CS 43: Computer Networks

TCP Congestion Control

October 29, 2020

SWARTHMORE COLLEGE

Slides Courtesy: Kurose & Ross, K. Webb, D. Choffnes

Moving down a layer!

Application Layer

Transport: end-to-end connections, reliability

Network: routing

Link (data-link): framing, error detection

Physical: 1’s and 0’s/bits across a medium (copper, the air, fiber)

Transport Layer perspective

Application is the boss

Transport: executing within the OS kernel

Network: ours to command!

Practical Reliability Questions

- What does connection establishment look like?
- How should we choose timeout values?
- How do we choose sequence numbers?
- How do the sender and receiver keep track of outstanding pipelined segments?
- How many segments should be pipelined?

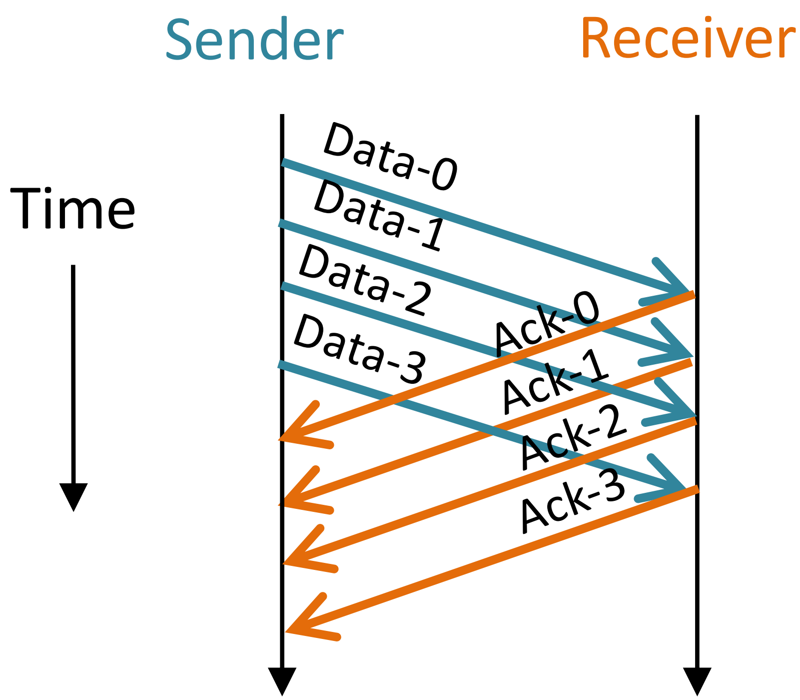
Practical Reliability Questions

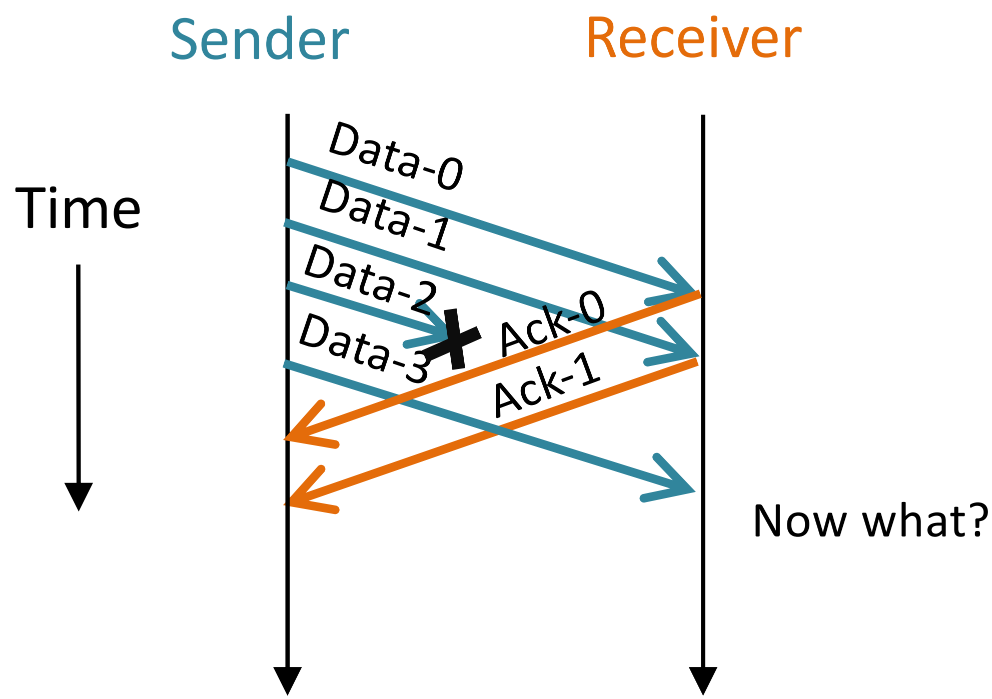
- What does connection establishment look like?
- How should we choose timeout values?
- How do we choose sequence numbers?
- How do the sender and receiver keep track of outstanding pipelined segments?
- How many segments should be pipelined?

Practical Reliability Questions

- What does connection establishment look like?
- How do we choose sequence numbers?
- How should we choose timeout values?
- How do the sender and receiver keep track of outstanding pipelined segments?
- How many segments should be pipelined?

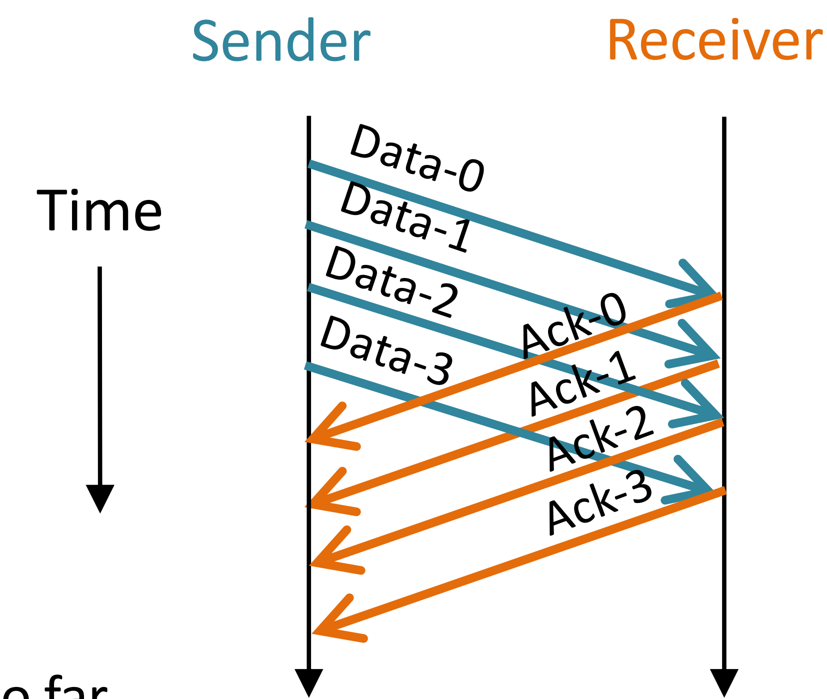
With pipelined data segments What should the sender keep track off? How about the receiver?



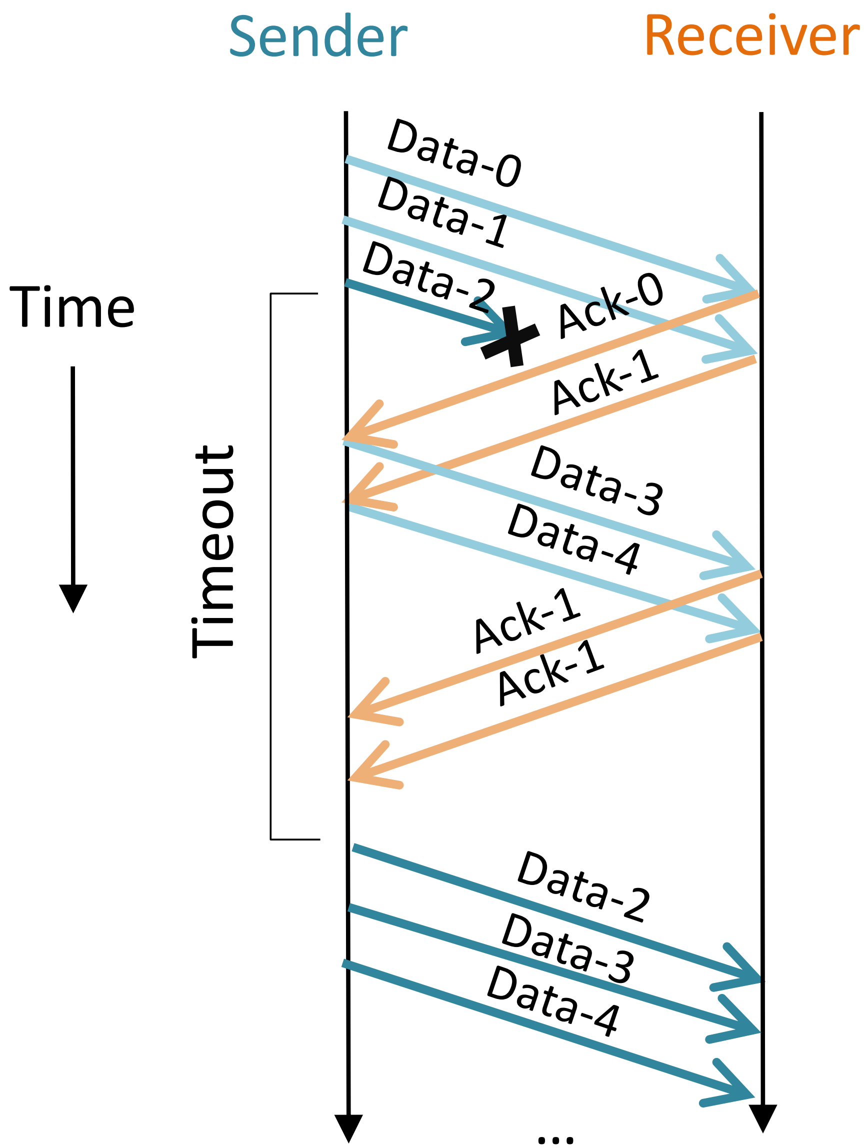


Windowing (Sliding Window)

- At the sender:
  - What's been ACKed
  - What's still outstanding
  - What to send next
- At the receiver:
  - Go-back-N
    - Highest sequence number received so far.
  - (Selective repeat)
    - Which sequence numbers received so far.
    - Buffered data.



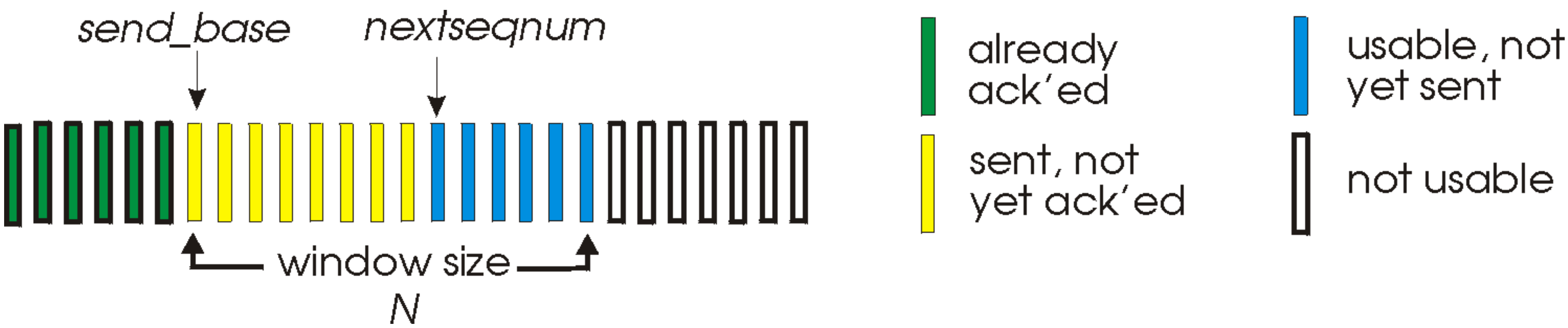
Recall: ARQ Protocol: Go-Back-N



- Retransmit from point of loss
  - Segments between loss event and retransmission are ignored
  - "Go-back-N" if a timeout event occurs

Go-back-N

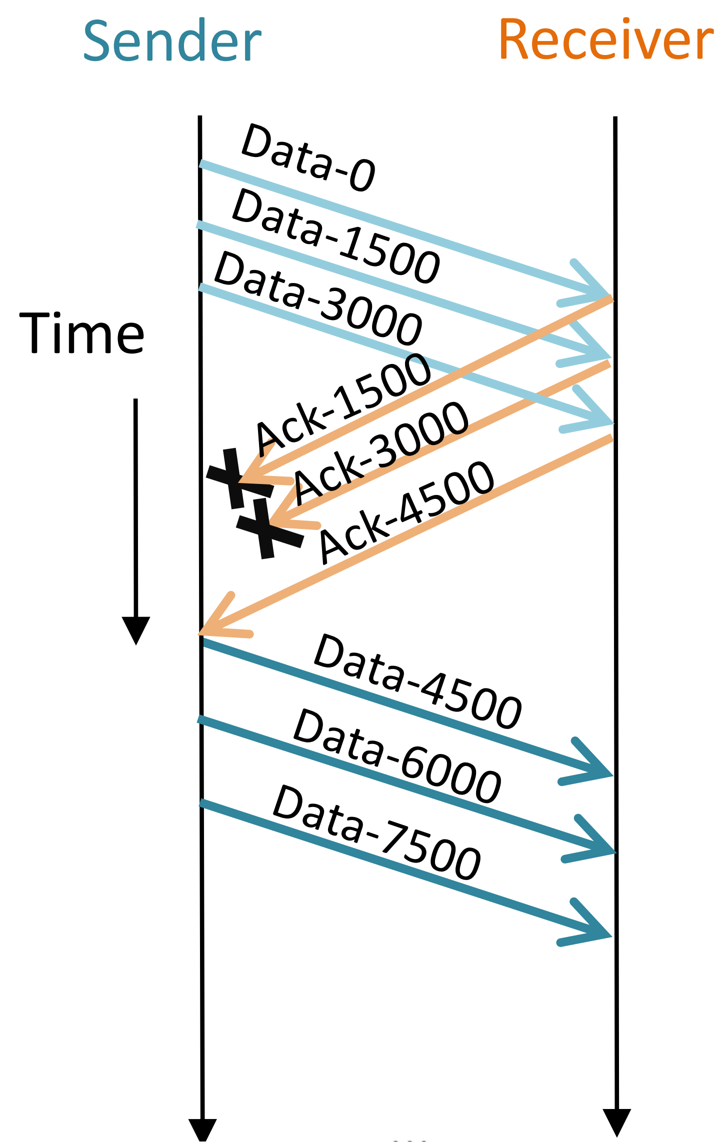
- At the sender:



- At the receiver:
  – Keep track of largest sequence number seen.
  – If it receives ANYTHING, sends back ACK for largest sequence number seen so far. (Cumulative ACK)

