

Transport Layer (Congestion Control)

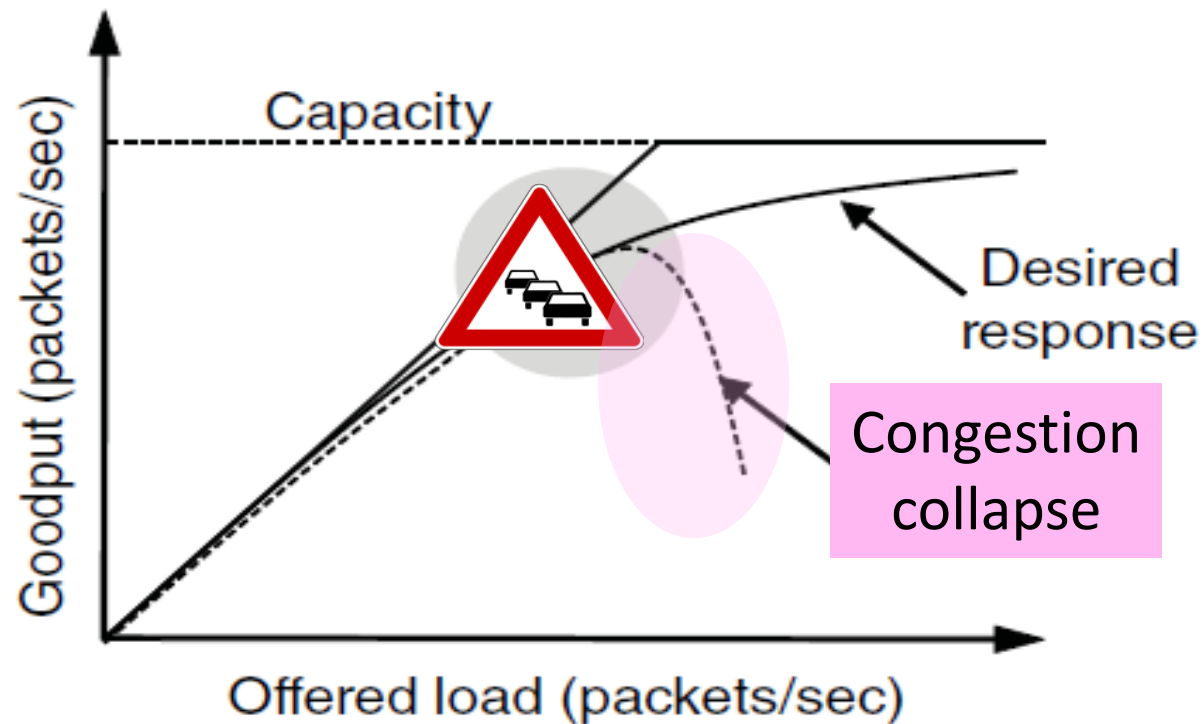
Congestion Collapse

Congestion Collapse in the 1980s

- Early TCP used fixed size window (e.g., 8 packets)
 - Initially fine for reliability
- But something 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 goodput fell



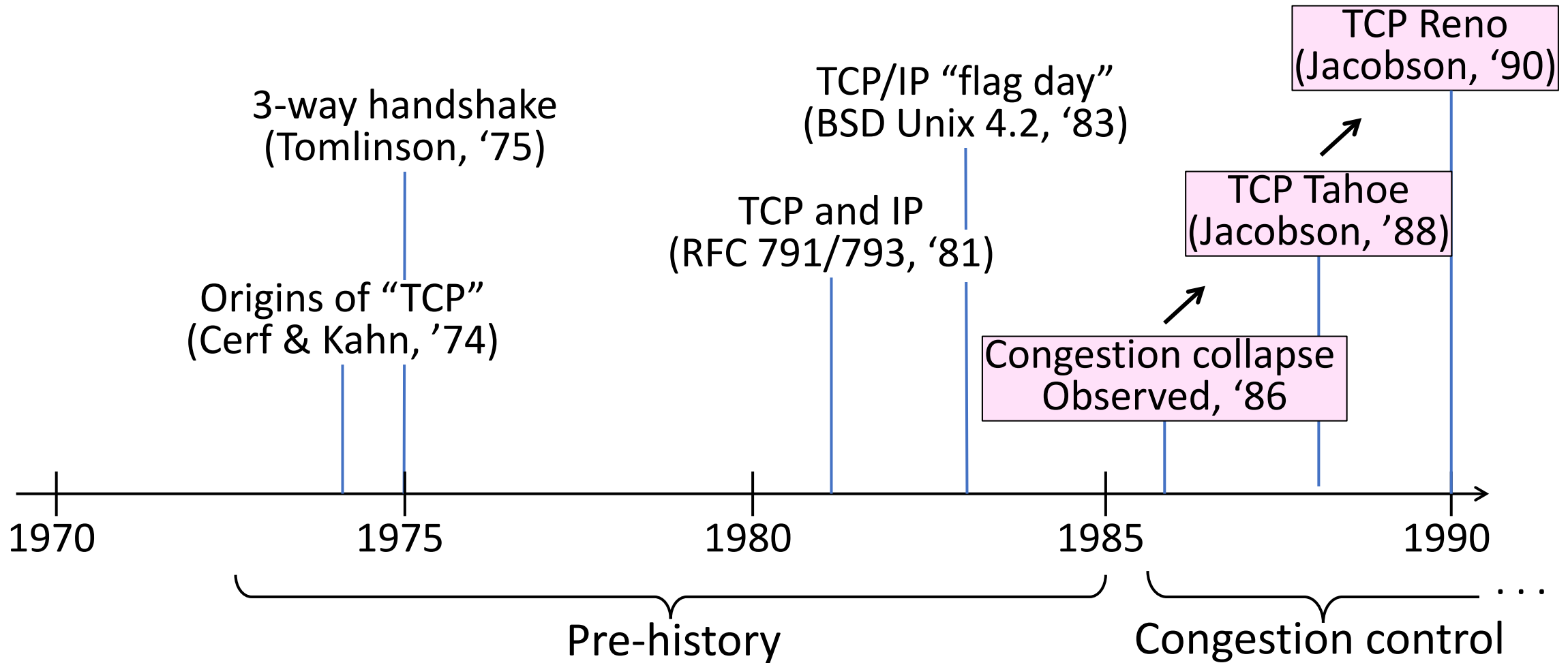
Van Jacobson (1950—)

