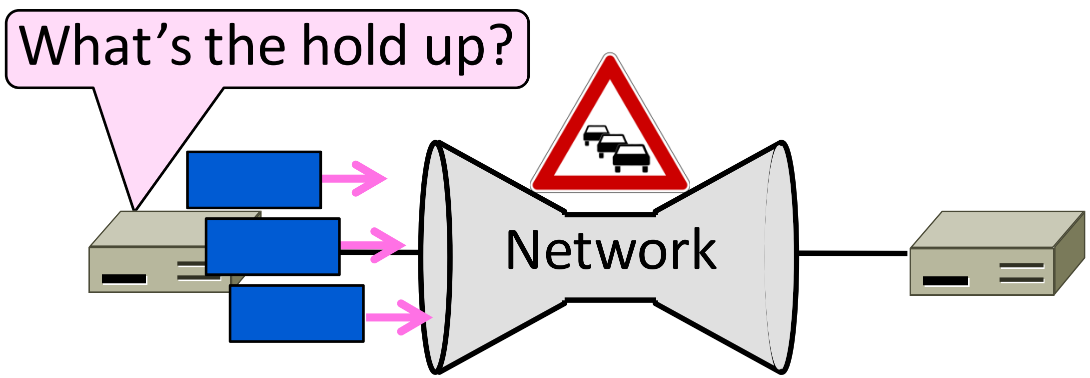
Where we are in the Course

- More fun in the Transport Layer!
  - The mystery of congestion control
  - Depends on the Network layer too



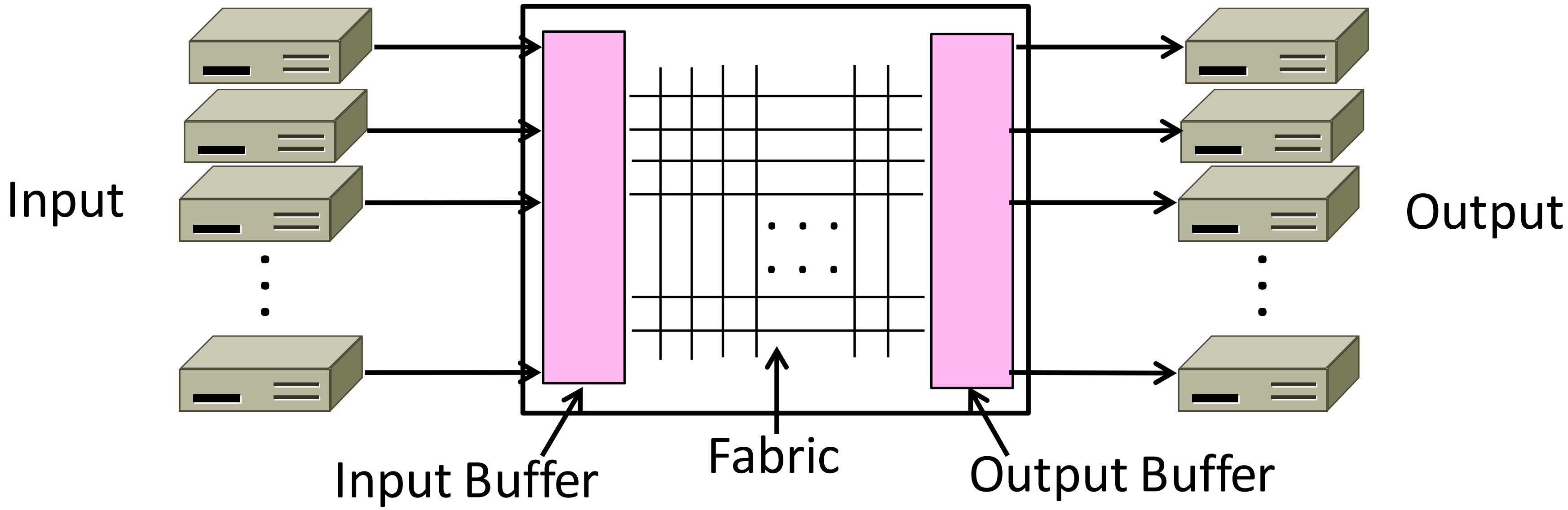
Topic

- Understanding congestion, a "traffic jam" in the network
  - Later we will learn how to control it



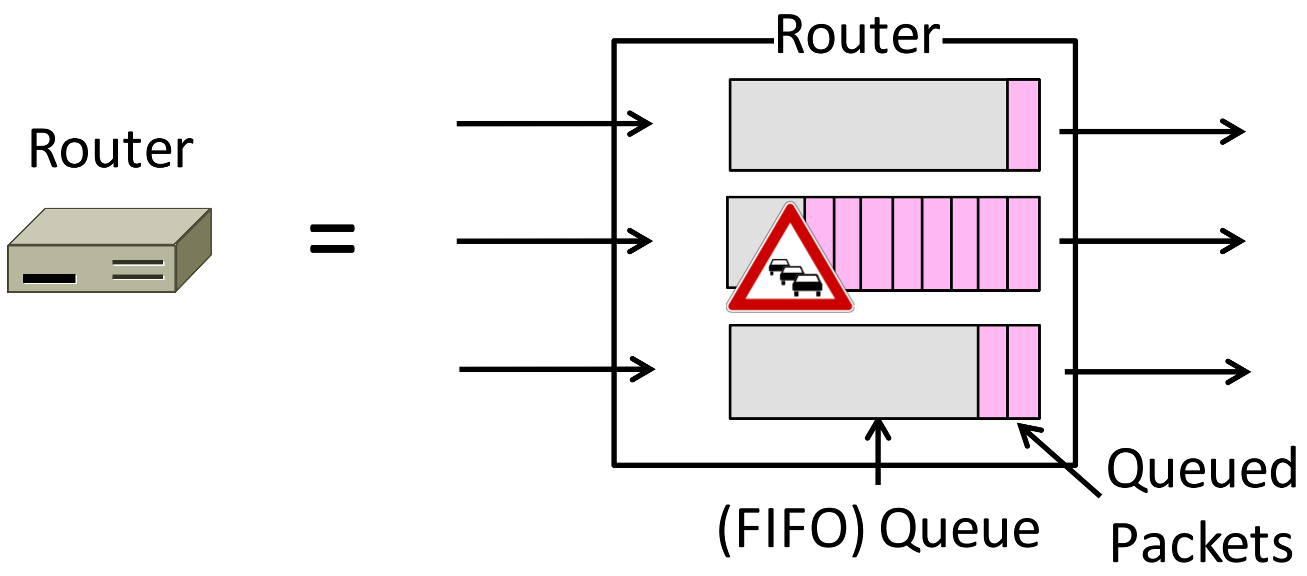
Nature of Congestion

- Routers/switches have internal buffering for contention



Nature of Congestion (2)

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

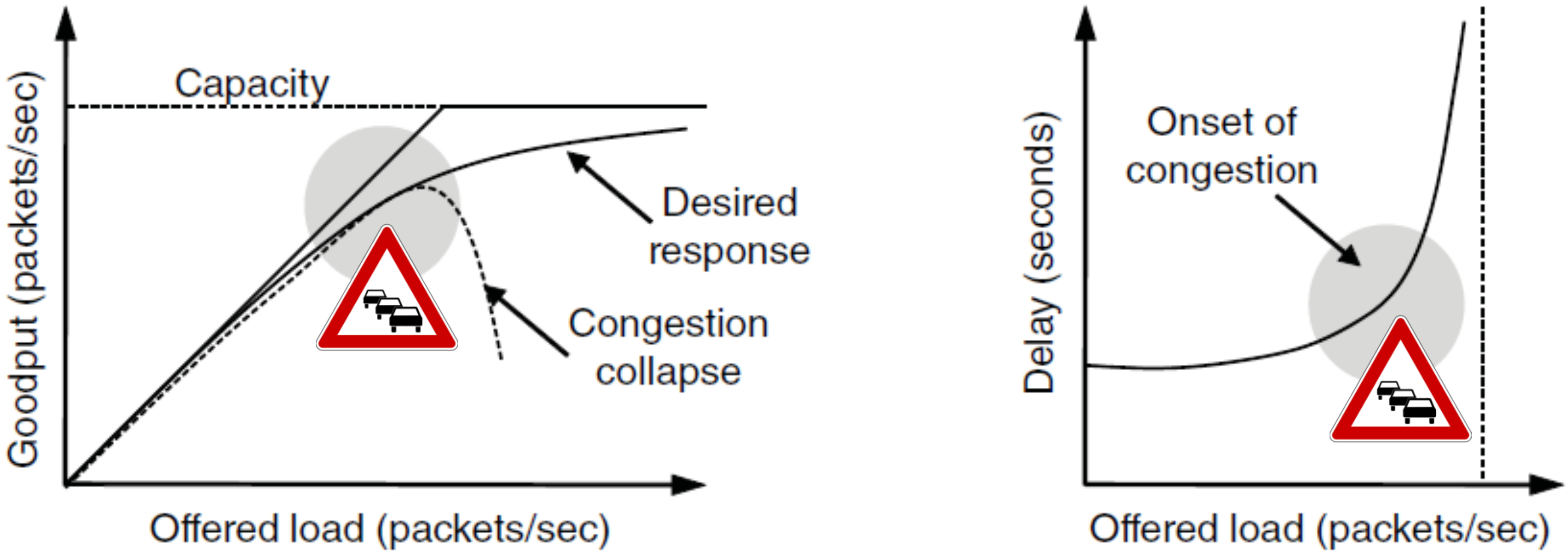


Nature of Congestion (3)

- Queues help by absorbing bursts when input > output rate
- But if input > output rate persistently, queue will overflow
  - This is congestion
- Congestion is a function of the traffic patterns – can occur even if every link have the same capacity

Effects of Congestion

- What happens to performance as we increase the load?



Effects of Congestion (3)

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

Bandwidth Allocation

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

Bandwidth Allocation (2)

- Key observation:
  - In an effective solution, Transport and Network layers must work together
- Network layer witnesses congestion
  - Only it can provide direct feedback
- Transport layer causes congestion
  - Only it can reduce offered load

Bandwidth Allocation (3)

- Why is it hard? (Just split equally!)
  - Number of senders and their offered load is constantly changing
  - Senders may lack capacity in different parts of the network
  - Network is distributed; no single party has an overall picture of its state

Bandwidth Allocation (4)

- Solution context:
  - Senders adapt concurrently based on their own view of the network
  - Design this adaption so the network usage as a whole is efficient and fair
  - Adaption is continuous since offered loads continue to change over time

Topics

- Nature of congestion
- Fair allocations
- AIMD control law
- TCP Congestion Control history
- ACK clocking
- TCP Slow-start
- TCP Fast Retransmit/Recovery
- Congestion Avoidance (ECN)

Fairness of Bandwidth Allocation (§6.3.1)

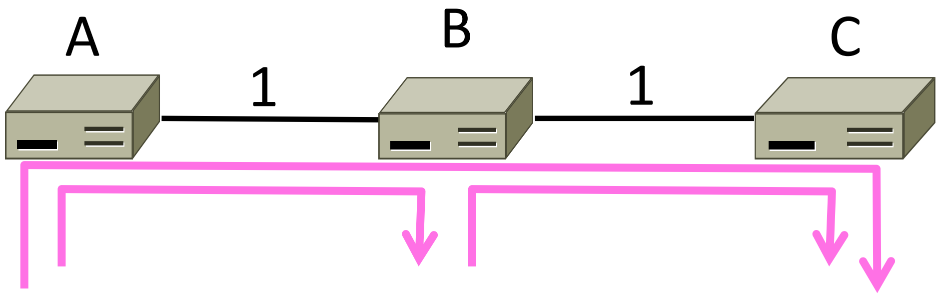
- What's a "fair" bandwidth allocation?
  - The max-min fair allocation

Recall

- We want a good bandwidth allocation to be fair and efficient
  - Now we learn what fair means
- Caveat: in practice, efficiency is more important than fairness

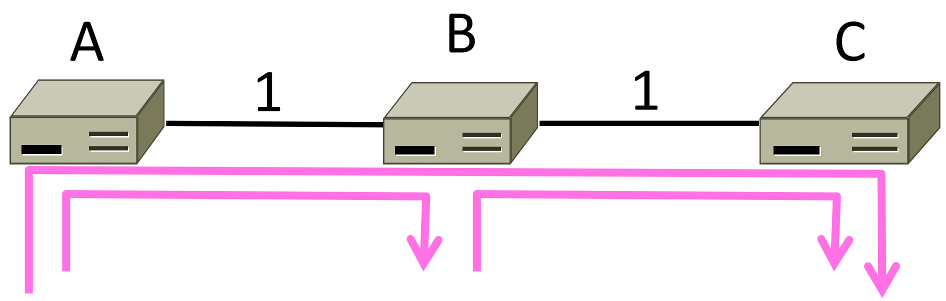
Efficiency vs. Fairness

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



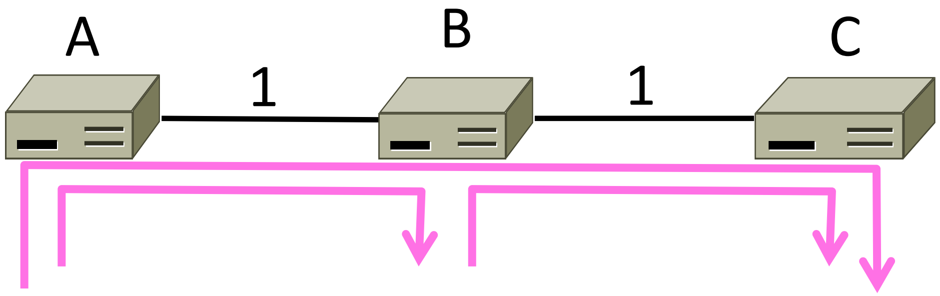
Efficiency vs. Fairness (2)

