Slow Start (TCP Additive Increase)

TCP congestion control overview

Sender uses <u>congestion window (cwnd)</u>
- Sending rate ($\approx$cwnd/RTT)

Sender uses loss as network congestion signal

Follow AIMD control law for a good allocation
- Goal is efficient and (roughly) fair allocation
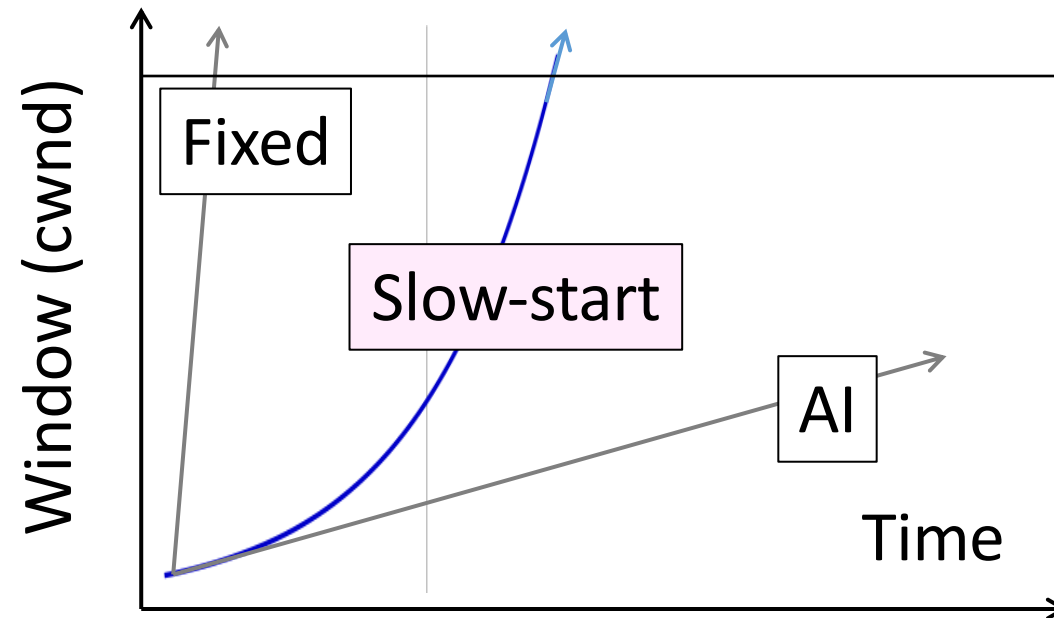
TCP "Slow Start" Problem

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 (2)

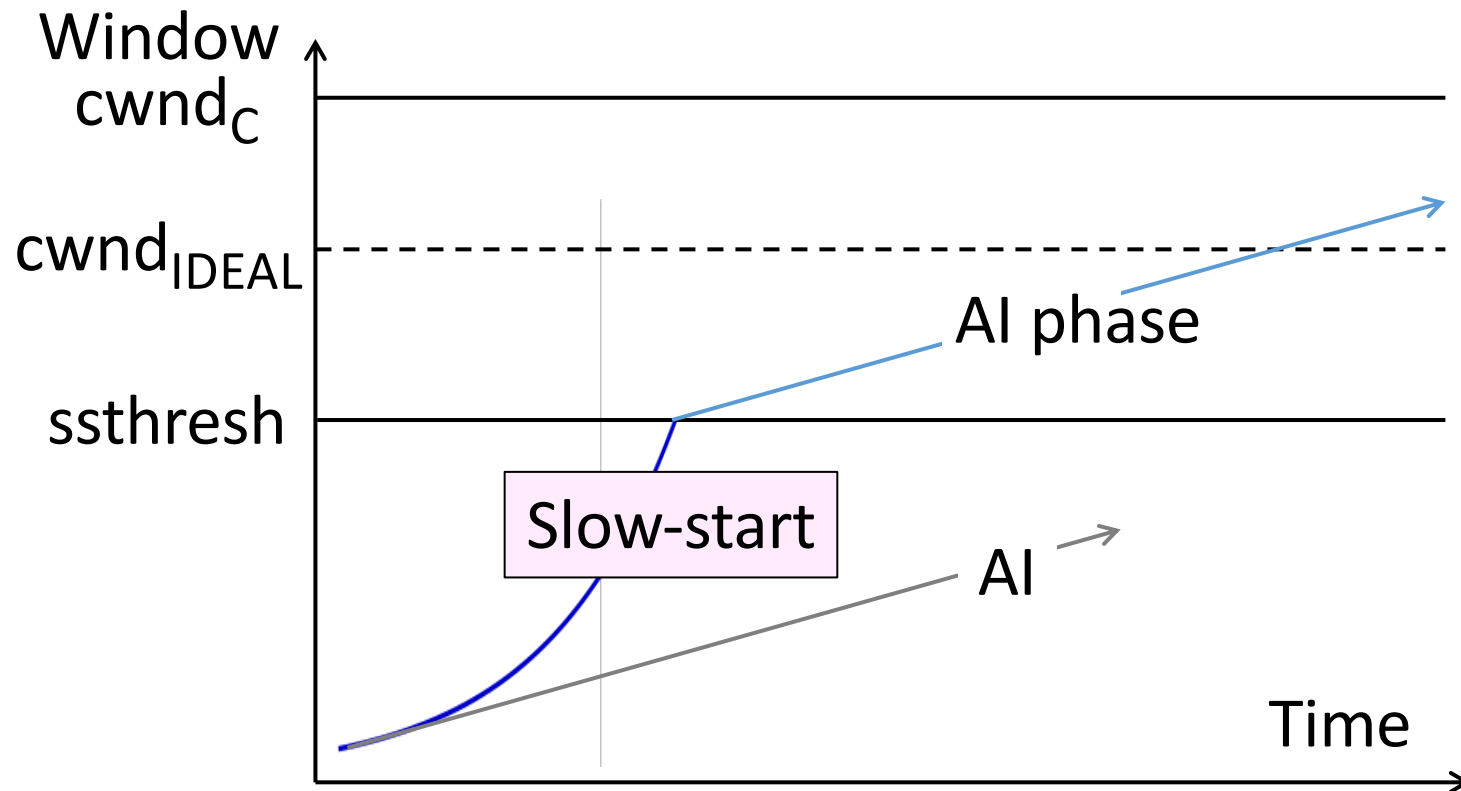Eventually packet loss will occur when the network is congested

