Recap

- TCP connection setup/teardown
- Sliding window, flow control
- Retransmission timeouts
- Fairness, max-min fairness
- AIMD achieves max-min fairness

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

TCP Tahoe/Reno

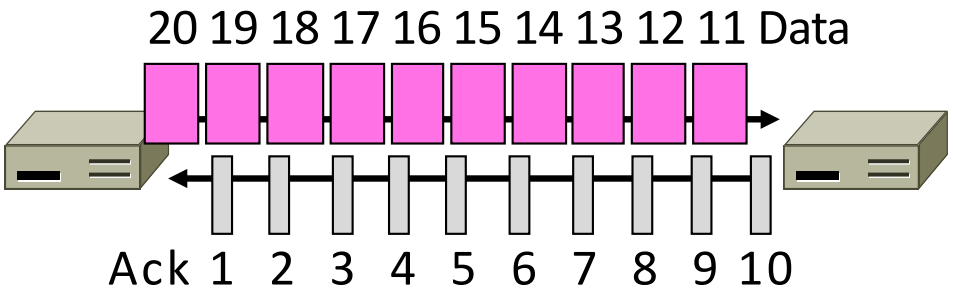
- Avoid congestion collapse without changing routers (or even receivers)
- Idea is to fix timeouts and introduce a congestion window (cwnd) over the sliding window to limit queues/loss
- TCP Tahoe/Reno implements AIMD by adapting cwnd using packet loss as the network feedback signal

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

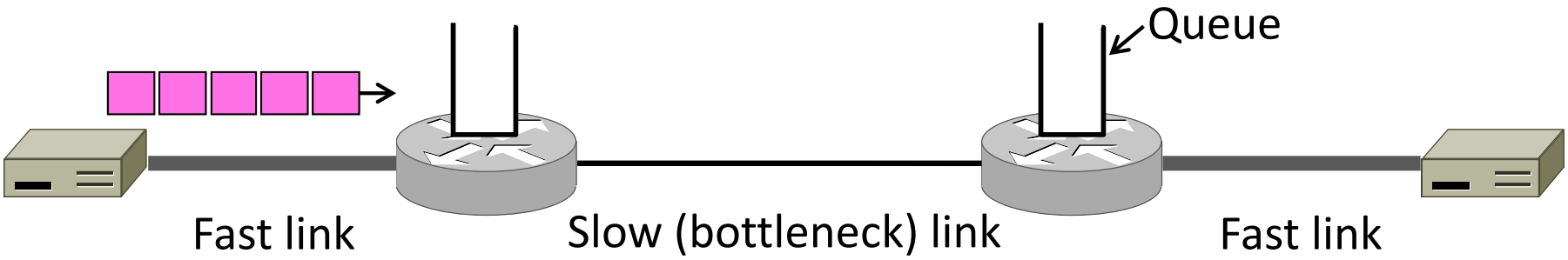
Sliding Window ACK Clock

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



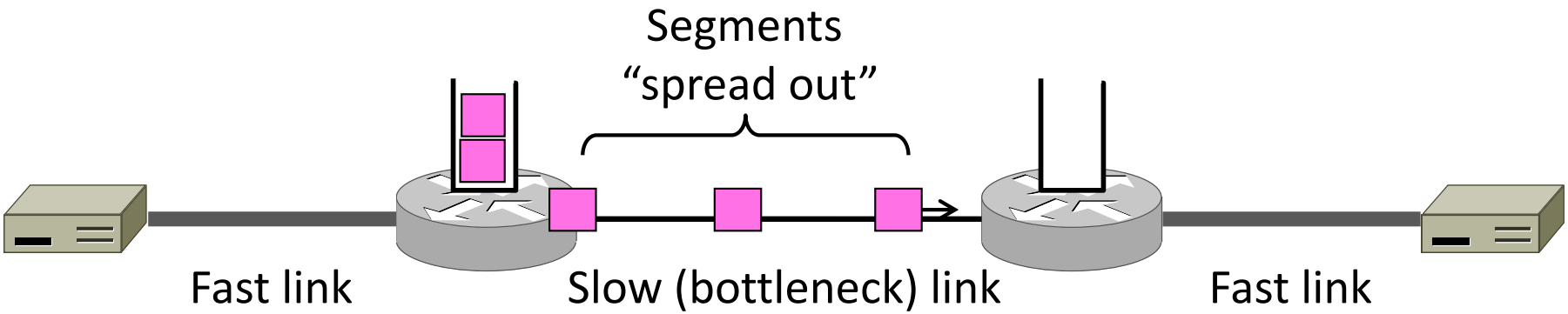
Benefit of ACK Clocking

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



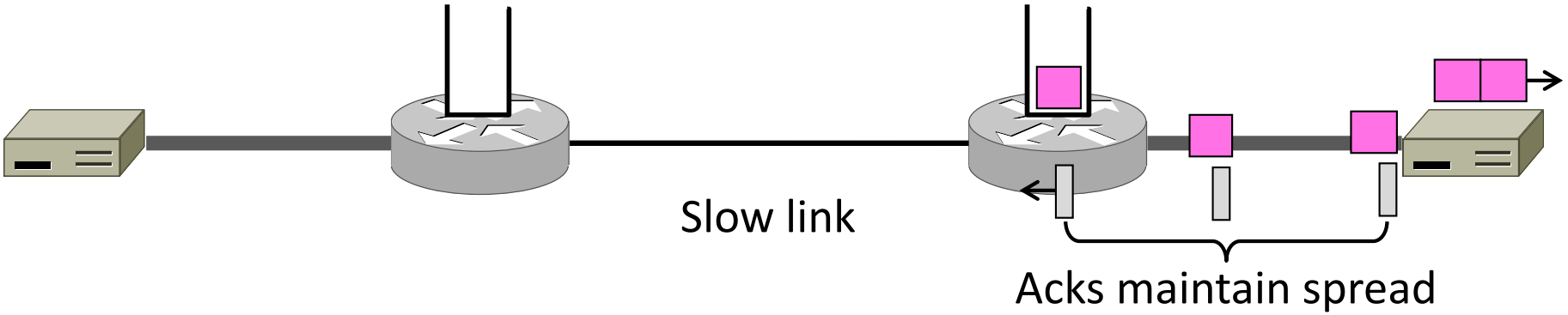
Benefit of ACK Clocking (2)

