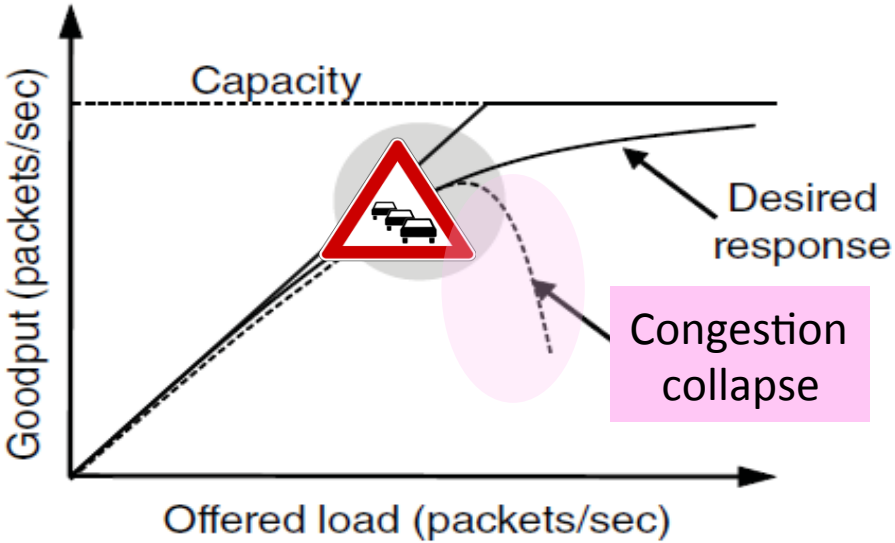
Congestion Collapse in the 1980s

- 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 goodput fell



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

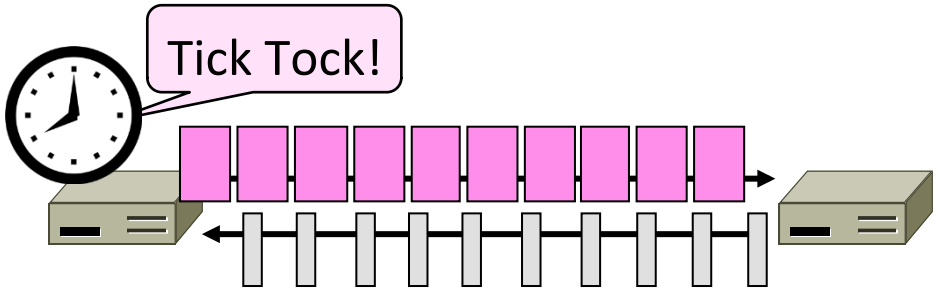
TCP Tahoe/Reno (2)

