UNIVERSITY OF WATERLOO

Congestion control - Part II

CS 456/656, Fall 2026

Mina Tahmasbi Arashloo

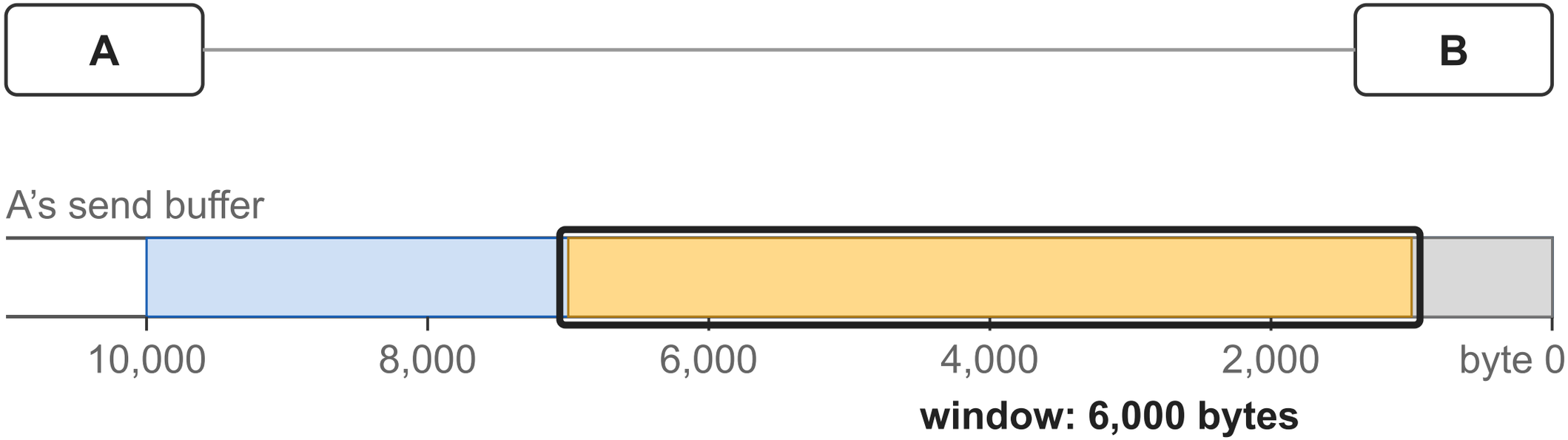
Recap: congestion control

- Congestion control algorithms use control packets and measurements to decide when to send more data into the network:
  - Typically part of the transport layer (Why?)
- This is a hard problem:
  - Keep the bottleneck busy
  - Keep the queues short
  - Share fairly with other flows
  - React quickly when things change
  - Without much direct input from the network
  - Without coordination with other flows
  - …
- Four decades of research

Recap: congestion control

- Congestion control algorithms use control packets and measurements to decide when to send more data into the network:
  - Typically part of the transport layer (Why?)
- This is a hard problem:
  - Keep the bottleneck busy
  - Keep the queues short
  - Share fairly with other flows
  - React quickly when things change
  - Without much direct input from the network
  - Without coordination with other flows
  - …
- Four decades of research
- → Let’s see how TCP classically does congestion control

Recap: the sliding window

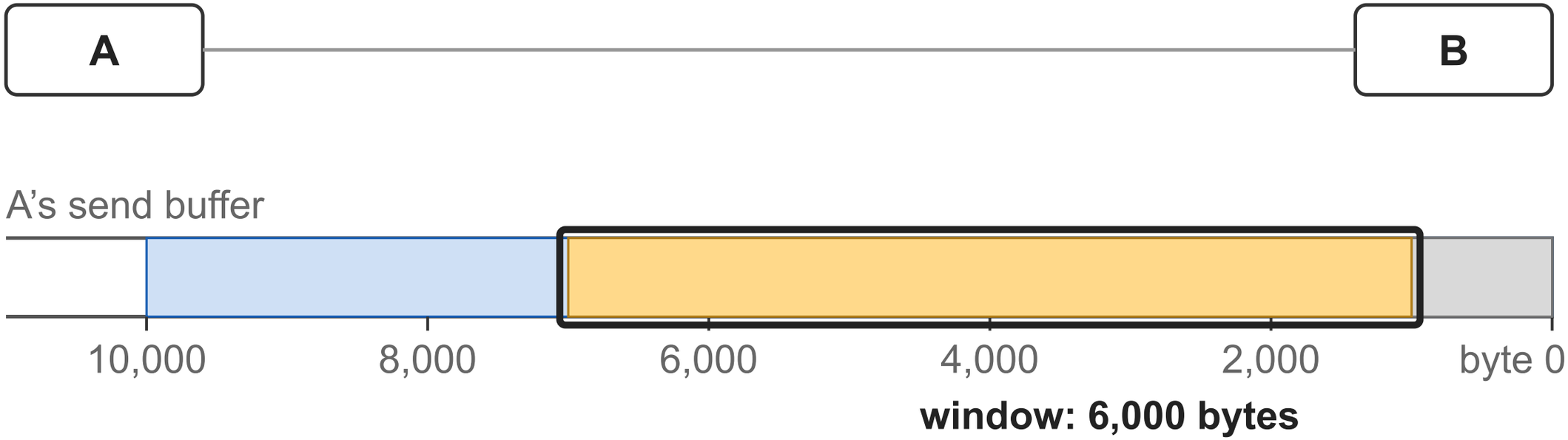


A only sends bytes inside the window

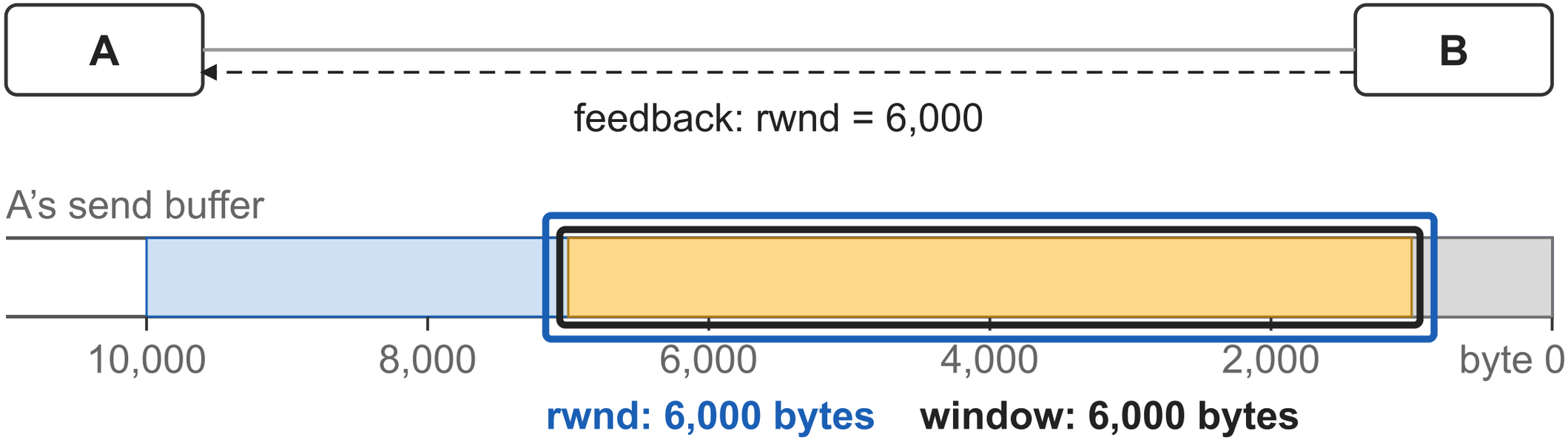
The window starts at the oldest byte without feedback

feedback received sent, waiting for feedback not sent yet

So far, sliding window = receive window



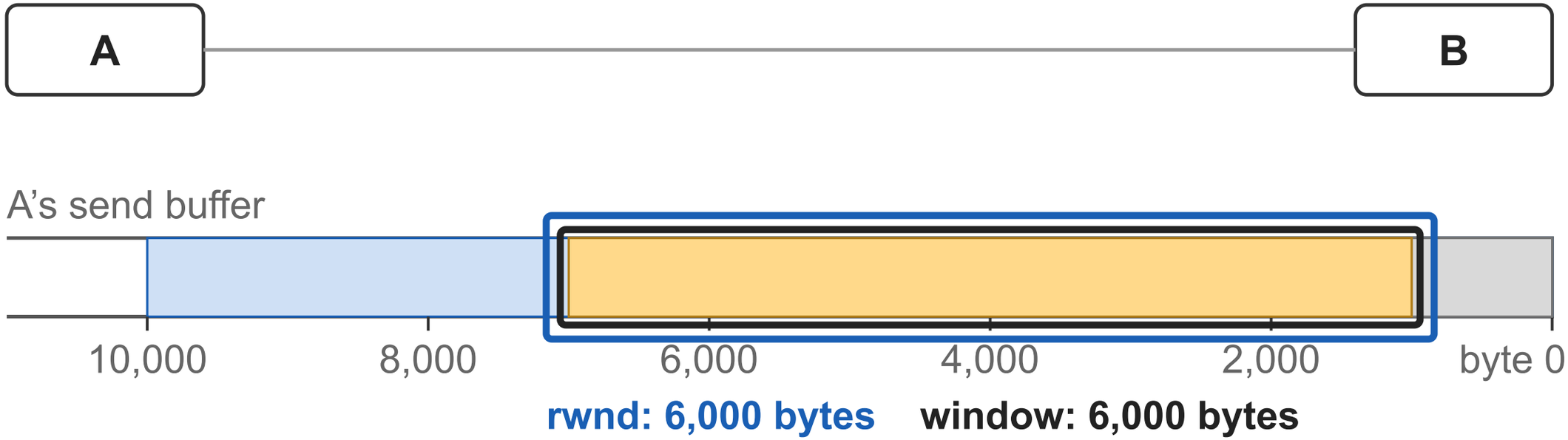
So far, sliding window = receive window



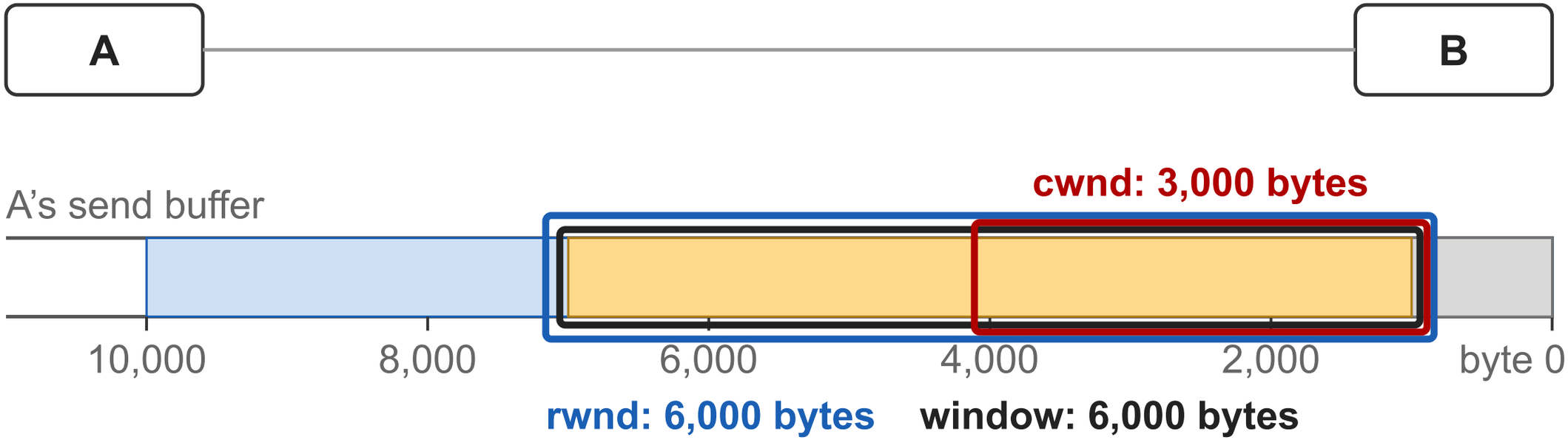
Receive window (rwnd): how far A can go beyond the oldest unacked byte, so B's receive buffer doesn't overflow

Sliding window size = rwnd

Congestion window

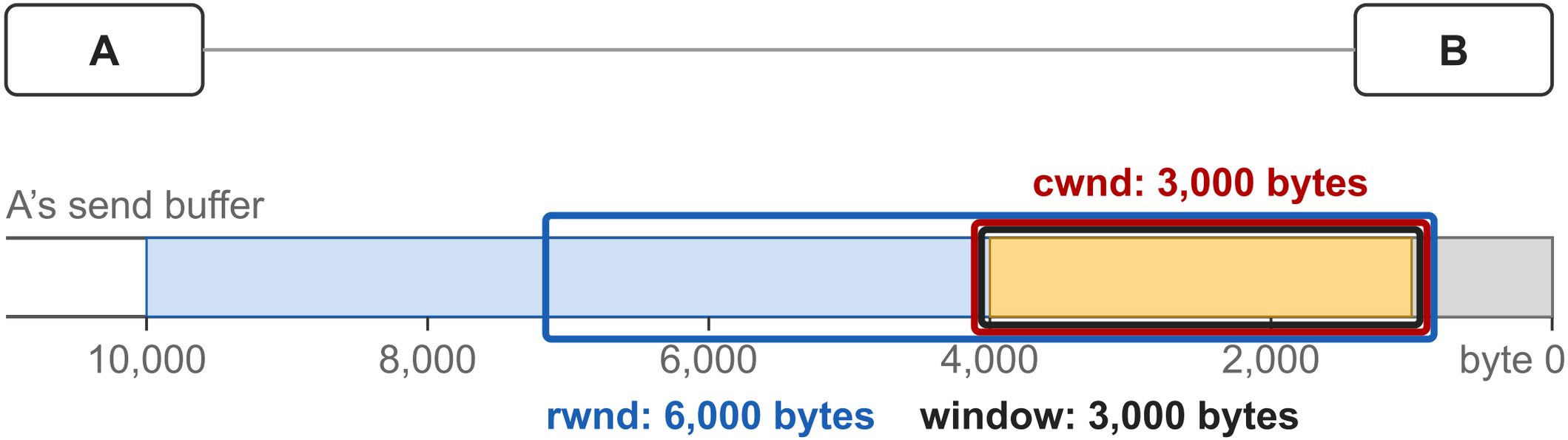


Congestion window



Congestion window (cwnd): how much capacity the network has for this flow at the moment

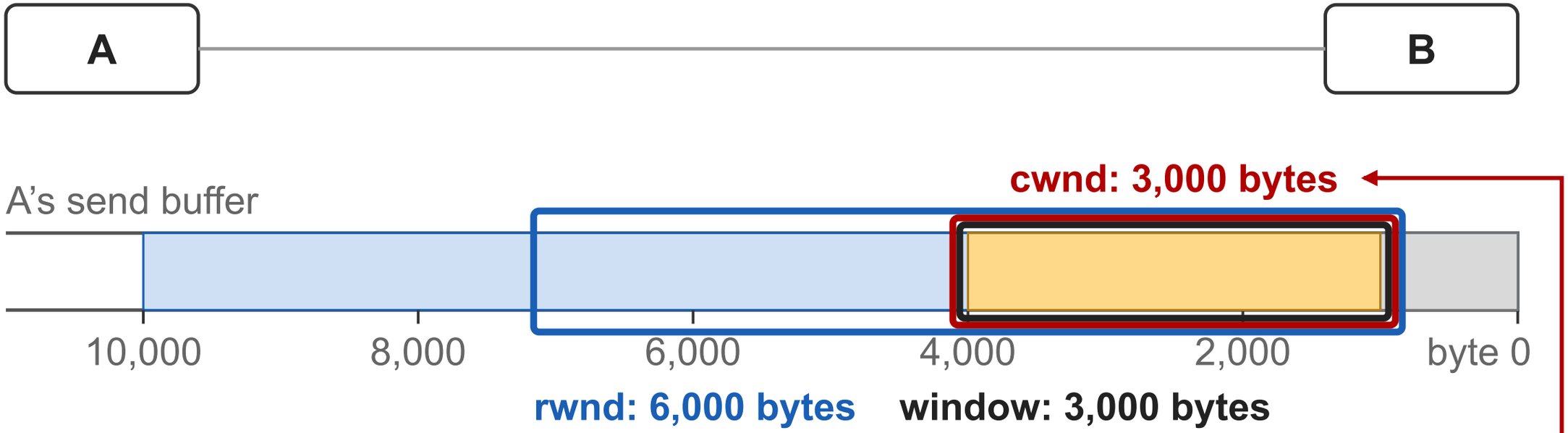
Congestion window



Congestion window (cwnd): how much capacity the network has for this flow at the moment

Sliding window size = the smaller of cwnd and rwnd

Congestion window



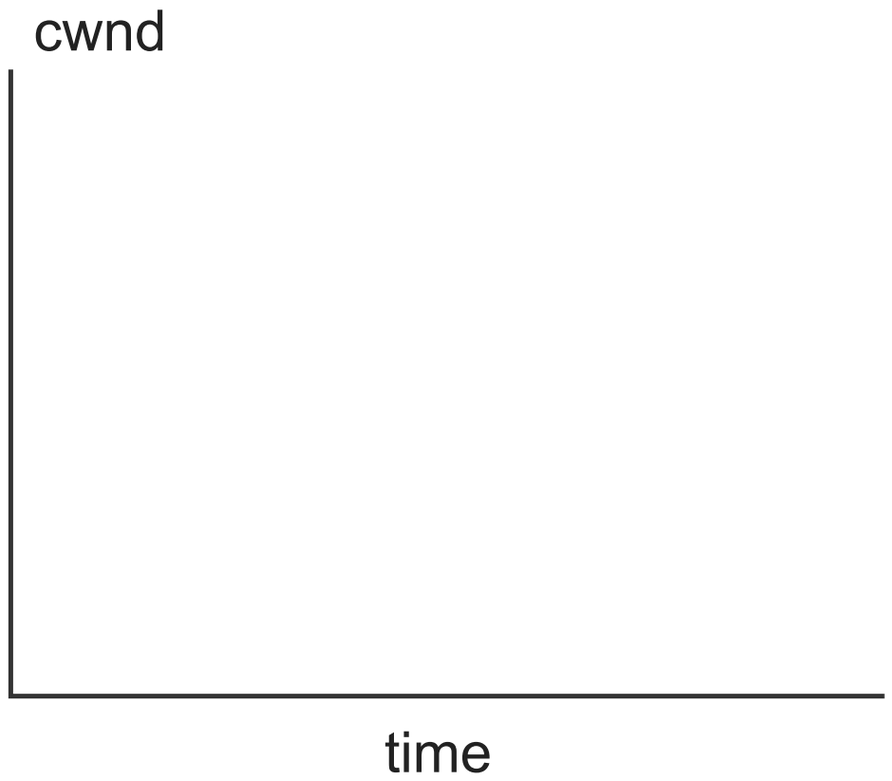
Congestion window (cwnd): how much capacity the network has for this flow at the moment

Sliding window size = the smaller of cwnd and rwnd

A needs to adjust cwnd based on signals from the network

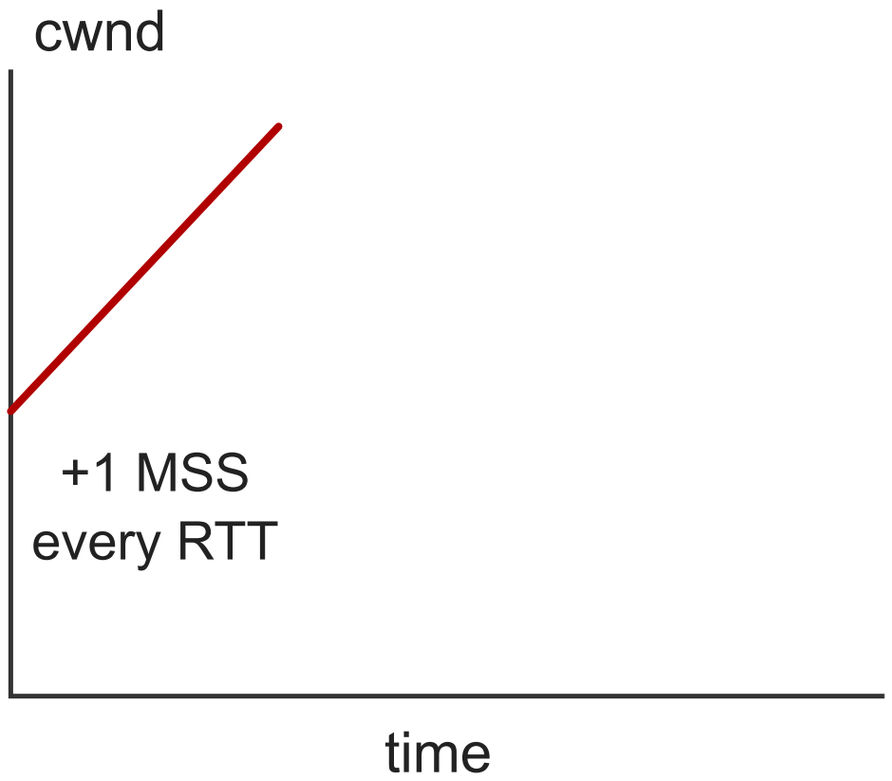
TCP congestion control - Basic AIMD

- Increase cwnd until a loss, then decrease it



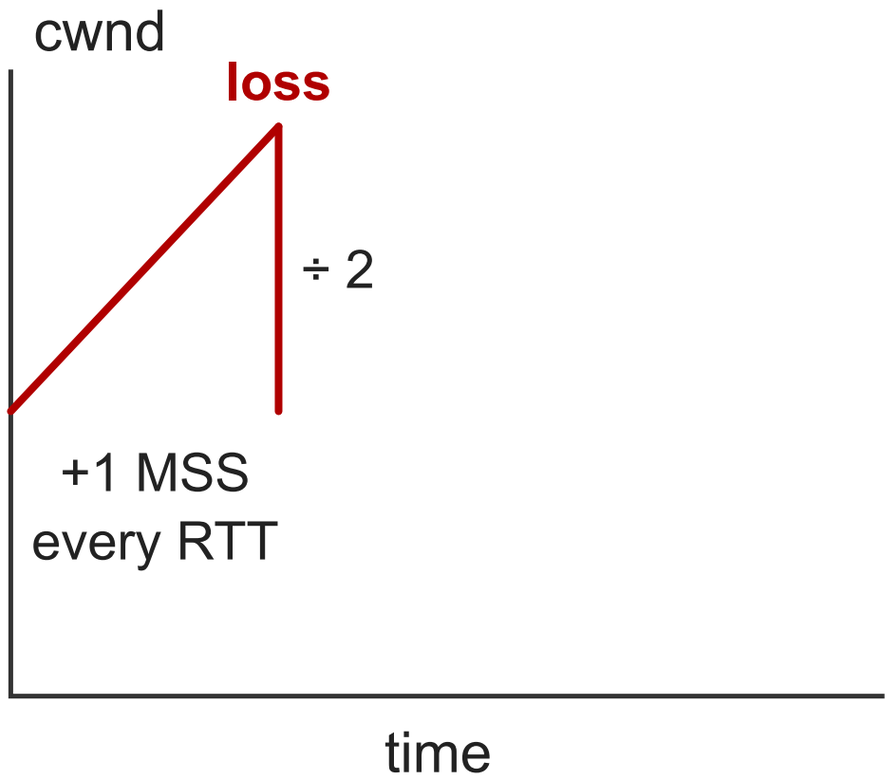
TCP congestion control - Basic AIMD

- Increase cwnd until a loss, then decrease it
- **Additive increase (AI):**
  - Increase cwnd by one MSS every RTT



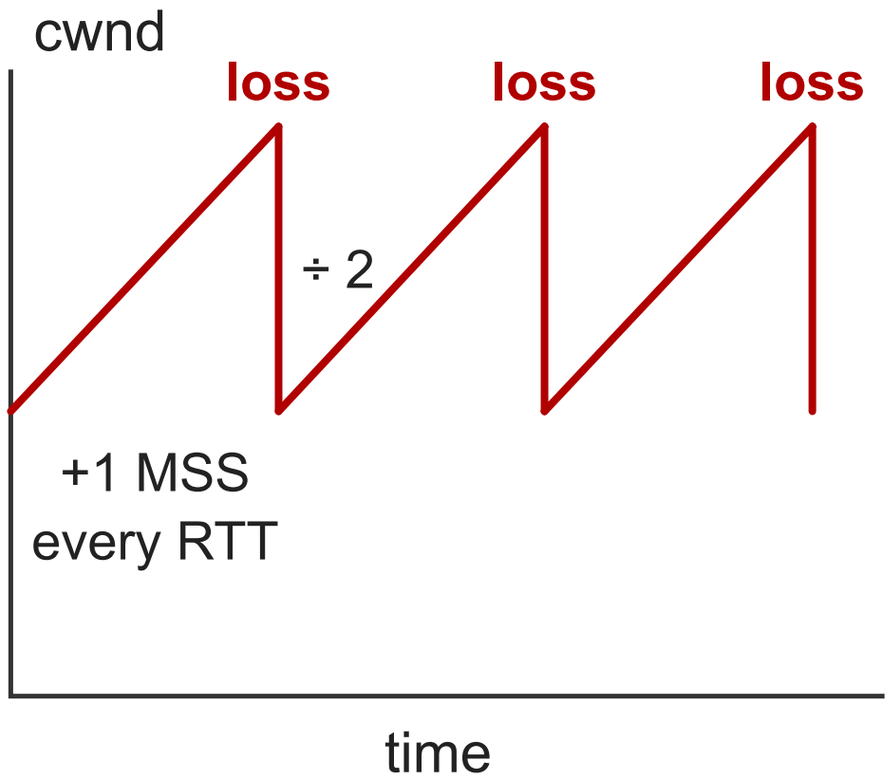
TCP congestion control - Basic AIMD

- Increase cwnd until a loss, then decrease it
- **Additive increase (AI):**
  - Increase cwnd by one MSS every RTT
- **Multiplicative decrease (MD):**
  - On a loss: cut cwnd in half