- 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

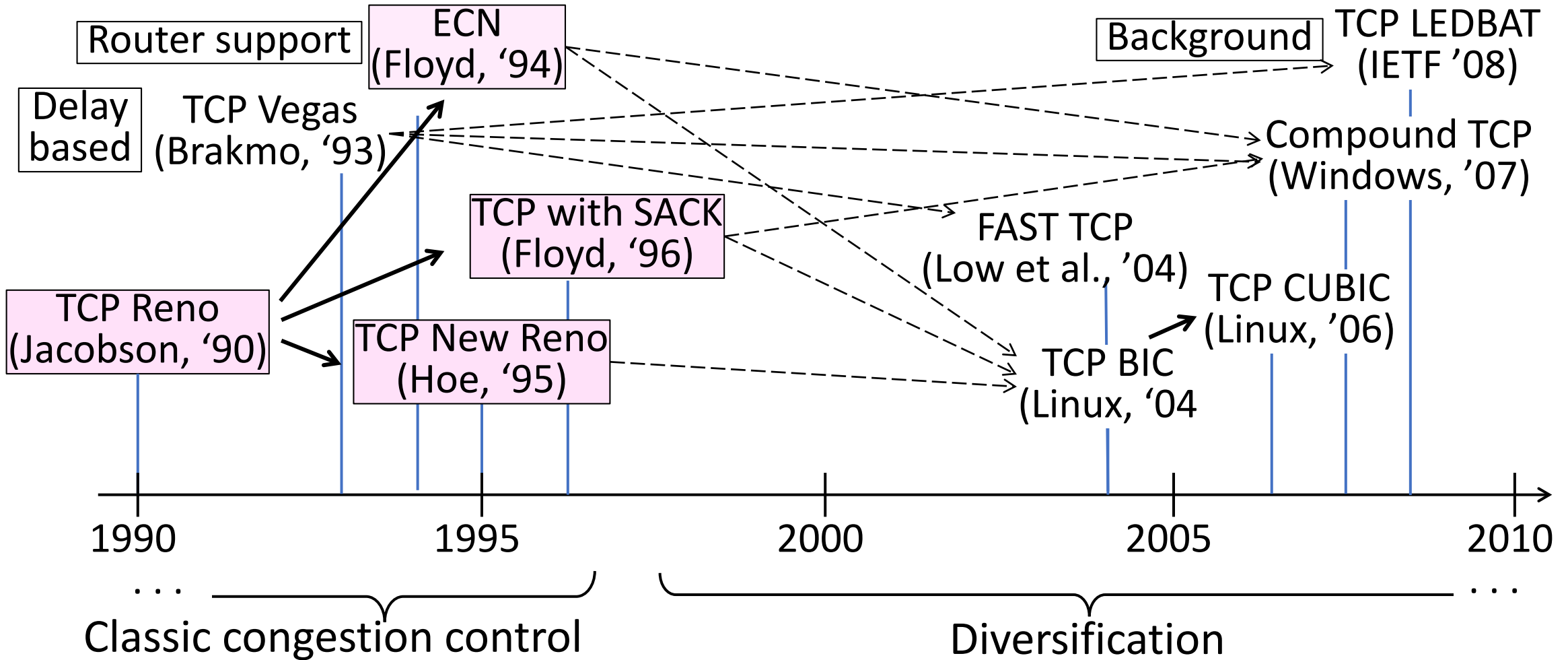
TCP Tahoe/Reno

- TCP extensions we will study:
 - ACK clocking
 - Adaptive timeout (mean and variance)
 - Slow-start
 - Fast Retransmission
 - Fast Recovery

TCP Timeline



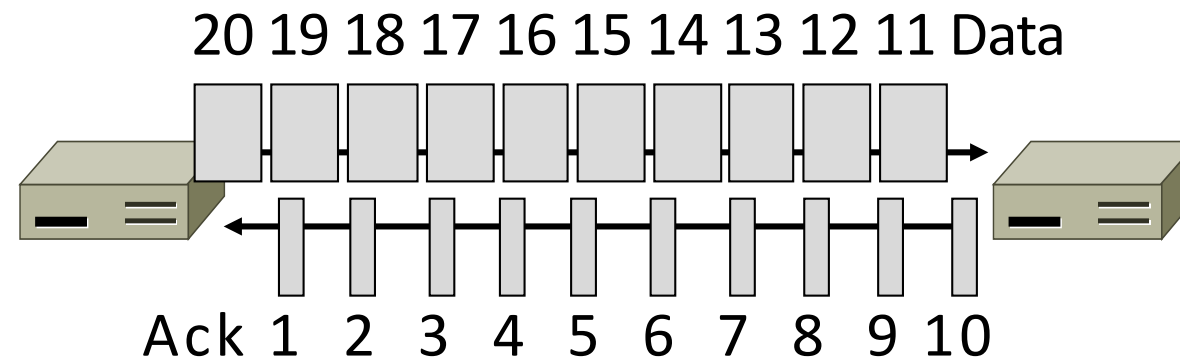
TCP Timeline (2)



