CS 43: Computer Networks Flow and Congestion Control

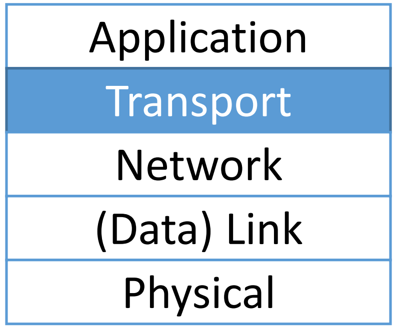
Kevin Webb

Swarthmore College

March 22, 2022

Recap

- Wrapping up the transport layer
- Rounding out TCP
  - Sliding window: how many bytes to pipeline
  - How big do we make that window?
    - Too small: waste capacity
    - Too large: congestion
    - Other concerns: fairness



Rate Control

Flow Control

- Don’t send so fast that we overload the receiver.

- Rate directly negotiated between one pair of hosts (the sender and receiver).

Congestion Control

- Don’t send so fast that we overload the network.

- Rate inferred by sender in response to “congestion events.”

Shared high-level goal: don’t waste capacity by sending something that is likely to be dropped.

Flow Control

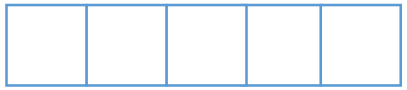
Problem: Sender can send at a high rate. Network can deliver at a high rate. The receiver is drowning in data.

- Example scenario:

Flow Control

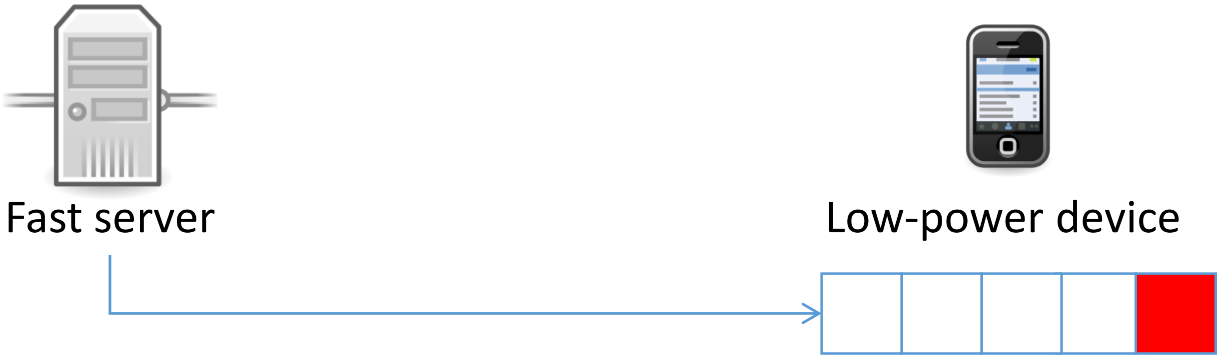
Fast server

Low-power device



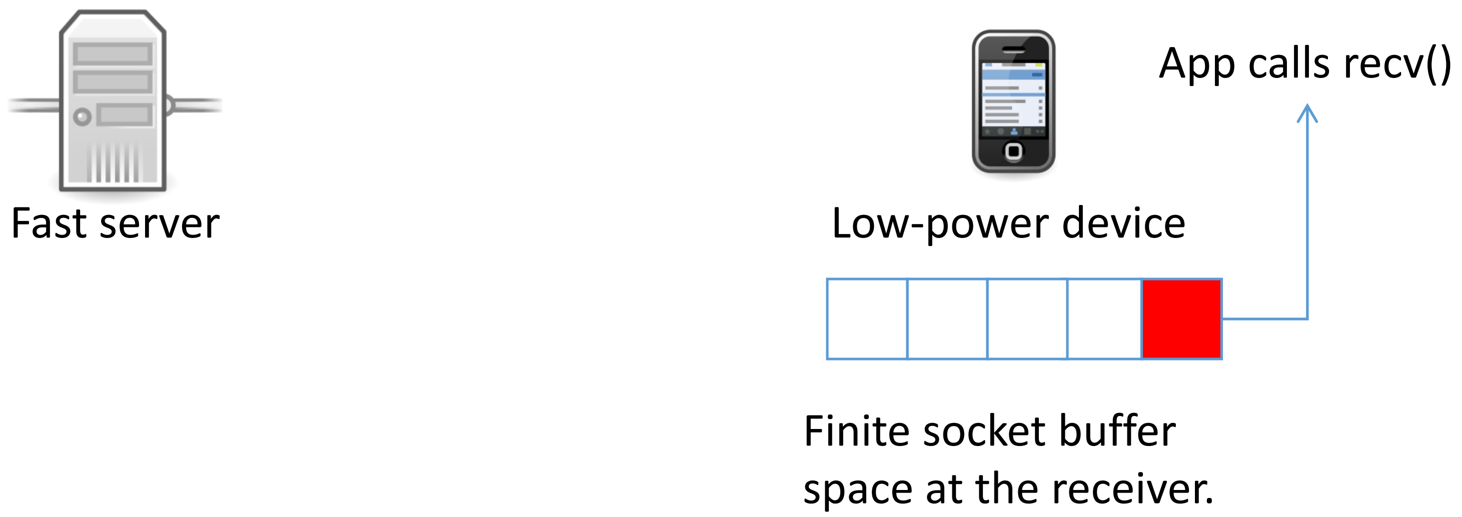
Finite socket buffer
space at the receiver.

Flow Control

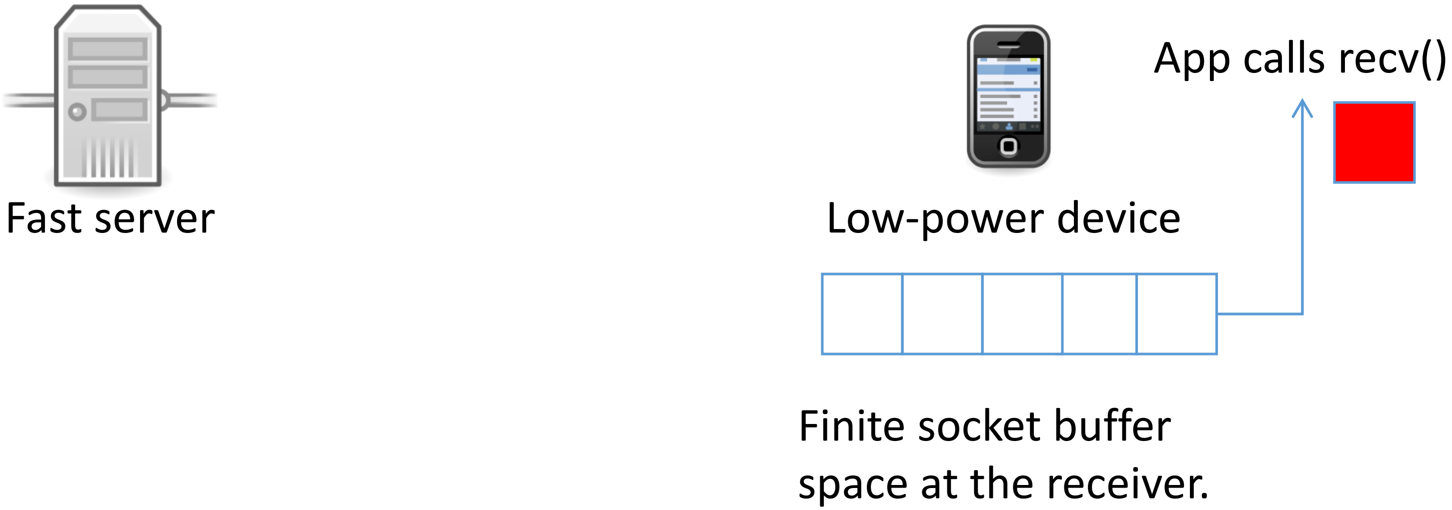


Finite socket buffer
space at the receiver.