- Segments are buffered and spread out on slow link



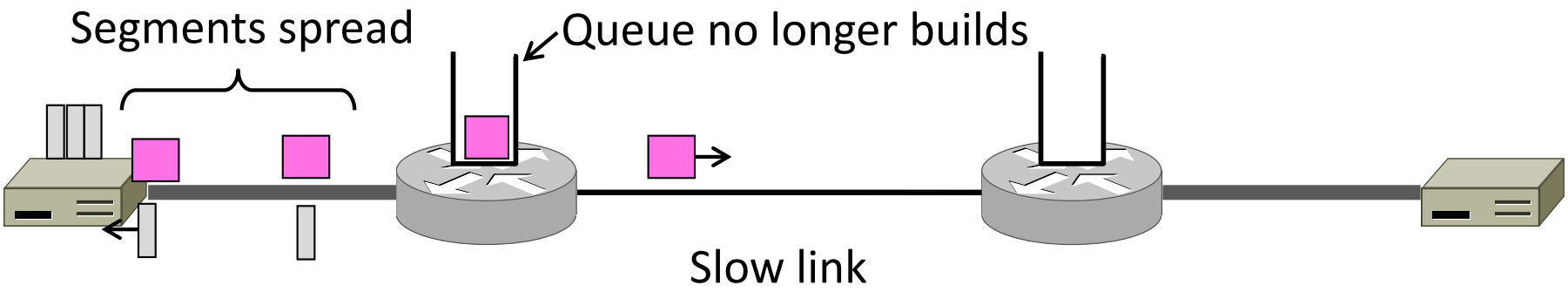
Benefit of ACK Clocking (3)

- 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!

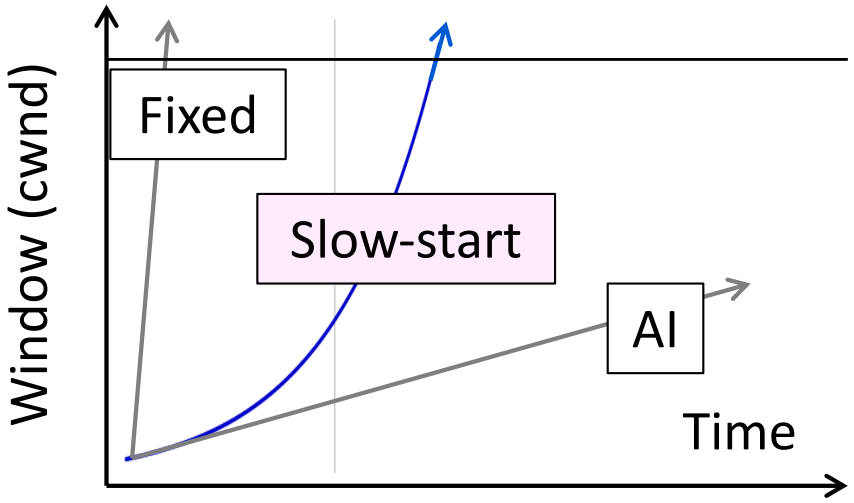
- The network has smoothed out the burst of data segments
- ACK clock transfers this smooth timing back to the sender
- Subsequent data segments are not sent in bursts so do not queue up in the network

TCP 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)

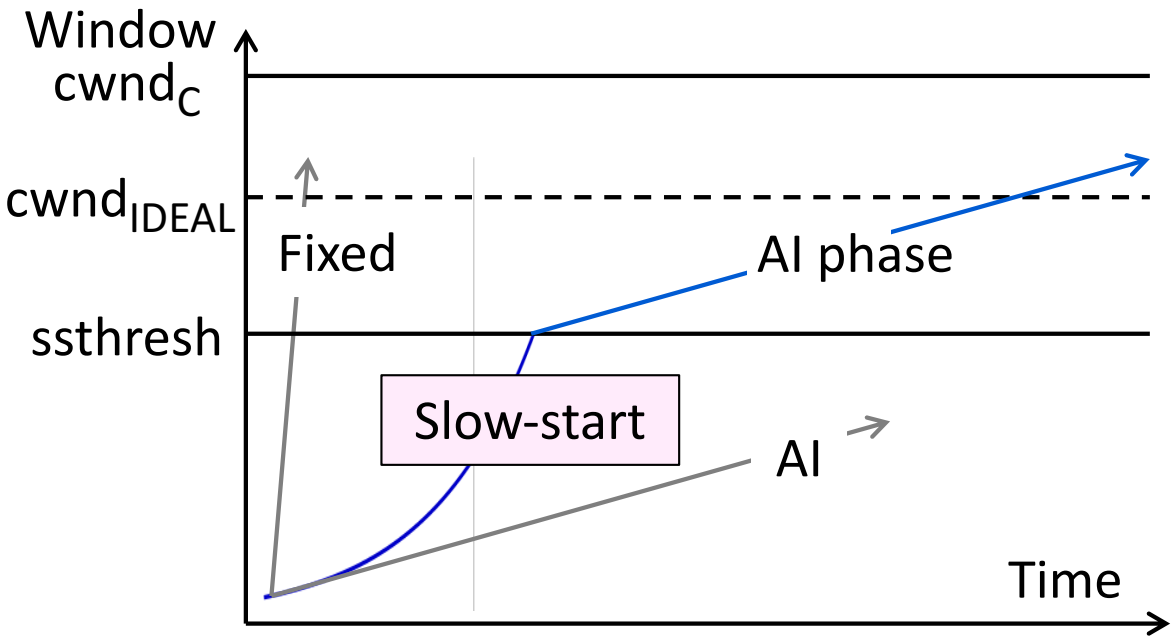
- Eventually packet loss will occur when the network is congested
  - Loss timeout tells us cwnd is too large
  - Next time, switch to AI beforehand
  - Slowly adapt cwnd near right value

- Question: what is the cwnd at which packet loss will happen during slow start?

