15-441/641: Computer Networks
The Transport Layer, Part 2 of 3

15-441 Spring 2019

Profs Peter Steenkiste & **Justine Sherry**

Carnegie Mellon University

Questions to discuss with a friend

- What are some things that make reliable transmission hard?
    - *Think: what went wrong in our reliable transmission race?*
- What is the difference between a “cumulative ACK” and a “basic ACK”?
    - What is one benefit of each?
- How do Selective Repeat and Go-back-N improve upon Stop-and-Wait?
- Can the transport layer guarantee:
    - That all packets will arrive at their destination?
    - That packets will be delivered at a certain throughput?
    - That packets will be delivered with a certain latency?

Last Time: Reliable Transmission

- When transmitting across the Internet, how can we be sure that every message reaches its destination?
  - Retransmit!
- Three approaches:
  - Stop and Wait
  - Go Back N
  - Selective Repeat

Stop-and-Wait: Summary

- **Sender:**
  - Transmit packets one by one. Label each with a sequence number. Set timer after transmitting.
  - If receive ACK, send the next packet.
  - If timer goes off, re-send the previous packet.
- **Receiver:**
  - When receive packet, send ACK.
  - If packet is corrupted, just ignore it — sender will eventually re-send.

Can I get some volunteers to act it out?

Selective Repeat

- **Sender:**
  - Send packets from the window. Set timeout for each packet.
  - On receiving ACKs for the “left side” of the window, slide forward.
    - Send packets that have now entered the window.
  - On timeout, retransmit only the timed out packet
- **Receiver**
  - Keep a buffer of size of the window.
  - On receiving packets, send ACKs for every packet.
  - If packets come in out of order, just store them in the buffer and send ACK anyway.

Can I get some volunteers to act it out?

Today’s Agenda

- #1: How big should we size the window?
- #2: How should we determine the BDP?
- #3: How does “plain” TCP work?

Today's Agenda

- **#1: How big should we size the window?**
- #2: How should we determine the BDP?
- #3: How does "plain" TCP work?

Sliding Windows

- A sender's "window" contains a set of packets that have been transmitted but not yet acked.
- Sliding windows improve the efficiency of a transport protocol.
- Two questions we need to answer to use windows:
  - (1) How do we handle loss with a windowed approach?
  - (2) How big should we make the window?

Last Time

- A sender's "window" contains a set of packets that have been transmitted but not yet acked.
- Sliding windows improve the efficiency of a transport protocol.
- Two questions we need to answer to use windows:
  - (1) How do we handle loss with a windowed approach?
  - (2) How big should we make the window?

Today

- A sender's "window" contains a set of packets that have been transmitted but not yet acked.
- Sliding windows improve the efficiency of a transport protocol.
- Two questions we need to answer to use windows:
  - (1) How do we handle loss with a windowed approach?
  - (2) How big should we make the window?

Why not send as fast as we can?