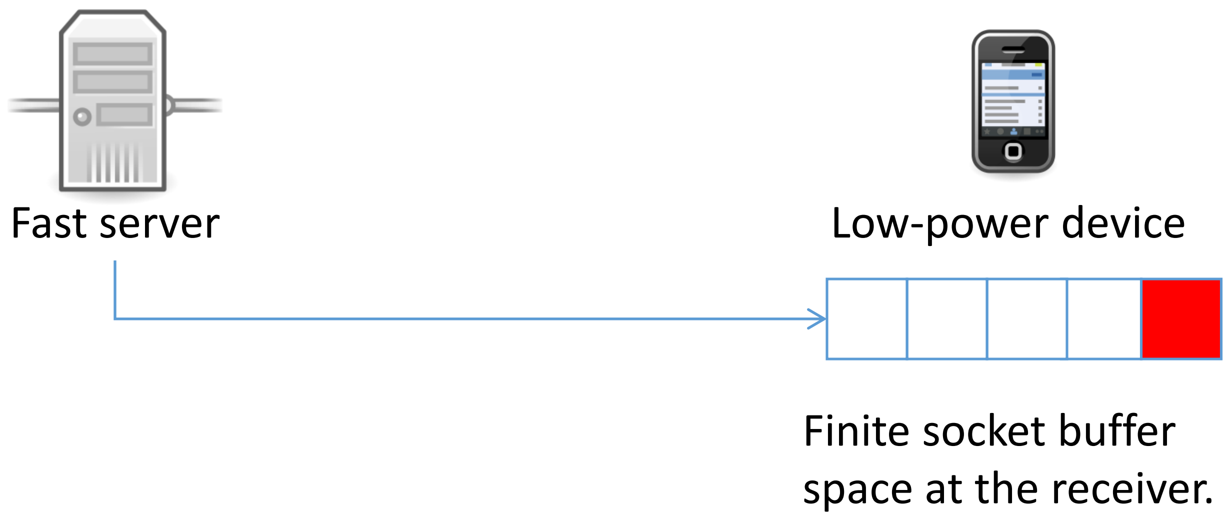
Flow Control



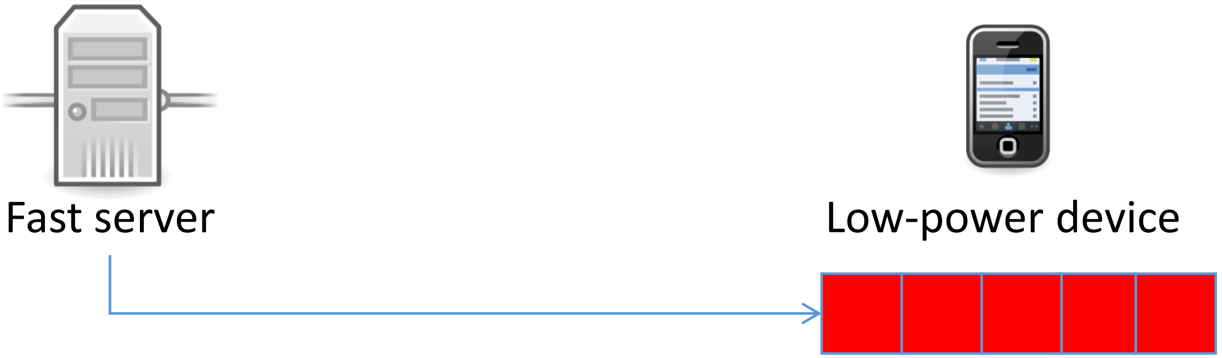
Flow Control



Flow Control

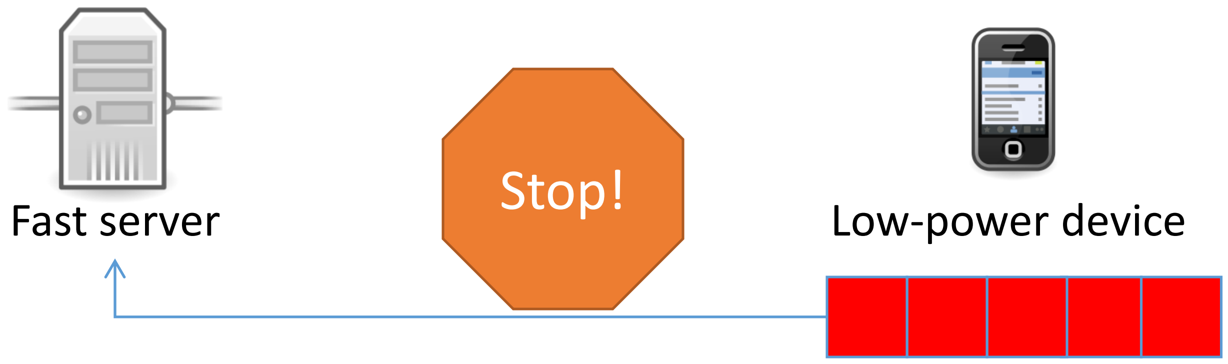


Flow Control



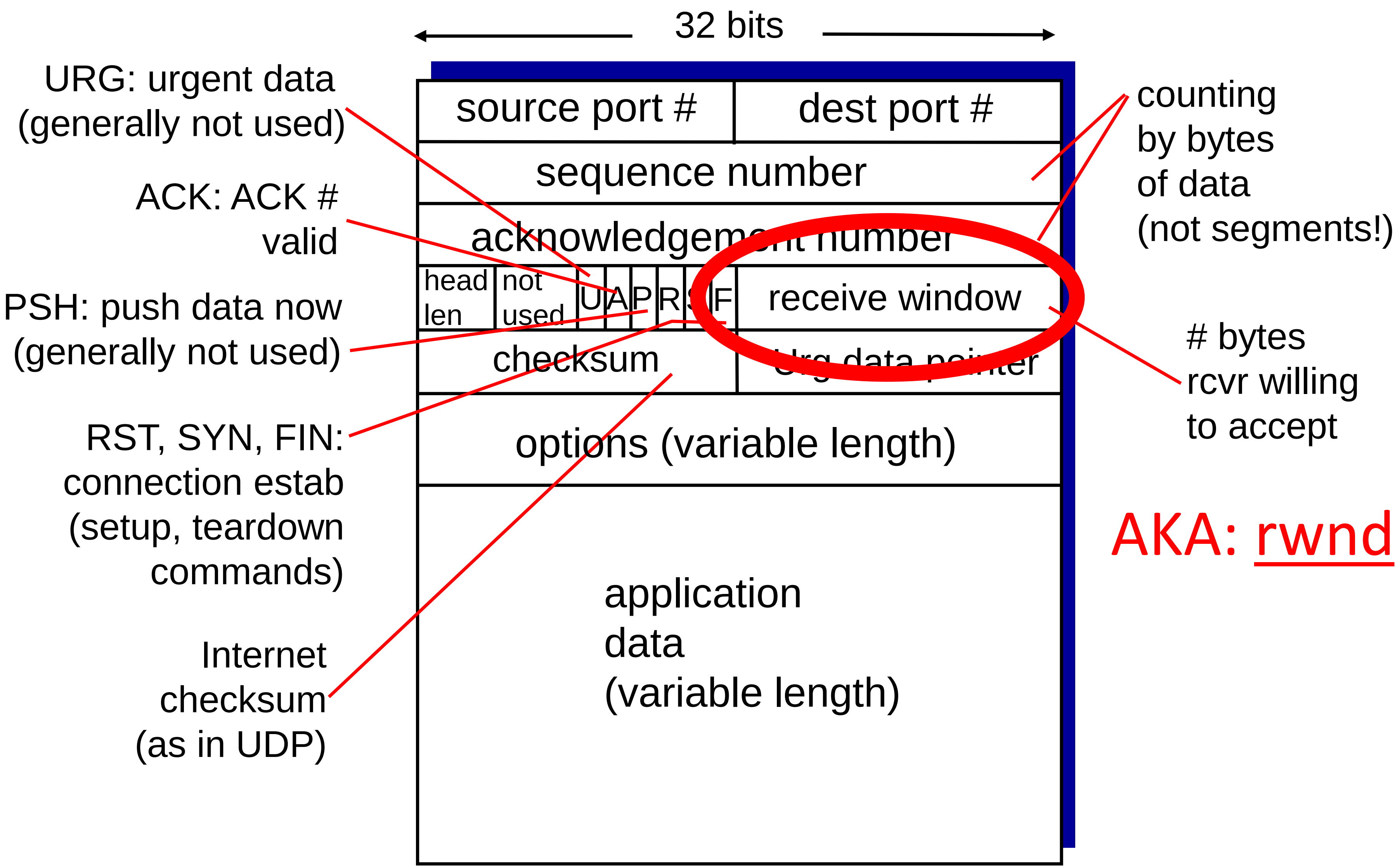
Finite socket buffer space at the receiver.

Flow Control



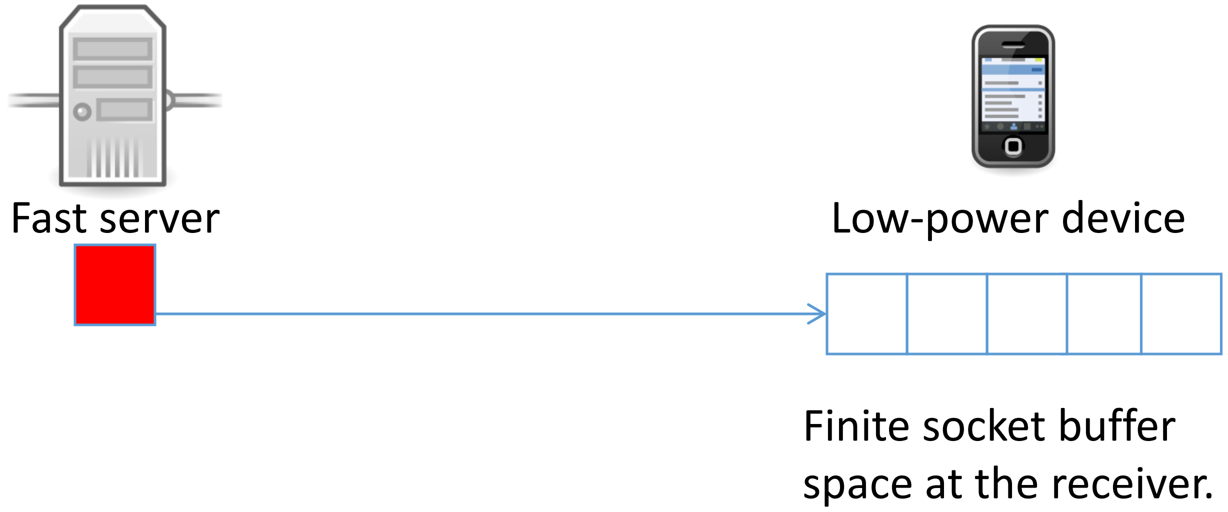
Finite socket buffer space at the receiver.

TCP Segments



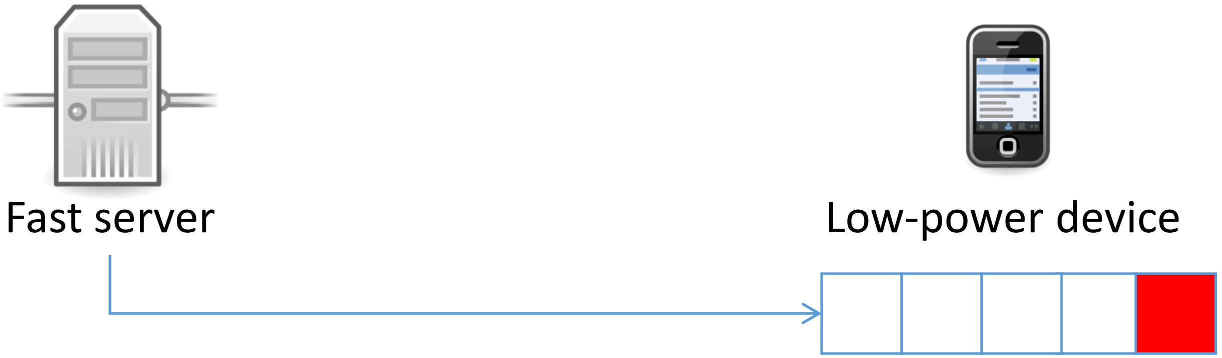
Flow Control

- Sender never sends more than rwnd.



Flow Control

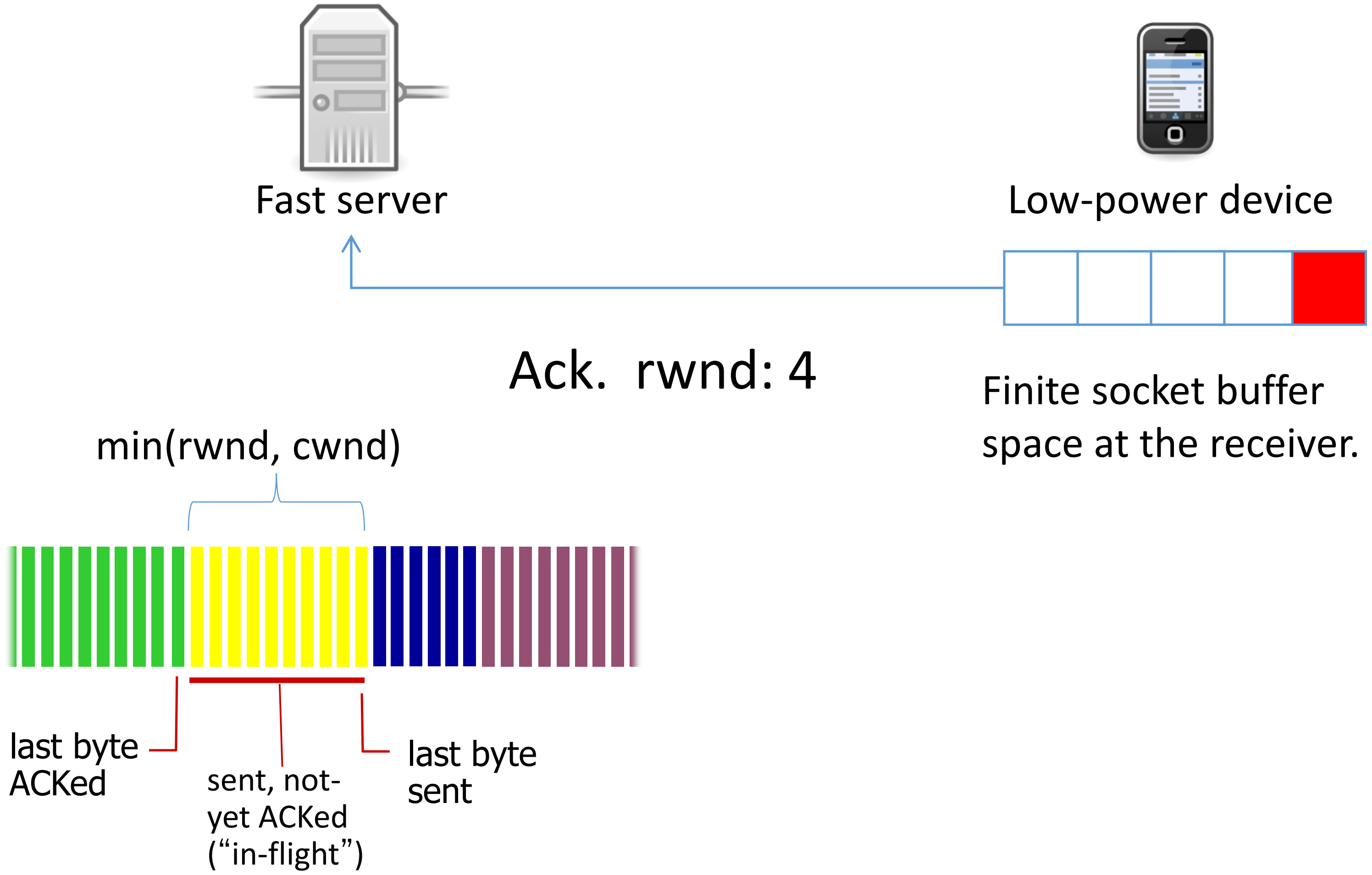
- Sender never sends more than rwnd.