Extent of subtitles

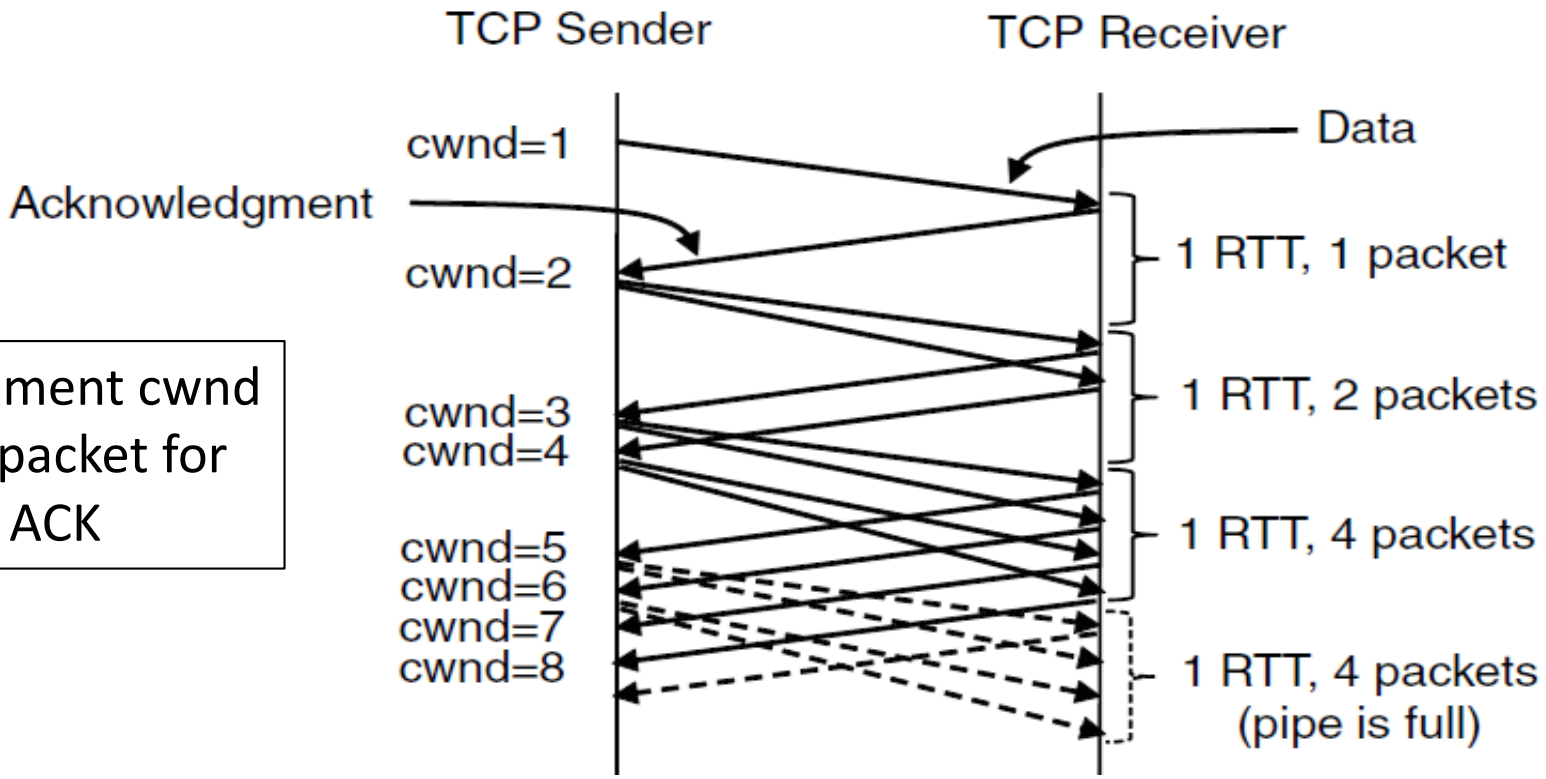
Slow-Start Solution (3)

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

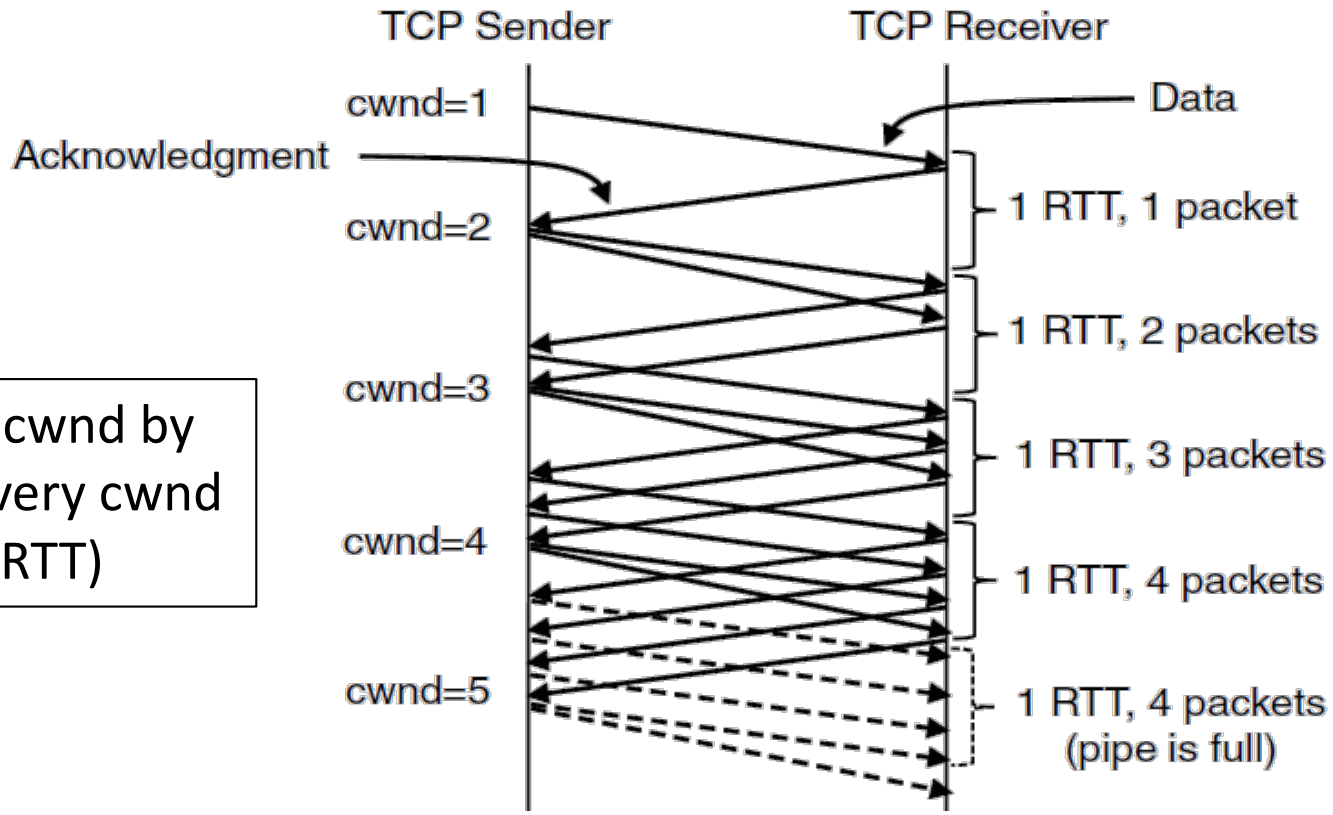


Slow-Start (Doubling) Timeline

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

- How can we improve on TCP Tahoe?