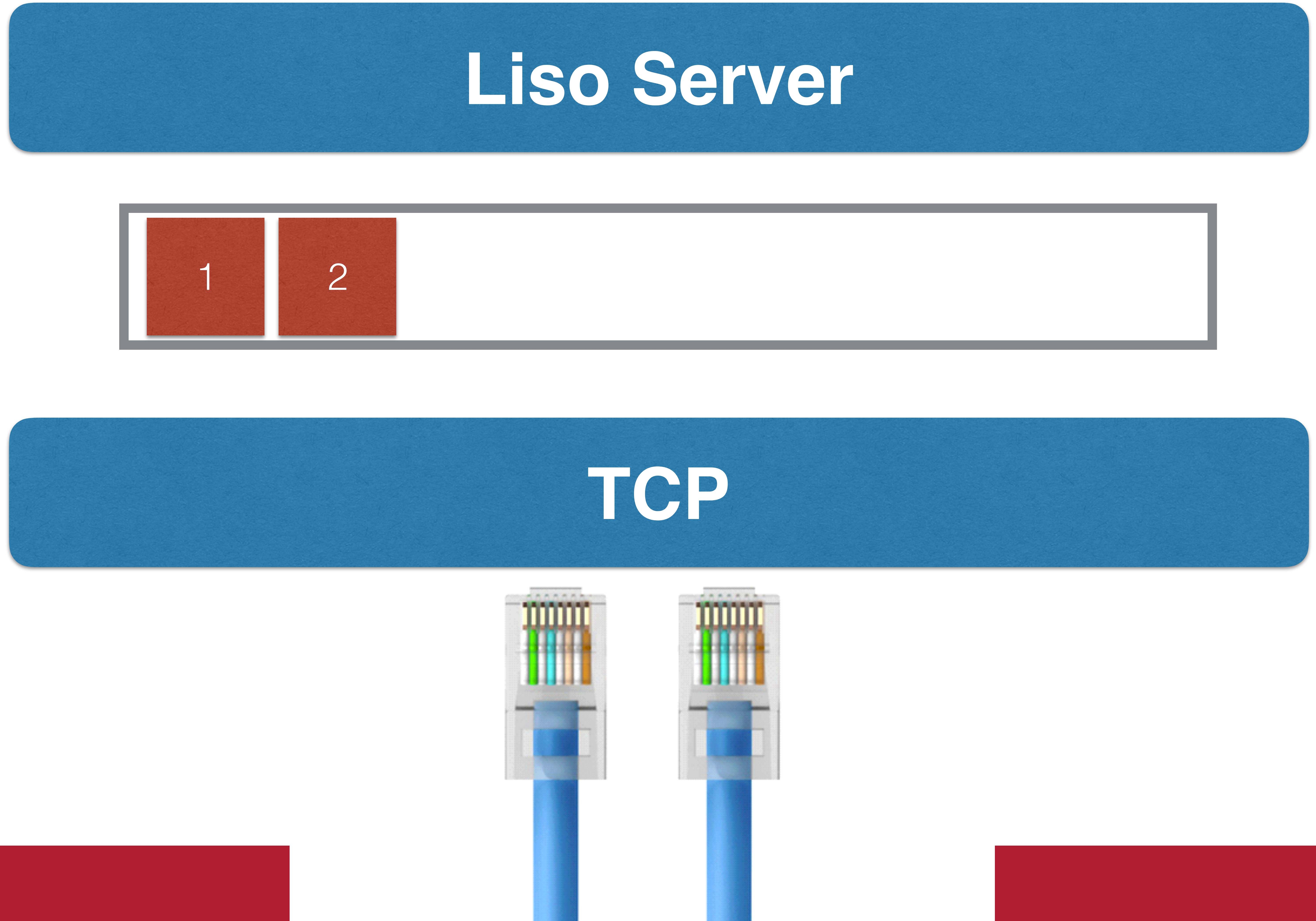
Problem #1: Flow Control

Yet another demo…
I need two volunteers, one of whom is confident reading out loud in English!

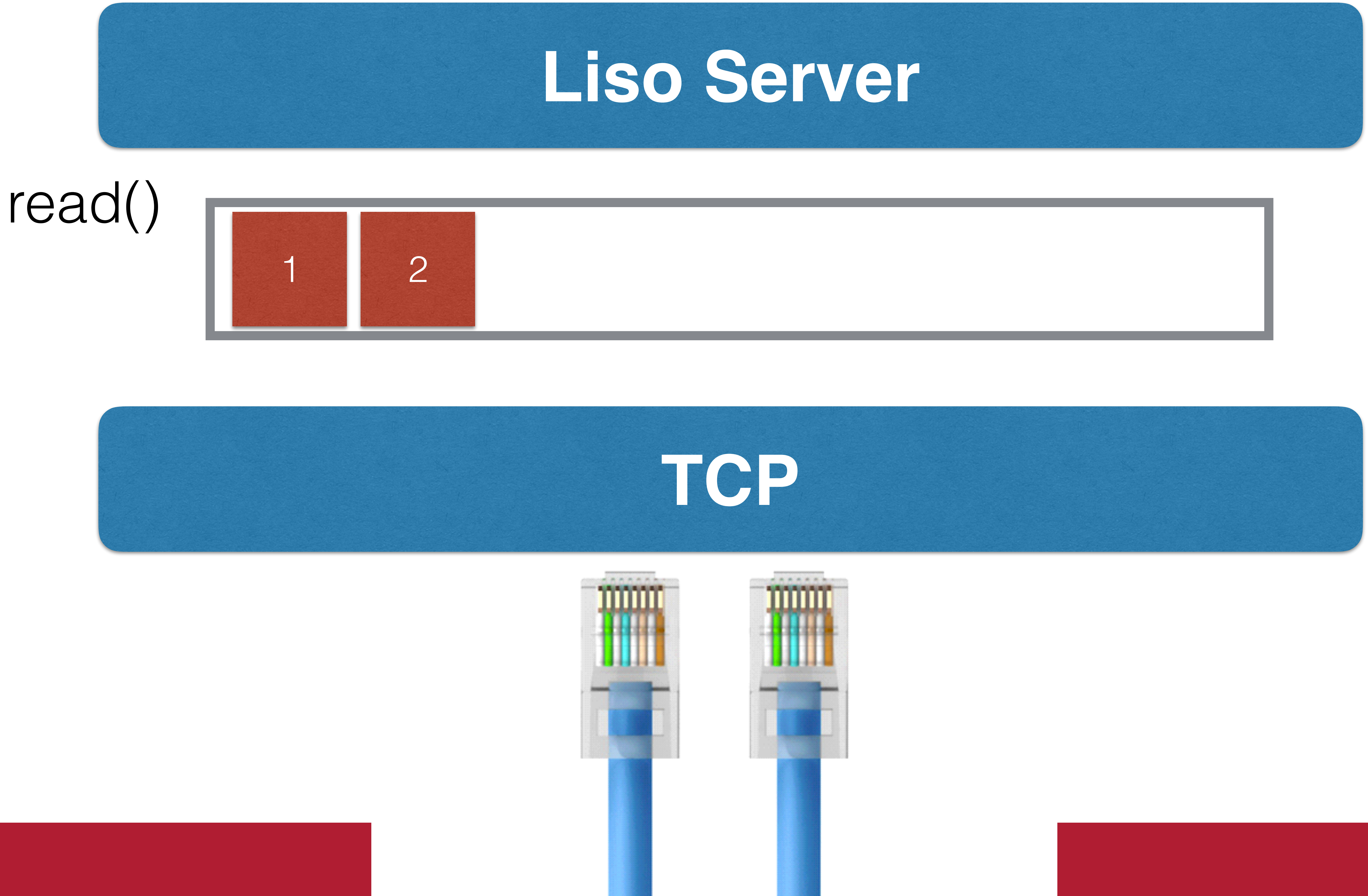
Flow Control: Don’t overload the receiver.

Bonus candy: who wrote the essay in the packets? What is the essay named?