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

- Together, they implement AIMD

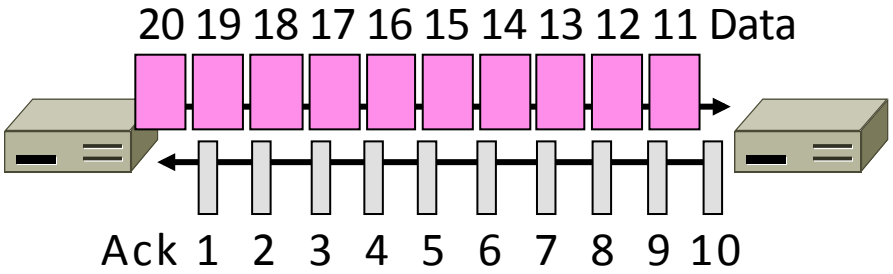
Ack Clocking

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



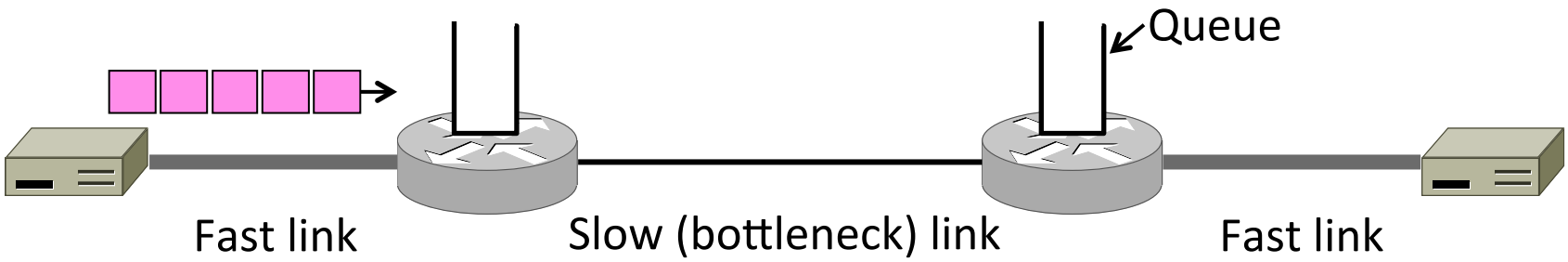
Sliding Window ACK Clock

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



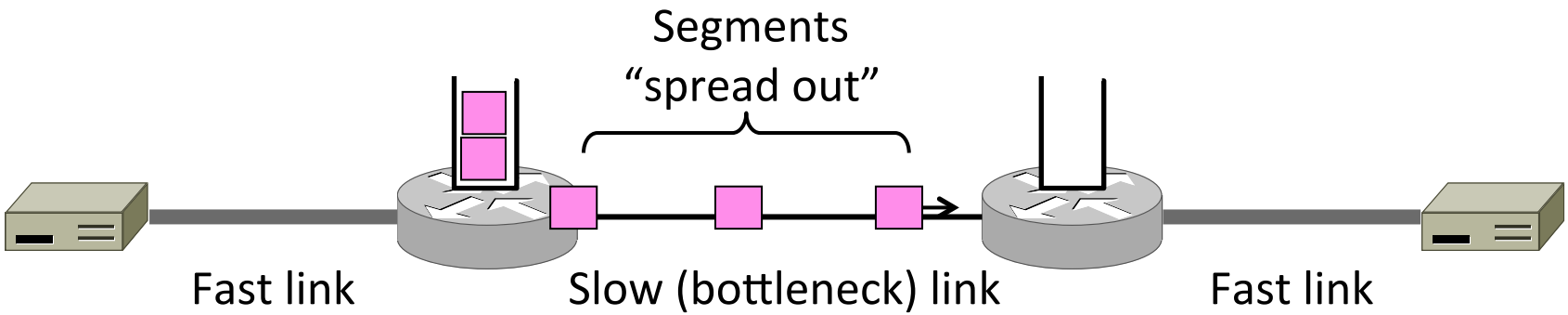
Benefit of ACK Clocking

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



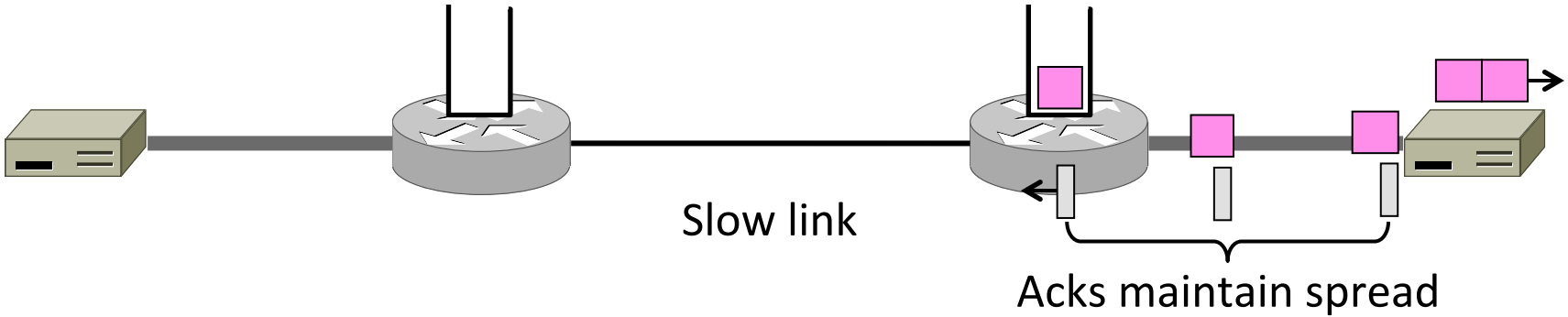
Benefit of ACK Clocking (2)

- Segments are buffered and spread out on slow link



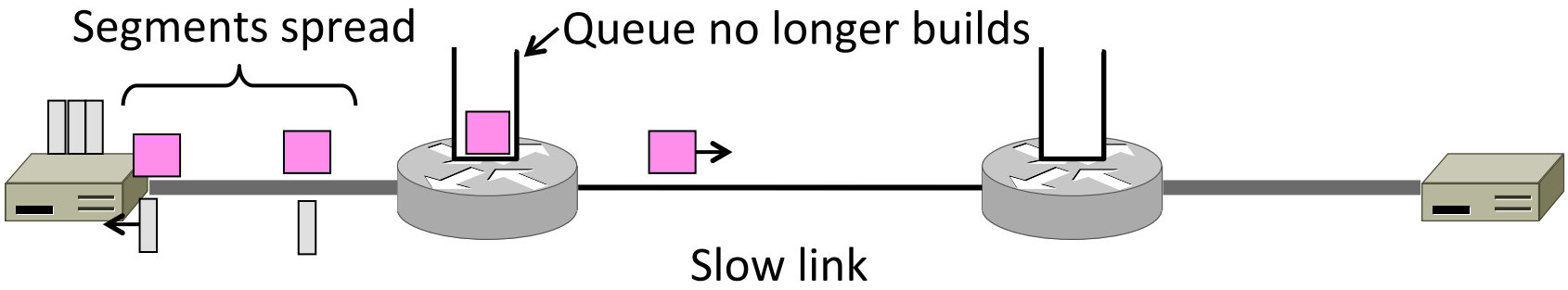
Benefit of ACK Clocking (3)

- 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!