Extent of subtitles

Timeout Misfortunes

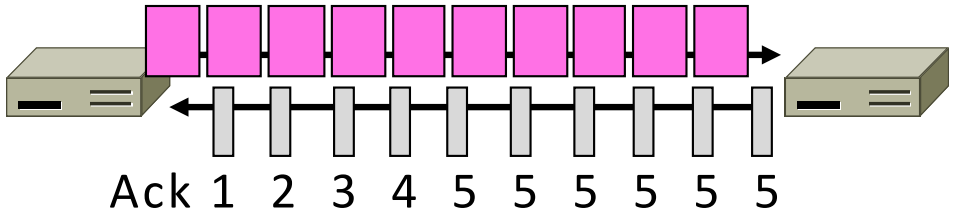
- Why do a slow-start after timeout?
  - Instead of MD cwnd (for AIMD)
- Timeouts are sufficiently long that the ACK clock will have run down
  - Slow-start ramps up the ACK clock
- We need to detect loss before a timeout to get to full AIMD
  - Done in TCP Reno

Inferring Loss from ACKs

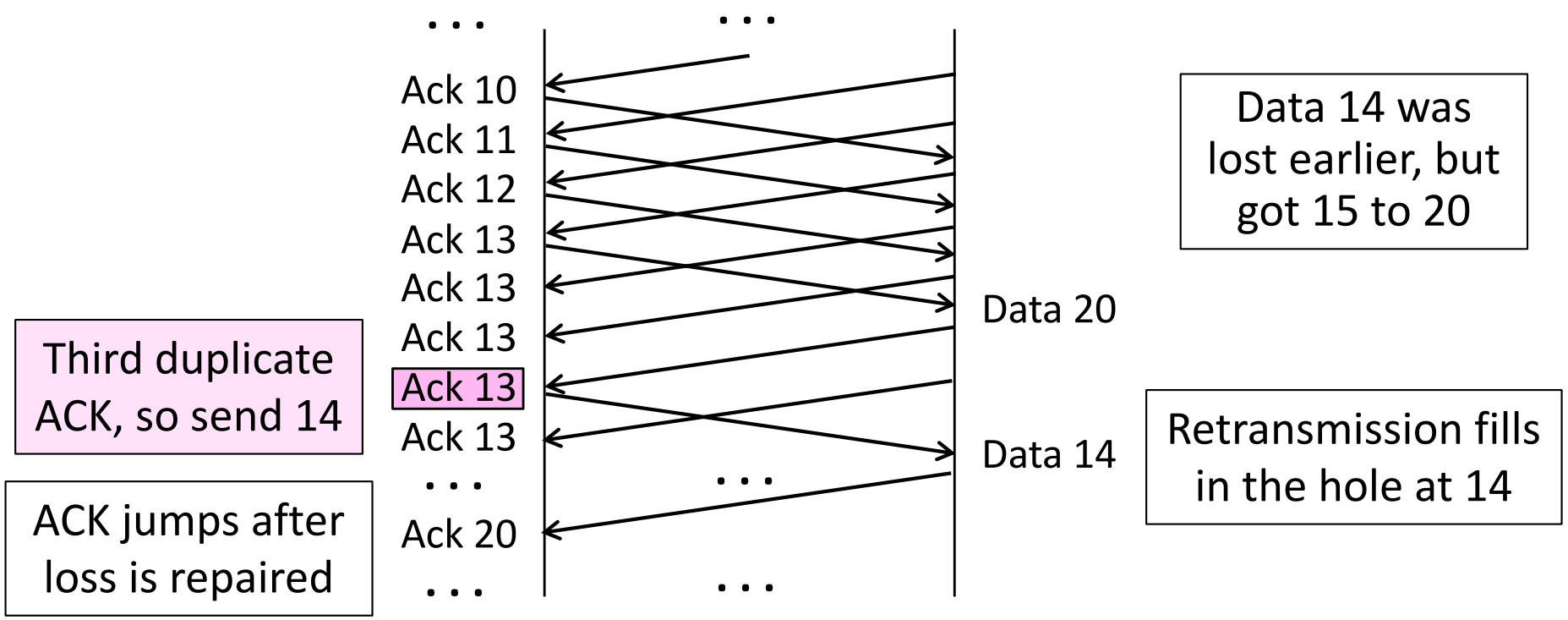
- TCP uses a cumulative ACK
  - Carries highest in-order seq. number
  - Normally a steady advance
- Duplicate ACKs give us hints about what data hasn't arrived
  - Tell us some new data did arrive, but it was not next segment
  - Thus the next segment may be lost

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)

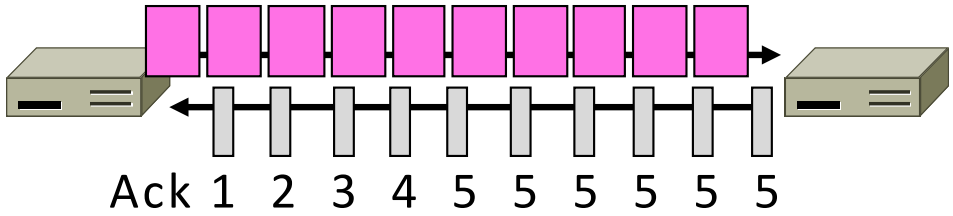
- It can repair single segment loss quickly, typically before a timeout
- However, we have quiet time at the sender/receiver while waiting for the ACK to jump
- And we still need to MD cwnd …

Inferring Non-Loss from ACKs

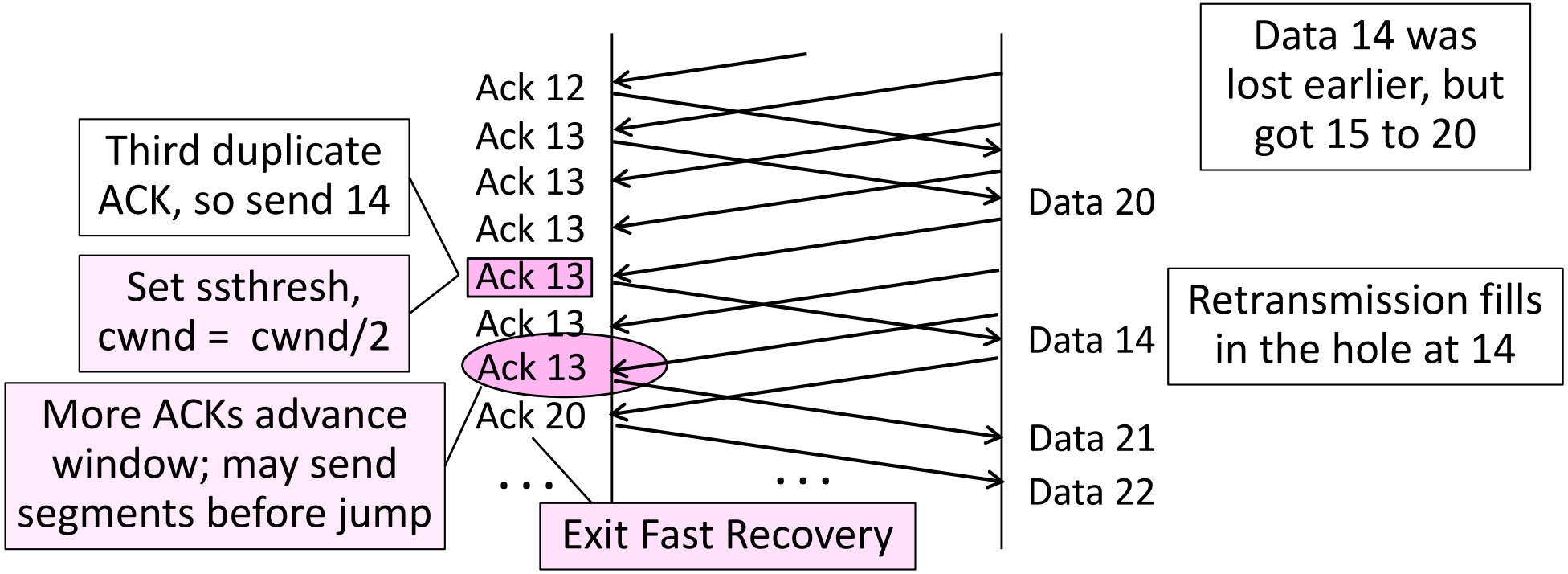
- Duplicate ACKs also give us hints about what data has arrived
  - Each new duplicate ACK means that some new segment has arrived
  - It will be the segments after the loss
  - Thus advancing the sliding window will not increase the number of segments stored in the network