- 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 (3)

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

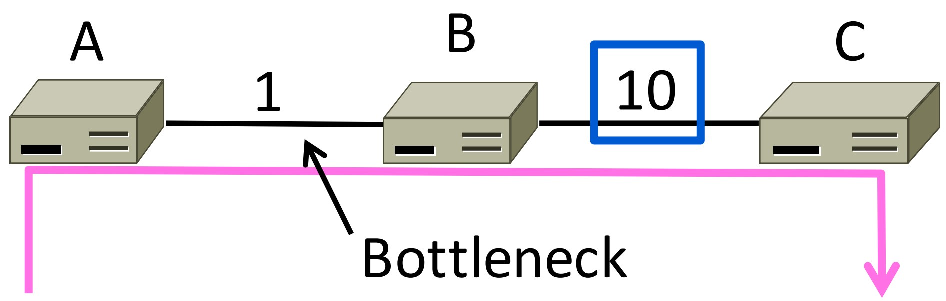


The Slippery Notion of Fairness

- Why is "equal per flow" fair anyway?
  - A→C uses more network resources (two links) 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
  - "Equal per flow" is good enough

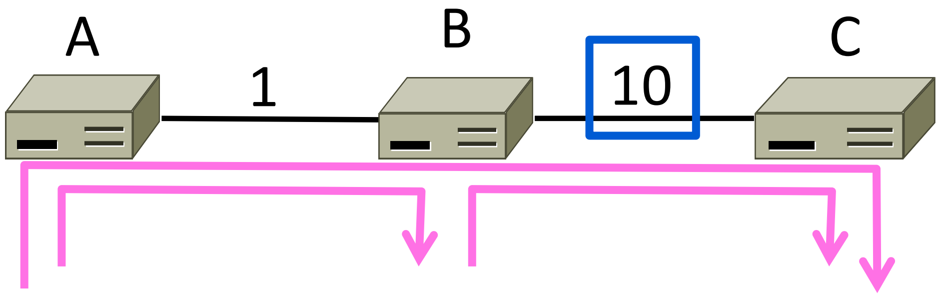
Generalizing "Equal per Flow"

- <u>Bottleneck</u> 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" (2)

- 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

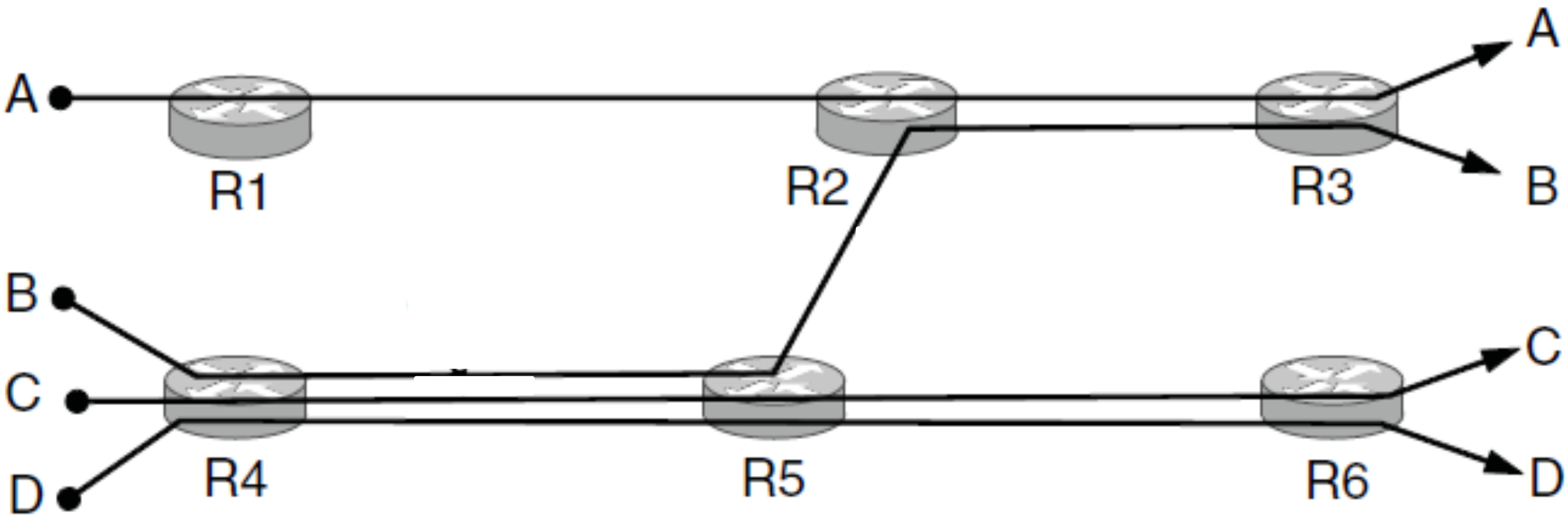
- Intuitively, flows bottlenecked on a link get an equal share of that link
- Max-min fair allocation is one that:
  - Increasing the rate of one flow will decrease the rate of a smaller flow
  - This "maximizes the minimum" flow

Max-Min Fairness (2)

- 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

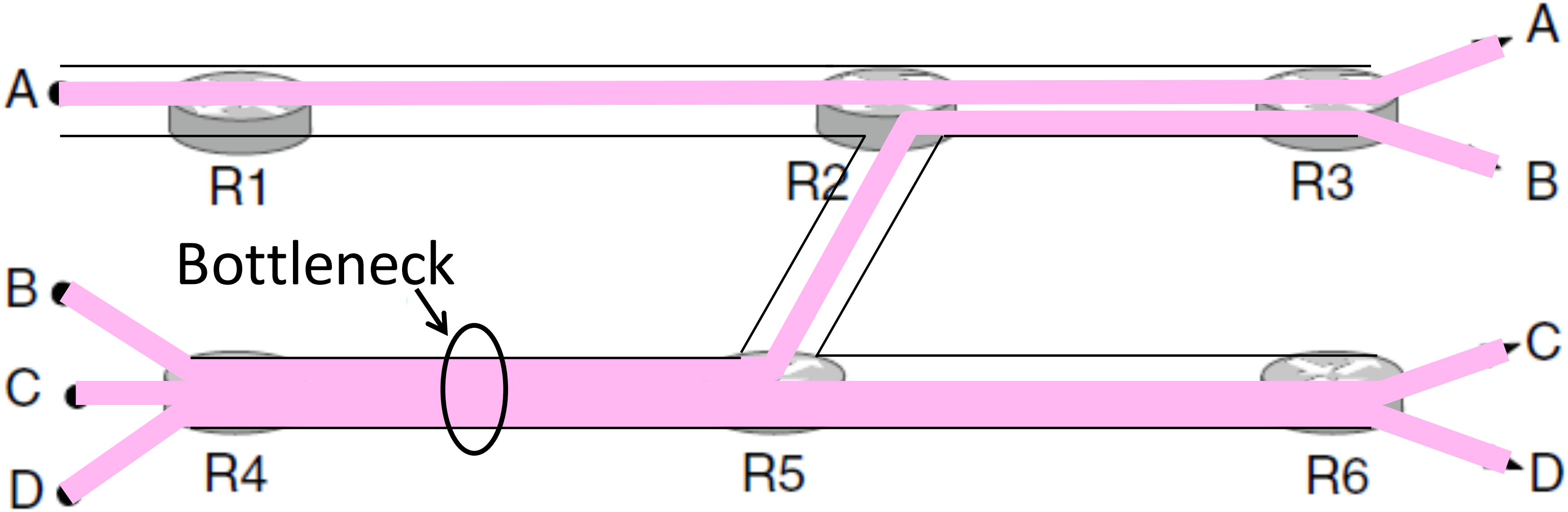
Max-Min Example

- Example: network with 4 flows, links equal bandwidth
  – What is the max-min fair allocation?



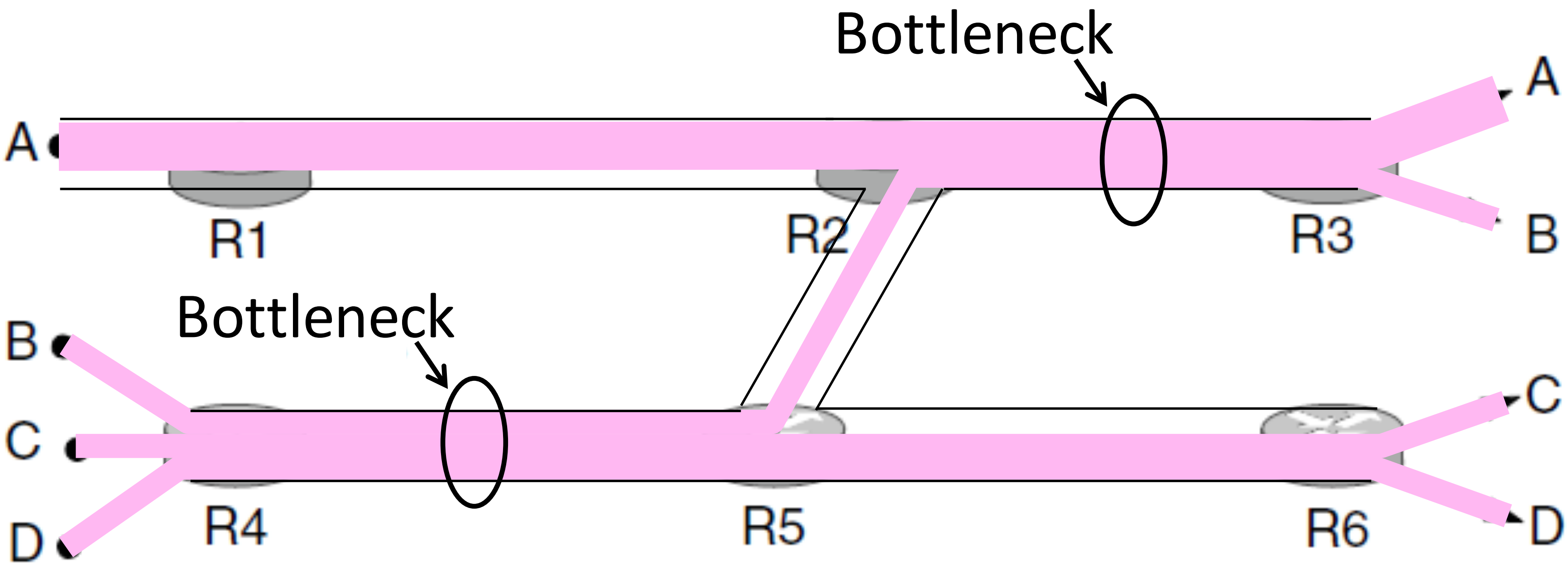
Max-Min Example (2)

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

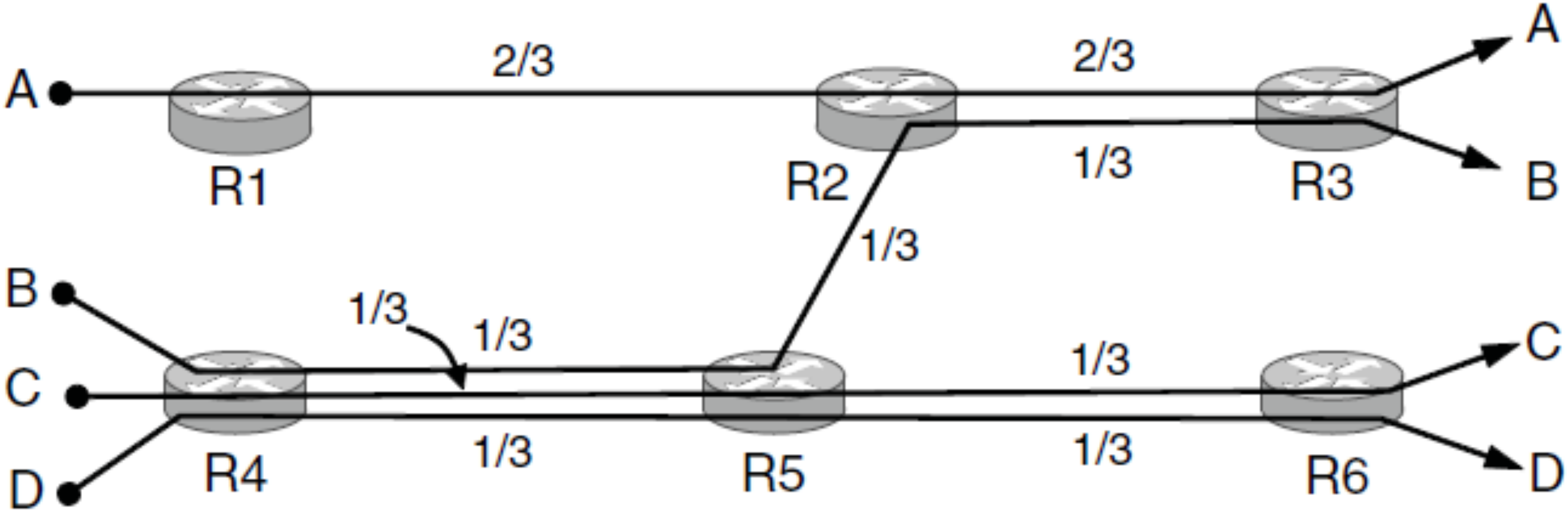


Max-Min Example (3)

