COMP 3331/9331: Computer Networks and Applications

Week 3

Congestion Control (Transport Layer)

Reading Guide: Chapter 3, Sections: 3.5-3.7

Transport Layer: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

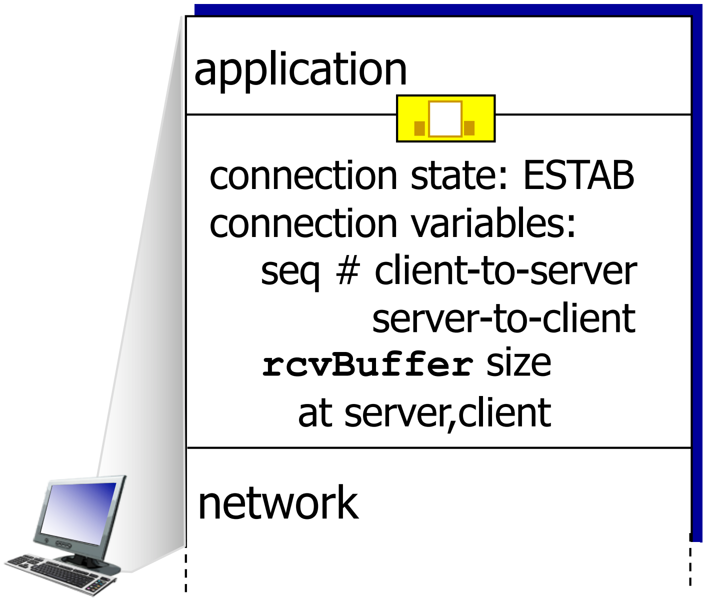
3.6 principles of congestion control

3.7 TCP congestion control

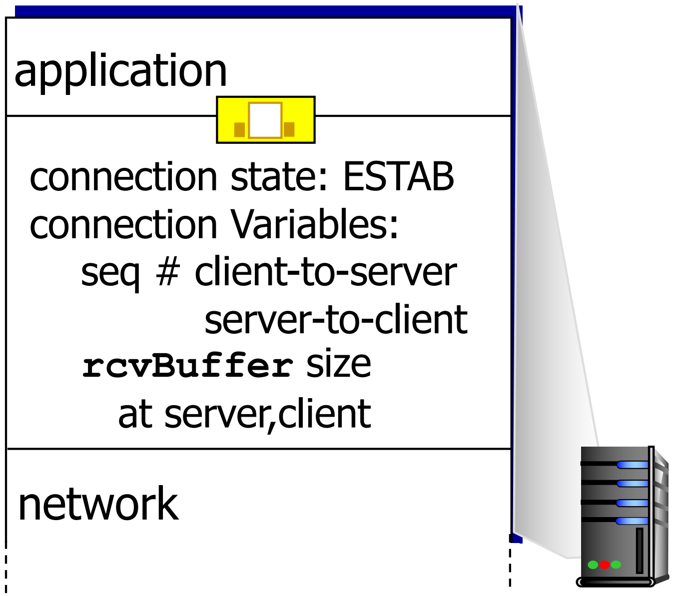
Connection Management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

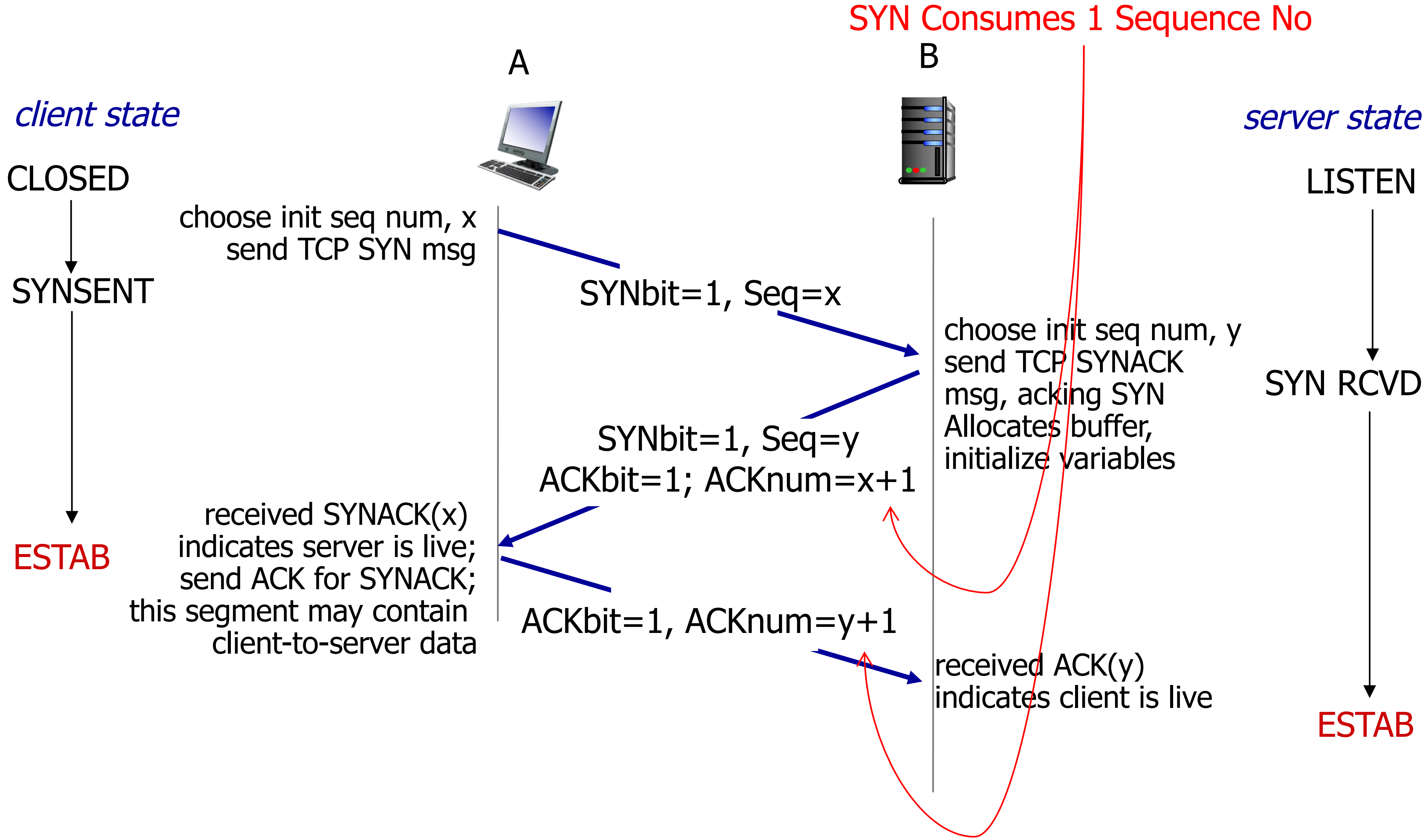



```
Socket clientSocket =
  newSocket("hostname","port
  number");
```

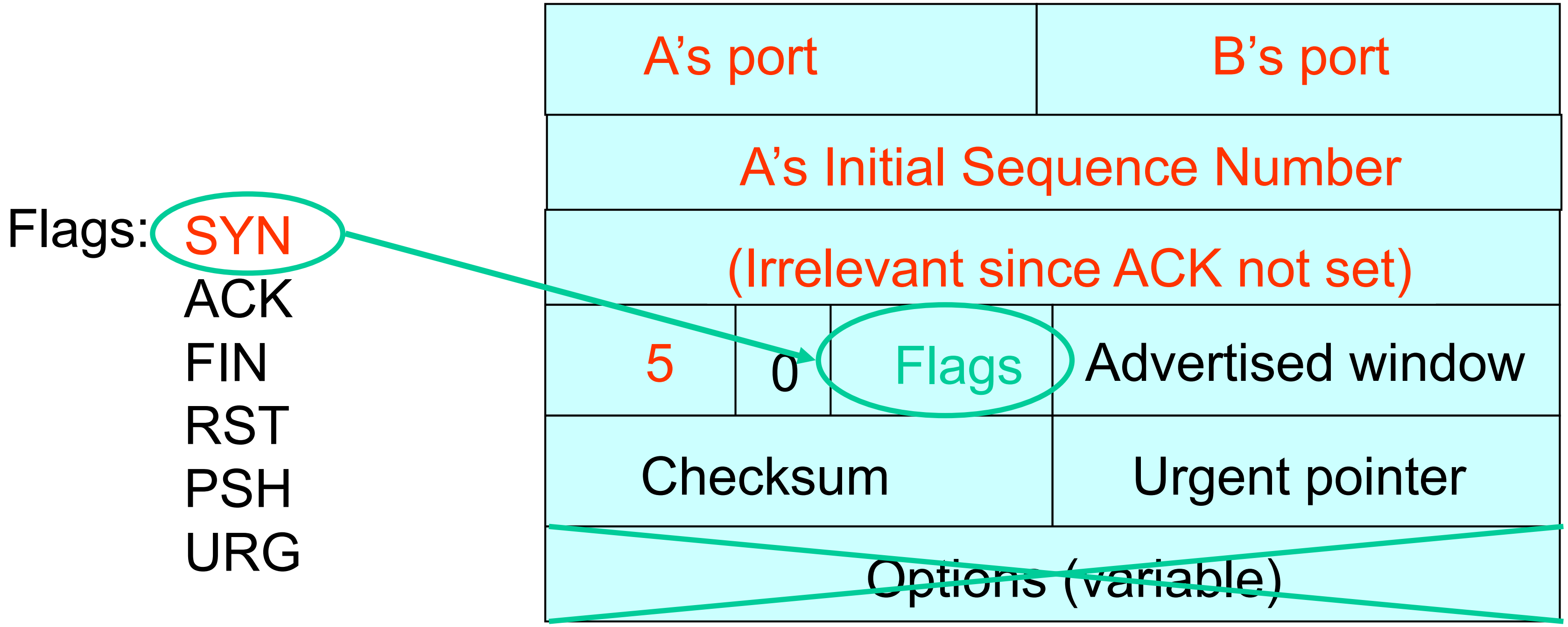



```
Socket connectionSocket =
  welcomeSocket.accept();
```