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



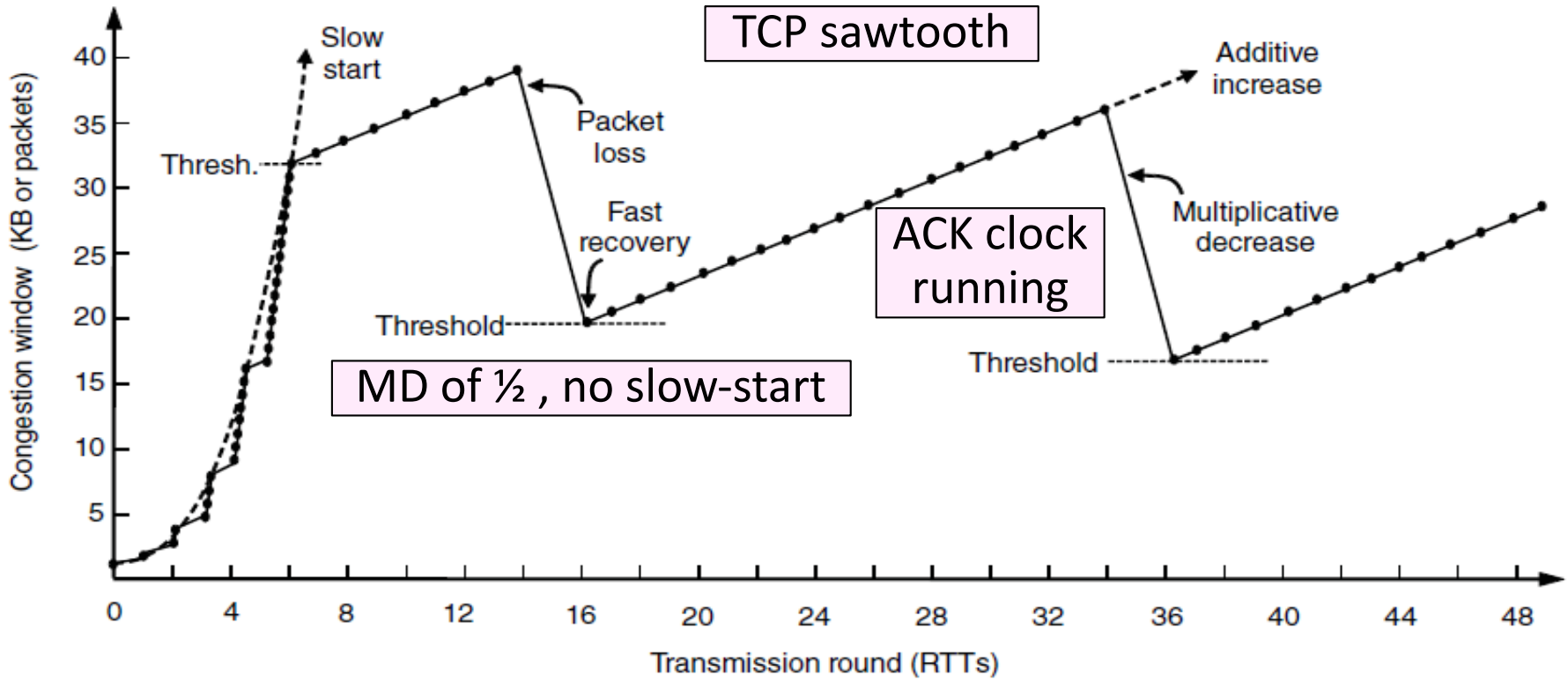
Fast Recovery (2)



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
Thresh.
Packet loss
Fast recovery
Threshold
MD of ½ , no slow-start
Additive increase
Multiplicative decrease
ACK clock running
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

- Check out simulation at:
  - http://guido.appenzeller.net/anims/
  - Or: goo.gl/sqmGWp

Extent of subtitles

Computer Networks

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
- Question: how can we do this with router support?