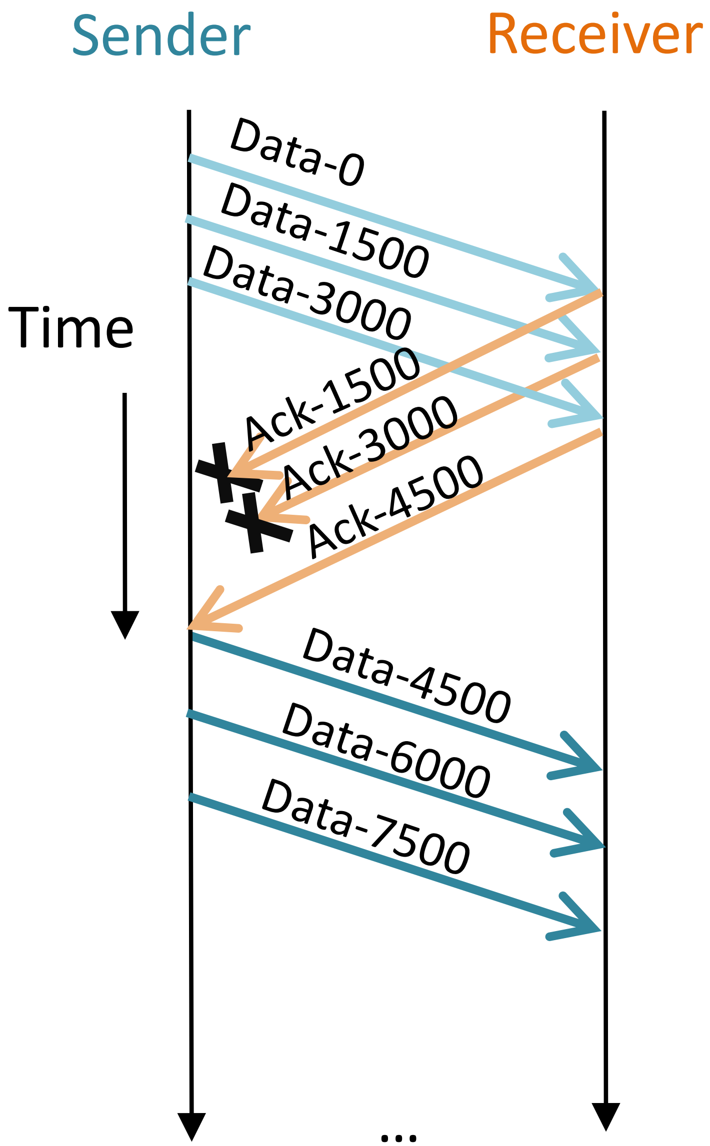
Cumulative Acknowledgements

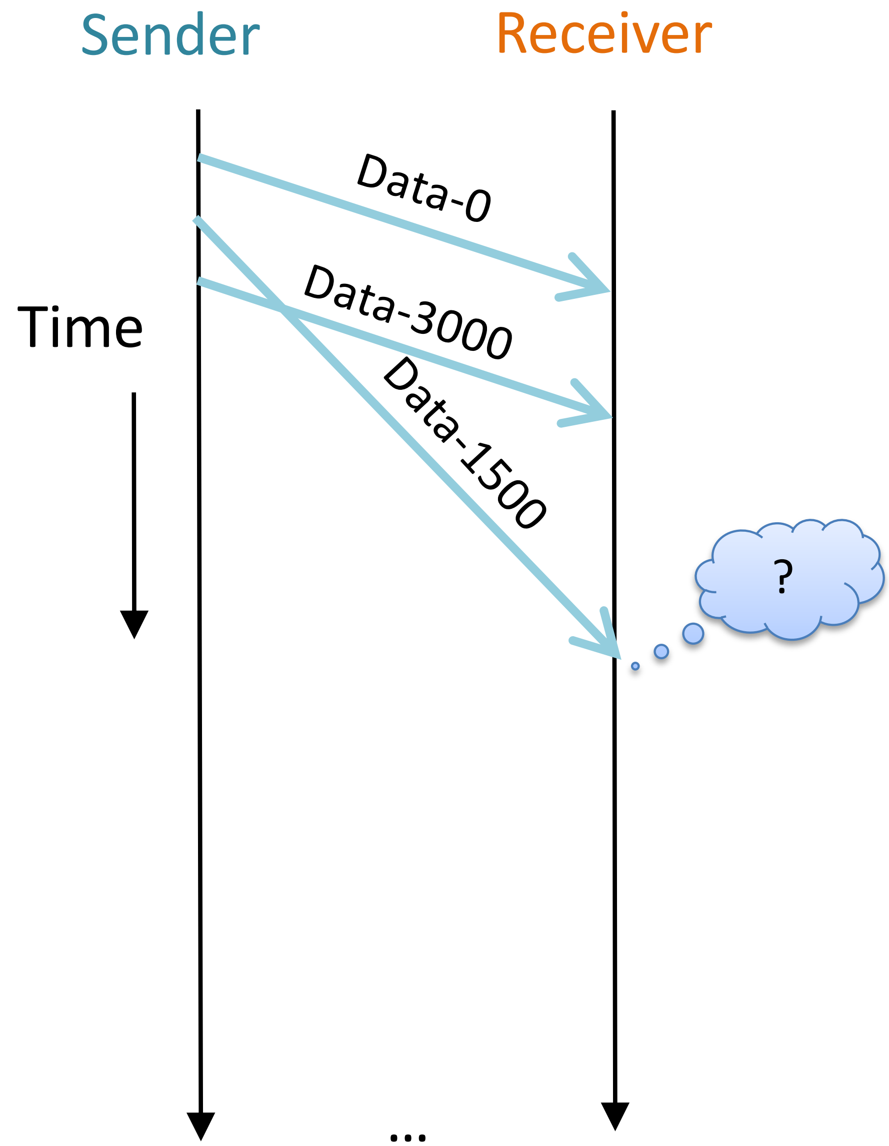
An ACK for sequence number N implies that all data prior to N has been received.



Cumulative Acknowledgements

An ACK for sequence number N implies that all data prior to N has been received.





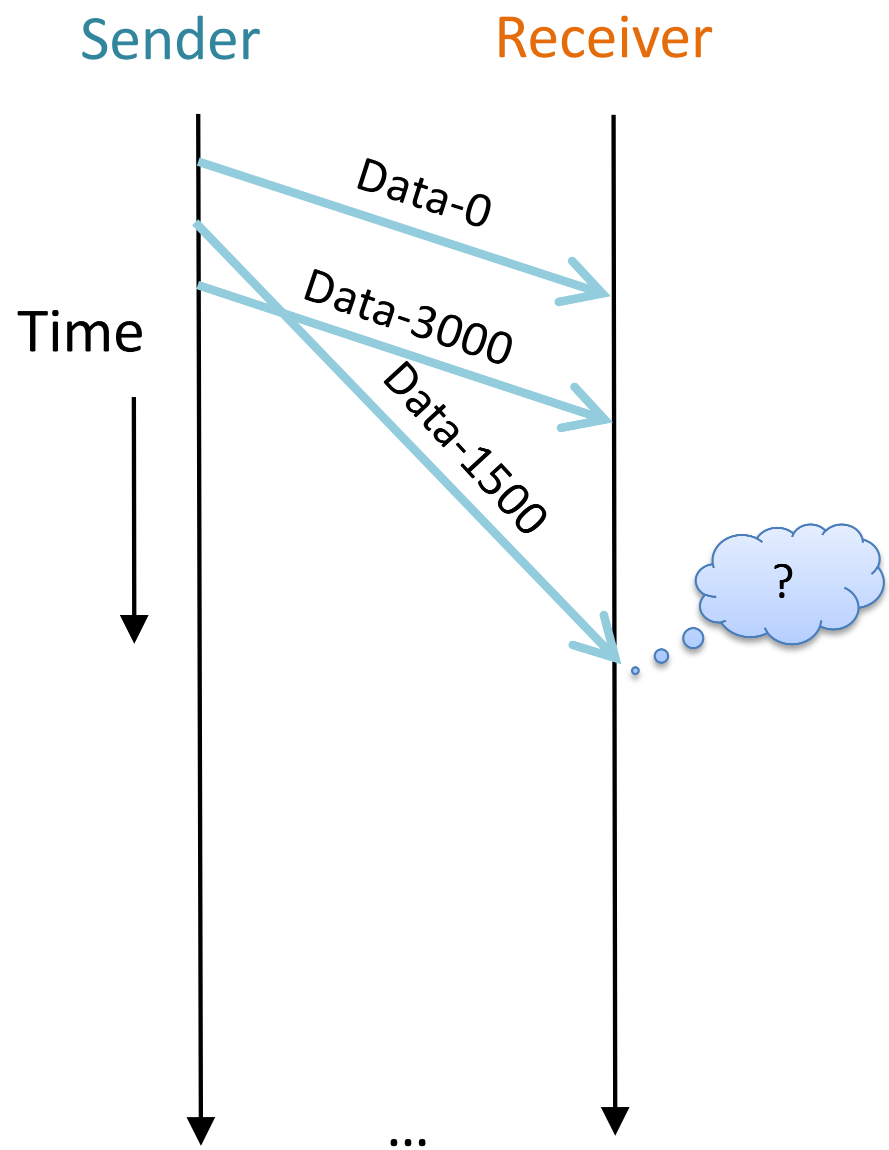
What should we do with an out-of-order segment at the receiver?

A. Drop it.

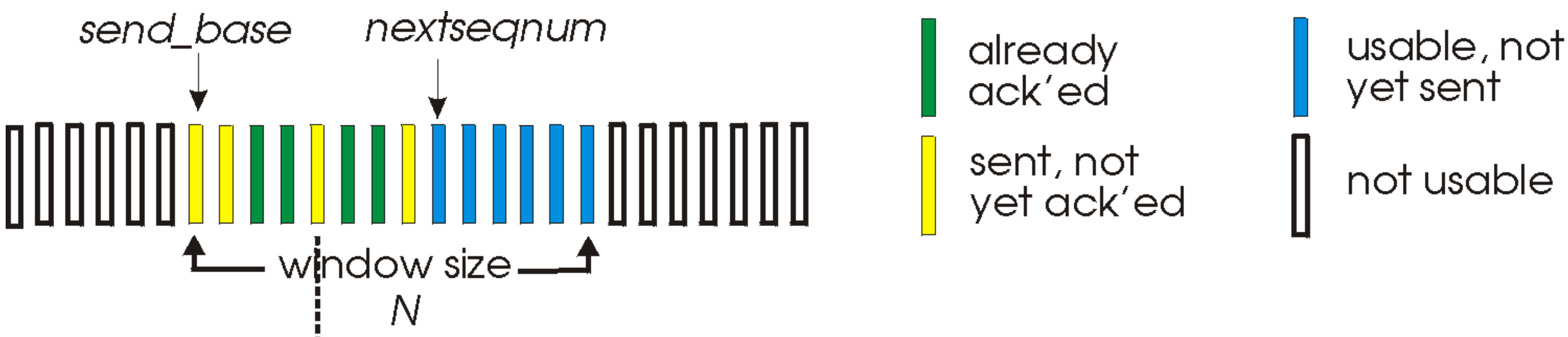
B. Save it and ACK it.

C. Save it, don't ACK it.

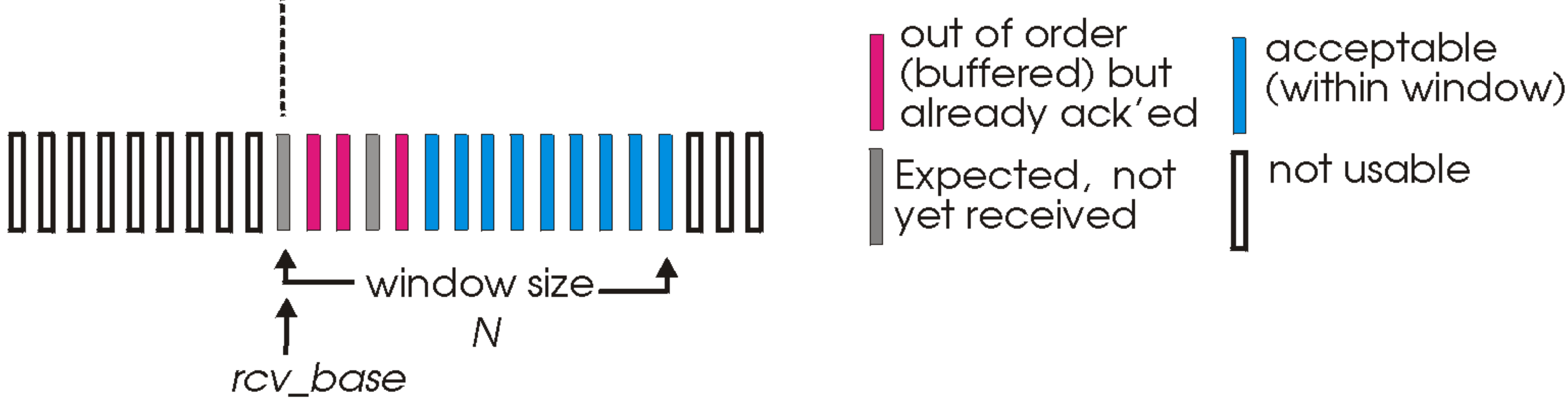
D. Something else (explain).



Selective Repeat



(a) sender view of sequence numbers



(b) receiver view of sequence numbers

If you were building a transport protocol, which would you use?

A. Go-back-N

B. Selective repeat

C. Something else (explain)

Sliding window

- How many bytes to pipeline?
- How big do we make that window?
  - Too small: link is under-utilized
  - Too large: congestion, packets dropped
  - Other concerns: fairness

Practical Reliability Questions

- What does connection establishment look like?
- How do we choose sequence numbers?
- How should we choose timeout values?
- How do the sender and receiver keep track of outstanding pipelined segments?
- How many segments should be pipelined?

Discussion: Why do we need rate control ?

A. to help the global network (core routers, and other end-hosts)
B. to help the receiver
C. to help the sender
D. some other reason

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

Rate Control

Flow Control

- Don't send so fast that we overload the <u>receiver</u>.
- Rate directly negotiated between one pair of hosts (the sender and receiver).

Congestion Control

- Don't send so fast that we overload the <u>network</u>.
- Rate inferred by sender in response to "congestion events."

<u>Shared high-level goal: don't waste capacity by sending something that is likely to be dropped.</u>

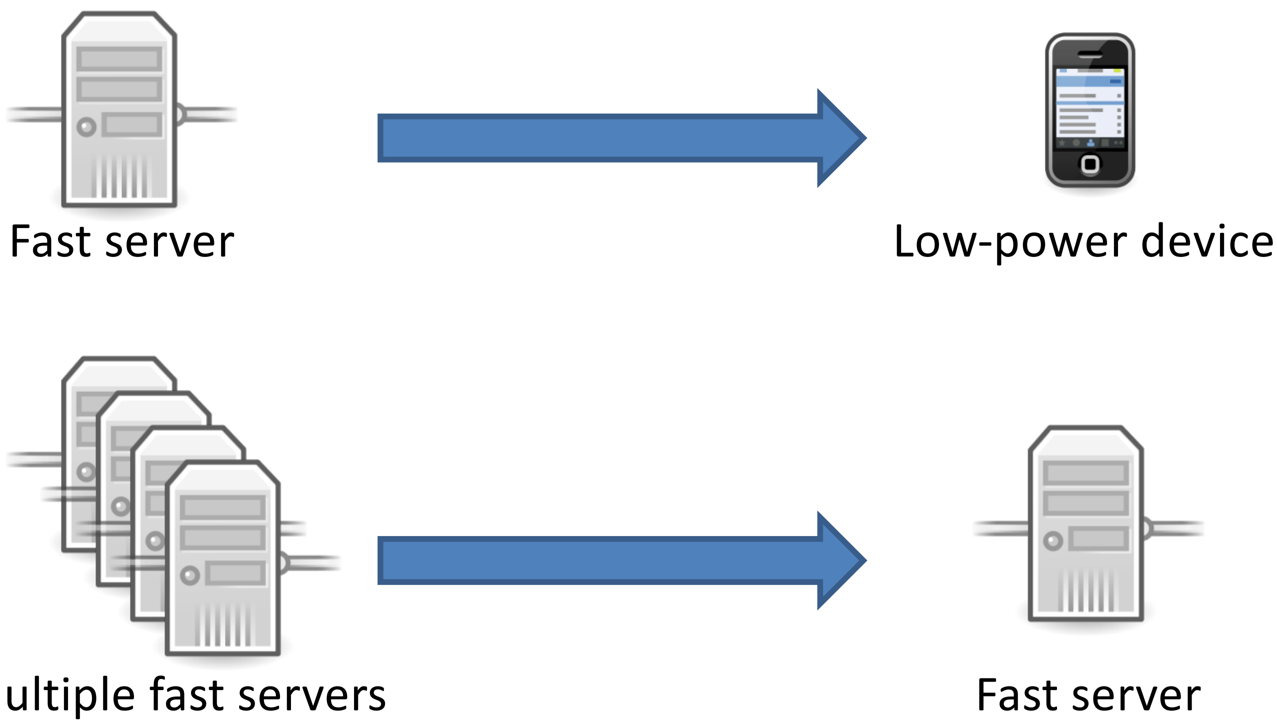
Flow Control

- Don't send so fast that we overload the receiver.
- Rate directly negotiated between one pair of hosts (the sender and receiver).

Flow Control

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

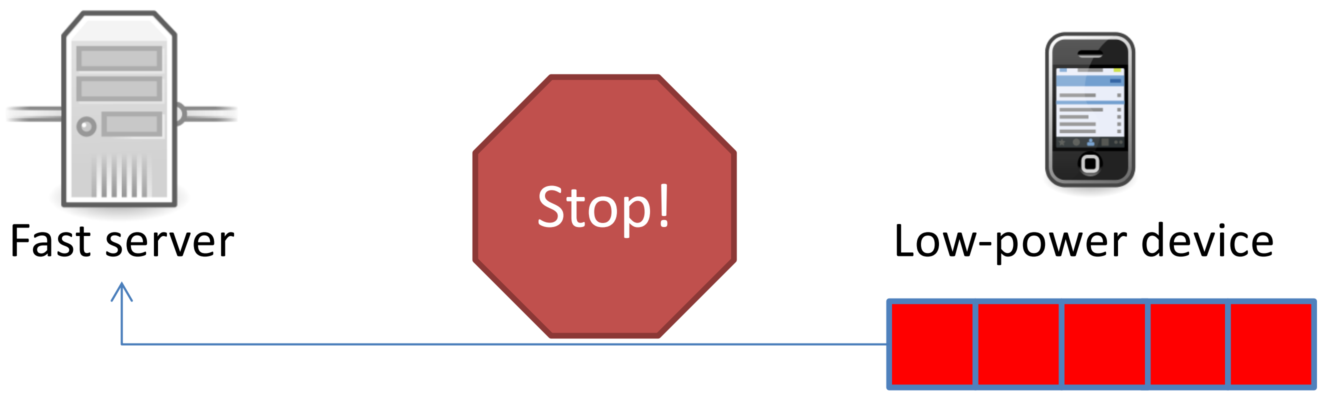
- Example scenarios:



Flow Control

Finite socket buffer space at the receiver.