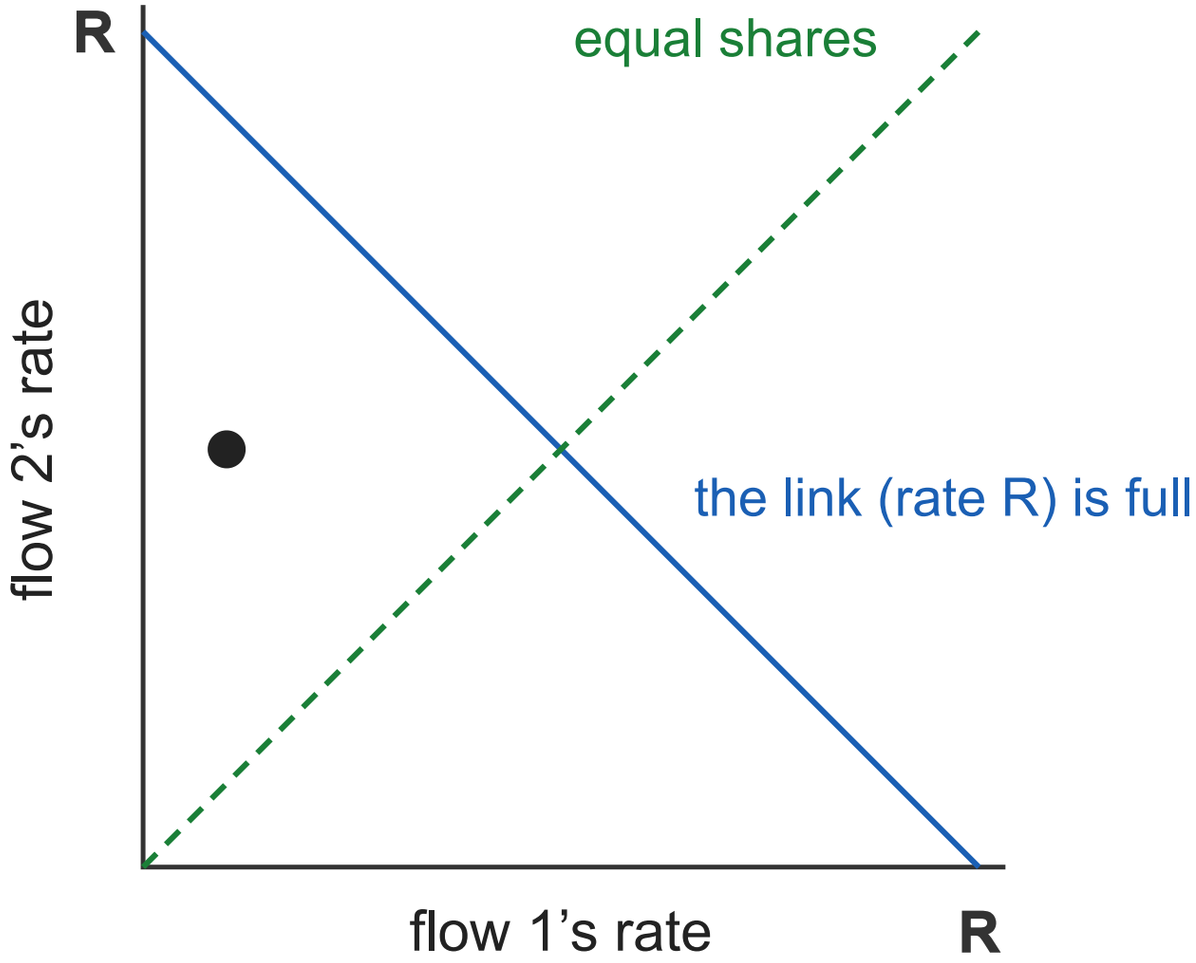


TCP congestion control - Basic AIMD

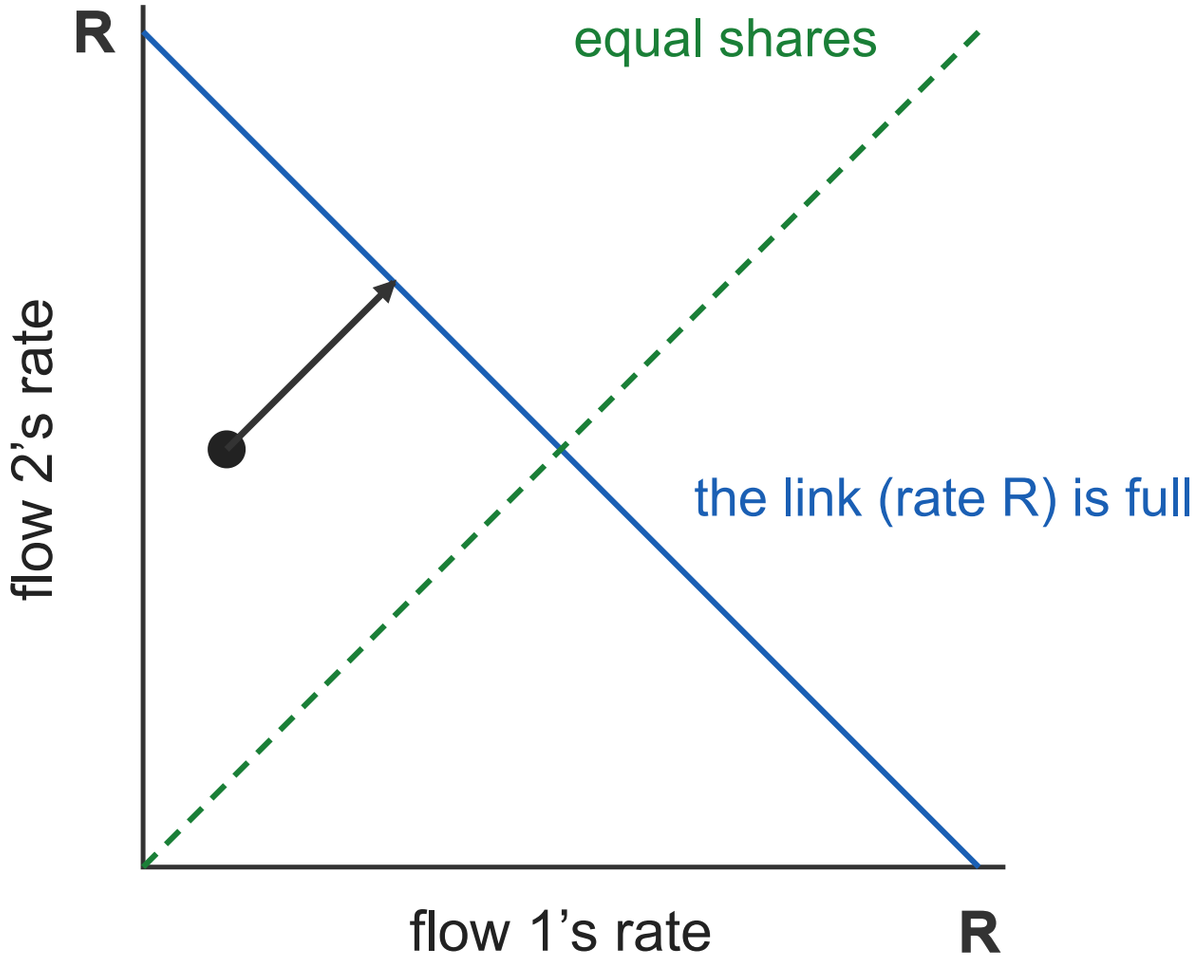
- Increase cwnd until a loss, then decrease it
- **Additive increase (AI):**
  - Increase cwnd by one MSS every RTT
- **Multiplicative decrease (MD):**
  - On a loss: cut cwnd in half
- Cautious probing, aggressive backing off on loss:
  - “the TCP sawtooth”



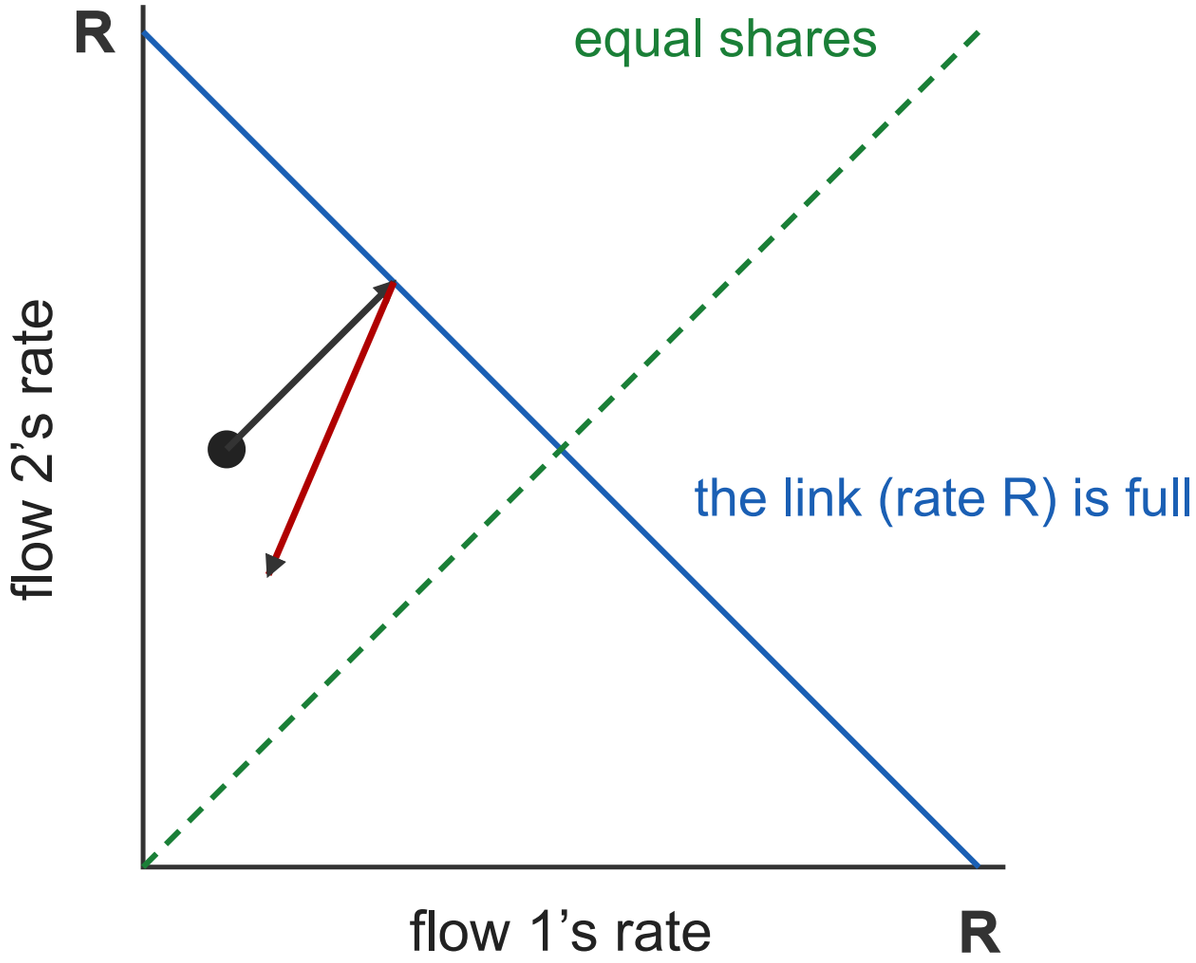
Two flows sharing a link under Basic AIMD



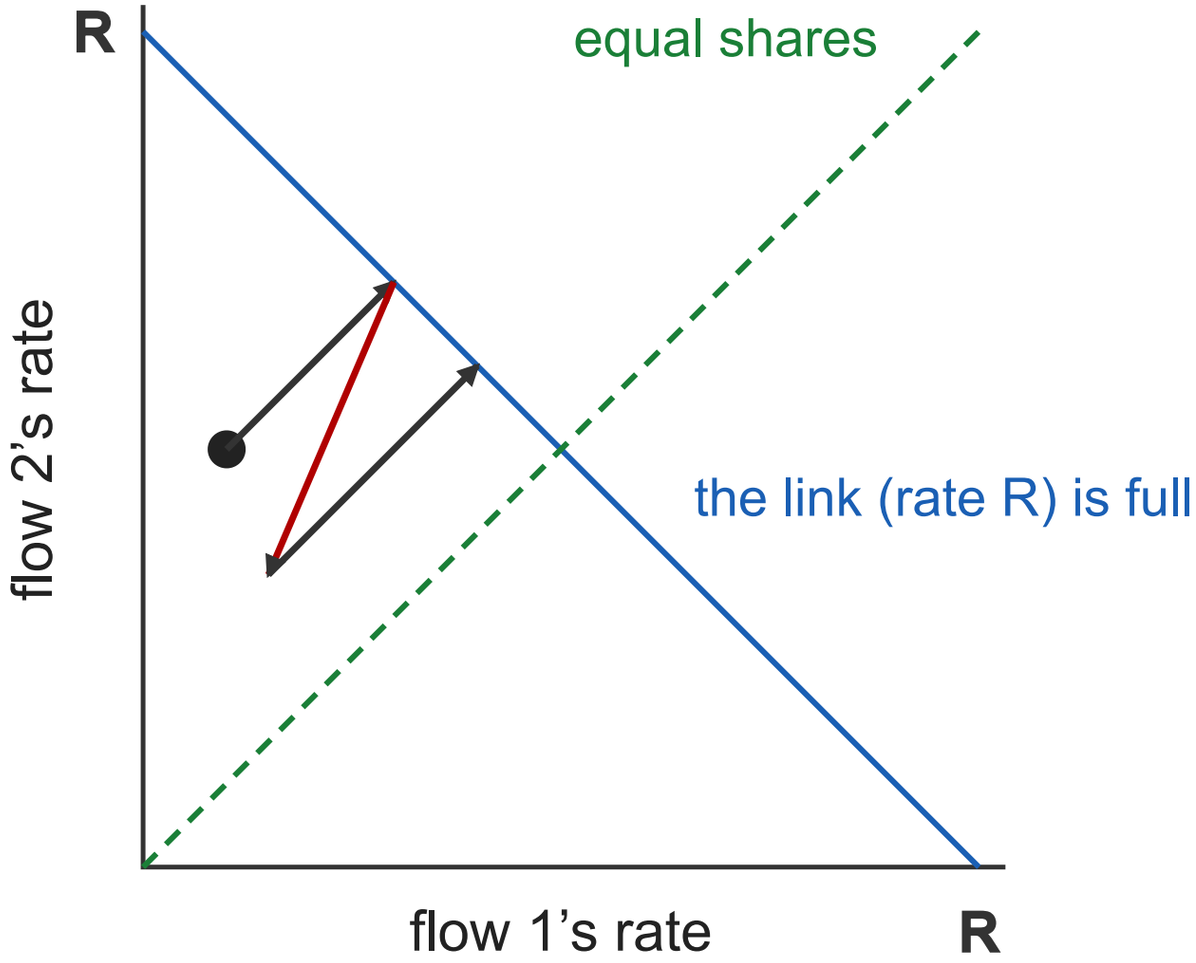
Two flows sharing a link under Basic AIMD



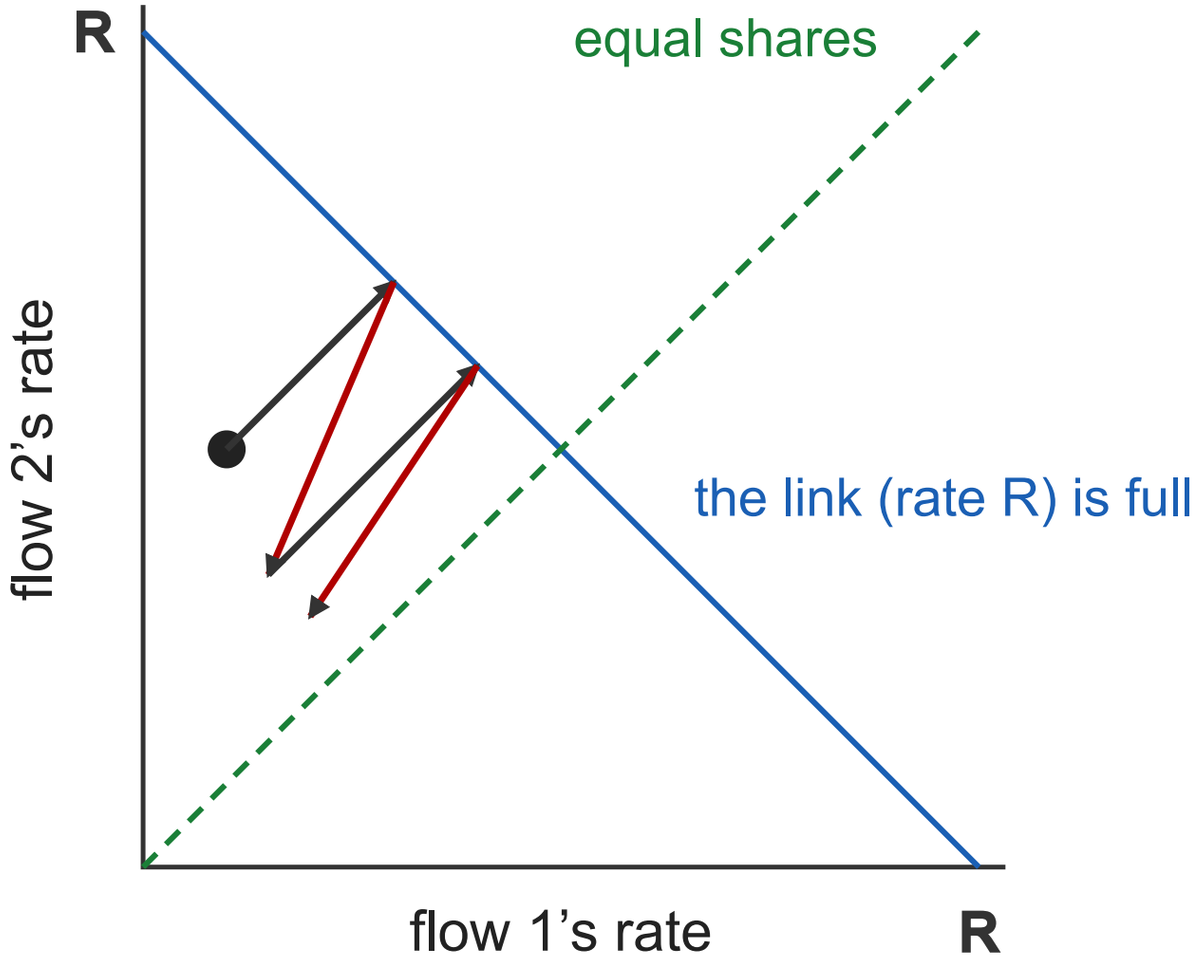
Two flows sharing a link under Basic AIMD



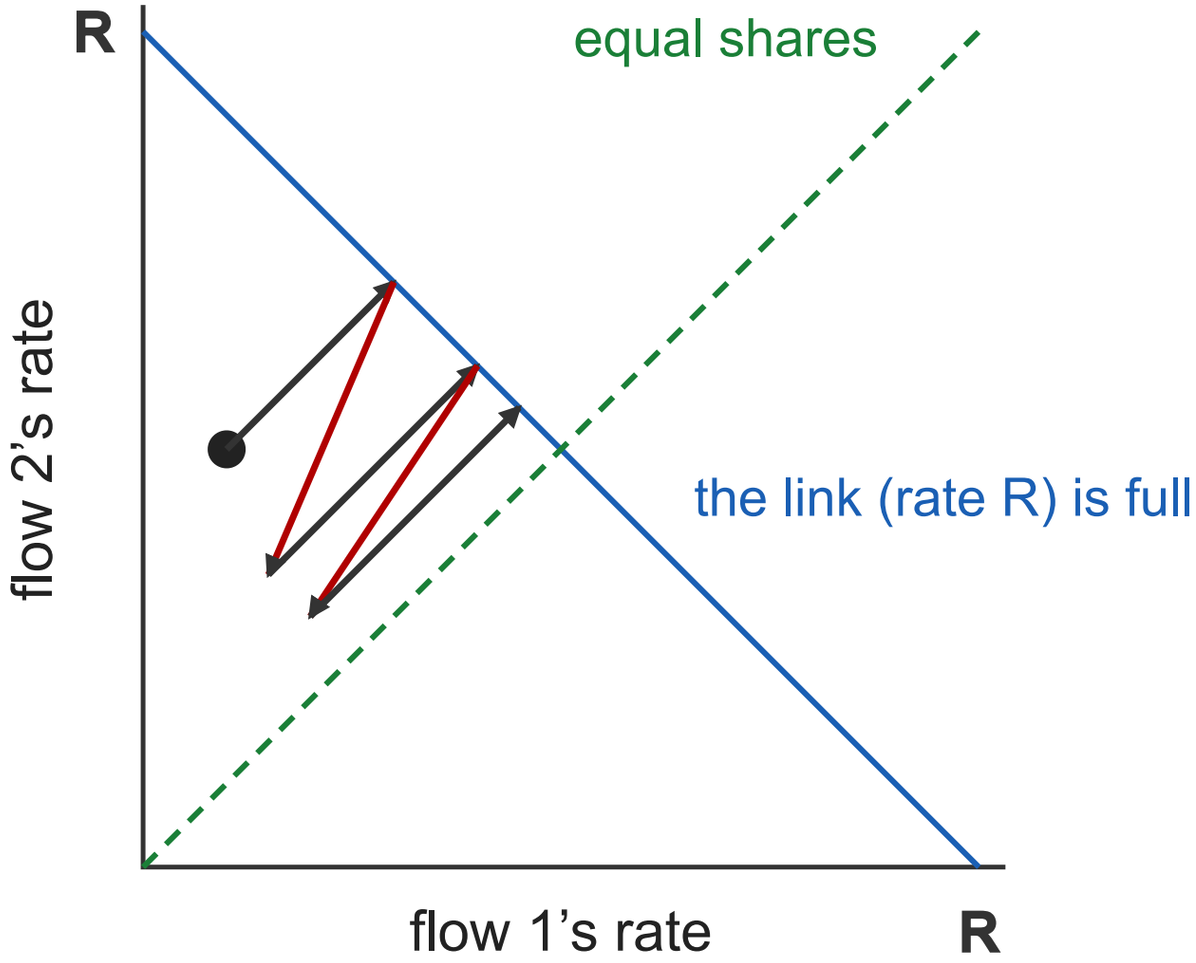
Two flows sharing a link under Basic AIMD



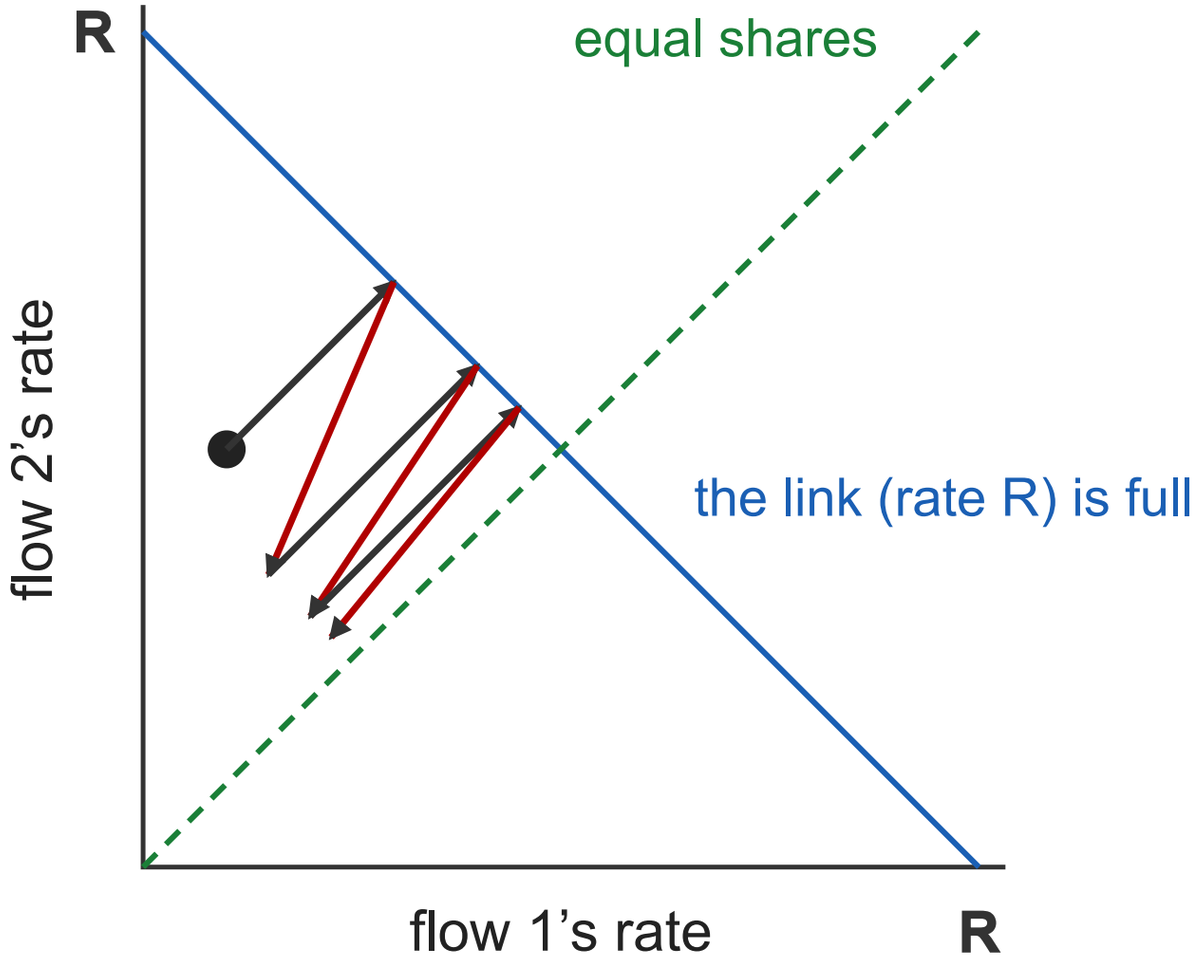
Two flows sharing a link under Basic AIMD