TCP 3-way handshake

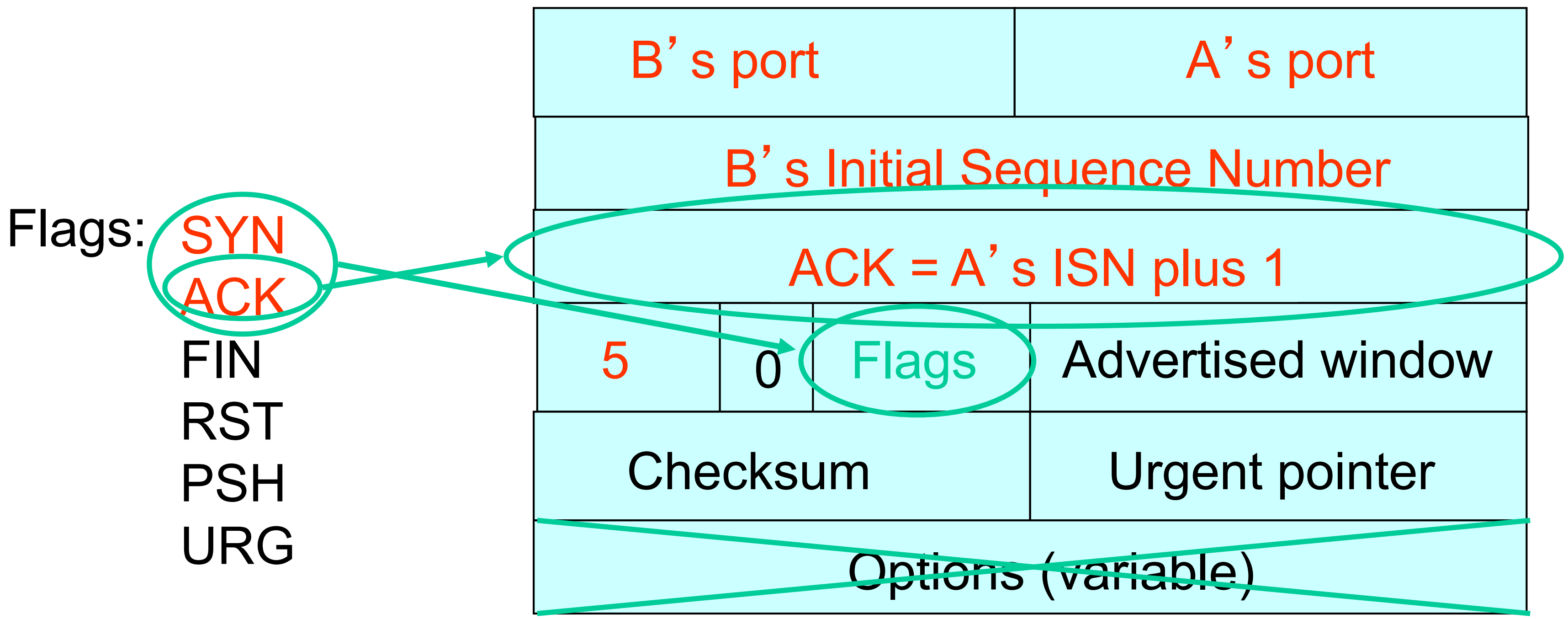


Step 1: A’s Initial SYN Packet



A tells B it wants to open a connection…

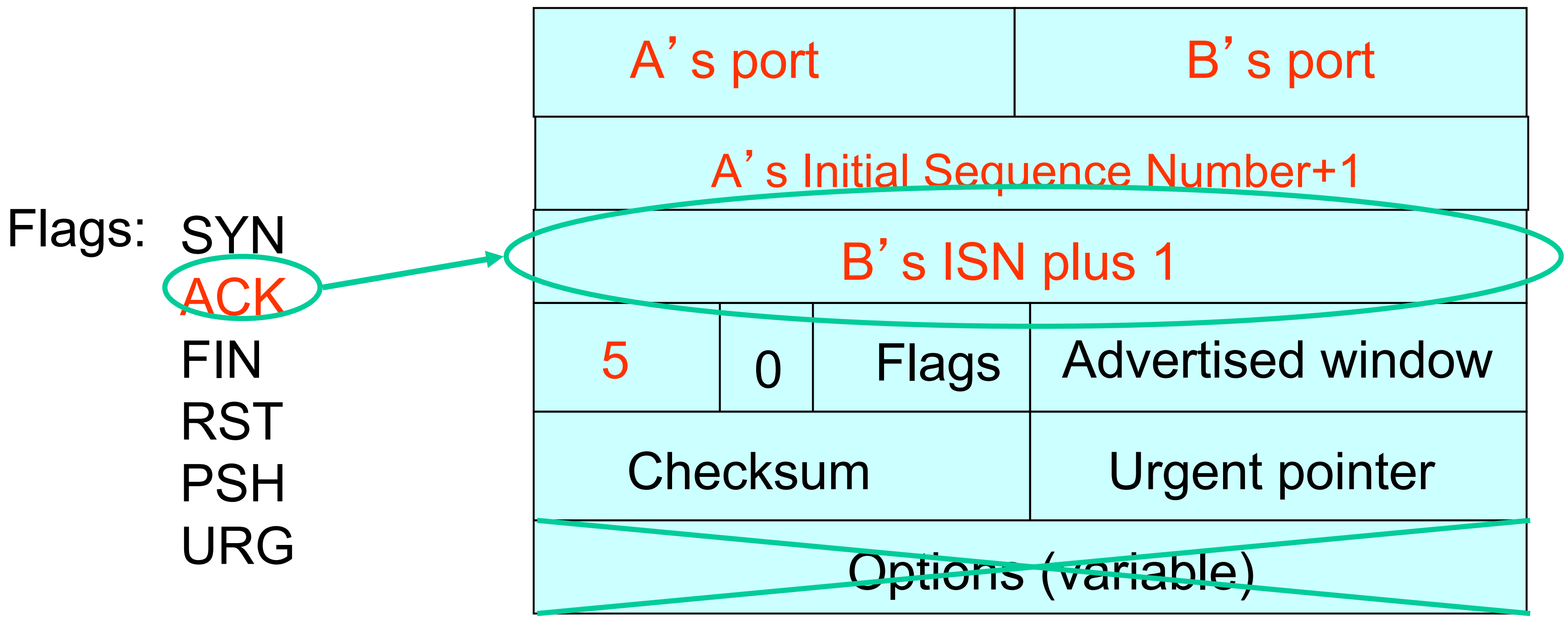
Step 2: B’s SYN-ACK Packet



B tells A it accepts, and is ready to hear the next byte…

… upon receiving this packet, A can start sending data

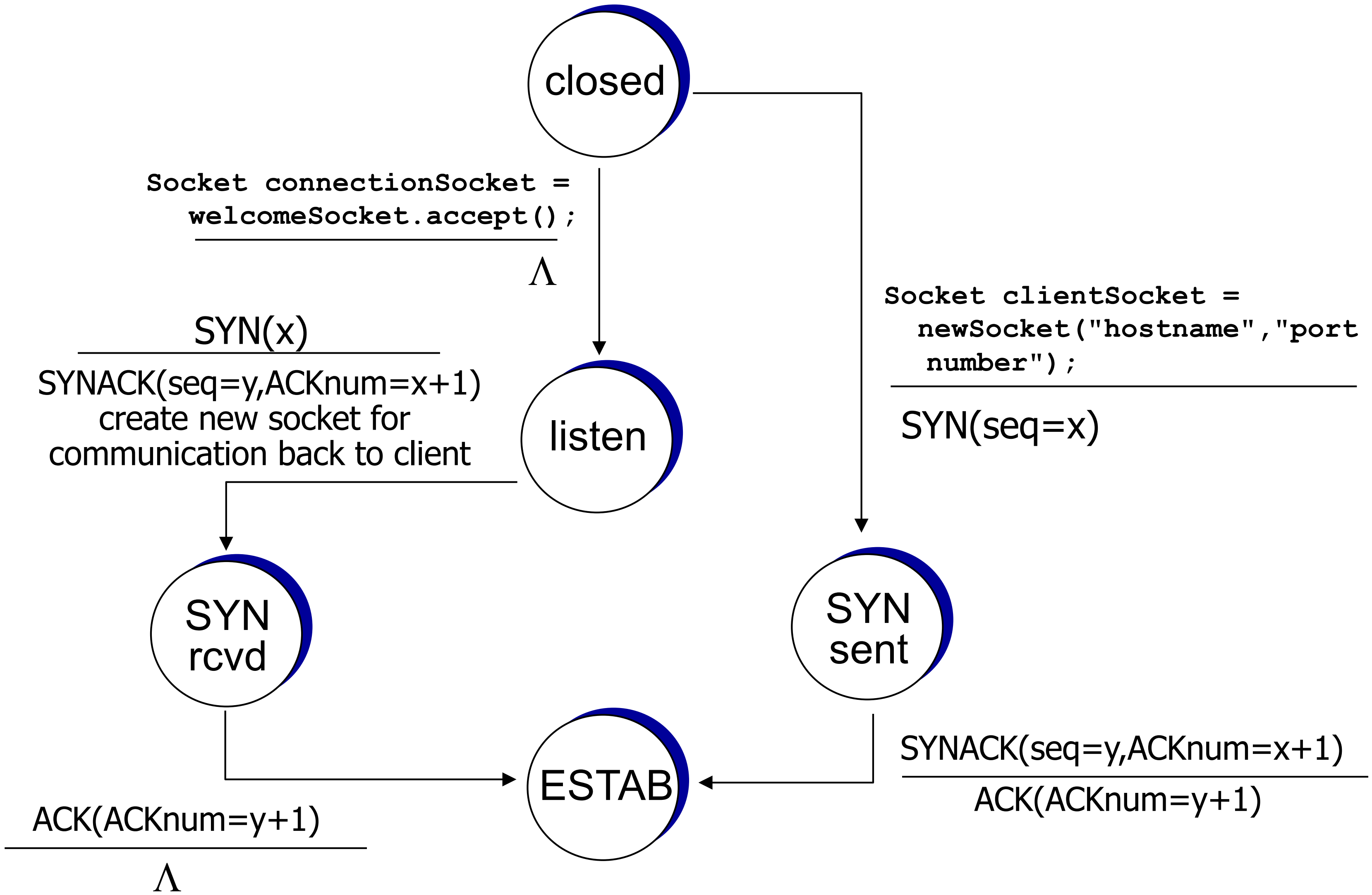
Step 3: A’s ACK of the SYN-ACK



A tells B it’s likewise okay to start sending

… upon receiving this packet, B can start sending data

TCP 3-way handshake: FSM



What if the SYN Packet Gets Lost?

- Suppose the SYN packet gets lost
  - Packet is lost inside the network, or:
  - Server discards the packet (e.g., it's too busy)
- Eventually, no SYN-ACK arrives
  - Sender sets a timer and waits for the SYN-ACK
  - … and retransmits the SYN if needed
- How should the TCP sender set the timer?
  - Sender has no idea how far away the receiver is
  - Hard to guess a reasonable length of time to wait
  - **SHOULD** (RFCs 1122,2988) use default of 3 second, RFC 6298 use default of 1 second

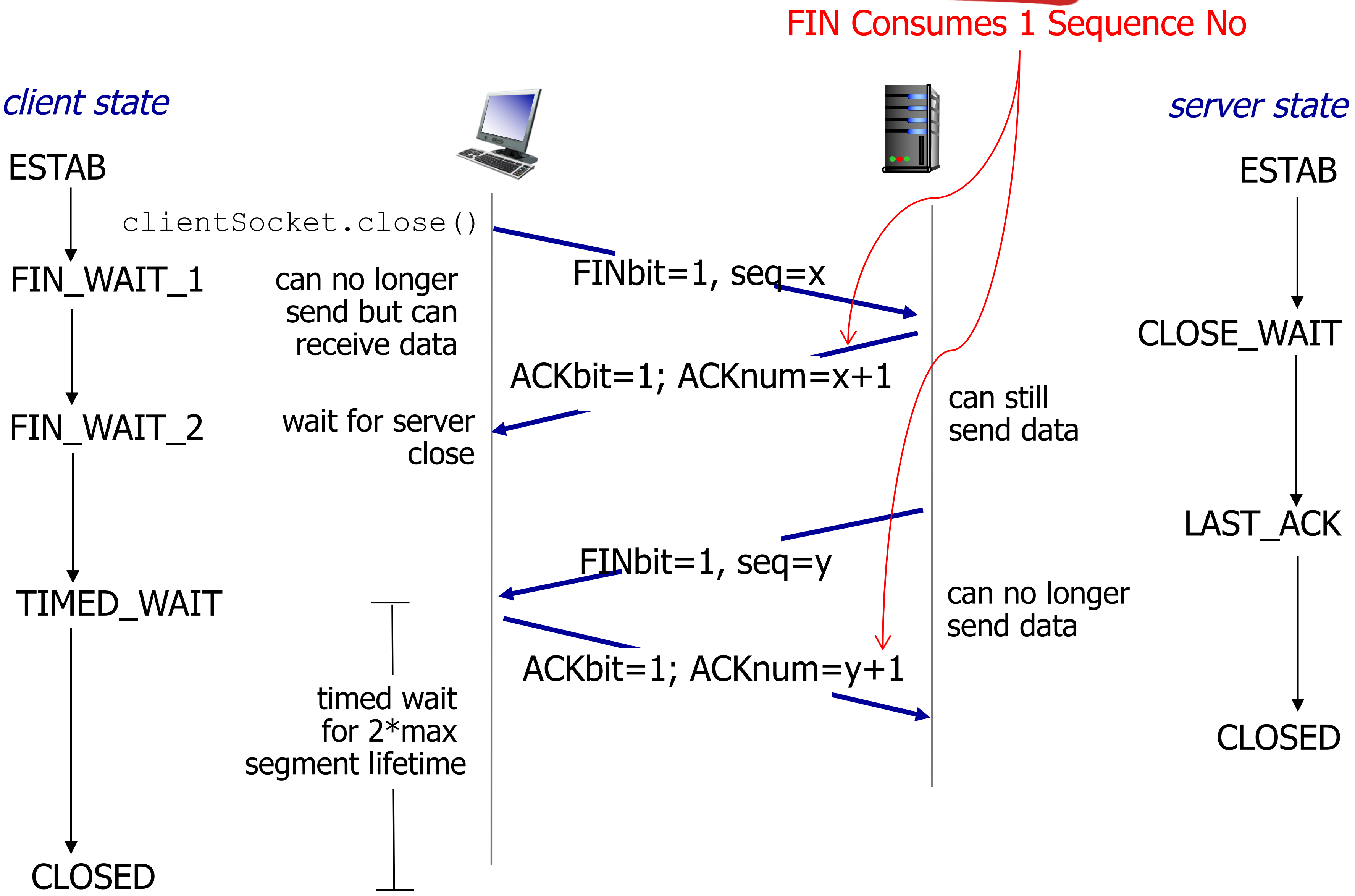
SYN Loss and Web Downloads

- User clicks on a hypertext link
  - Browser creates a socket and does a "connect"
  - The "connect" triggers the OS to transmit a SYN
- If the SYN is lost…
  - 1-3 seconds of delay: can be very long
  - User may become impatient
  - … and click the hyperlink again, or click "reload"
- User triggers an "abort" of the "connect"
  - Browser creates a new socket and another "connect"
  - Essentially, forces a faster send of a new SYN packet!
  - Sometimes very effective, and the page comes quickly

TCP: closing a connection

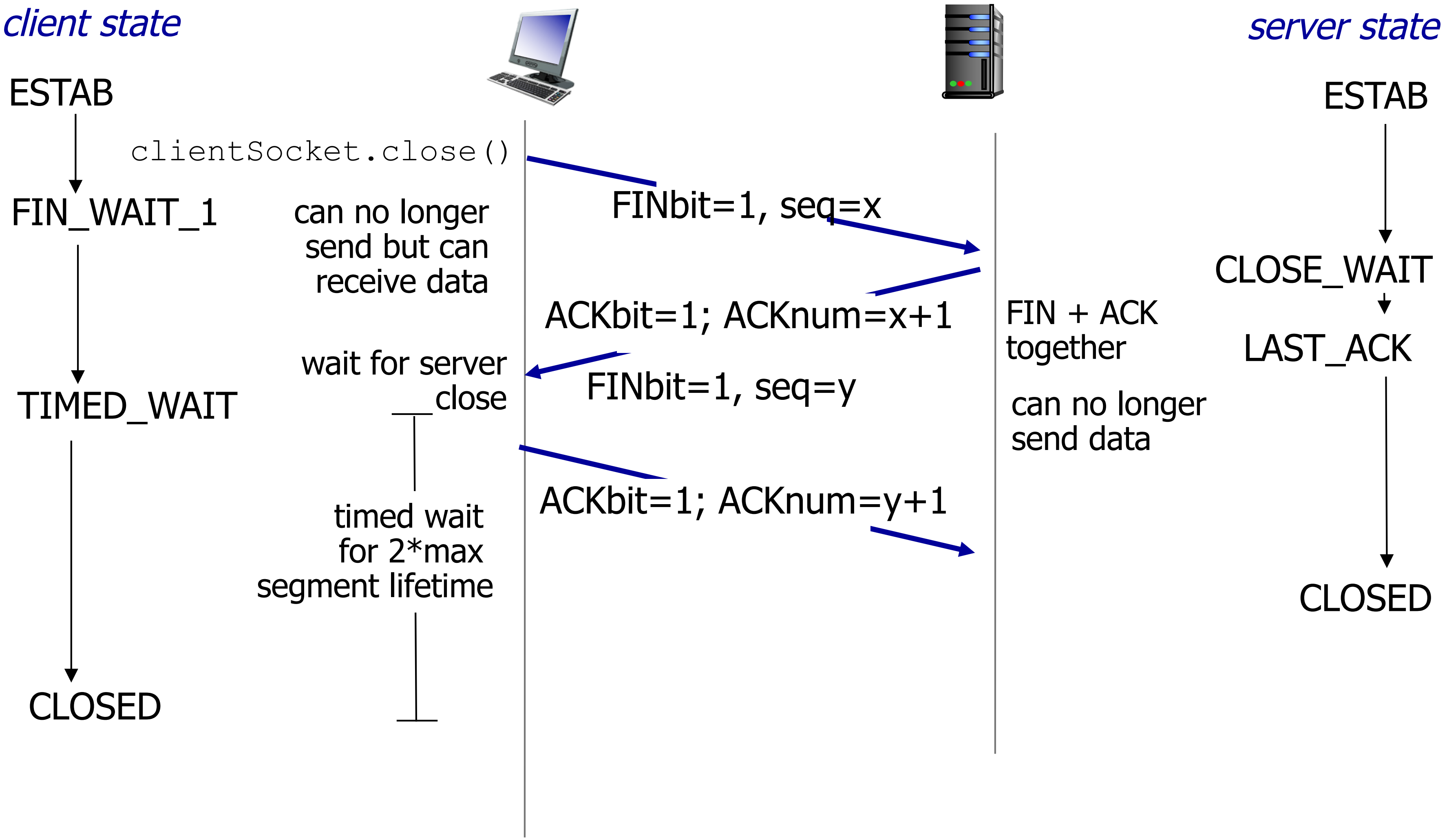
- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