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

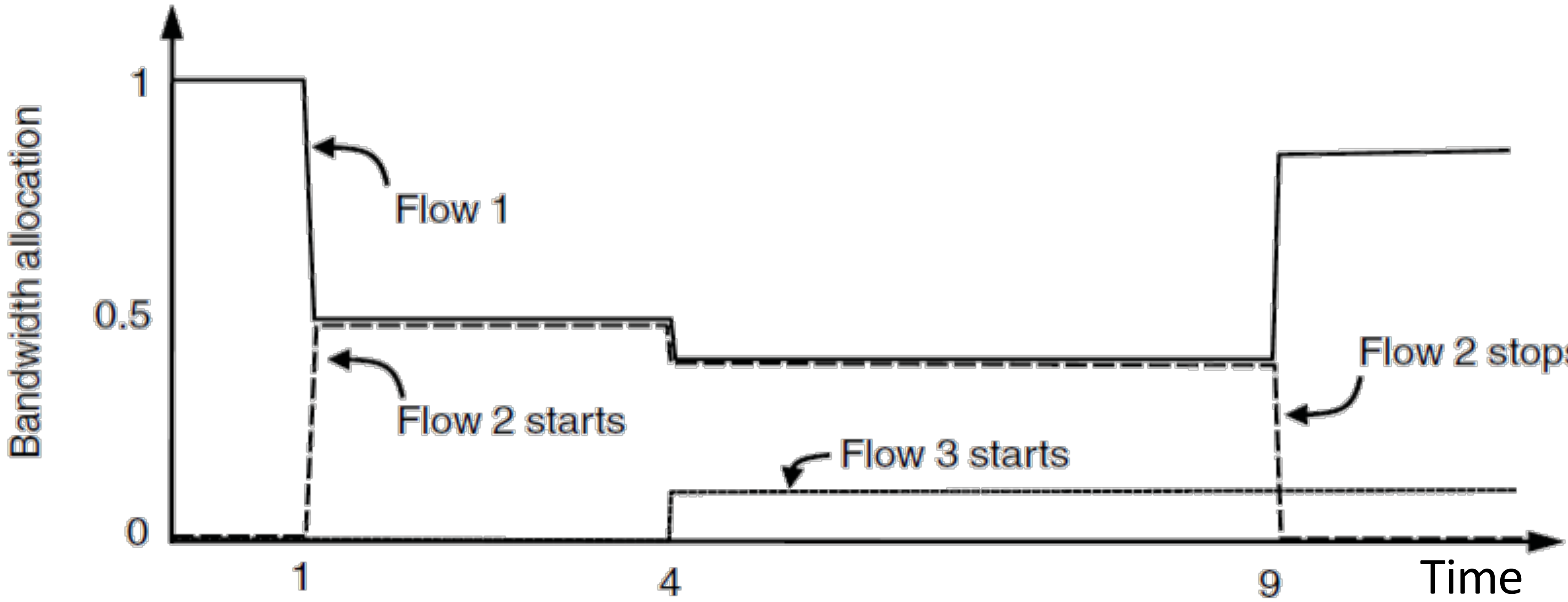


Max-Min Example (4)

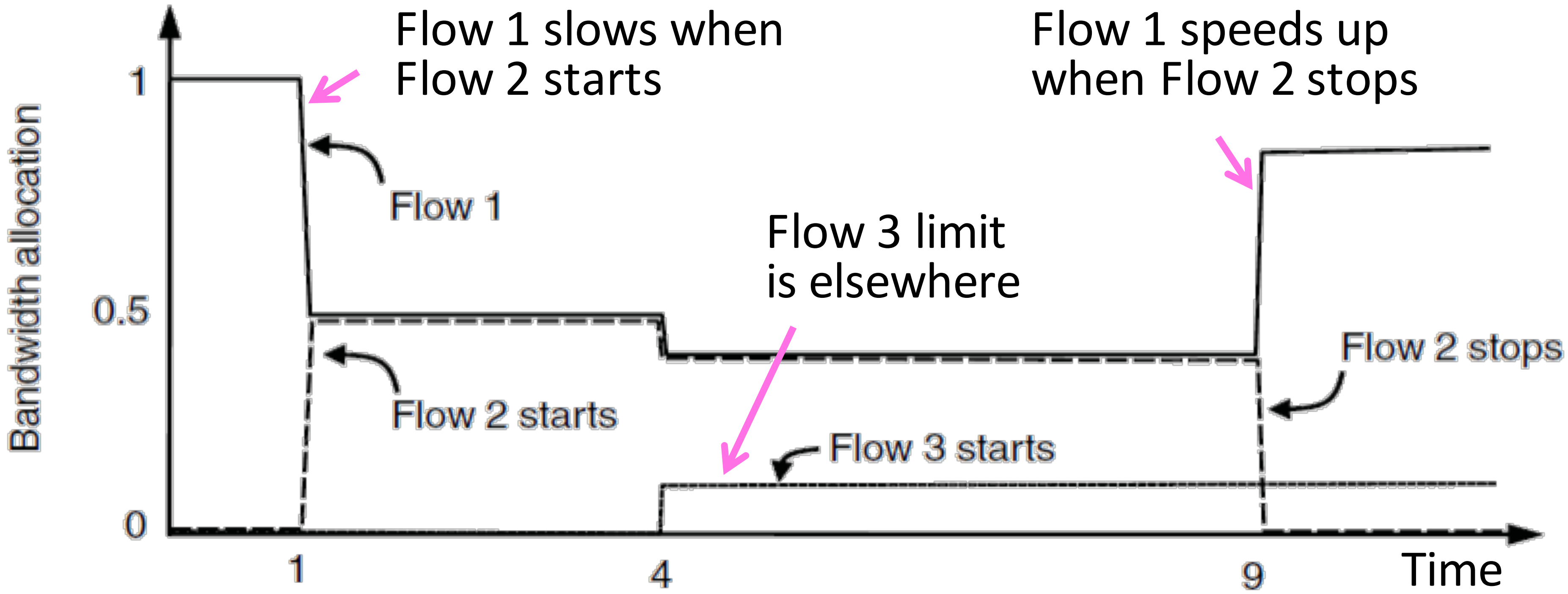
- End with A=2/3, B, C, D=1/3, and R2—R3, R4—R5 full
  - Other links have extra capacity that can't be used
- 

Adapting over Time

- Allocation changes as flows start and stop



Adapting over Time (2)



Recall

- Want to allocate capacity to senders
  - Network layer provides feedback
  - Transport layer adjusts offered load
  - A good allocation is efficient and fair

- How should we perform the allocation?
  - Several different possibilities …

Bandwidth Allocation Models

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

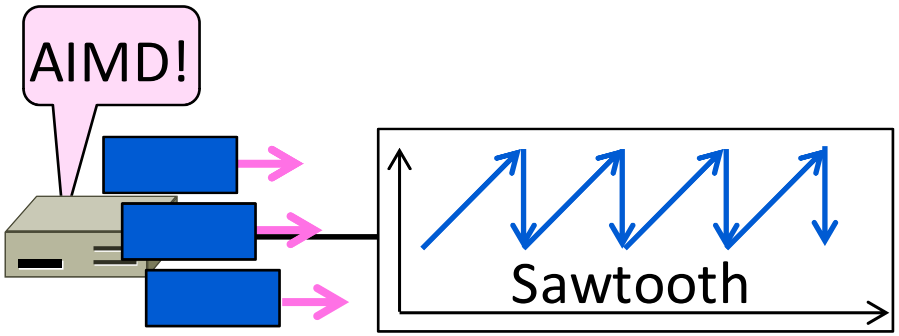
TCP is a closed loop, host-driven, and window-based

Bandwidth Allocation Models (2)

- We'll look at closed-loop, host-driven, and window-based
- Network layer returns feedback on current allocation to senders
  - At least tells if there is congestion
- Transport layer adjusts sender's behavior via window in response
  - How senders adapt is a control law

Additive Increase Multiplicative Decrease (AIMD) (§6.3.2)

- Bandwidth allocation models
  - Additive Increase Multiplicative Decrease (AIMD) control law

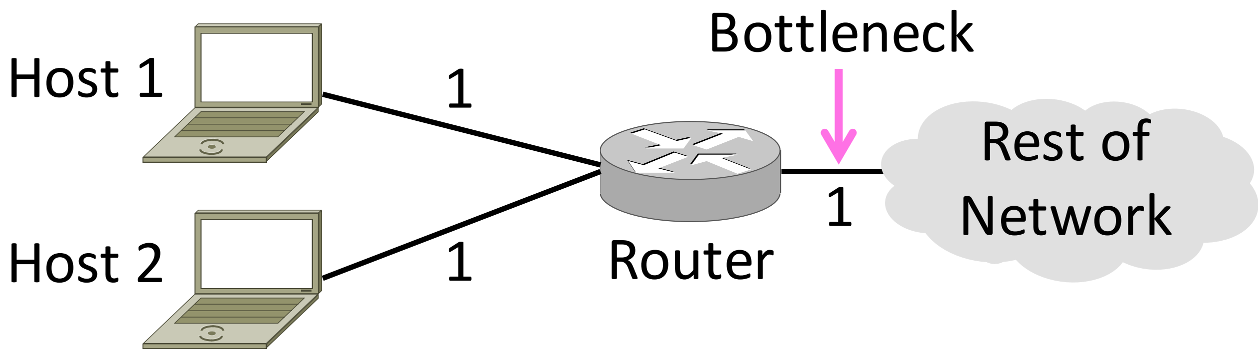


Additive Increase Multiplicative Decrease

- AIMD is a control law hosts can use to reach a good allocation
  - Hosts additively increase rate while network is not congested
  - Hosts multiplicatively decrease rate when congestion occurs
  - Used by TCP ☺
- Let's explore the AIMD game …