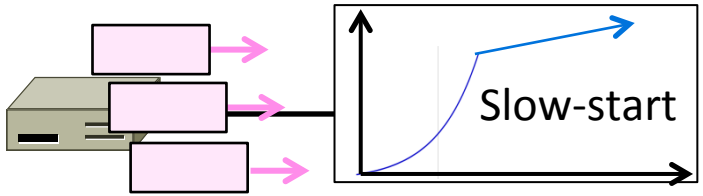
- The network has smoothed out the burst of data segments
- ACK clock transfers this smooth timing back to the sender
- Subsequent data segments are not sent in bursts so do not queue up in the network

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

Slow Start

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

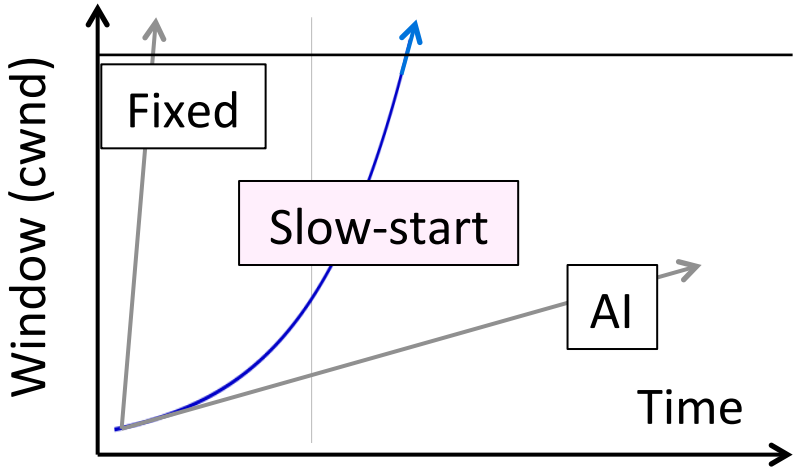


TCP Startup Problem

- 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



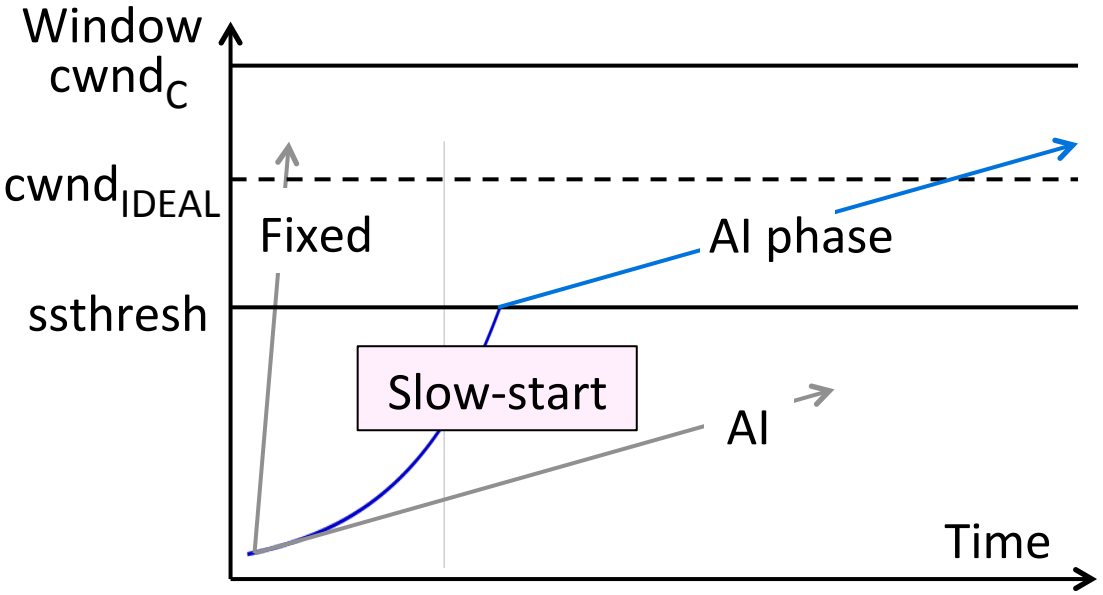
Slow-Start Solution (2)