Normal Termination, One at a Time

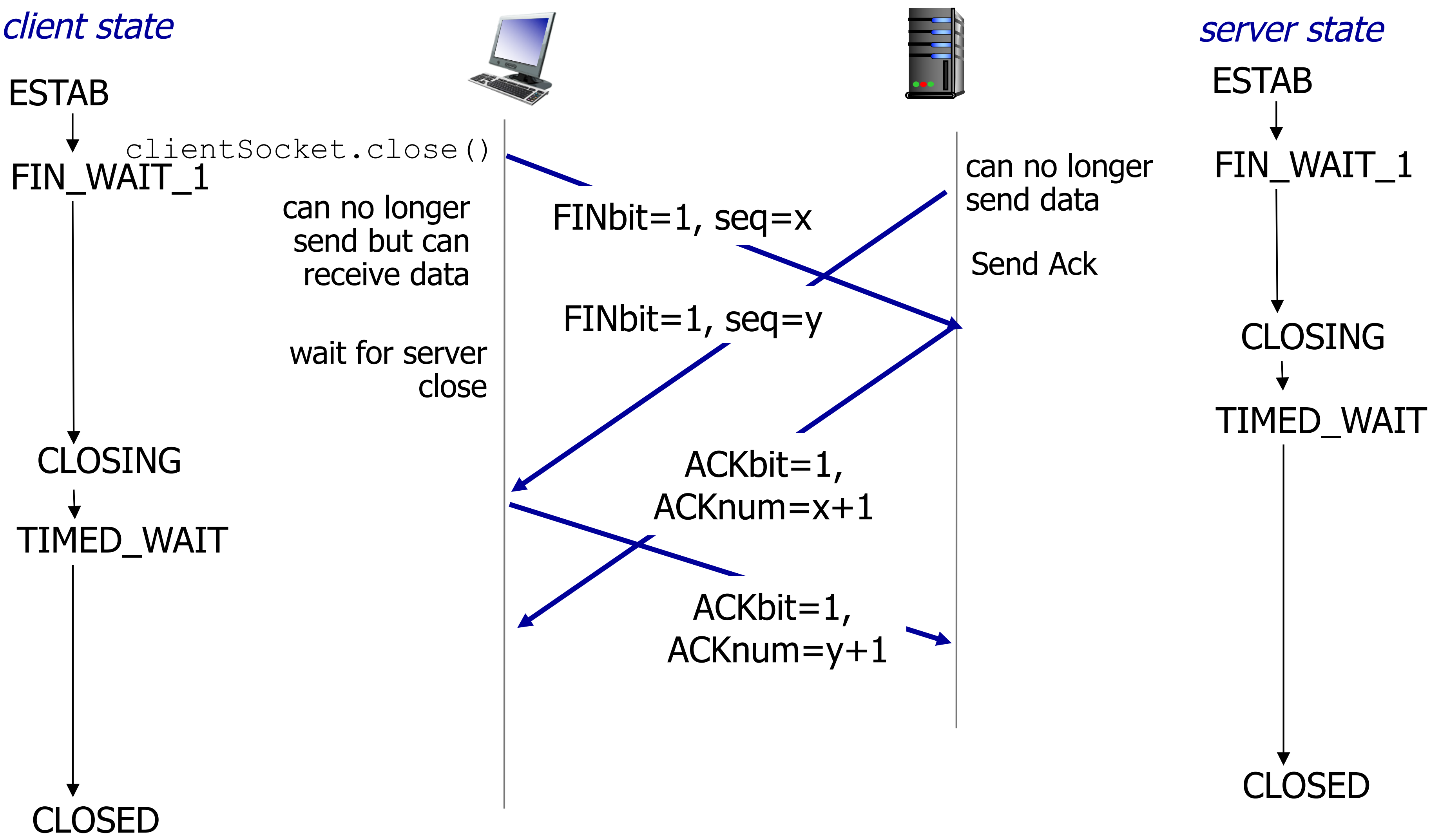


TIMED_WAIT: Can retransmit ACK if last ACK is lost

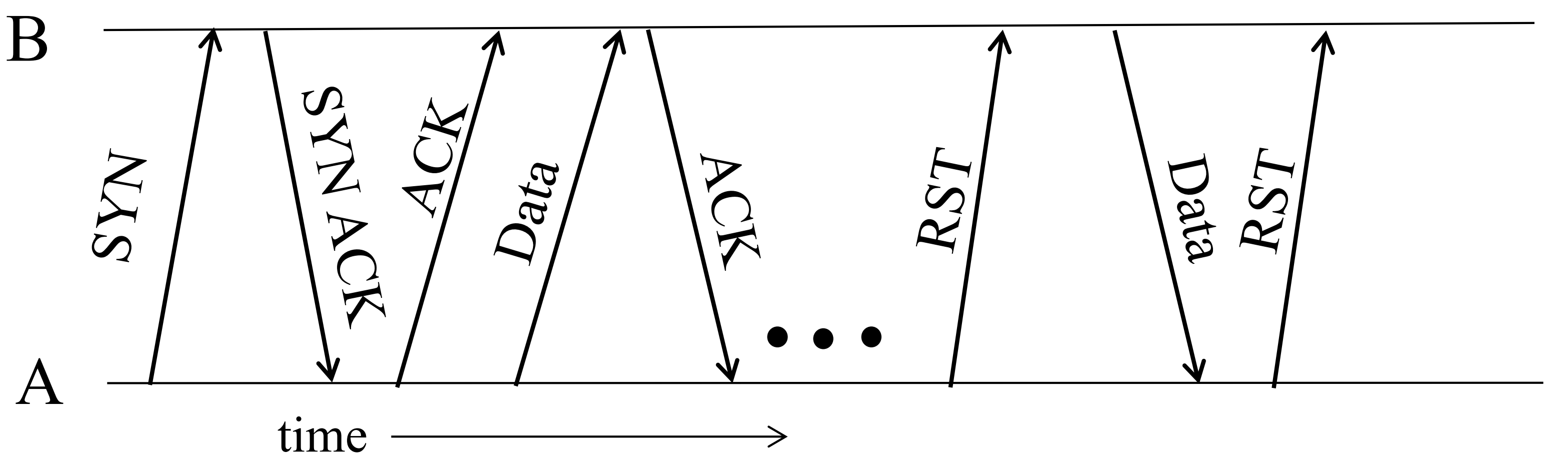
Normal Termination, Both Together



Simultaneous Closure

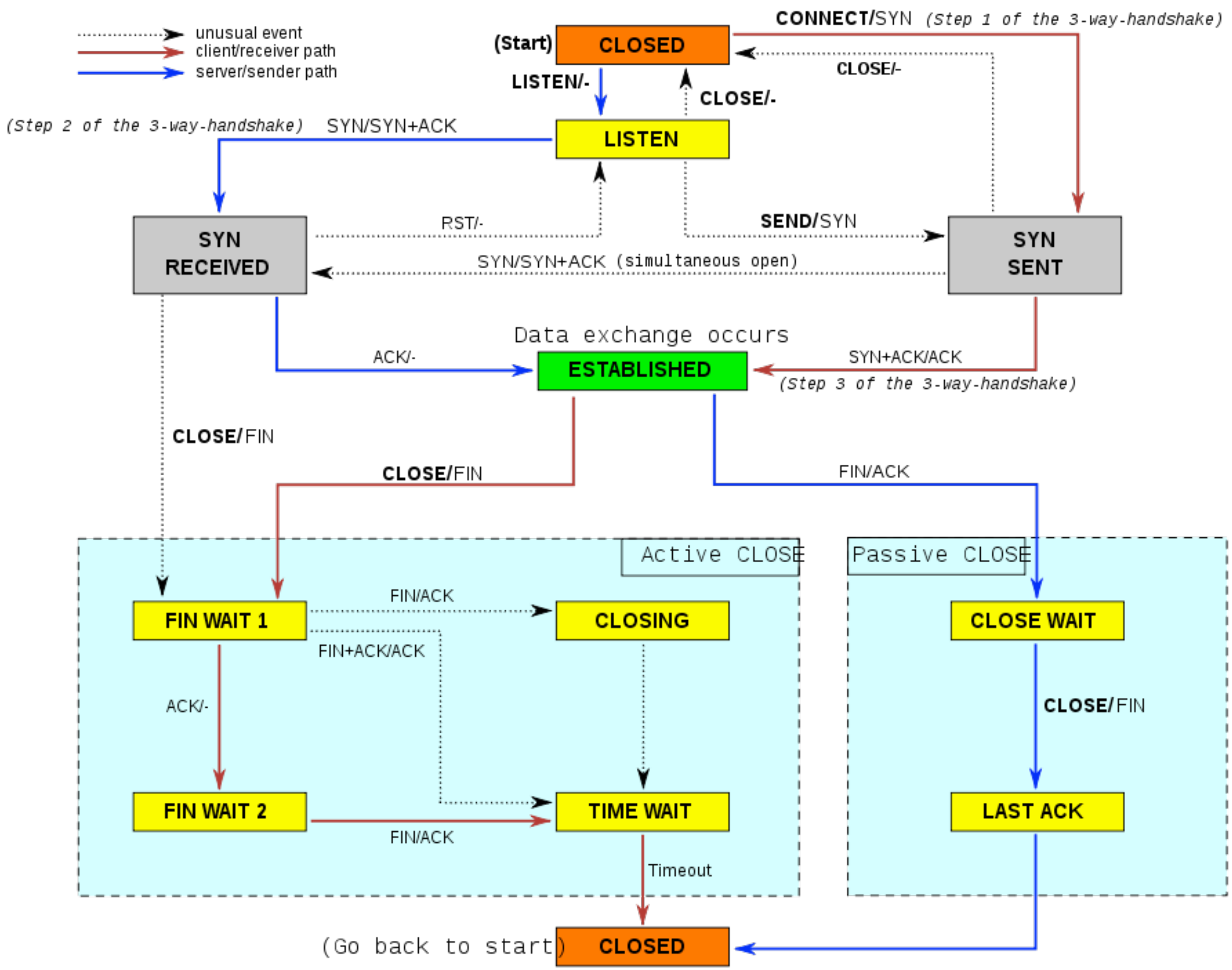


Abrupt Termination



- A sends a RESET (**RST**) to B
  - E.g., because application process on A crashed
- That's it
  - B does not ack the **RST**
  - Thus, **RST** is not delivered reliably
  - And: any data in flight is lost
  - But: if B sends anything more, will elicit another **RST**

TCP Finite State Machine



TCP SYN Attack (SYN flooding)

- Miscreant creates a fake SYN packet
  - Destination is IP address of victim host (usually some server)
  - Source is some spoofed IP address
- Victim host on receiving creates a TCP connection state i.e allocates buffers, creates variables, etc and sends SYN ACK to the spoofed address (half-open connection)
- ACK never comes back
- After a timeout connection state is freed
- However for this duration the connection state is unnecessarily created
- Further miscreant sends large number of fake SYNs
  - Can easily overwhelm the victim
- Solutions:
  - Increase size of connection queue
  - Decrease timeout wait for the 3-way handshake
  - Firewalls: list of known bad source IP addresses
  - TCP SYN Cookies (explained on next slide)