Finite socket buffer
space at the receiver.

Flow Control

- Sender never sends more than rwnd.

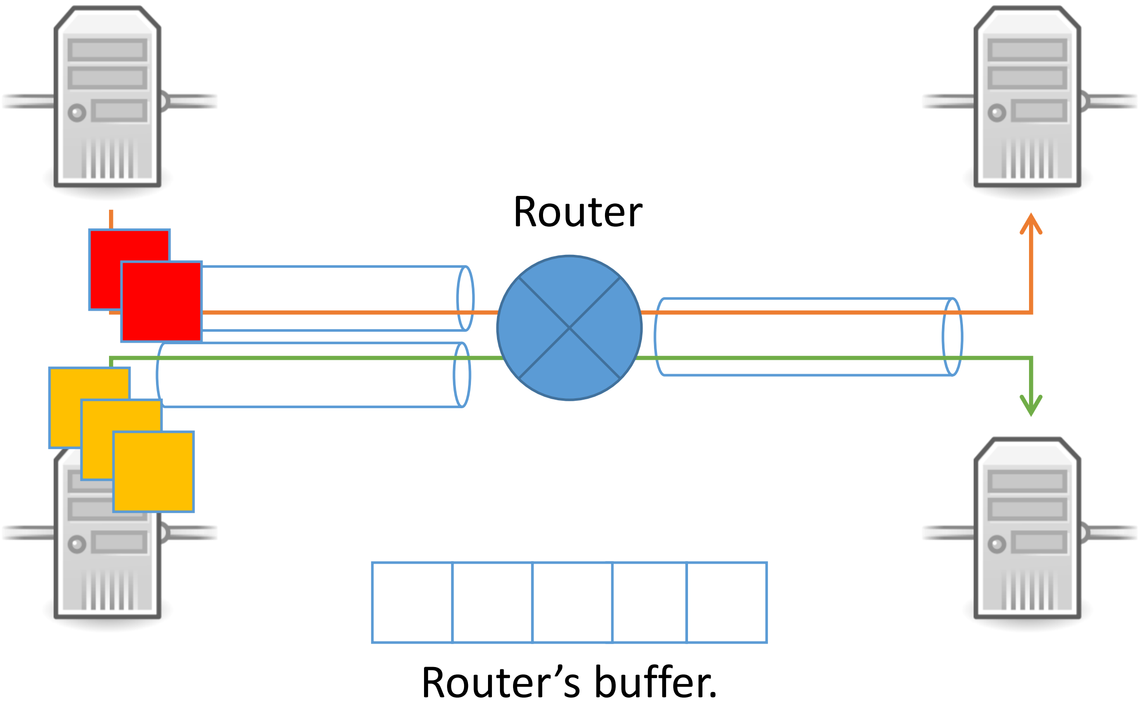


Congestion

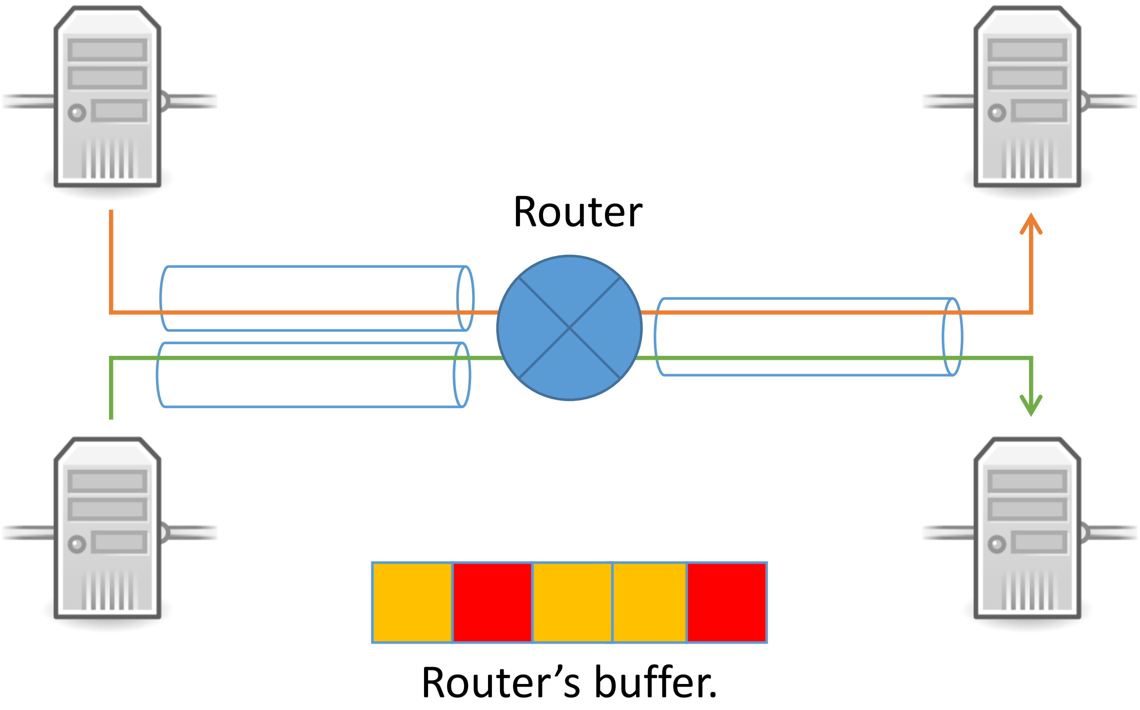
- Flow control is (relatively) easy. The receiver knows how much space it has.

- What about the network devices?

Congestion

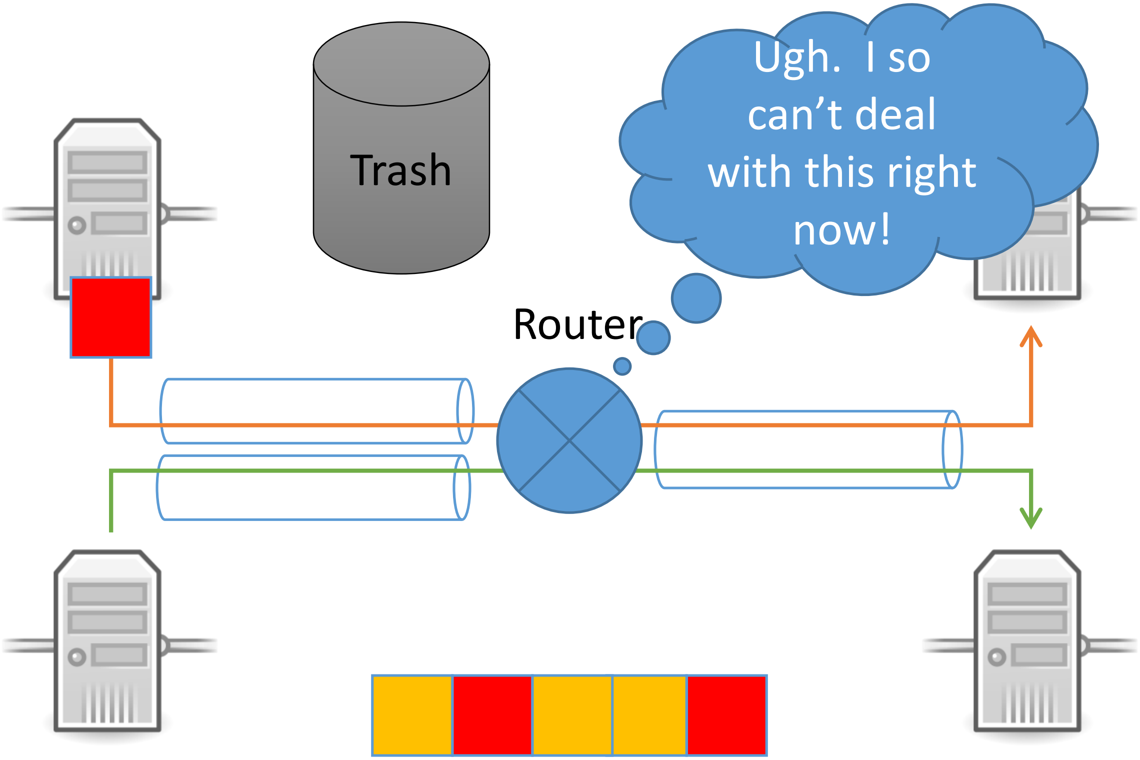


Congestion



Incoming rate is faster than
outgoing link can support.

Congestion



Router's buffer.

Incoming rate is faster than outgoing link can support.

What’s the worst that can happen?

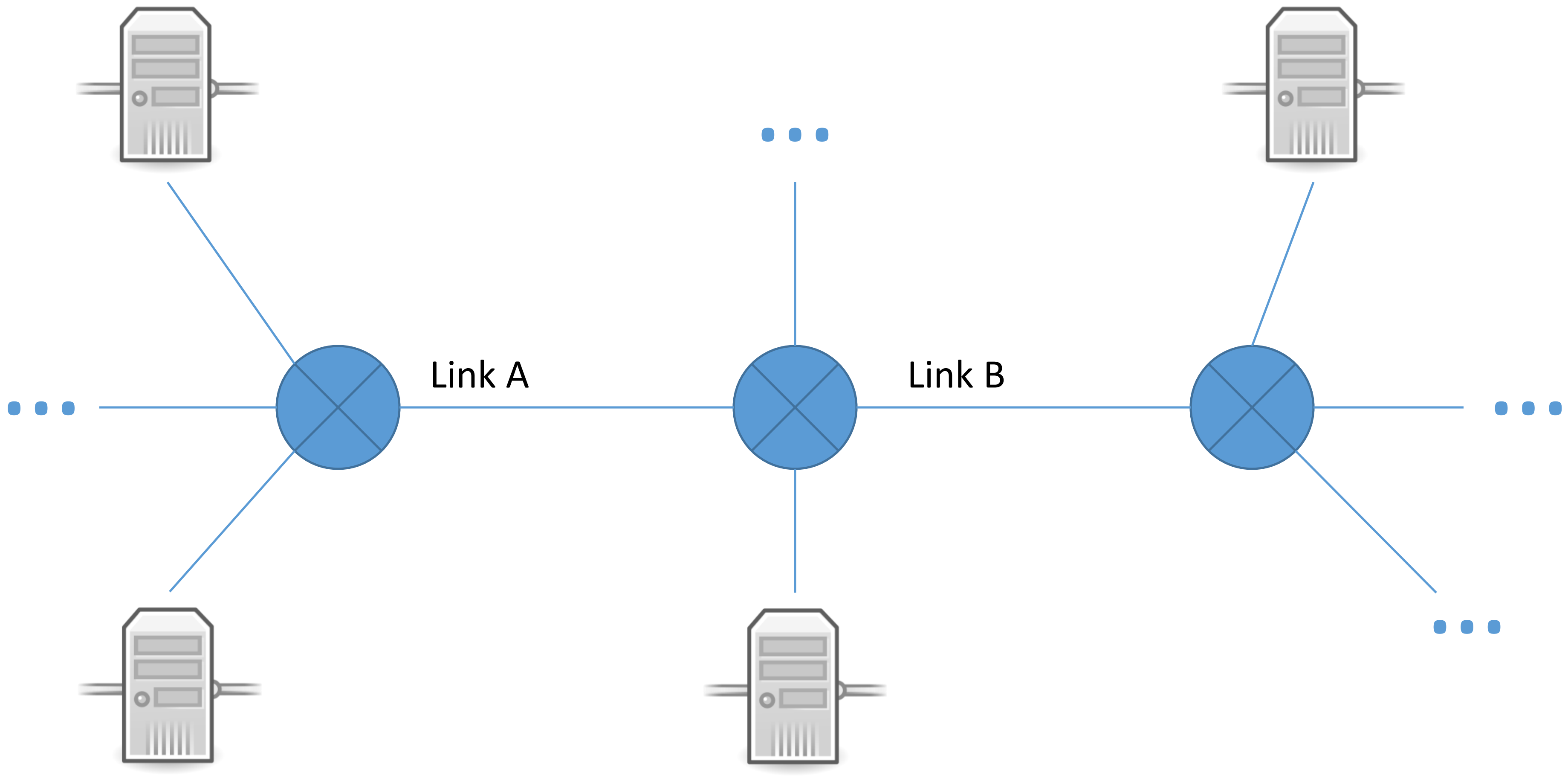
A. This is no problem. Senders just keep transmitting, and it’ll all work out.