- 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

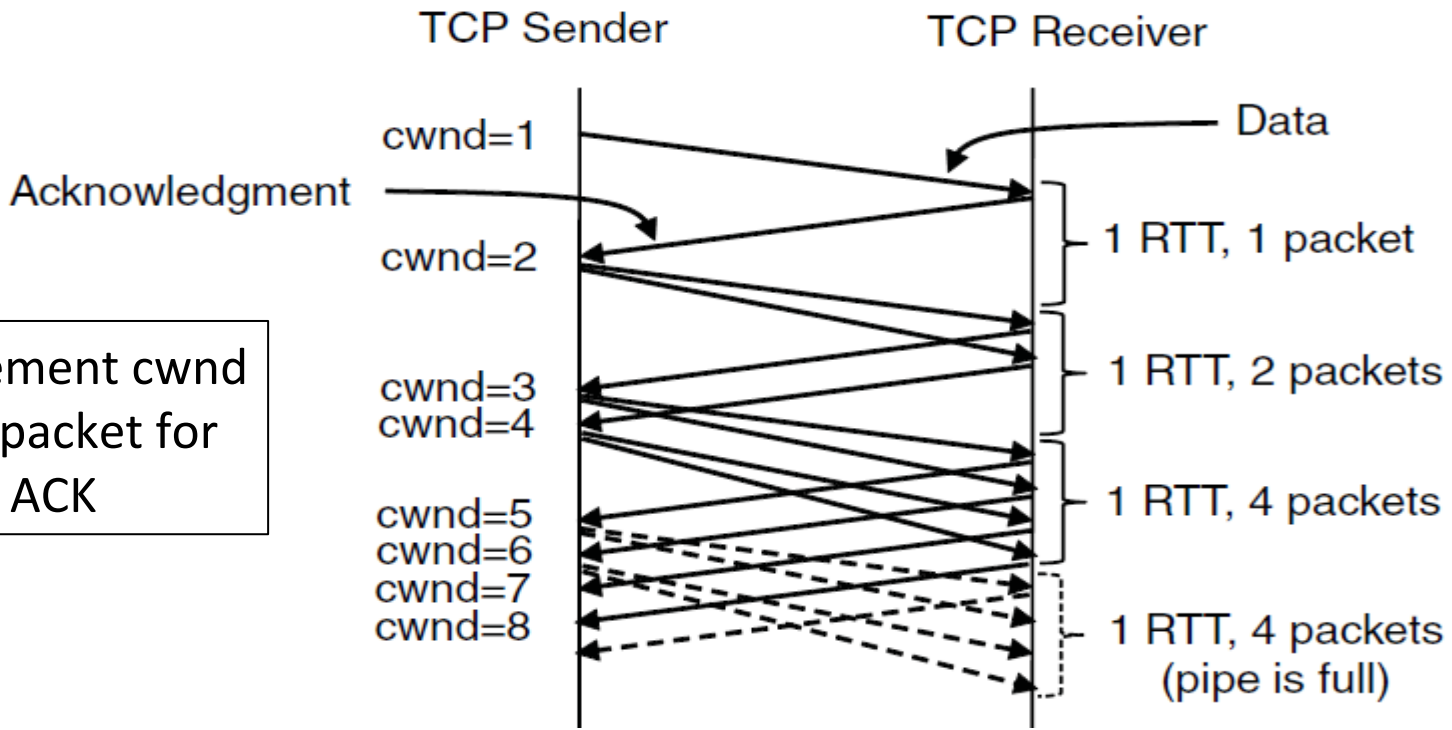
- In terms of cwnd:
  - Expect loss for $cwnd_C \approx 2BD+queue$
  - Use ssthresh = $cwnd_C/2$ to switch to AI

Slow-Start Solution (3)