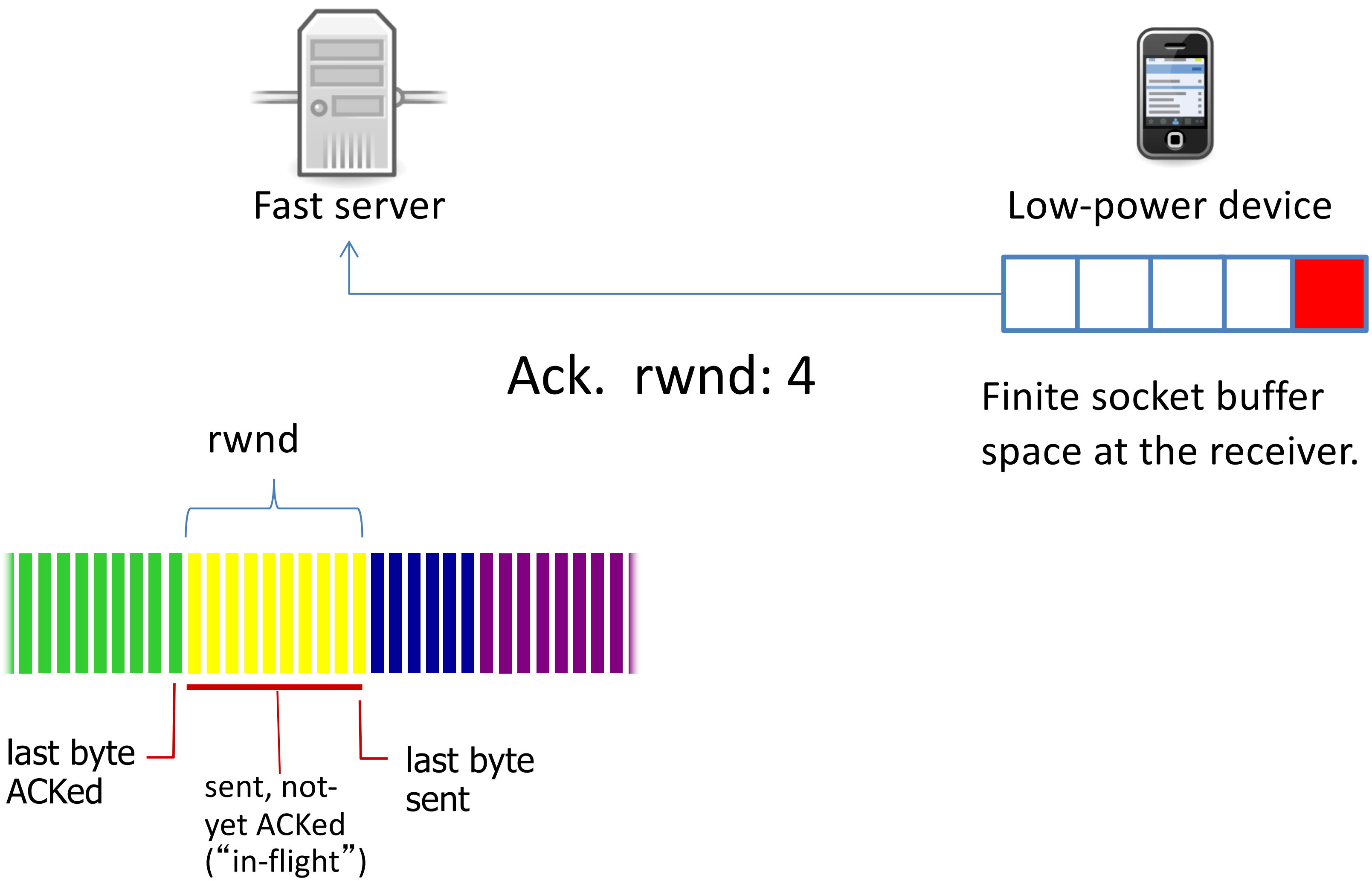
Flow Control



Finite socket buffer
space at the receiver.

Flow Control

- Sender never sends more than rwnd.

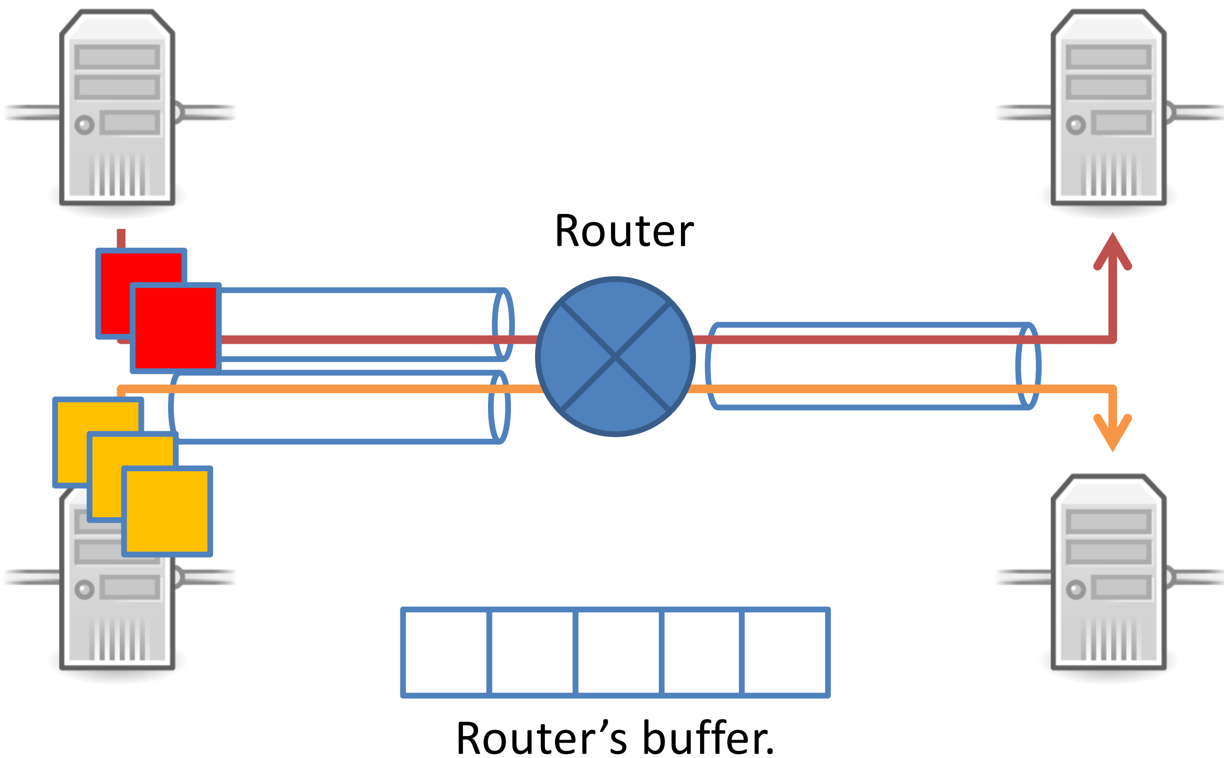


Congestion

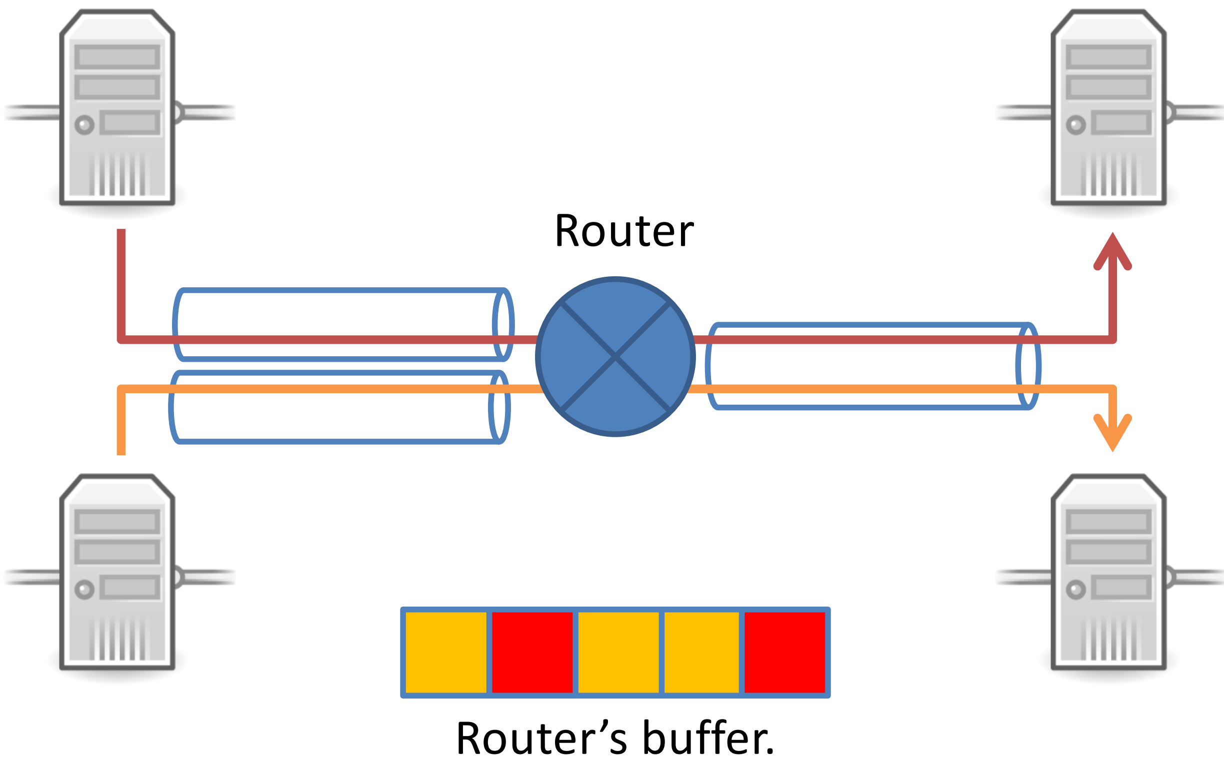
- Flow control is (relatively) easy. The receiver knows how much space it has.

- What about the network devices?

Congestion

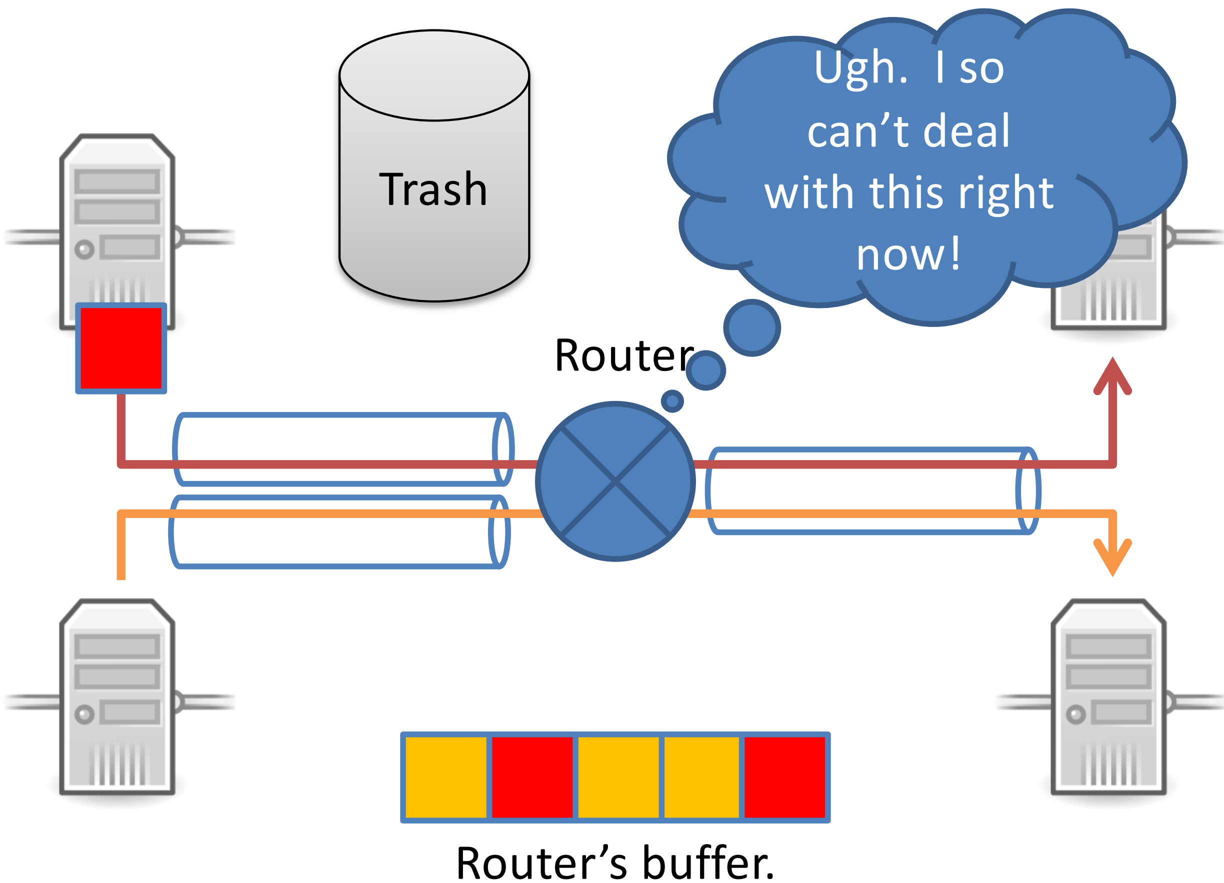


Congestion



Incoming rate is faster than outgoing link can support.

Congestion



Incoming rate is faster than outgoing link can support.

What’s the worst that can happen?

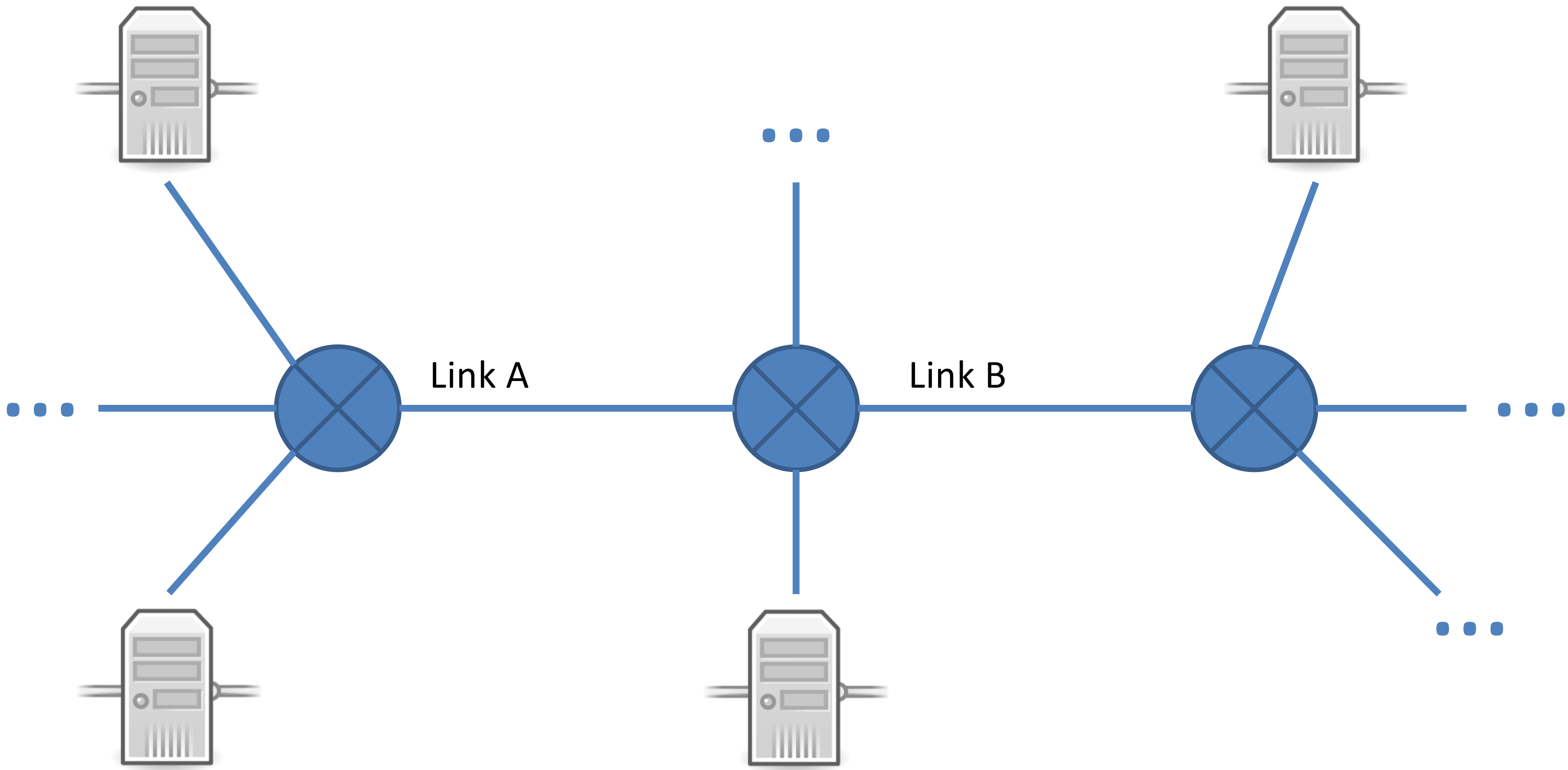
A. This is no problem. Senders just keep transmitting, and it’ll all work out.