ACK Clocking

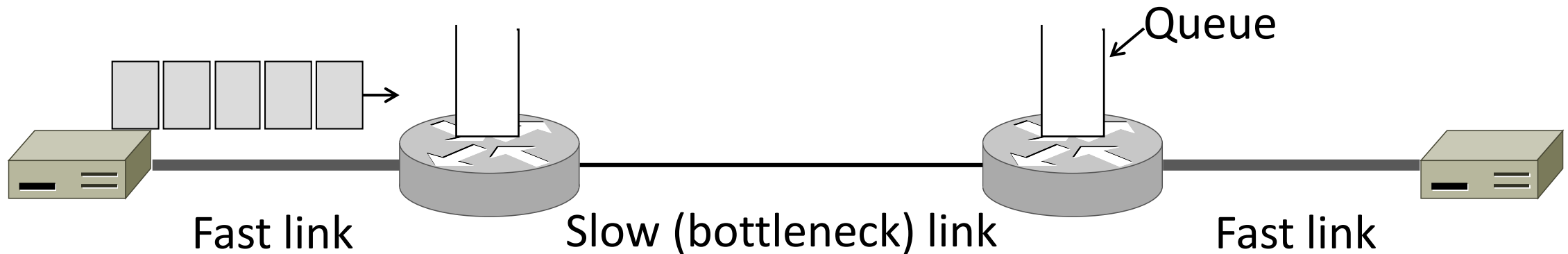
Sliding Window ACK Clock

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



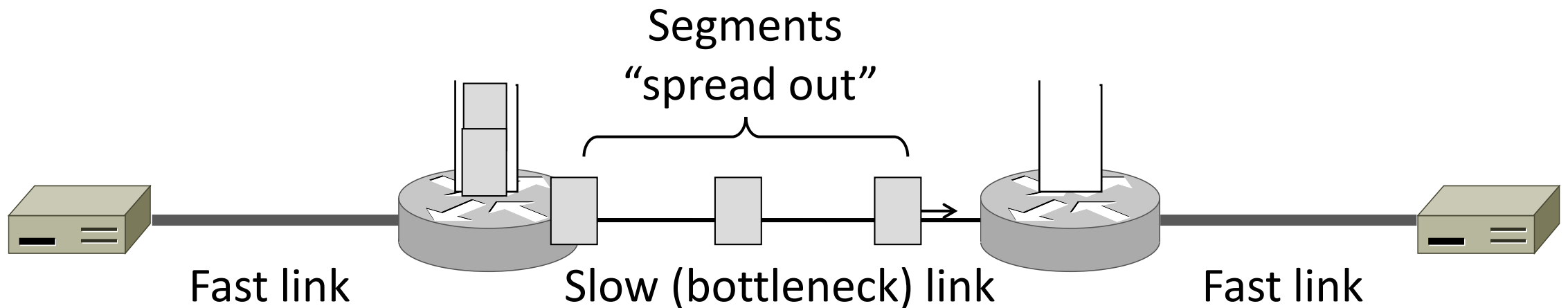
Benefit of ACK Clocking

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



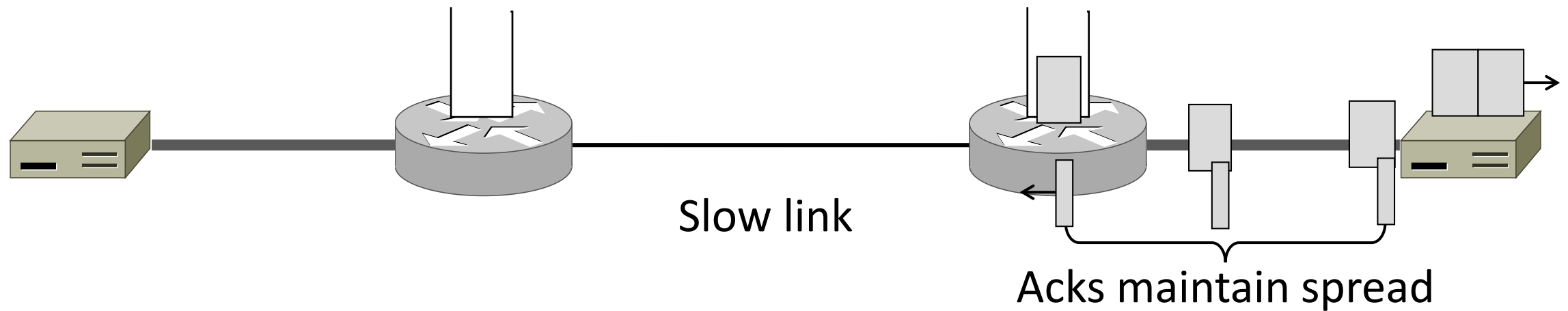
Benefit of ACK Clocking (2)

- Segments are buffered and spread out on slow link



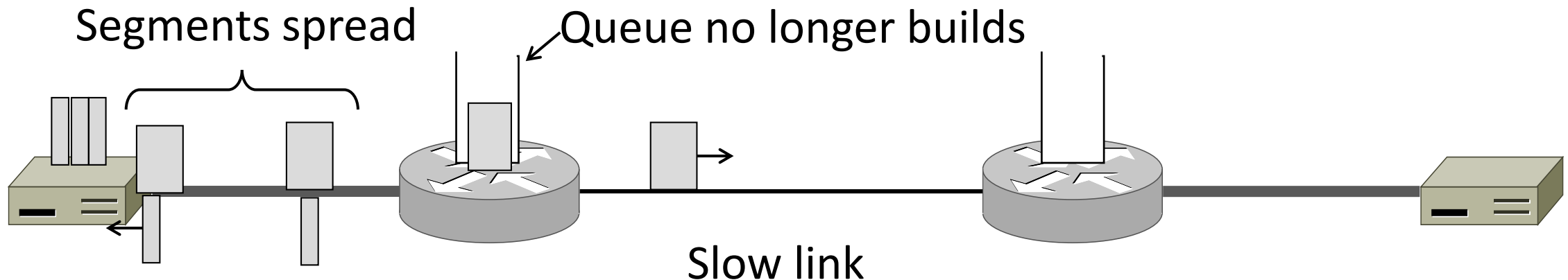
Benefit of ACK Clocking (3)

- 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 rate without queuing!



Benefit of ACK Clocking (4)

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

TCP Uses ACK Clocking

- TCP uses a sliding window for congestion control 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

Slow Start (TCP Additive Increase)

Practical AIMD

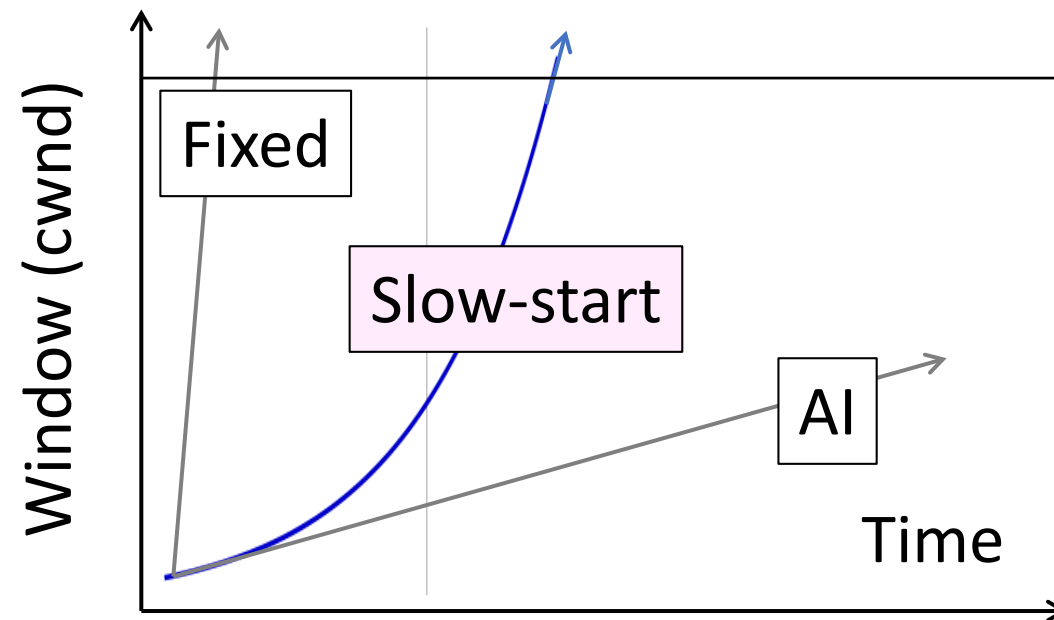
- We want TCP to follow an AIMD control law for a good allocation
- Sender uses a congestion window or cwnd to set its rate ($\approx \text{cwnd}/\text{RTT}$)
- Sender uses **loss** as network **congestion signal**
- Need TCP to work across a very large range of rates and RTTs