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

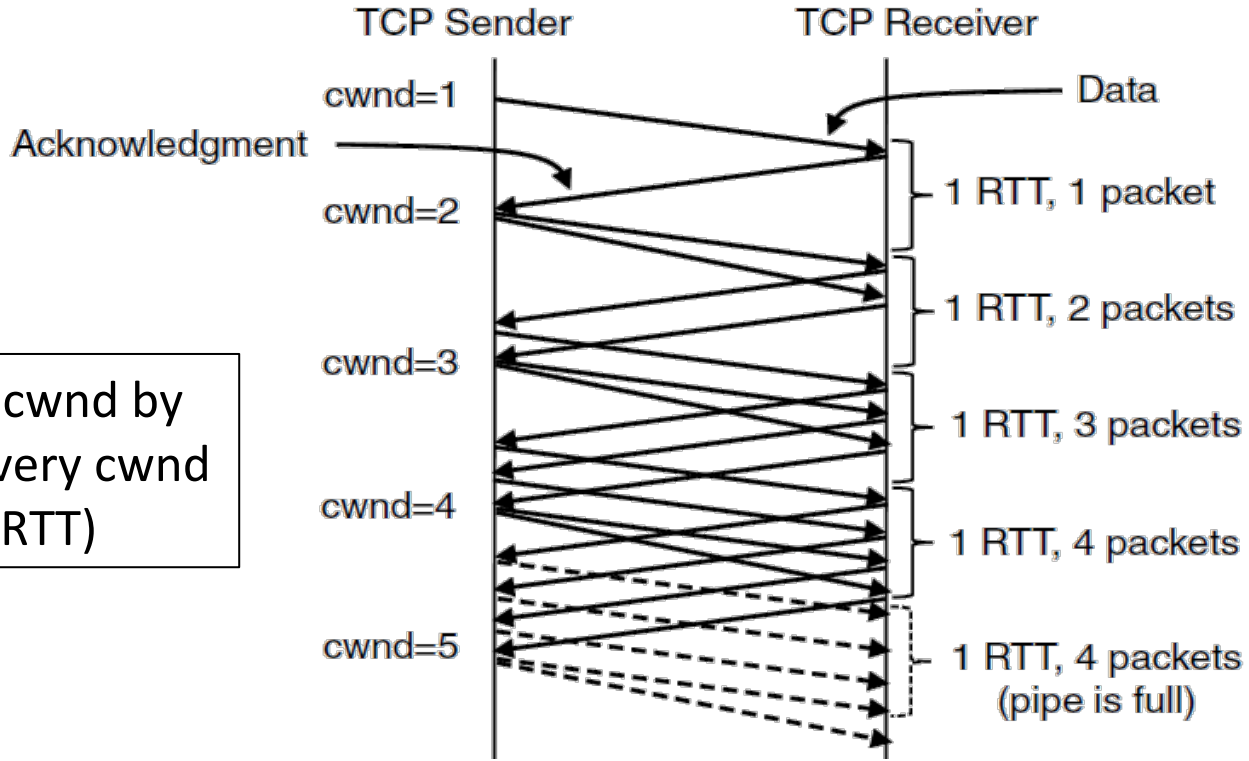


Slow-Start (Doubling) Timeline



Increment cwnd by 1 packet for each ACK

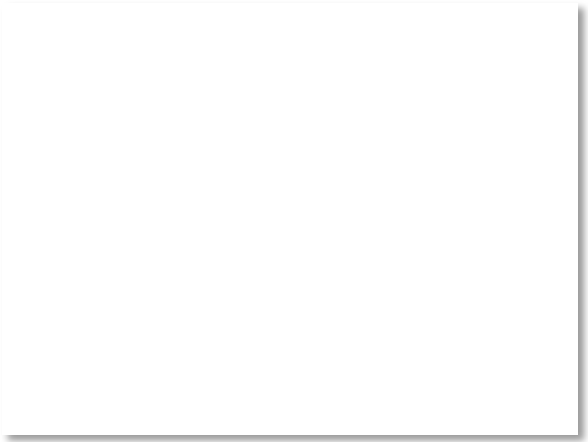
Additive Increase Timeline



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

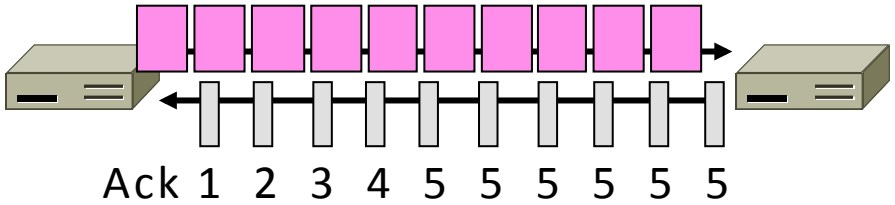
- Why do a slow-start after timeout?
  - Instead of MD cwnd (for AIMD)
- Timeouts are sufficiently long that the ACK clock will have run down
  - Slow-start ramps up the ACK clock
- We need to detect loss before a timeout to get to full AIMD
  - Done in TCP Reno (next time)