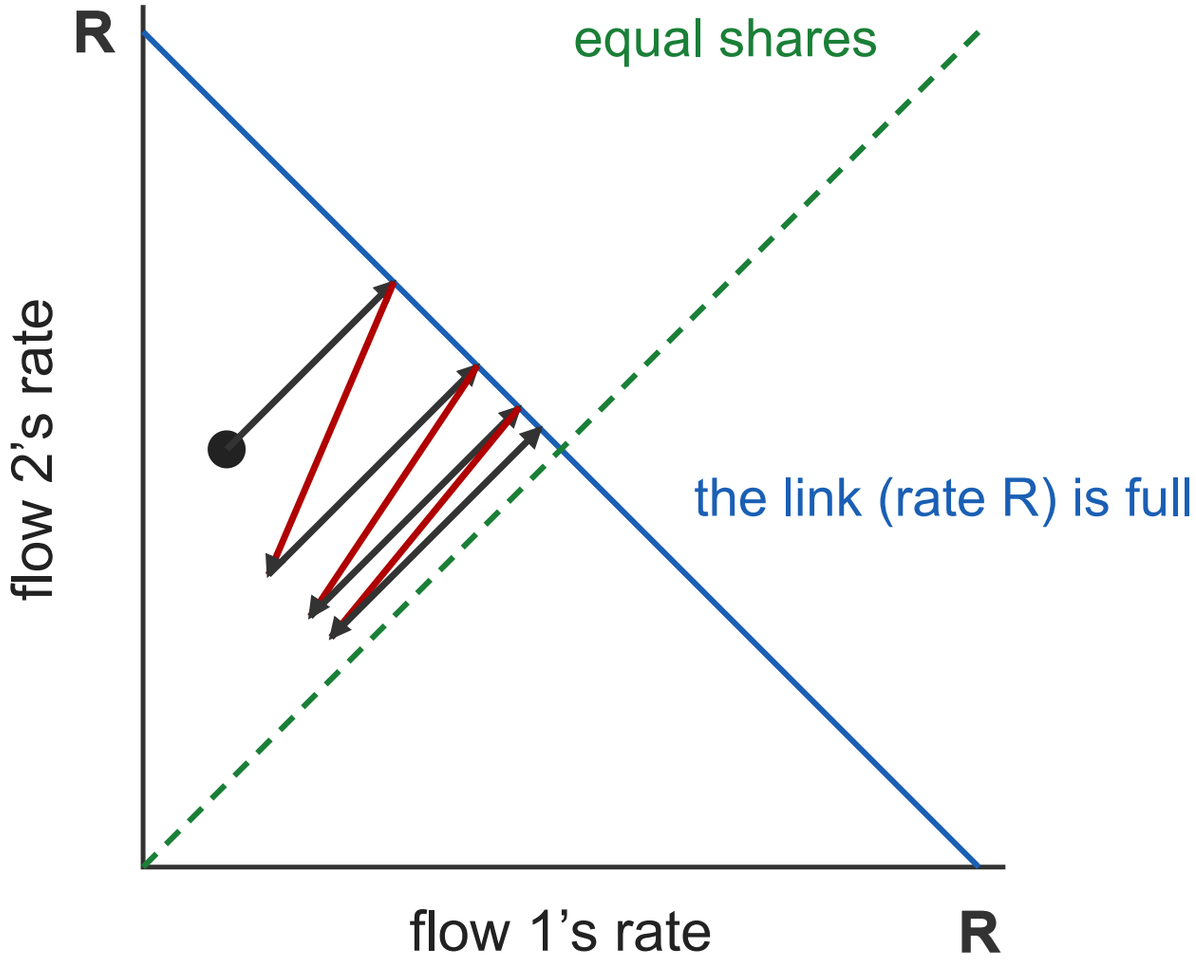
Two flows sharing a link under Basic AIMD



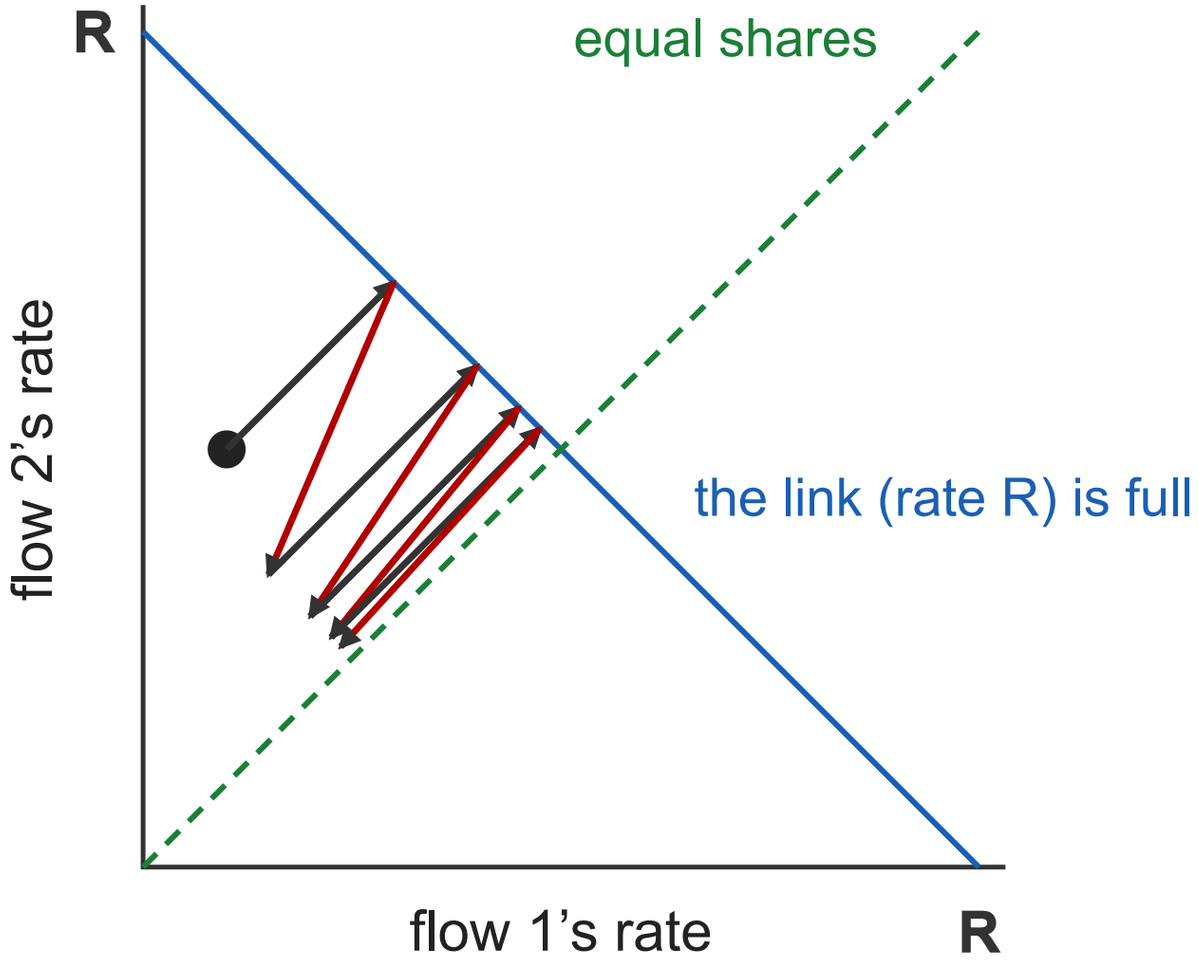
Two flows sharing a link under Basic AIMD



Two flows sharing a link under Basic AIMD



Two flows sharing a link under Basic AIMD



Under idealized assumptions, they share the link fairly: each gets R/2

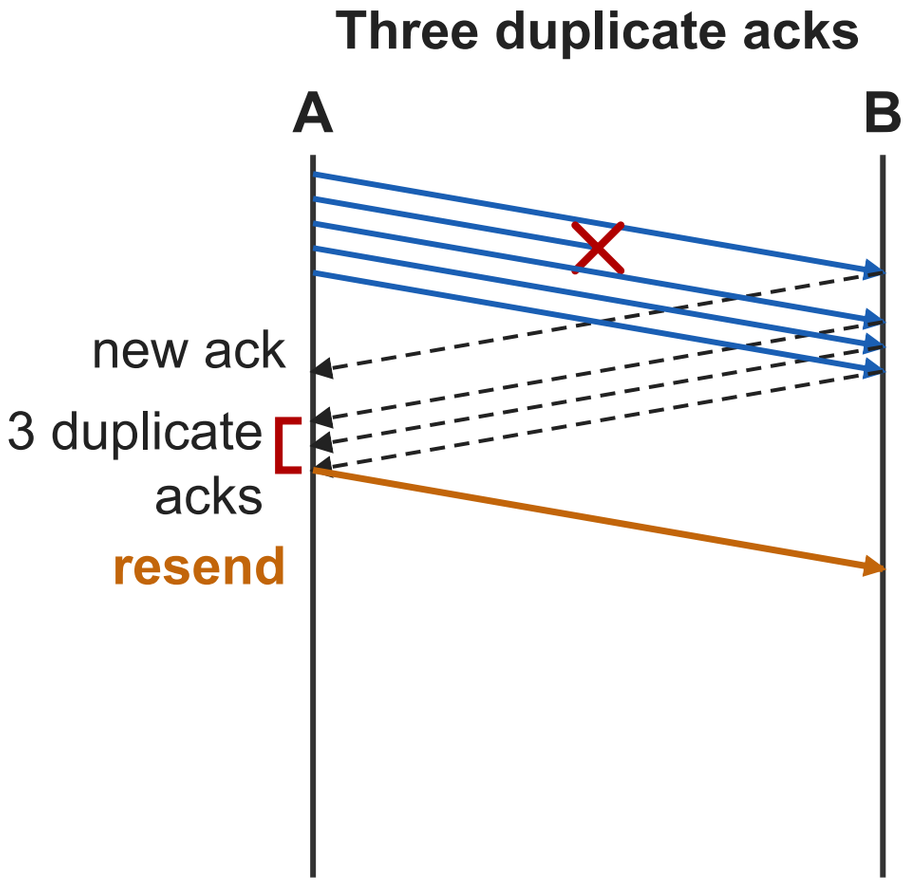
– Same RTT

– Fixed number of flows

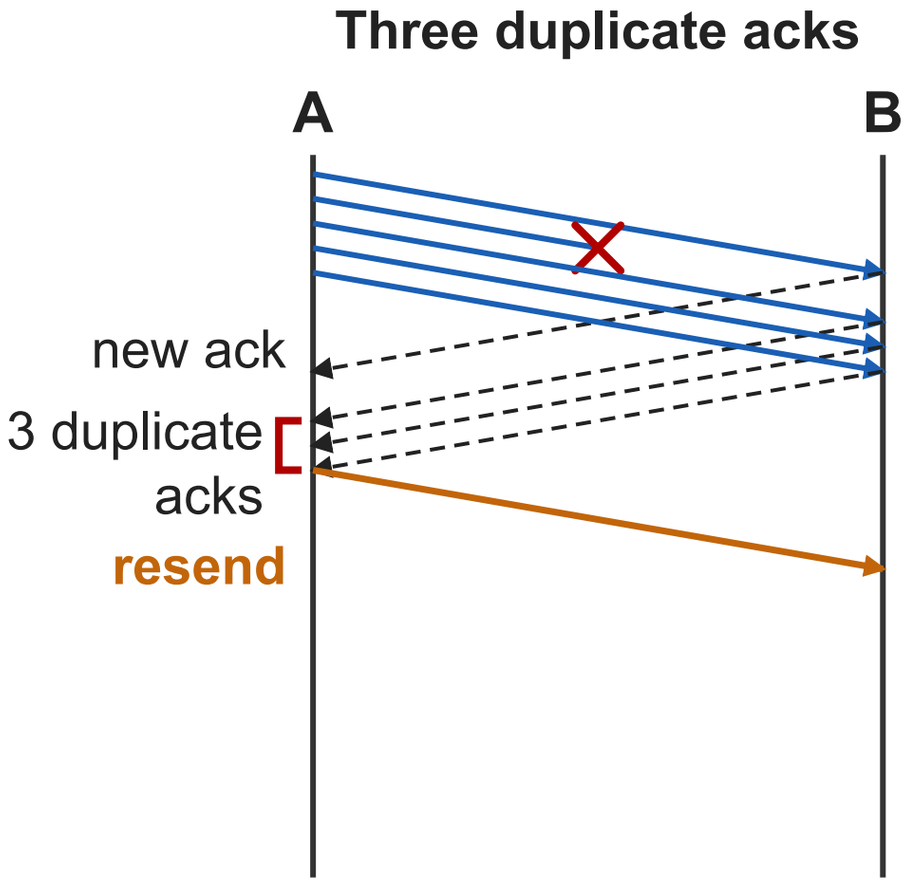
– Both only doing AIMD

– …

Classic TCP: two kinds of loss signal

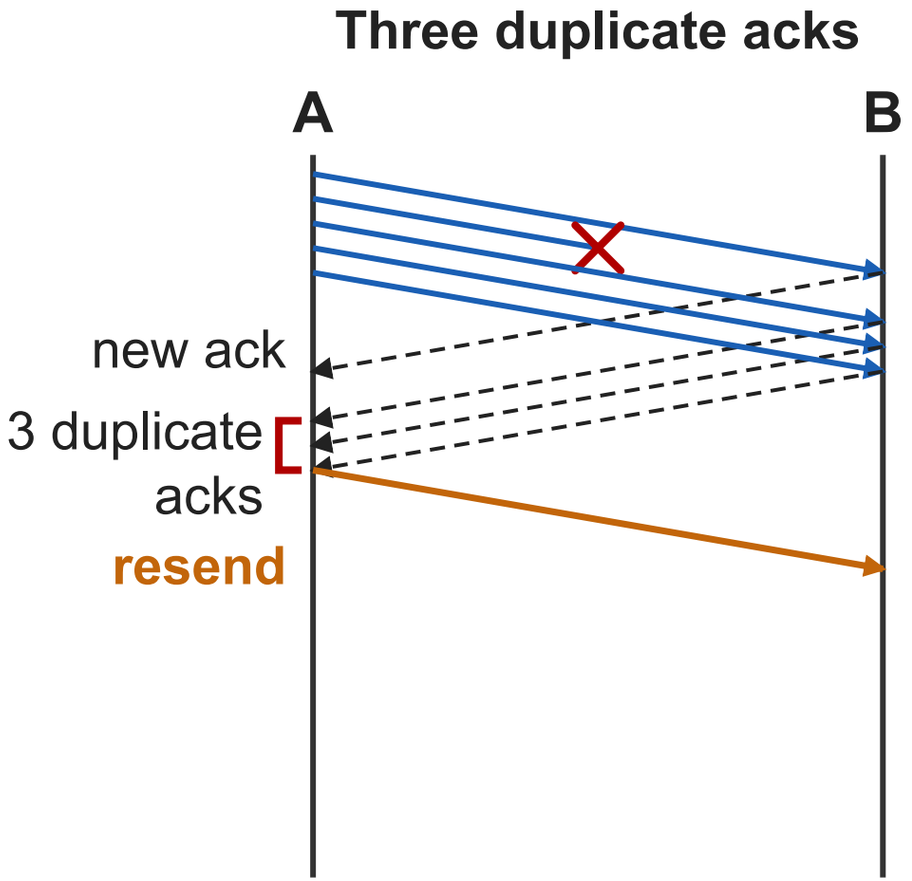


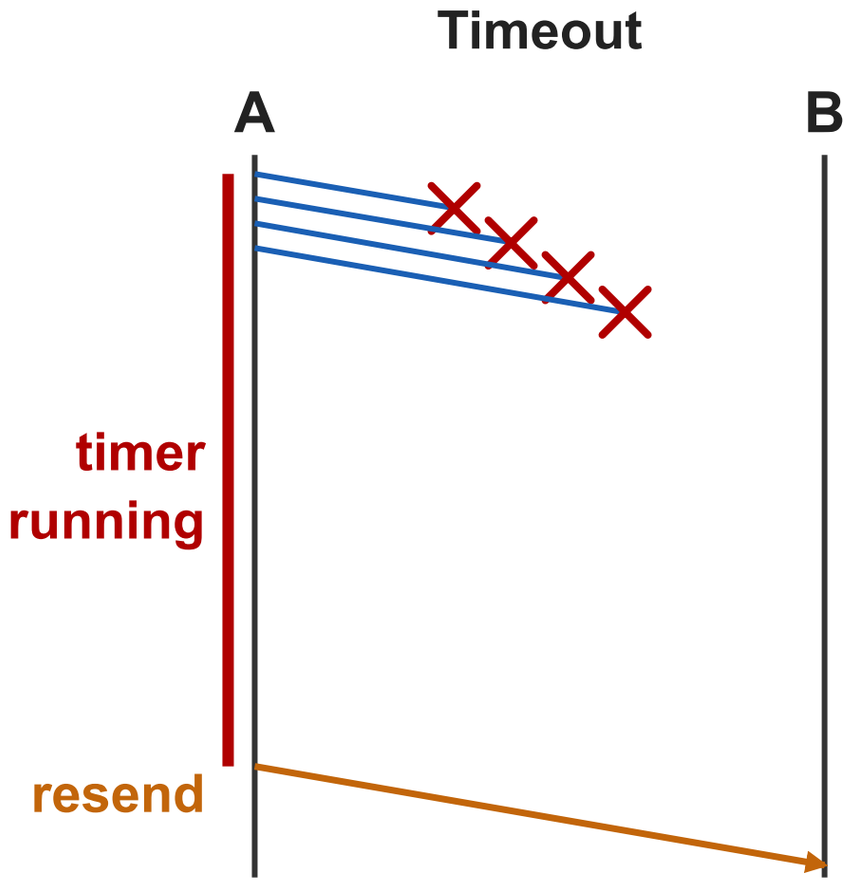
Classic TCP: two kinds of loss signal



Later packets still get through: **mild congestion**. Resend at once (fast retransmit) and cut cwnd in half

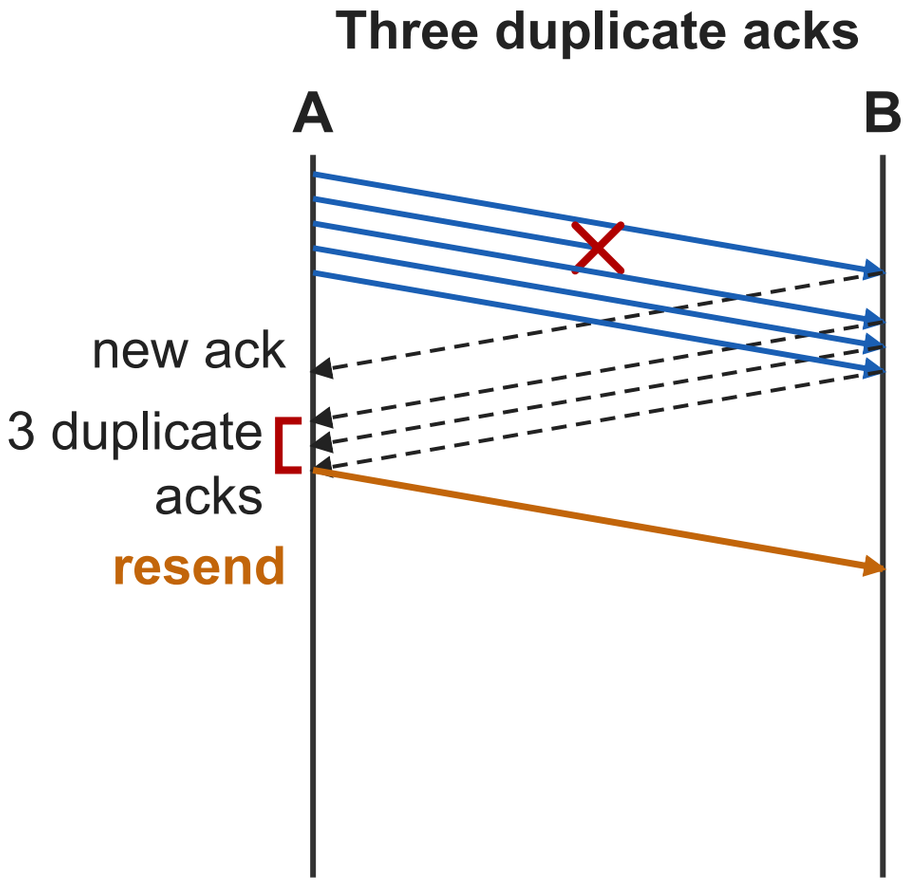
Classic TCP: two kinds of loss signal



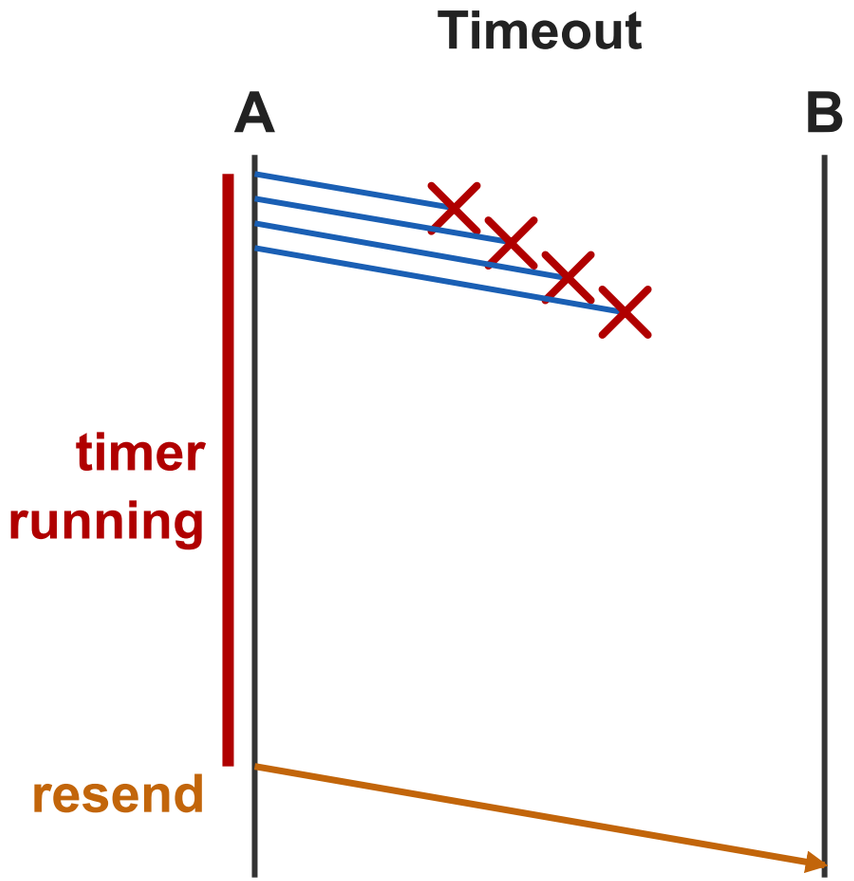


Later packets still get through: **mild congestion**. Resend at once (fast retransmit) and cut cwnd in half