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

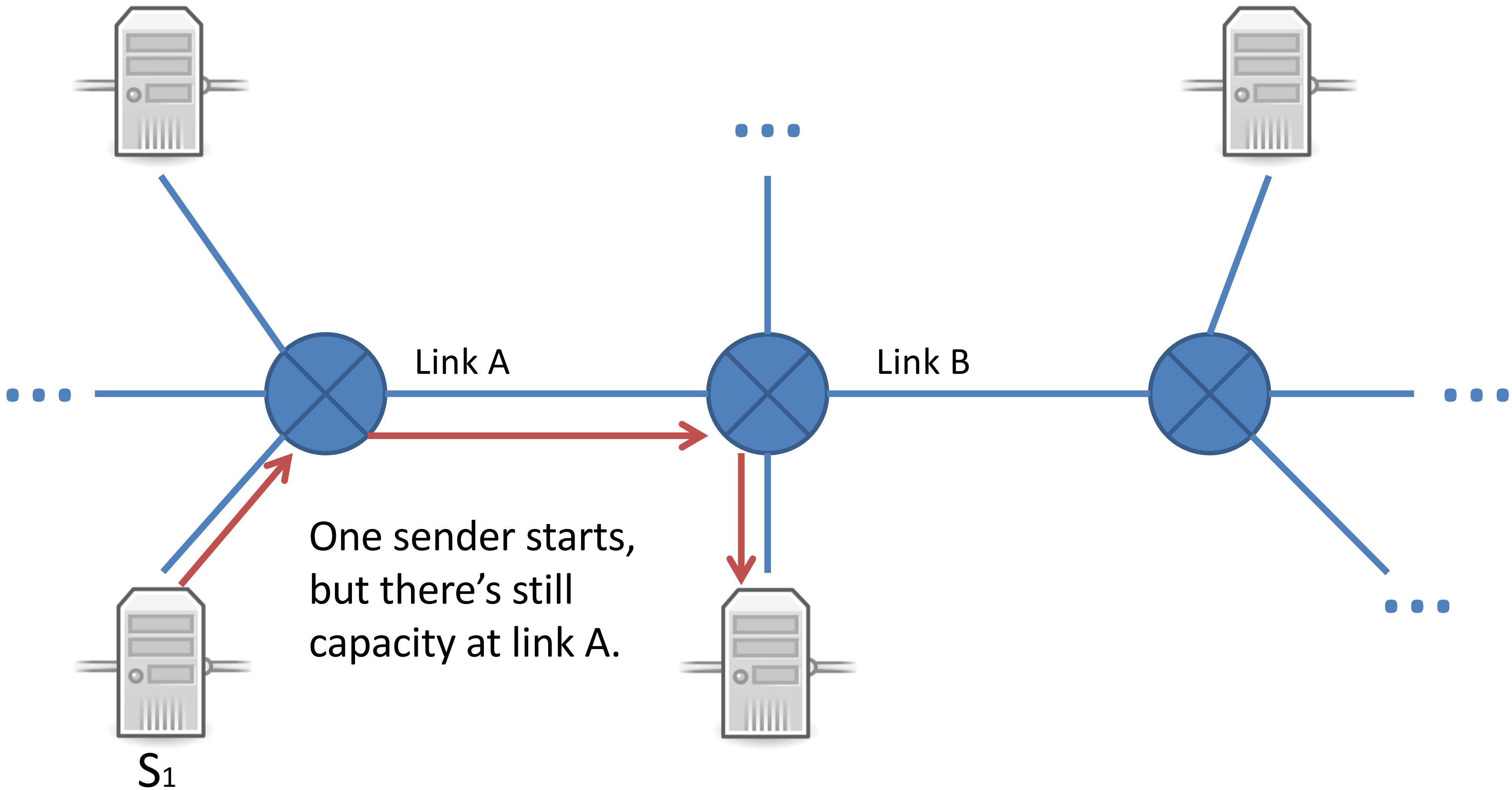
C. Retransmissions will become very frequent, causing a serious loss of efficiency.