Inferring Loss from ACKs

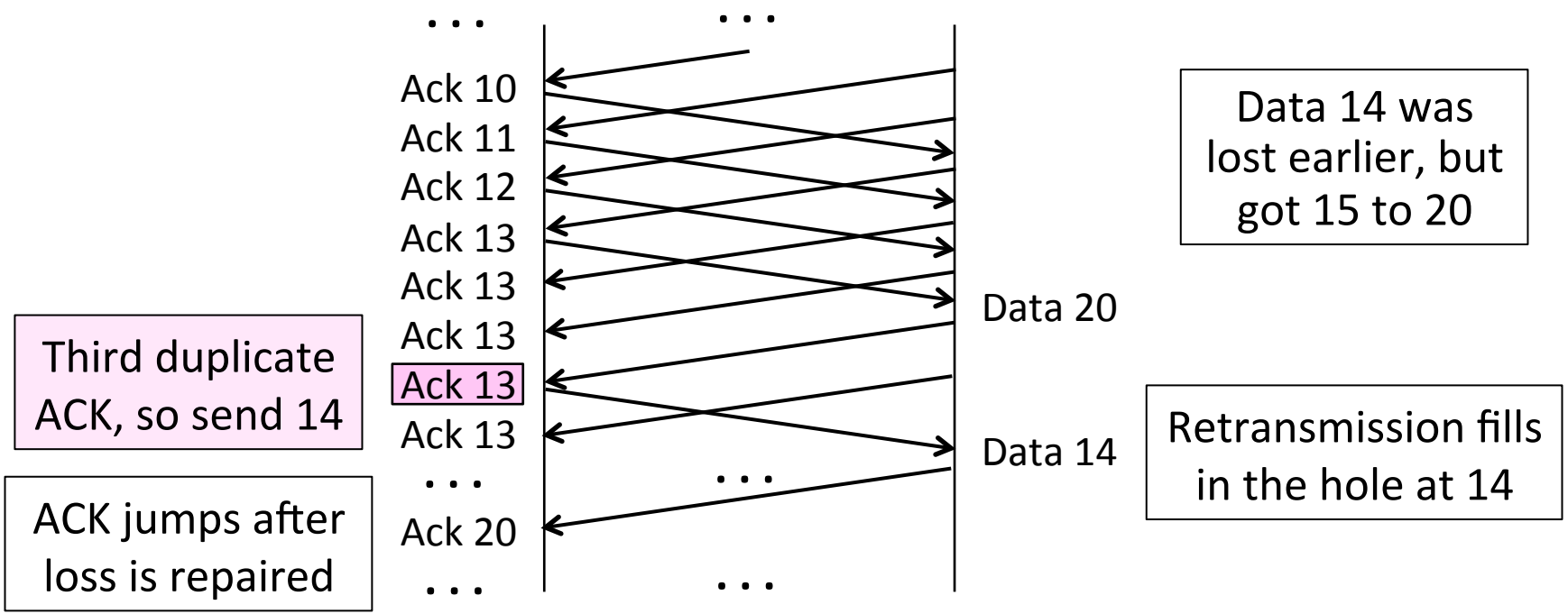
- TCP uses a cumulative ACK
  - Carries highest in-order seq. number
  - Normally a steady advance
- Duplicate ACKs give us hints about what data hasn't arrived
  - Tell us some new data did arrive, but it was not next segment
  - Thus the next segment may be lost

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)

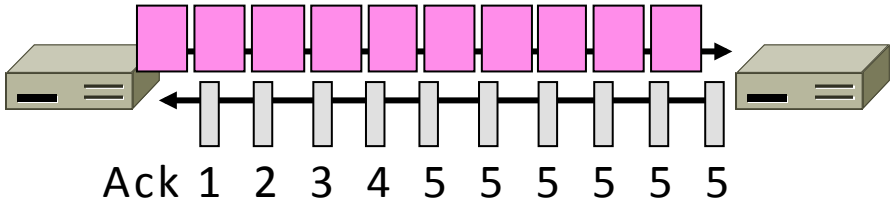


Fast Retransmit (3)