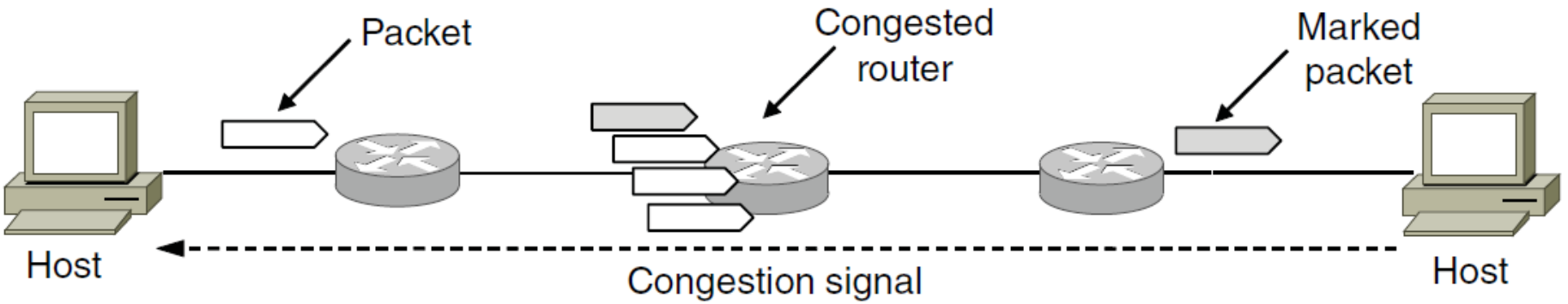
Feedback Signals

- 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

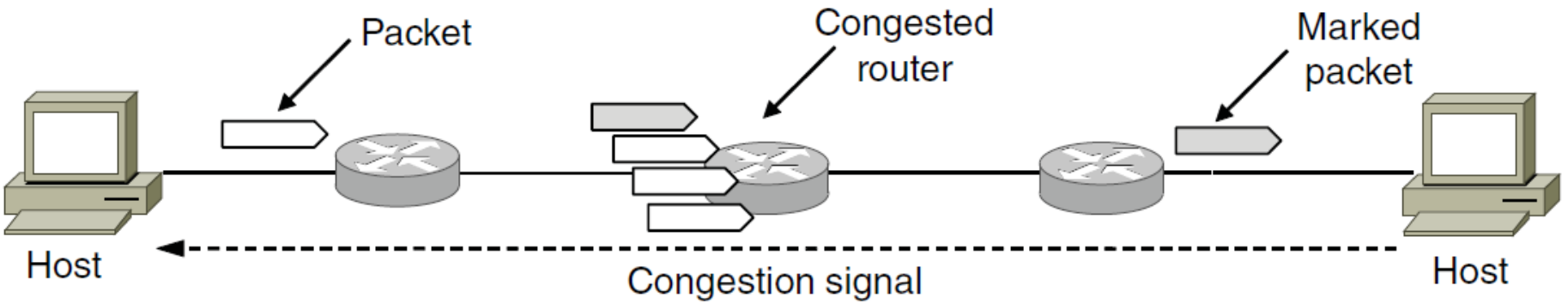
ECN (Explicit Congestion Notification)

- 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

TCP Variants

- There are many different strains of TCP including:
  - Loss-based congestion control: Reno, BIC, Cubic
  - Delay-based congestion control: Vegas, Veno, Westwood
  - High-speed congestion control: Scalable, HighSpeed, HTCP

Delay Based Congestion Control

- Basic idea:
  - Before packet loss occurs, detect the early stage of congestion in the routers between source and destination
  - Additively decrease the sending rate when incipient congestion is detected

TCP Vegas

- *Expected = cwnd/BaseRTT*
- *Actual = cwnd/RTT*
- *DIFF = (Expected-Actual)*

if *(DIFF*BaseRTT* < α *)*

cwnd = cwnd + 1

else if *(DIFF*BaseRTT* > β *)*

cwnd = cwnd – 1

else *cwnd = cwnd*

BaseRTT: the minimum of all measured *RTT*

RTT: the actual round-trip time of a tagged packet

α and β are constant values
that are set by experimentation

TCP Vegas

- Modified Slow Start
  - Try to find the correct window size without incurring a loss
  - exponentially increasing its window every *other* RTT and use the other RTT to calculate *DIFF*
  - As soon as Vegas detects queue buildup during slow start, it transitions to congestion avoidance

Cubic

- Two key modifications:
  - Cubic window growth with inflection point at congestion window at previous loss

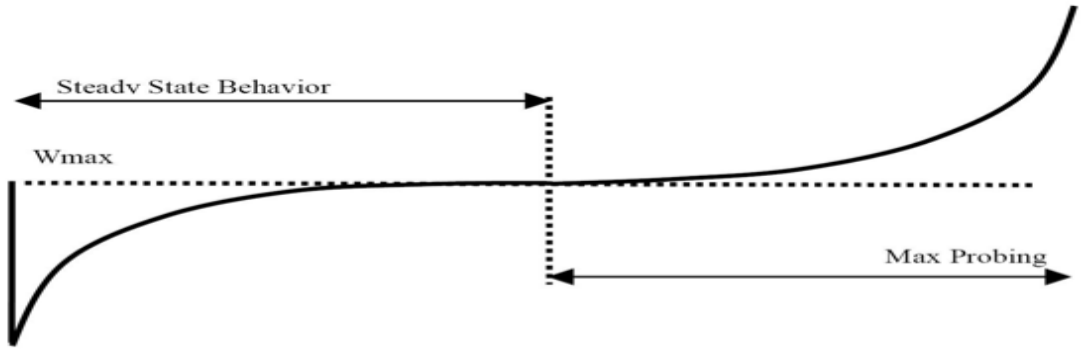


Fig. 2: The Window Growth Function of CUBIC

  - Safe exit for slow start (i.e., transition from exponential growth to linear growth)

Multipath

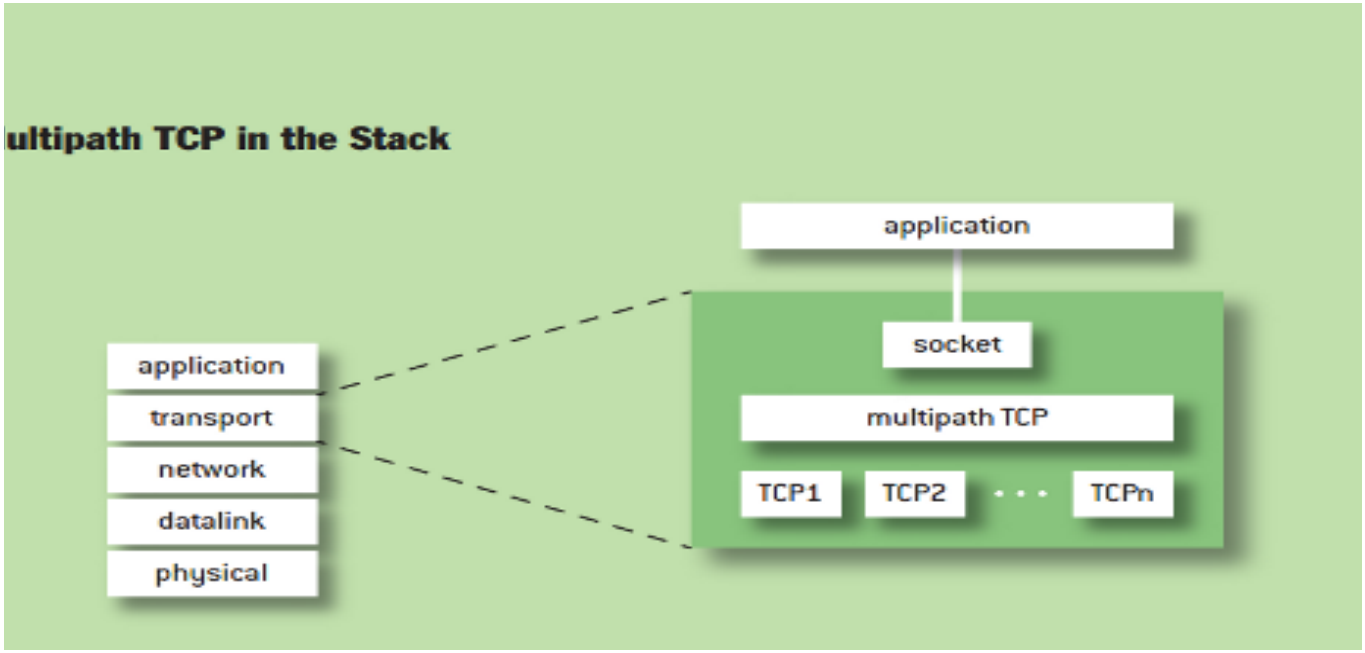
- Mobile user
  - WiFi and cellular at the same time
- High-end servers
  - Multiple Ethernet cards
- Data centers
  - Rich topologies with many paths

- Question: what are the benefits of multipath?