D. The network will become completely unusable.

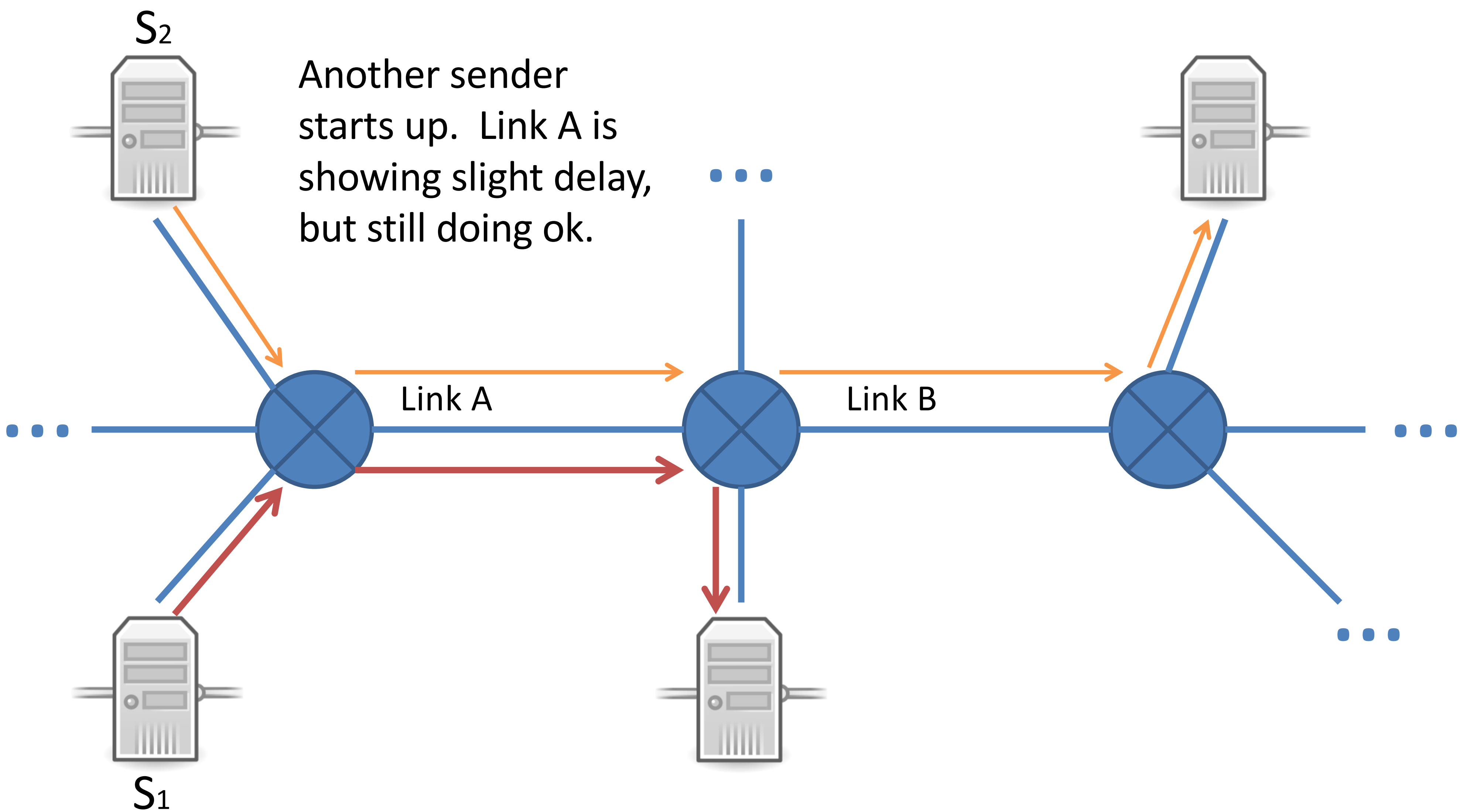
Congestion Collapse



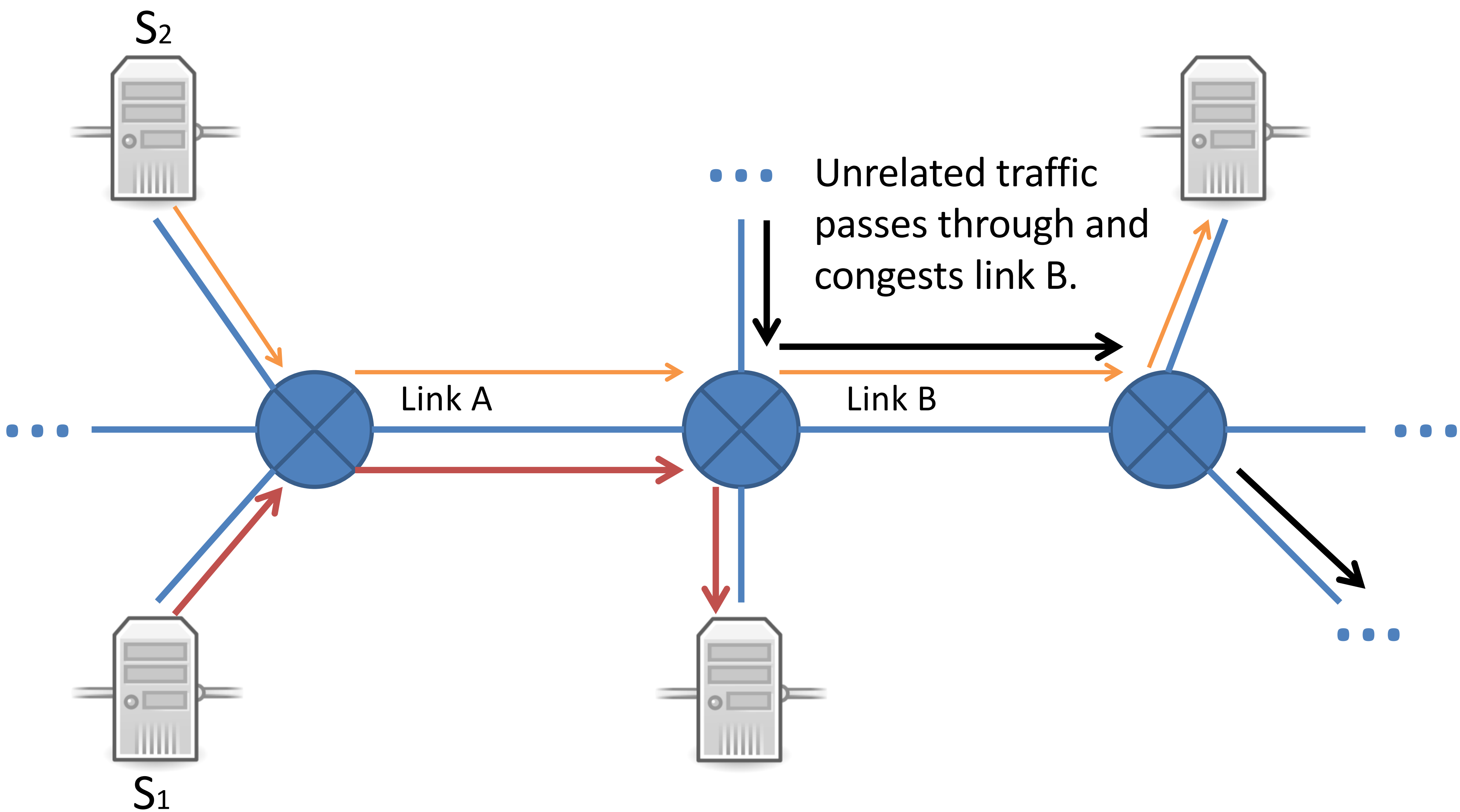
Congestion Collapse



Congestion Collapse

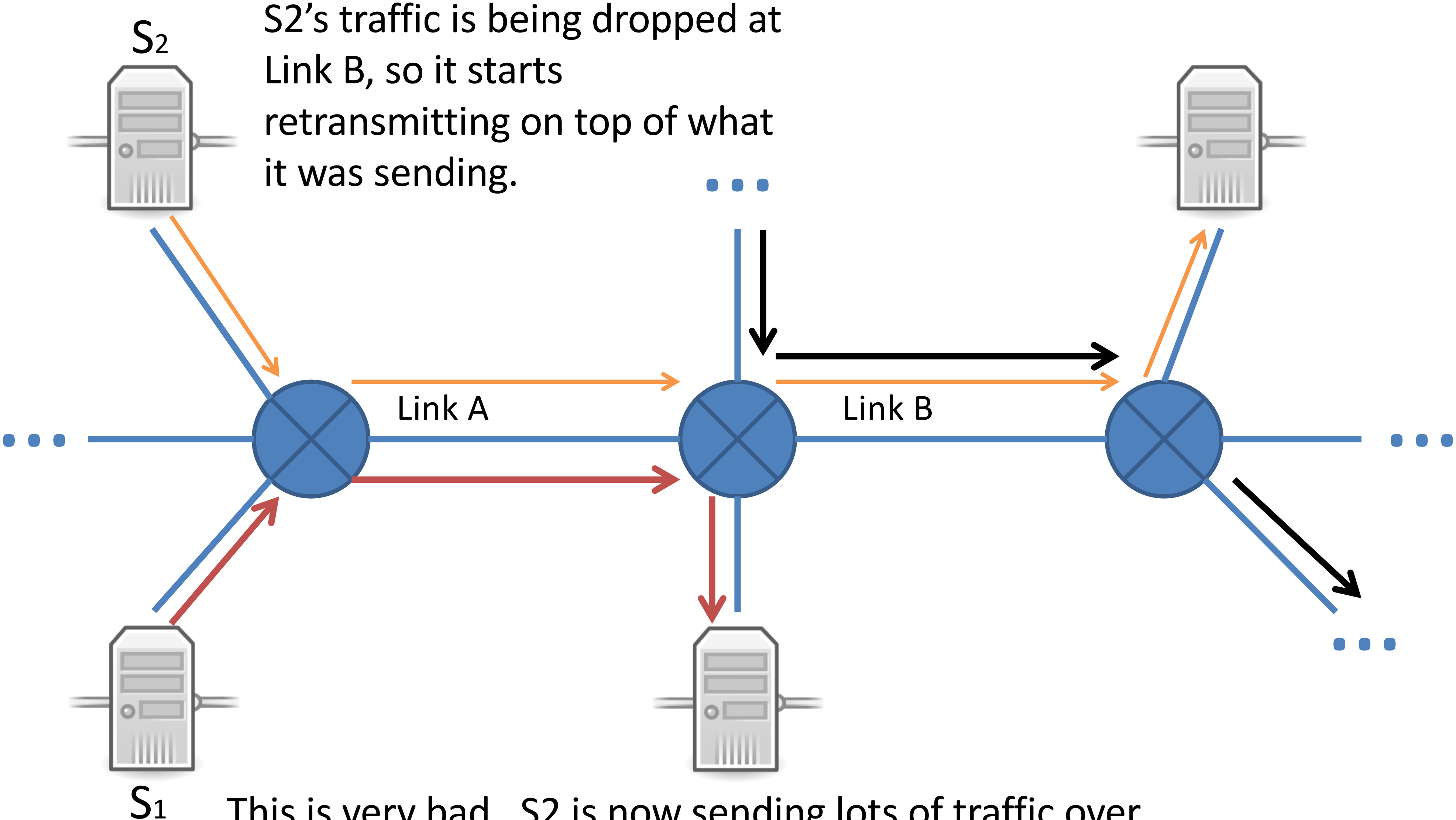


Congestion Collapse



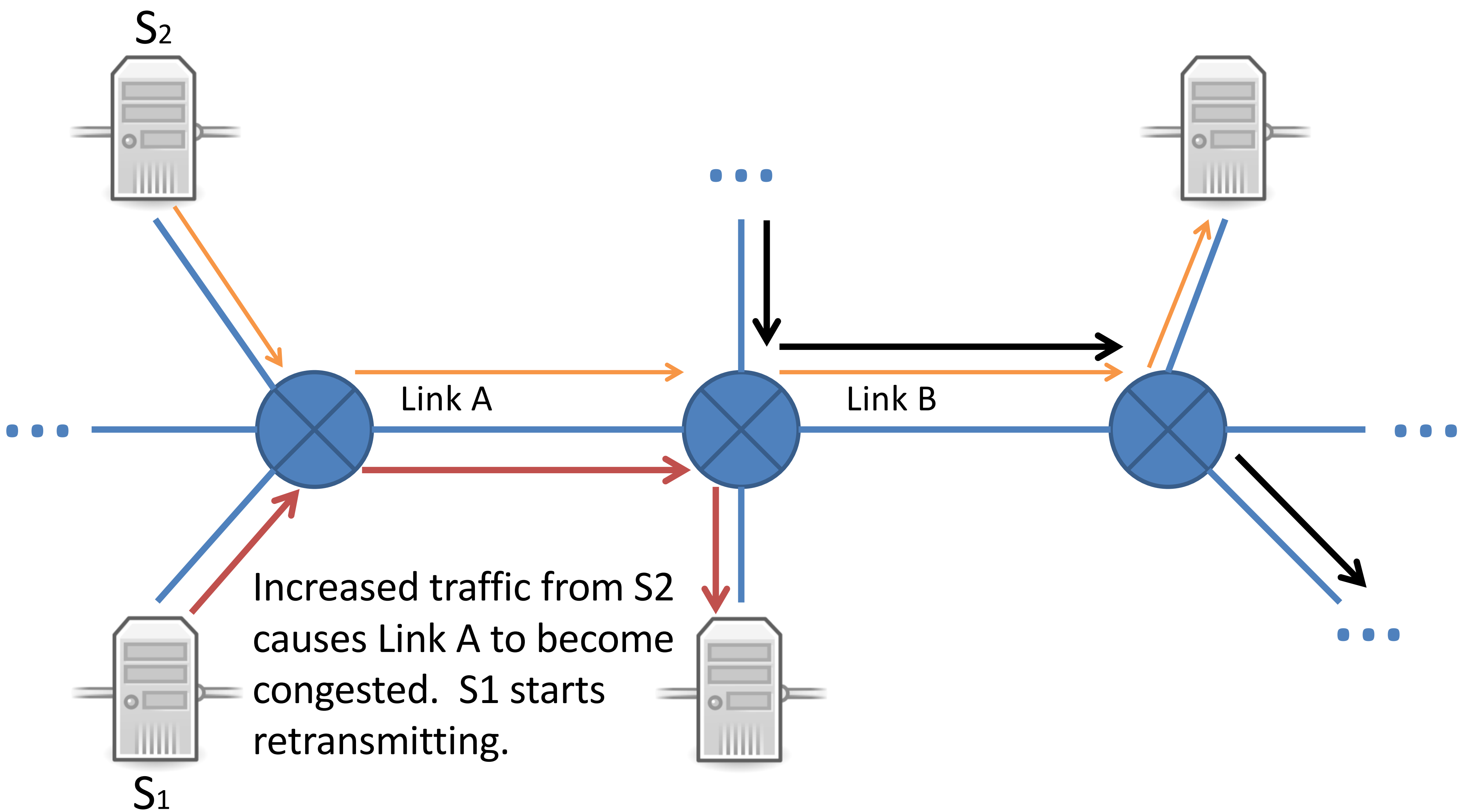
Congestion Collapse

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



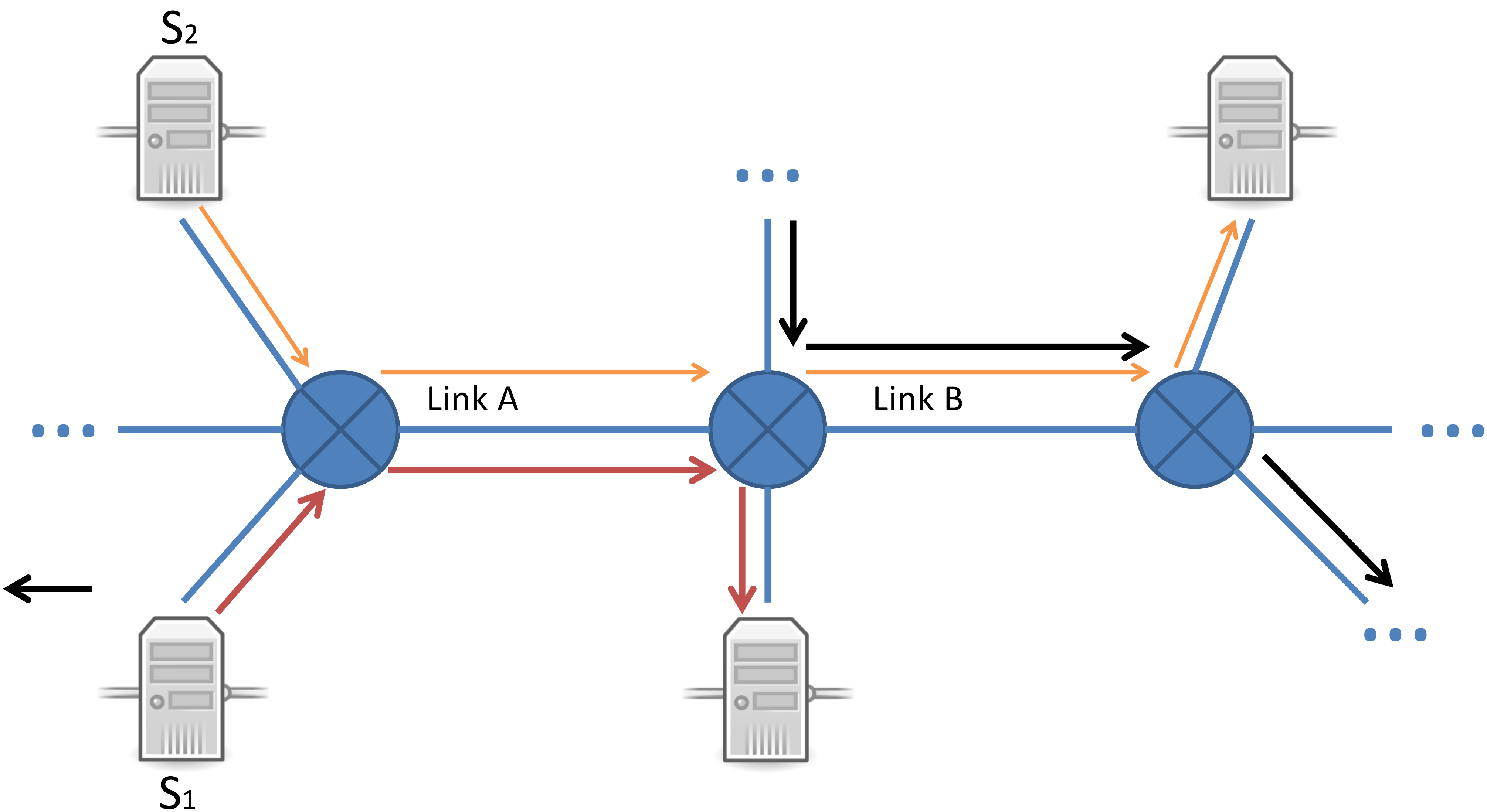
This is very bad. S2 is now sending lots of traffic over link A that has no hope of crossing link B.

Congestion Collapse



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

Congestion Collapse

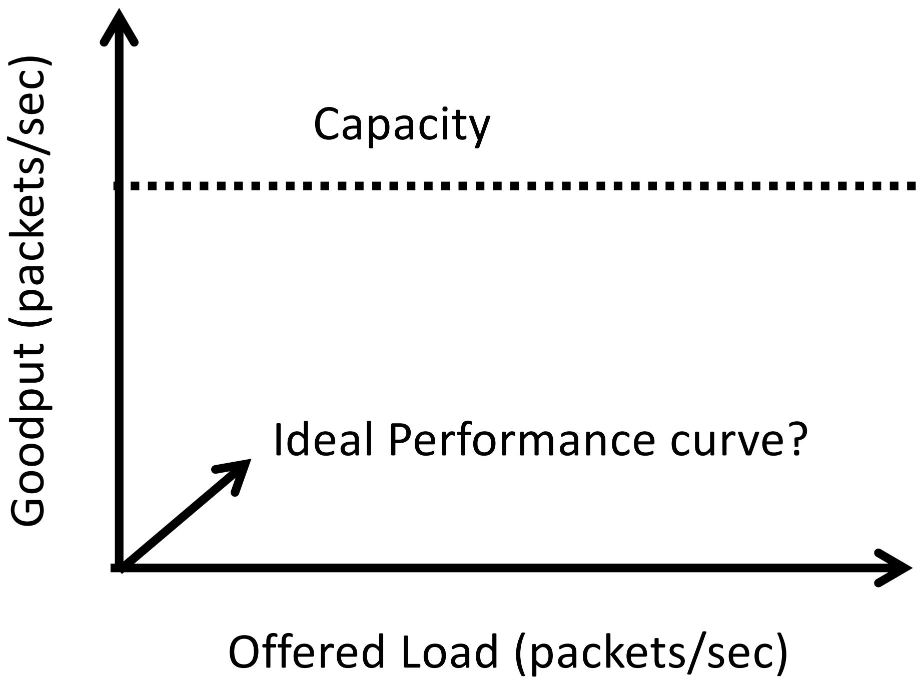


What problems do we have without congestion control?

A. Affects Latency
B. Affects loss rate
C. Affects network capacity
D. Affects application layer performance
E. More than one of the above

Effects of congestion: what happens to performance when we increase the load?

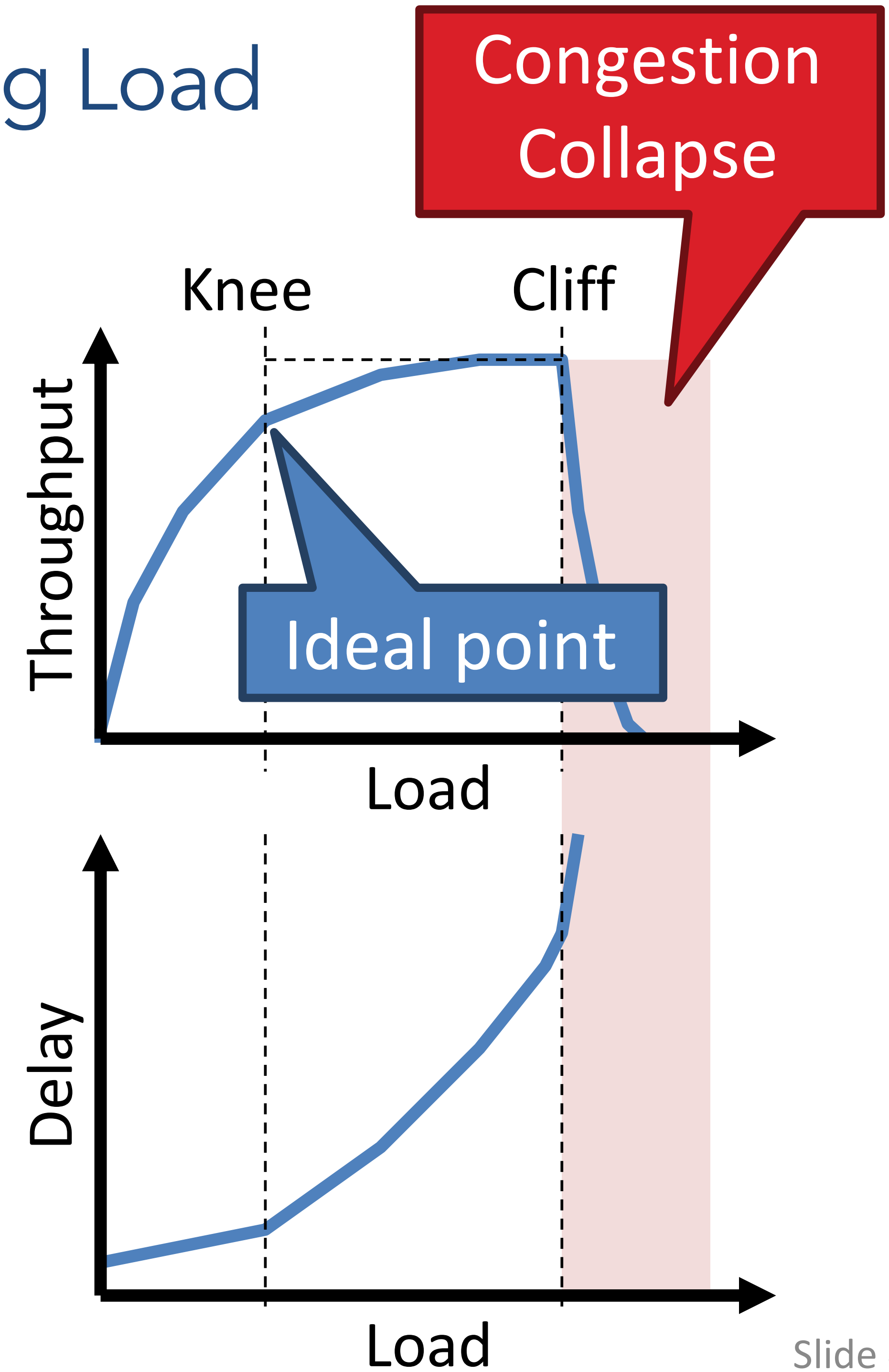
Goodput: Packets through the network that are not retransmissions
Offered Load: All packets in the network



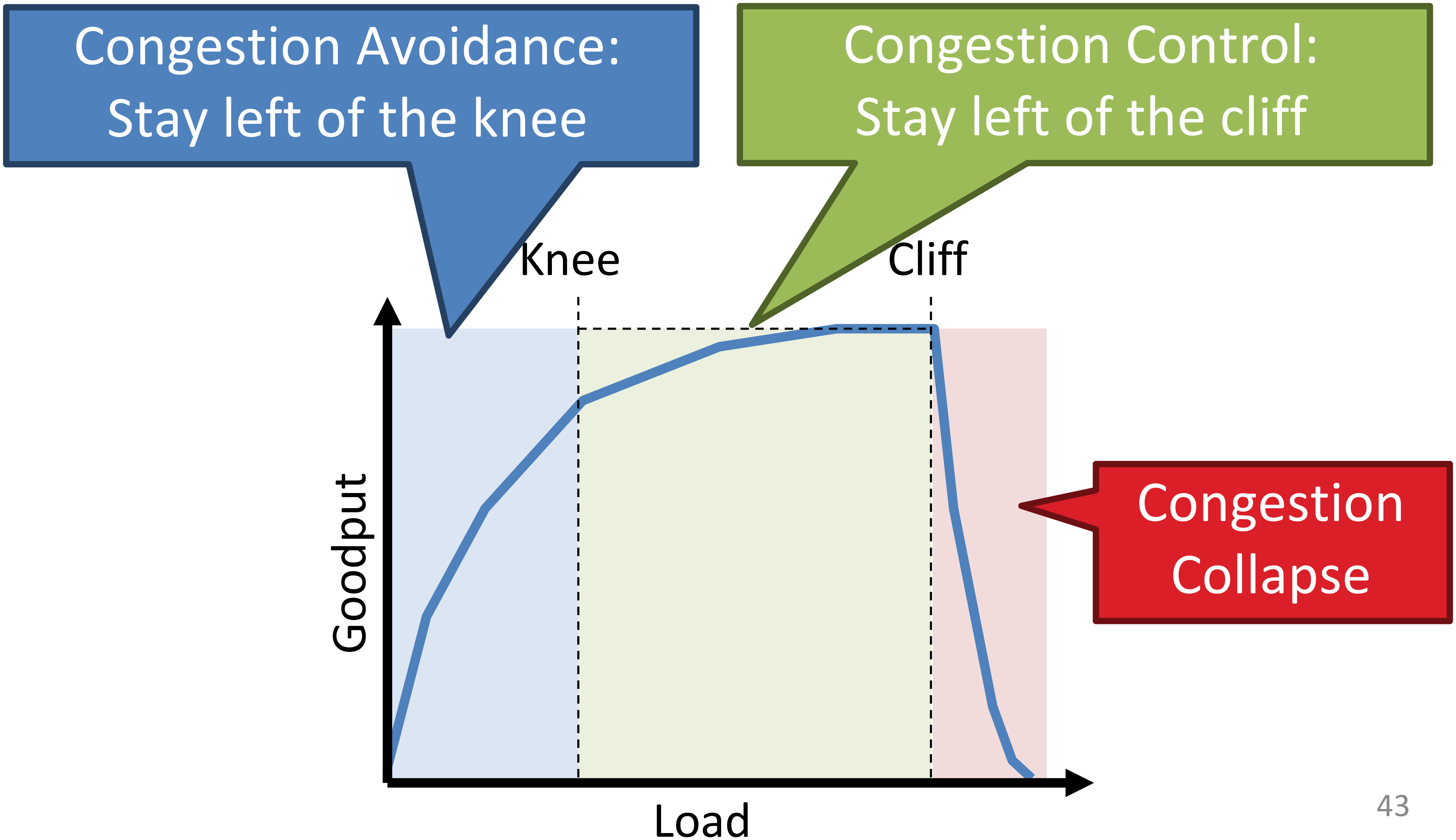
A. Linear, x = y
B. Linear 2x = y
C. Exponential growth
D. Something else

The Danger of Increasing Load

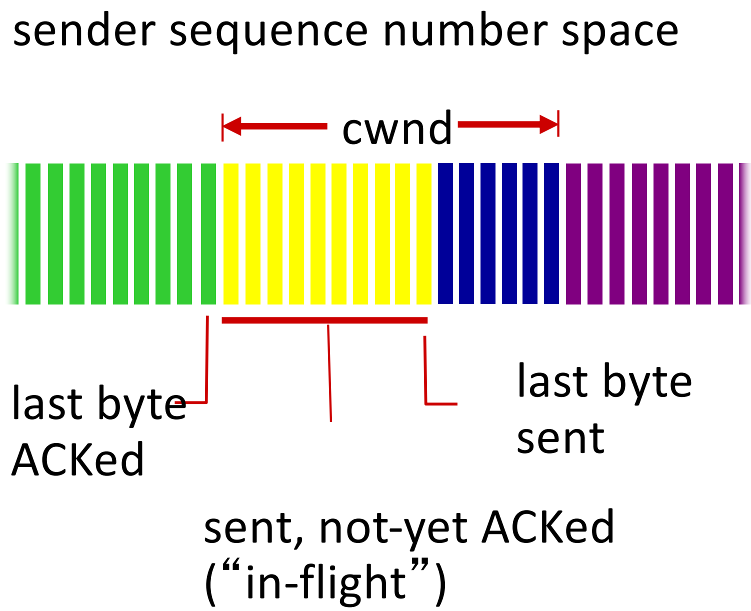
- Knee – point after which
  - Throughput increases very slow
  - Delay increases fast
- Cliff – point after which
  - Throughput → 0
  - Delay → ∞



Cong. Control vs. Cong. Avoidance



TCP Congestion Control: details



TCP sending rate:

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

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

- sender limits transmission:

$$\text{LastByteSent-LastByteAcked} \leq \text{cwnd}$$

- cwnd is dynamic, function of perceived network congestion

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 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.

What is a "congestion event" from the perspective of a sender in TCP?