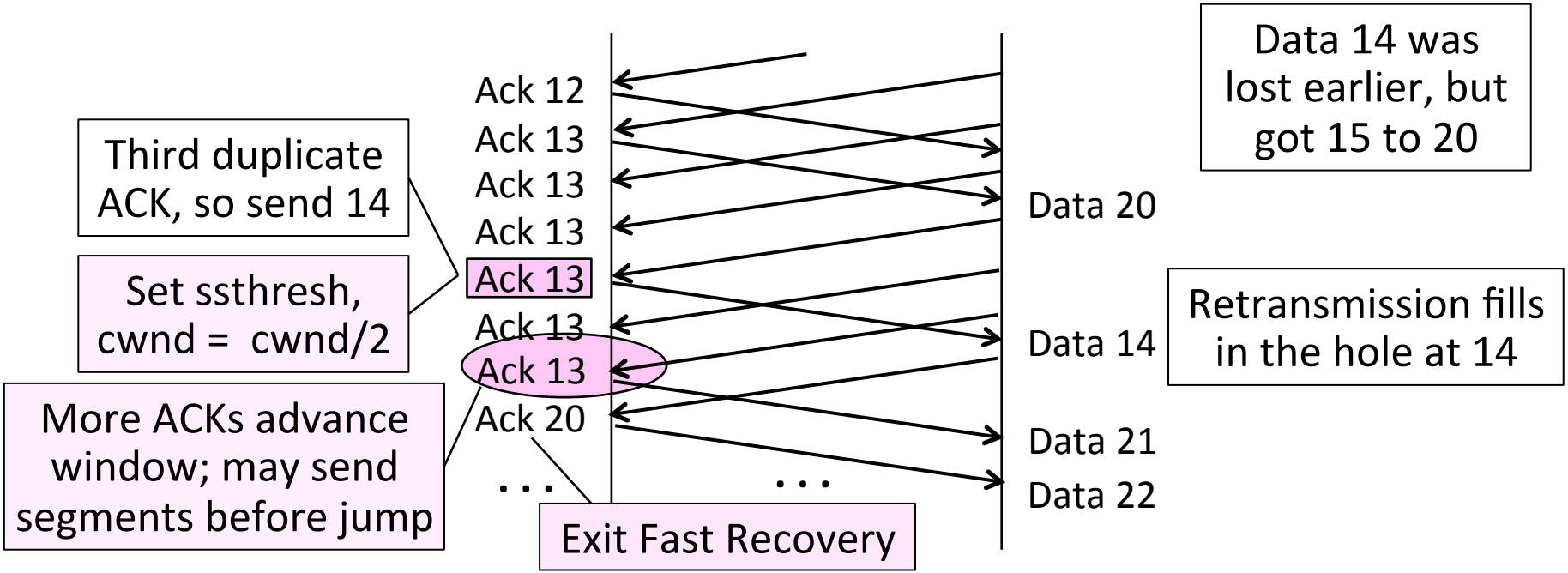
- It can repair single segment loss quickly, typically before a timeout
- However, we have quiet time at the sender/receiver while waiting for the ACK to jump
- And we still need to MD cwnd …

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



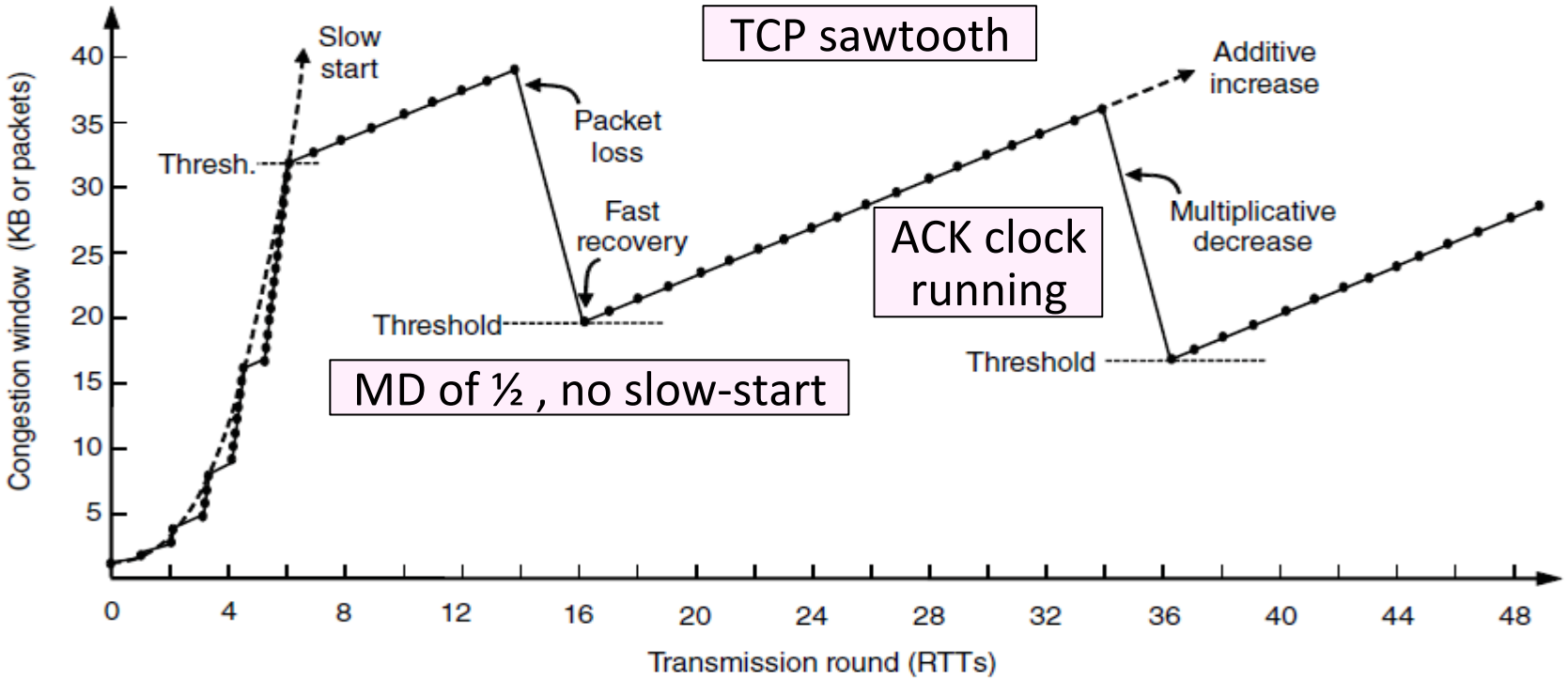
Fast Recovery (2)