TCP Startup Problem

- We want to quickly reach the right rate, $cwnd_{IDEAL}$, but it varies greatly
 - Fixed sliding window doesn't adapt and is rough on the network (loss!)
 - Additive Increase 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

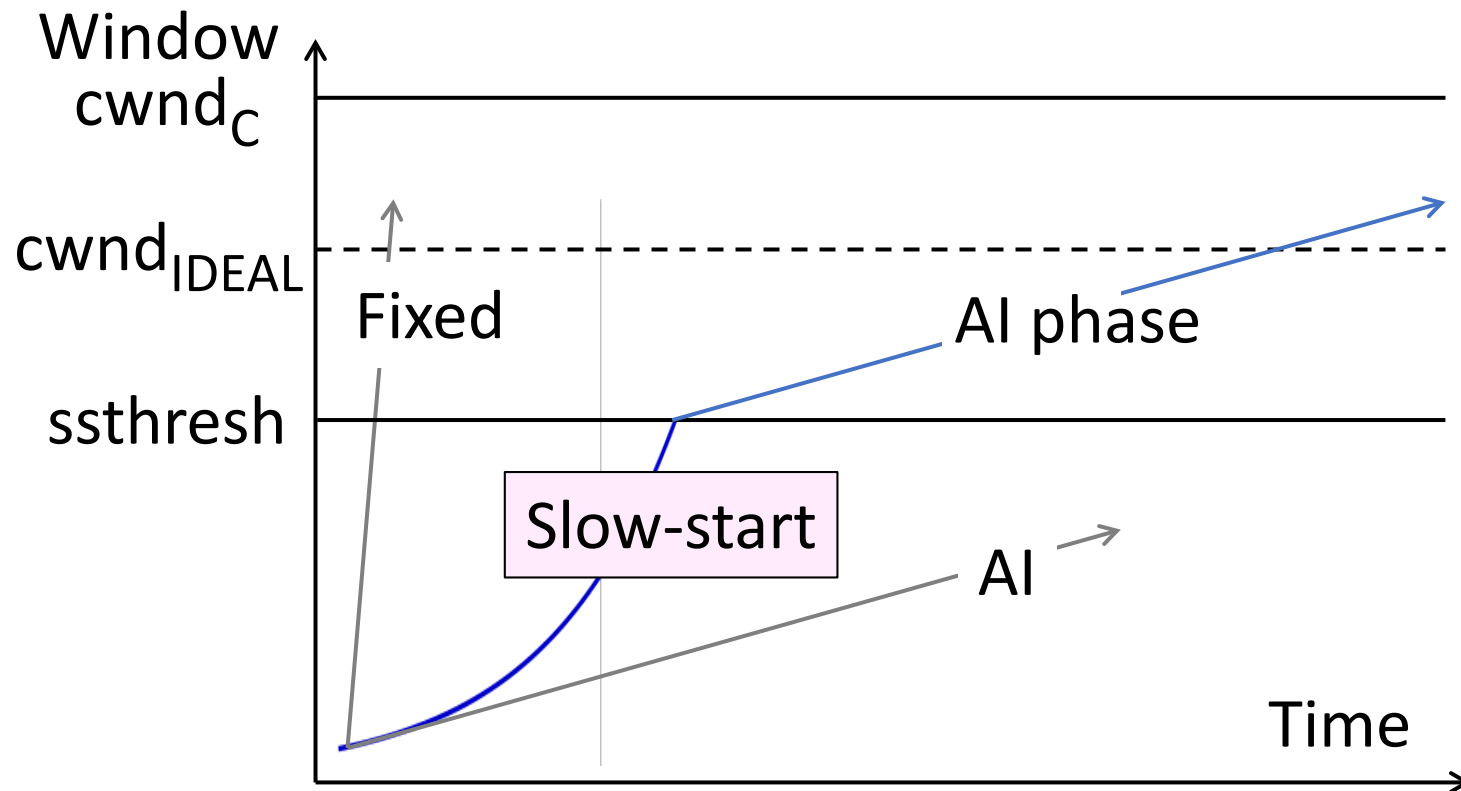


Slow-Start Solution (2)

- 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 $\text{cwnd}_c \approx 2BD + \text{queue}$
 - Use $\text{ssthresh} = \text{cwnd}_c / 2$ to switch to AI

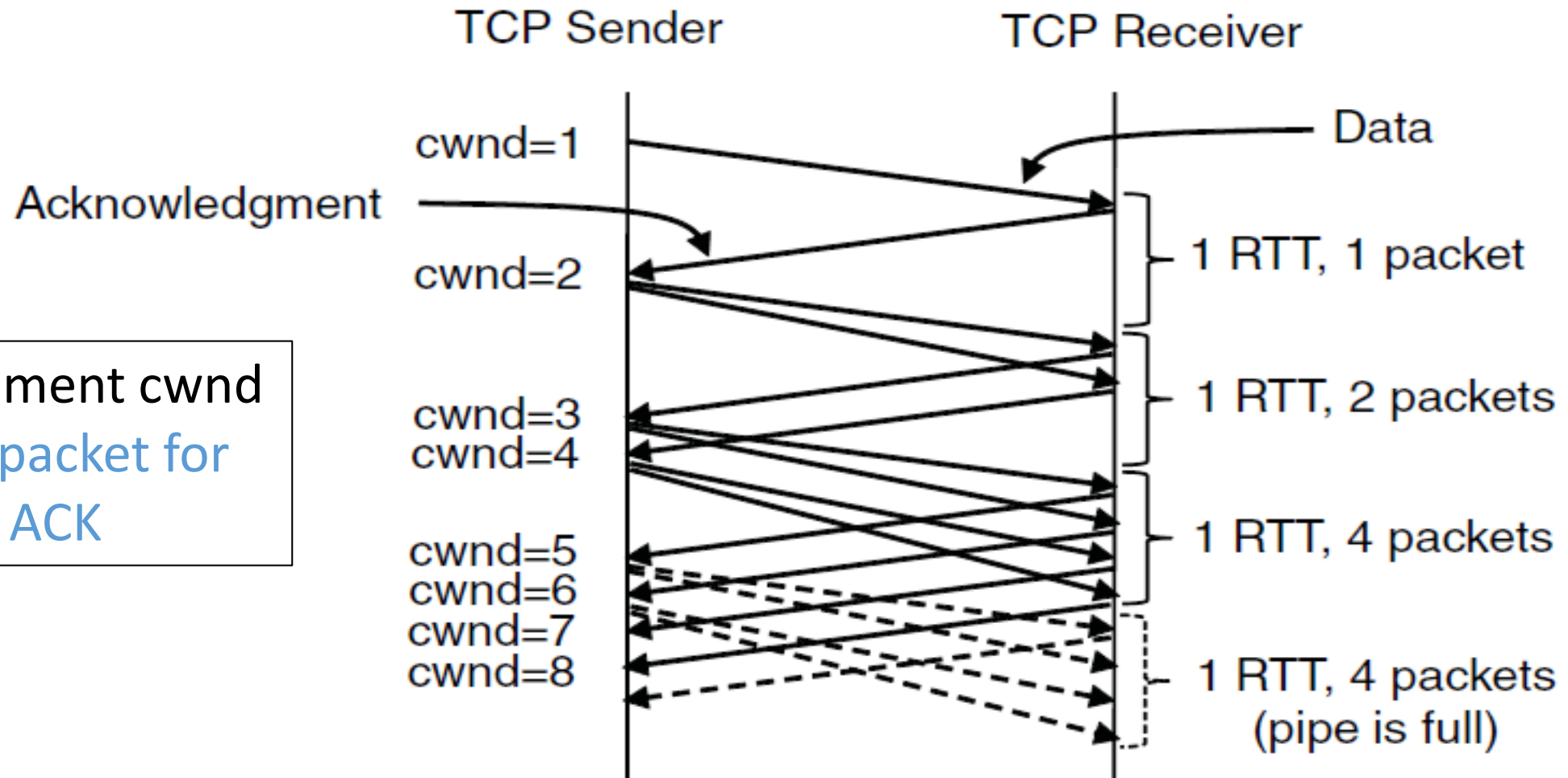
Slow-Start Solution (3)