B. There will be retransmissions, but the network will still perform without much trouble.

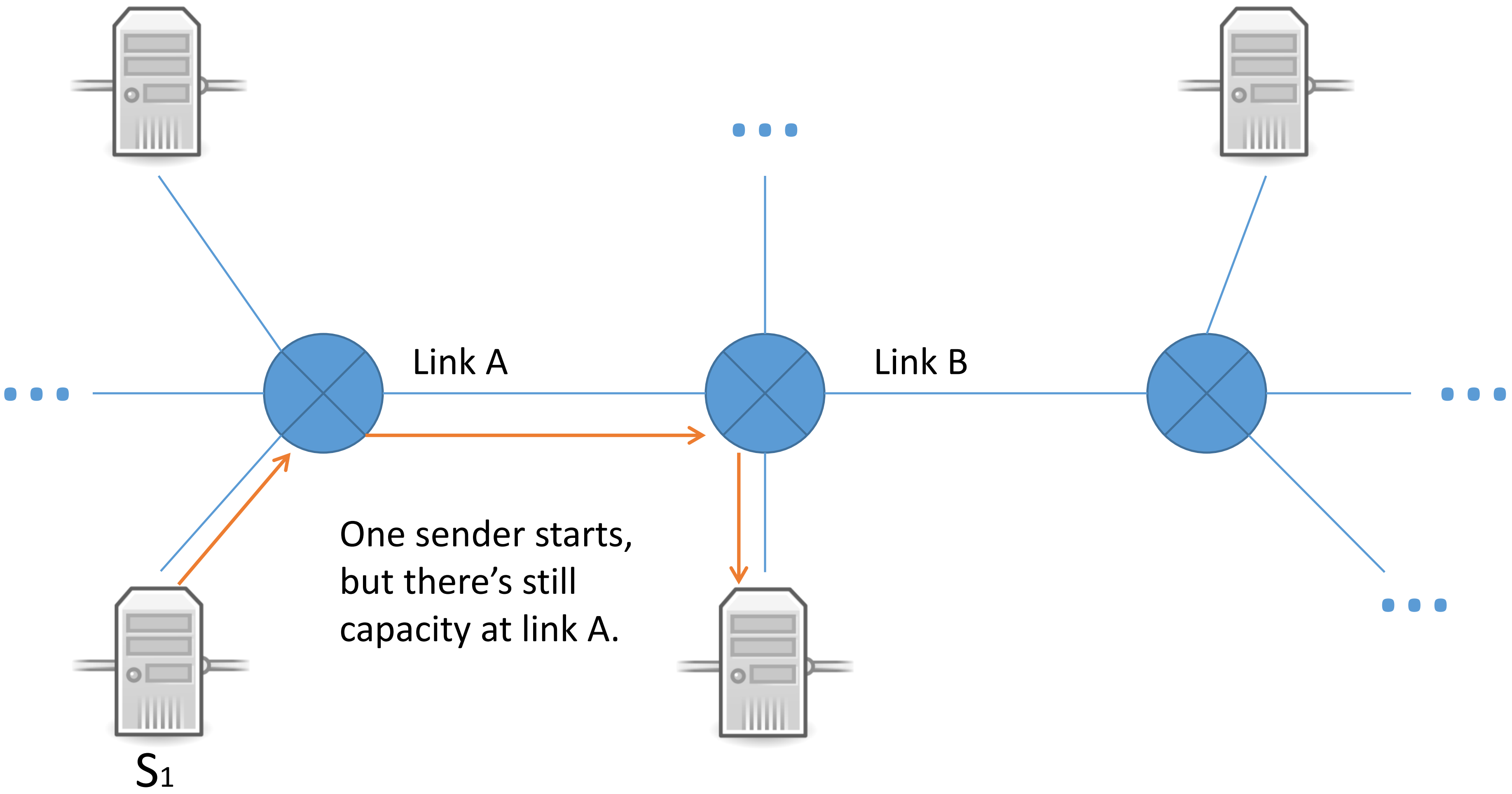
C. Retransmissions will become very frequent, causing a serious loss of efficiency.