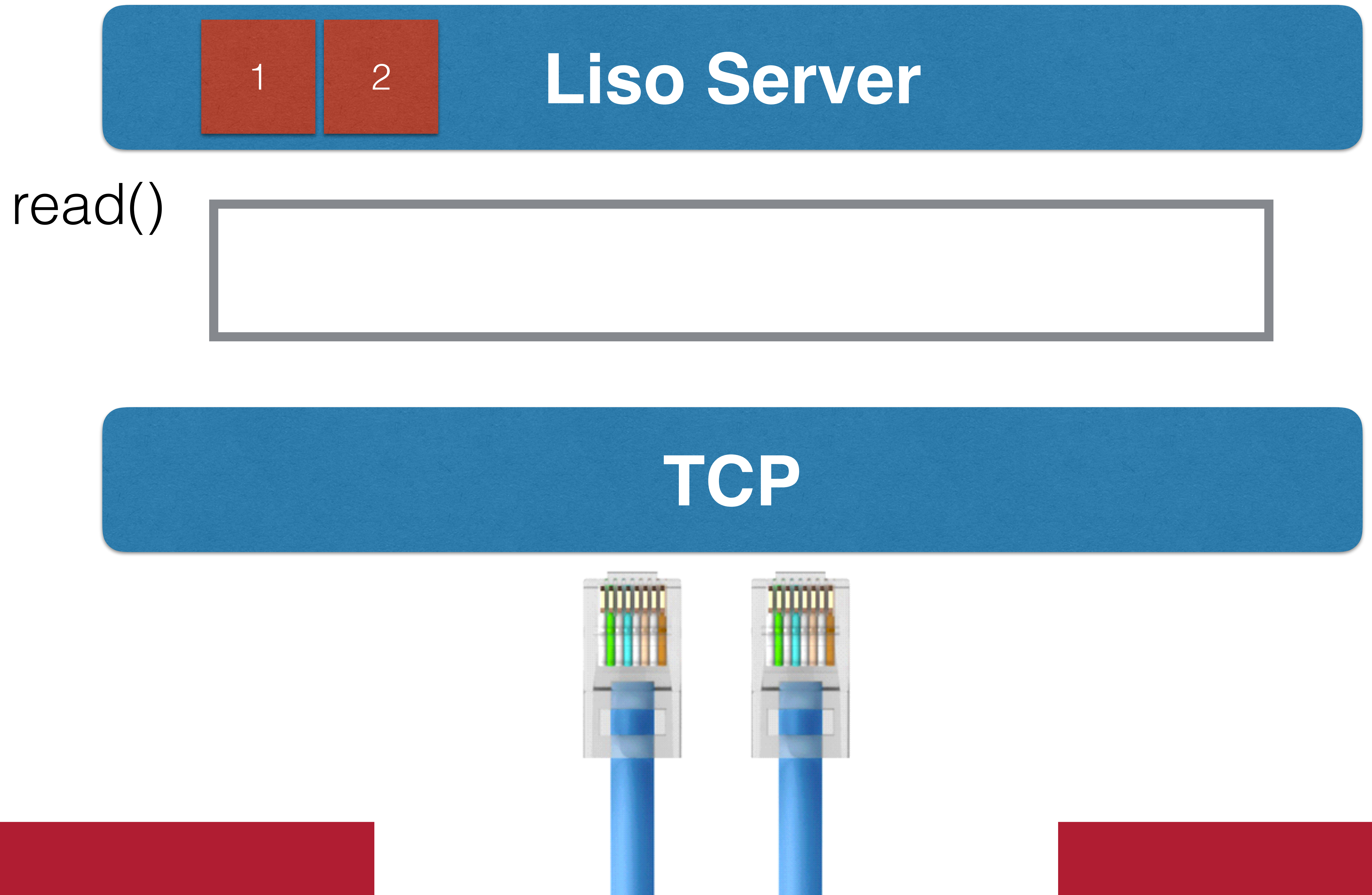
Receive Buffer



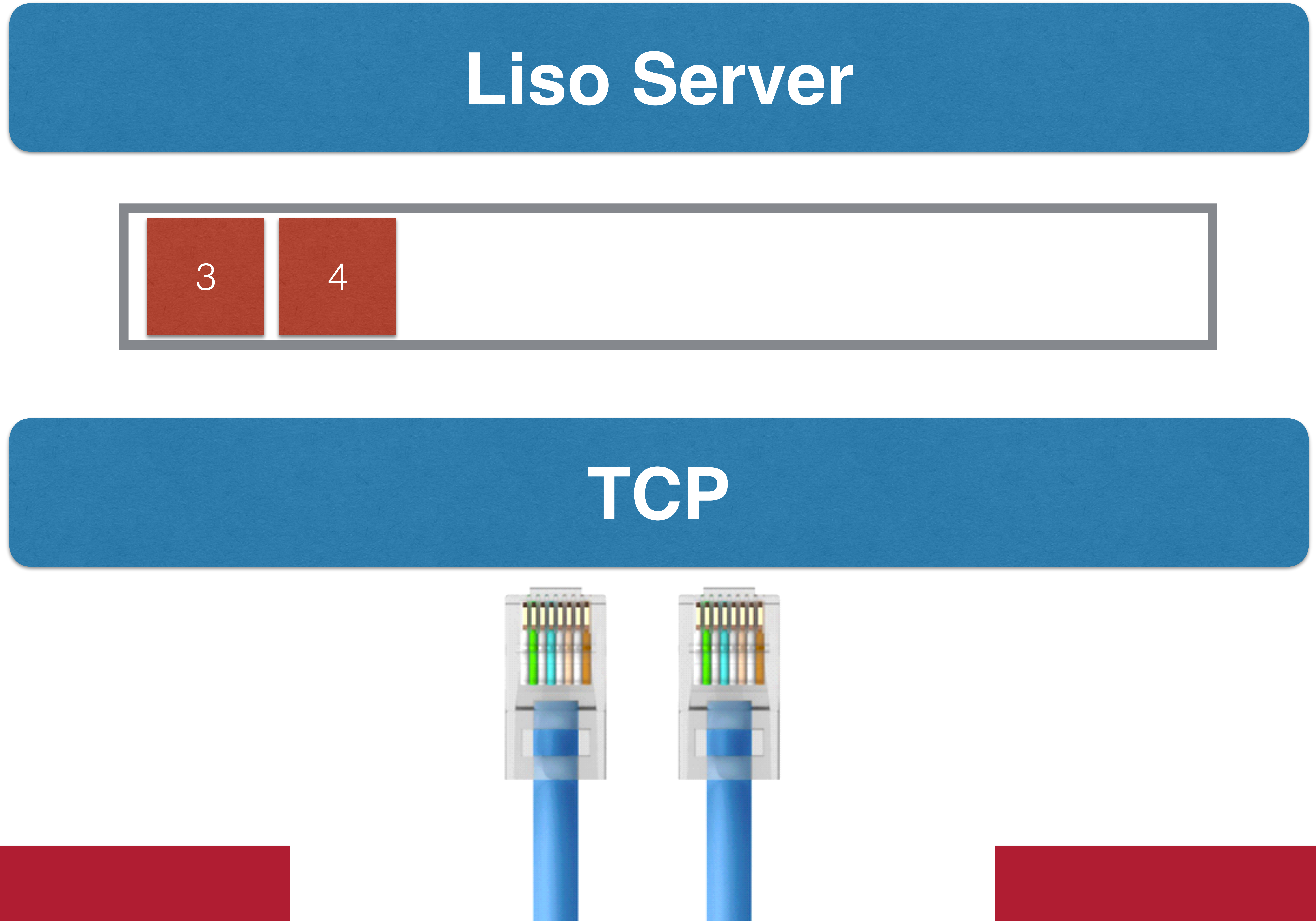
Receive Buffer



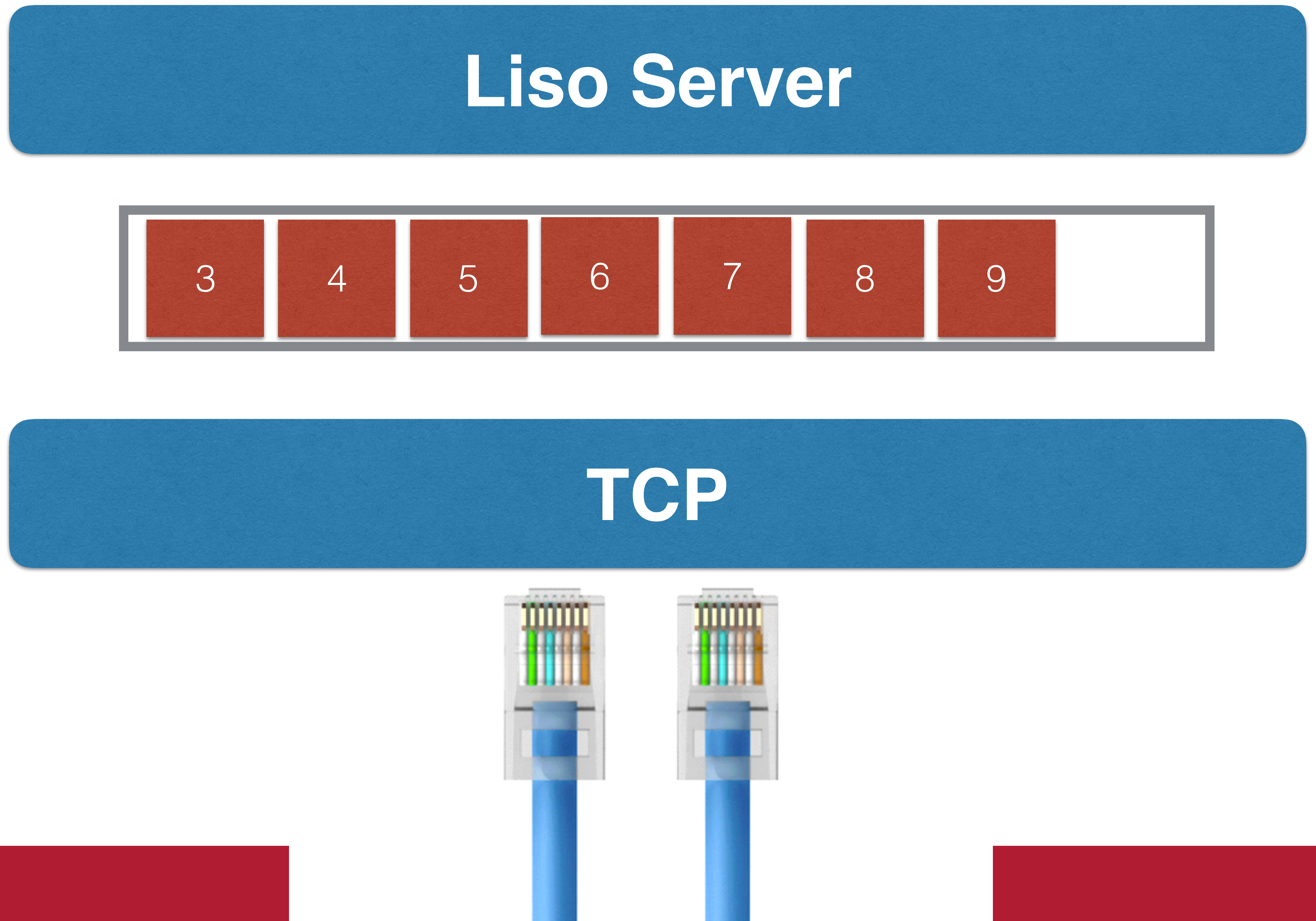
Receive Buffer



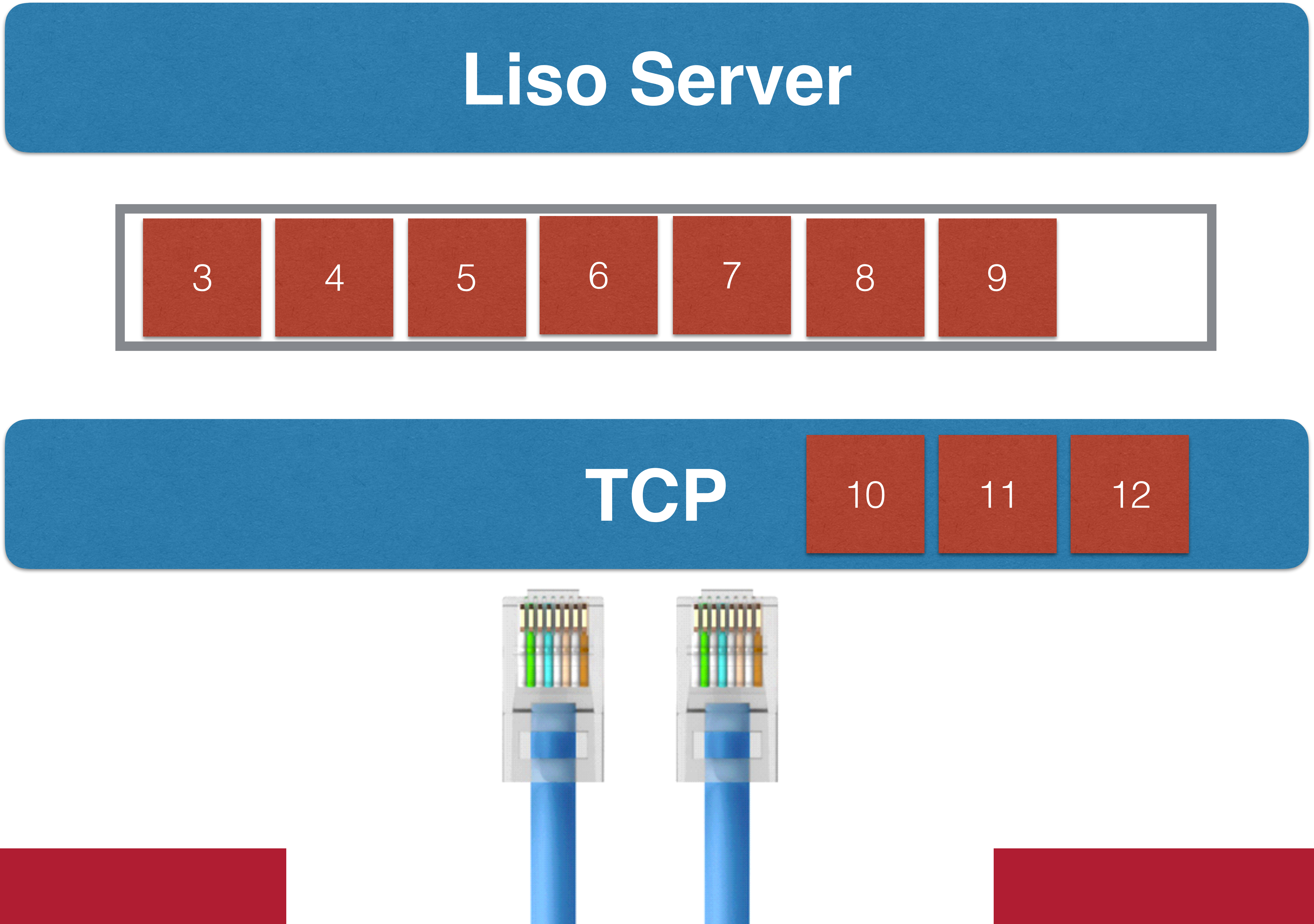
Receive Buffer