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



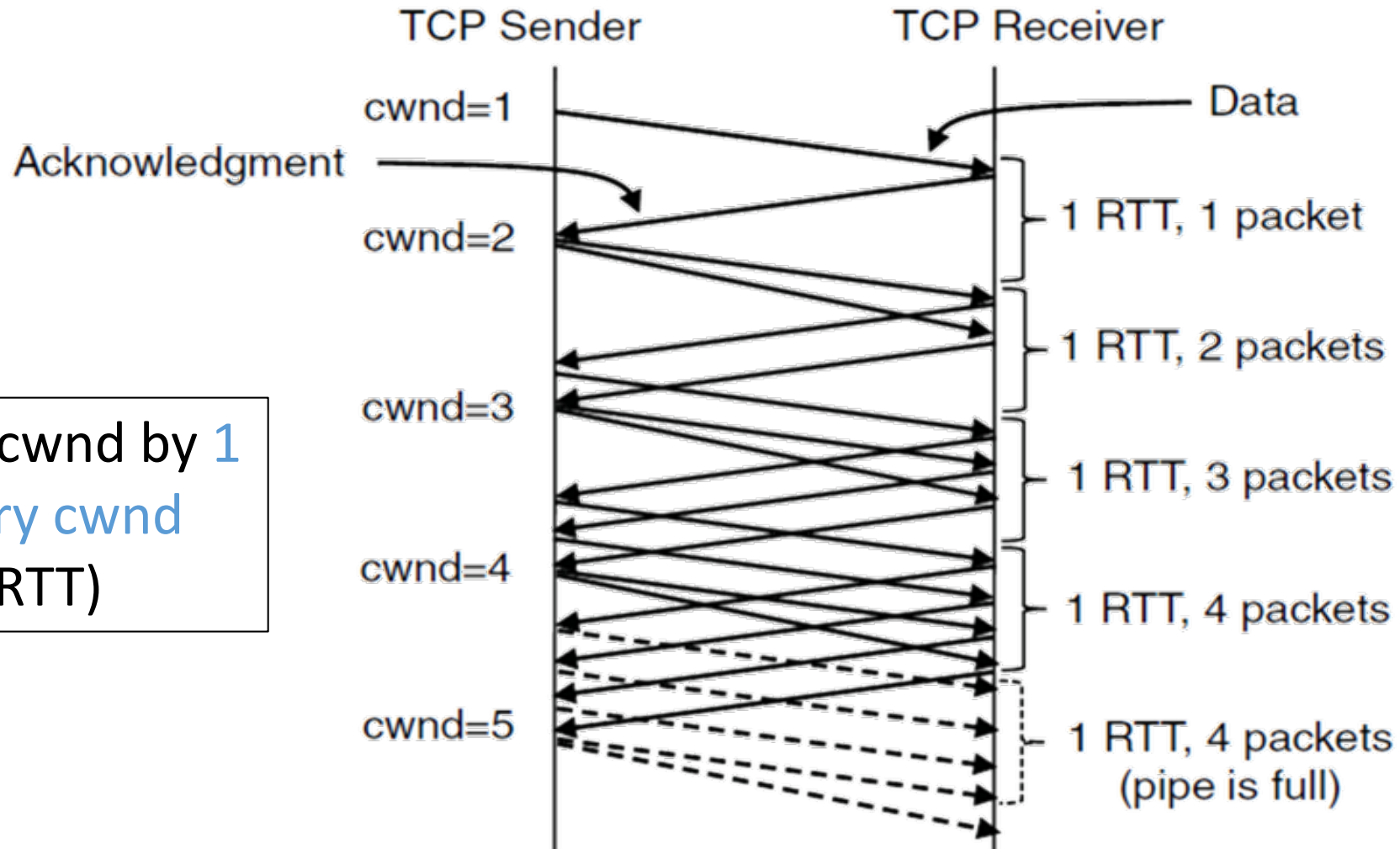
Slow-Start (Doubling) Timeline

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 $\text{cwnd} = 1$ (or small value)
 - $\text{cwnd} += 1$ packet per ACK
- Later Additive Increase phase
 - $\text{cwnd} += 1/\text{cwnd}$ packets per ACK
 - Roughly adds 1 packet per RTT
- Switching threshold (initially infinity)
 - Switch to AI when $\text{cwnd} > \text{ssthresh}$
 - Set $\text{ssthresh} = \text{cwnd}/2$ 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

Fast Recovery
(TCP Multiplicative Decrease)

Practical AIMD (2)

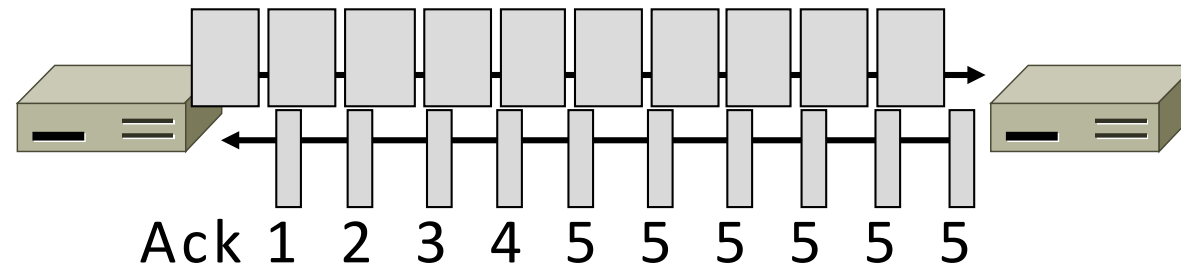
- We want TCP to follow an AIMD control law for a good allocation among competing streams
- Sender uses a congestion window or cwnd to set its rate ($\approx \text{cwnd}/\text{RTT}$)
- Sender uses slow-start to ramp up the ACK clock, followed by Additive Increase
- But after a timeout, sender slow-starts again with $\text{cwnd}=1$ (as it no ACK clock)