- 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
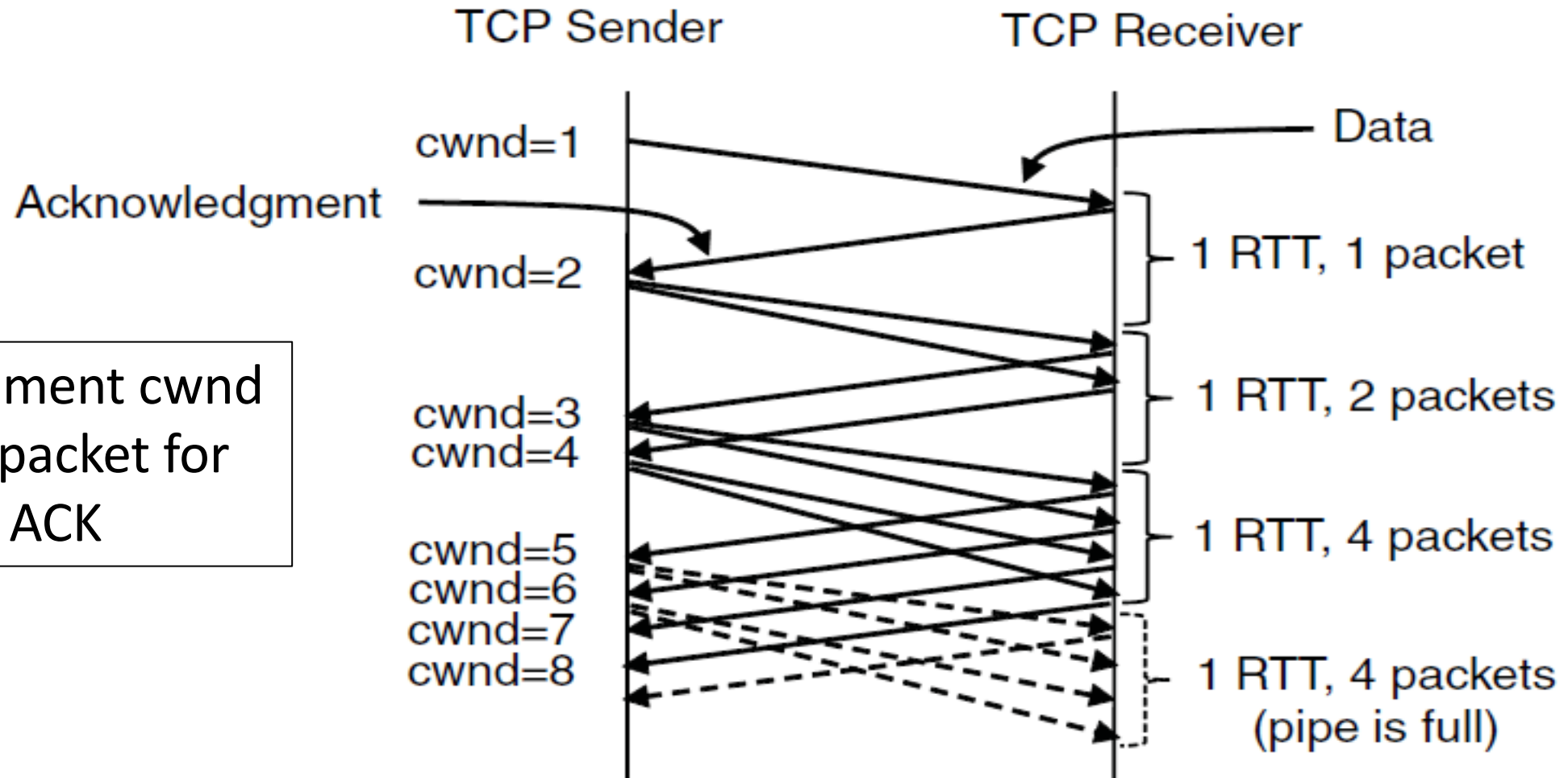
Slow-Start Solution (3)