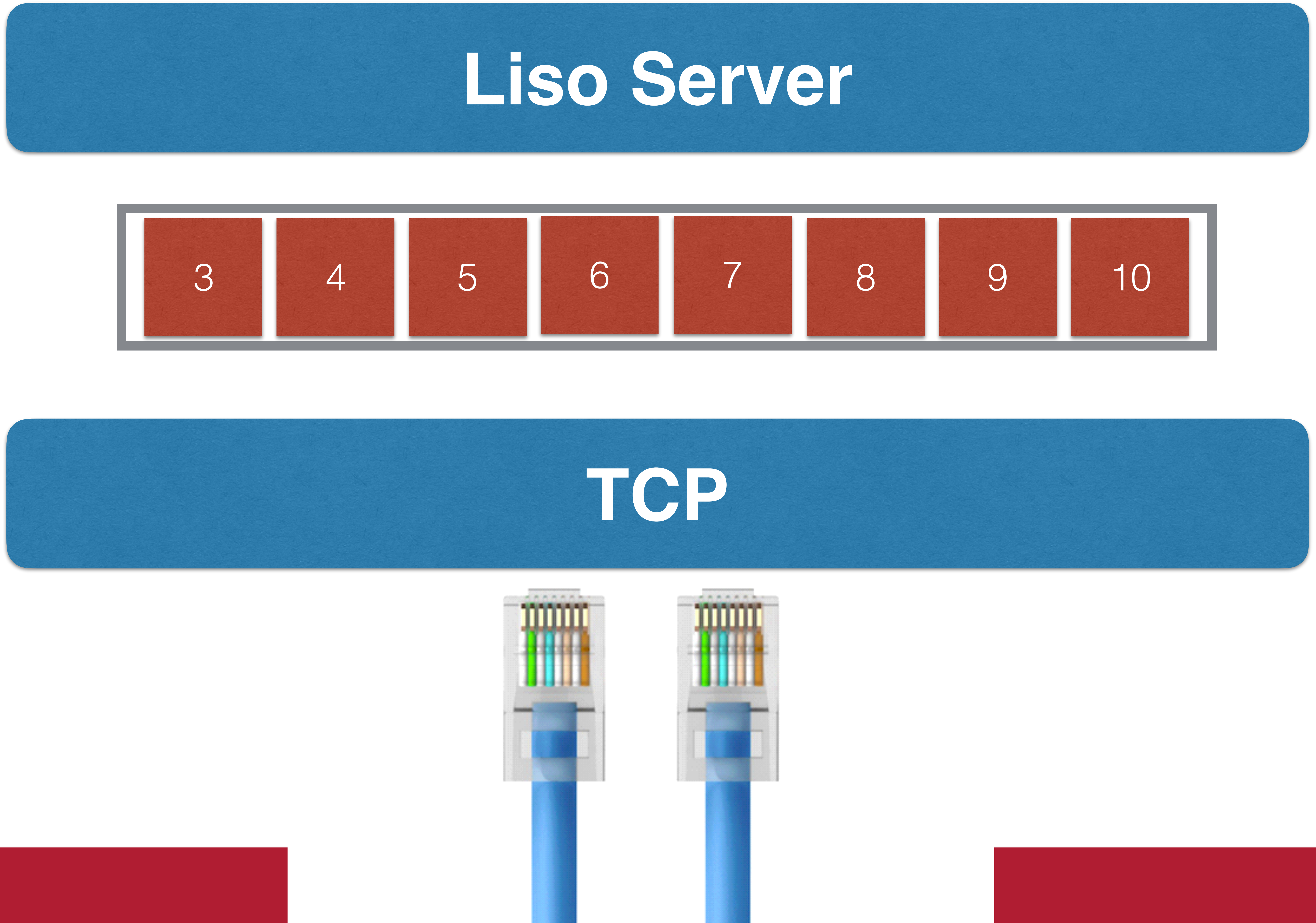
Receive Buffer



Receive Buffer



Receive Buffer



11 and 12 just get dropped :(

Solution: Advertised Window

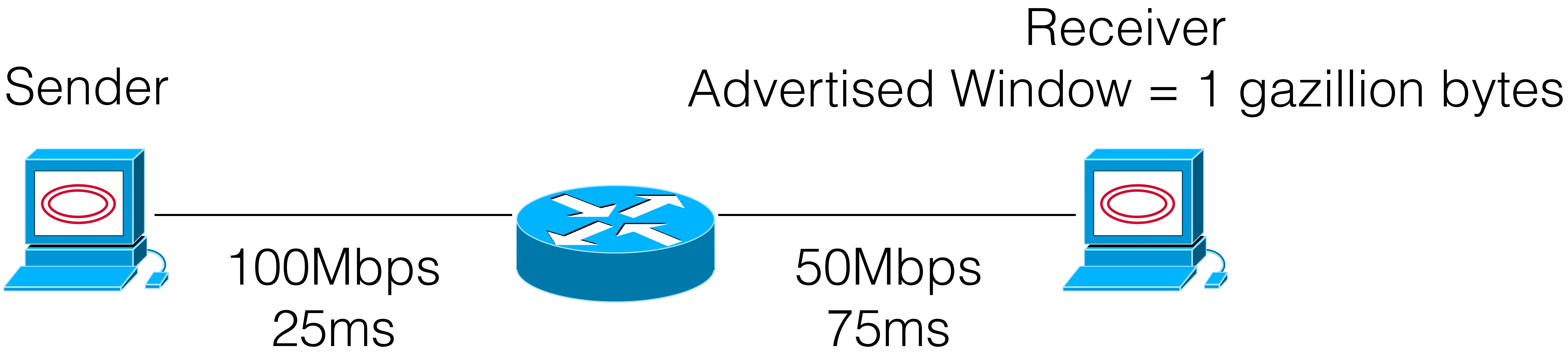