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

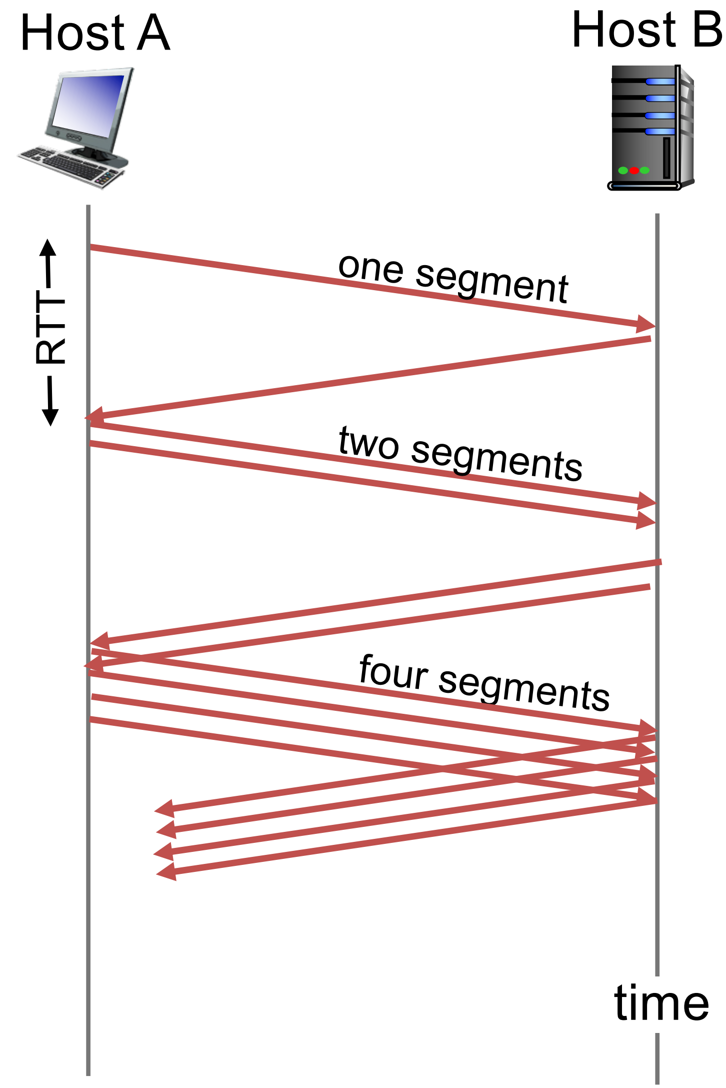
What we care about is segment loss, and both B and C give us a way to know that a segment loss has occurred.

TCP Congestion Control Phases

- Slow start
  - Sender has no idea of network's congestion
  - Start conservatively, increase rate quickly
- Congestion avoidance
  - Increase rate slowly
  - Back off when congestion occurs
    - How much depends on TCP version

TCP Slow Start

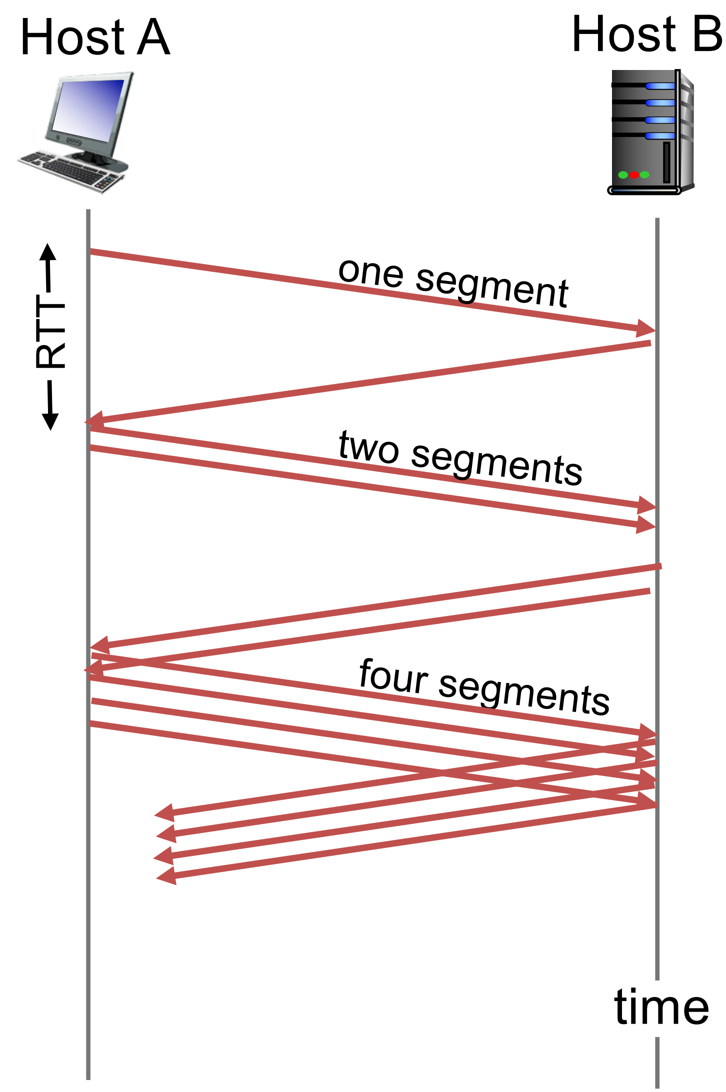
- When connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



- 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

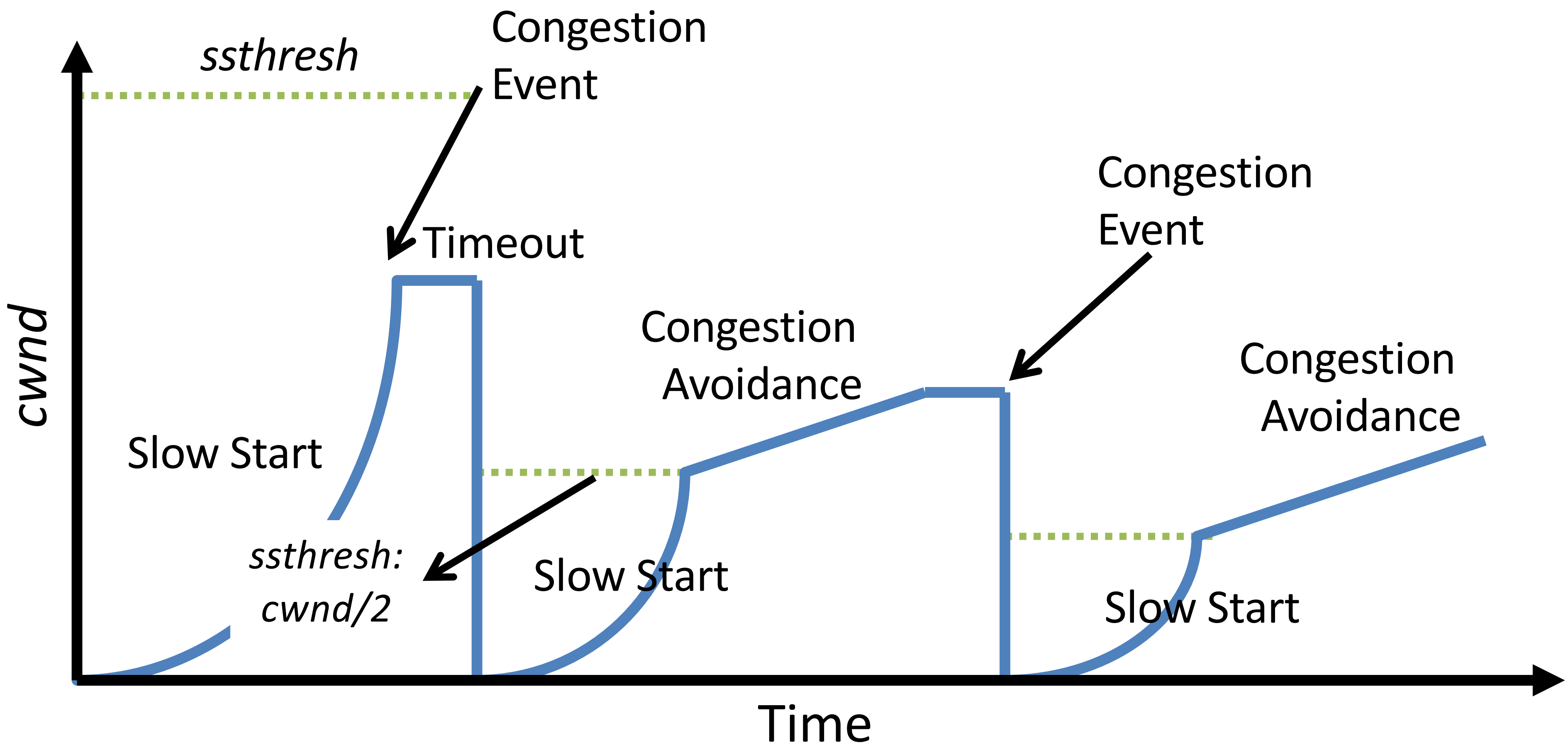


TCP Congestion Avoidance

- `ssthresh`: Threshold where slow start ends
  - initially unlimited

- 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

TCP: Big picture



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 (Tahoe vs. Reno)

Loss indicated by timeout:

Tahoe and Reno:

- cwnd set to 1 MSS;
- window then grows exponentially (as in slow start) to threshold,
- then grows linearly

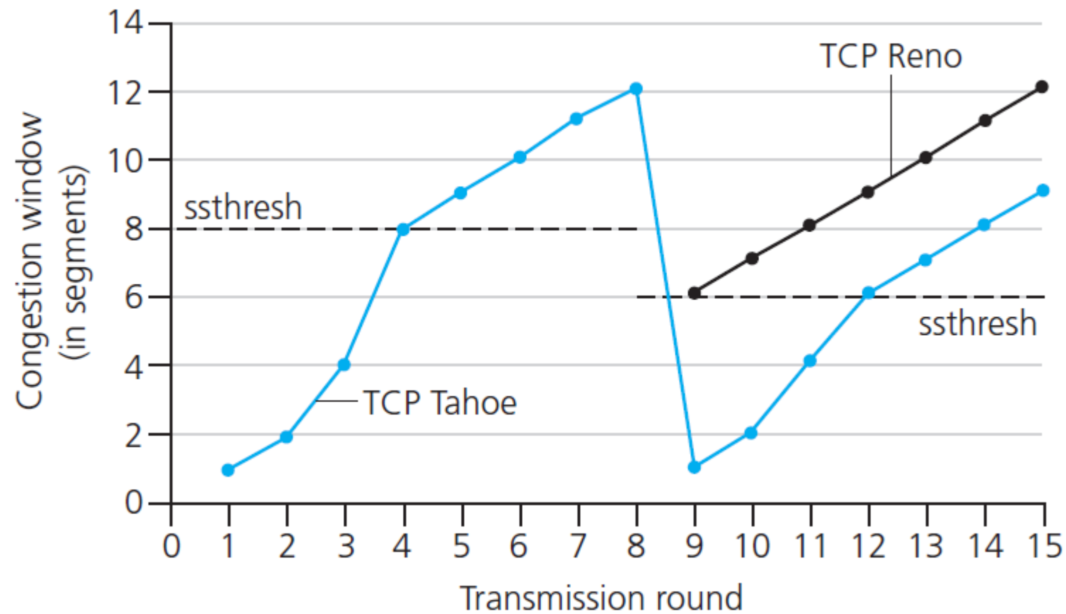
Loss indicated by 3 duplicate ACKs:

- Tahoe:
  - cwnd set to 1 MSS;
  - window grows exponentially (as in slow start) to threshold
  - then grows linearly
- Reno
  - cwnd is cut in half window then grows linearly
  - dup ACKs indicate network capable of delivering some segments

TCP: switching from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

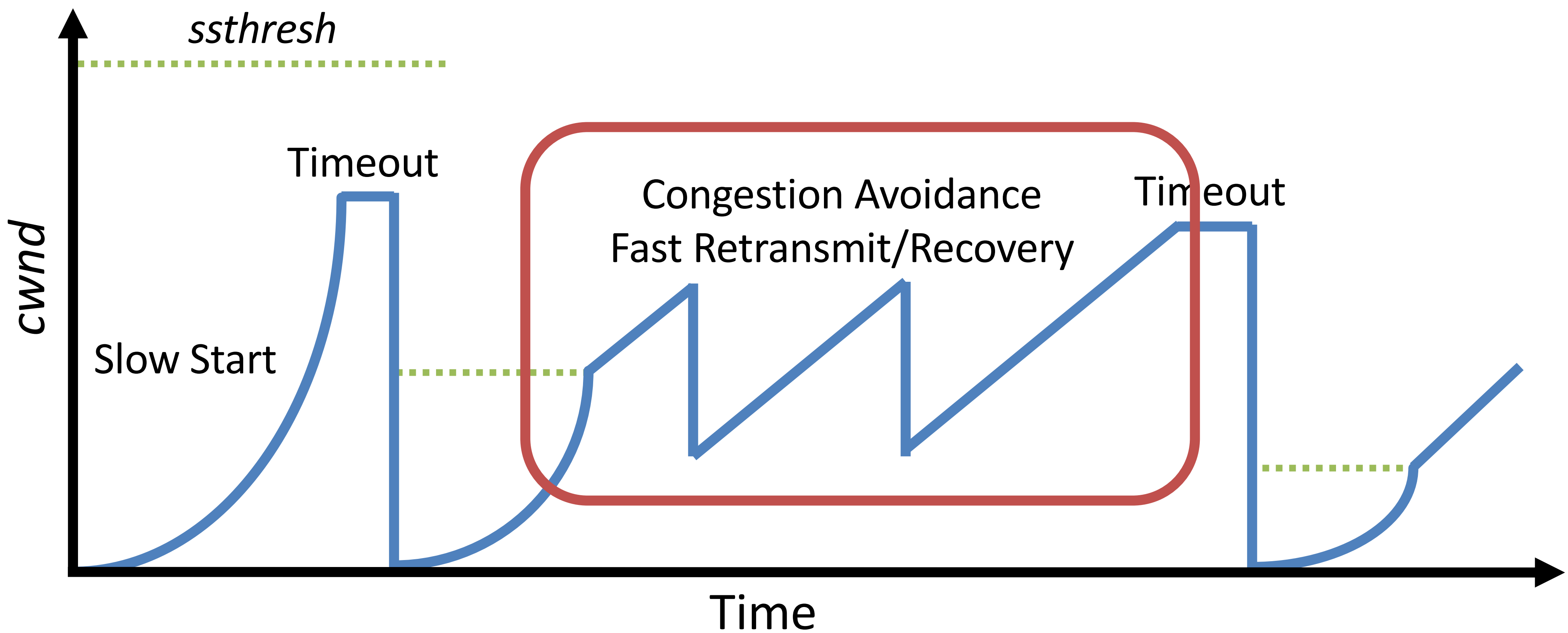
A: when cwnd gets to 1/2 of its value before timeout.



Implementation:

- variable ssthresh
- on loss event, ssthresh is set to 1/2 of cwnd just before loss event

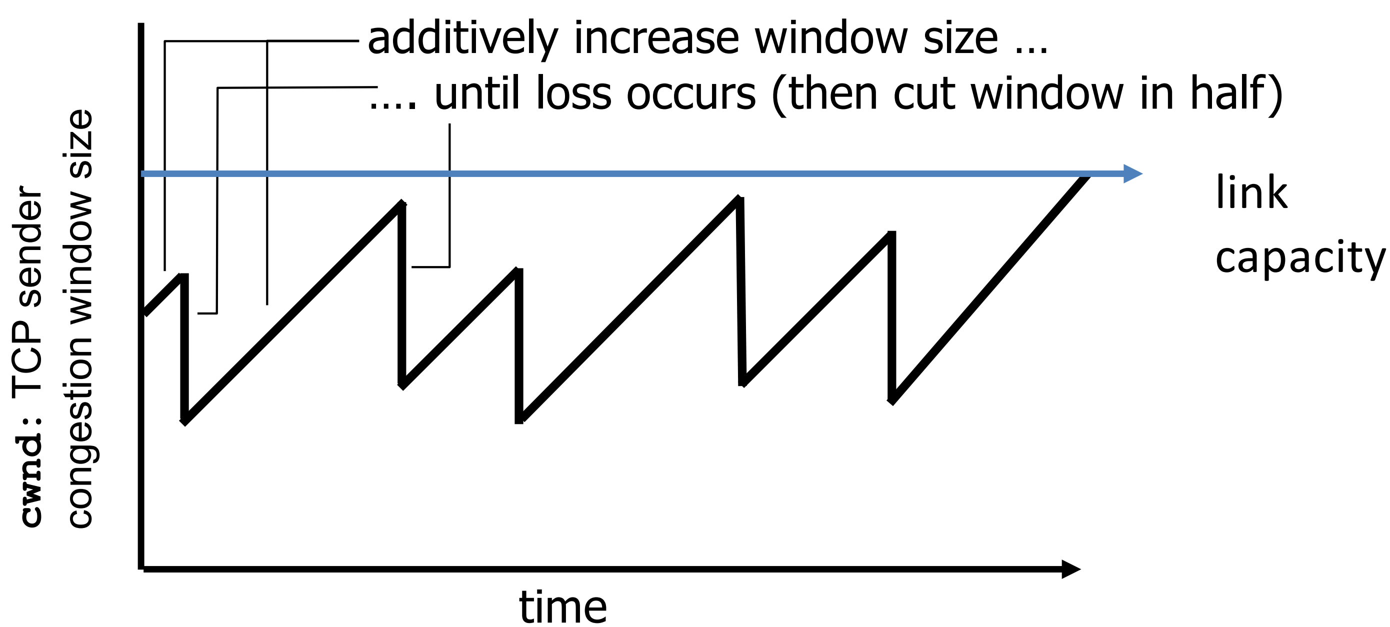
Fast Retransmit and Fast Recovery



Additive Increase, Multiplicative Decrease (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

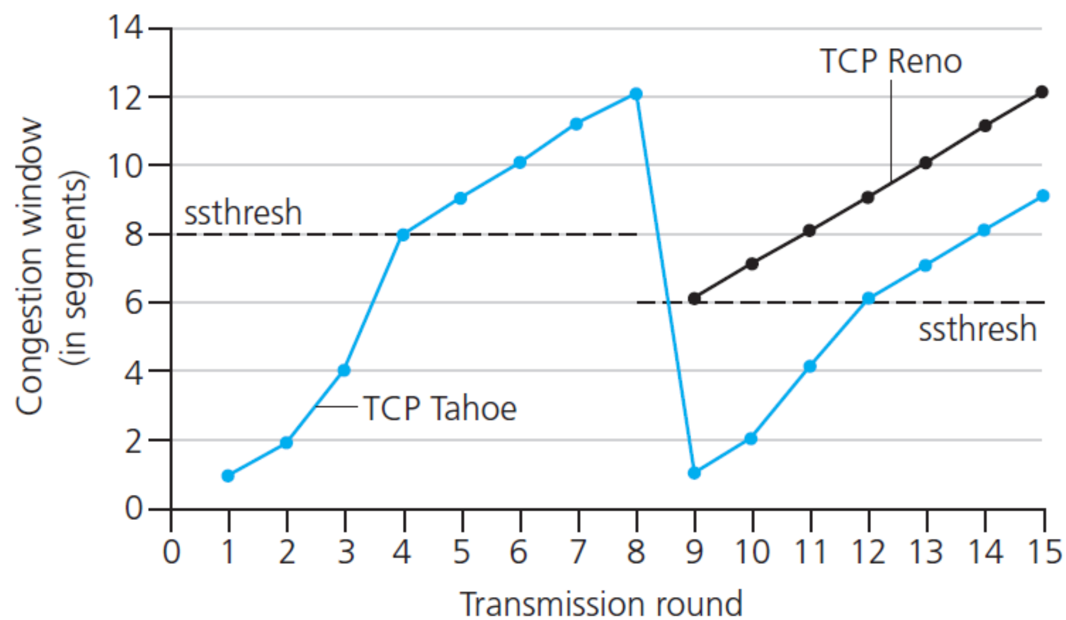
AIMD saw tooth behavior: probing for bandwidth



TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

A: when **cwnd** gets to 1/2 of its value before timeout.