TCP SYN Cookie

- On receipt of SYN, server does not create connection state
- It creates an initial sequence number (*init_seq*) that is a hash of source & dest IP address and port number of SYN packet (secret key used for hash)
  - Replies back with SYN ACK containing *init_seq*
  - Server does not need to store this sequence number
- If original SYN is genuine, an ACK will come back
  - Same hash function run on the same header fields to get the initial sequence number (*init_seq*)
  - Checks if the ACK is equal to (*init_seq*+1)
  - Only create connection state if above is true
- If fake SYN, no harm done since no state was created

http://etherealmind.com/tcp-syn-cookies-ddos-defence/

Transport Layer: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

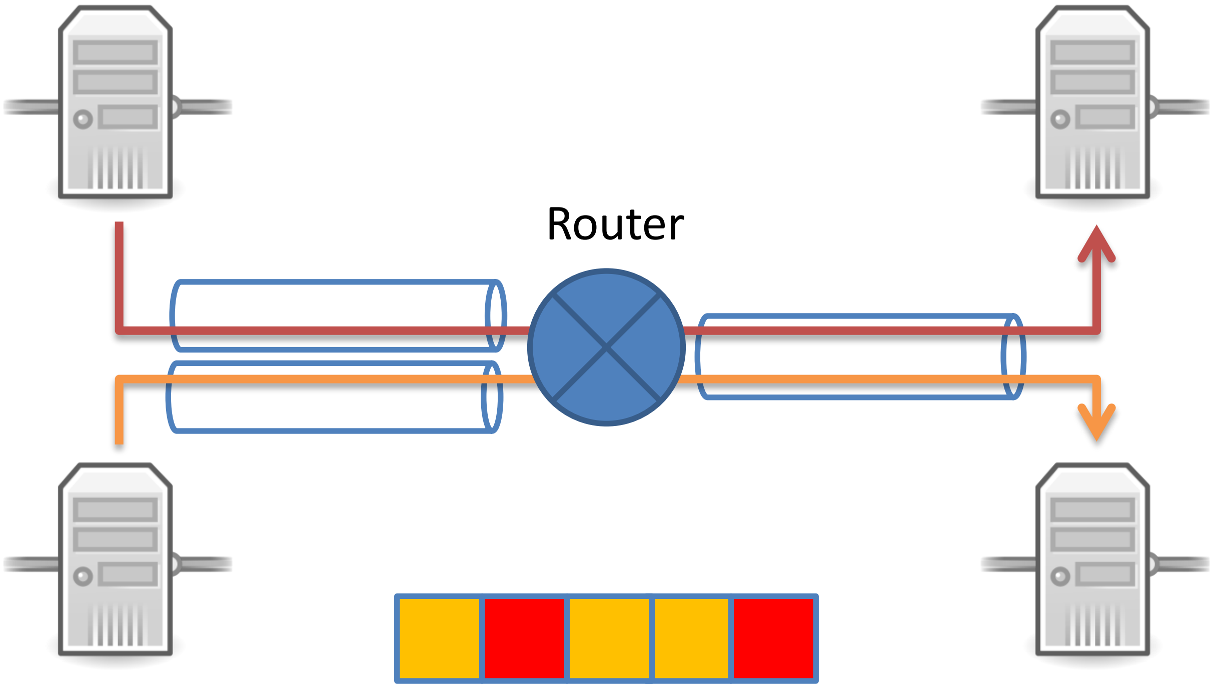
3.7 TCP congestion control

Principles of congestion control

congestion:

- informally: “too many sources sending too much data too fast for *network* to handle”
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

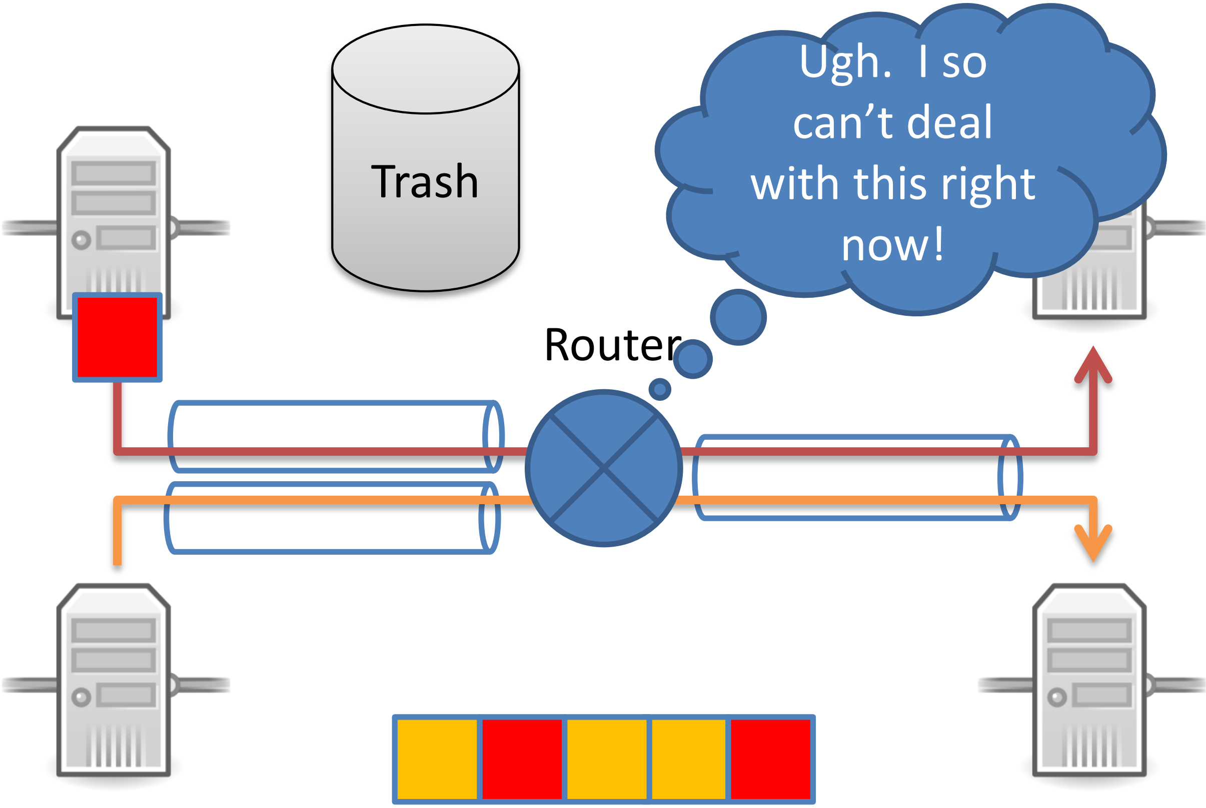
Congestion



Router's buffer.

Incoming rate is faster than outgoing link can support.

Congestion



Router's buffer.

Incoming rate is faster than outgoing link can support.

Quiz: What's the worst that can happen?

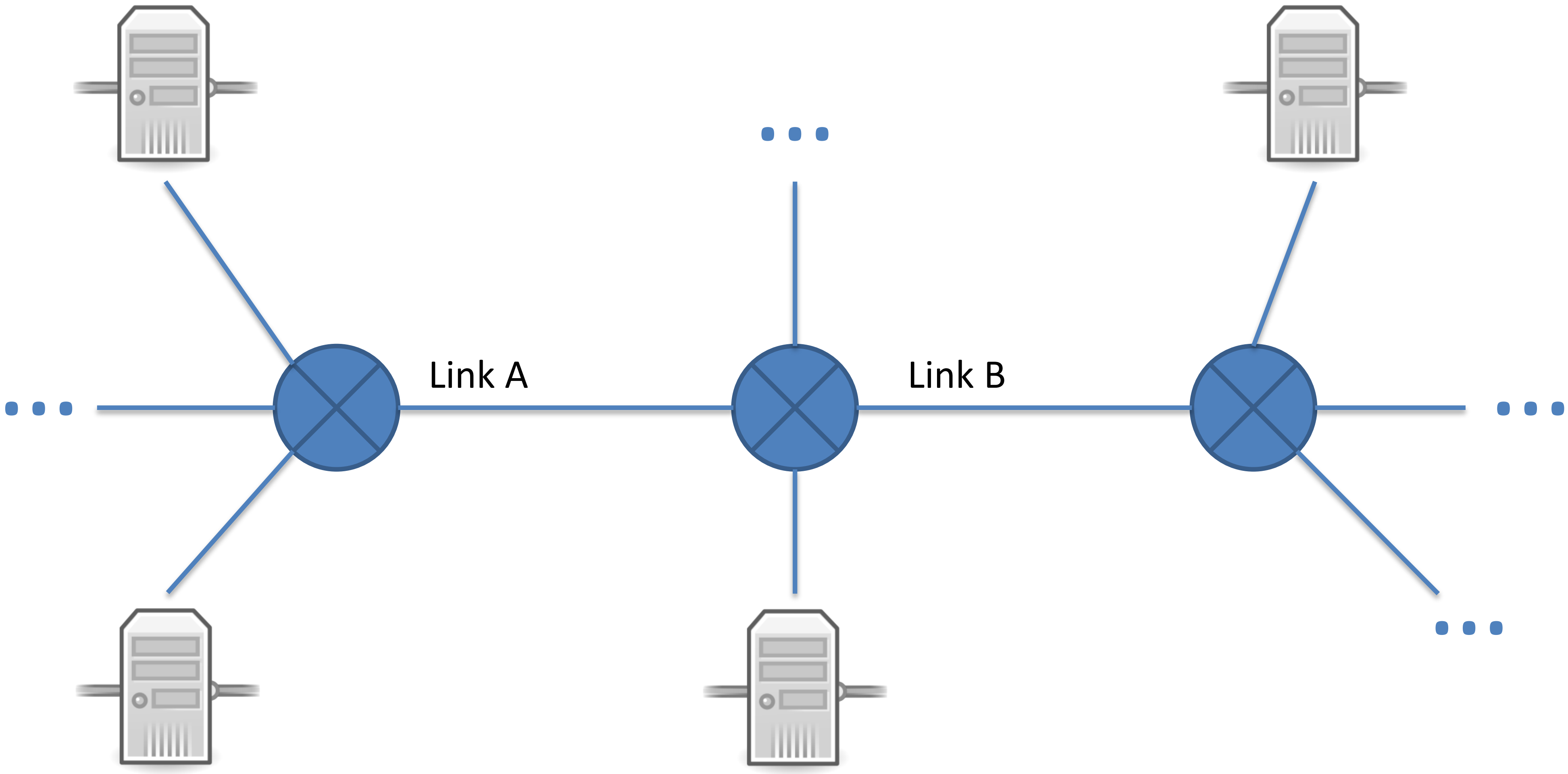
A: This is no problem. Senders just keep transmitting, and it'll all work out.

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

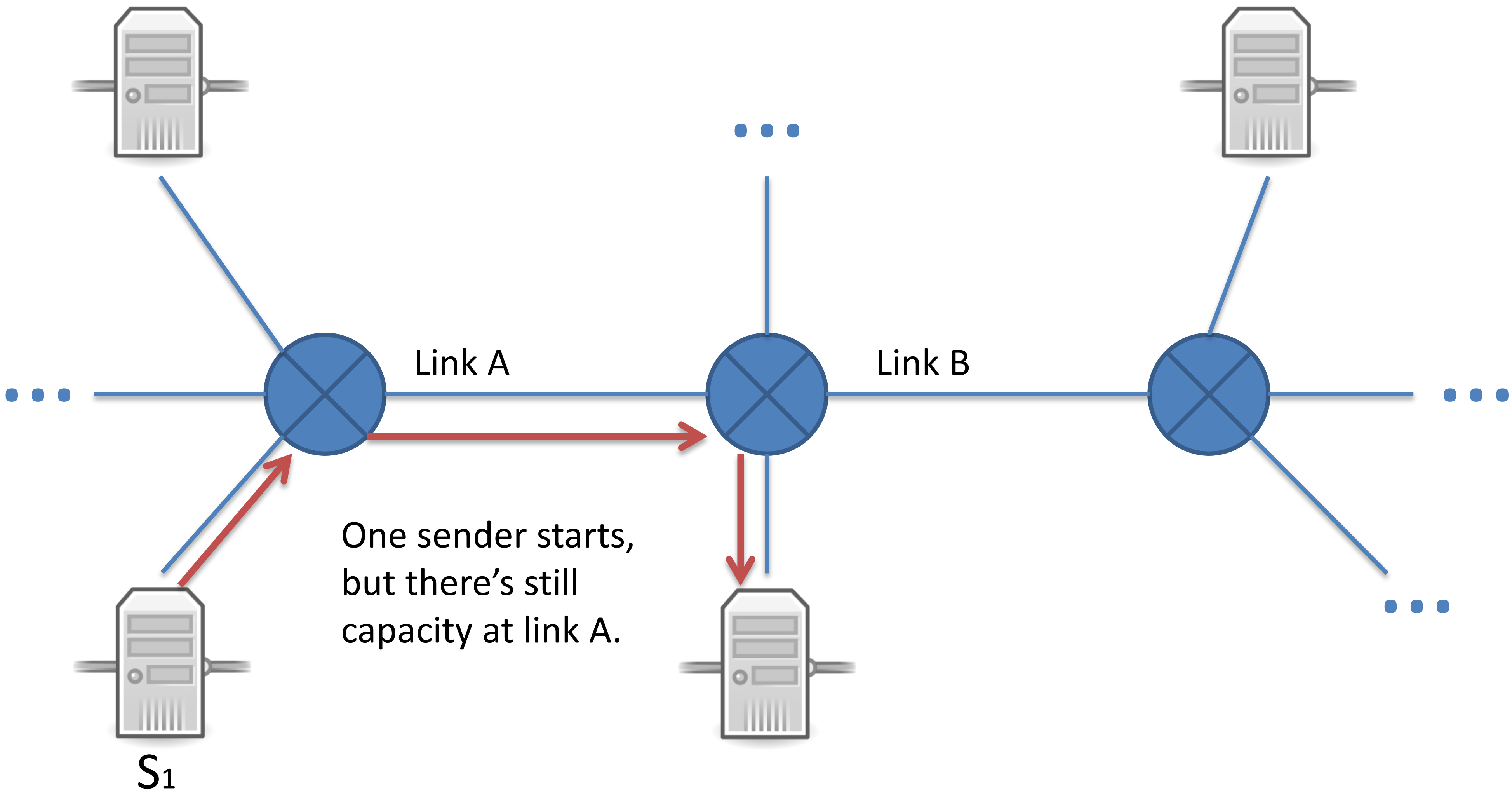
C: Retransmissions will become very frequent, causing a serious loss of efficiency