- Receiver uses an “Advertised Window” (W) to prevent sender from overflowing its window
  - Receiver indicates value of W in ACKs
  - Sender limits number of bytes it can have in flight <= W
- If I only have 10KB left in my buffer, tell the receiver in my next ACK!

How big should we make the window?

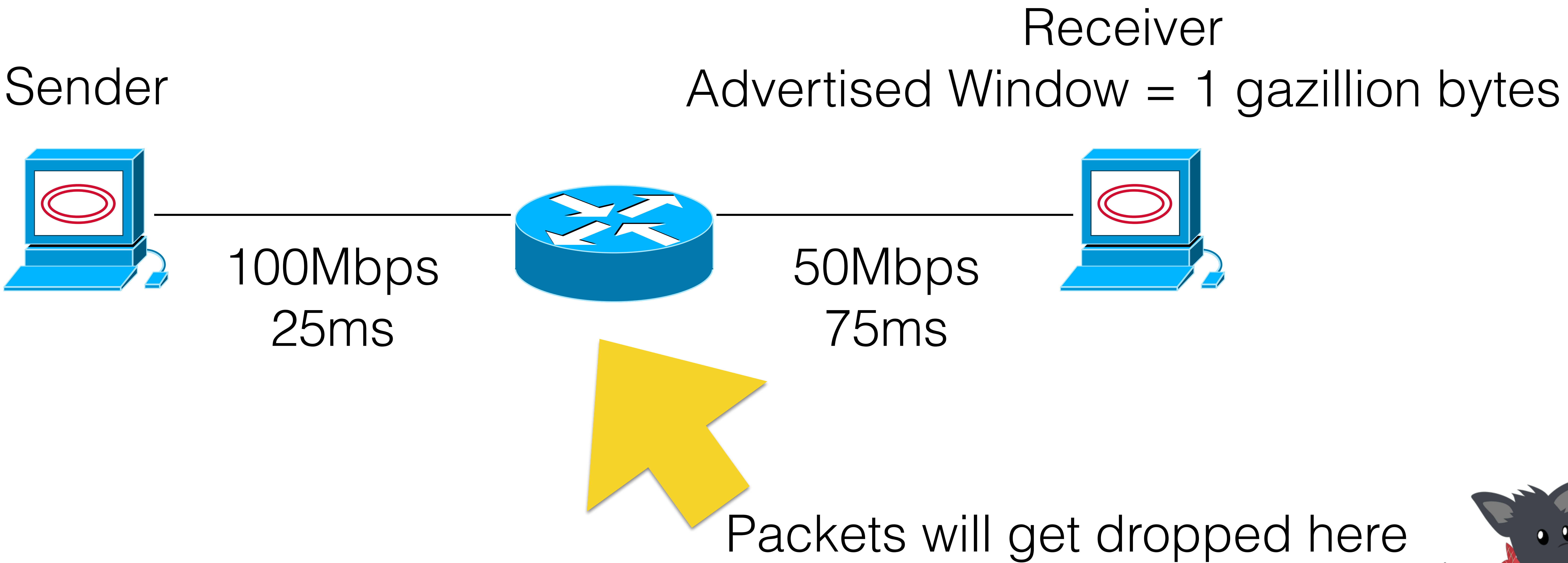
- Window should be:
    - Less than or equal to the advertised window so that we do not overload the receiver.
        - This is called Flow Control.