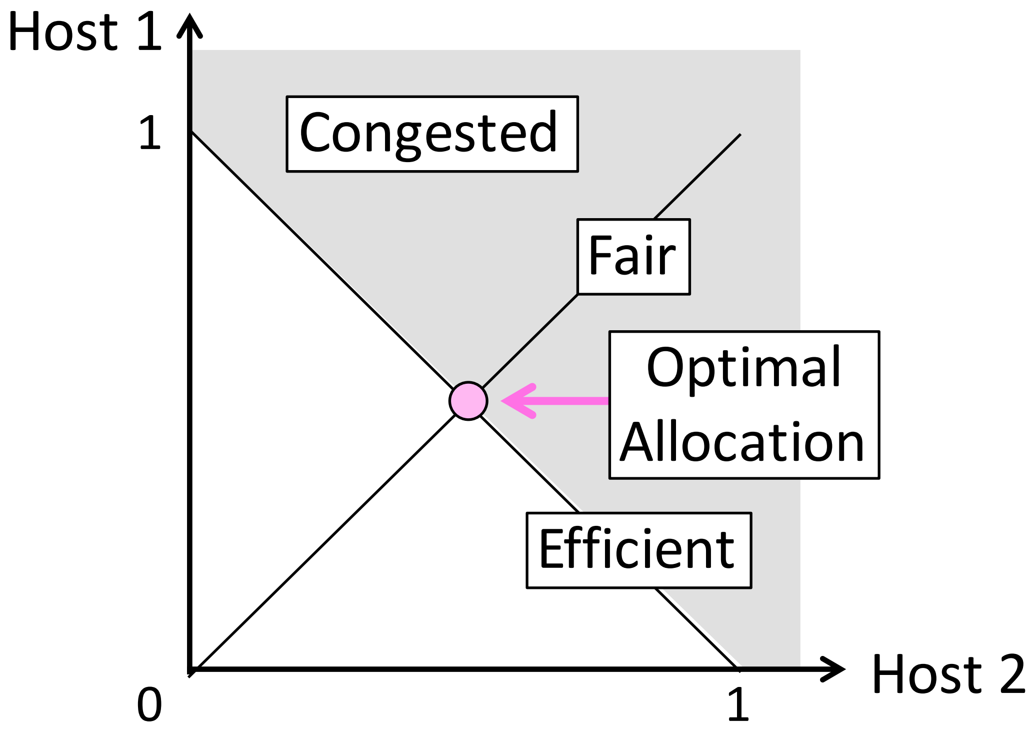
AIMD Game

- 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



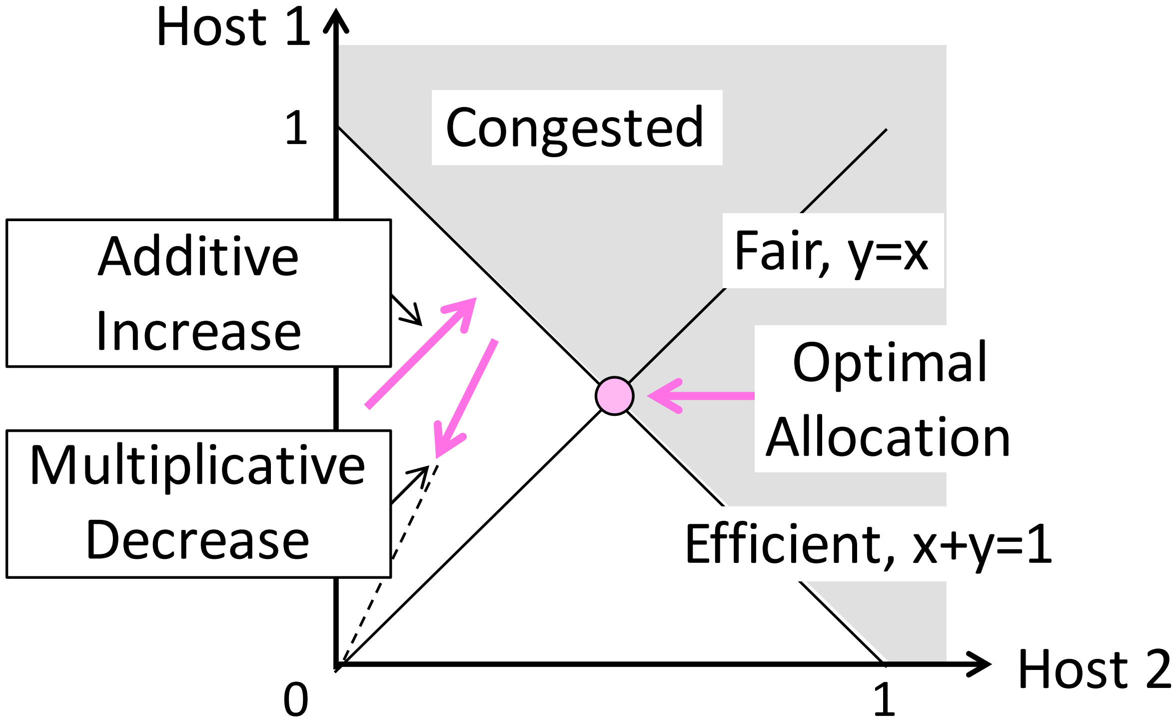
AIMD Game (2)

- Each point is a possible allocation



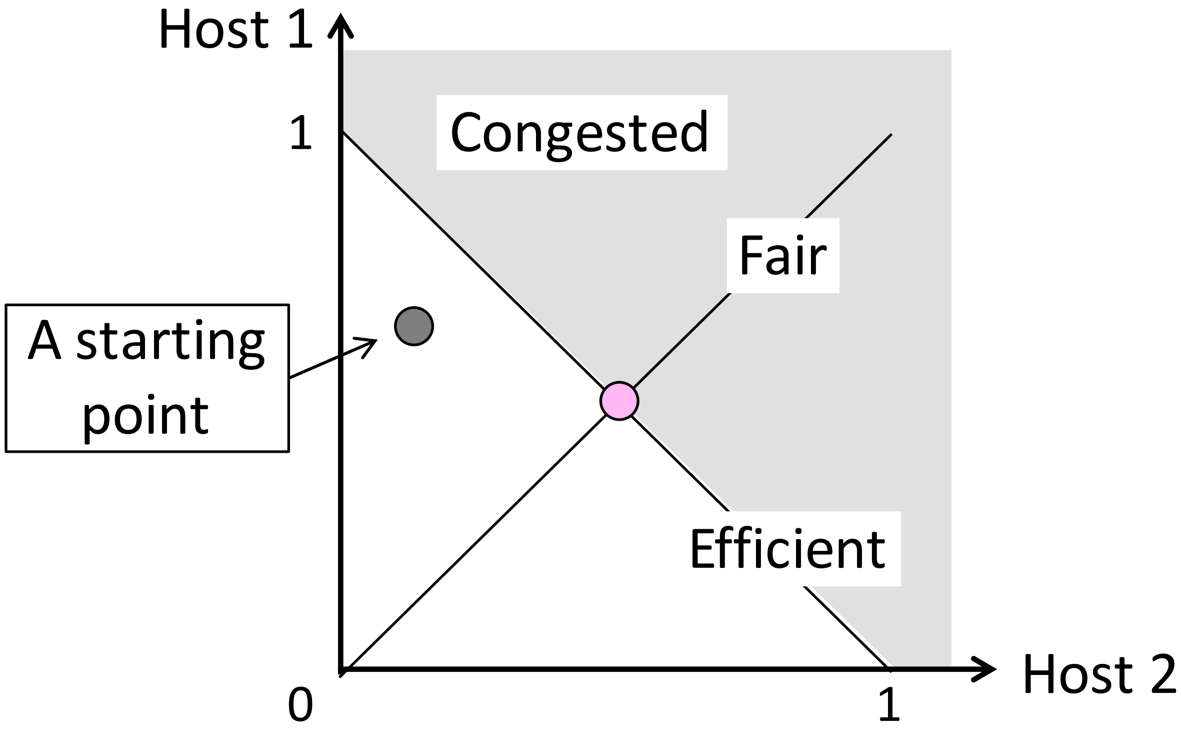
AIMD Game (3)

- AI and MD move the allocation



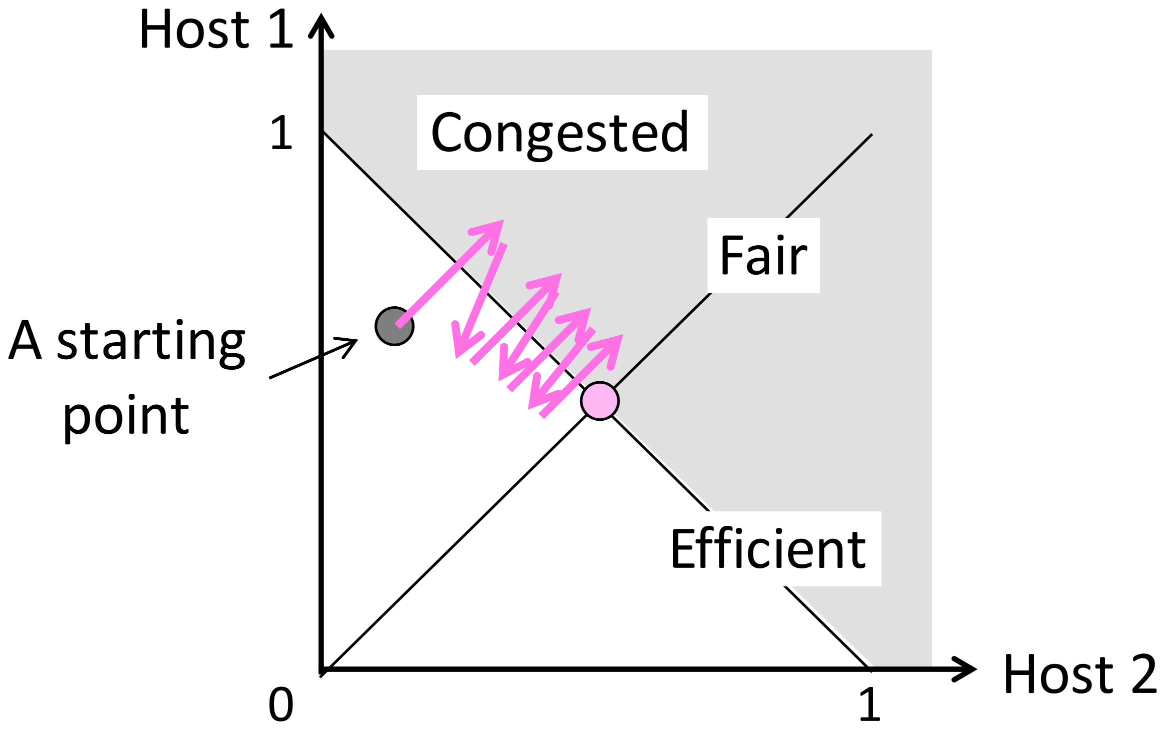
AIMD Game (4)

- Play the game!



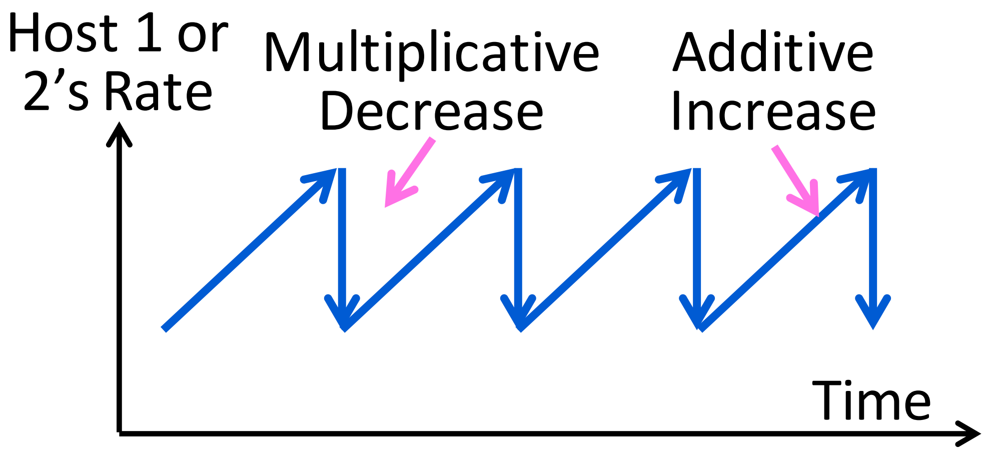
AIMD Game (5)

- 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, AIAD)
- Requires only binary feedback from the network

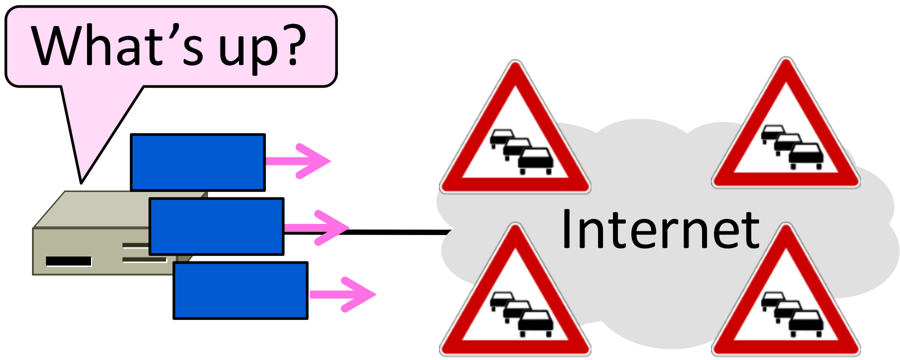
Feedback Signals

- 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
Packet delay	Compound TCP (Windows)	+Hear about congestion early -Need to infer congestion
Router indication	TCPs with Explicit Congestion Notification	+Hear about congestion early -Require router support

History of TCP Congestion Control (§6.5.10)

- The story of TCP congestion control
  - Collapse, control, and diversification

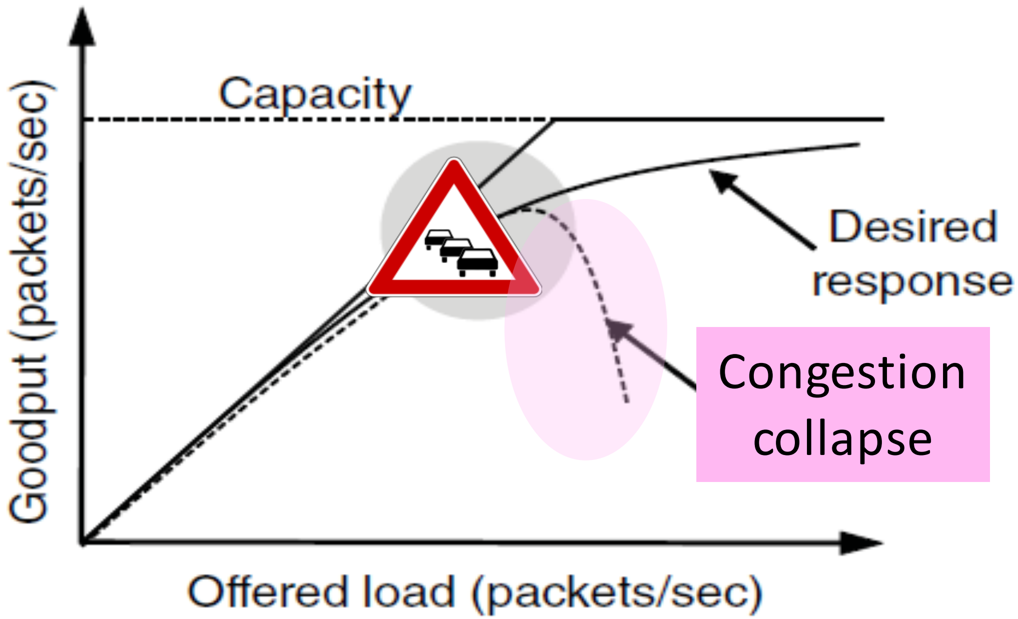


Congestion Collapse in the 1980s

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

Congestion Collapse (2)

- Queues became full, retransmissions clogged the network, and <u>goodput</u> fell



Van Jacobson (1950—)

- Widely credited with saving the Internet from congestion collapse in the late 80s
  - Introduced congestion control principles
  - Practical solutions (TCP Tahoe/Reno)
- Much other pioneering work:
  - Tools like traceroute, tcpdump, pathchar
  - IP header compression, multicast tools