D: The network will become completely unusable

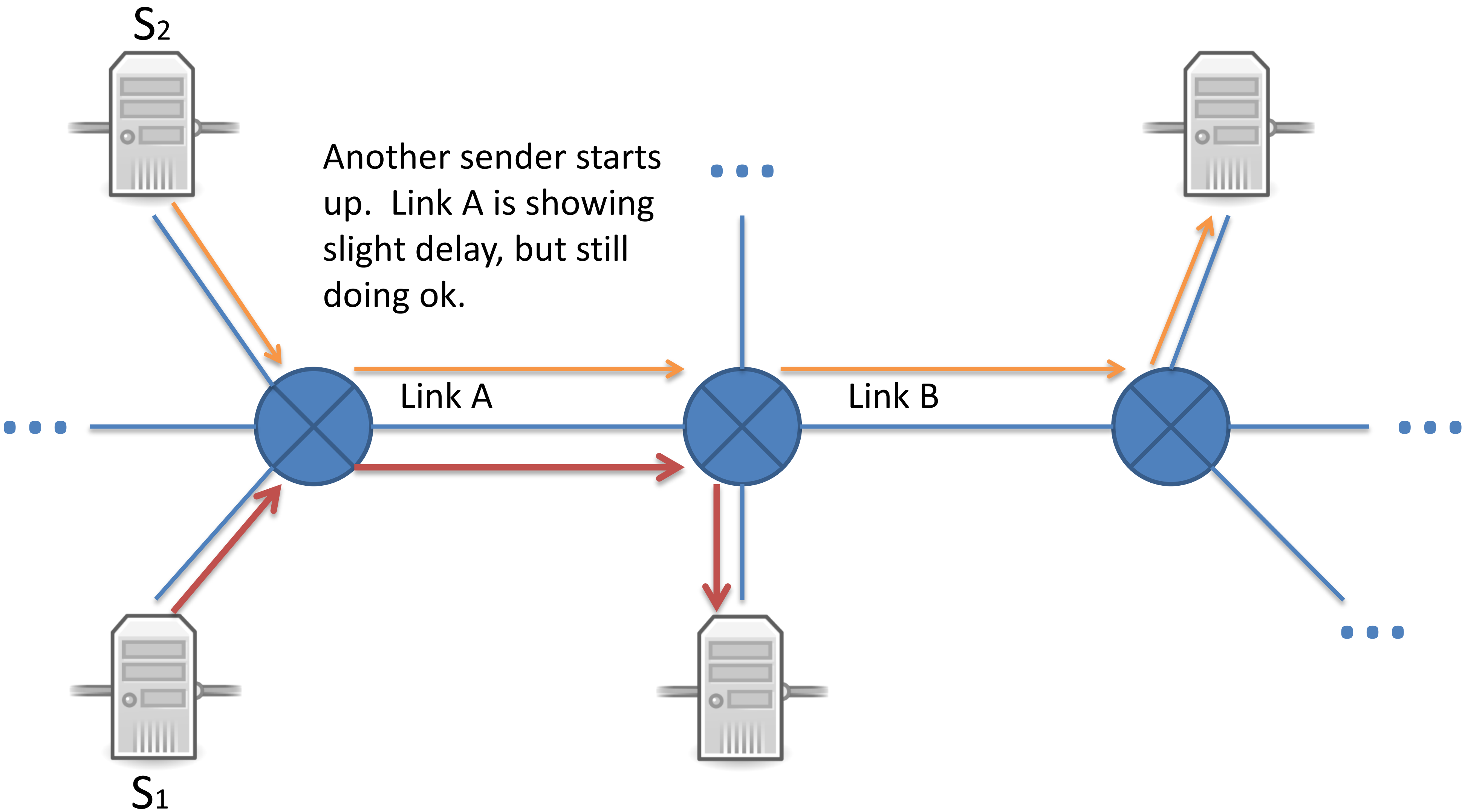
Congestion Collapse



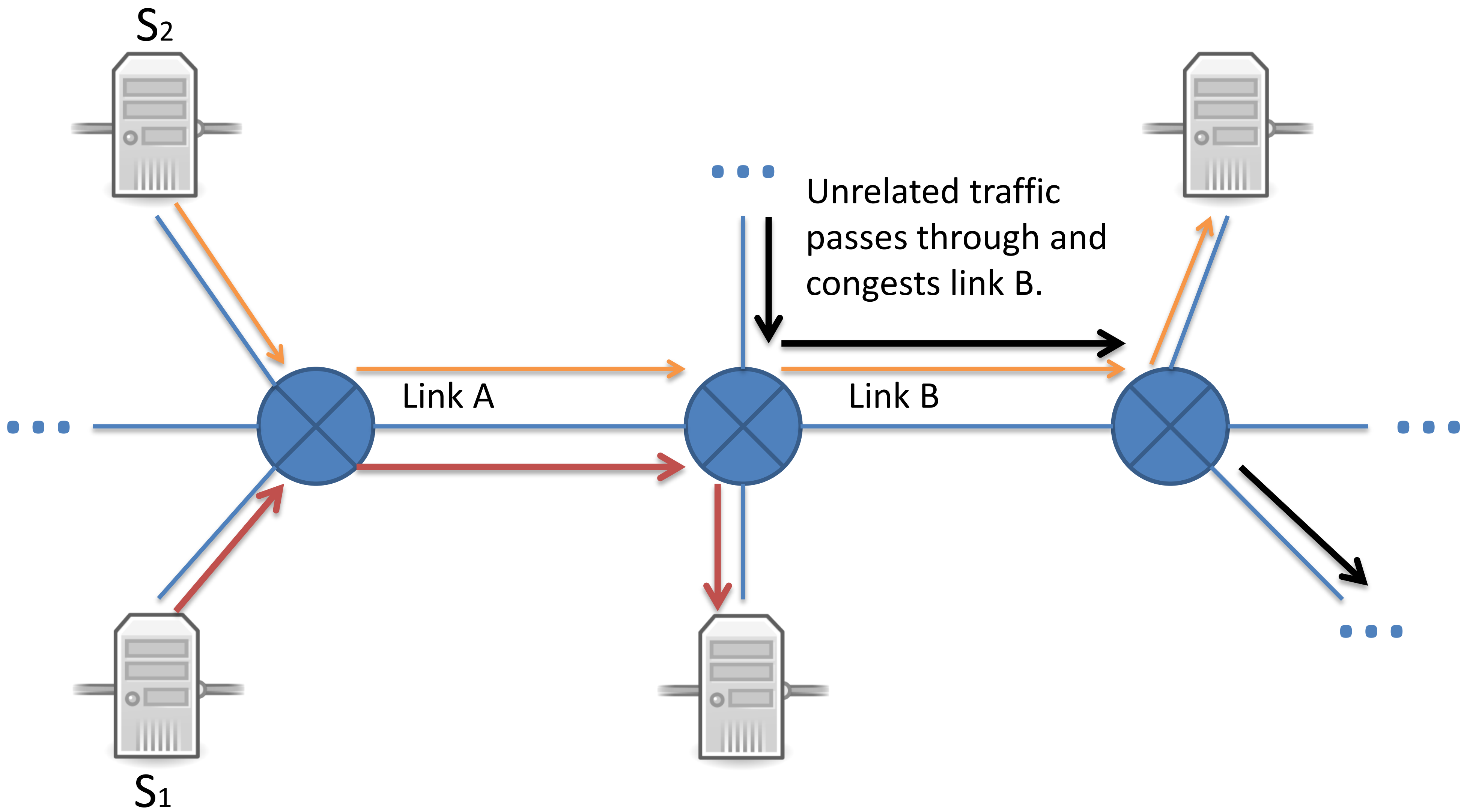
Congestion Collapse



Congestion Collapse

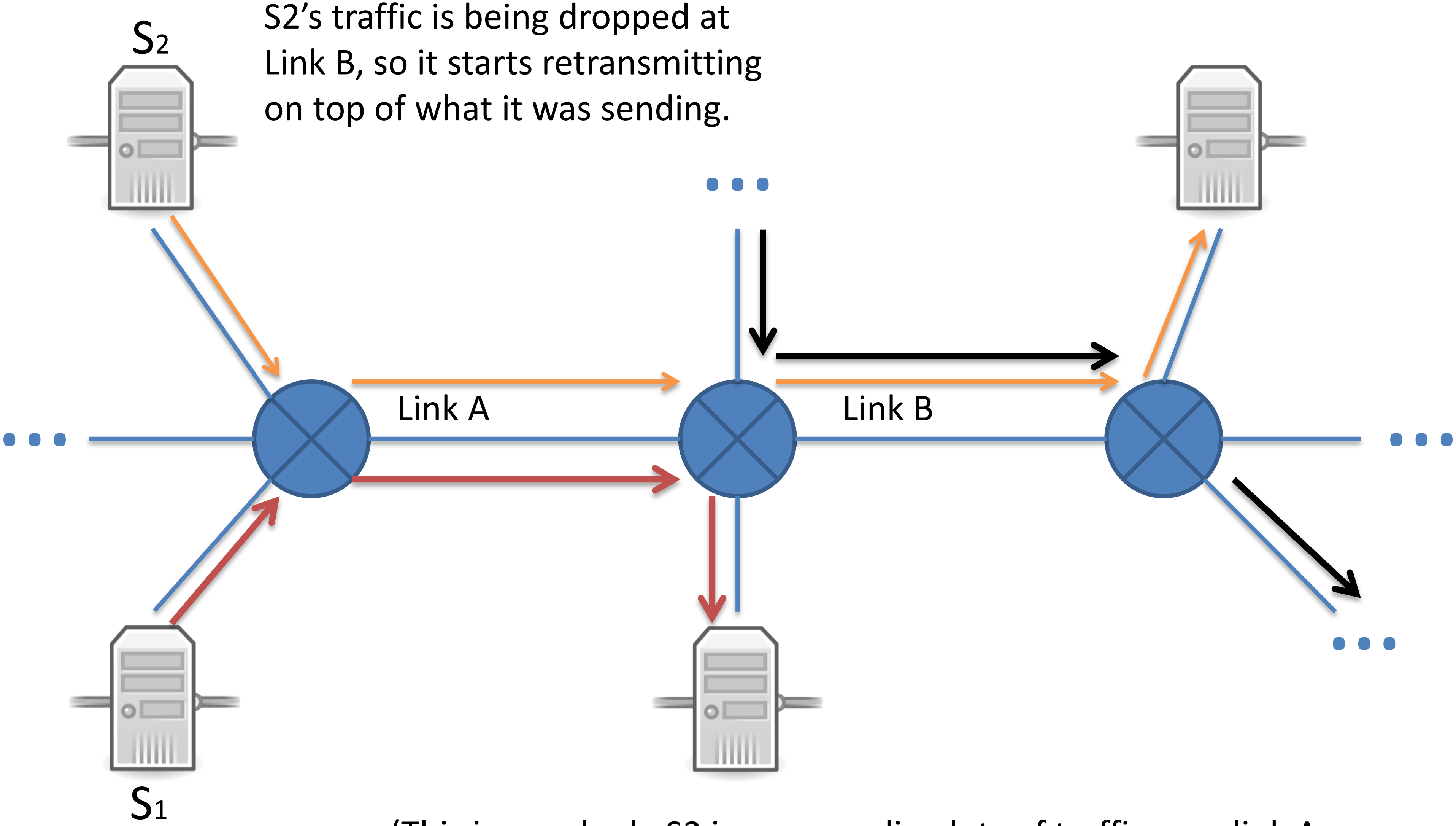


Congestion Collapse



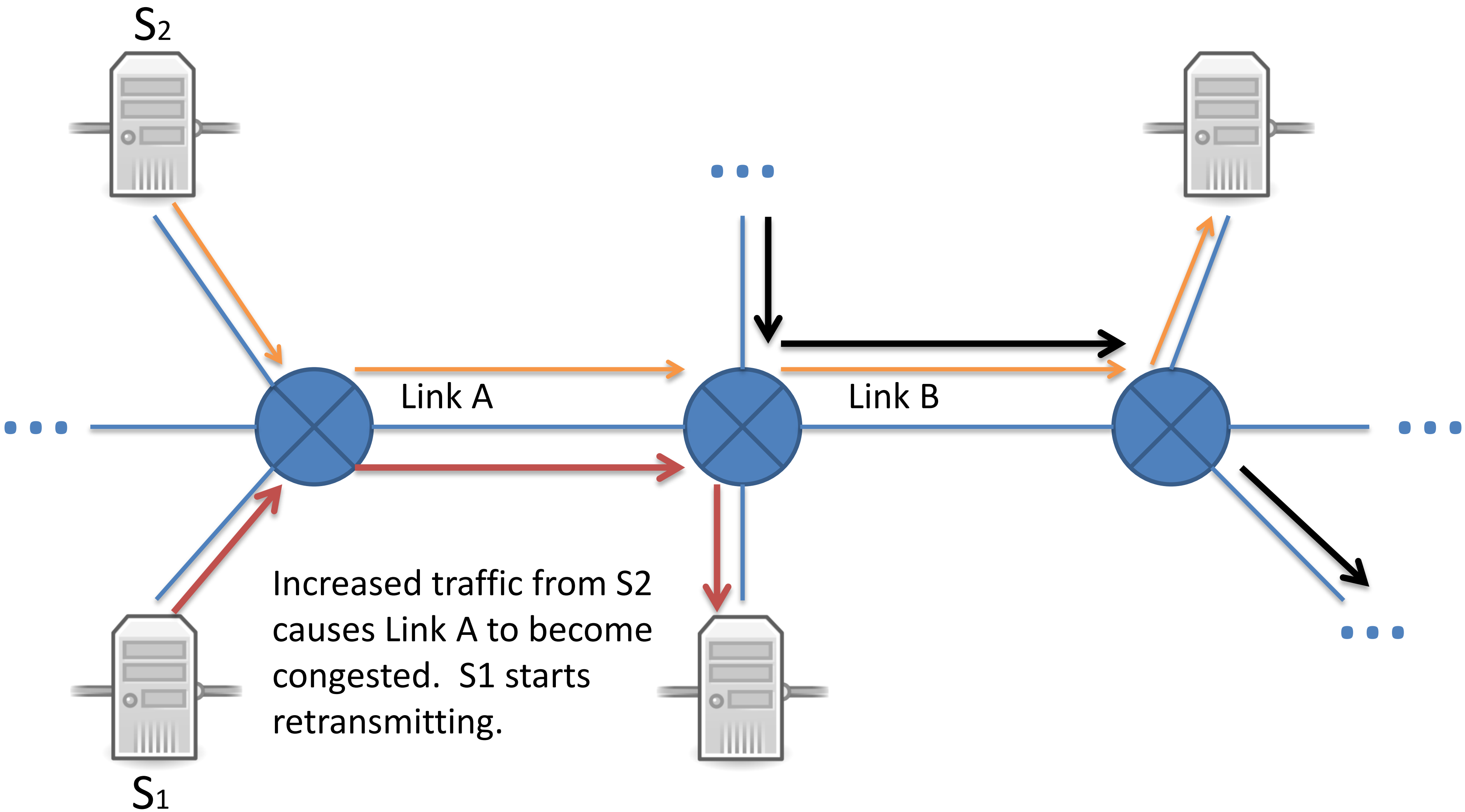
Congestion Collapse

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



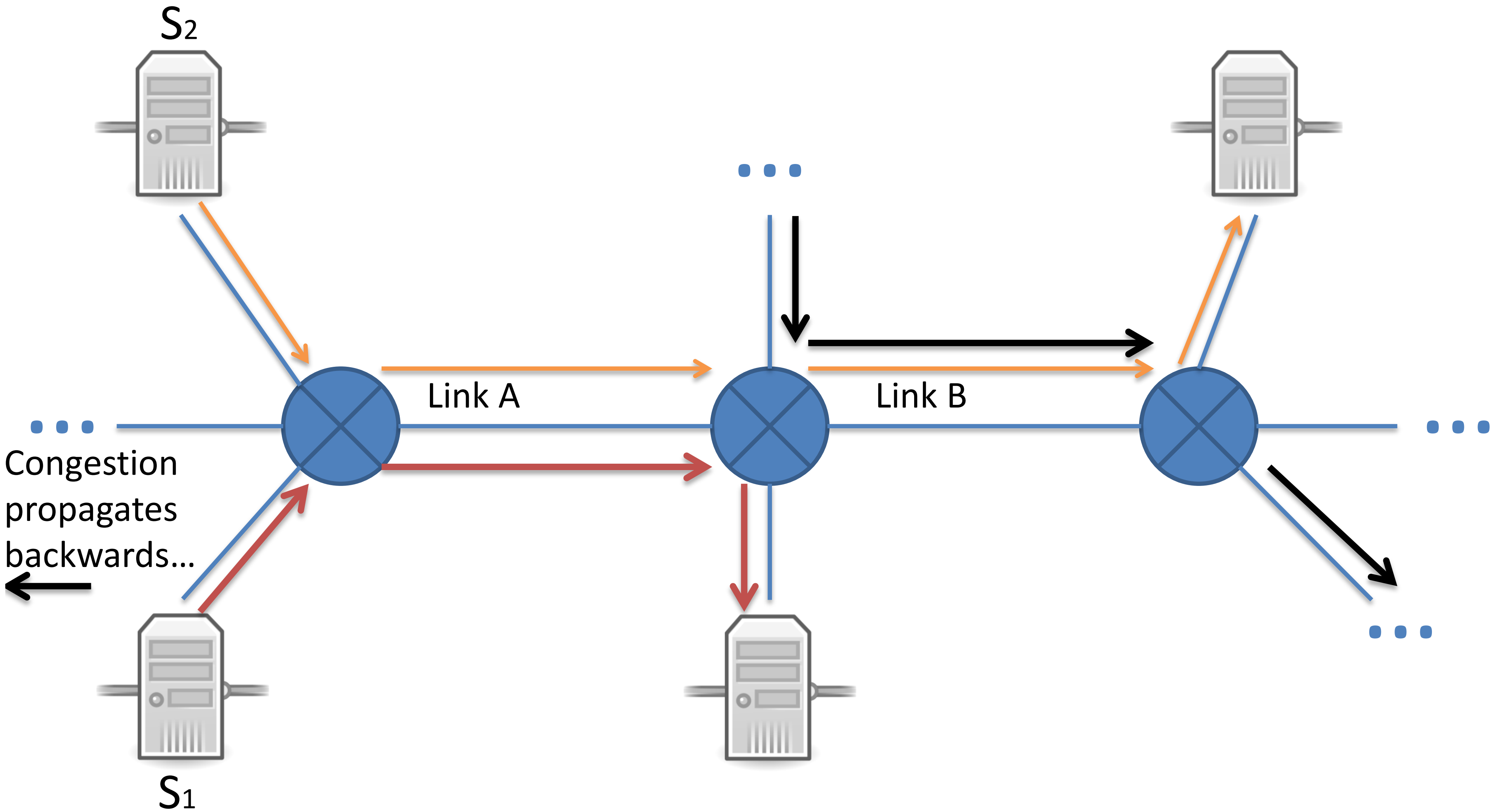
(This is very bad. S2 is now sending lots of traffic over link A that has no hope of crossing link B.)

Congestion Collapse



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

Congestion Collapse



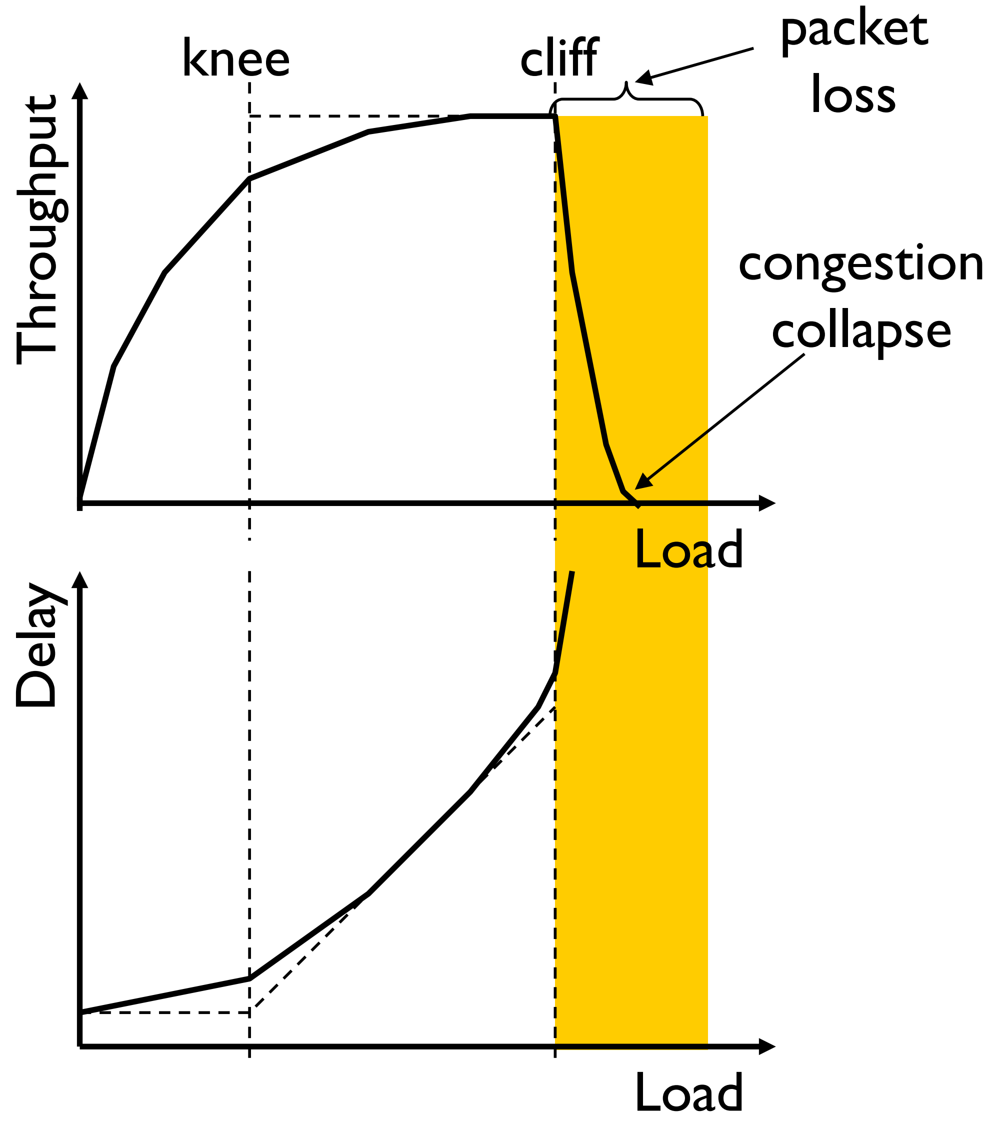
Without congestion control

congestion:

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

Cost of Congestion

- Knee – point after which
  - Throughput increases slowly
  - Delay increases fast
- Cliff – point after which
  - Throughput starts to drop to zero (congestion collapse)
  - Delay approaches infinity



Congestion Collapse

This happened to the Internet (then NSFnet) in 1986

- Rate dropped from a *blazing* 32 Kbps to 40bps
- This happened on and off for *two years*
- In 1988, Van Jacobson published "Congestion Avoidance and Control"
- The fix: senders voluntarily limit sending rate

Approaches towards congestion control

two broad approaches towards congestion control:

end-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate for sender to send at

Transport Layer: Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

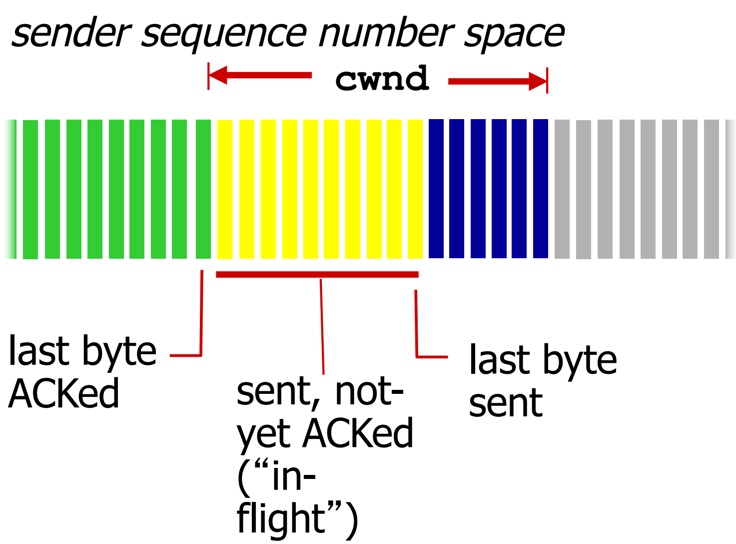
3.7 TCP congestion control