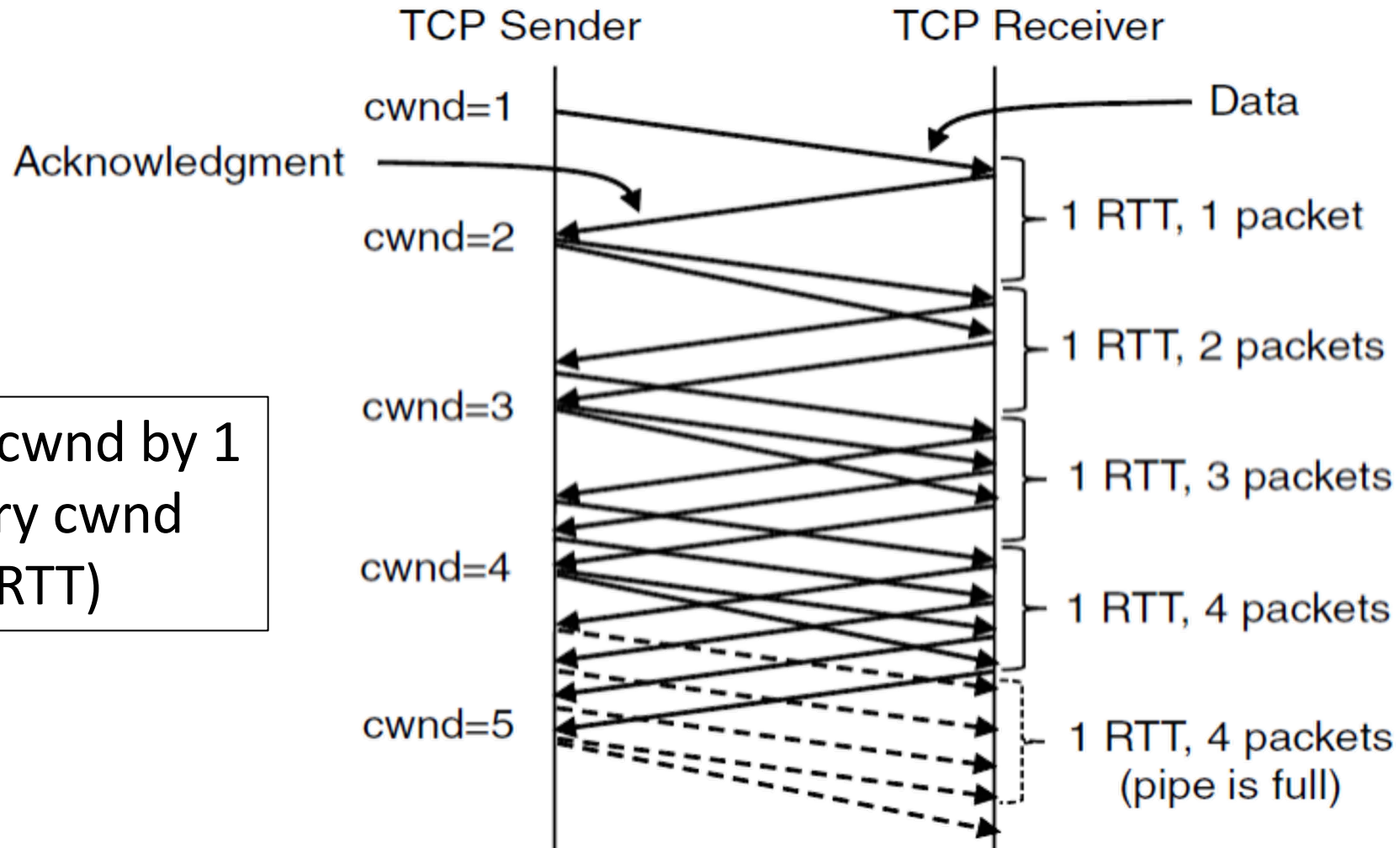- Combined behavior, **after first time**
  - Most time spent near right value

Slow-Start (Doubling) Timeline

Increment cwnd by 1 packet for each ACK

TCP Sender   TCP Receiver

Data
cwnd=1
Acknowledgment
cwnd=2   1 RTT, 1 packet

cwnd=3   1 RTT, 2 packets
cwnd=4

cwnd=5   1 RTT, 4 packets
cwnd=6
cwnd=7
cwnd=8   1 RTT, 4 packets
                           (pipe is full)

Additive Increase Timeline

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

TCP Sender   TCP Receiver

cwnd=1   Data
Acknowledgment
    1 RTT, 1 packet
cwnd=2
    1 RTT, 2 packets
cwnd=3
    1 RTT, 3 packets
cwnd=4
    1 RTT, 4 packets
cwnd=5
    1 RTT, 4 packets (pipe is full)

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
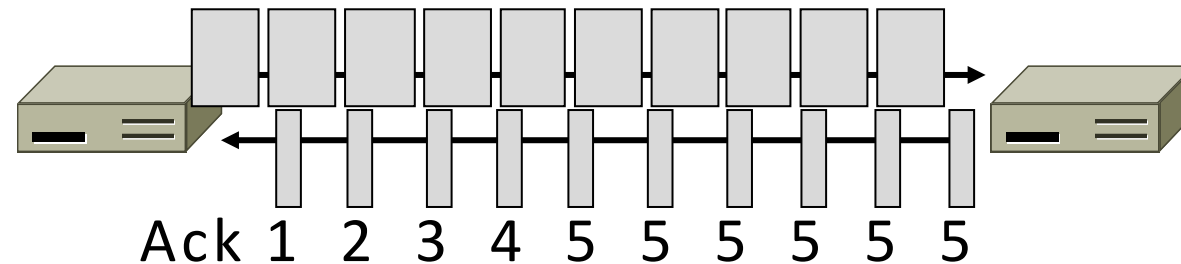- Set ssthresh = cwnd/2 after loss
- Begin with slow-start after timeout