Alright, so let's set the window to W?

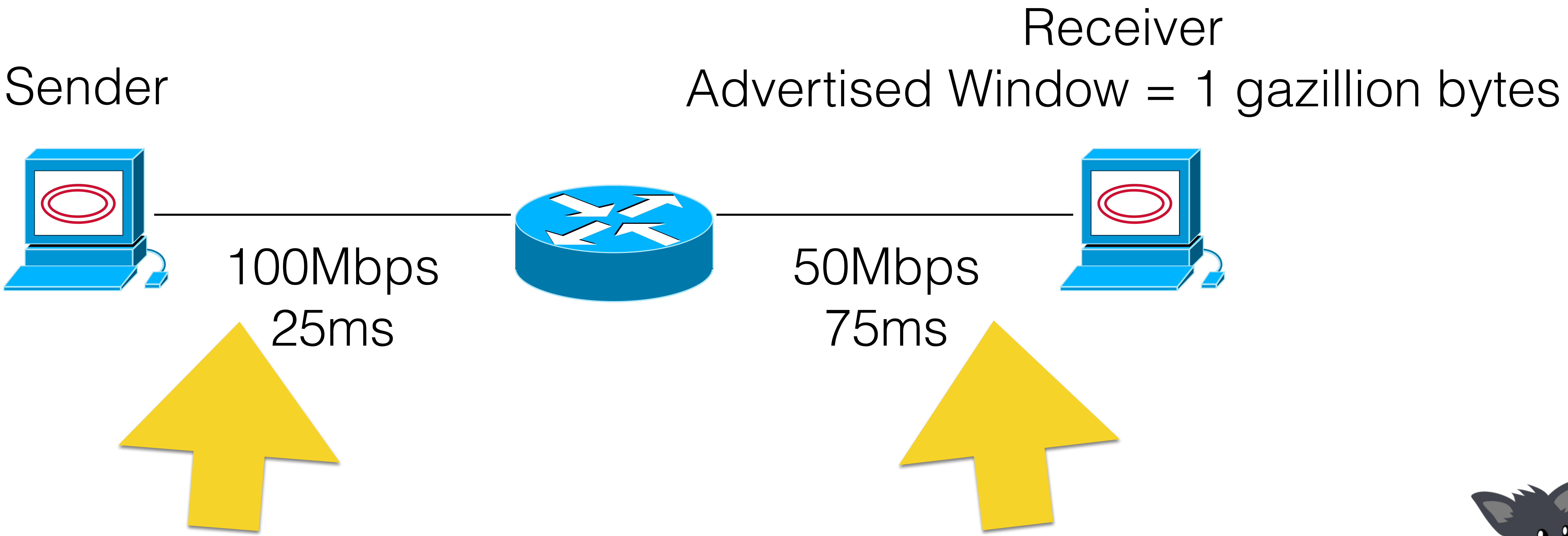
What will happen here?



What will happen here?



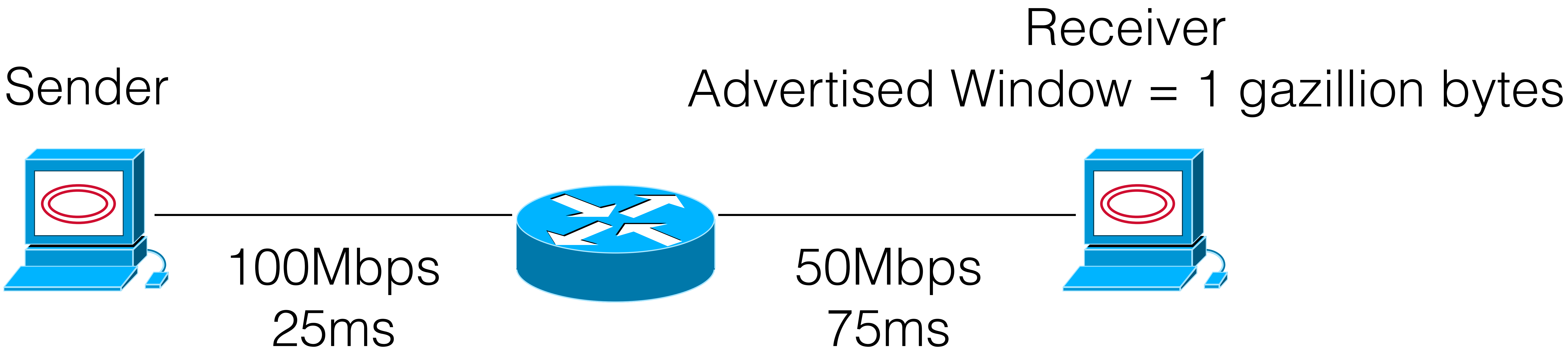
What will happen here?