D. The network will become completely unusable.

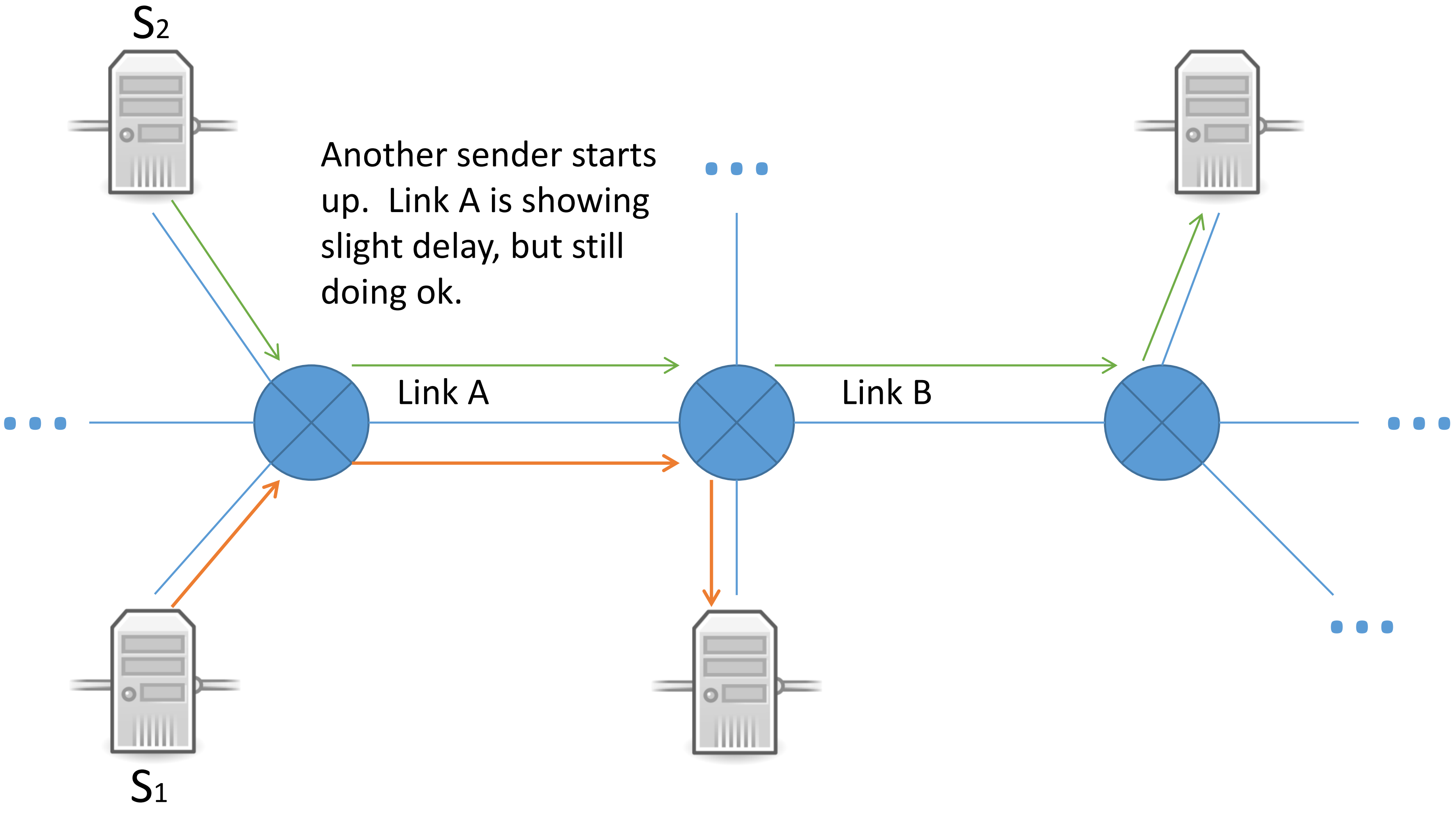
Congestion Collapse



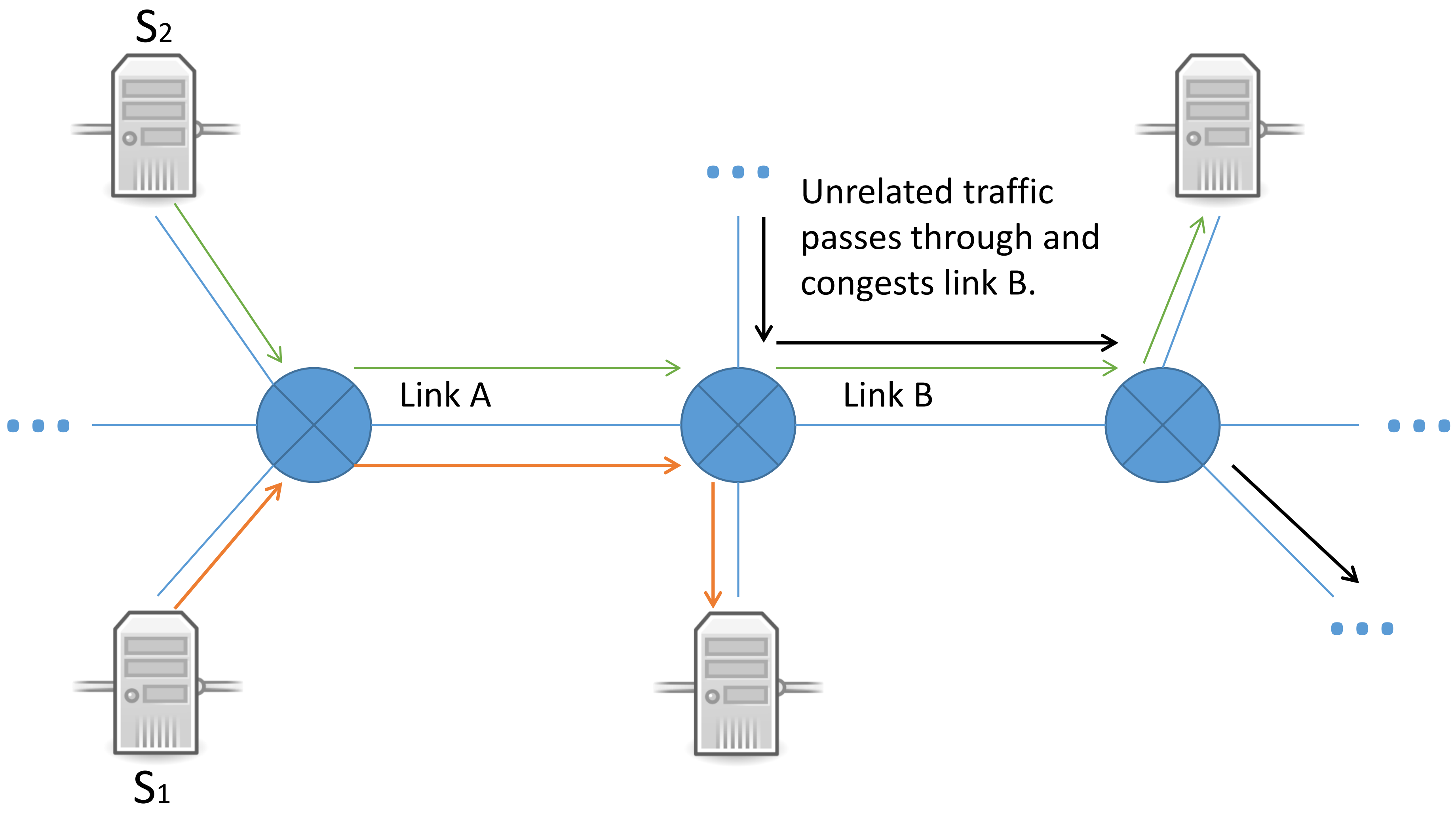
Congestion Collapse



Congestion Collapse

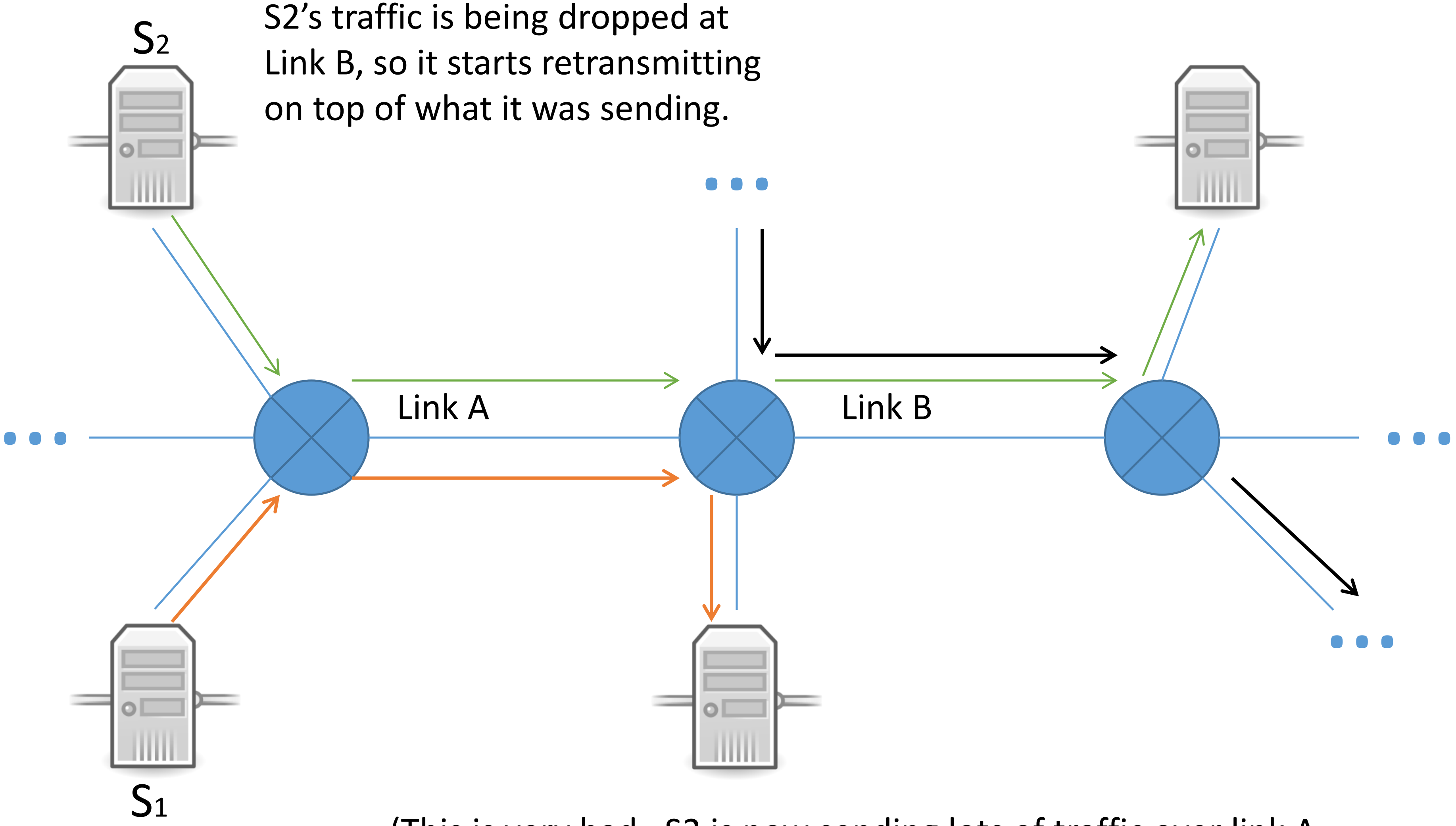


Congestion Collapse



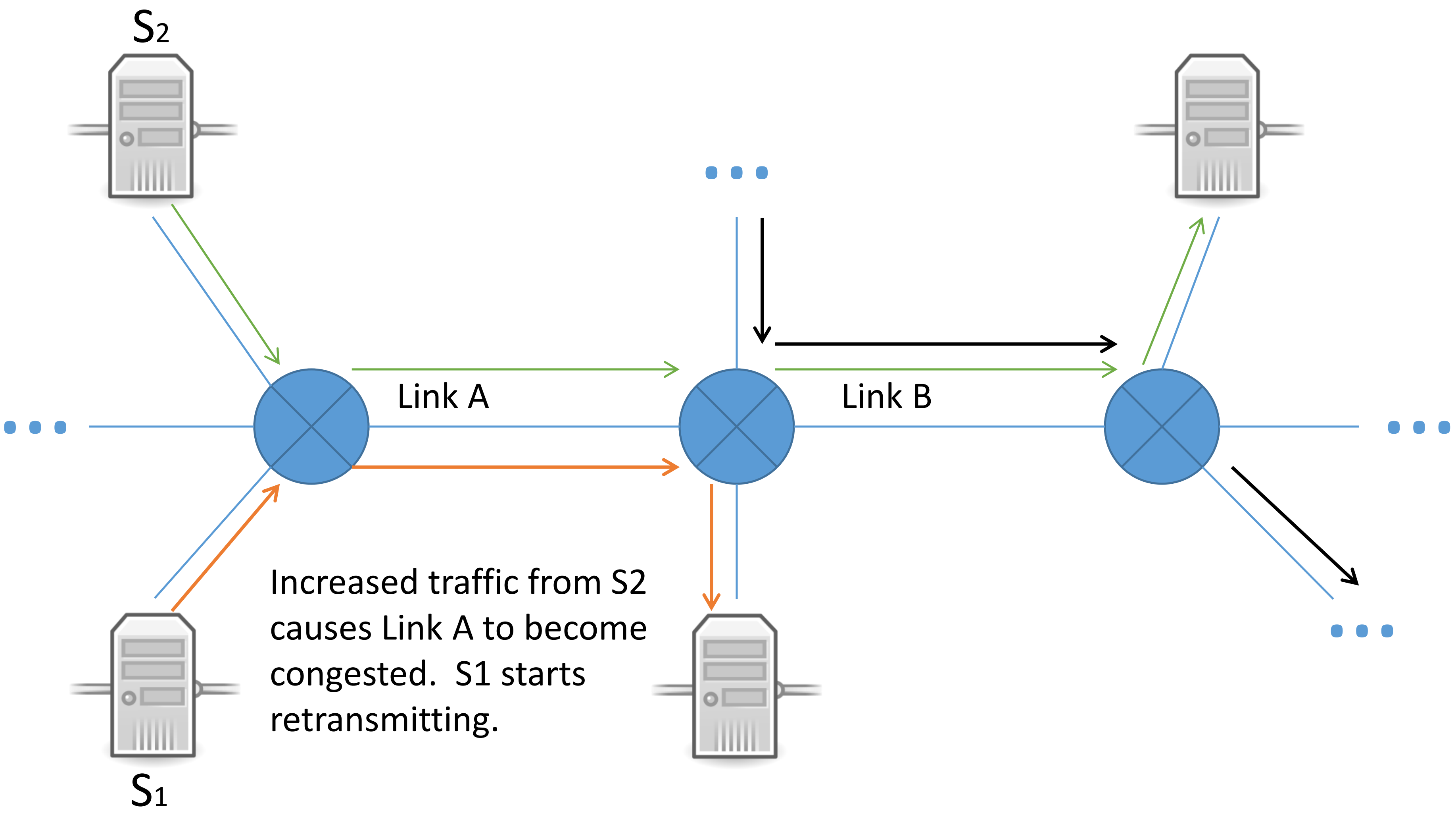
Congestion Collapse

S2's traffic is being dropped at Link B, so it starts retransmitting on top of what it was sending.



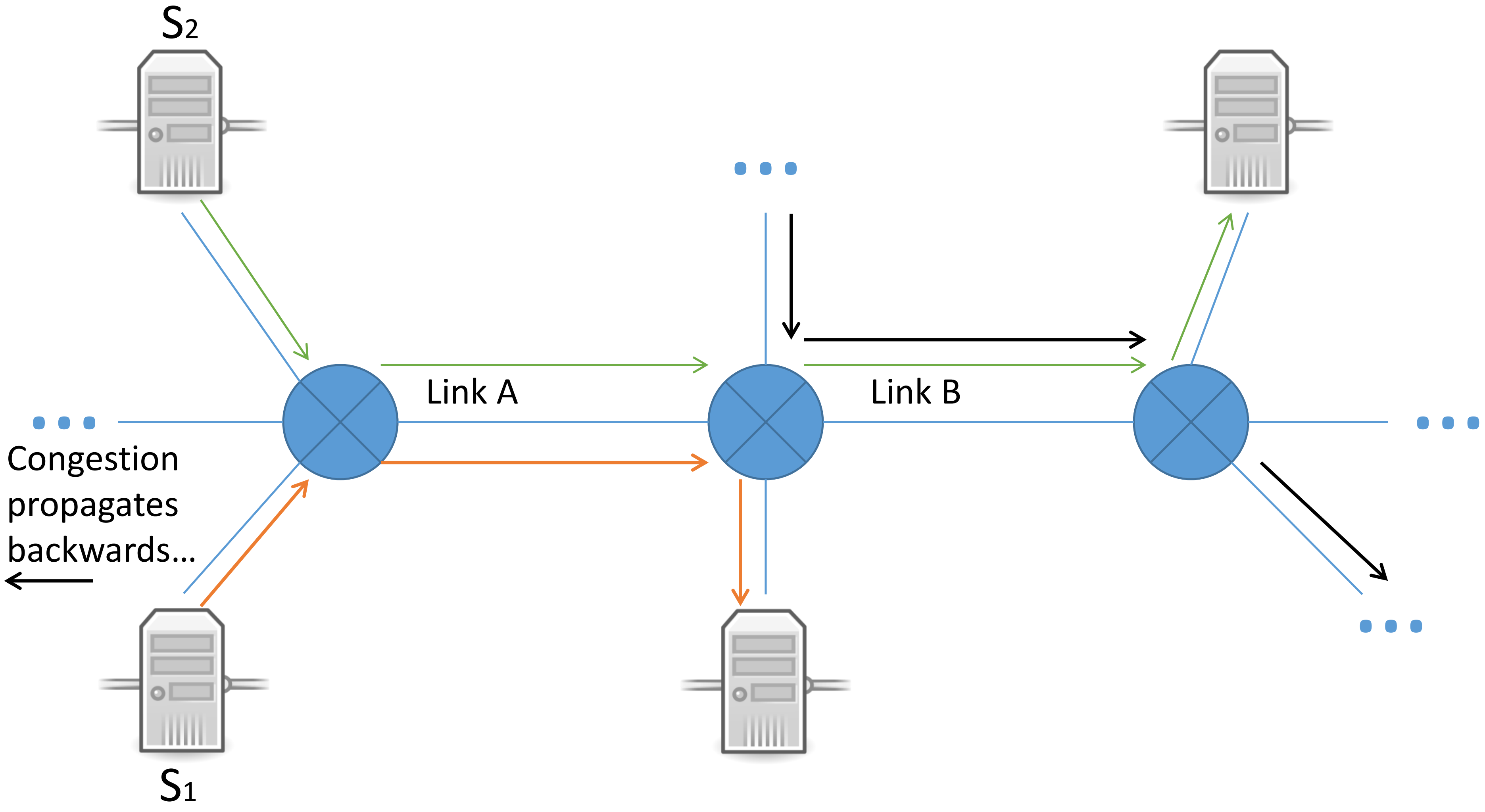
(This is very bad. S2 is now sending lots of traffic over link A that has no hope of crossing link B.)