Implementation:

- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

TCP Variants

- There are tons of them!

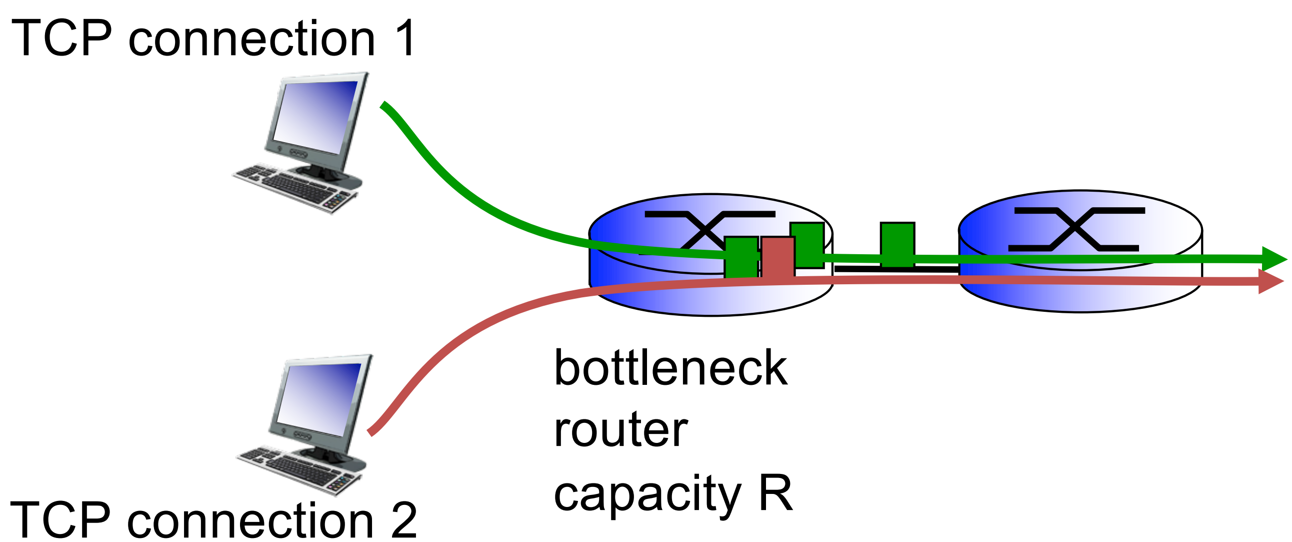
- Tahoe, Reno, New Reno, Vegas, Hybla, BIC, CUBIC, Westwood, Compound TCP, DCTCP, YeAH-TCP, …

- Each tweaks and adjusts the response to congestion.

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

TCP Fairness

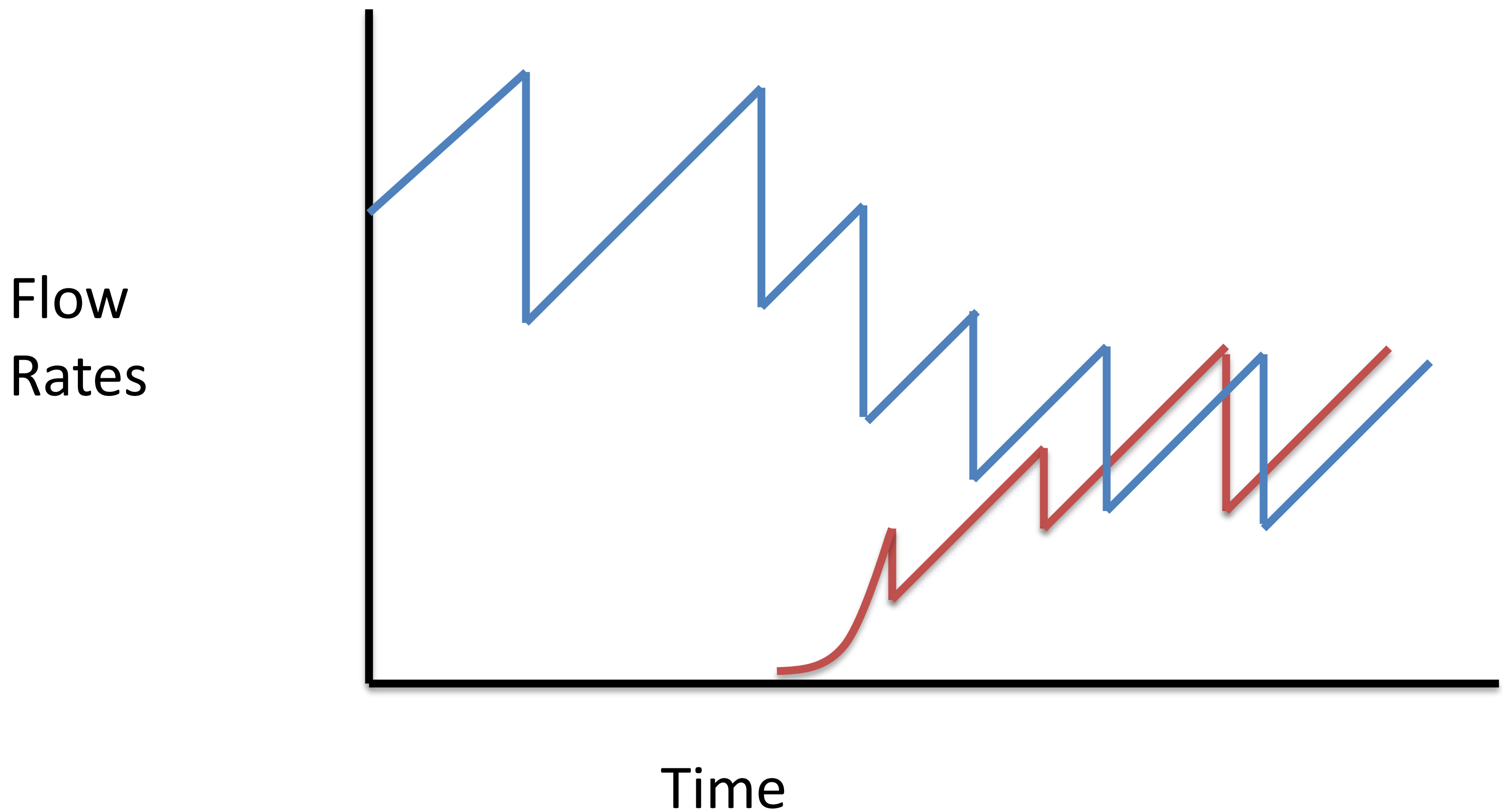
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



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

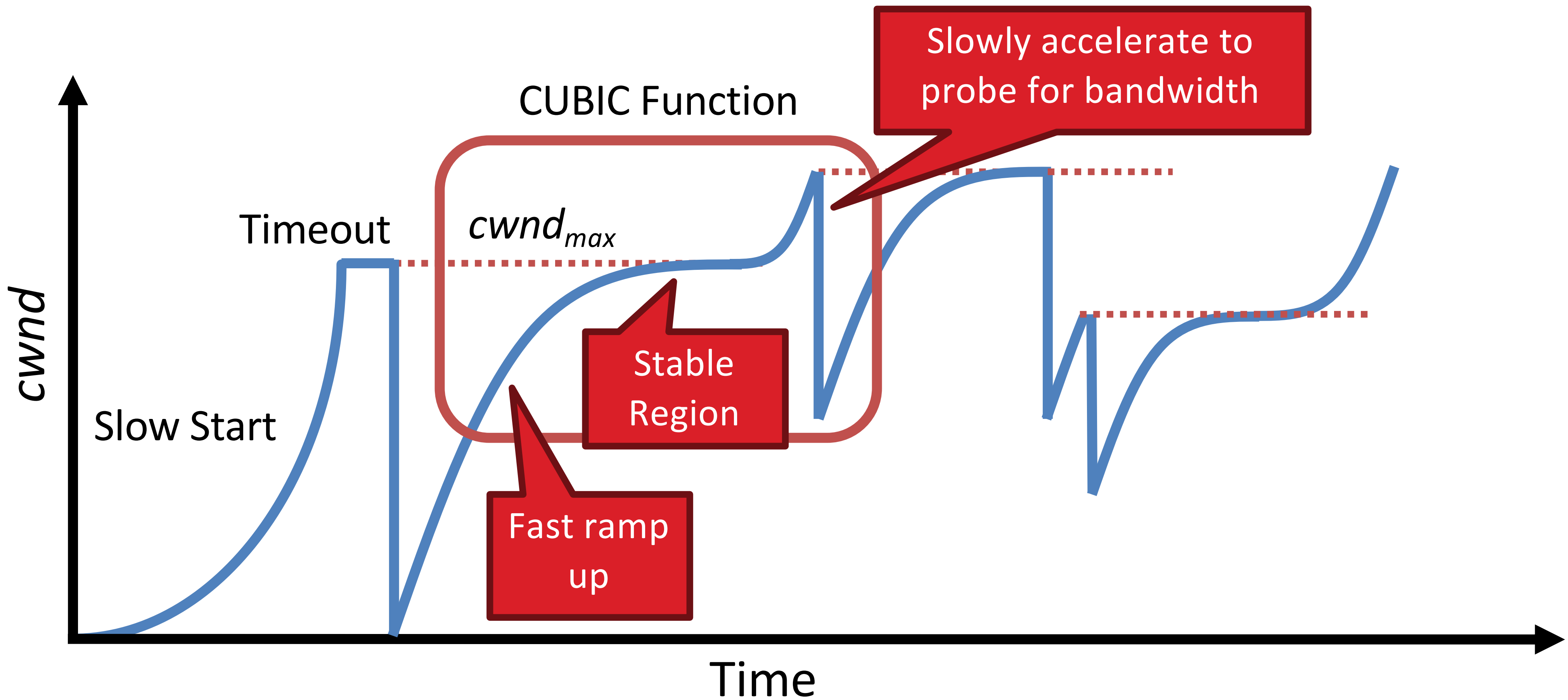
- min(rwnd, cwnd) determines window size for TCP segment pipelining (typically cwnd)

- AIMD: additive increase, multiplicative decrease

Additional Slides

(not assessable)

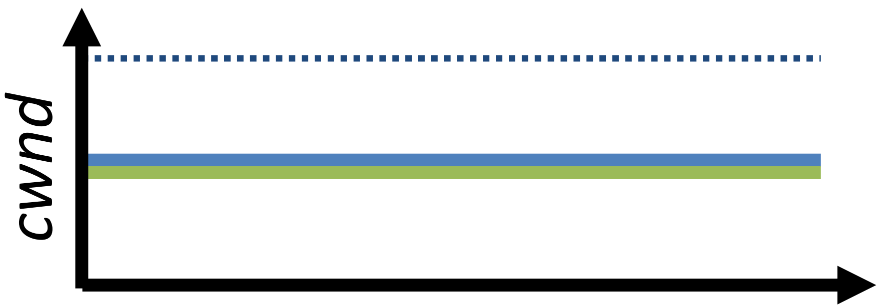
TCP CUBIC Example



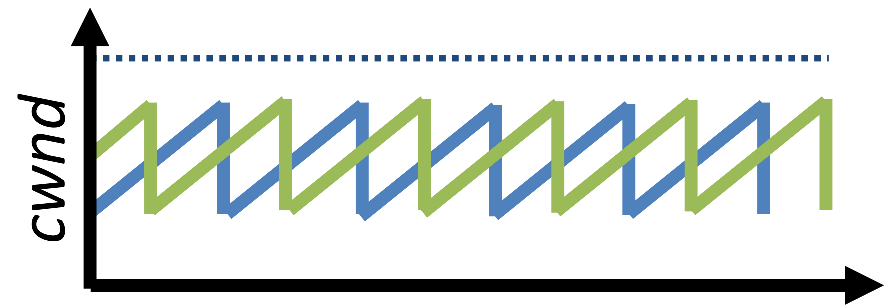
- Less wasted bandwidth due to fast ramp up
- Stable region and slow acceleration help maintain fairness
  - Fast ramp up is more aggressive than additive increase
  - To be fair to Tahoe/Reno, CUBIC needs to be less aggressive