Inferring Loss from ACKs

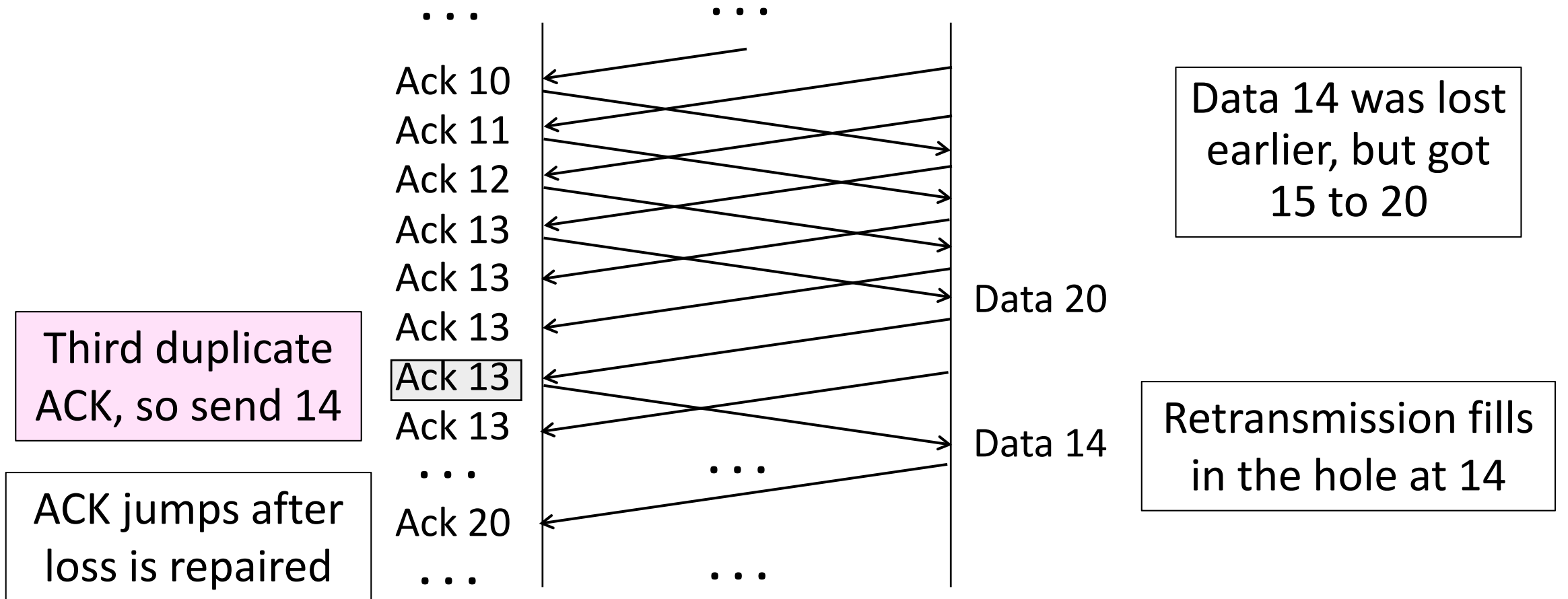
- TCP uses a cumulative ACK
 - Carries highest in-order seq. number
 - Normally a steady advance
- Duplicate ACKs give us hints about what data hasn't arrived
 - Tell us some new data did arrive, but it was not next segment
 - Thus the next un-ack'ed segment may be lost

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)



Fast Retransmit (3)

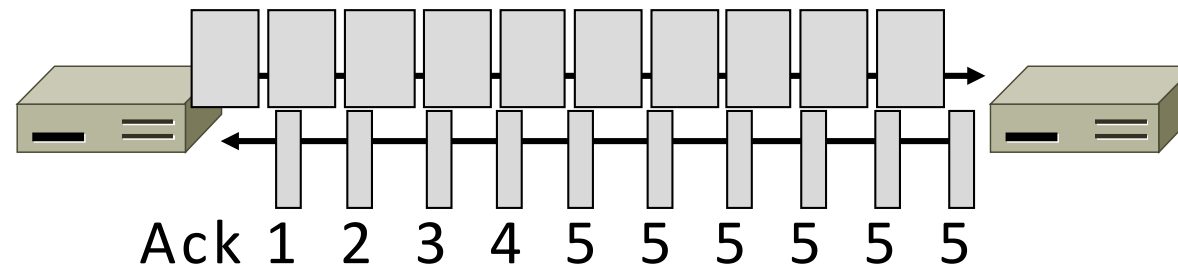
- It can repair single segment loss quickly, typically before a timeout
- However, we have quiet time at the sender/receiver while waiting for the ACK to jump
- And we still need to MD cwnd ...