Fast Recovery
(TCP Multiplicative Decrease)
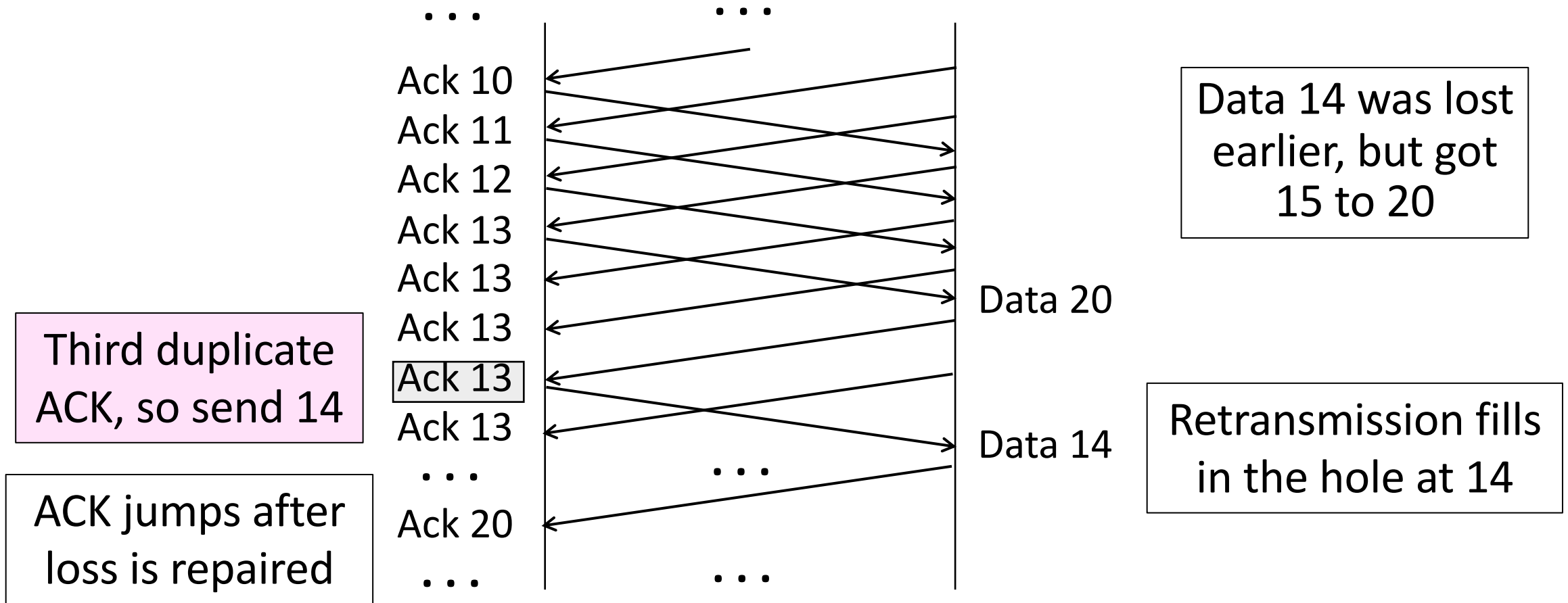
Inferring Loss from ACKs

- 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

Ack 1   2   3   4   5   5   5   5   5   5

Fast Retransmit (2)

. . .   . . .

Ack 10

Ack 11   Data 14 was lost
                earlier, but got
Ack 12   15 to 20

Ack 13

Ack 13

Ack 13   Data 20

Third duplicate
ACK, so send 14   Ack 13

Ack 13   Data 14   Retransmission fills
                                    in the hole at 14
. . .   . . .

ACK jumps after   Ack 20
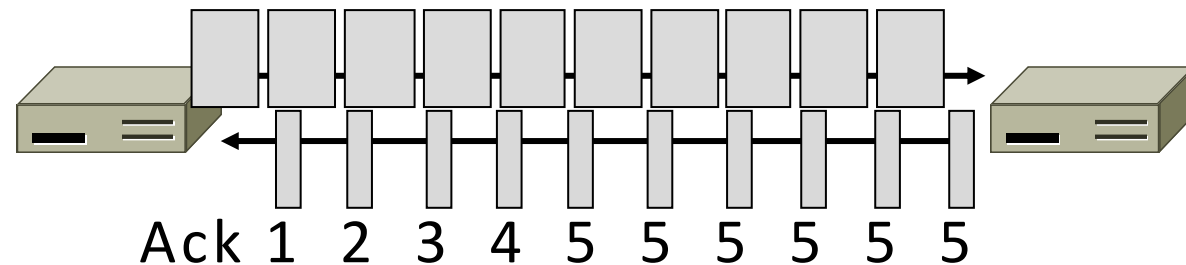loss is repaired

. . .   . . .

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 …
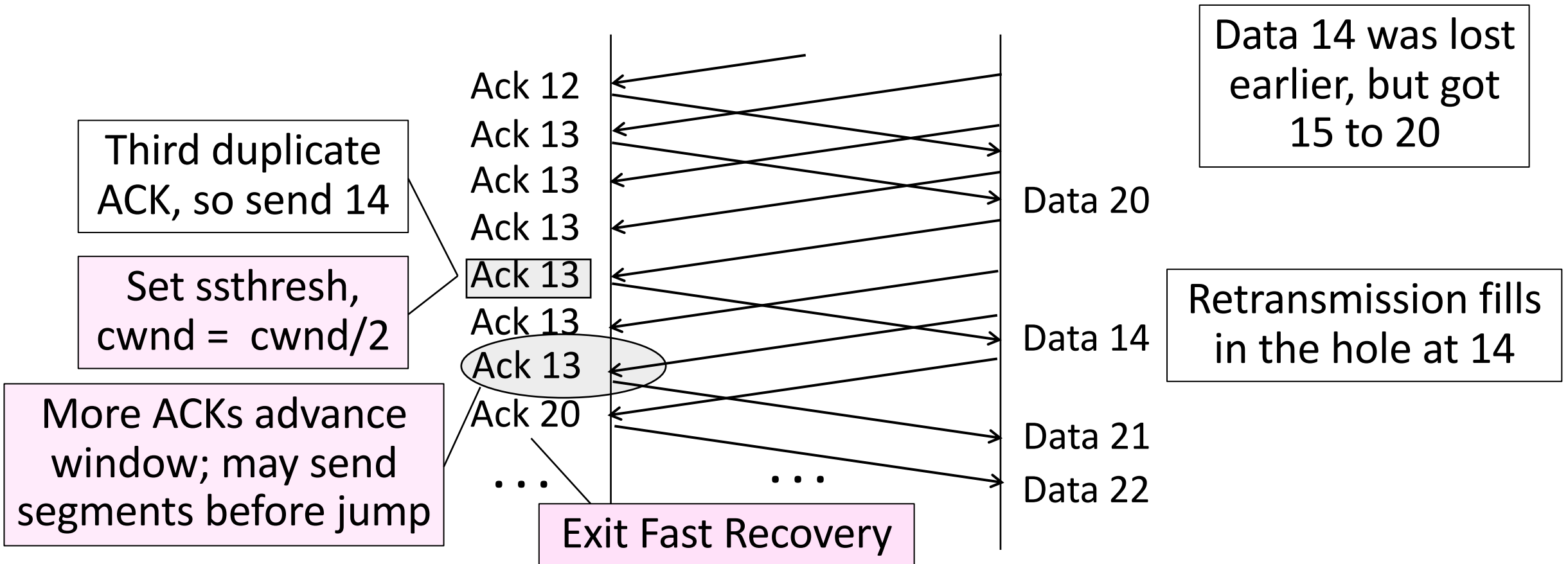
Inferring Non-Loss from ACKs