TCP's Approach in a Nutshell

- TCP connection has window
  - Controls number of packets in flight

- *TCP sending rate:*
  - *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

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



- Vary window size to control sending rate

All These Windows…

- Congestion Window: CWND
  - How many bytes can be sent without overflowing routers
  - Computed by the sender using congestion control algorithm

- Flow control window: Advertised / Receive Window (RWND)
  - How many bytes can be sent without overflowing receiver's buffers
  - Determined by the receiver and reported to the sender

- Sender-side window = minimum{CWND, RWND}
  - Assume for this lecture that RWND >> CWND

CWND

- This lecture will talk about CWND in units of MSS
  - (Recall MSS: Maximum Segment Size, the amount of payload data in a TCP packet)
  - This is only for pedagogical purposes
- Keep in mind that real implementations maintain CWND in bytes

Two Basic Questions

- How does the sender detect congestion?

- How does the sender adjust its sending rate?

Quiz: What is a "congestion event"

A: A segment loss (but how can the sender be sure of this?)

B: Increased delays

C: Receiving duplicate acknowledgement (s)

D: A retransmission timeout firing

E: Some subset of A, B, C & D (what is the subset?)

Quiz: 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 1 and start raising it again

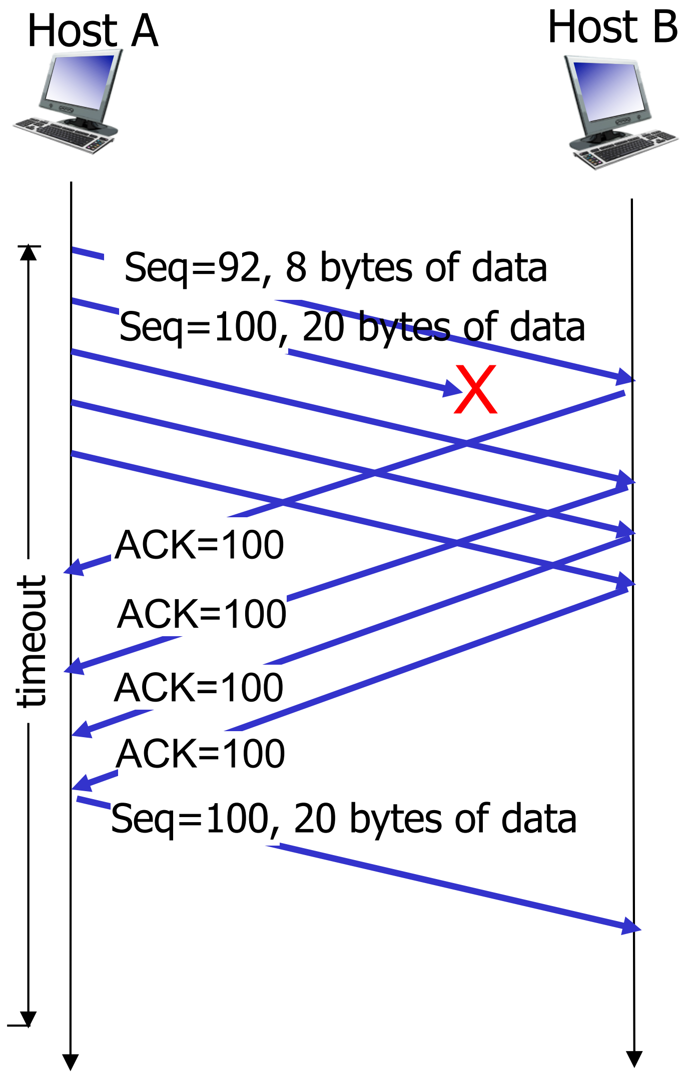
C: We should raise it until a "congestion event", then go back to median value and start raising it again.