Classic TCP: two kinds of loss signal



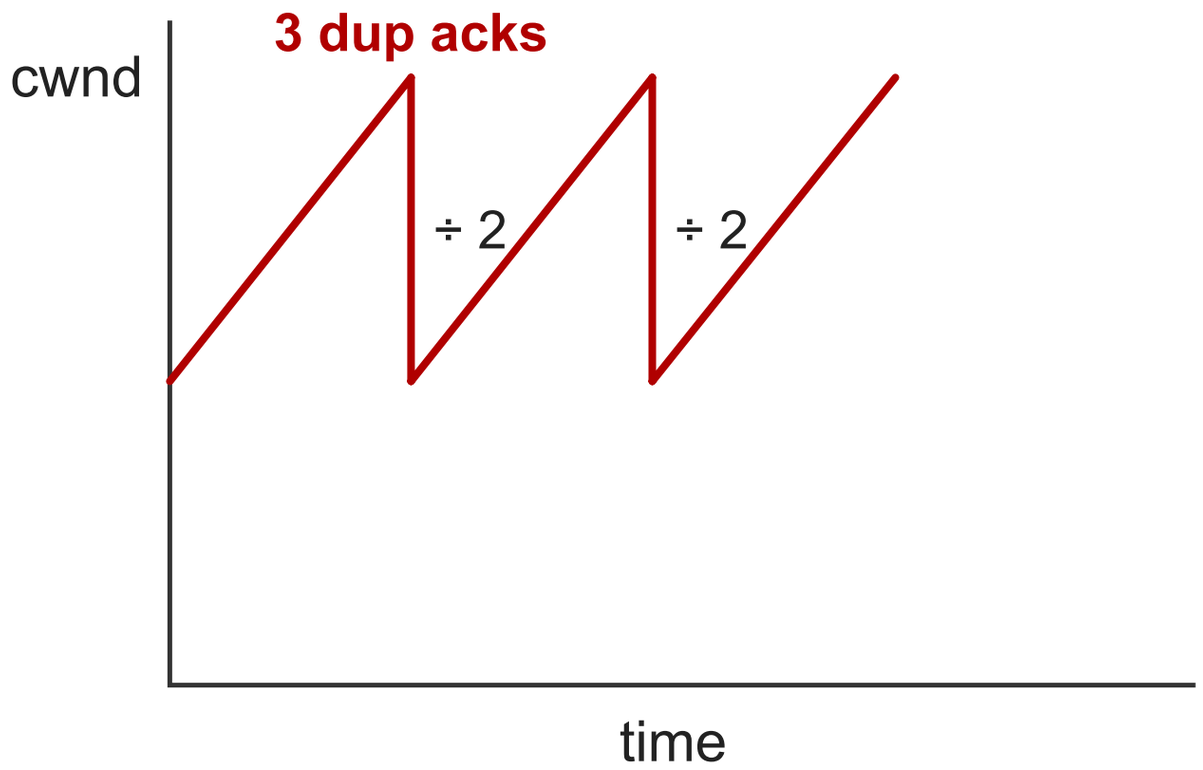
Later packets still get through: **mild congestion**. Resend at once (fast retransmit) and cut cwnd in half



Nothing is getting through: **severe congestion**. cwnd goes back to 1 MSS

Classic TCP: AIMD, with both loss signals

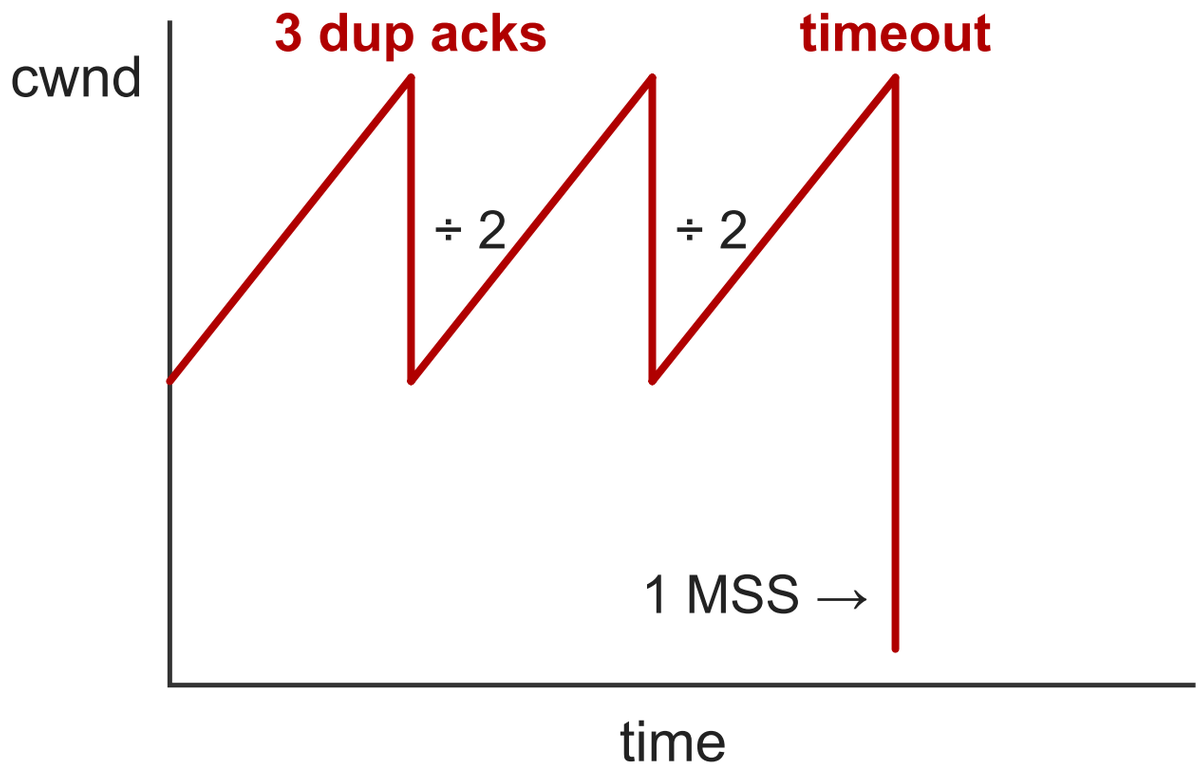
3 duplicate acks:
cut cwnd in half



Classic TCP: AIMD, with both loss signals

3 duplicate acks:
cut cwnd in half

Timeout:
cwnd = 1 MSS

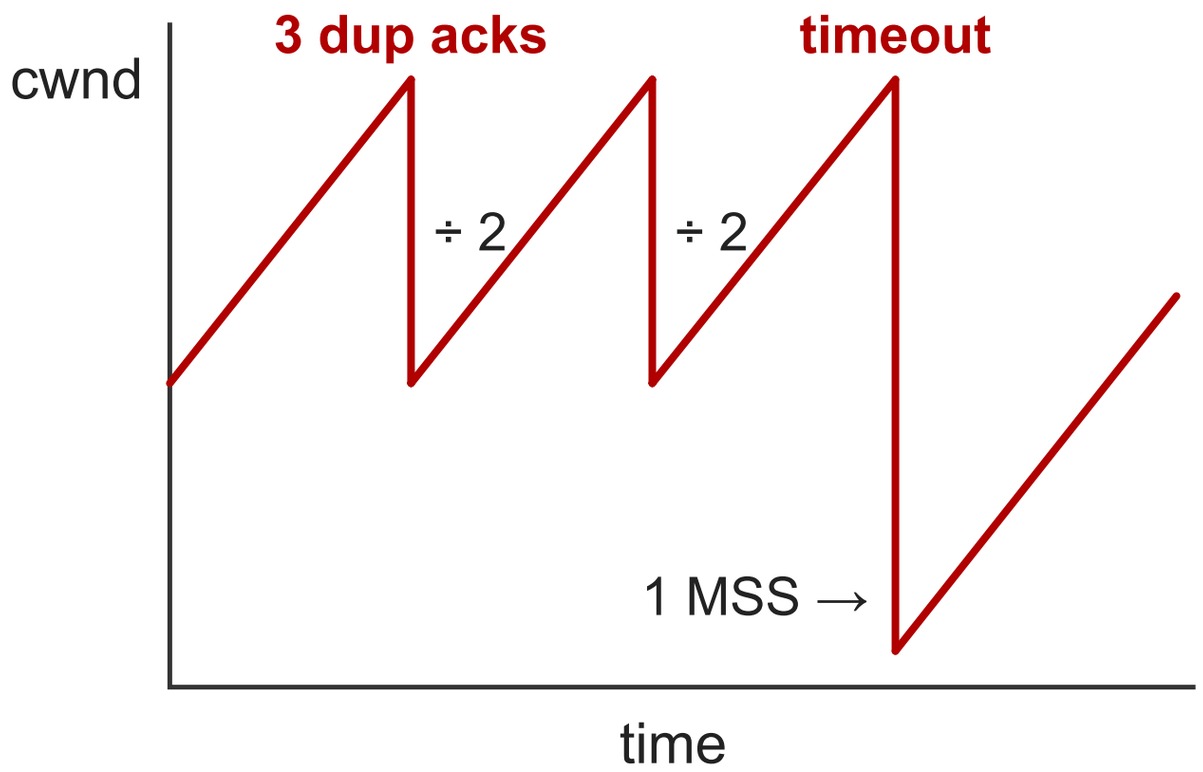


Classic TCP: AIMD, with both loss signals

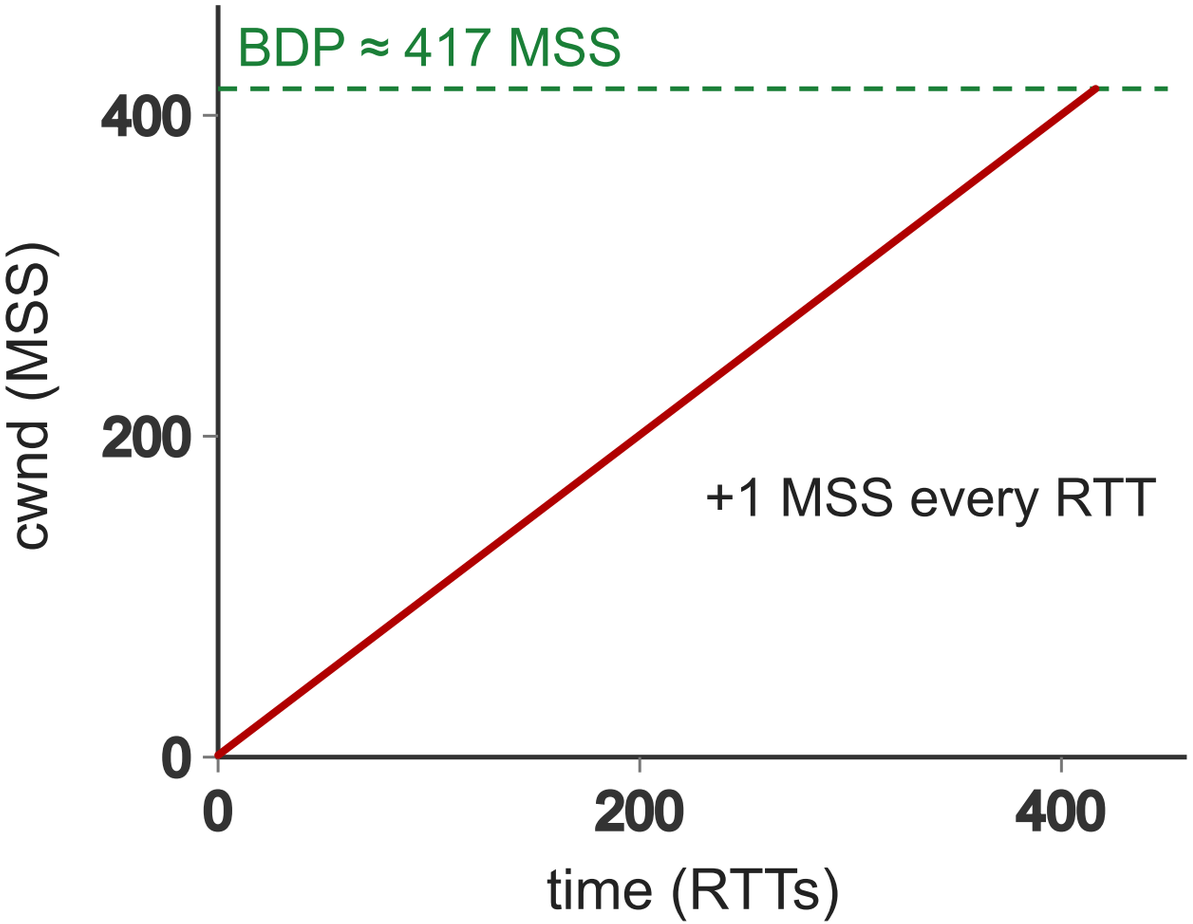
3 duplicate acks:
cut cwnd in half

Timeout:
cwnd = 1 MSS

Then additive increase again,
from 1 MSS, one MSS every RTT



Additive increase is, sometimes, too slow

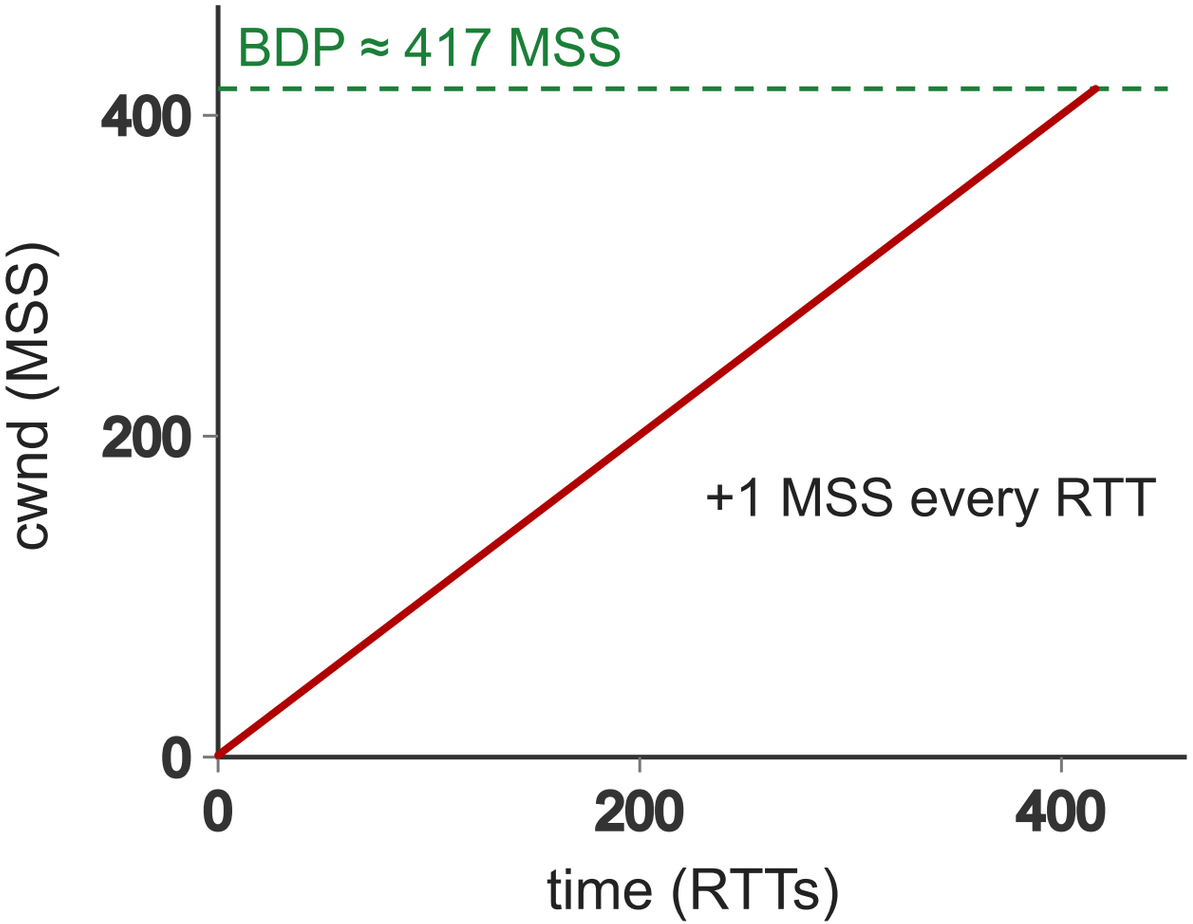


Example: 100 Mb/s, RTT 50 ms, MSS = 1,500 bytes: BDP = 625,000 bytes ≈ 417 MSS

At the start, or after a timeout: cwnd = 1 MSS

+1 MSS every RTT: 416 RTTs ≈ 21 seconds to use the available network capacity

Additive increase is, sometimes, too slow



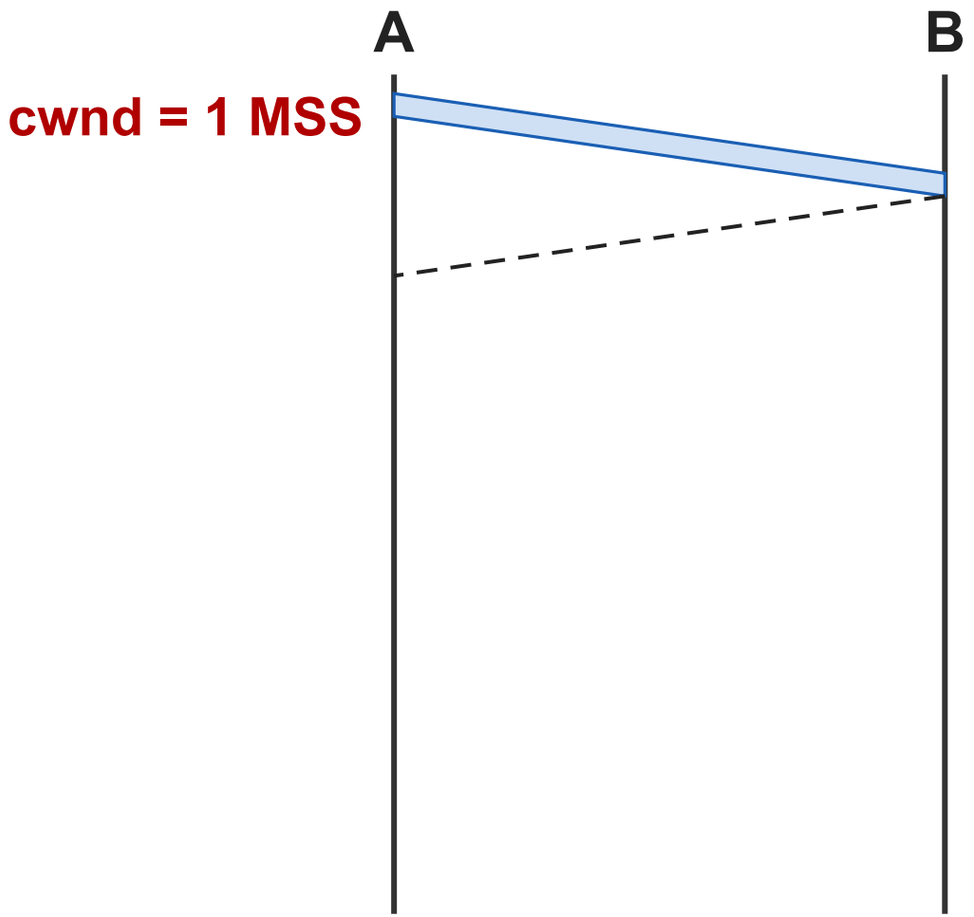
Example: 100 Mb/s, RTT 50 ms, MSS = 1,500 bytes: BDP = 625,000 bytes ≈ 417 MSS

At the start, or after a timeout: cwnd = 1 MSS

+1 MSS every RTT: 416 RTTs ≈ 21 seconds to use the available network capacity

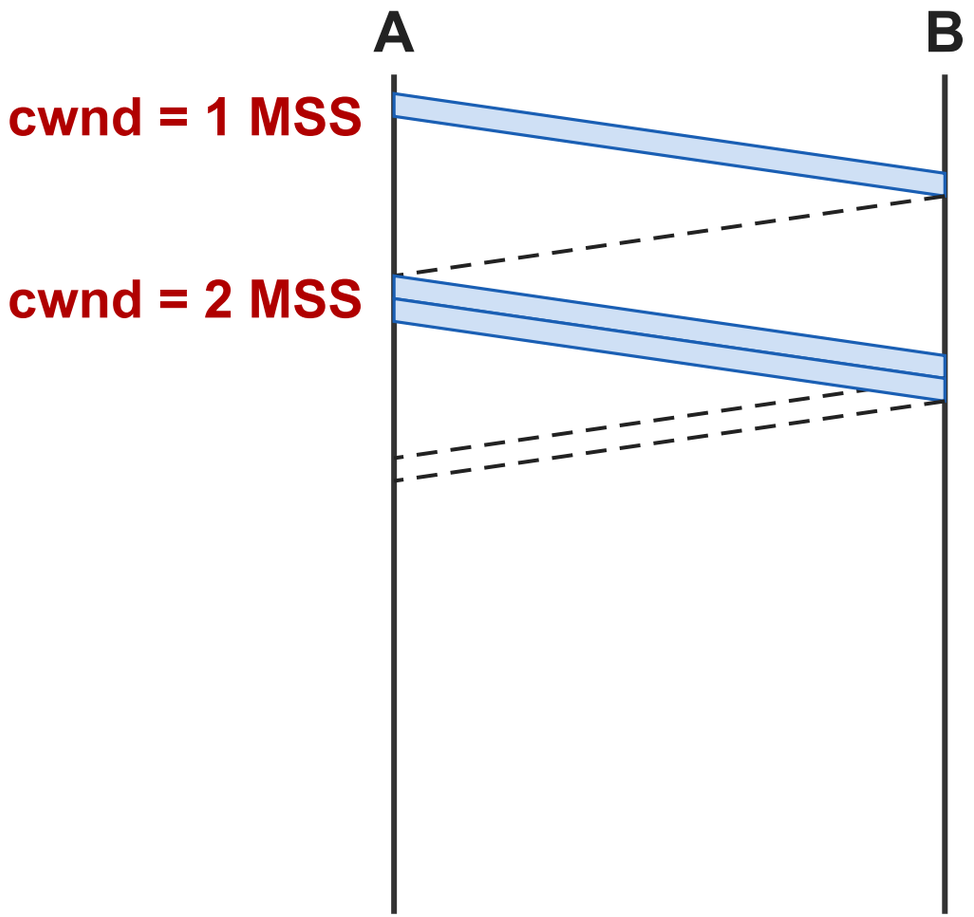
Until then the bottleneck is partly idle