Source: Wikipedia (public domain)

TCP Tahoe/Reno

- Avoid congestion collapse without changing routers (or even receivers)

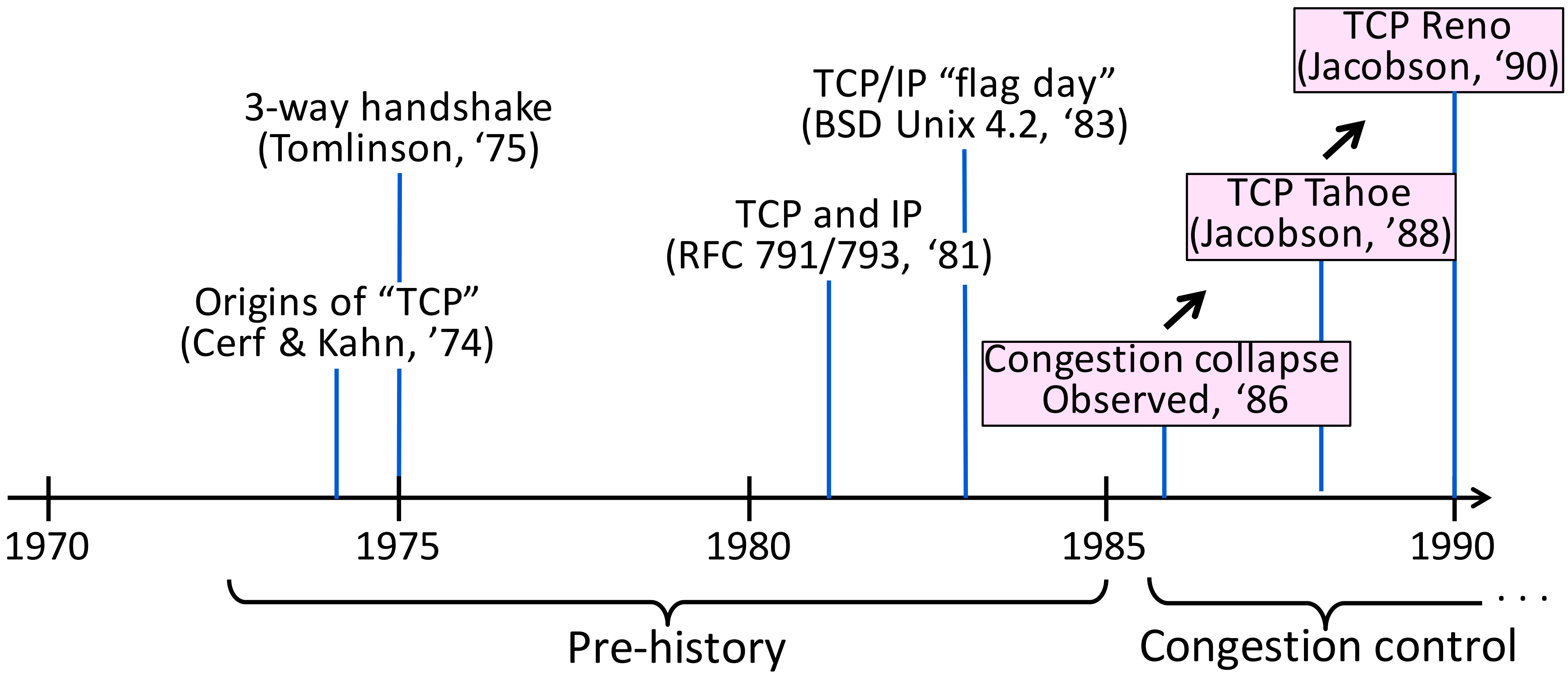
- Idea is to fix timeouts and introduce a congestion window (cwnd) over the sliding window to limit queues/loss

- TCP Tahoe/Reno implements AIMD by adapting cwnd using packet loss as the network feedback signal

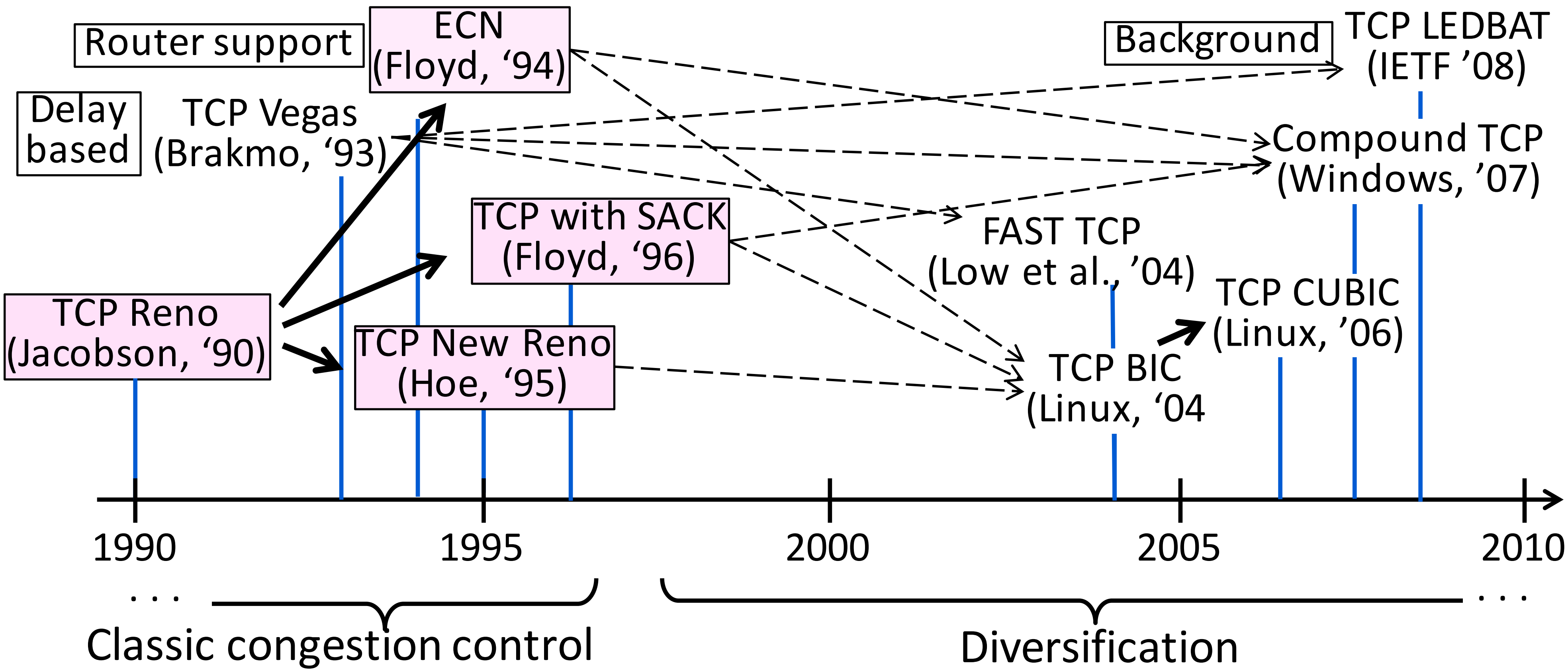
TCP Tahoe/Reno (2)

- TCP behaviors we will study:
  - ACK clocking
  - Adaptive timeout (mean and variance)
  - Slow-start
  - Fast Retransmission
  - Fast Recovery
- Together, they implement AIMD

TCP Timeline

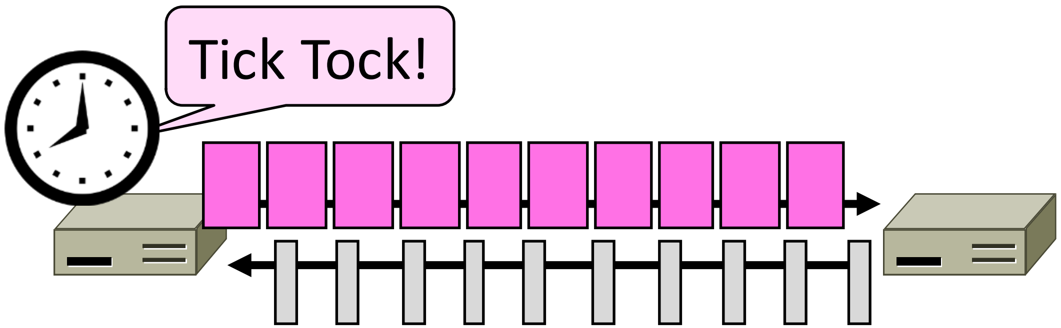


TCP Timeline (2)



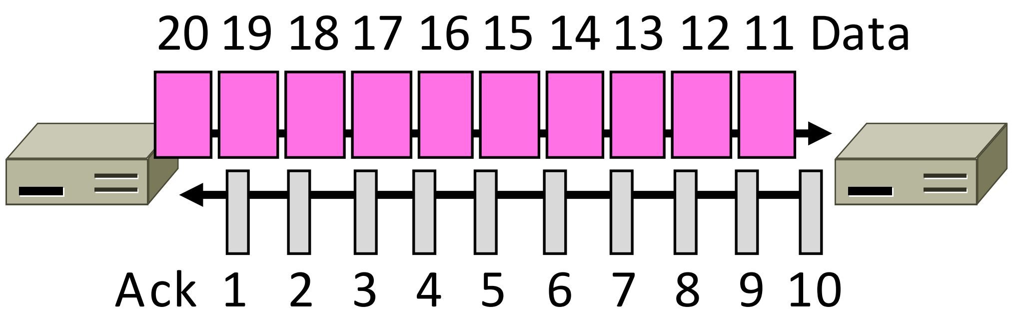
TCP Ack Clocking (§6.5.10)

- The self-clocking behavior of sliding windows, and how it is used by TCP
  - The "ACK clock"



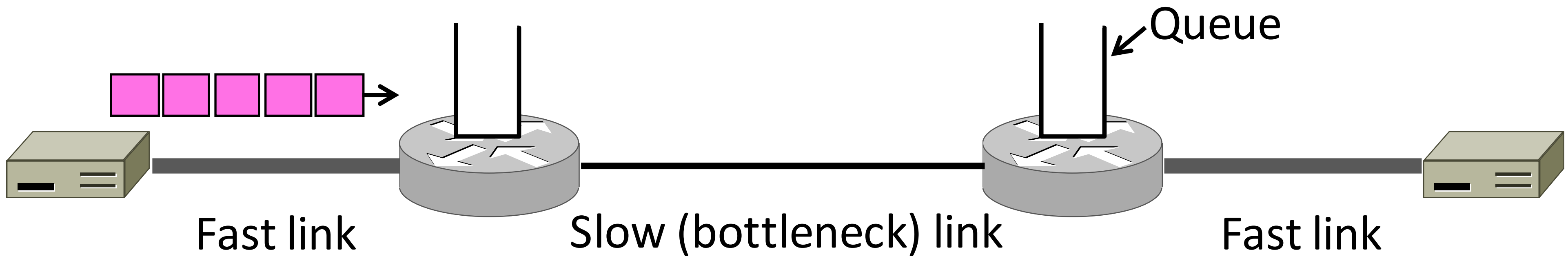
Sliding Window ACK Clock

- Each in-order ACK advances the sliding window and lets a new segment enter the network
  - ACKs "clock" data segments



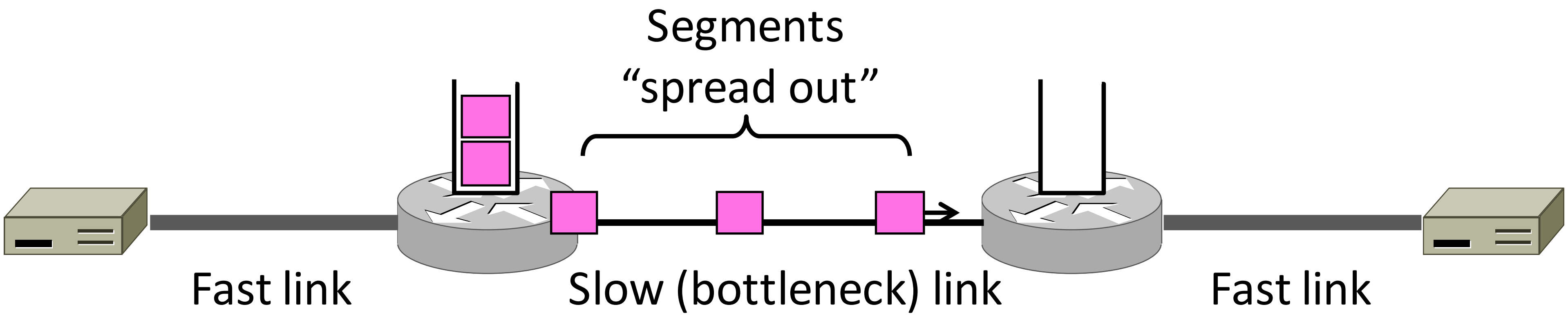
Benefit of ACK Clocking

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



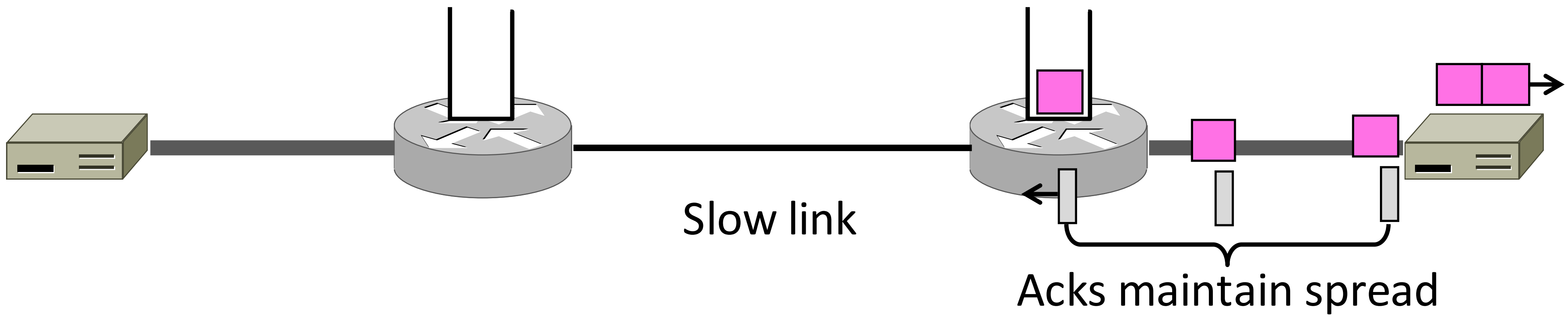
Benefit of ACK Clocking (2)