D: We should sent as fast as possible at all times.

Not All Losses the Same

- Duplicate ACKs: isolated loss
  - dup ACKs indicate network capable of delivering some segments
- Timeout: much more serious
  - Not enough dup ACKs
  - Must have suffered several losses
- Will adjust rate differently for each case

RECAP: TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

Rate Adjustment

- Basic structure:
  - Upon receipt of ACK (of new data): increase rate
  - Upon detection of loss: decrease rate

- How we increase/decrease the rate depends on the phase of congestion control we're in:
  - Discovering available bottleneck bandwidth *vs.*
  - Adjusting to bandwidth variations

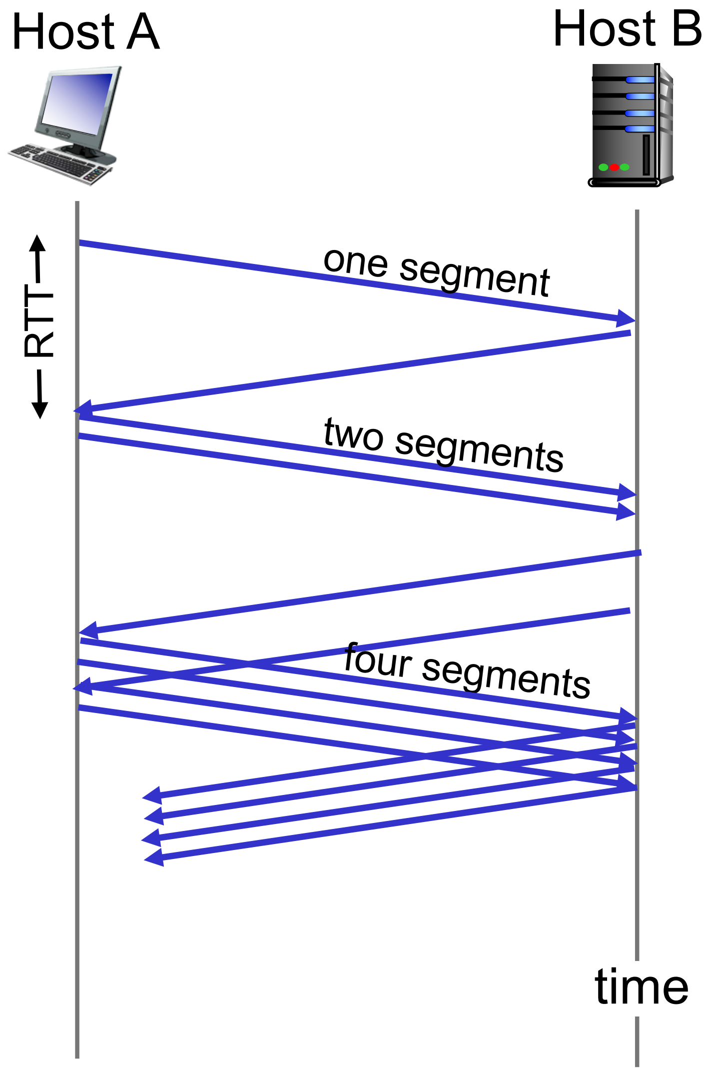
Bandwidth Discovery with Slow Start (SS)

- Goal: estimate available bandwidth
  - start slow (for safety)
  - but ramp up quickly (for efficiency)

- Consider
  - RTT = 100ms, MSS=1000bytes
  - Window size to fill 1Mbps of BW = 12.5 packets
  - Window size to fill 1Gbps = 12,500 packets
  - Either is possible!

TCP Slow Start

- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - Simpler implementation achieved by incrementing **cwnd** for every ACK received
- *summary:* initial rate is slow but ramps up exponentially fast



Adjusting to Varying Bandwidth

- Slow start gave an estimate of available bandwidth

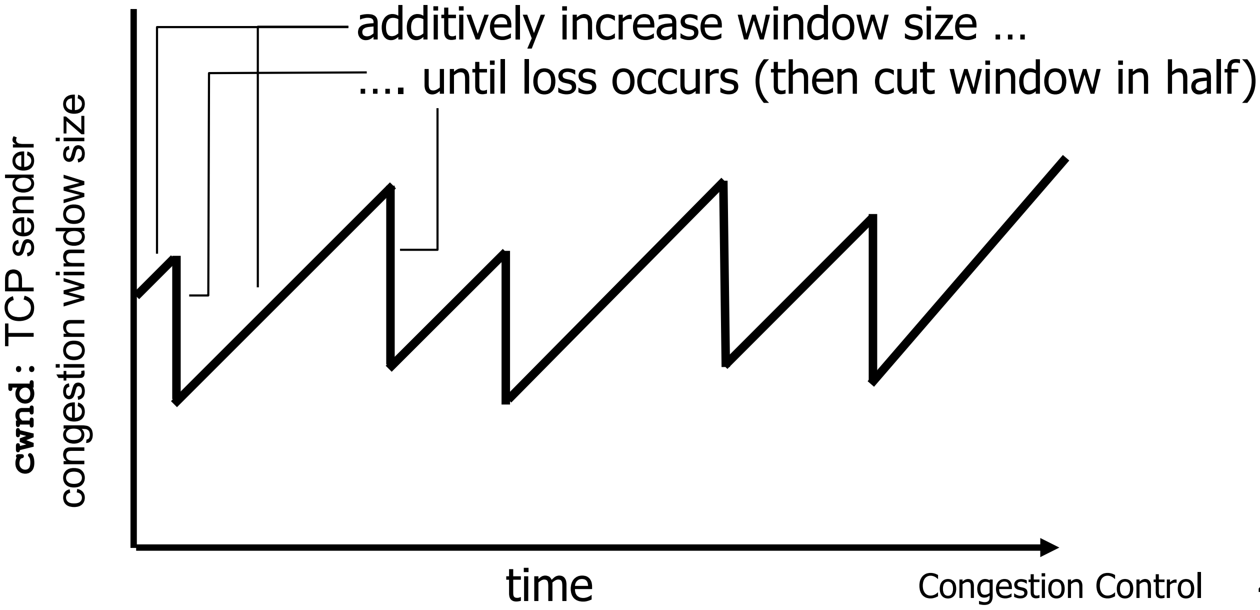
- Now, want to track variations in this available bandwidth, oscillating around its current value
  - Repeated probing (rate increase) and backoff (rate decrease)
  - Known as Congestion Avoidance (CA)

- TCP uses: "Additive Increase Multiplicative Decrease" (AIMD)
  - We'll see why shortly…

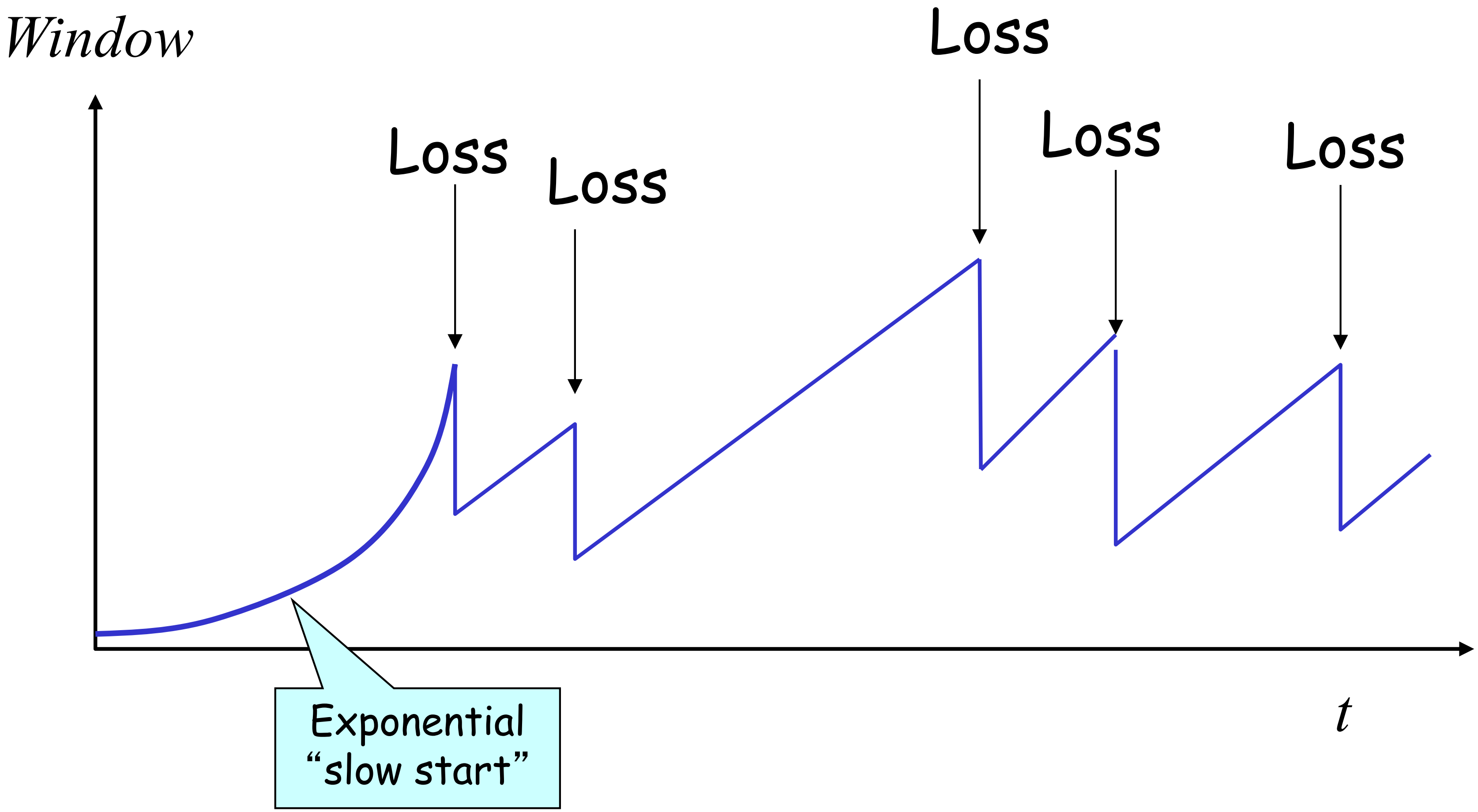
AIMD

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **`cwnd`** by 1 MSS every RTT until loss detected
    - For each successful RTT, **`cwnd = cwnd +1`**
    - Simple implementation: for each ACK, **`cwnd = cwnd + 1/cwnd`**
  - *multiplicative decrease:* cut **`cwnd`** in half after loss

AIMD saw tooth behavior: probing for bandwidth



Leads to the TCP "Sawtooth"



Slow-Start vs. AIMD

- When does a sender stop Slow-Start and start Additive Increase?

- Introduce a "slow start threshold" (ssthresh)
  - Initialized to a large value
  - On timeout, ssthresh = CWND/2

- When CWND = ssthresh, sender switches from slow-start to AIMD-style increase

Implementation

- State at sender
  - CWND (initialized to a small constant)
  - ssthresh (initialized to a large constant)
  - [Also dupACKcount and timer, as before]
- Events
  - ACK (new data)
  - dupACK (duplicate ACK for old data)
  - Timeout

Event: ACK (new data)

- If CWND < ssthresh
  - CWND += 1

- *CWND packets per RTT*
- *Hence after one RTT with no drops:*
  CWND = 2xCWND

Event: ACK (new data)

- If CWND < ssthresh
  - CWND += 1

Slow start phase

- Else
  - CWND = CWND + 1/CWND

"Congestion Avoidance" phase (additive increase)

- *CWND packets per RTT*
- *Hence after one RTT with no drops:*
  CWND = CWND + 1