Congestion Collapse



Increased traffic from S2 causes Link A to become congested. S1 starts retransmitting.

Congestion Collapse



Without Congestion Control

- Congestion…
  - Increases delivery latency
  - Increases loss rate
  - Increases retransmissions, many unnecessary
  - Wastes capacity on traffic that is never delivered
  - Increases congestion, cycle continues…

Congestion Collapse

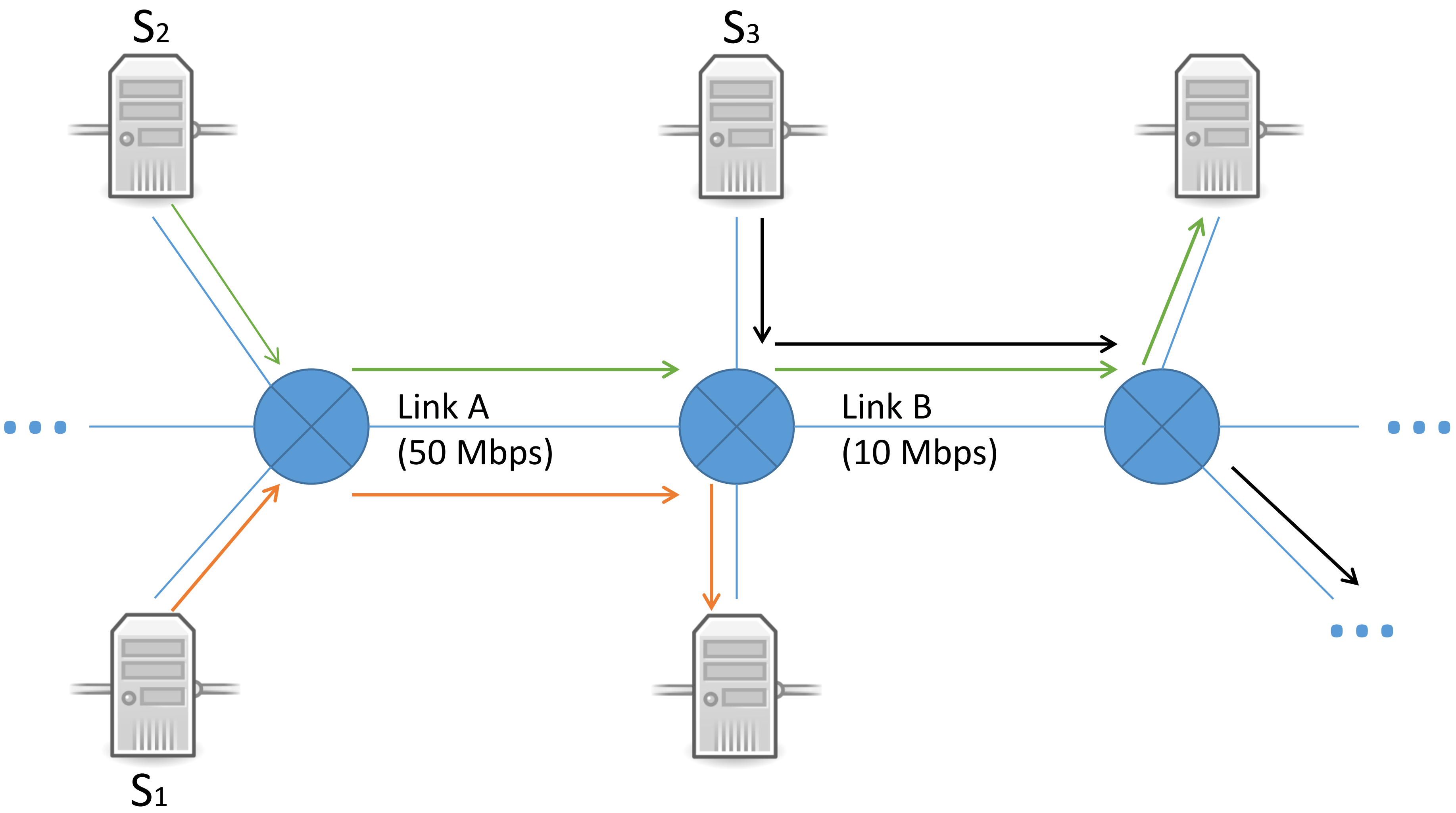
- This happened to the Internet (then NSFnet) in 1986.
  - Rate dropped from a blazing 32 kbps to 40 bps
  - This happened on and off for *two years*
  - In 1988, Van Jacobson published
    "Congestion Avoidance and Control"
  - The fix: senders voluntarily limit sending rate

Intuition so far…

- Senders voluntarily limit how quickly they send to prevent over filling the network.

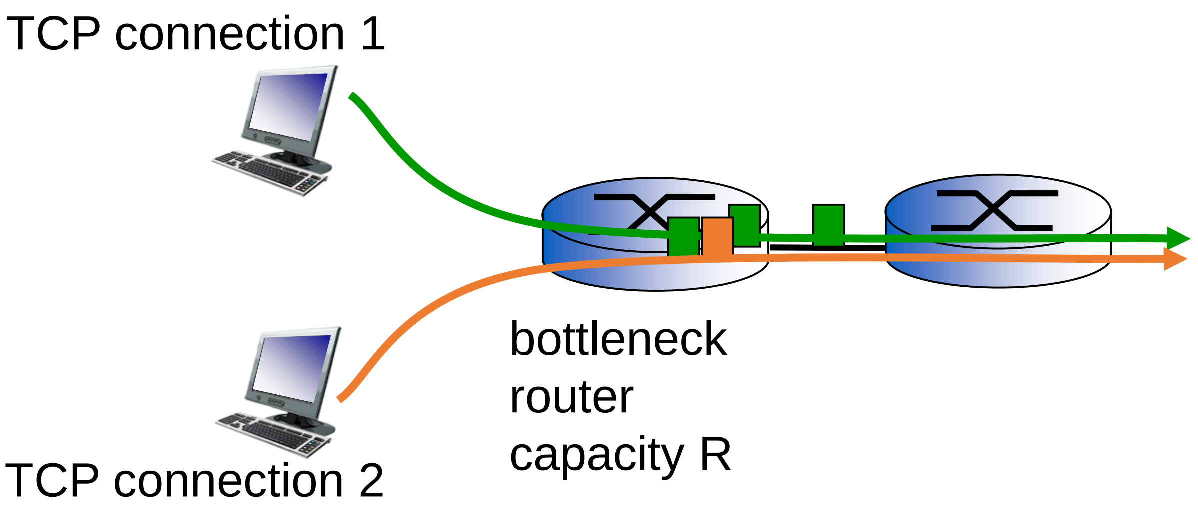
- General goal: sender should send at a rate that roughly corresponds to an equal "fair share" of its most bottlenecked link

"Equal fair share"



TCP Fairness

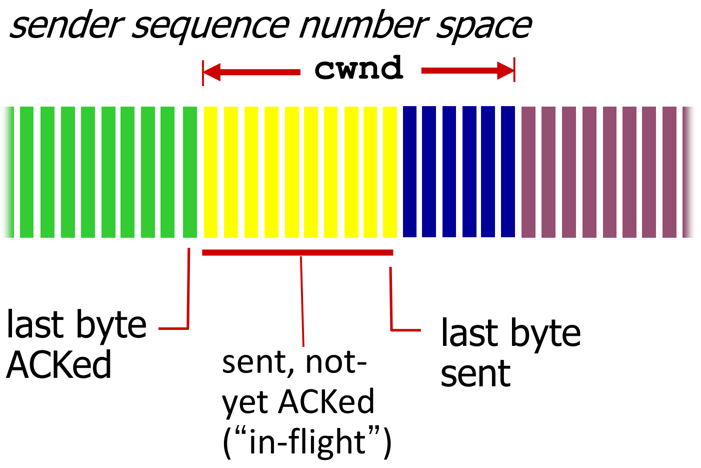
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Challenges

- How can a sender determine its "fair share" of a link?
- What if "fair shares" are changing as new conversations start or stop?
- Initially: a sender knows nothing about the state of the network
  - Send too little, capacity goes unused
  - Send too much, cause congestion for everyone else

TCP Congestion Control: mechanism details



- sender limits transmission:

$$\boxed{\texttt{LastByteSent-LastByteAcked} \leq \texttt{cwnd}}$$

- **`cwnd`** is dynamic, based on perceived network congestion

TCP sending rate:

- send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\boxed{\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}}$$

How should we set cwnd?

A. We should keep raising it until a “congestion event”, then back off slightly until we notice no more events.

B. We should raise it quickly until a “congestion event”, then go back to 0 and start raising it again.

C. We should raise it until a “congestion event”, then go back to a median value and start raising it again.

D. We should send as fast as possible at all times.

E. Some other strategy (what?)

What is a “congestion event”?

A. A segment loss

B. Receiving duplicate acknowledgement(s)

C. A retransmission timeout firing

D. Some subset of the above

E. All of the above

TCP Congestion Control Phases

- "Slow Start" Phase
  - Start conservatively, increase rate quickly
  - Intuition: sender unsure of network’s congestion, probe for capacity

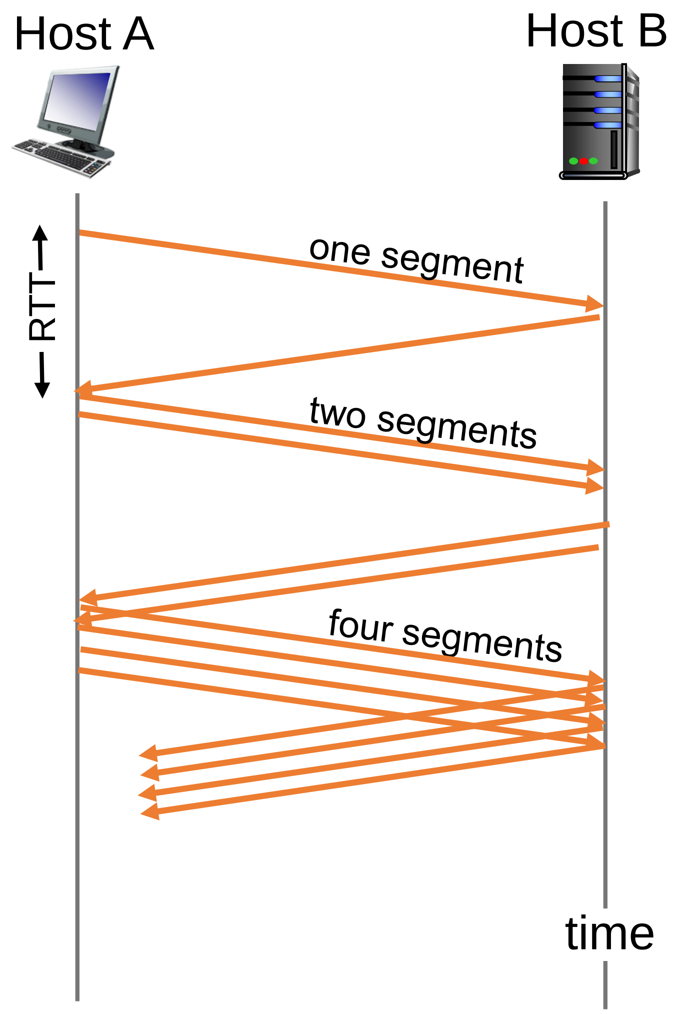
- "Congestion Avoidance" Phase
  - Increase rate slowly
  - Back off when congestion occurs
    - How much depends on TCP version
  - Intuition: once we're close to "fair share", try to hover around that point, but still adapt to changes (e.g., new TCP competitors starting / stopping)

TCP Slow Start

Commonly 1448 bytes across the Internet today.