- 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

Ack 1 2 3 4 5 5 5 5 5 5

Fast Recovery (2)

Ack 12
Ack 13
Ack 13
Ack 13
Ack 13
Ack 13
Ack 13
Ack 20

Data 20
Data 14
Data 21
Data 22

Data 14 was lost earlier, but got 15 to 20

Third duplicate ACK, so send 14

Set ssthresh, cwnd = cwnd/2

More ACKs advance window; may send segments before jump

Exit Fast Recovery

Retransmission fills in the hole at 14

. . .

. . .

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
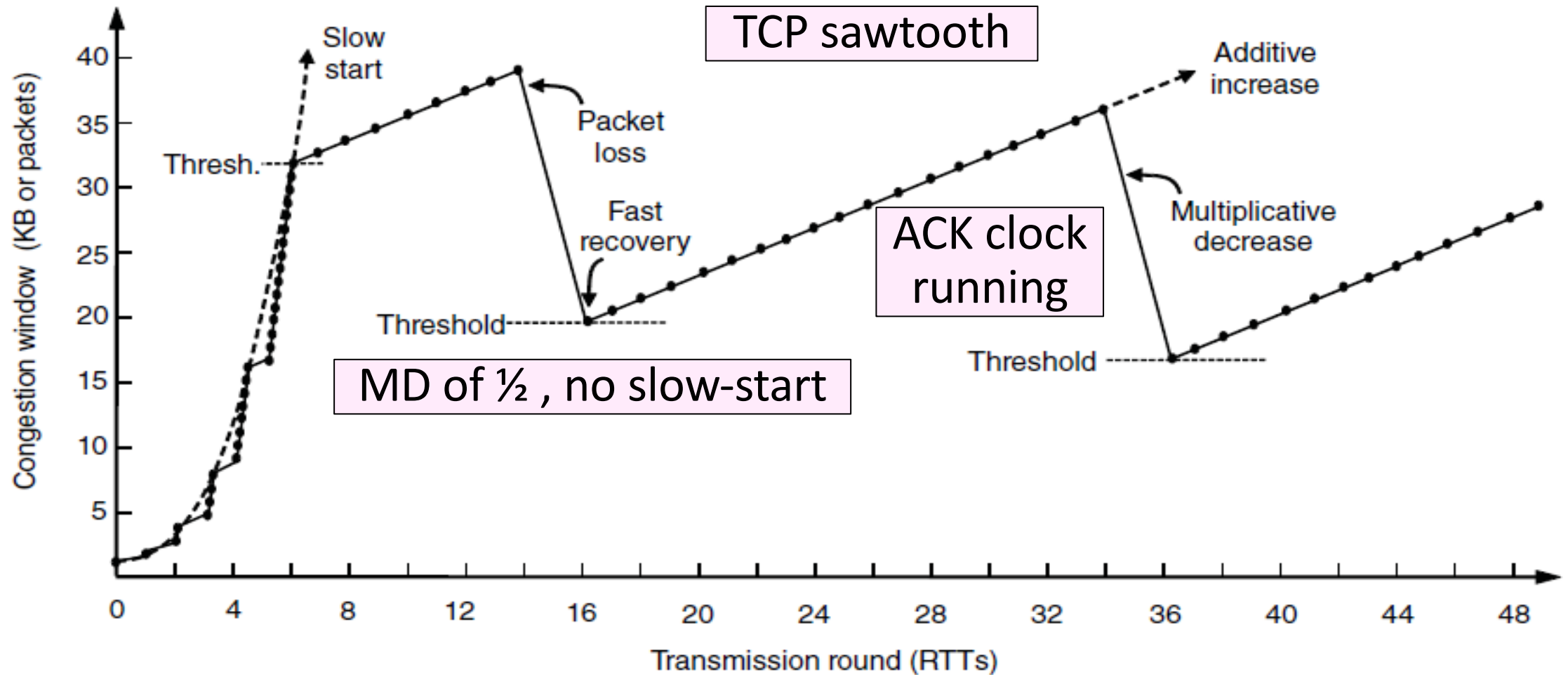
TCP Reno

TCP 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-Assisted Congestion Control