“Slow” Start



Start: cwnd = 1 MSS

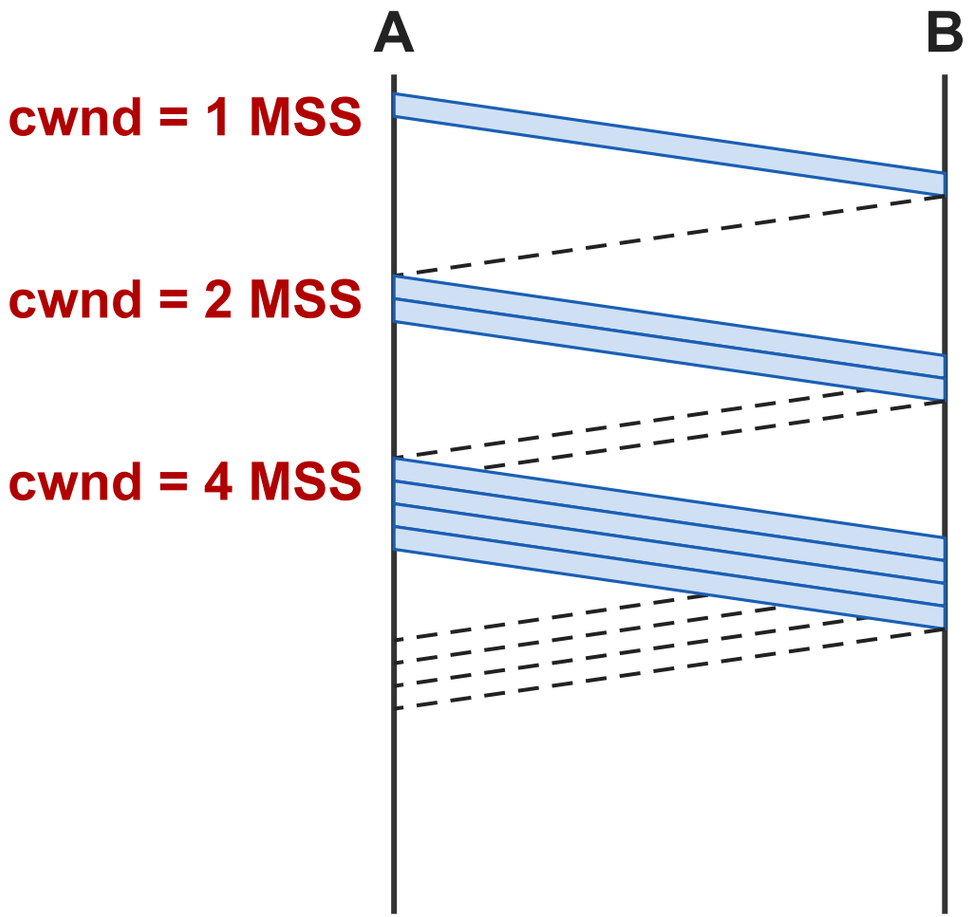
“Slow” Start



Start: cwnd = 1 MSS

+1 MSS to cwnd for every ack:
cwnd doubles every RTT

“Slow” Start

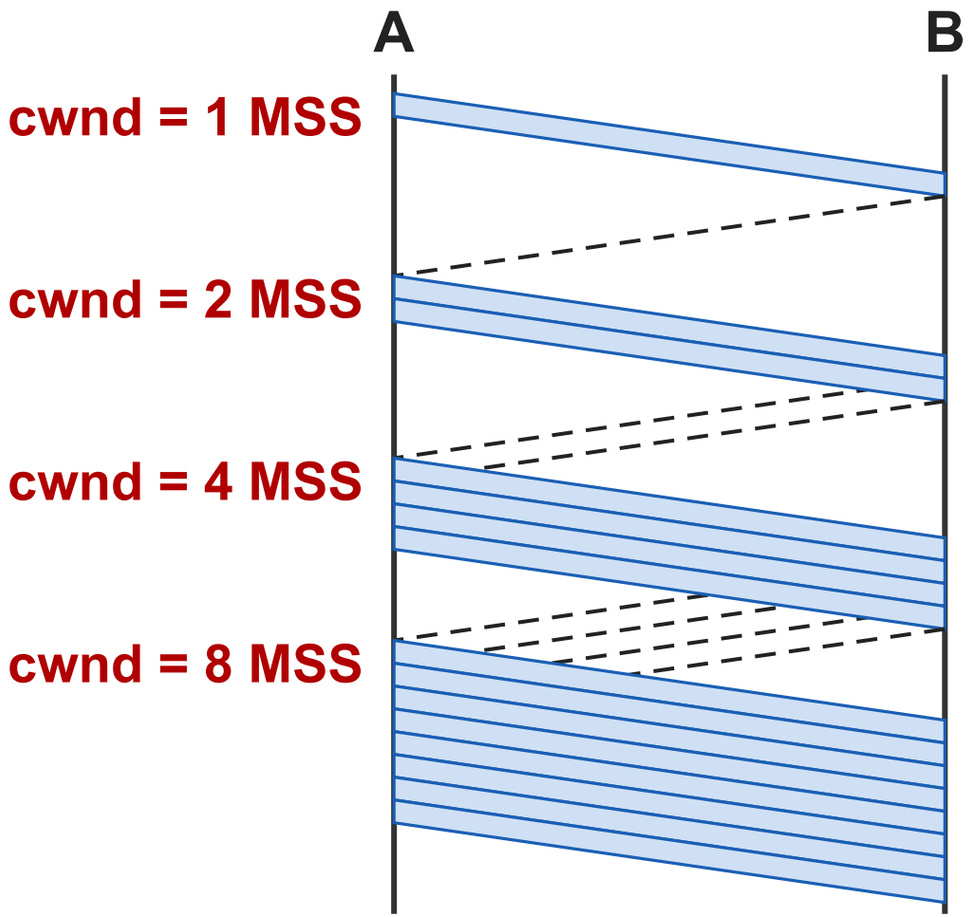


Start: cwnd = 1 MSS

+1 MSS to cwnd for every ack:
cwnd doubles every RTT

From 1 to 417 MSS in about
9 RTTs (0.45 s), not 416

“Slow” Start



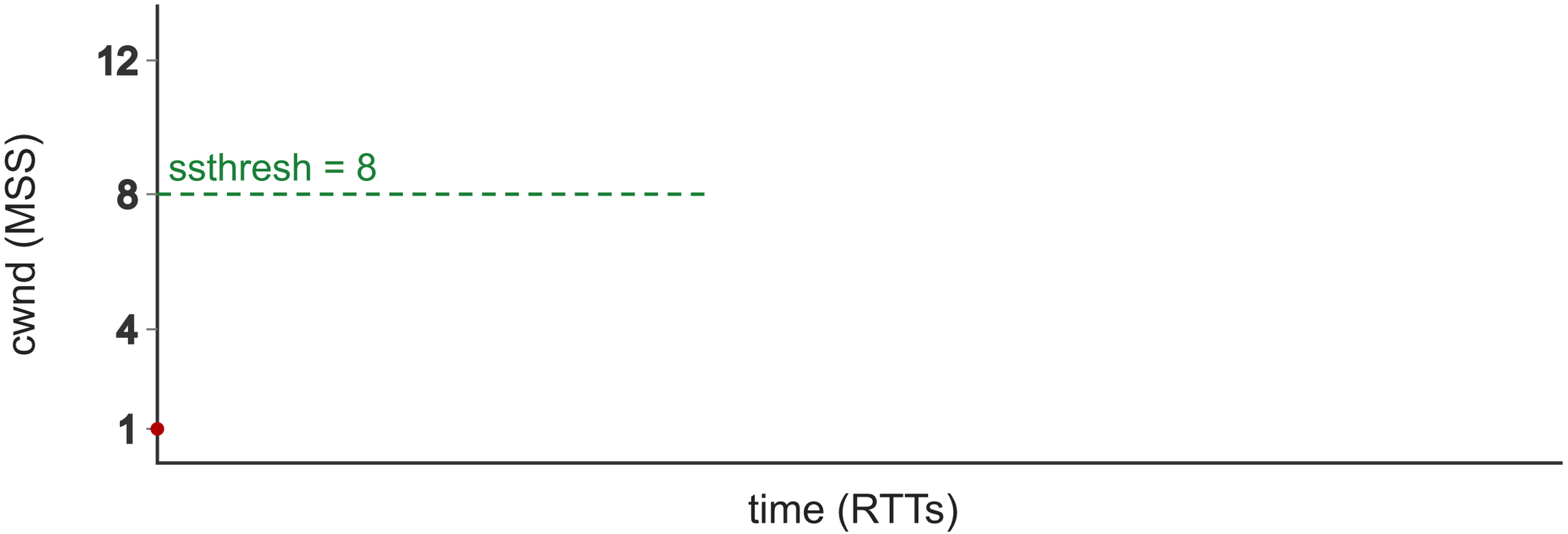
Start: cwnd = 1 MSS

+1 MSS to cwnd for every ack:
cwnd doubles every RTT

From 1 to 417 MSS in about
9 RTTs (0.45 s), not 416

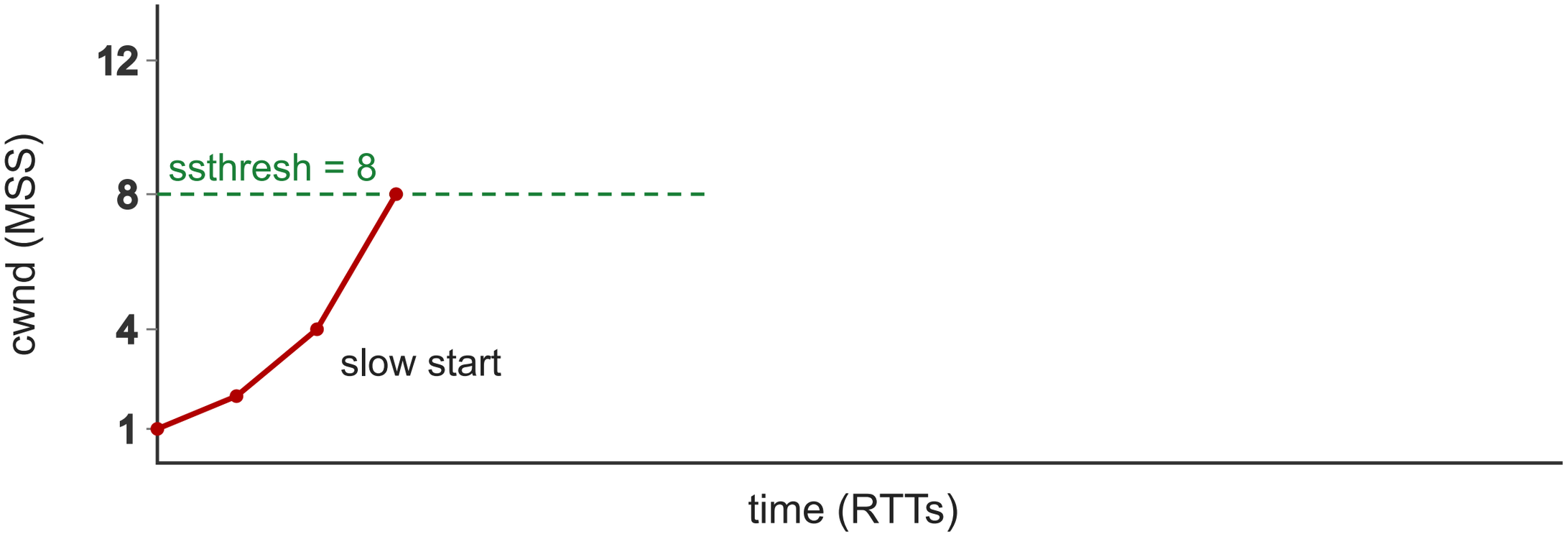
Doubles until the slow start
threshold (ssthresh)

Classic TCP Congestion control



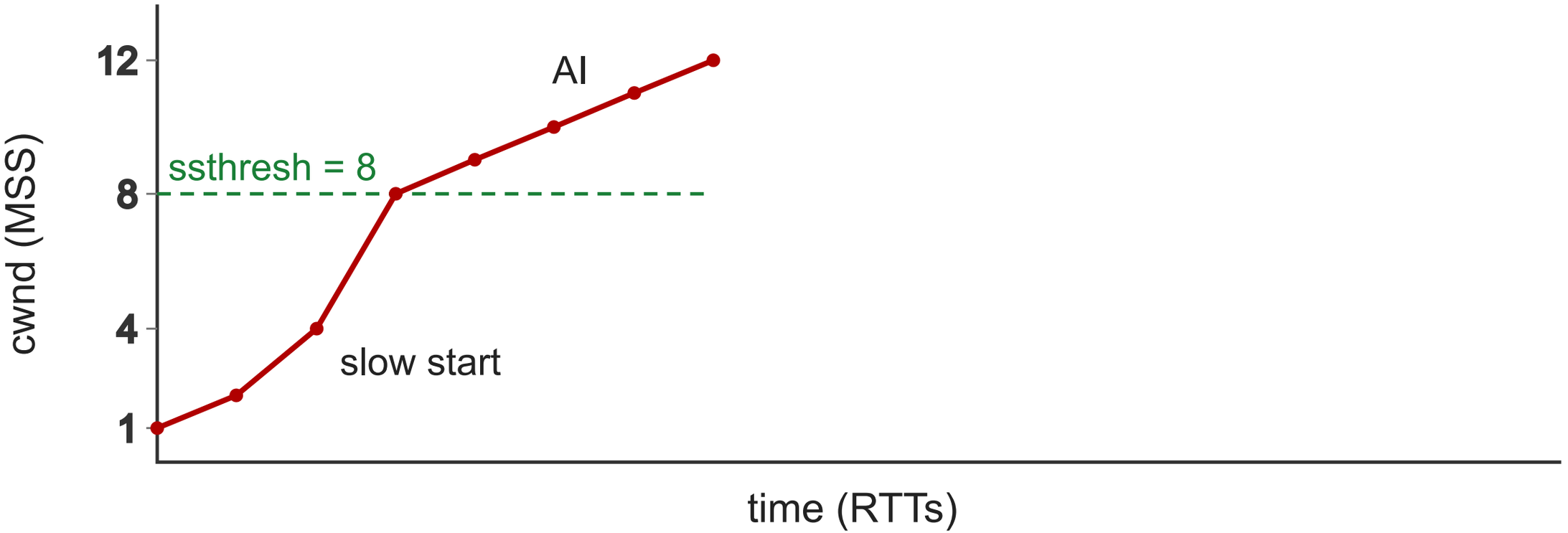
Start: cwnd = 1 MSS, below ssthresh

Classic TCP Congestion control



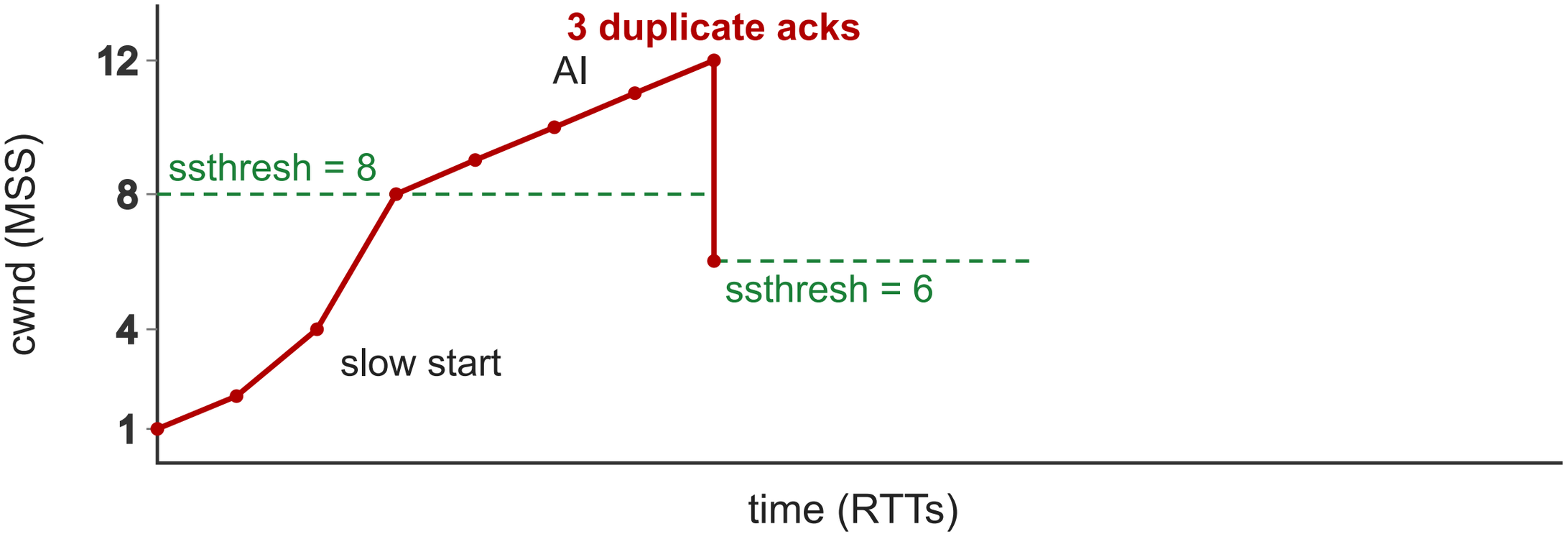
Slow start: cwnd doubles every RTT, up to ssthresh

Classic TCP Congestion control



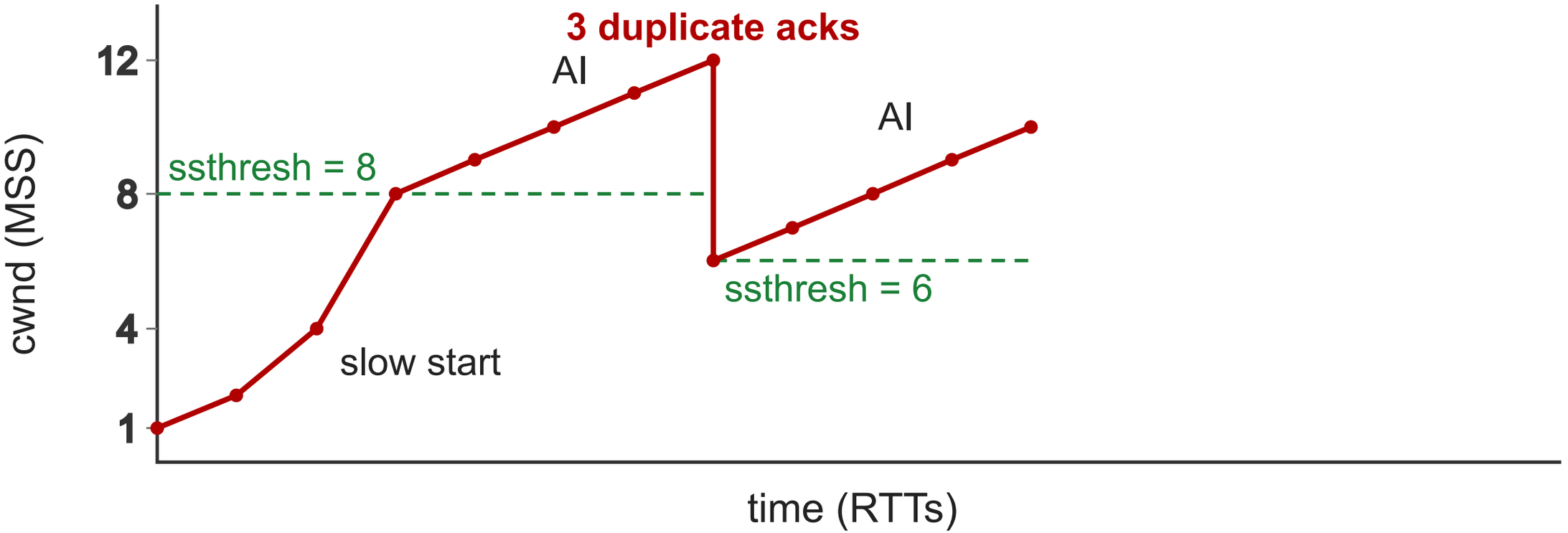
Above ssthresh: additive increase, +1 MSS every RTT

Classic TCP Congestion control



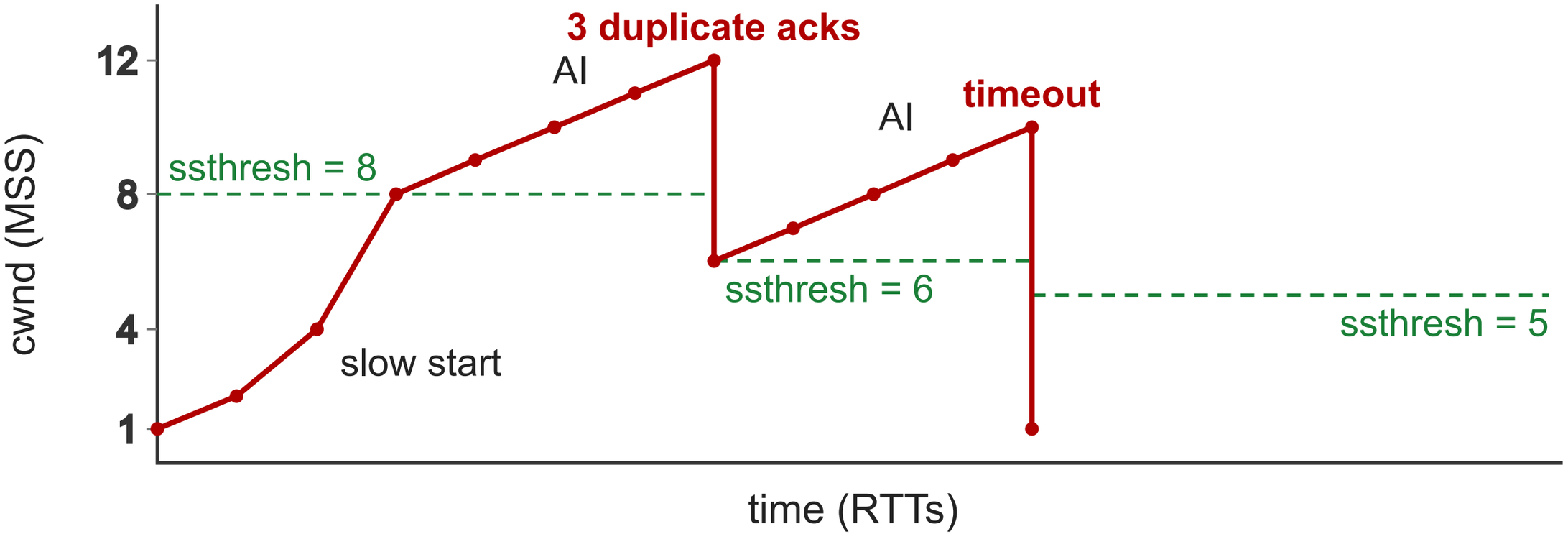
3 duplicate acks: ssthresh = cwnd / 2, cwnd = ssthresh

Classic TCP Congestion control



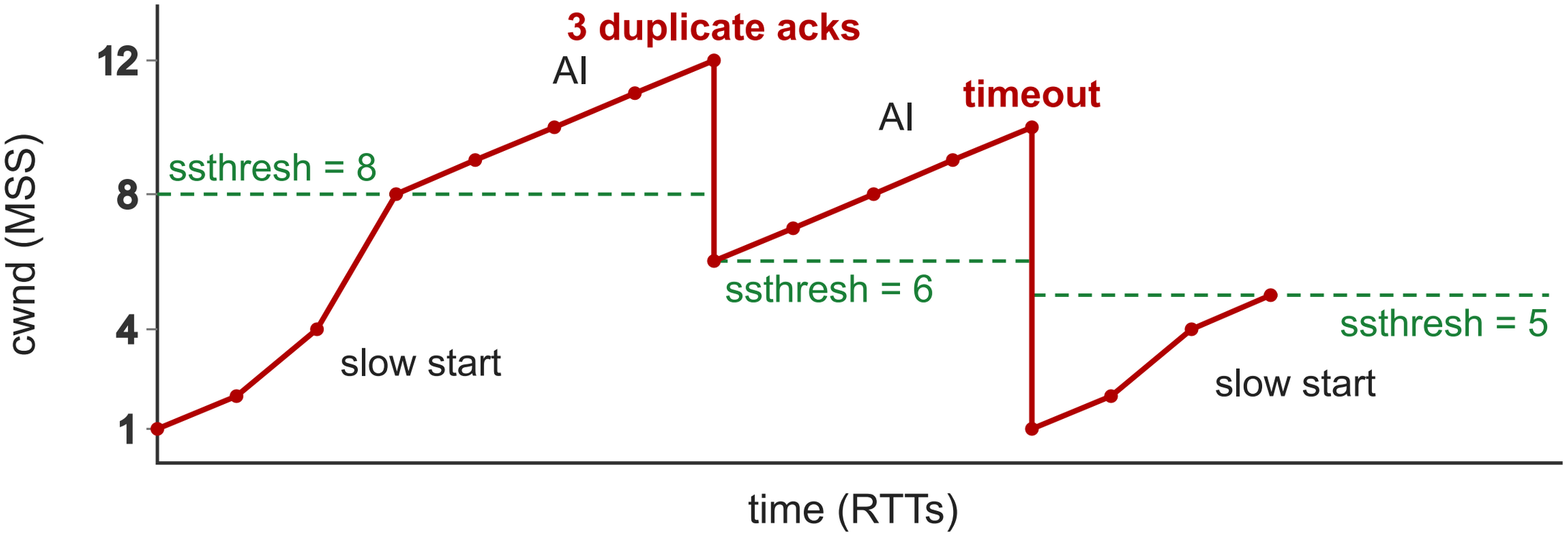
Additive increase again

Classic TCP Congestion control



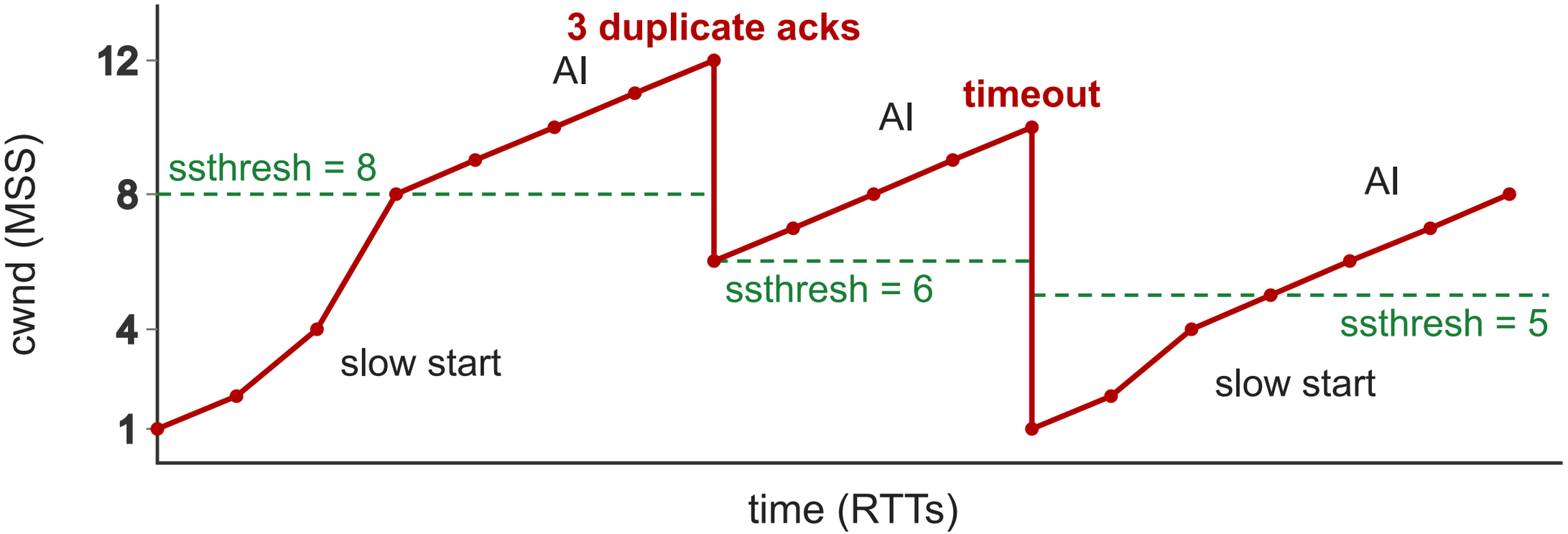
Timeout: ssthresh = cwnd / 2, cwnd = 1 MSS

Classic TCP Congestion control



Slow start again, up to the new ssthresh

Classic TCP Congestion control



Then additive increase

There are even more details (e.g., inflating cwnd with duplicate acks), but we won't get into them in this course