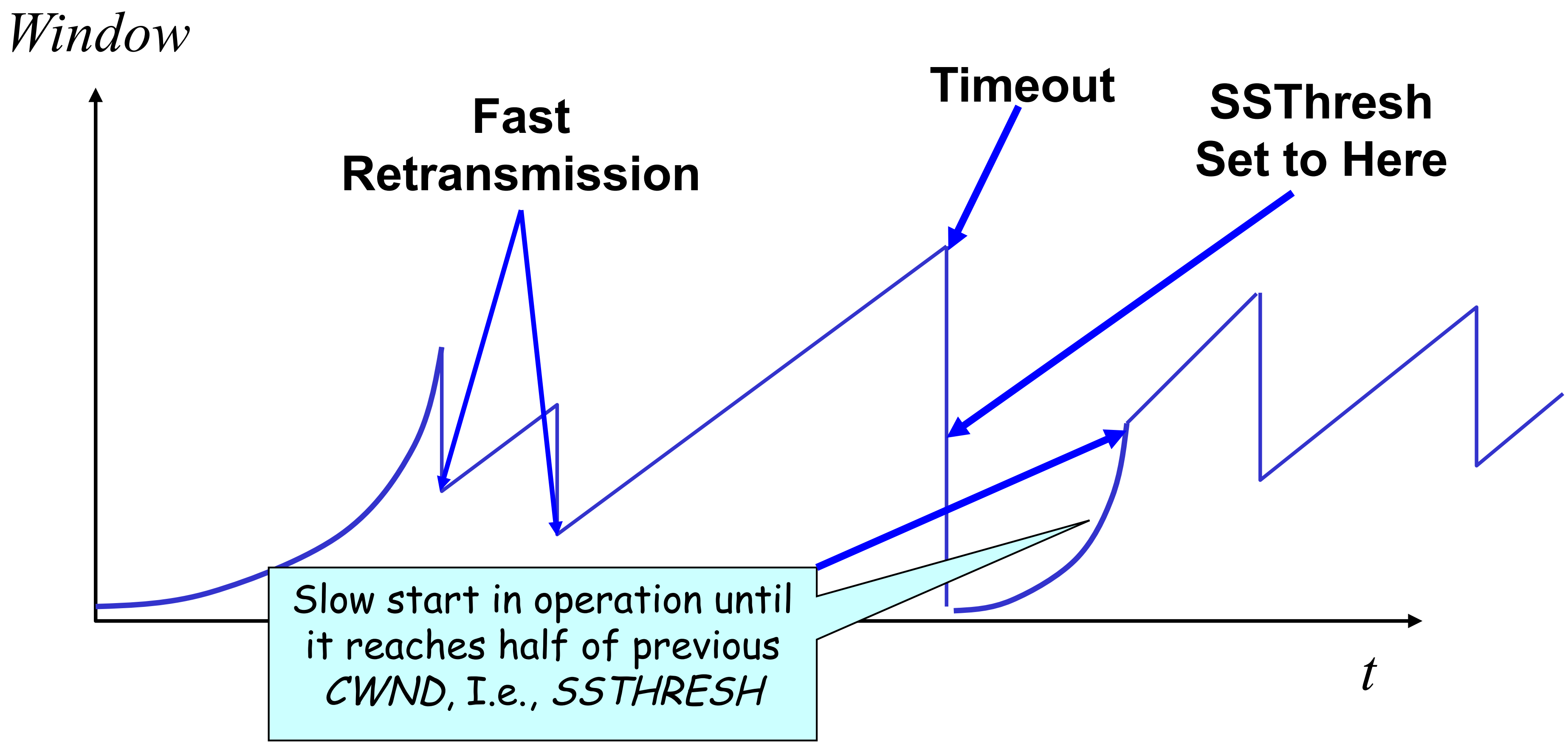
Event: dupACK

- dupACKcount ++
- If dupACKcount = 3 /* fast retransmit */
  - ssthresh = CWND/2
  - CWND = CWND/2

Event: TimeOut

- On Timeout
  - ssthresh ← CWND/2
  - CWND ← 1

Example



Slow-start restart: Go back to CWND = 1 MSS, but take advantage of knowing the previous value of CWND

One Final Phase: Fast Recovery

- The problem: congestion avoidance too slow in recovering from an isolated loss

Example (window in units of MSS, not bytes)

- Consider a TCP connection with:
  - CWND=10 packets (of size MSS, which is 100 bytes)
  - Last ACK was for byte # 101
    - i.e., receiver expecting next packet to have seq. no. 101

- 10 packets [101, 201, 301,…, 1001] are in flight
  - Packet 101 is dropped
  - What ACKs do they generate?
  - And how does the sender respond?

Timeline

- ACK 101 (due to 201) cwnd=10 dupACK#1 (no xmit)
- ACK 101 (due to 301) cwnd=10 dupACK#2 (no xmit)
- ACK 101 (due to 401) cwnd=10 dupACK#3 (no xmit)
- RETRANSMIT 101 ssthresh=5 cwnd= 5
- ACK 101 (due to 501) cwnd=5 + 1/5 (no xmit)
- ACK 101 (due to 601) cwnd=5 + 2/5 (no xmit)
- ACK 101 (due to 701) cwnd=5 + 3/5 (no xmit)
- ACK 101 (due to 801) cwnd=5 + 4/5 (no xmit)
- ACK 101 (due to 901) cwnd=5 + 5/5 (no xmit)
- ACK 101 (due to 1001) cwnd=6 + 1/5 (no xmit)
- ACK 1101 (due to 101) ← only now can we transmit new packets
- Plus no packets in flight so ACK "clocking" (to increase CWND) stalls for another RTT

Solution: Fast Recovery

Idea: Grant the sender temporary "credit" for each dupACK so as to keep packets in flight

- If dupACKcount = 3
  - ssthresh = cwnd/2
  - cwnd = ssthresh + 3
- While in fast recovery
  - cwnd = cwnd + 1 for each additional duplicate ACK
- Exit fast recovery after receiving new ACK
  - set cwnd = ssthresh

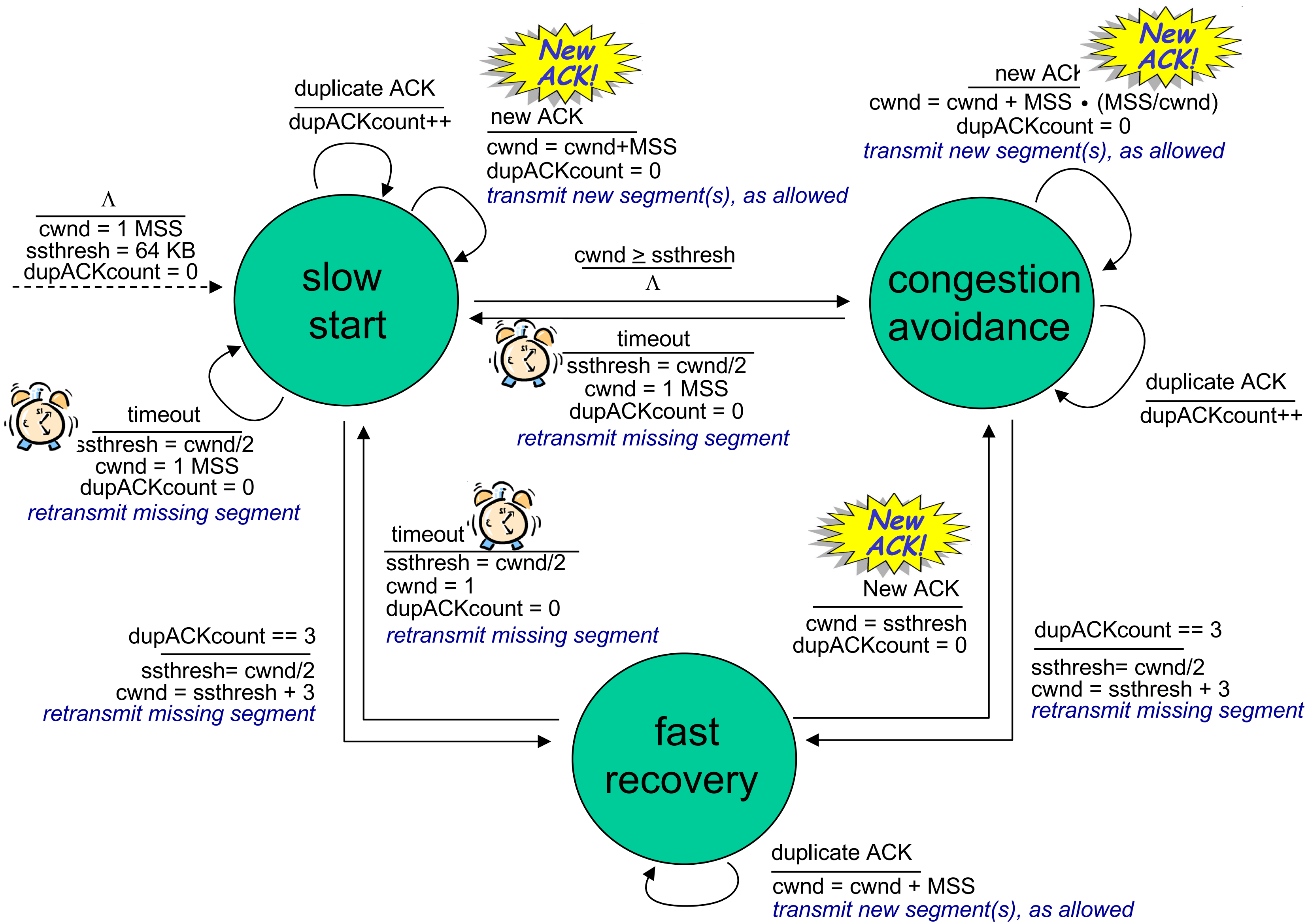
Example

- Consider a TCP connection with:
  - CWND=10 packets (of size MSS = 100 bytes)
  - Last ACK was for byte # 101
    - i.e., receiver expecting next packet to have seq. no. 101
- 10 packets [101, 201, 301,…, 1001] are in flight
  - Packet 101 is dropped

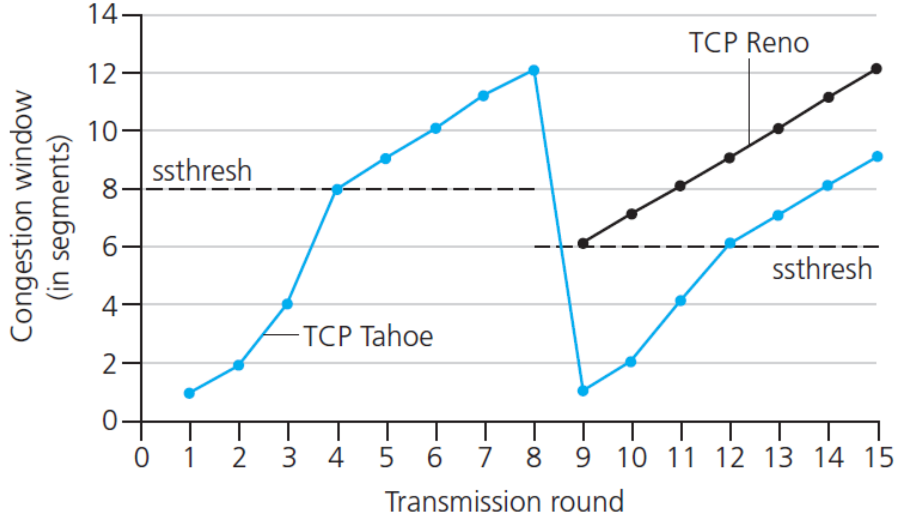
Timeline

- ACK 101 (due to 201) cwnd=10 dup#1
- ACK 101 (due to 301) cwnd=10 dup#2
- ACK 101 (due to 401) cwnd=10 dup#3
- REXMIT 101 ssthresh=5 cwnd= 8 (5+3)
- ACK 101 (due to 501) cwnd= 9 (no xmit)
- ACK 101 (due to 601) cwnd=10 (no xmit)
- ACK 101 (due to 701) cwnd=11 (xmit 1101)
- ACK 101 (due to 801) cwnd=12 (xmit 1201)
- ACK 101 (due to 901) cwnd=13 (xmit 1301)
- ACK 101 (due to 1001) cwnd=14 (xmit 1401)
- ACK 1101 (due to 101) cwnd = 5 (xmit 1501) ← exiting fast recovery
- Packets 1101-1401 already in flight
- ACK 1201 (due to 1101) cwnd = 5 + 1/5 ← back in congestion avoidance

Summary: TCP Congestion Control

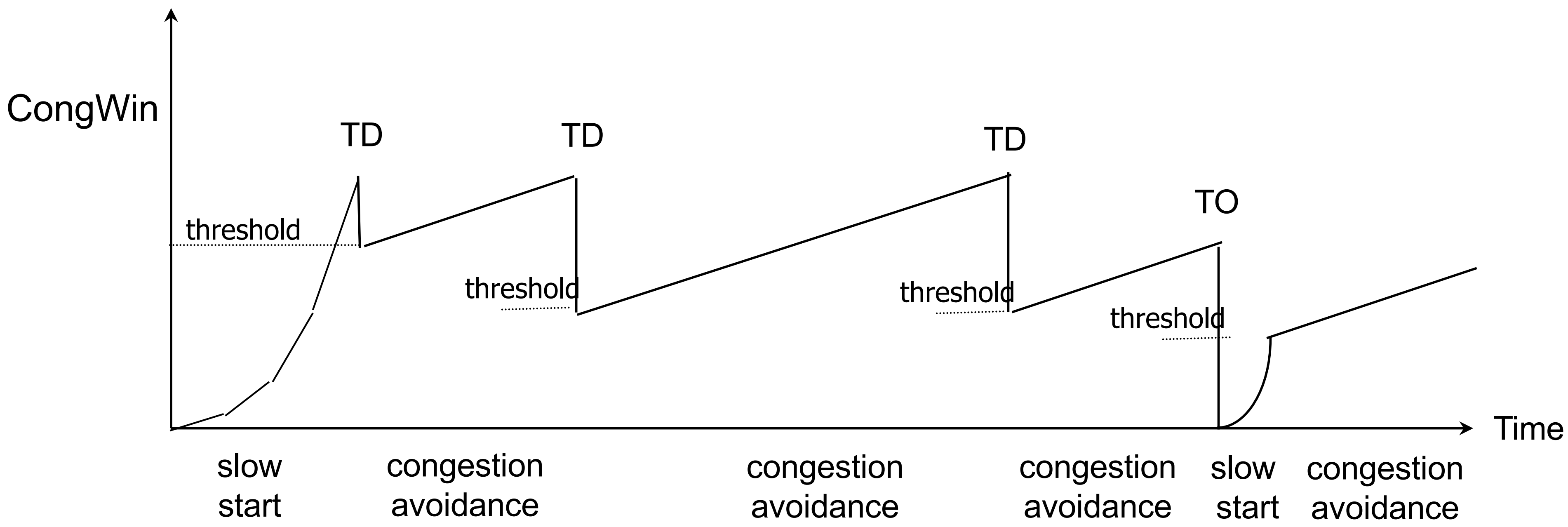


TCP Flavours



- TCP-Tahoe
  - cwnd =1 on triple dup ACK & timeout
- TCP-Reno
  - cwnd =1 on timeout
  - cwnd = cwnd/2 on triple dup ACK
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK (NOT COVERED IN THE COURSE)
  - incorporates selective acknowledgements

TCP/Reno: Big Picture



TD: Triple duplicate acknowledgements
TO: Timeout

Transport Layer: Summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

next:

- leaving the network "edge" (application, transport layers)
- into the network "core"