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 Other events can cause loss
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 <u>marks</u> affected packets (IP header)
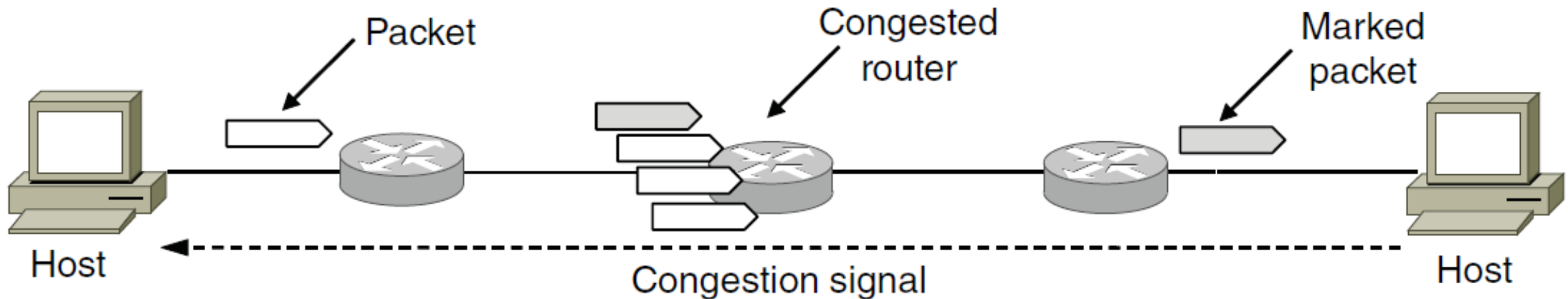
ECN (2)

- Marked packets arrive at receiver
  - TCP receiver informs TCP sender of the congestion
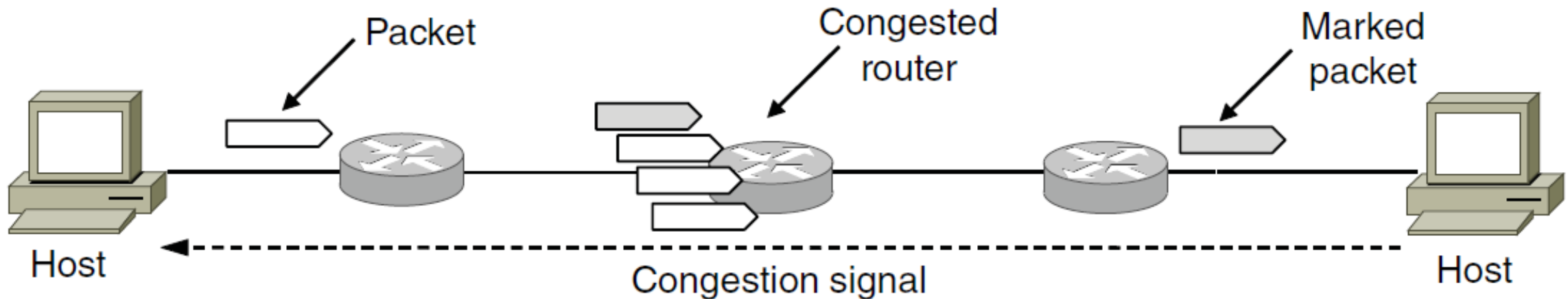
ECN (3)

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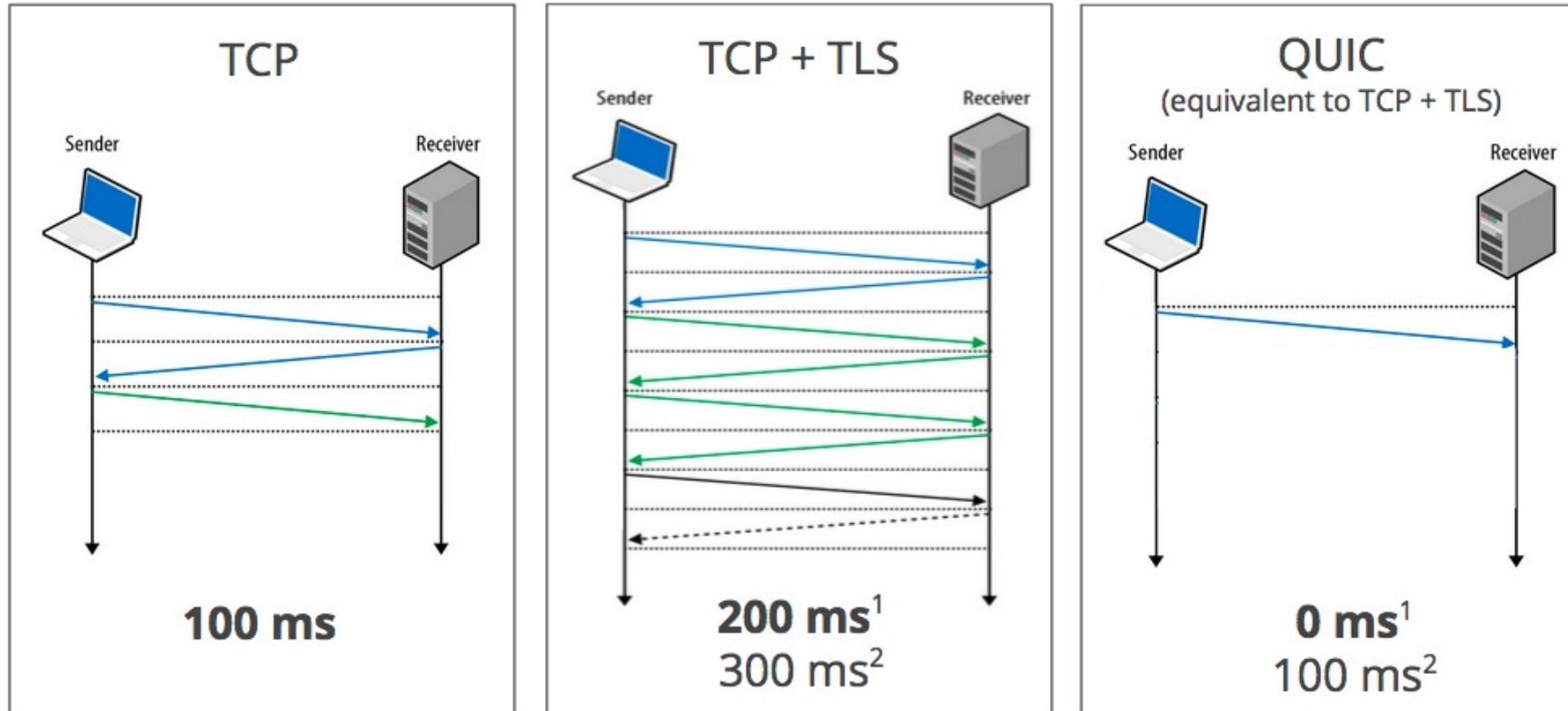 - Routers and hosts must be upgraded