Classic TCP on a fast, long path

Classic TCP on a fast, long path

- Example: the bottleneck link rate is 10 Gbps, RTT 100 ms, MSS = 1,500 bytes, one flow: BDP ≈ 83,000 MSS

Classic TCP on a fast, long path

- Example: the bottleneck link rate is 10 Gbps, RTT 100 ms, MSS = 1,500 bytes, one flow: BDP ≈ 83,000 MSS
- cwnd grows until the queue overflows: a loss, and cwnd halves to about 41,700 MSS

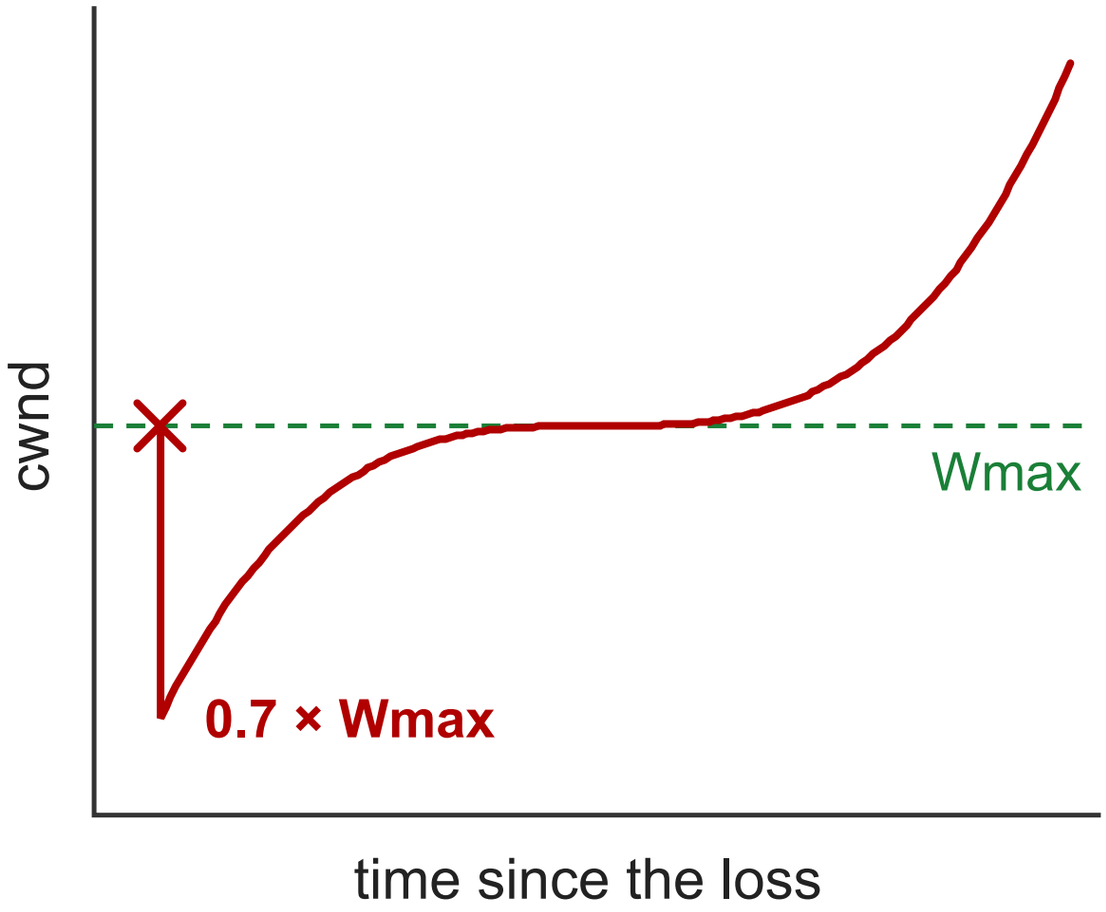
Classic TCP on a fast, long path

- Example: the bottleneck link rate is 10 Gbps, RTT 100 ms, MSS = 1,500 bytes, one flow: BDP ≈ 83,000 MSS
- cwnd grows until the queue overflows: a loss, and cwnd halves to about 41,700 MSS
- If the capacity available to the flow is still close to what it was, it should get back near 83,000 MSS

Classic TCP on a fast, long path

- Example: the bottleneck link rate is 10 Gbps, RTT 100 ms, MSS = 1,500 bytes, one flow: BDP ≈ 83,000 MSS
- cwnd grows until the queue overflows: a loss, and cwnd halves to about 41,700 MSS
- If the capacity available to the flow is still close to what it was, it should get back near 83,000 MSS
- But at +1 MSS every RTT, that takes about 41,700 RTTs: over an hour, with the link partly idle

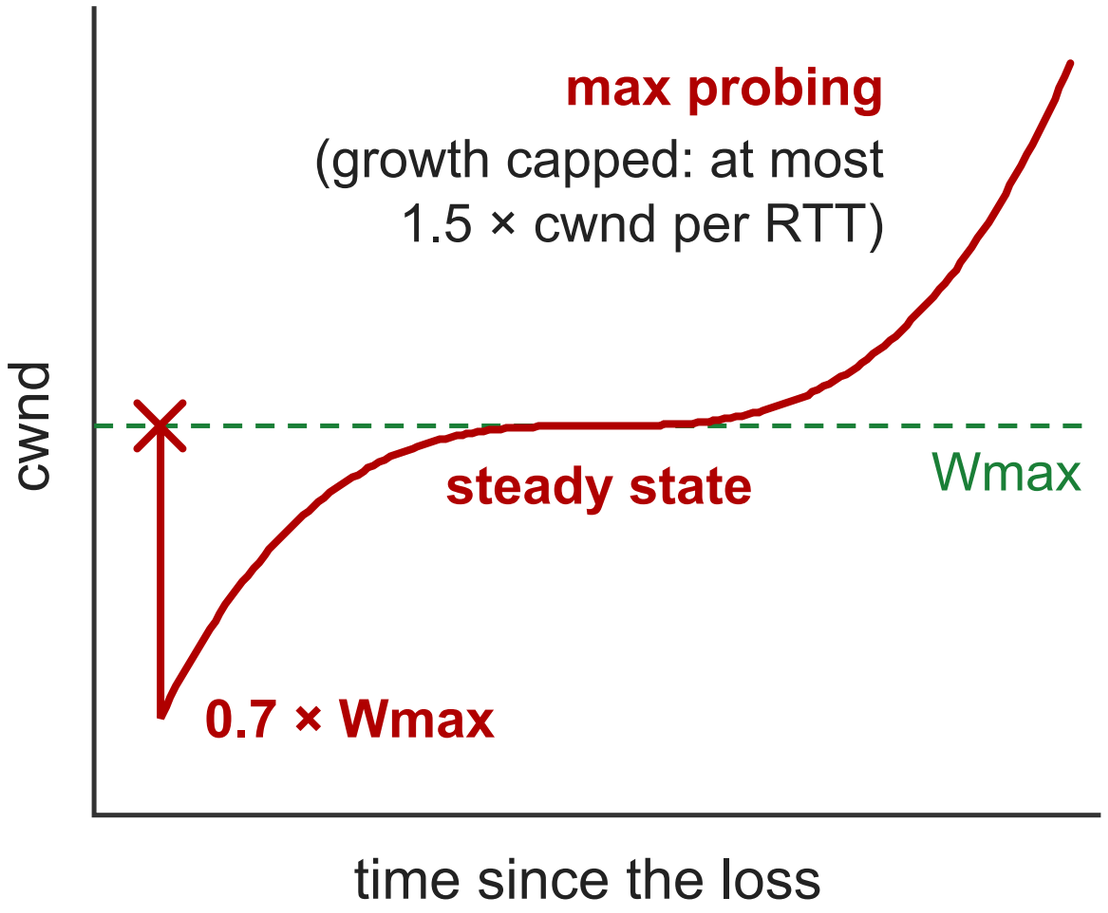
TCP CUBIC



On a loss: remember Wmax, the cwnd where the path last got congested; cut cwnd to 0.7 × Wmax

Then $W(t) = C\,(t - K)^3 + Wmax$,
t: the time since the loss, by the sender's clock (seconds)

TCP CUBIC



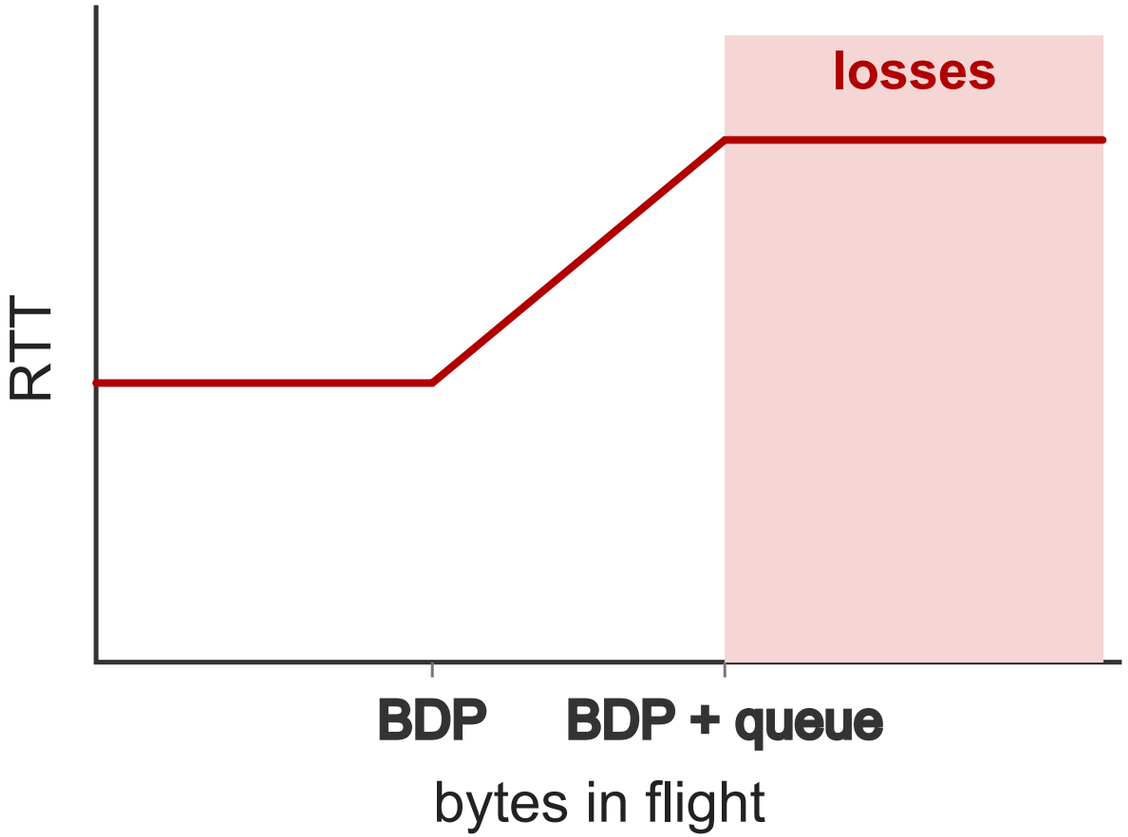
On a loss: remember Wmax, the cwnd where the path last got congested; cut cwnd to 0.7 × Wmax

Then $W(t) = C\,(t - K)^3 + Wmax$,
t: the time since the loss, by the sender's clock (seconds)

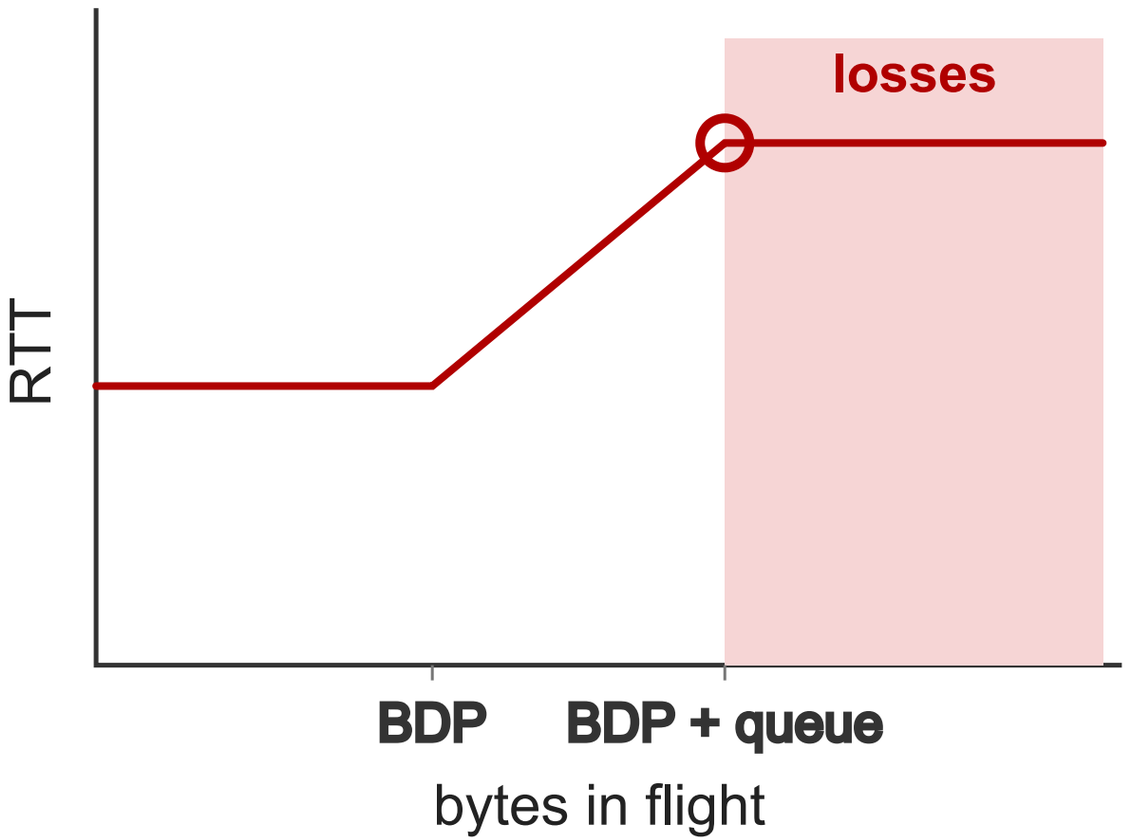
Fast while far below Wmax, slow near it (steady state), faster again past it (max probing)

cwnd is adjusted based on time not RTT: flows with different RTTs grow their cwnd at the same pace

Loss-based algorithms fill the queue

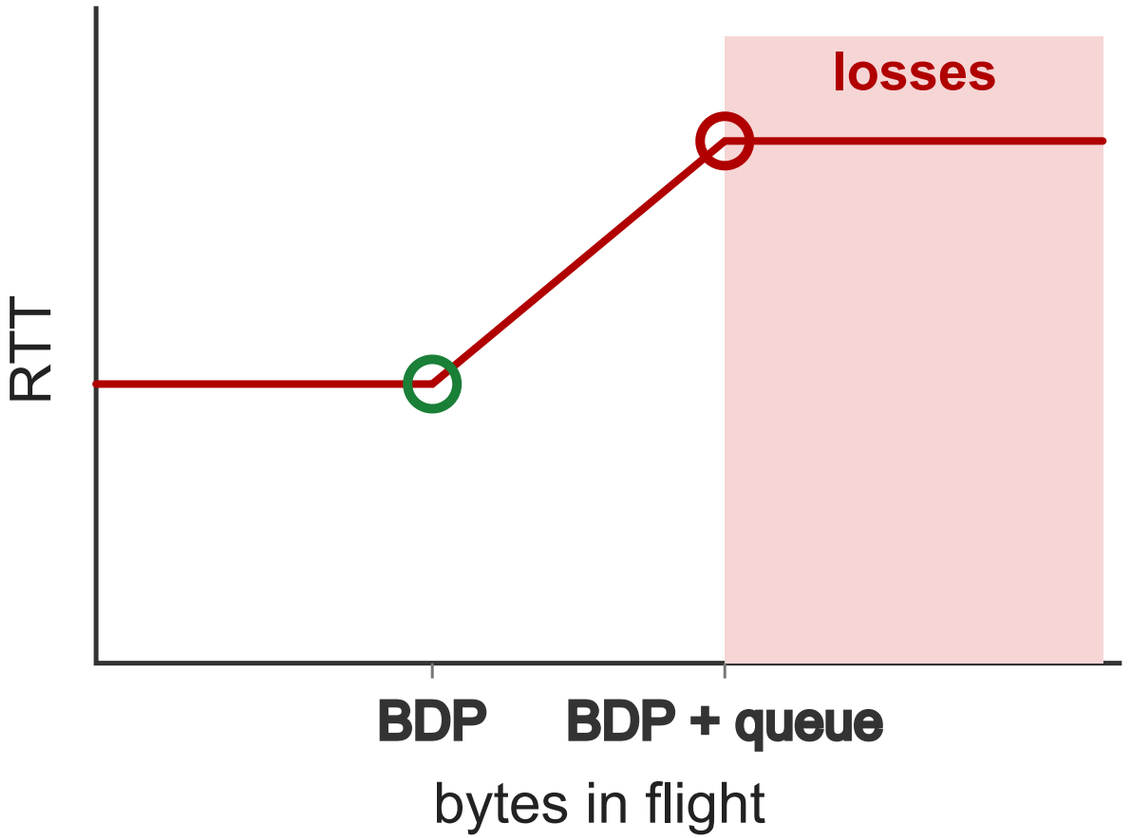


Loss-based algorithms fill the queue



Loss-based (classic TCP, CUBIC) only detect congestion and back off here, when the queue is already full and the loss has already happened

Loss-based algorithms fill the queue



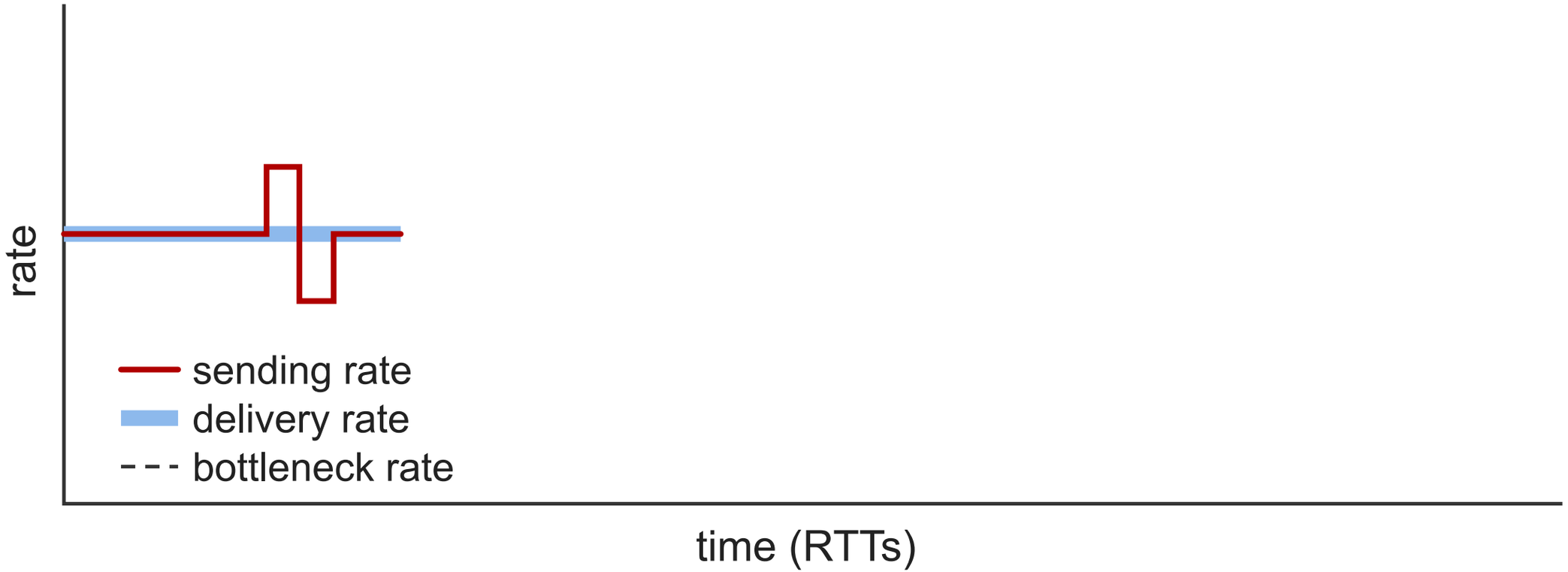
Loss-based (classic TCP, CUBIC) only detect congestion and back off here, when the queue is already full and the loss has already happened

Ideally, we would like to aim for "the knee": one BDP in flight, the link full, no queue

TCP BBR (version 1)

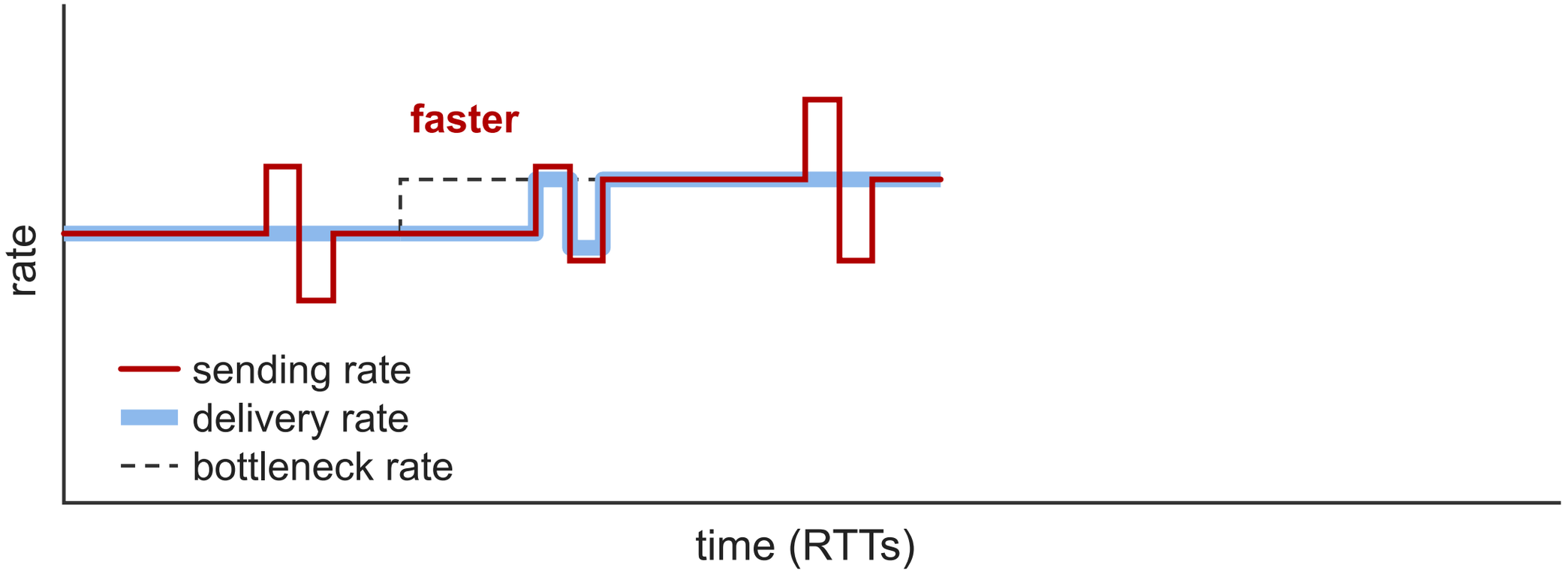
- Google, 2016: **B**ottleneck **B**andwidth and **R**ound-trip propagation time
- Keeps track of:
  - BtlBw: the highest delivery rate seen in the last ~10 RTTs
  - RTprop: the lowest RTT seen in the last ~10 seconds
- Does not use the sliding window for pacing. Uses a rate limiter to pace packets at BtlBw
- For safety, bounds in-flight packets to 2 × BDP (BtlBw × RTprop is BBR's estimate of BDP)

BBR in steady state



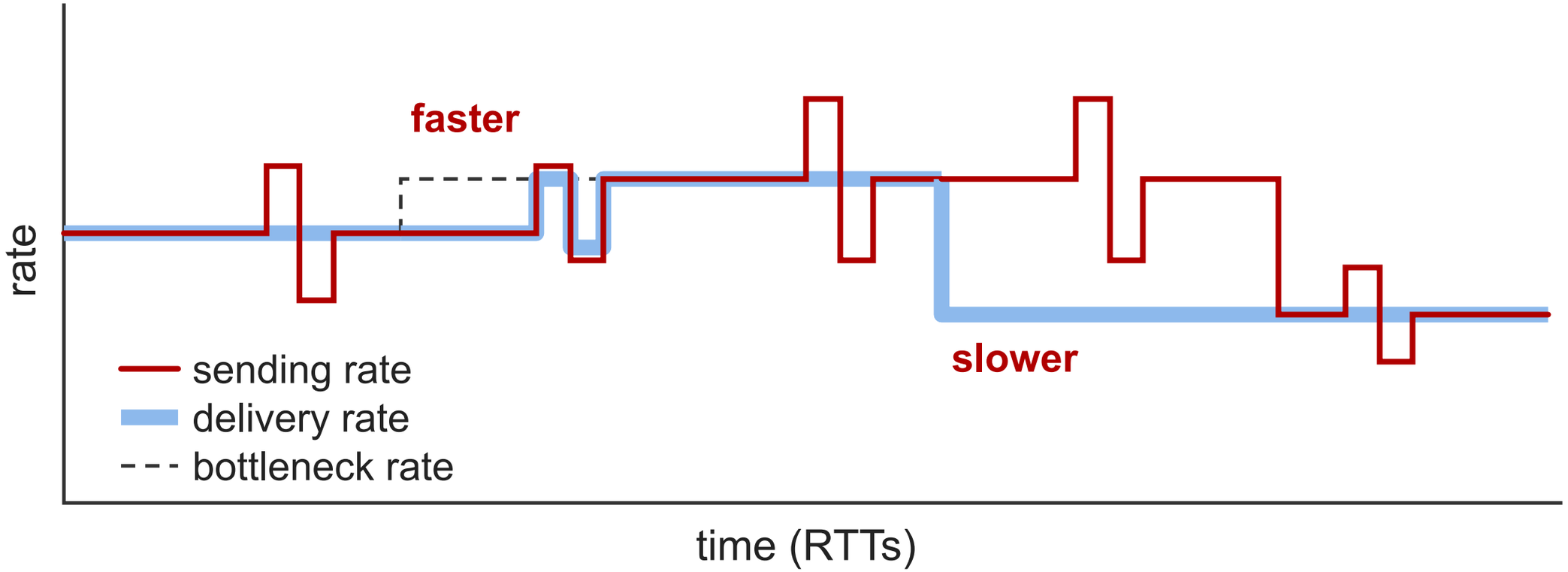
Steady state: six RTTs at BtlBw, one at 1.25 × BtlBw (does the delivery rate go up?), one at 0.75 × to drain what that queued.