Inferring Non-Loss from ACKs

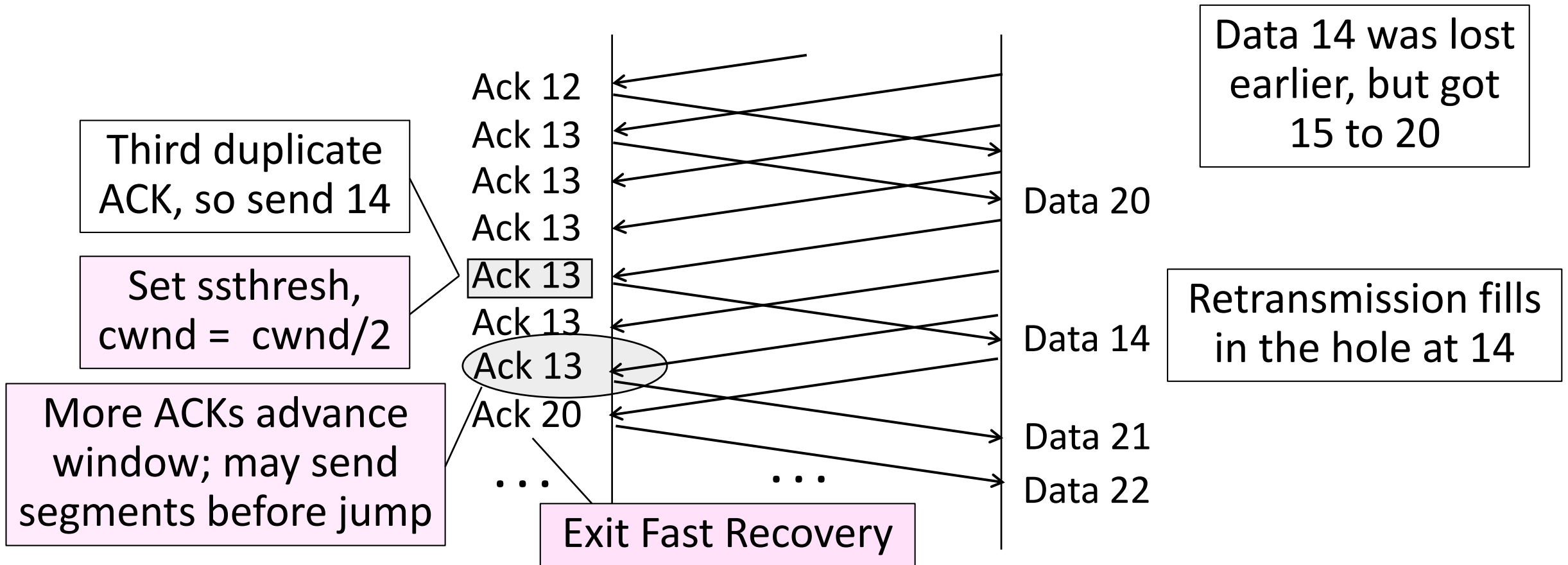
- Duplicate ACKs also give us hints about what data has arrived
 - Each new duplicate ACK means that some new segment has arrived
 - It will be the segments after the loss
 - Thus advancing the sliding window will not increase the number of segments stored in the network

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



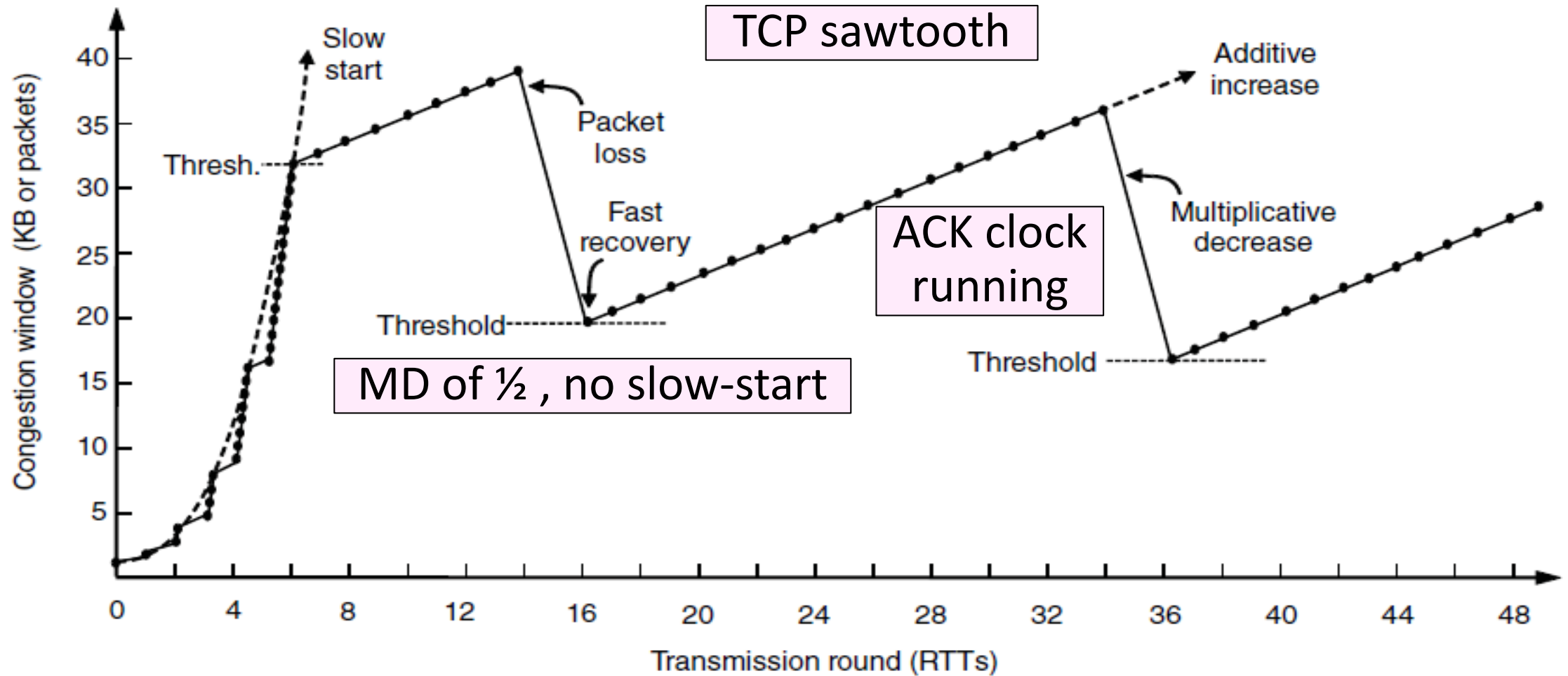
Fast Recovery (2)



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 $\frac{1}{2}$

TCP Reno



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
- Selective ACK (SACK) is a better idea
 - Receiver sends ACK ranges so sender can retransmit without guesswork

Network-Side Congestion Control

Congestion Avoidance vs. Control

- Classic TCP drives the network into congestion and then recovers
 - Needs to **experience 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?

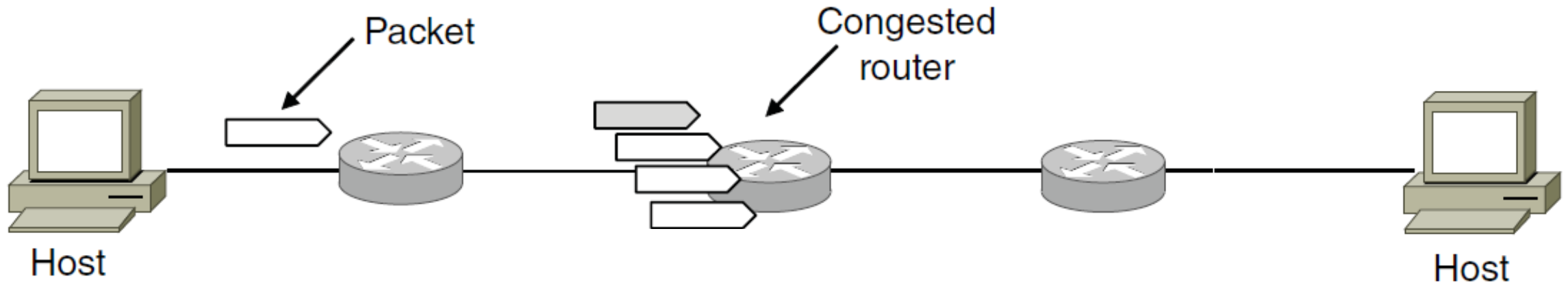
Feedback Signals

- 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

Congestion Notification

- Congested router wants to tell source hosts to slow down



Random Early Detection (RED)