Synchronization of Flows

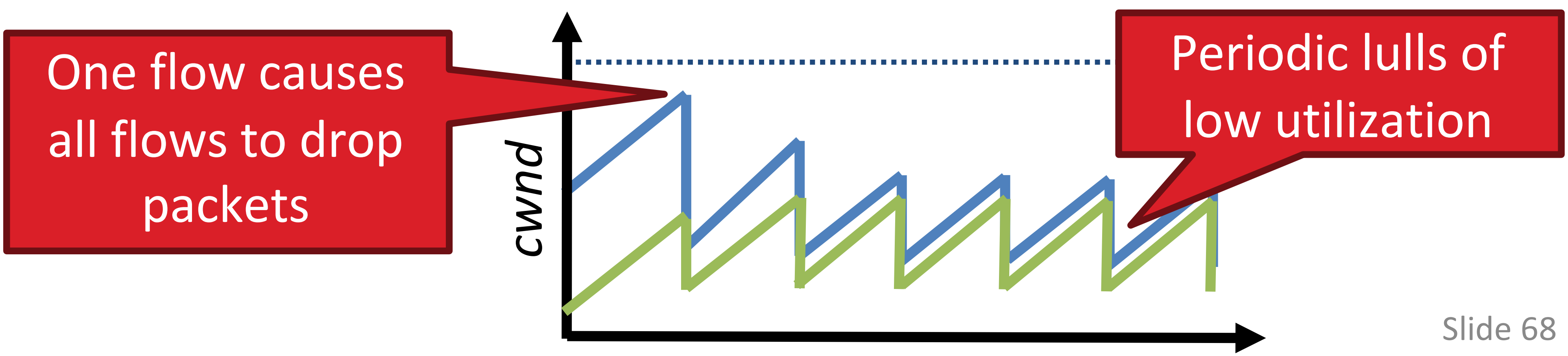
- Ideal bandwidth sharing



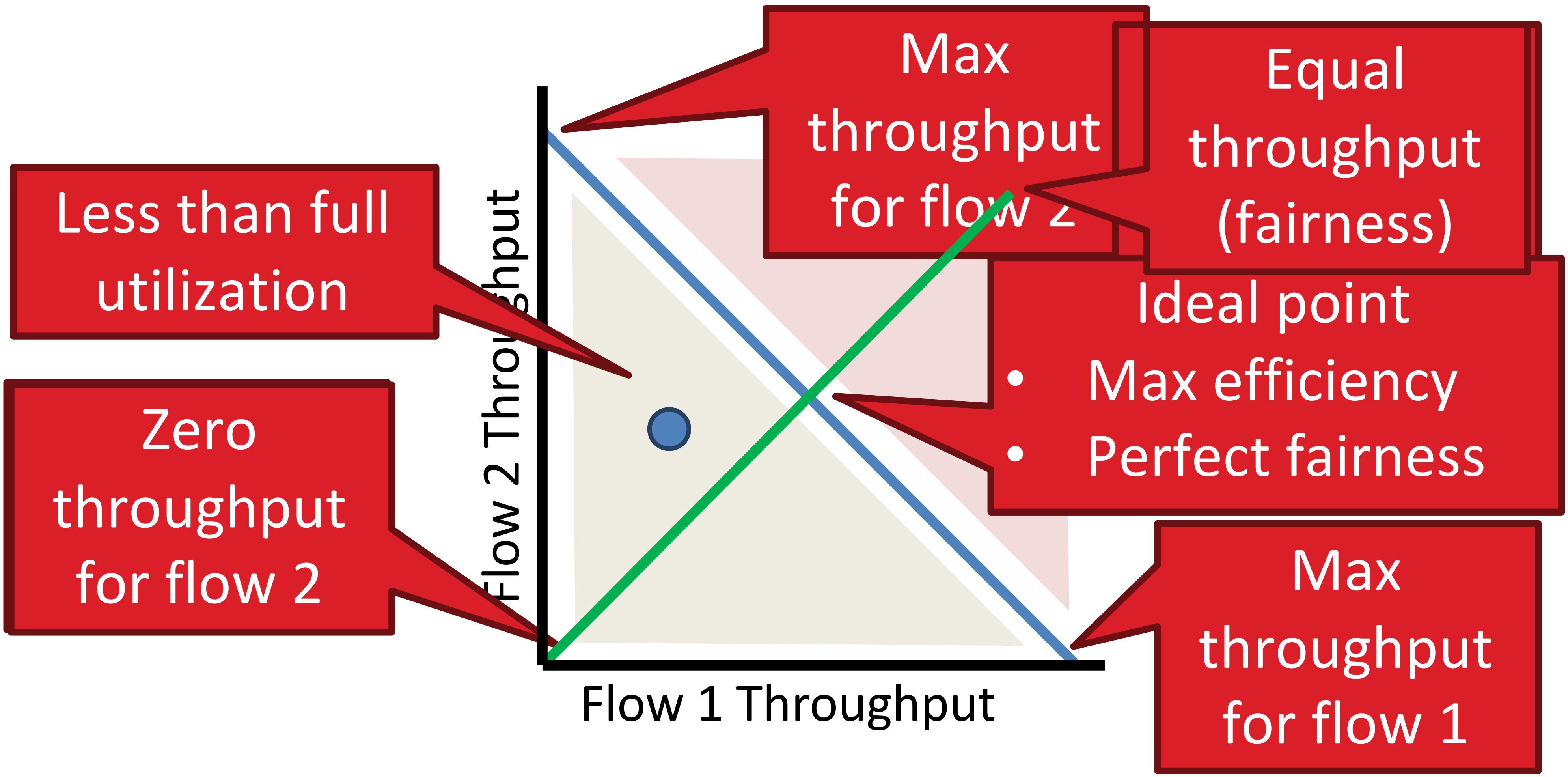
- Oscillating, but high overall utilization



- In reality, flows synchronize

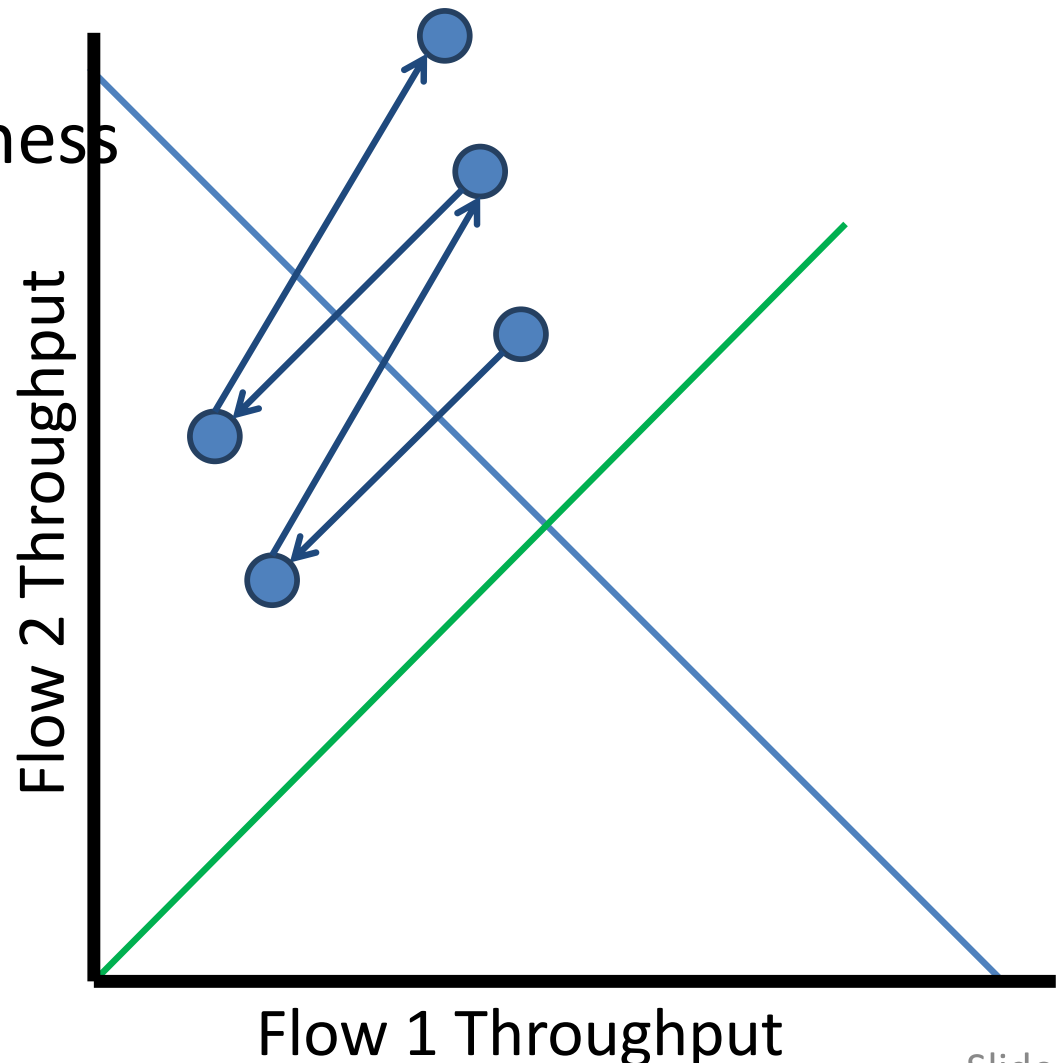


Utilization and Fairness



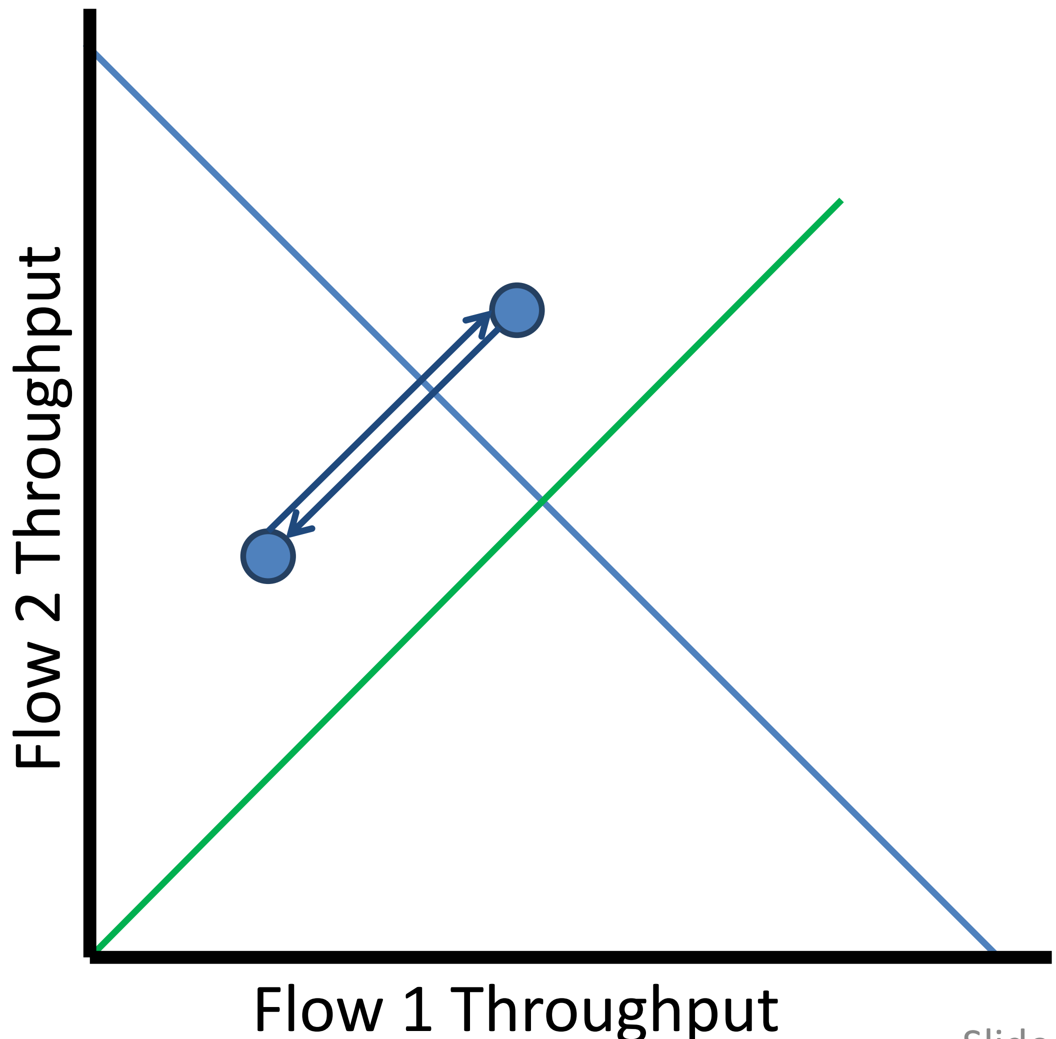
Multiplicative Increase, Additive Decrease

- Not stable!
- Veers away from fairness



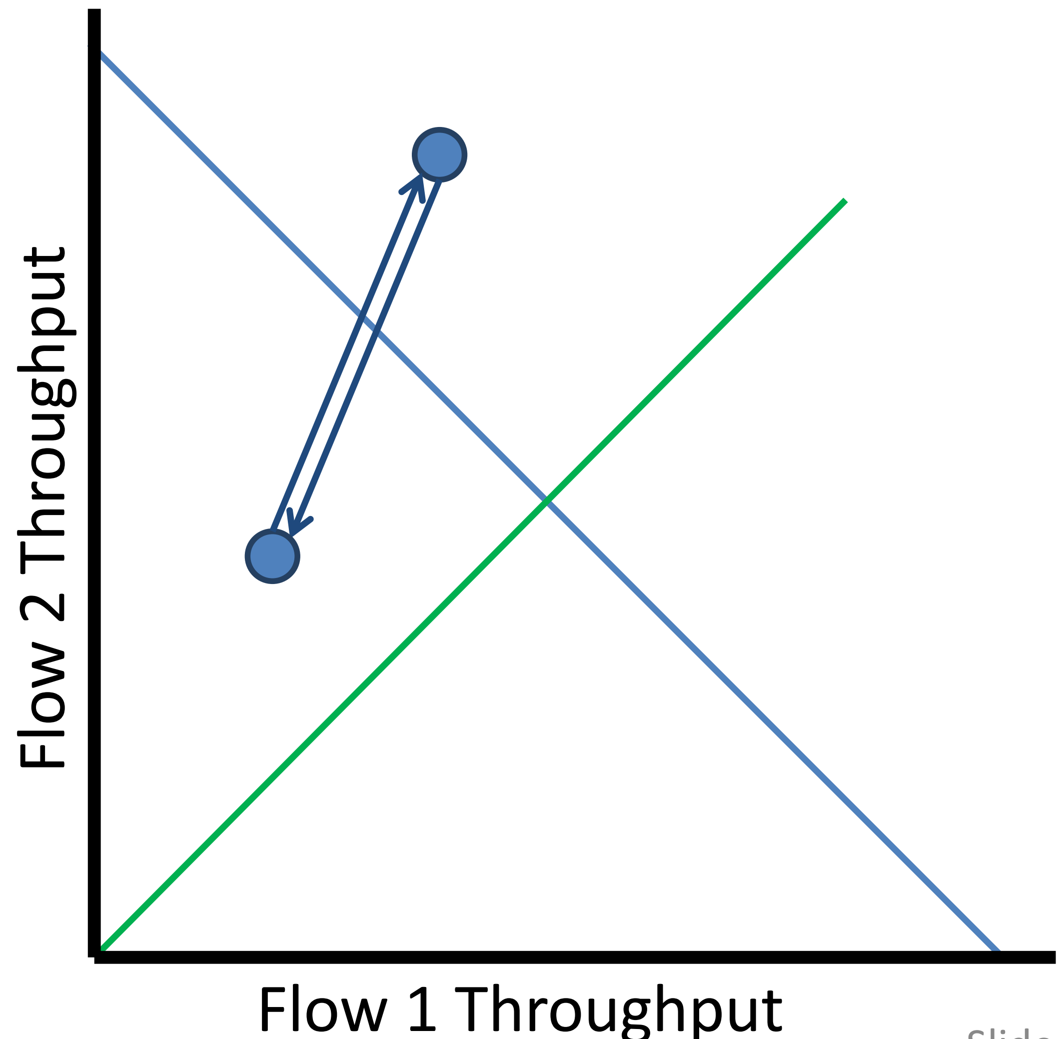
Additive Increase, Additive Decrease

- Stable
- But does not converge to fairness



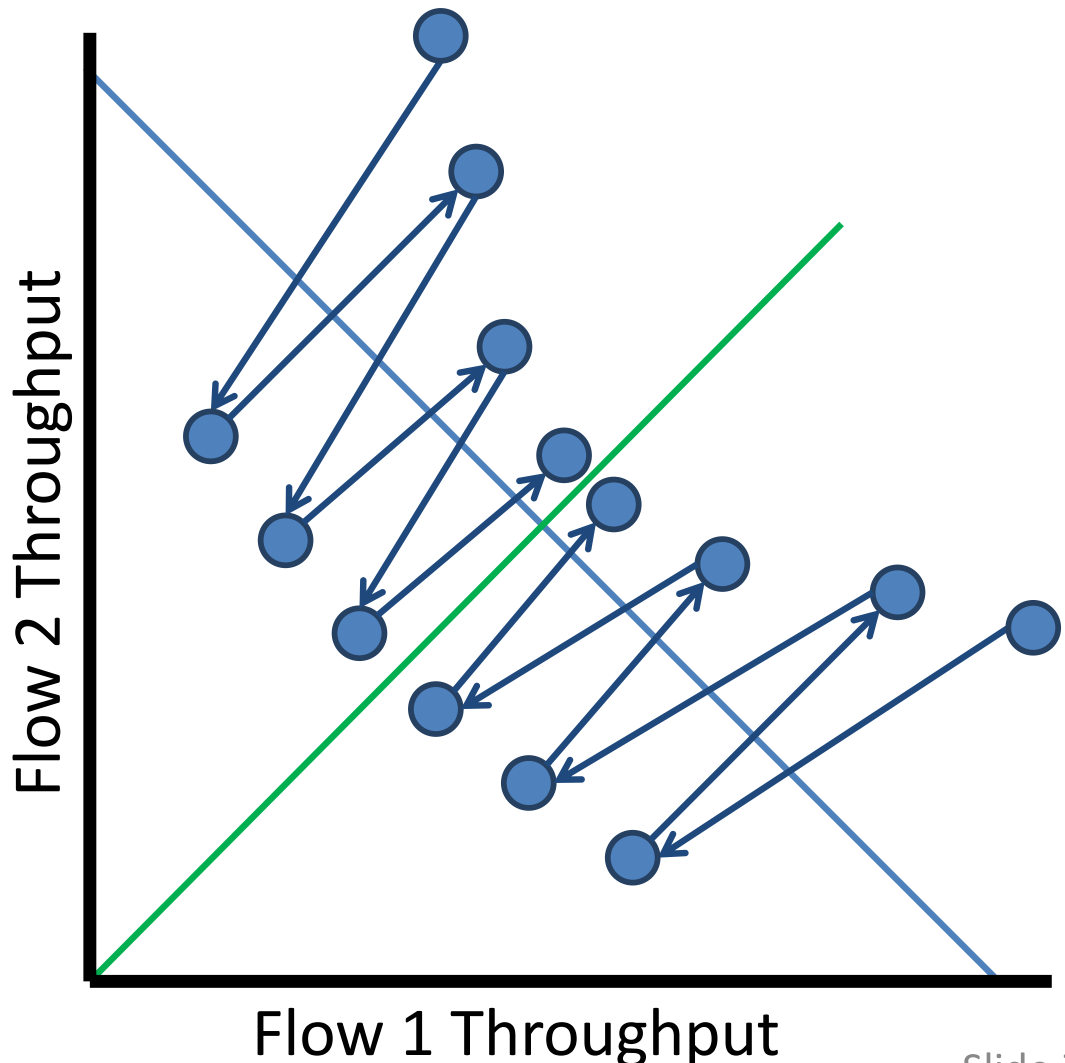
Multiplicative Increase, Multiplicative Decrease

- Stable
- But does not converge to fairness



Additive Increase, Multiplicative Decrease

- Converges to stable and fair cycle
- Symmetric around *y=x*



TCP: Big Picture