BBR in steady state



The bottleneck gets faster (another flow left). During the next probe, the delivery rate goes up with the sending rate, so BtlBw goes up to the new bottleneck rate.

BBR in steady state



The bottleneck gets slower (another flow arrived): the delivery rate drops at once, but BtlBw is the highest of the last ~10 RTTs, so BBR sends too fast and a queue builds. Then its rate settles at the new rate, but it does not drain that queue right away.

What if the network helps?

In lecture 10, we discussed the following:

- You are a sender communicating with a receiver. **You don’t get any extra information from the network** or other senders and receivers that share the network with you.
- How can you figure out your “right” share, and notice when it changes?

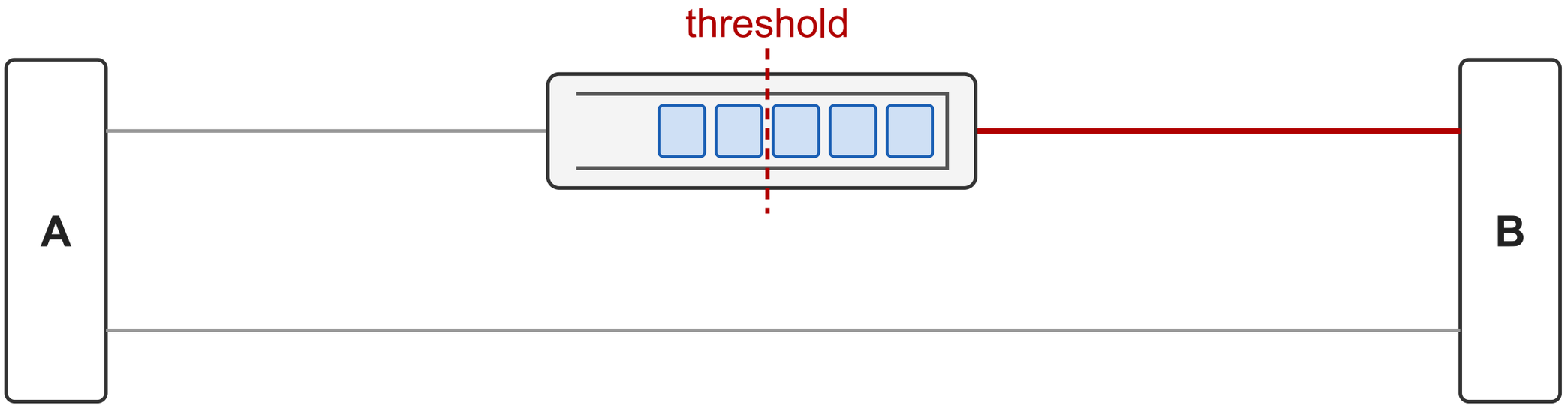
What if the network helps?

In lecture 10, we discussed the following:

- You are a sender communicating with a receiver. **You don't get any extra information from the network** or other senders and receivers that share the network with you.
- How can you figure out your "right" share, and notice when it changes?

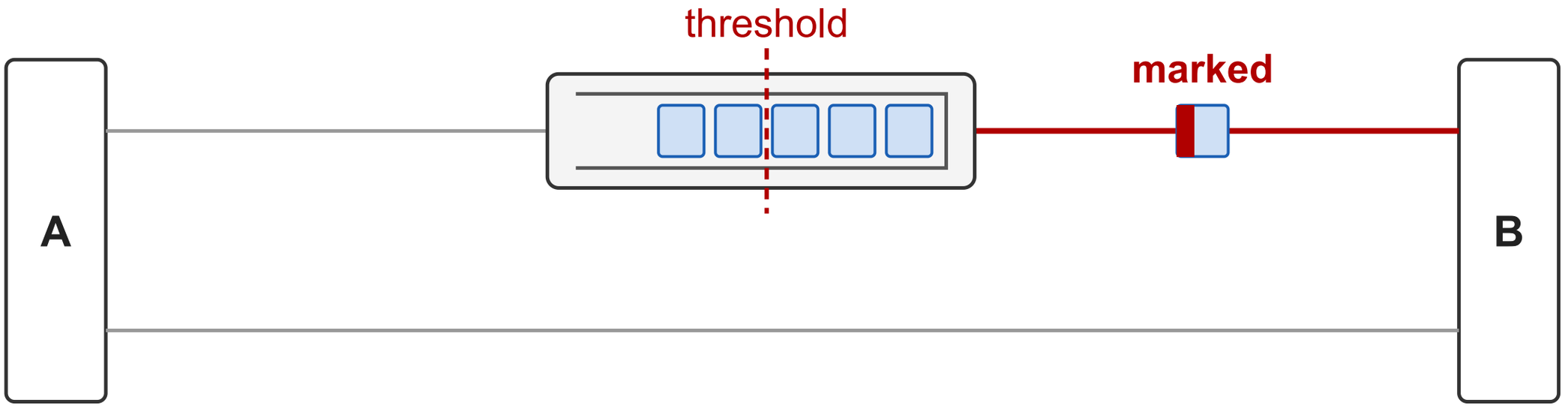
What if the network helps?

ECN: the switch marks packets



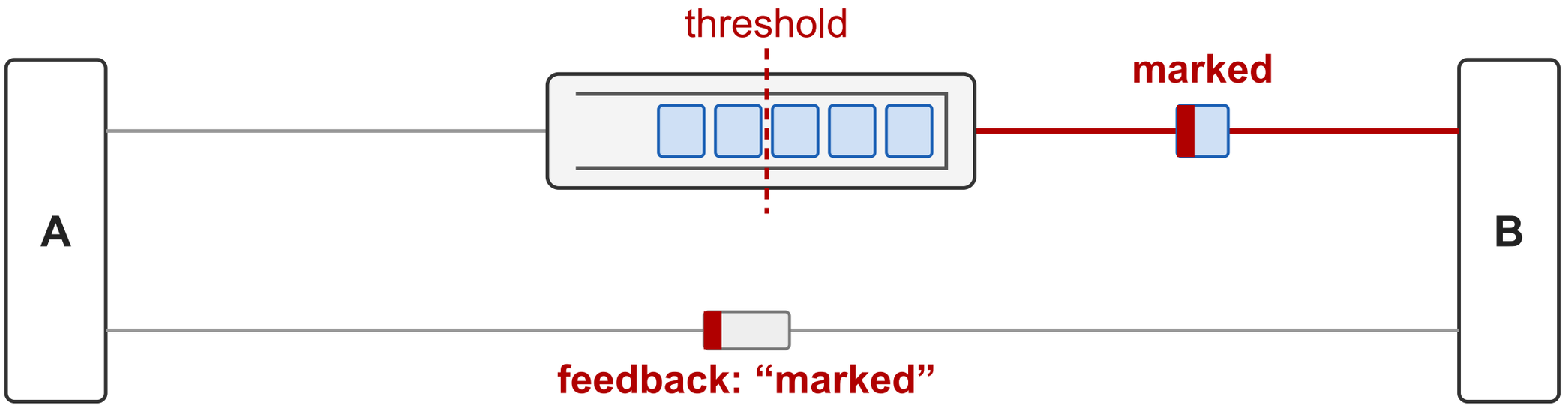
The switch’s queue is past a threshold: congestion is starting, nothing is lost yet.

ECN: the switch marks packets



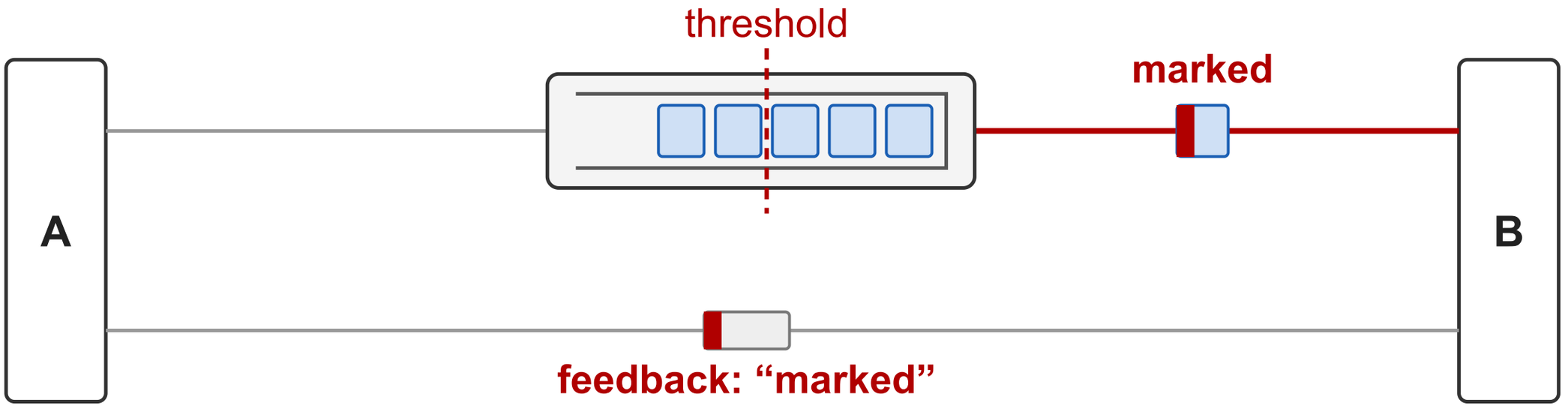
The switch sets a bit in the packet's metadata: ECN, Explicit Congestion Notification (RFC 3168, 2001).

ECN: the switch marks packets



B sees the mark and echoes it to A in its TCP metadata (the feedback).

ECN: the switch marks packets



A slows down as if a packet were lost (classic TCP halves cwnd), but nothing was lost and nothing has to be resent. Doing this requires support from the switches and both endpoints.

DCTCP (Data-Center TCP)

- TCP with ECN: halve cwnd on a marked packet, at most once per RTT
- DCTCP: track α = fraction of marked packets, averaged over time; every RTT, cwnd ← cwnd × (1 − α / 2)
  - Only a few marked → small decrease. Every packet marked → halve the congestion window
- **DCTCP estimates the extent of congestion from the fraction of marked packets, and reacts accordingly**

Fairness?

- Congestion control: each flow adjusts to its “fair” share of the network

Fairness?

- Congestion control: each flow adjusts to its “fair” share of the network
- How should “fair” be defined?

Fairness?

- Congestion control: each flow adjusts to its “fair” share of the network
- How should “fair” be defined?
- Imagine traffic from N connections (transport channels) sharing a bottleneck link. What is fair?

Equal shares per connection?

- For AIMD: equal shares if flows have similar RTTs. Otherwise, the one with the shorter RTT can get more of the link. Why?

Equal shares per connection?

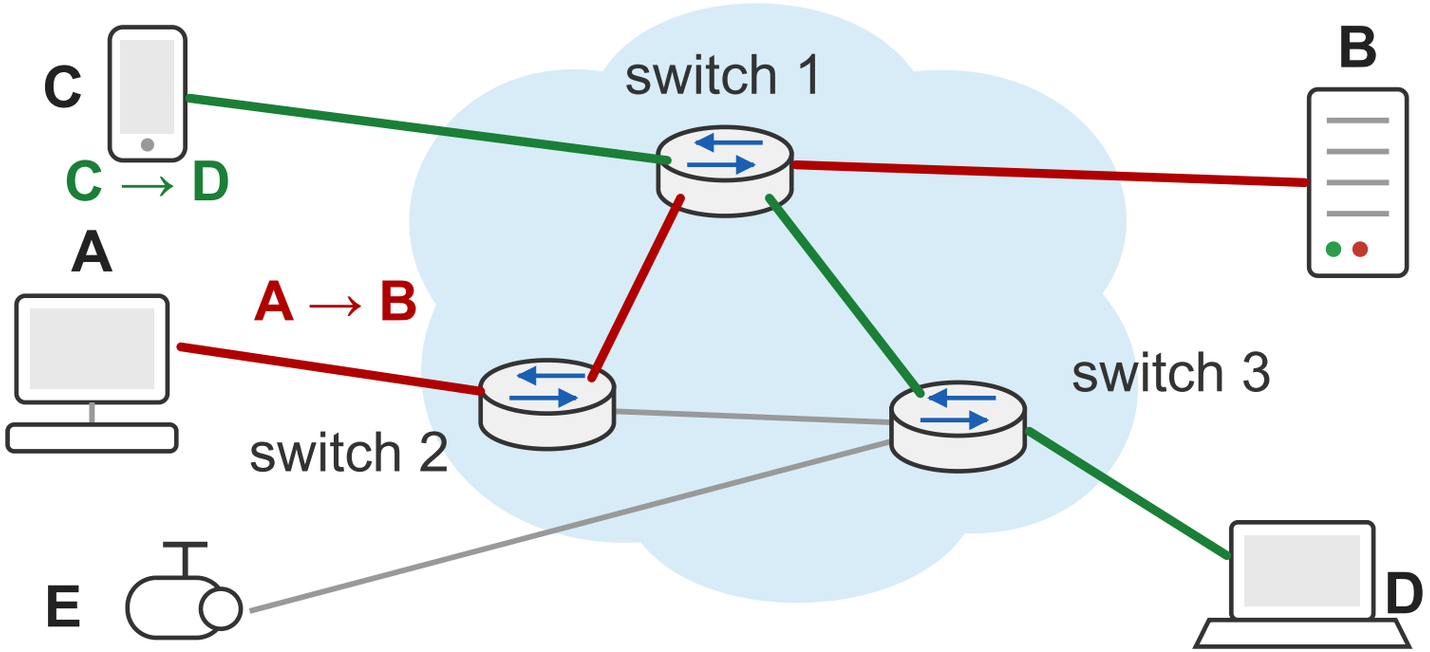
- For AIMD: equal shares if flows have similar RTTs. Otherwise, the one with the shorter RTT can get more of the link. Why?
- CUBIC, BBR, and DCTCP improve on this

Equal shares per connection?

- For AIMD: equal shares if flows have similar RTTs. Otherwise, the one with the shorter RTT can get more of the link. Why?
- CUBIC, BBR, and DCTCP improve on this
- But, most of the time, the guarantee is only between flows of the same kind

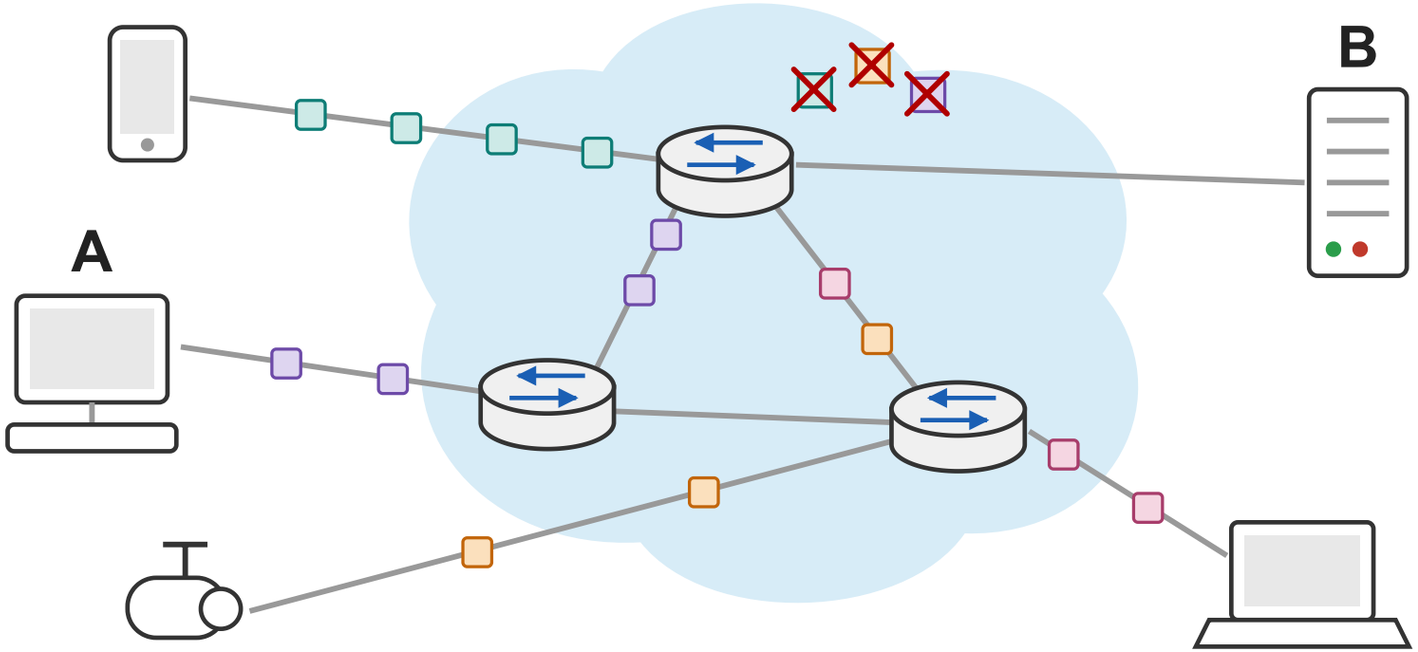
Problem 1: Finding paths

The forwarding table in each switch should be populated such that all endpoints are reachable from each other, even under link and switch failures

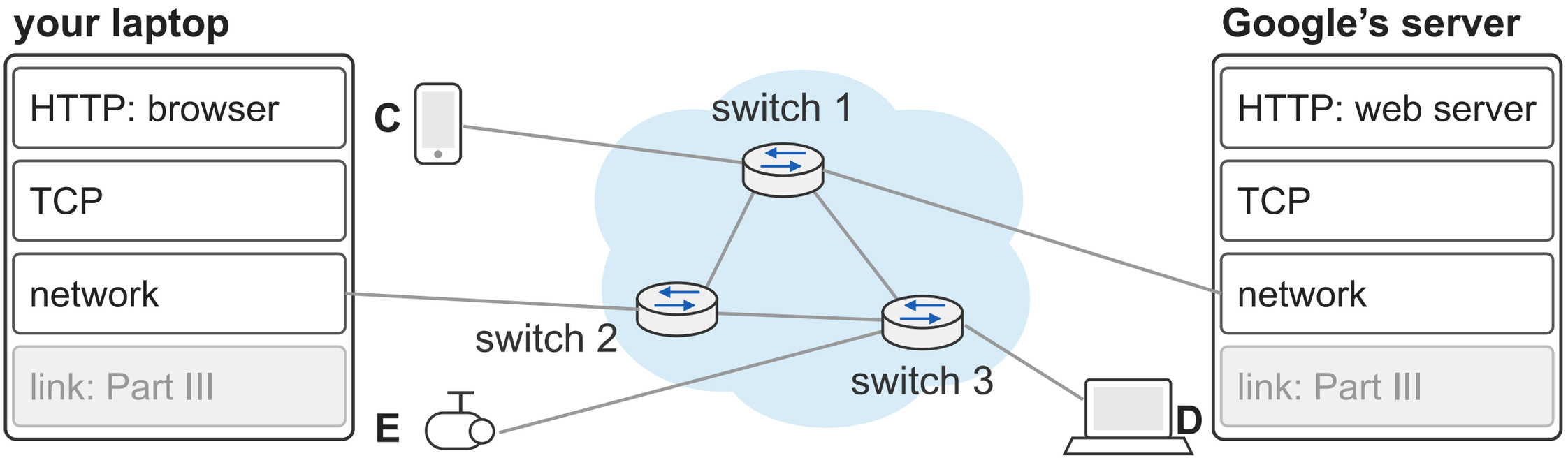


Problem 2: Congestion control

In a packet switched network, how can we ensure endpoints can utilize available network capacity without creating congestion?

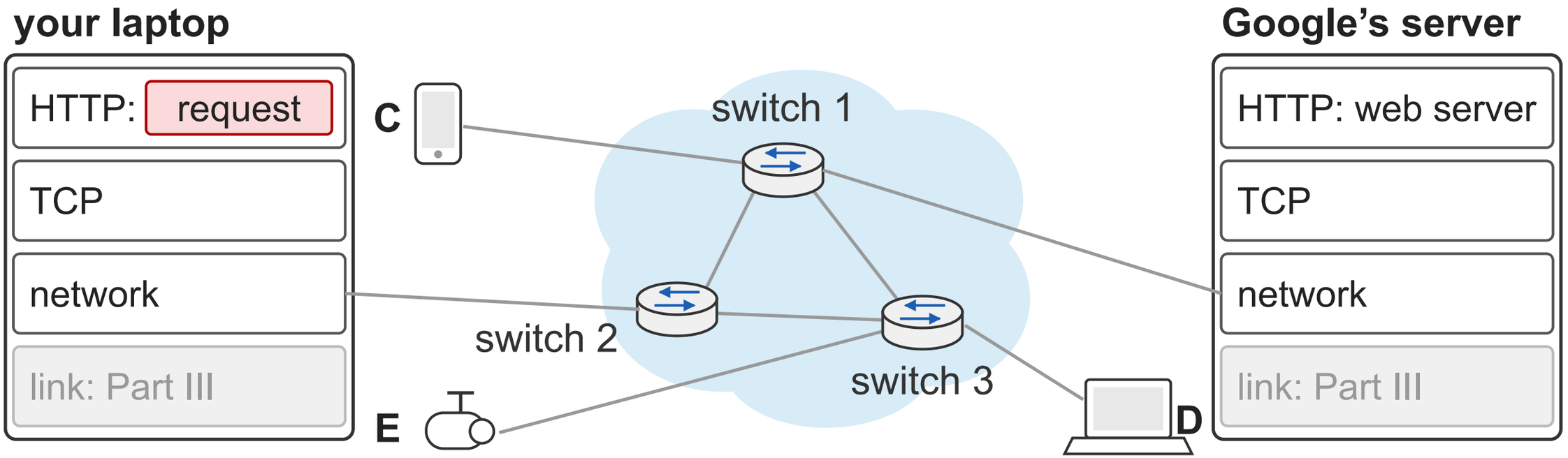


All together so far: typing www.google.com



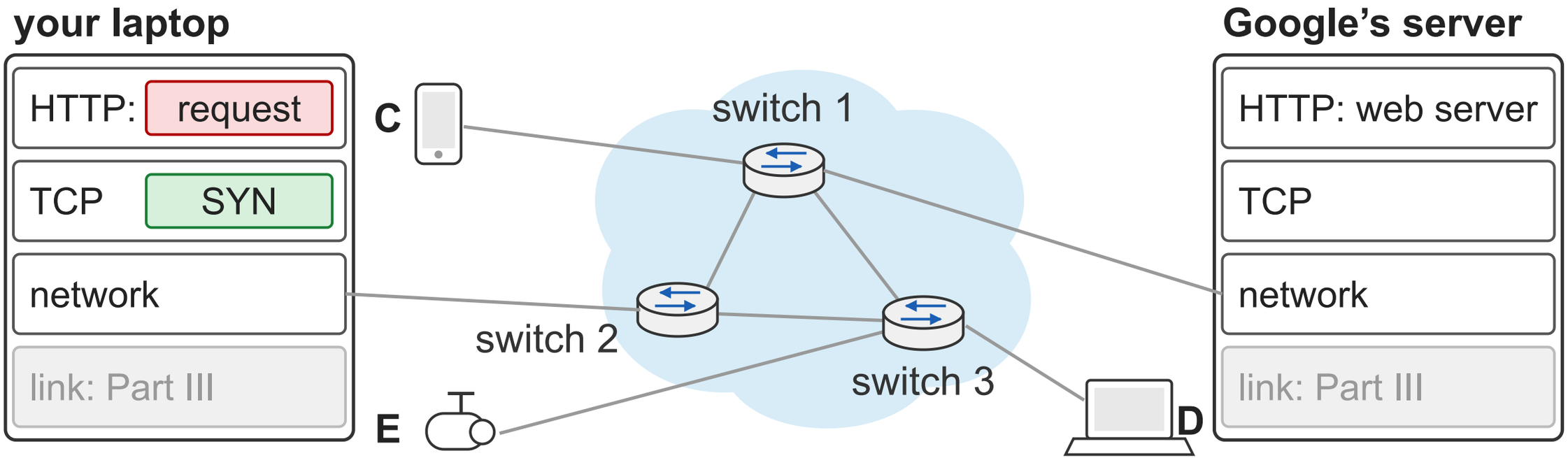
You open your laptop and type www.google.com.