Multipath TCP Protocol

Working With Unmodified Apps

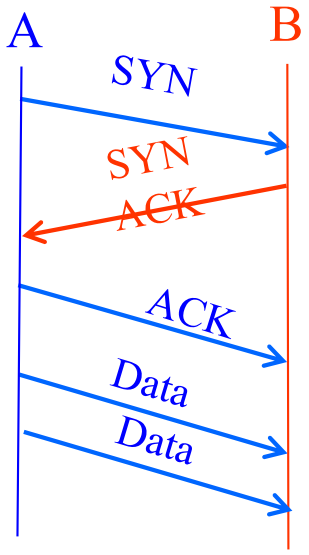
- Present the same socket API and expectations
  - Identified by the “five tuple” (IP address, port #, protocol)



From http://queue.acm.org/detail.cfm?id=2591369

Working With Unmodified Hosts

- Establish the TCP connection in the normal way
  - Create a socket to a single remote IP address/port

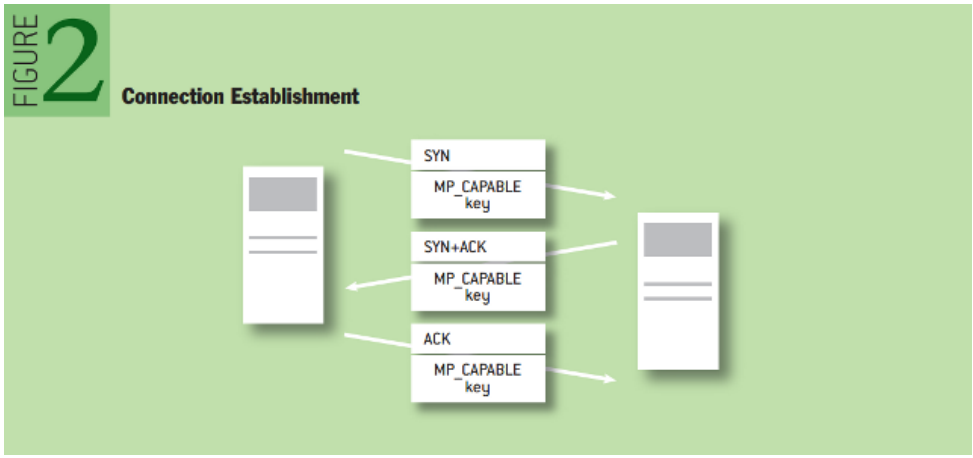


Each host tells its *Initial Sequence Number (ISN)* to the other host.

- And then add more subflows, if possible

Negotiating MPTCP Capability

- How do hosts know they both speak MPTCP?
  - During the 3-way SYN/SYN-ACK/ACK handshake



- If SYN-ACK doesn't contain MP_CAPABLE
  - Don't try to add any subflows!

Adding Subflows, Idealized

- How to associate a new subflow with the connection?
  - Use a token generated from original subflow set-up
- How to start using the new subflow?
  - Simply start sending packets with new IP/port pairs
  - … and associate them with the existing connection
- How could two end-points learn about extra IP addresses for establishing new subflows?
  - Implicitly: one end-point establishes a new subflow, to already-known address(es) at the other end-point

Sequence Numbers

- Challenges across subflows
  - Out-of-order packets due to RTT differences
  - Access networks that rewrite sequence numbers
  - Middleboxes upset by discontinuous TCP byte stream
  - Need to retransmit lost packets on a different subflow

- Two levels of sequence numbers
  - Sequence numbers per subflow
  - Sequence numbers for the entire connection
- Enables
  - Efficient detection of loss on each subflow
  - Retransmission of lost packet on a different subflow

Receive Buffer Space

- Each TCP connection has a receive buffer
  - Buffer space to store incoming data
  - … until it is read by the application
- TCP flow control
  - Receiver advertises the available buffer space
  - … using the “receive window”
- Should each subflow have its own receive window?
  - Starvation of some subflows in a connection?
  - Fairness relative to other TCP connections?
  - Fragmentation of the available buffer space?
- Instead, use a common receive window

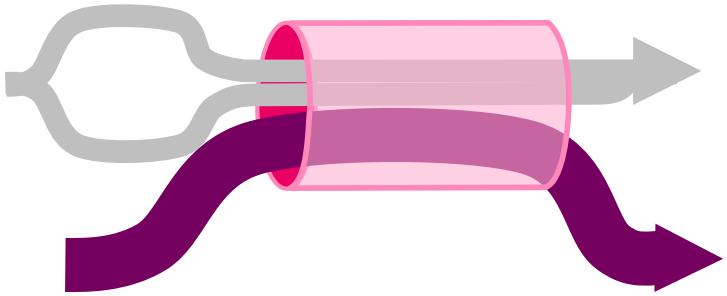
Fairness and Efficiency in Multipath Congestion Control

Slides from Damon Wischik

Goal #1: Fairness at Shared Bottlenecks

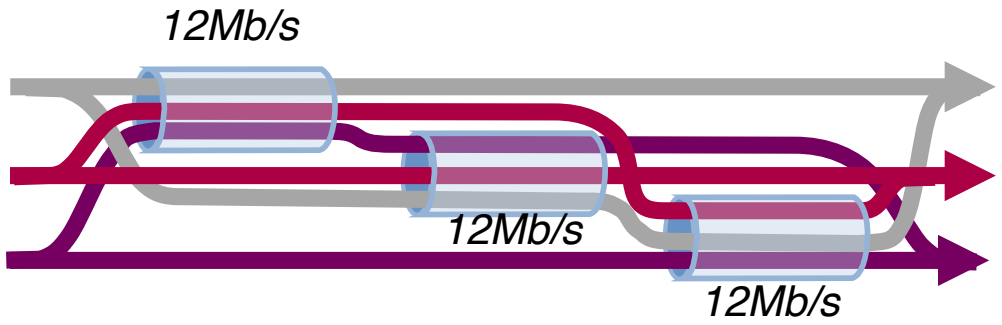
A multipath TCP flow with two subflows

Regular TCP



To be fair, Multipath TCP should take as much capacity as TCP at a bottleneck link, no matter how many paths it is using.

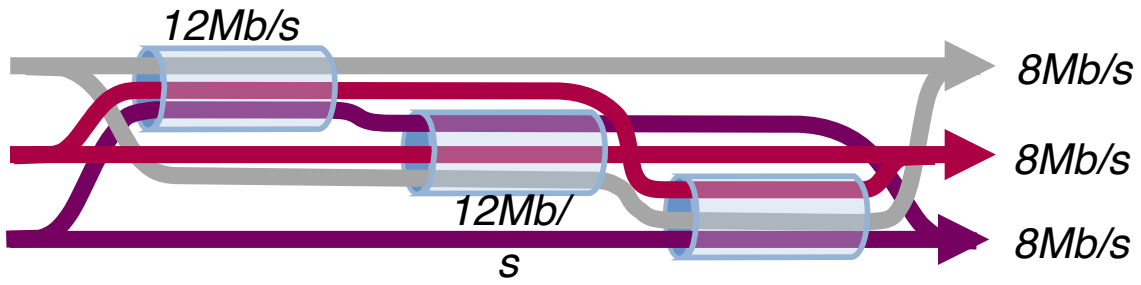
Goal #2: Use Efficient Paths



Each flow has a choice of a 1-hop and a 2-hop path.