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

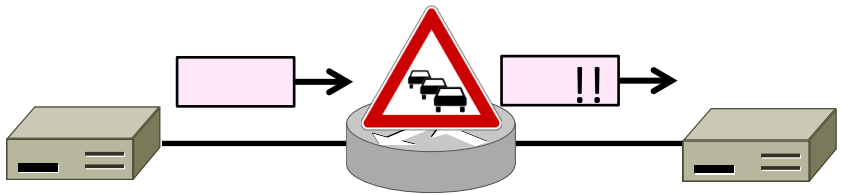

TCP sawtooth
Slow start
Thresh.
Packet loss
Fast recovery
Threshold
MD of ½ , no slow-start
Additive increase
ACK clock running
Multiplicative decrease
Threshold
Congestion window (KB or packets)
Transmission round (RTTs)
0
4
8
12
16
20
24
28
32
36
40
44
48
5
10
15
20
25
30
35
40

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

Topic

- 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?

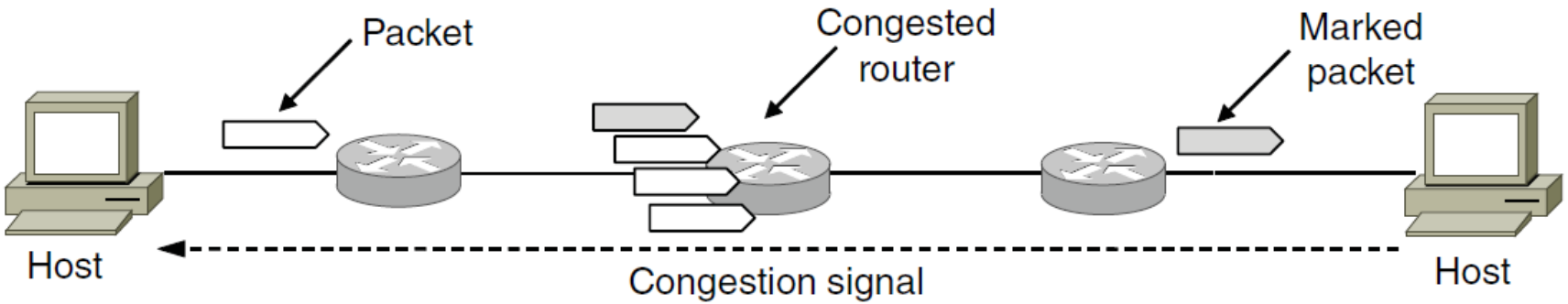
Feedback Signals

- 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

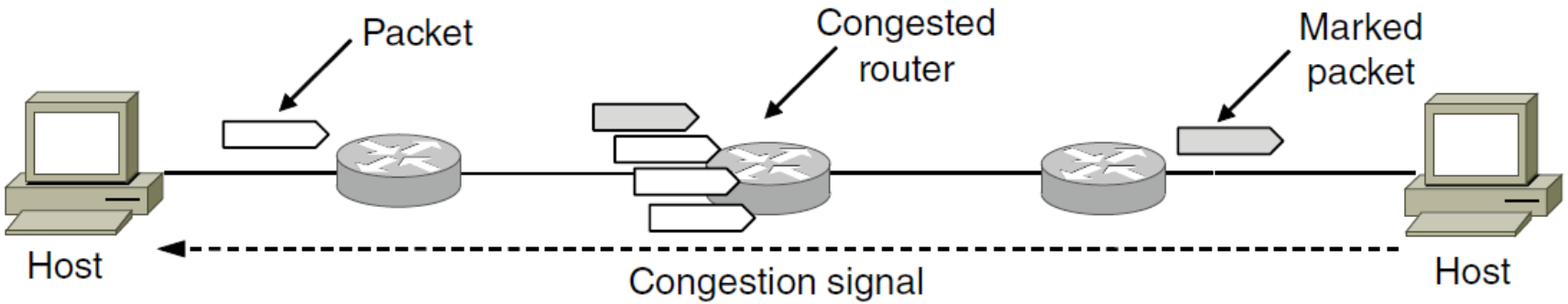
ECN (Explicit Congestion Notification)

- 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
- Disadvantages:
  - Routers and hosts must be upgraded