What's new in transport protocols?
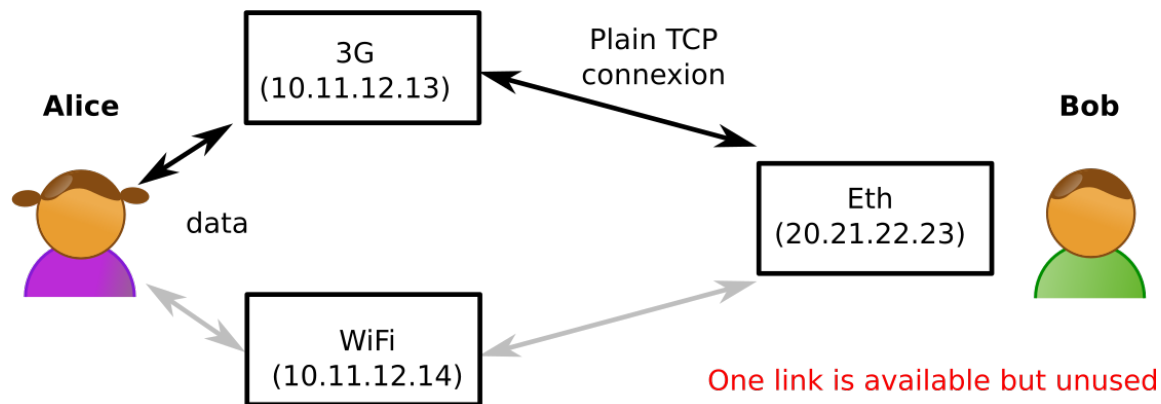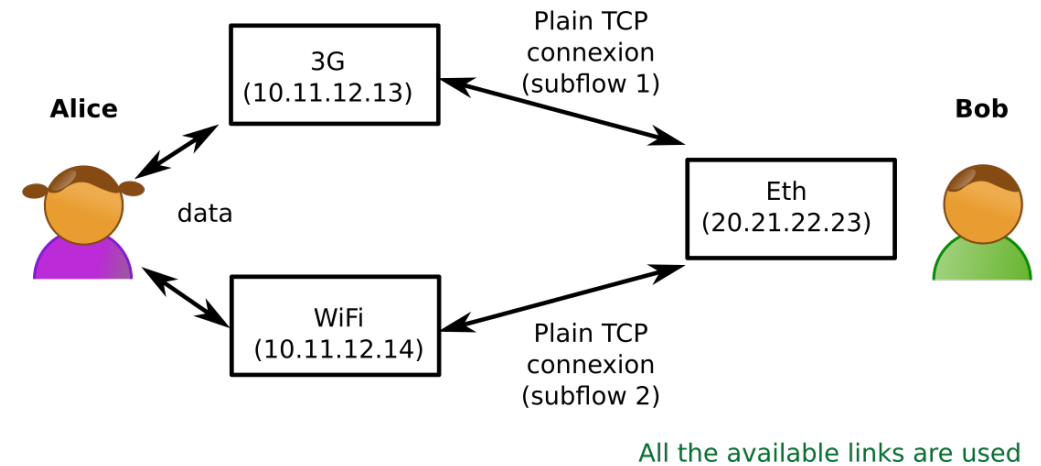
QUIC