All together so far: typing www.google.com



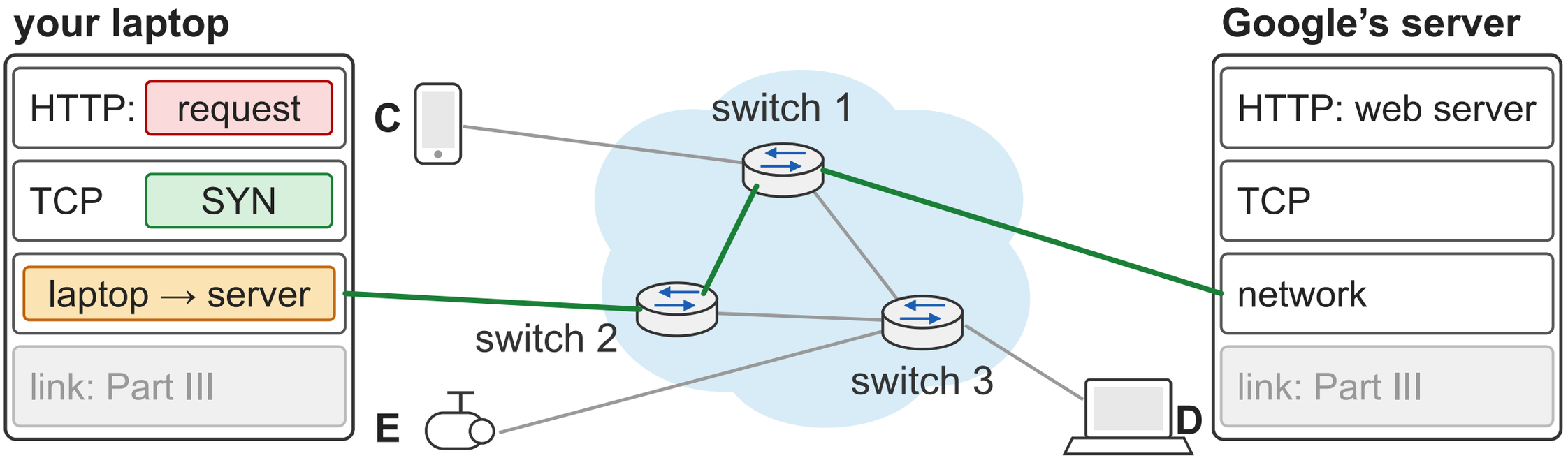
The browser writes an HTTP request (GET /index.html) and asks TCP for a channel to the web server.

All together so far: typing www.google.com



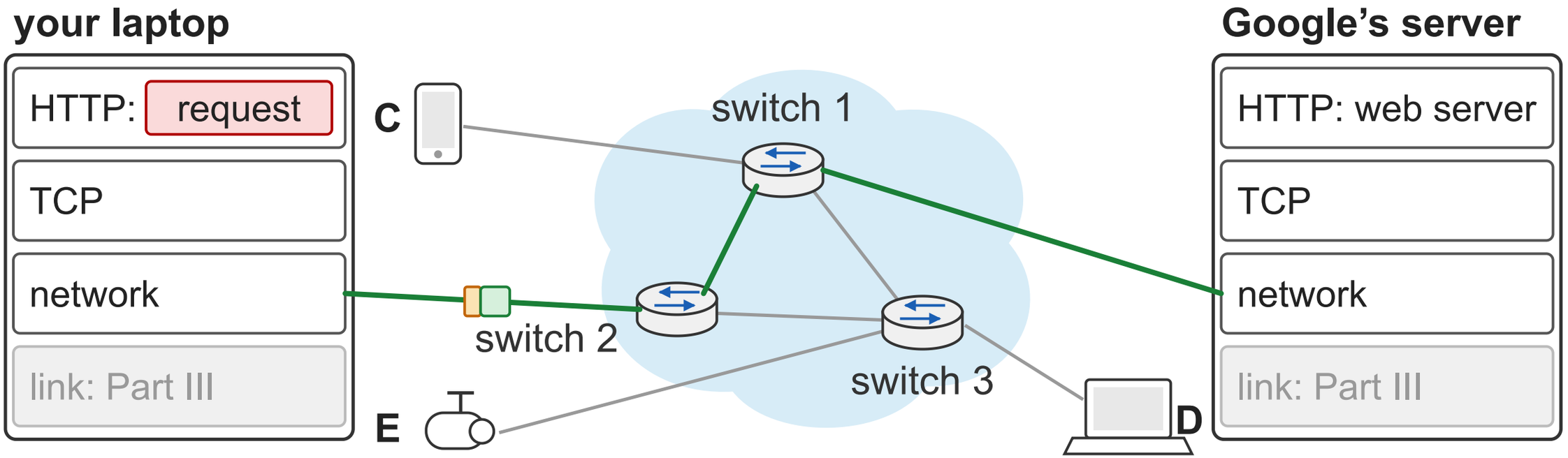
TCP starts the handshake with a SYN.

All together so far: typing www.google.com



The network layer adds its own metadata to the segment: the source and destination addresses (laptop → server). The control plane has already filled the switches' tables.

All together so far: typing www.google.com

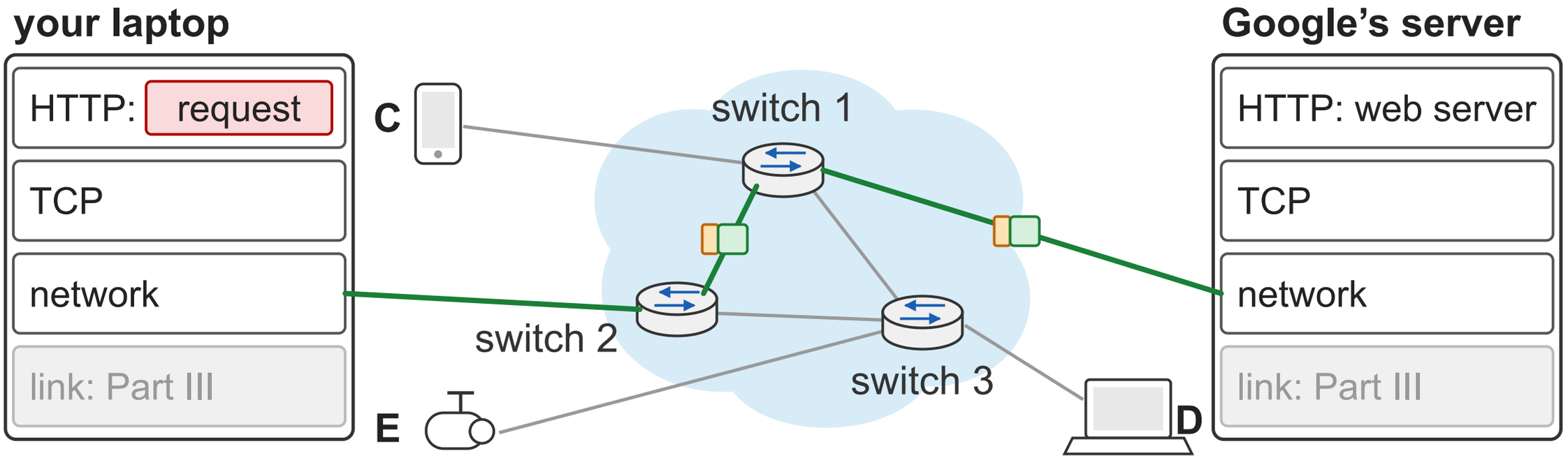


Switch 2 looks up the destination in its table: the link to switch 1.

switch 2's forwarding table
to the server: the link to switch 1

switch 1's forwarding table
to the server: the link to the server

All together so far: typing www.google.com

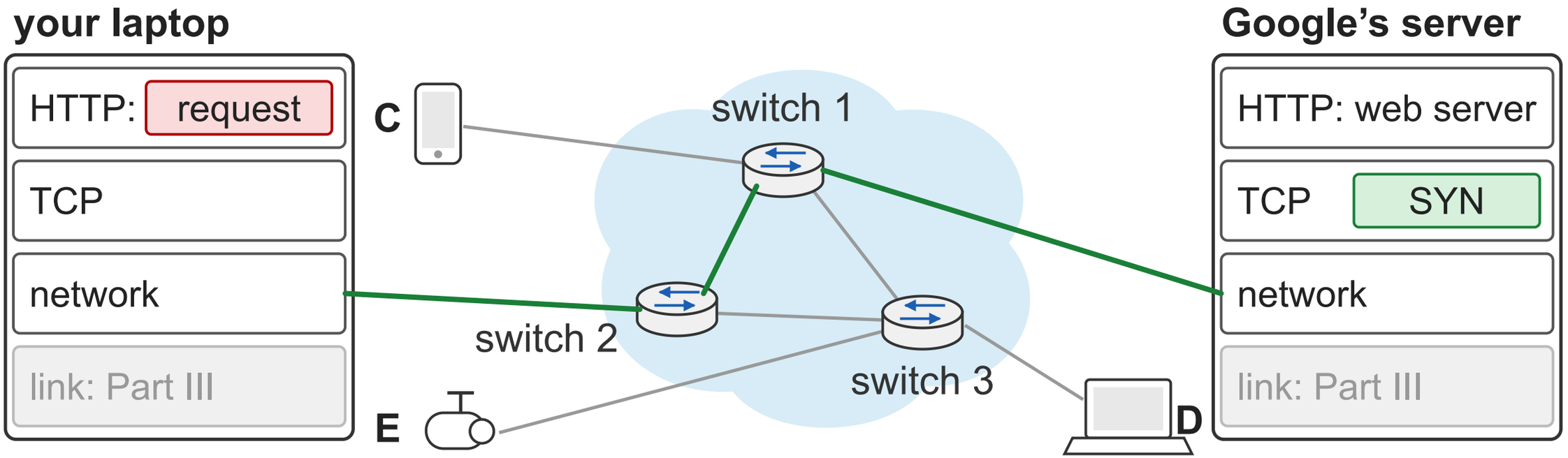


Switch 1 does the same (one lookup per packet). The SYN reaches the server.

switch 2's forwarding table
to the server: the link to switch 1

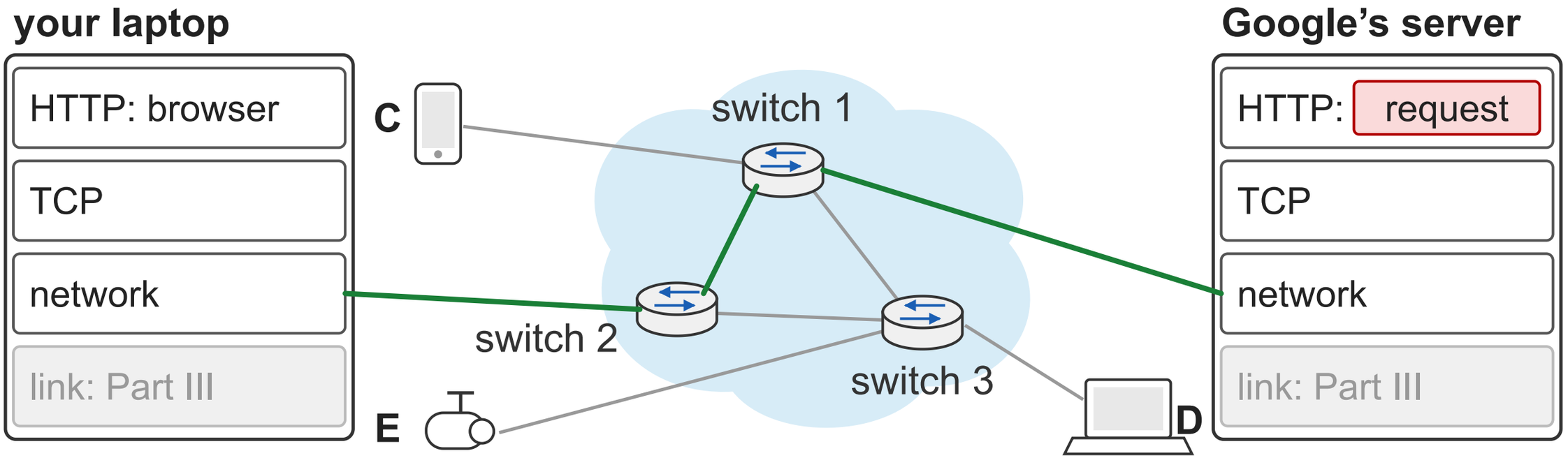
switch 1's forwarding table
to the server: the link to the server

All together so far: typing www.google.com



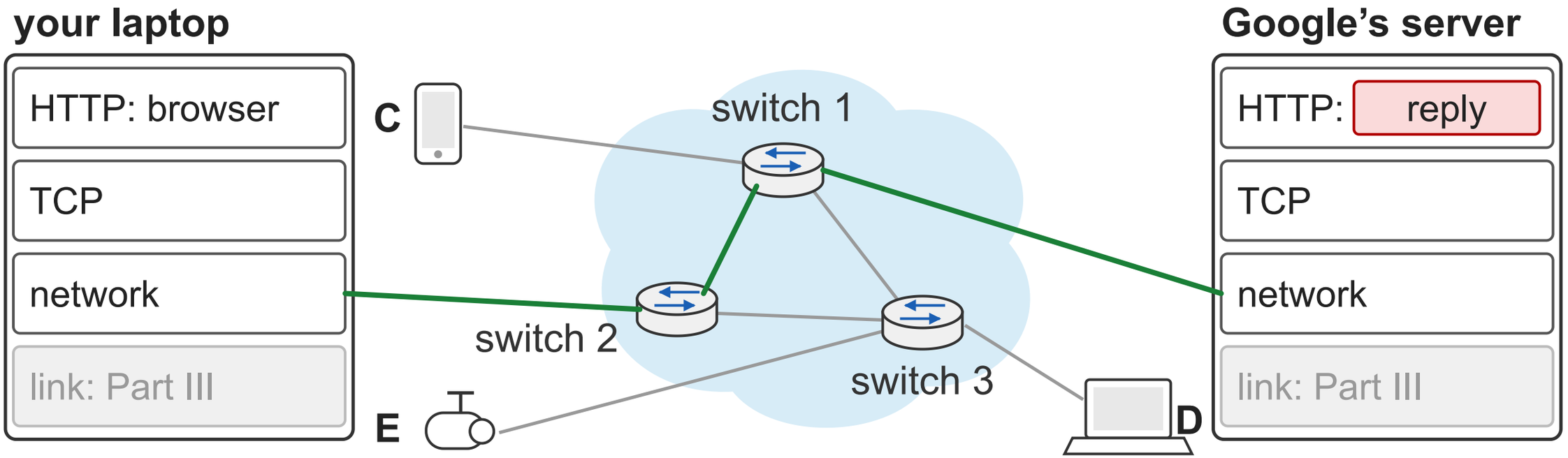
SYN + ACK comes back the same way, then the ACK: the connection is set up.

All together so far: typing www.google.com



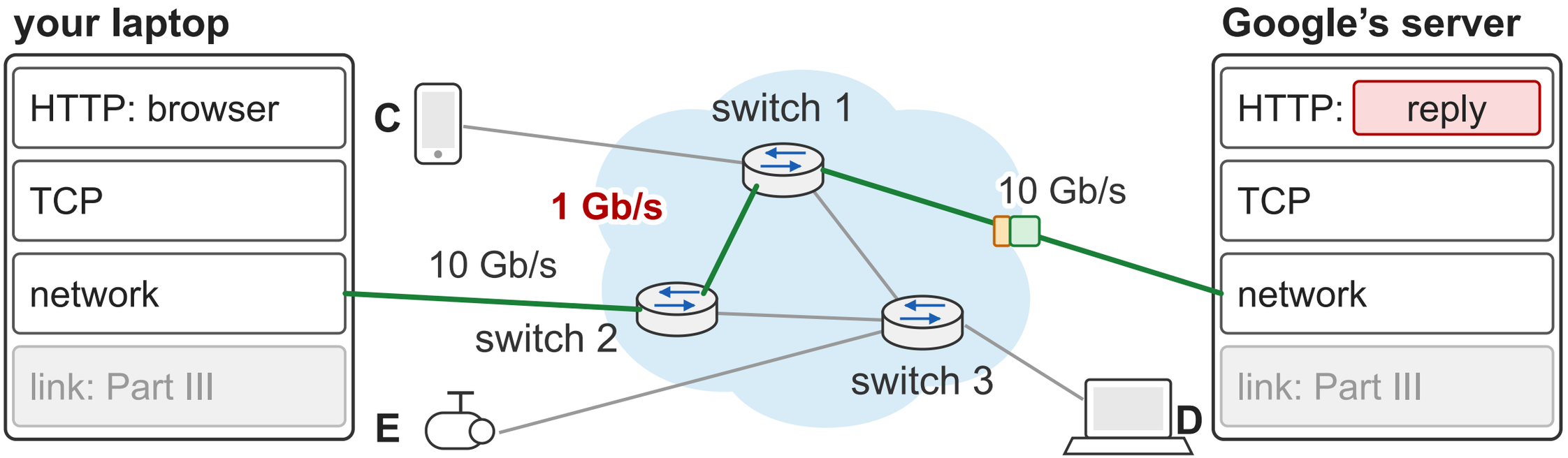
The request goes in a TCP segment, hop by hop, to the web server, which reads it and acknowledges it.

All together so far: typing www.google.com



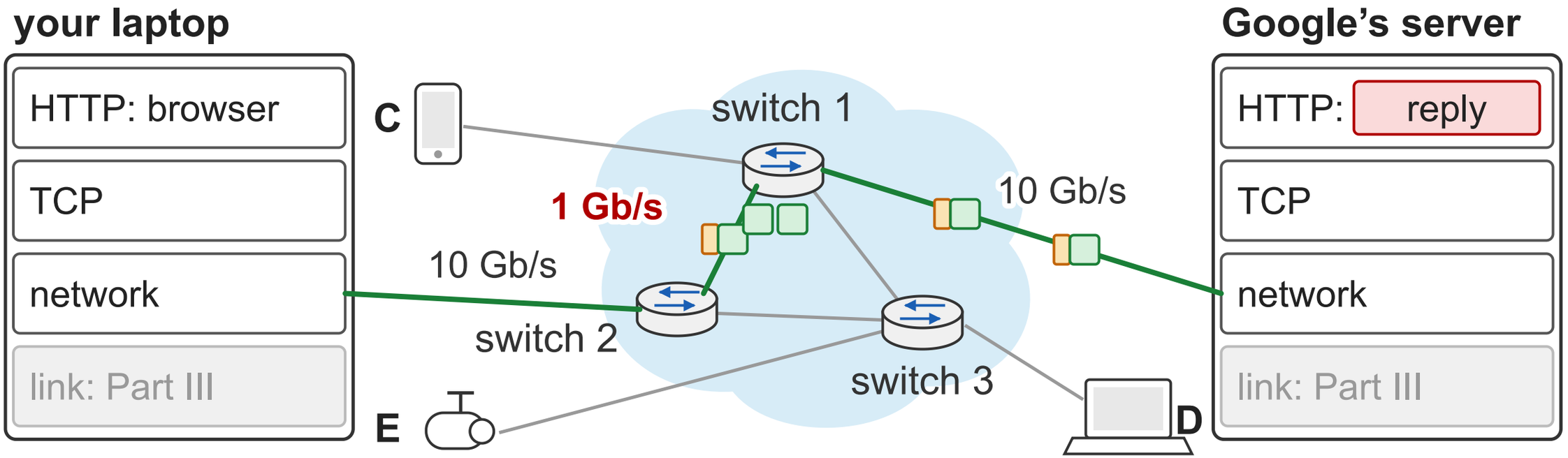
The web server writes the reply: 200 OK and then the page. This will be many segments. The transport layer, using congestion control and flow control together, will pace the segments.

All together so far: typing www.google.com



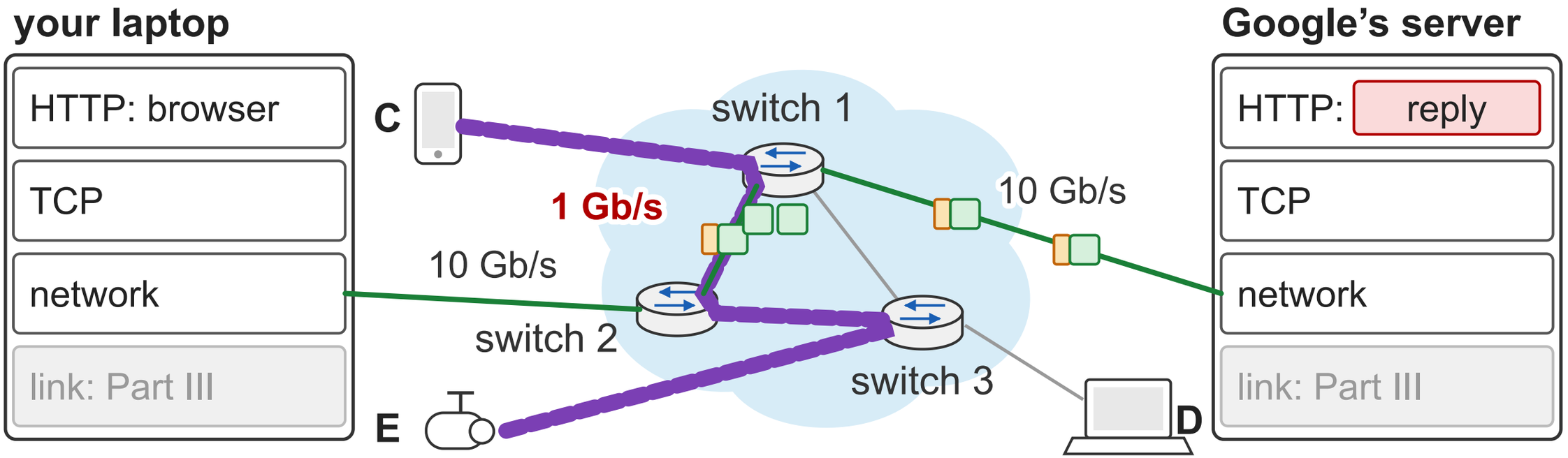
Let's suppose flow control is not the limiting factor. Congestion control: cwnd starts at 1 MSS, and slow start doubles it every RTT; the acks clock out new segments at the bottleneck's pace.

All together so far: typing www.google.com



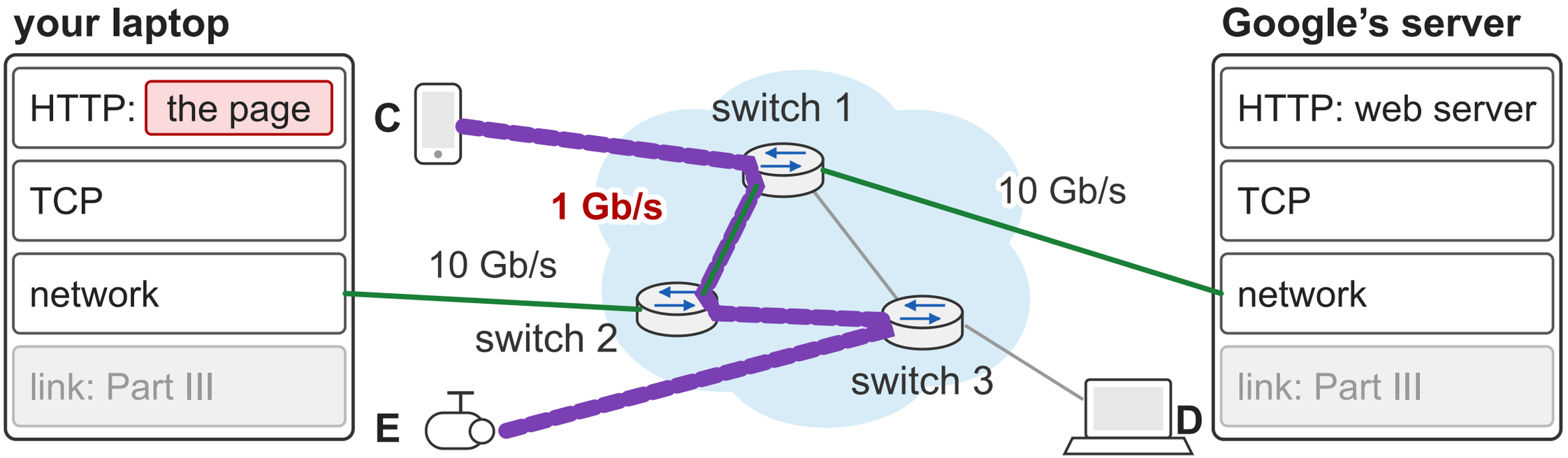
Past the BDP, segments wait at switch 1 for the 1 Gb/s link, and when the queue is full one is lost: TCP halves cwnd and goes on with AIMD (or CUBIC, or BBR).

All together so far: typing www.google.com



Another flow, C to E (dashed), shares the 1 Gb/s link. Each sender runs its own congestion control, and flows of the same kind share the link fairly (under certain assumptions).

All together so far: typing www.google.com



The laptop's TCP puts the segments in order and hands them up: the browser renders the page.