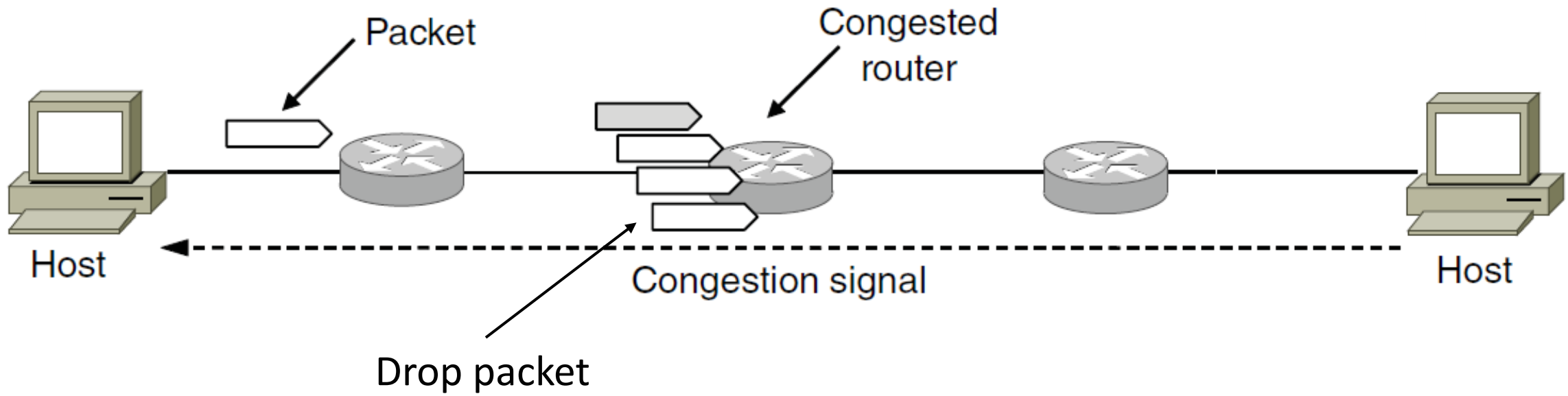
- Challenge: how to let source(s) know to slow down?
- Idea: drop a packet!
 - We know they're using TCP, make use of that fact
- Receiver will signal that loss through duplicate ACKs
- Result: Router signals congestion to sender
 - But without adding headers or doing packet inspection

Random Early Detection (RED)

- How does router choose which packet to drop?
- Want most urgently for the biggest flows to slow down
 - The biggest flows likely have the most packets in the buffer
- **Drop at random**, with probability depending on queue size
 - If queue empty, accept packet always
 - If queue full, always drop
 - As queue approaches full, increase likelihood of packet drop
 - Example: 1 queue slot left, 10 packets expected, 90% chance of drop

RED (Random Early Detection)

- Router detects the onset of congestion via its queue
 - Prior to congestion, drop a packet to signal

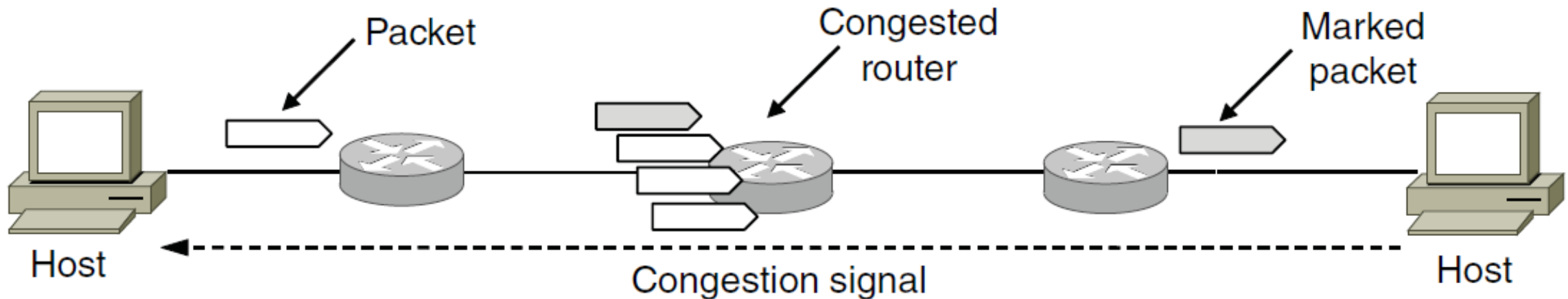


Alternative: Explicit Congestion Notification

- Dropping packets does slow down senders, but...
- It also drops packets
- Is there a way for **router** to **communicate** with **senders** without dropping packets
- Yes, but it requires some help from TCP

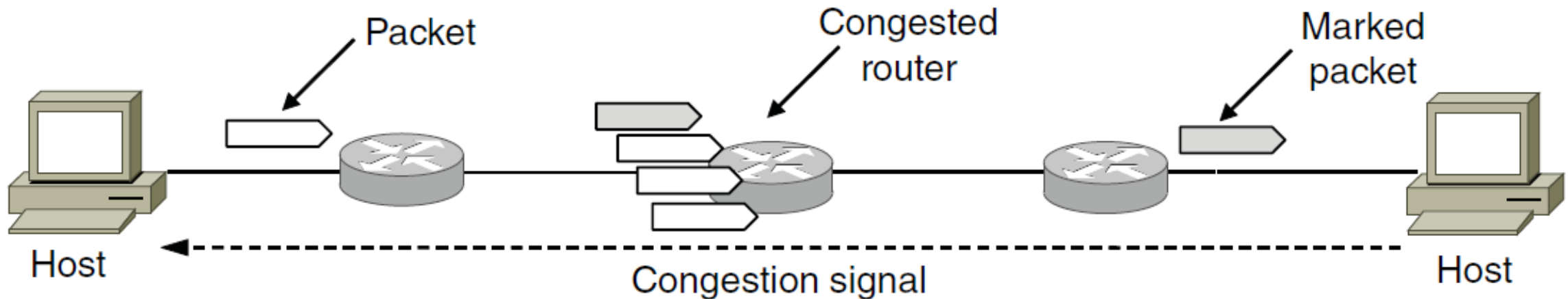
ECN (Explicit Congestion Notification)

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



ECN (2)

- Marked packets arrive at **receiver**
 - 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
- **Disadvantages:**
 - Routers and hosts required changes