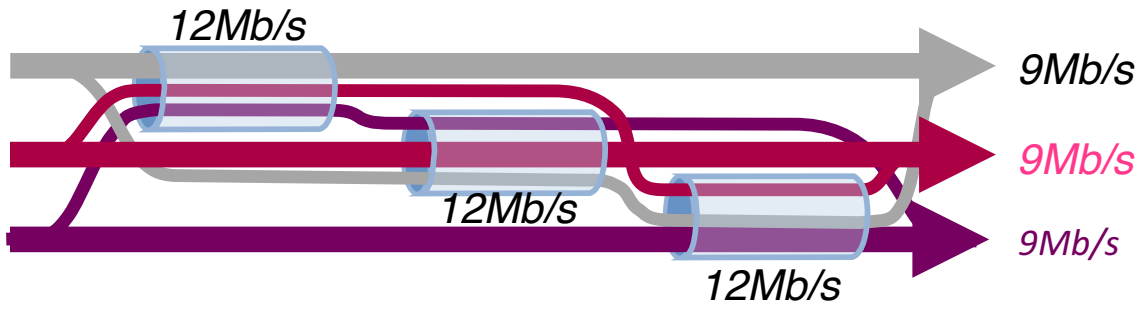
How should split its traffic?

Use Efficient Paths



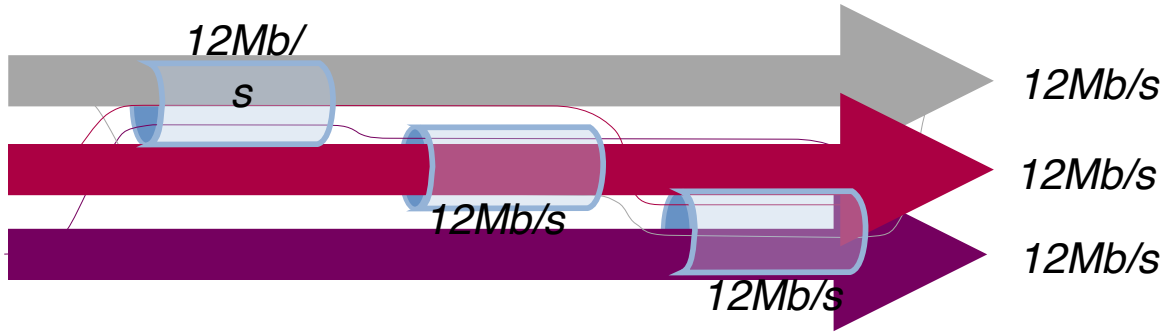
If each flow split its traffic 1:1 ...

Use Efficient Paths



If each flow split its traffic 2:1 …

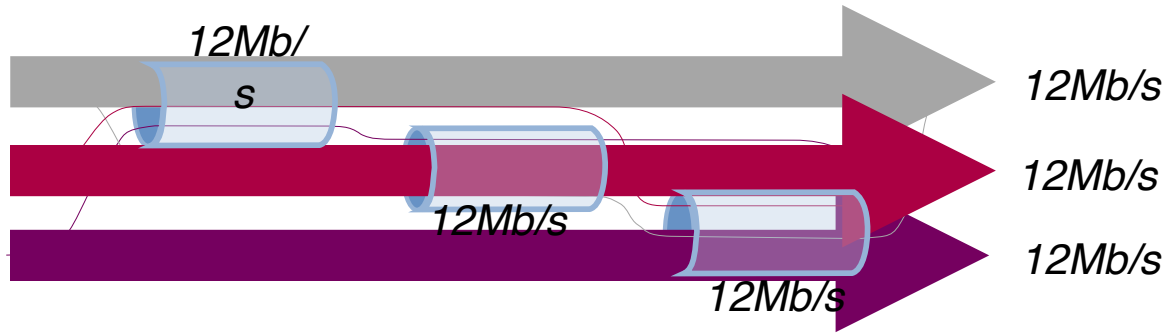
Use Efficient Paths



Better: Each connection on a one-hop path

Each connection should send all traffic on the least-congested paths

Use Efficient Paths



Better: Each connection on a one-hop path

Each connection should send all traffic on the least-congested paths

But keep some traffic on the alternate paths as a probe

Goal #3: Be Fair Compared to TCP

- Least-congested paths may not be best!
  - Due to differences in round-trip time
- Two paths
  - WiFi: high loss, low RTT
  - Cellular: low loss, high RTT
- Using the least-congested path
  - Choose the cellular path, due to low loss
  - But, the RTT is high
  - So throughput is low!

Be Fair Compared to TCP

- *To be fair, Multipath TCP should give a connection at least as much throughput as it would get with a single-path TCP on the best of its paths.*
  - *Ensure incentive for deploying MPTCP*

- *A Multipath TCP should take no more capacity on any path (or collection of paths) than if it was a single-path TCP flow using the best of those paths.*
  - *Do no harm!*

Achieving These Goals

- Regular TCP
  - Maintain a congestion window w
  - On an ACK, increase by 1/w (increase 1 per window)
  - On a loss, decrease by w/2

- MPTCP
  - Maintain a congestion window per path w_r
  - On an ACK on path r, increase w_r
  - On a loss on path r, decrease by $w_r/2$

- How much to increase w_r on an ACK??
  - If r is the only path at that bottleneck, increase by $1/w_r$

If Multiple Paths Share Bottleneck?

- Don't take any more bandwidth on a link than the best of the TCP paths would
  - But, where might the bottlenecks be?
  - Multiple paths might share the *same* bottleneck

- So, consider all possible *subsets* of the paths
  - Set R of paths
  - Subset S of R that includes path r
- E.g., consider path 3
  - Suppose paths 1, 3, and 4 share a bottleneck
  - … but, path 2 does not
  - Then, we care about S = {1,3,4}

Achieving These Goals

- What is the *best* of these subflows achieving?
  - Path s is achieving throughput of w_s/RTT_s
  - So best path is getting $\max_s(w_s/RTT_s)$
- What *total* bandwidth are these subflows getting?
  - Across *all* subflows sharing that bottleneck
  - Sum over s in S of w_s/RTT_s
- Consider the *ratio* of the two
  - Increase by less if many subflows are sharing
- And pick the results for the set S with min ratio
  - To account for the *most* paths sharing a bottleneck