MPTCP

BBR

DCTCP

QUIC

https://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html

MPTCP: Multipath TCP

By Aclarembeau - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=49727919
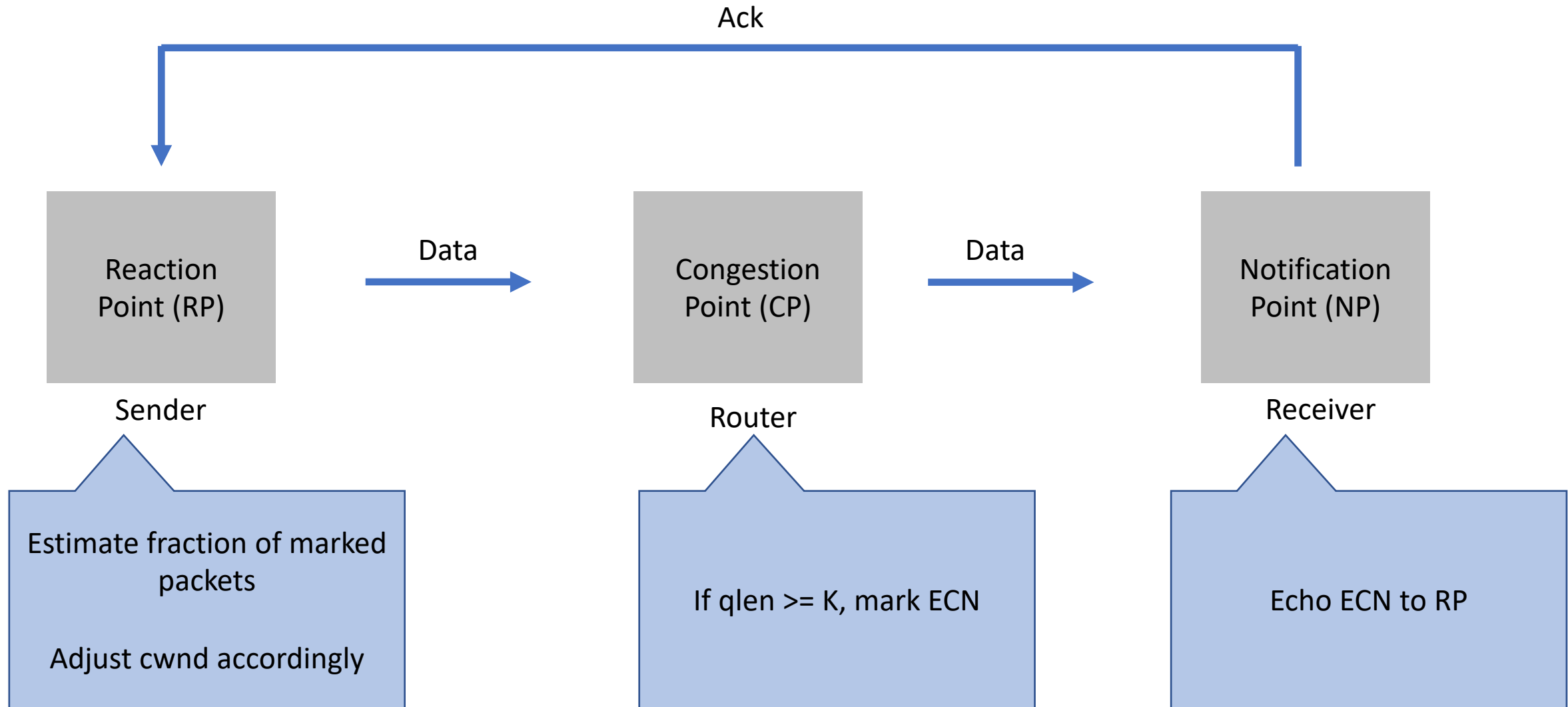
BBR:
Bottleneck Bandwidth and Round trip propagation
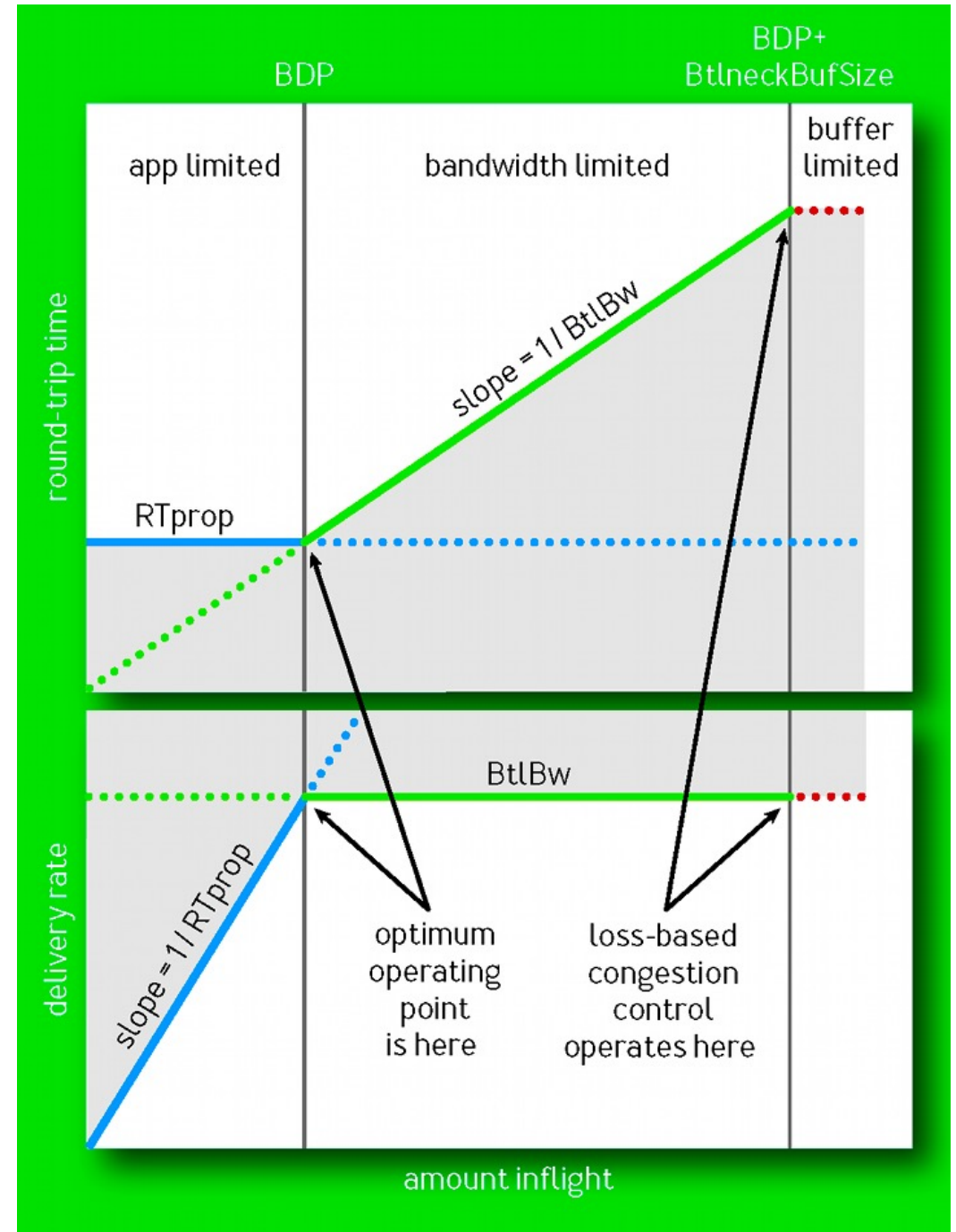
DCTCP at a glance

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, ….