- When connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 segment (1 MSS)
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- ***Summary:*** initial rate is slow but ramps up exponentially

- When do we stop?

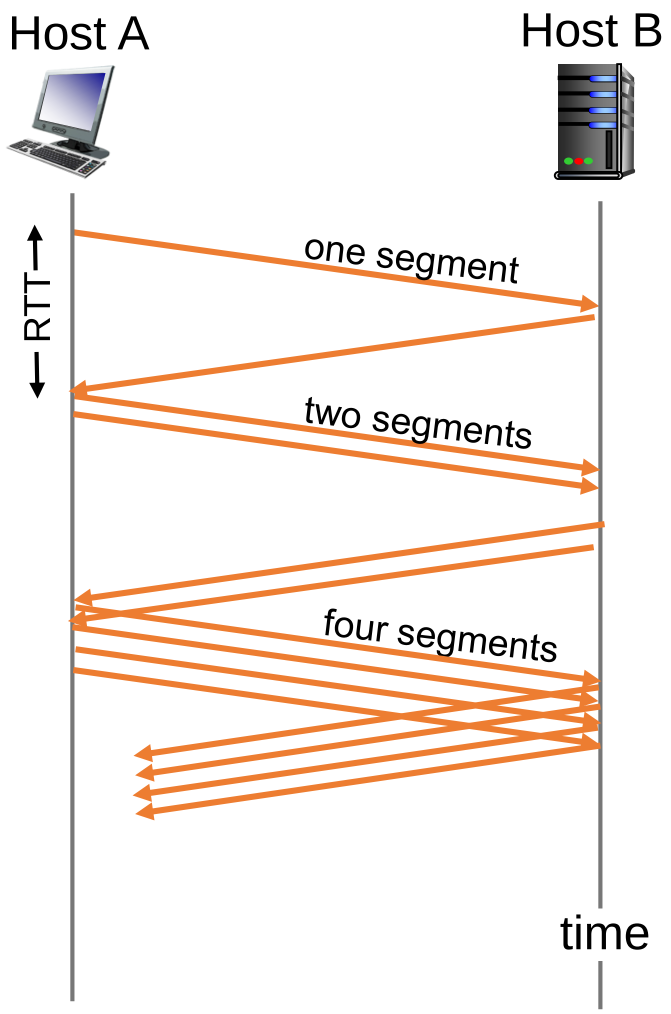


TCP Slow Start

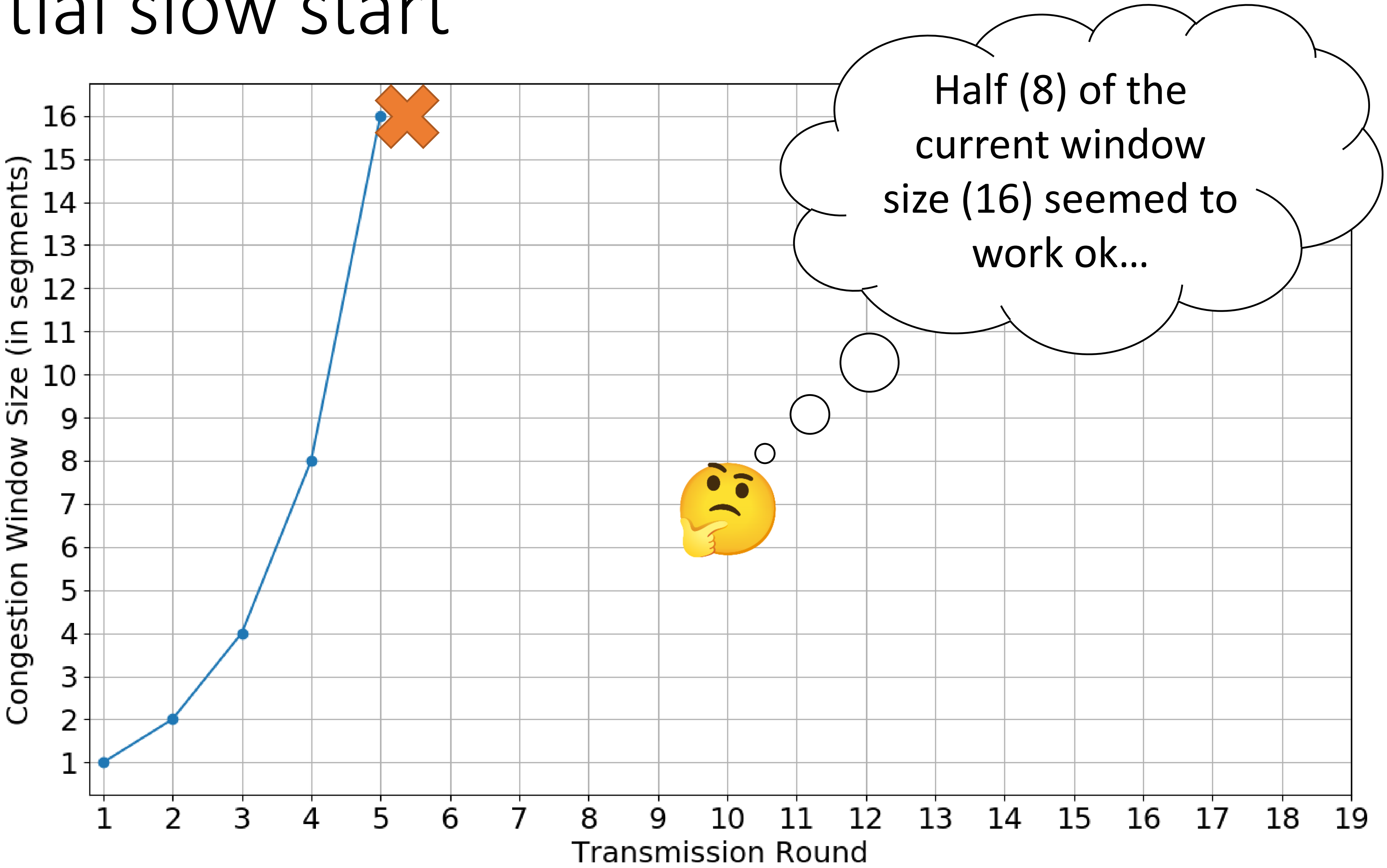
- When do we stop?

- Initially
  - On a congestion event

- Later
  - On a congestion event
  - When we cross a previously-determined threshold



Initial slow start



TCP Congestion Avoidance

- `ssthresh`: Threshold where slow start ends
- During initial slow start, threshold is unlimited
  - On congestion event, set it to **half** of current window size
- In congestion avoidance, instead of doubling, increase `cwnd` by one MSS every RTT.
  - Increase `cwnd` by MSS/cwnd bytes for each ACK
  - Back off on congestion event

We can determine that a packet was lost two different ways: via 3 duplicate ACKS, or via a timeout. We should…

A. Treat these events differently.

B. Treat these events the same.

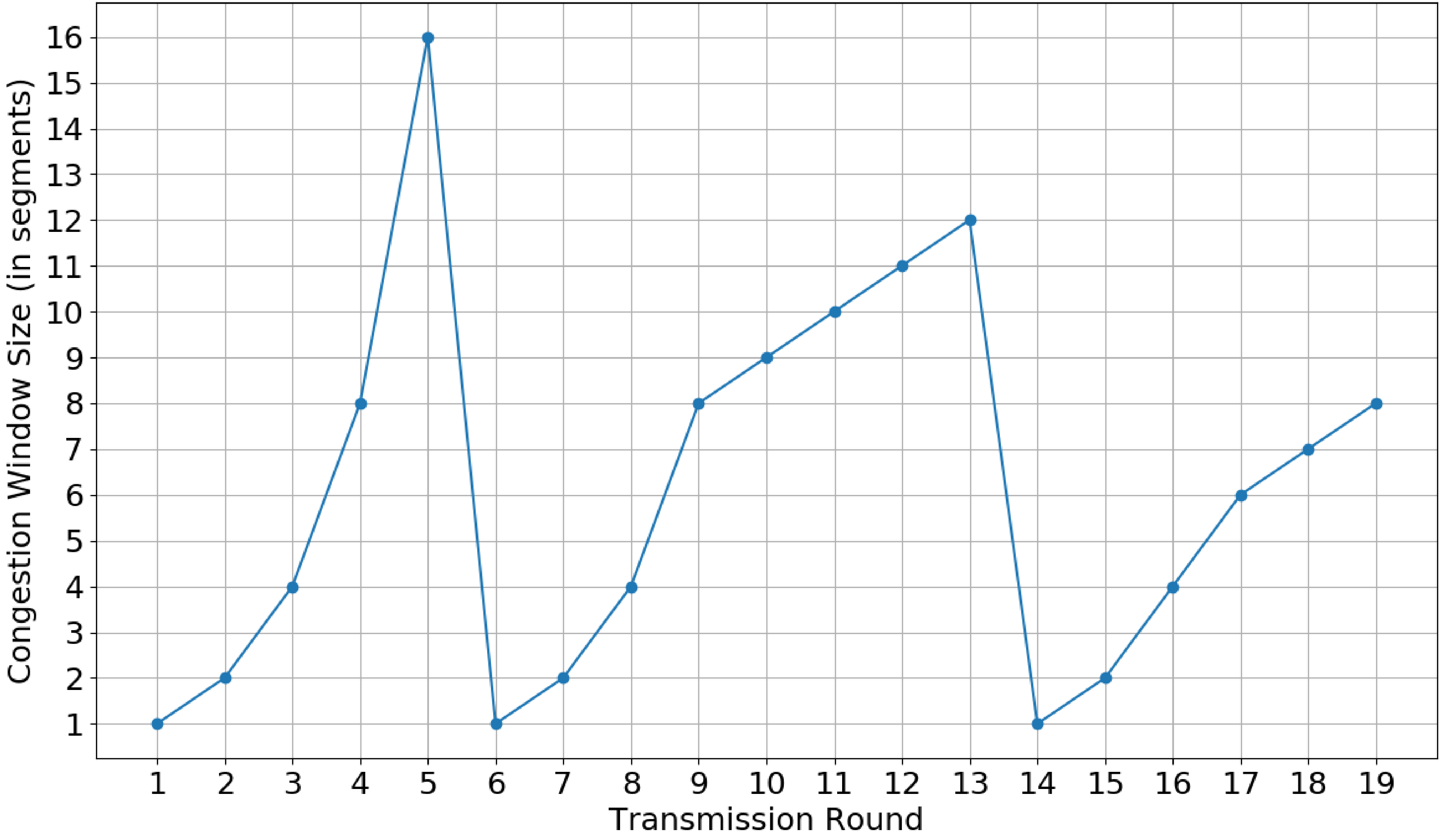
(For discussion: Is one of these events worse than the other, or do they represent equally bad scenarios? If they're not equal, which is worse?)

Detecting, Reacting to Loss (TCP "Tahoe")

- Loss indicated by timeout:
  - **`cwnd`** set to 1 segment (1 MSS);
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- Loss indicated by 3 duplicate ACKs:
  - **`cwnd`** set to 1 segment (1 MSS);
  - window then grows exponentially (as in slow start) to threshold, then grows linearly

(Tahoe handles both of these the same way).