- Segments are buffered and spread out on slow link



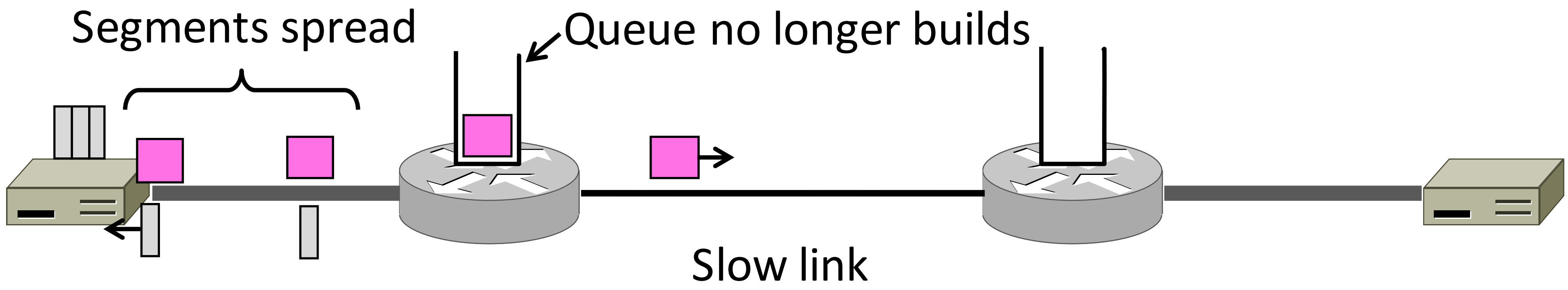
Benefit of ACK Clocking (3)

- ACKs maintain the spread back to the original sender



Benefit of ACK Clocking (4)

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



Benefit of ACK Clocking (4)

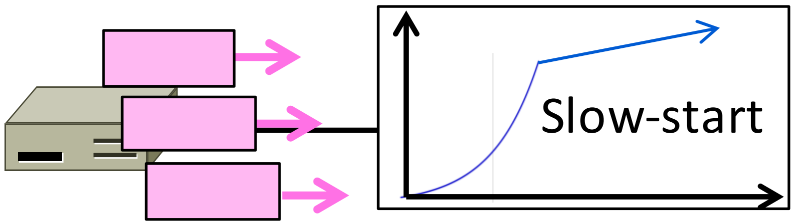
- Helps the network run with low levels of loss and delay!
- The network has smoothed out the burst of data segments
- ACK clock transfers this smooth timing back to the sender
- Subsequent data segments are not sent in bursts so they do not queue up in the network

TCP Uses ACK Clocking

- TCP uses a sliding window because of the value of ACK clocking
- Sliding window controls how many segments are inside the network
  - Called the congestion window, or cwnd
  - Rate is roughly cwnd/RTT
- TCP only sends small bursts of segments to let the network keep the traffic smooth

TCP Slow Start (§6.5.10)

- How TCP implements AIMD, part 1
  - "Slow start" is a component of the AI portion of AIMD



Considerations

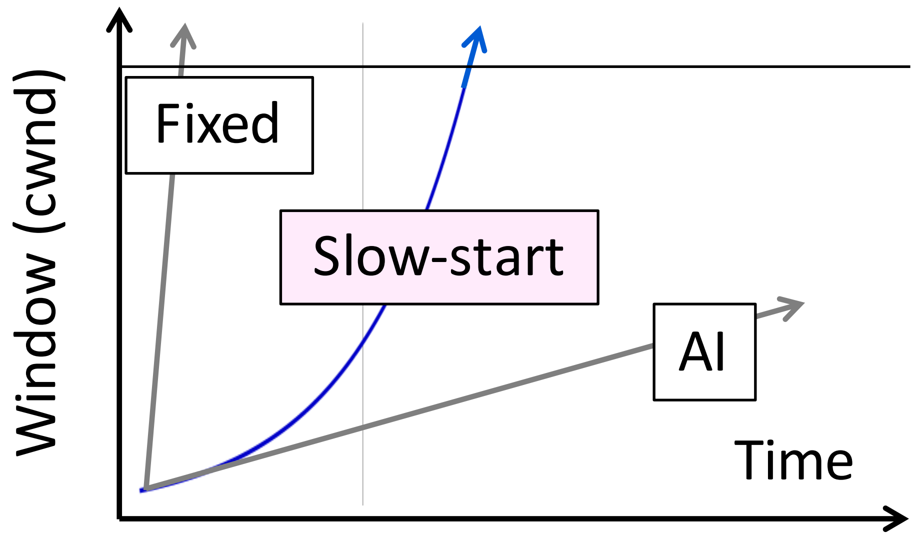
- We want TCP to follow an AIMD control law for a good allocation
- Sender uses a congestion window or cwnd to set its rate (≈cwnd/RTT)
- Sender uses packet loss as the network congestion signal
- Need TCP to work across a very large range of rates and RTTs

TCP Startup Problem

- We want to quickly near the right rate, $cwnd_{IDEAL}$, but it varies greatly
  - Fixed sliding window doesn't adapt and is rough on the network (loss!)
  - AI with small bursts adapts cwnd gently to the network, 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



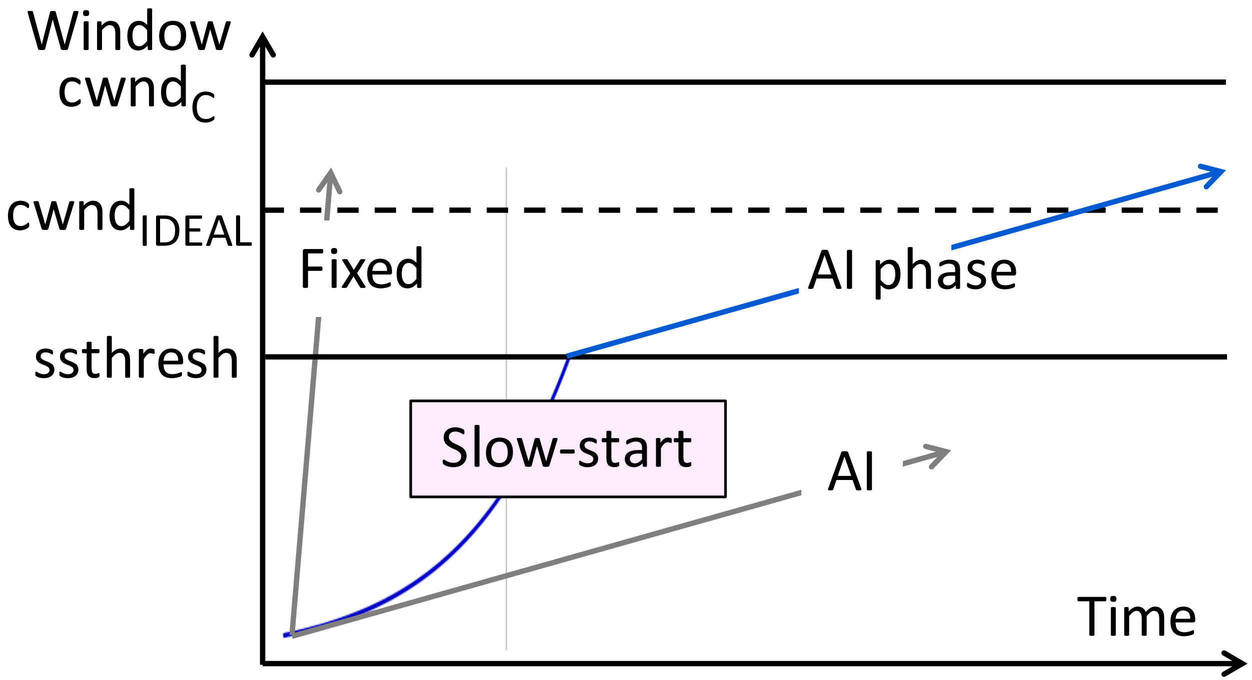
Slow-Start Solution (2)

- 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

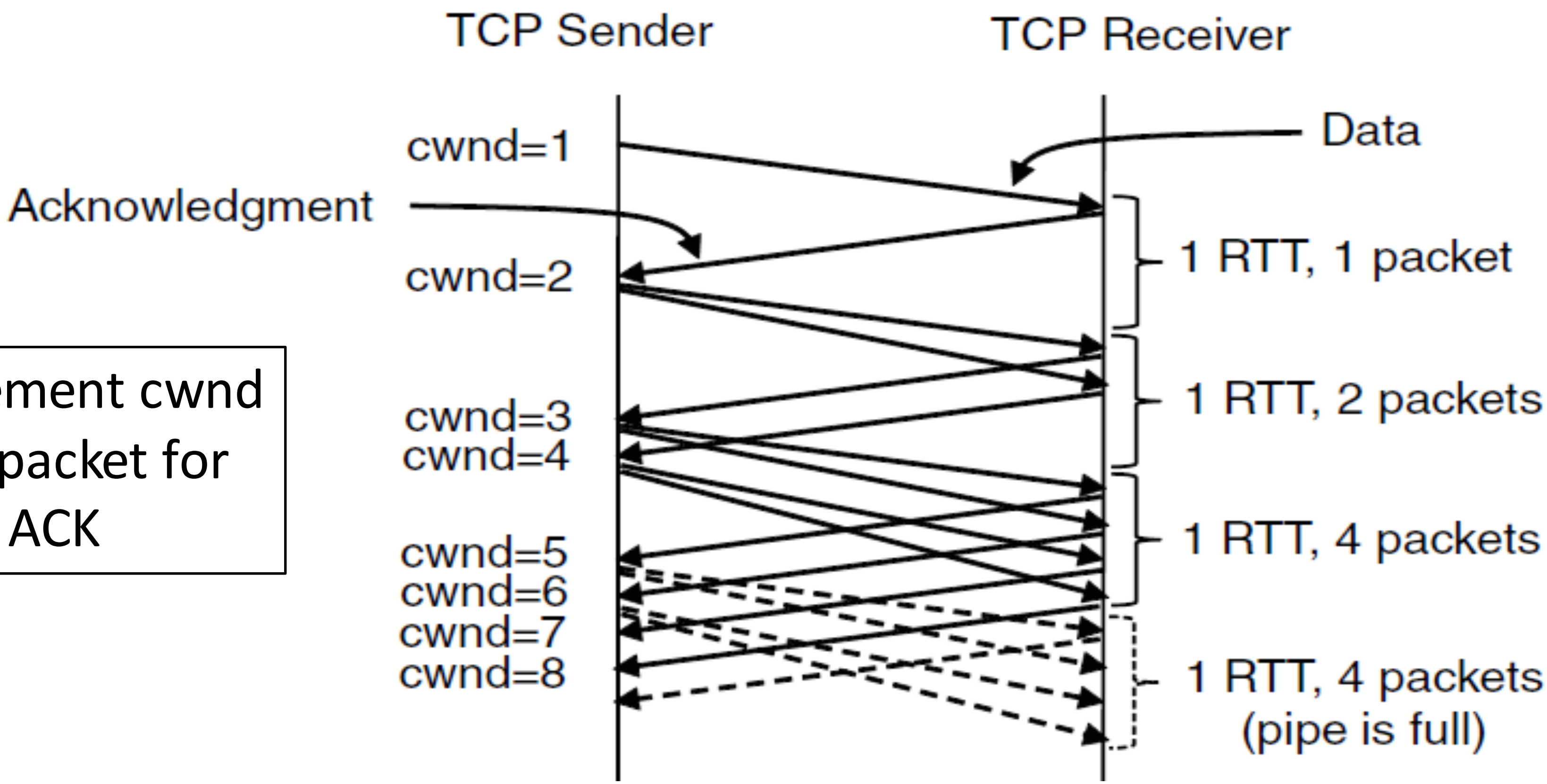
- In terms of cwnd:
  - Expect loss for $cwnd_C \approx$ 2BD+queue
  - Use ssthresh = $cwnd_C/2$ to switch to AI after observing loss

Slow-Start Solution (3)