TCP Tahoe



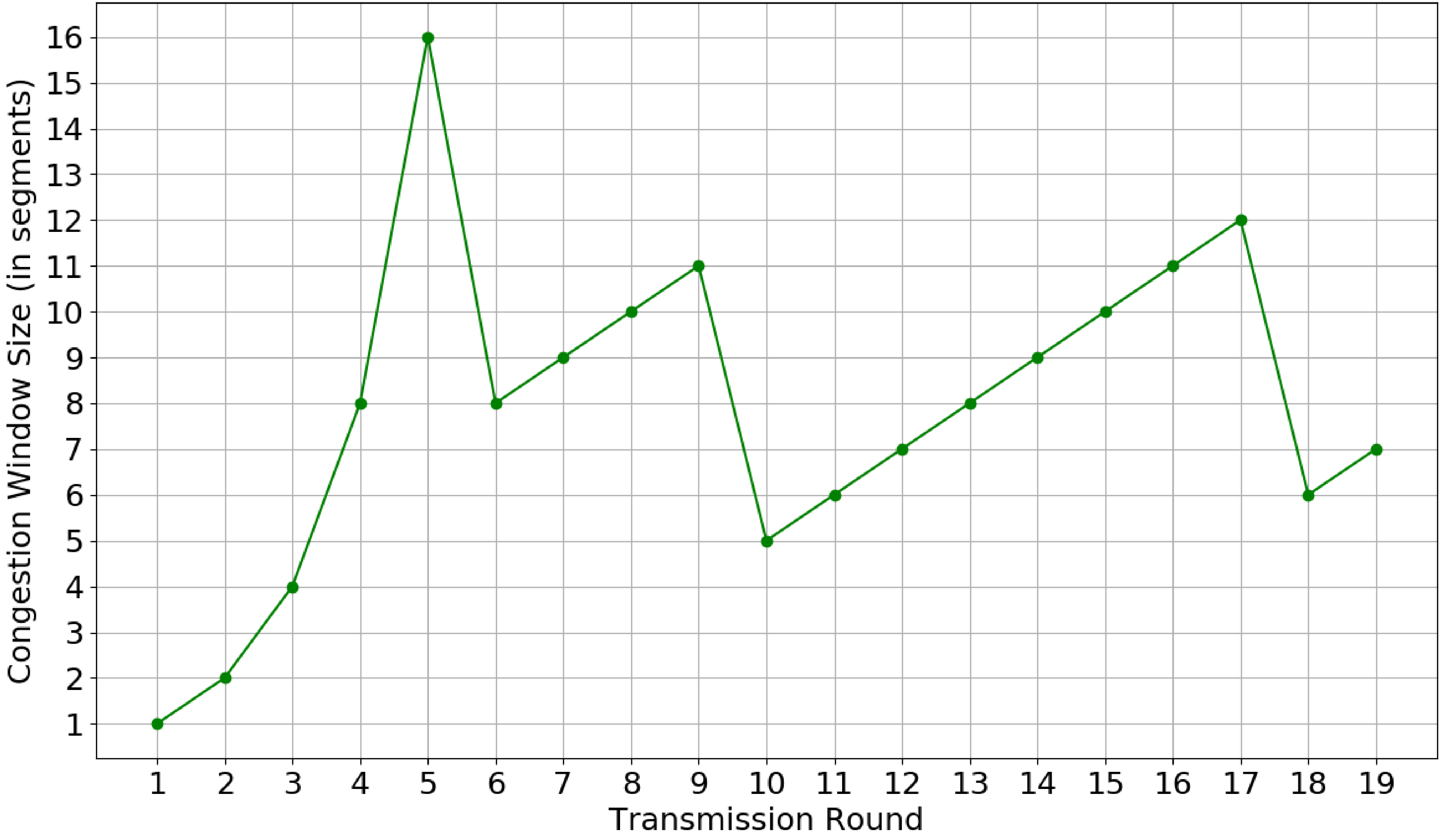
Congestion Window Size (in segments)
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Transmission Round

Detecting, Reacting to Loss (TCP "Reno")

- Loss indicated by timeout:
  - **`cwnd`** set to 1 segment (1 MSS);
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- Loss indicated by 3 duplicate ACKs:
  - **`cwnd`** is cut in half window then grows linearly
  - dup ACKs indicate network capable of delivering some segments

TCP Reno

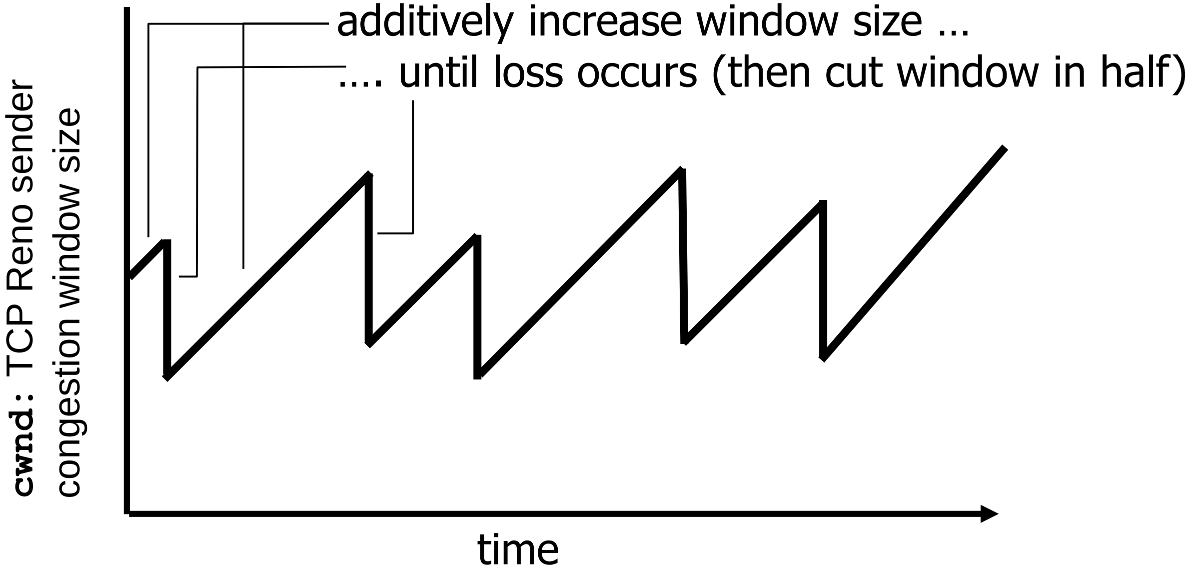
Note: This picture assumes losses are detected via duplicate ACKs. Timeouts still go to 1 and slow start up to ssthresh.



Congestion Avoidance: AIMD

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **`cwnd`** by 1 MSS (Maximum Segment Size) every RTT until loss detected
  - *multiplicative decrease*: cut **`cwnd`** in half after loss

AIMD saw tooth behavior: probing for bandwidth



TCP Variants

- There are tons of them!
  See: https://en.wikipedia.org/wiki/TCP_congestion_control

- Tahoe, Reno, New Reno, Vegas, Hybla, BIC, CUBIC, Westwood, Compound TCP, DCTCP, YeAH-TCP, BBR, …

- Each tweaks and adjusts the response to congestion.

- Why not just find a cwnd value that works, and stick with it?

Challenges (Revisited)

- How can a sender determine its "fair share" of a link?

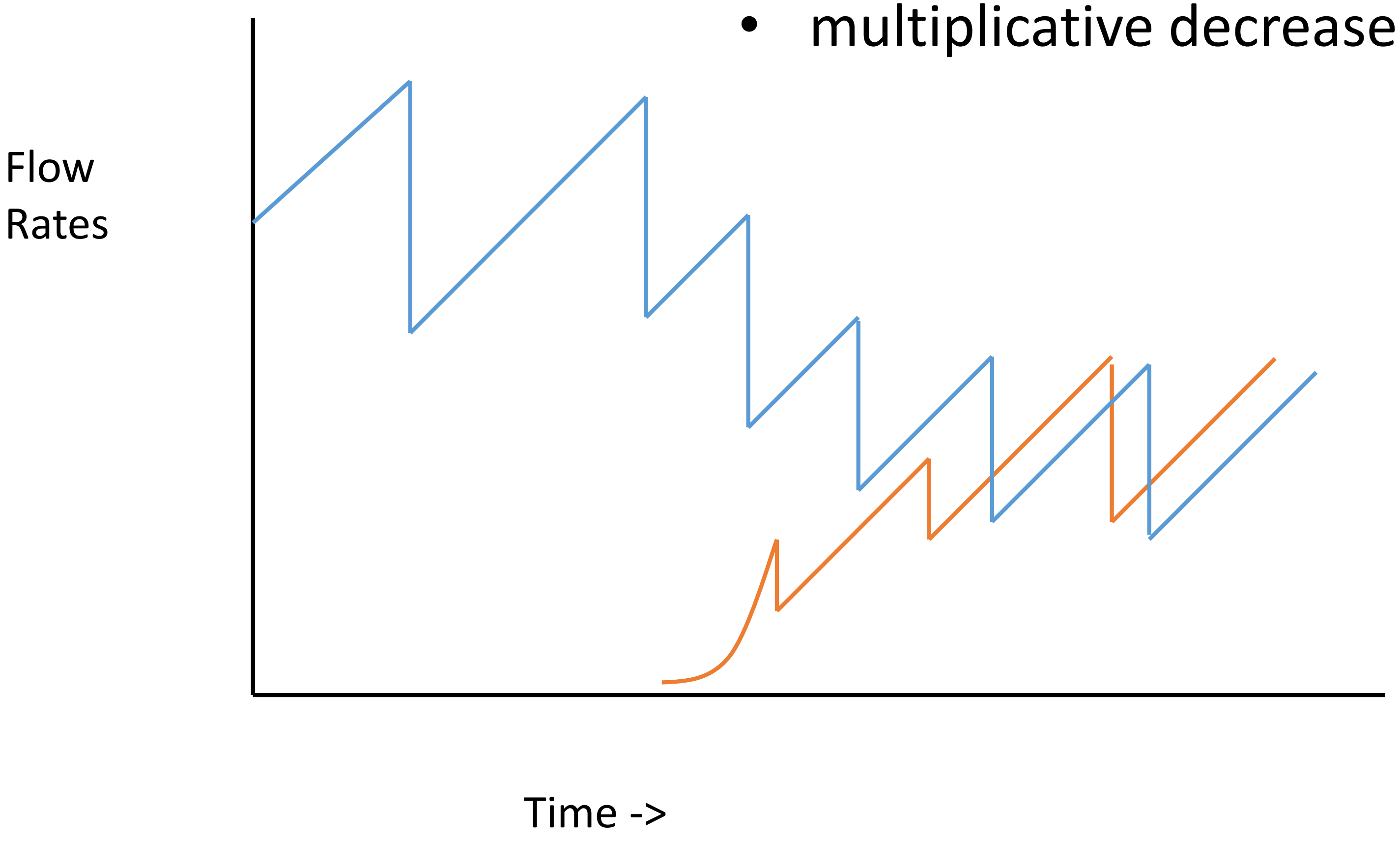
- **What if "fair shares" are changing as new conversations start or stop?**

- Initially: a sender knows nothing about the state of the network
  - Send too little, capacity goes unused
  - Send too much, cause congestion for everyone else

TCP Fairness

Two competing sessions:

- additive increase gives slope of 1, as throughput increases
- multiplicative decrease decreases throughput proportionally



Since TCP is fair, does this mean we no longer have to worry about bandwidth hogging?

A. Yep, solved it!

B. No, we can still game the system.

If you wanted to cheat to get extra traffic through, how might you do it?

Fairness (more)

Fairness and UDP

- Multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- Instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

- Application can open multiple parallel connections between two hosts
- Web browsers do this
- e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

Summary

- TCP has mechanisms to control sending rate:
  - Flow control: don’t overload receiver
  - Congestion control: don’t overload network

- `min(rwnd, cwnd)` determines window size for TCP segment pipelining (typically `cwnd`)

- Two congestion control phases (TCP Reno):
  - slow start: multiplicative increase, up to a threshold (if not first time)
  - congestion avoidance: additive increase, multiplicative decrease (AIMD)