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

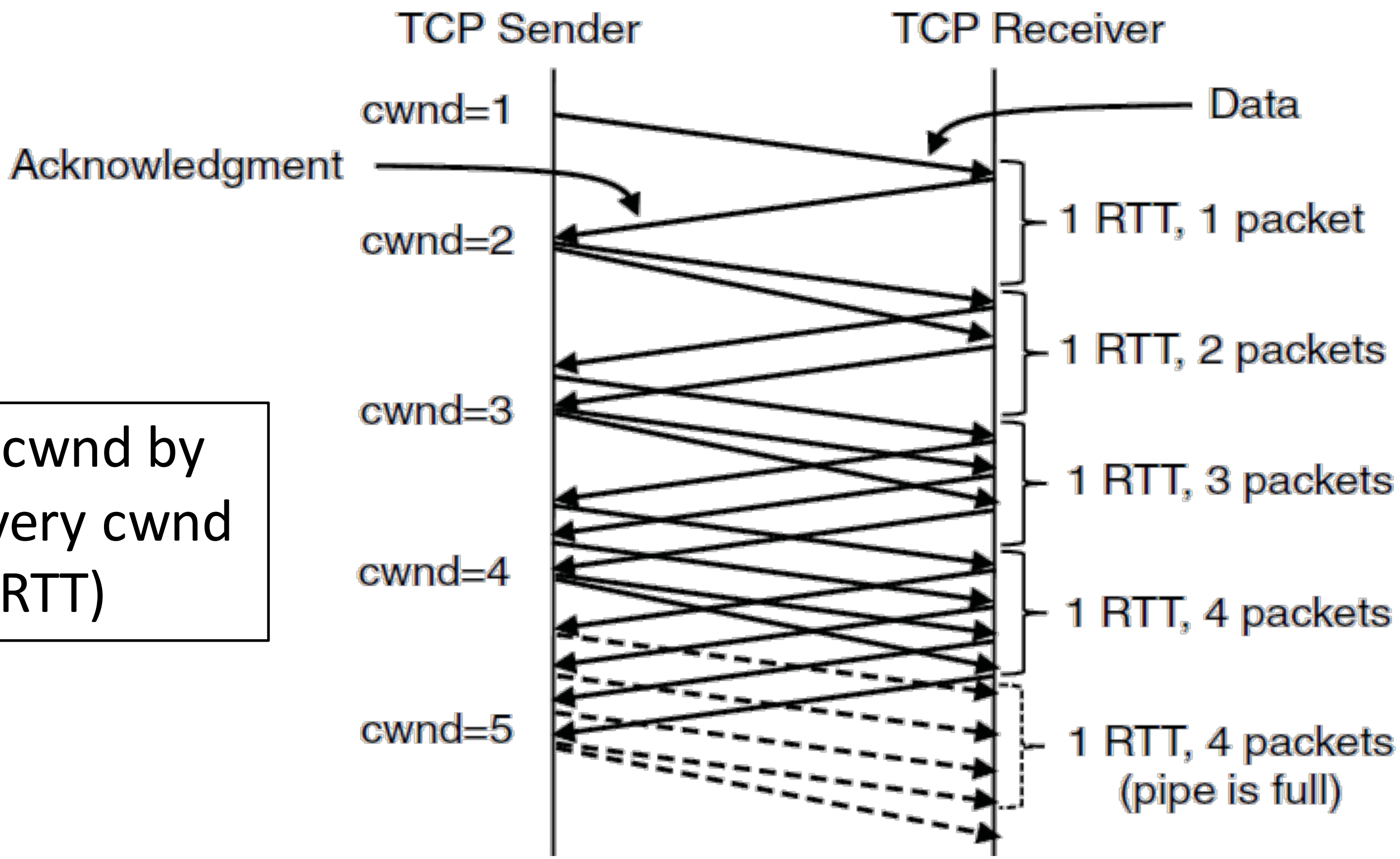


Slow-Start (Doubling) Timeline



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)

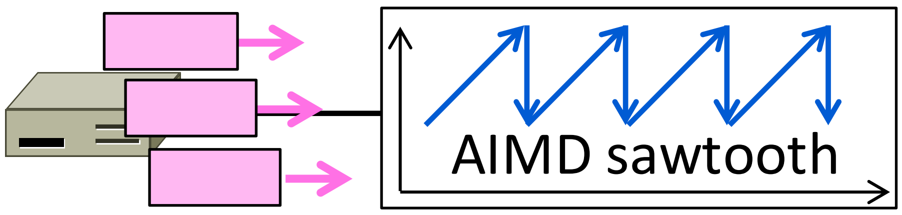
- Initial slow-start (doubling) phase
  - Start with cwnd = 1 (or small value)
  - cwnd += 1 packet per ACK
- Later Additive Increase phase
  - cwnd += 1/cwnd packets per ACK
  - Roughly adds 1 packet per RTT
- Switching threshold (initially infinity)
  - Switch to AI when cwnd > ssthresh
  - Set ssthresh = cwnd/2 after loss
  - Begin with slow-start after timeout

Timeout Misfortunes

- Why do a slow-start after timeout?
  - Instead of MD cwnd (for AIMD)
- Timeouts are sufficiently long that the ACK clock will have run down
  - Slow-start ramps up the ACK clock
- We need to detect loss before a timeout to get to full AIMD
  - Done in TCP Reno

TCP Fast Retransmit / Fast Recovery (§6.5.10)

- How TCP implements AIMD, part 2
  - "Fast retransmit" and "fast recovery" are the MD portion of AIMD



Recall

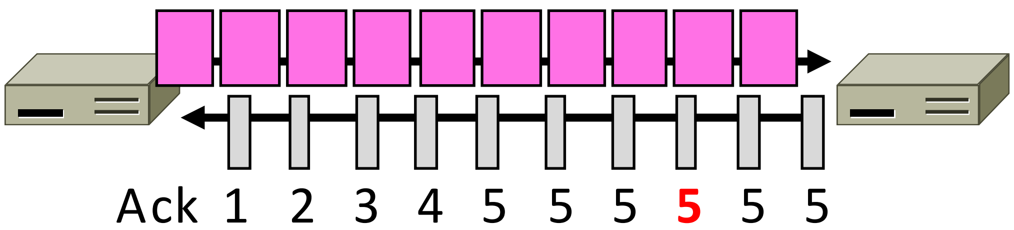
- We want TCP to follow an AIMD control law for a good allocation
- Sender uses a <u>congestion window</u> or <u>cwnd</u> to set its rate (≈cwnd/RTT)
- Sender uses slow-start to ramp up the ACK clock, followed by Additive Increase
- But after a timeout, sender slow-starts again with cwnd=1 (as it no ACK clock)

Inferring Loss from ACKs

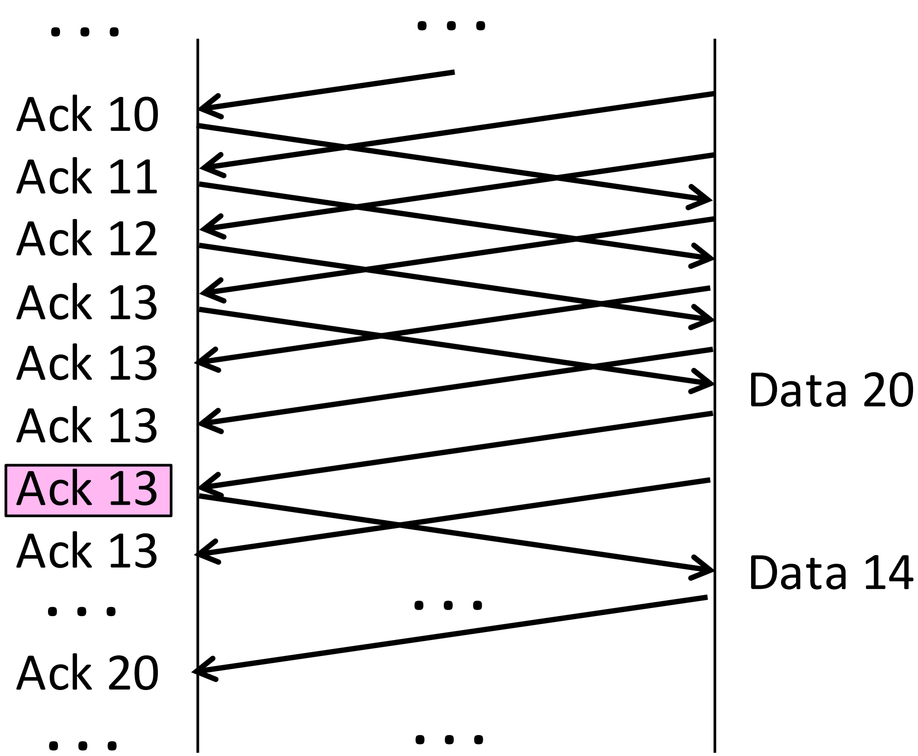
- TCP uses a cumulative ACK
  - Carries highest in-order seq. number
  - Normally a steady advance
- Duplicate ACKs give us hints about what data hasn't arrived
  - Tell us some new data did arrive, but it was not next segment
  - Thus the next segment may be lost

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 (2)



Third duplicate ACK, so send 14

ACK jumps after loss is repaired

Data 14 was lost earlier, but got 15 to 20

Retransmission fills in the hole at 14

Fast Retransmit (3)

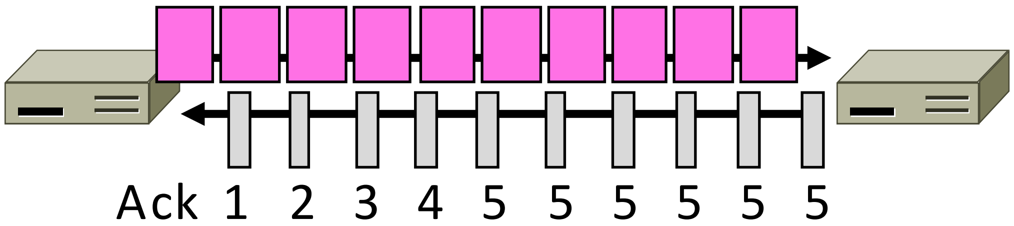
- It can repair single segment loss quickly, typically before a timeout
- However, we have quiet time at the sender/receiver while waiting for the ACK to jump
- And we still need to MD cwnd …

Inferring Non-Loss from ACKs

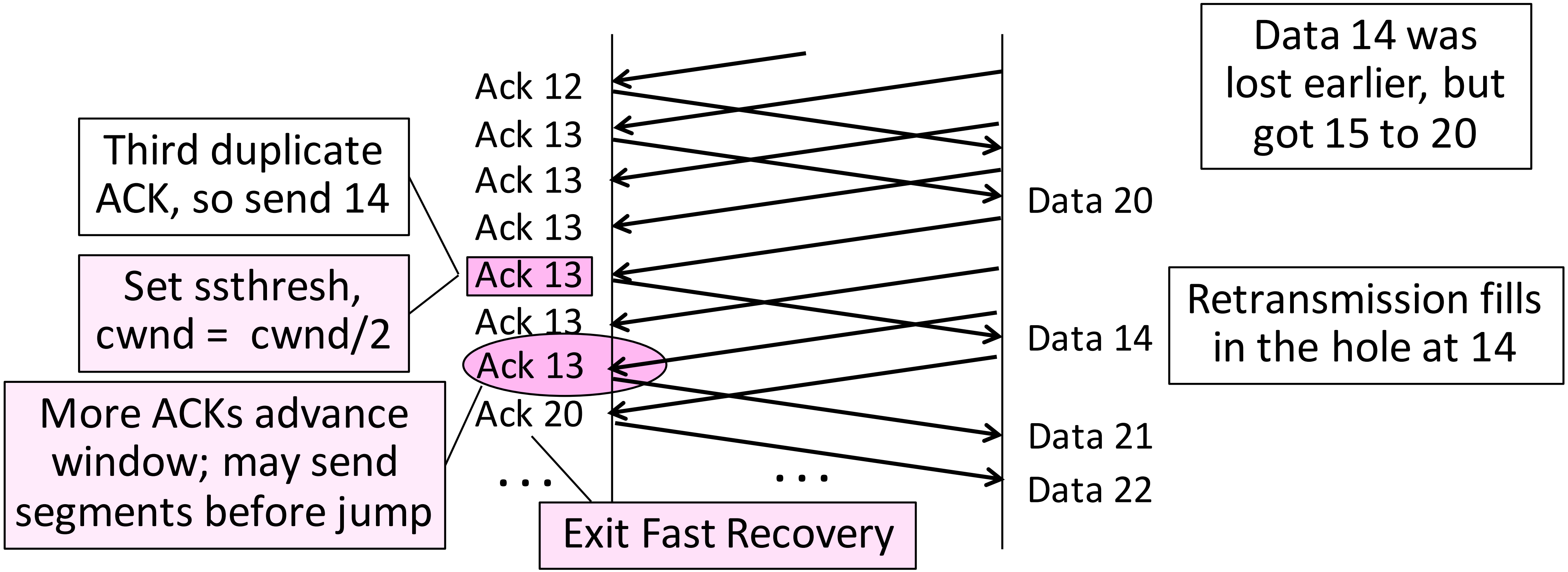
- Duplicate ACKs also give us hints about what data has arrived
  - Each new duplicate ACK means that some new segment has arrived
  - It will be the segments after the loss
  - Thus advancing the sliding window will not increase the number of segments stored in the network

Fast Recovery

- 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



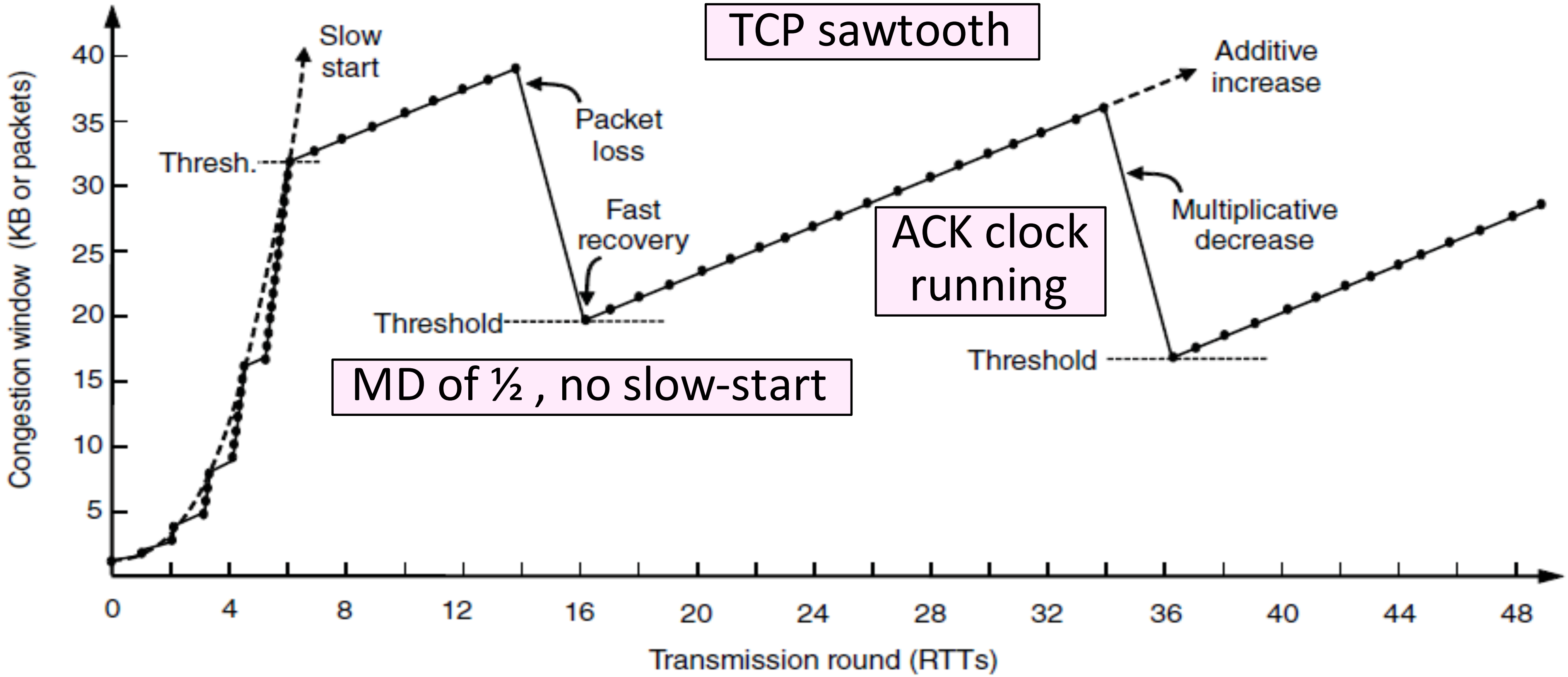
Fast Recovery (2)



Fast Recovery (3)

- 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

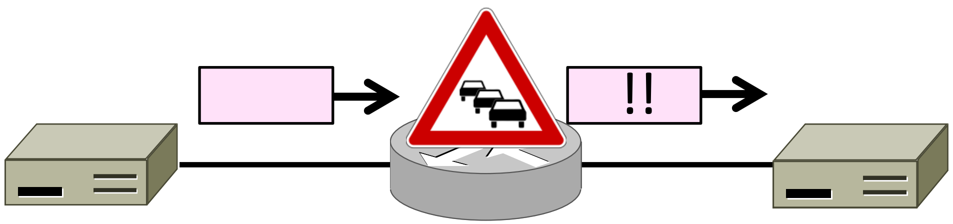

TCP sawtooth
Slow start
Additive increase
Packet loss
Fast recovery
Thresh.
Threshold
Threshold
ACK clock running
Multiplicative decrease
MD of ½ , no slow-start
Congestion window (KB or packets)
40
35
30
25
20
15
10
5
0
4
8
12
16
20
24
28
32
36
40
44
48
Transmission round (RTTs)

TCP Reno, NewReno, and SACK

- Reno can repair one loss per RTT
  - Multiple losses cause a timeout
- NewReno further refines ACK heuristics
  - Repairs multiple losses without timeout
- SACK is a better idea
  - Receiver sends ACK ranges so sender can retransmit without guesswork

Explicit Congestion Notification (§5.3.4, §6.5.10)

- How routers can help hosts to avoid congestion
  - Explicit Congestion Notification



Congestion Avoidance vs. Control

- Classic TCP drives the network into congestion and then recovers
  - Needs to see loss to slow down
- Would be better to use the network but avoid congestion altogether!
  - Reduces loss and delay
- But how can we do this?

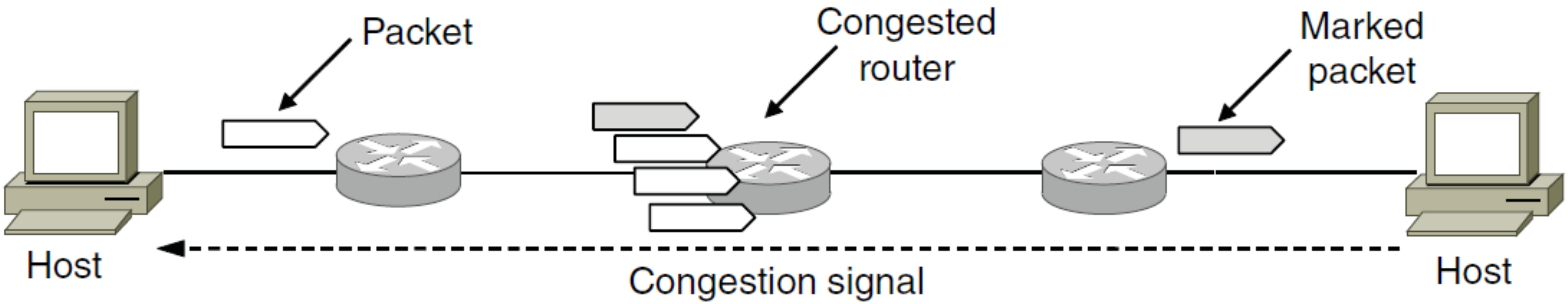
Feedback Signals

- Delay and router signals can let us avoid congestion

Signal	Example Protocol	Pros / Cons
Packet loss	Classic TCP Cubic TCP (Linux)	Hard to get wrong Hear about congestion late
Packet delay	Compound TCP (Windows)	Hear about congestion early Need to infer congestion
Router indication	TCPs with Explicit Congestion Notification	Hear about congestion early Require router support

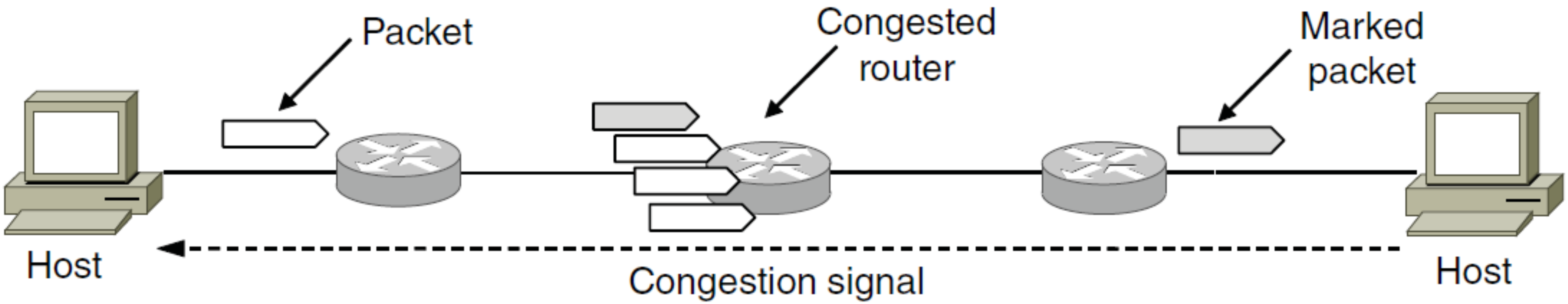
ECN (Explicit Congestion Notification)

- Router detects the onset of congestion via its queue
  - When congested, it marks affected packets (IP header)



ECN (2)

- Marked packets arrive at receiver; treated as loss
  - TCP receiver reliably informs TCP sender of the congestion



ECN (3)

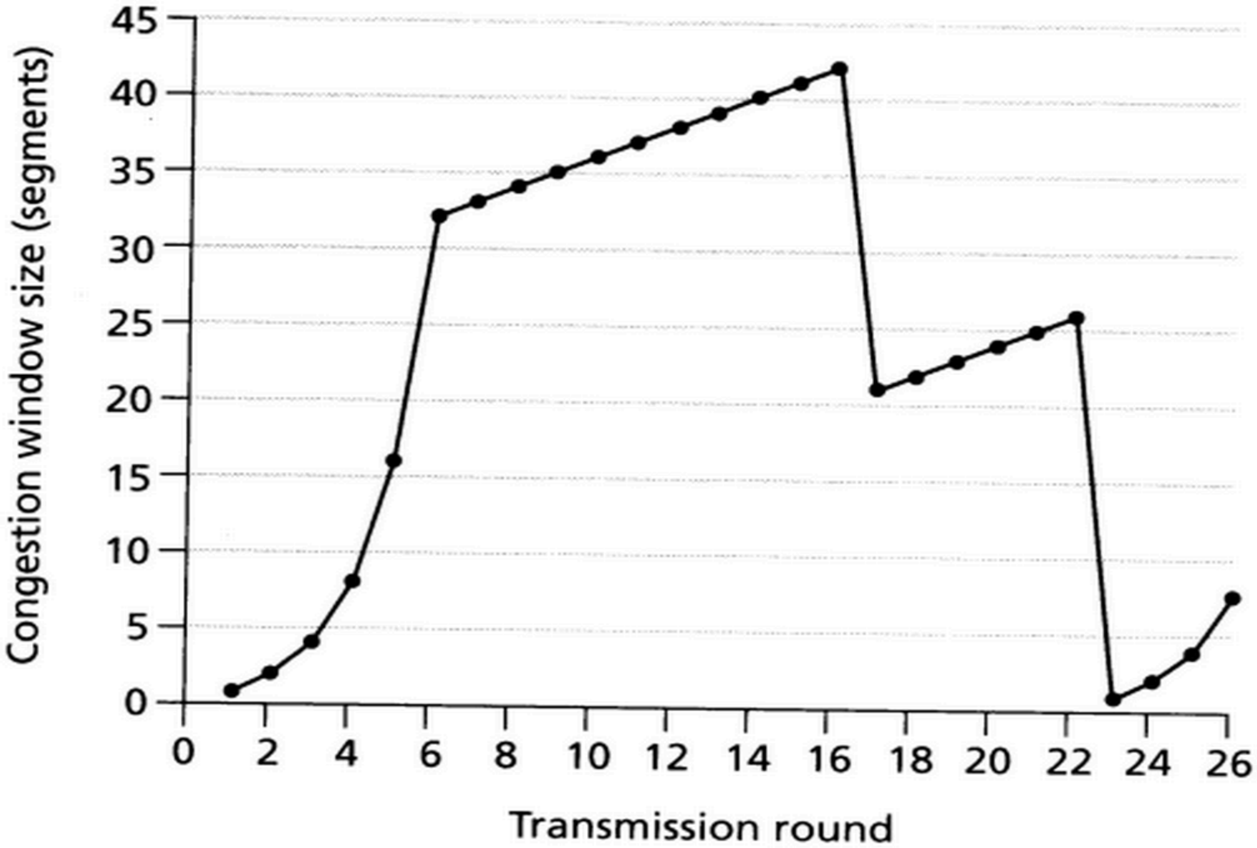
- Advantages
  - Routers deliver clear signal to hosts
  - Congestion is detected early, no loss
  - No extra packets need to be sent
- Disadvantage
  - Routers and both sender and receiver must be upgraded

Example (1)

Assume a TCP sender without fast retransmit, but with slow start and additive increase. Also assume:

- Segments n, n+1, n+2, …, n+10 transmitted at times 0,1,2,…,10 ms
- Transmission time / segment = 1 ms
- RTT (2 x propagation + transmision + ack processing + ack transmission) = 10 ms
- Segment n is lost (only)
- In order segments and ACKs
- Retransmission timer for segment n is 60 ms, starting at the end of transmission
- cwnd = ssthresh = 64 at time 0
- offeredWindow = 70

Example (2)


Congestion window size (segments)
45
40
35
30
25
20
15
10
5
0
0
2
4
6
8
10
12
14
16
18
20
22
24
26
Transmission round