Arrival rate is faster than departure rate

How big should we set the window to be?

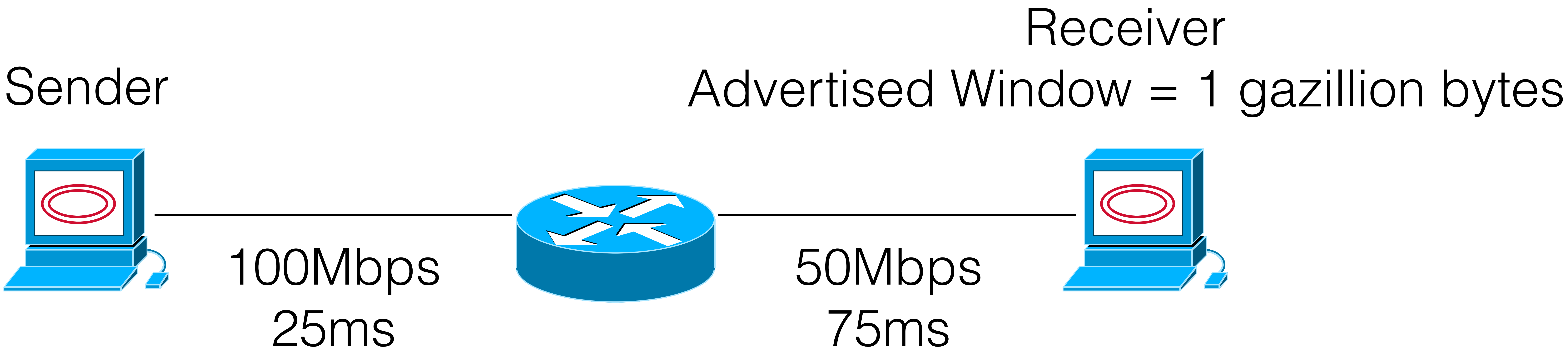
“I just want to send at 50Mbps — how does that translate into a window size?”



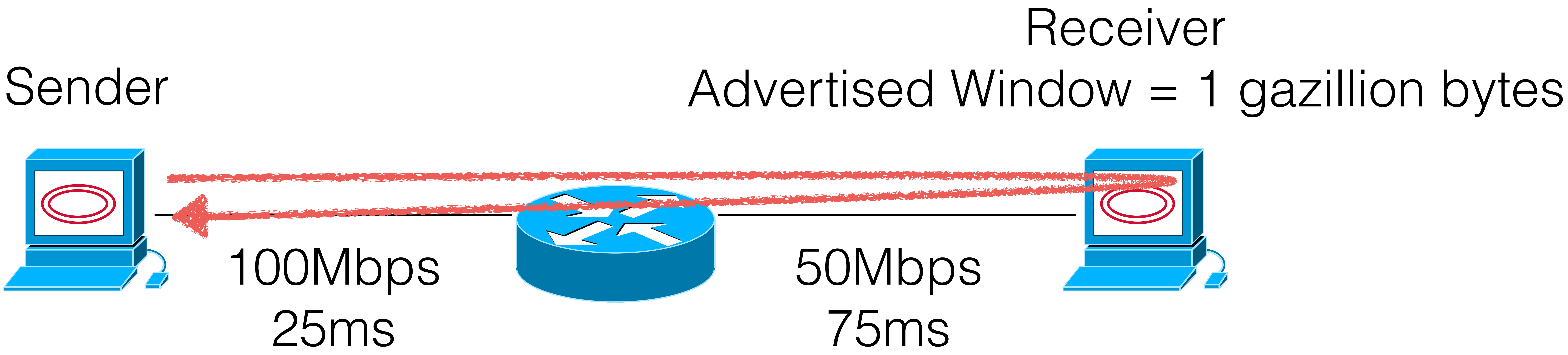
Remind me: what is the definition of a Window?

Recall: Window is the number of bytes I may have transmitted but not yet received an ACK for.

How long will it take for me to receive an ACK back for the first packet?



How long will it take for me to receive an ACK back for the first packet?



One round-trip-time (RTT) = 200 milliseconds

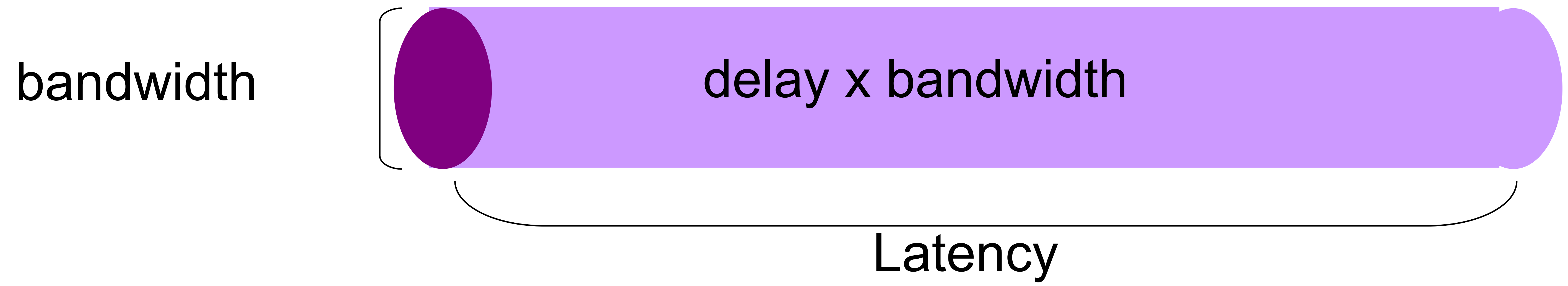
How much data will I send, at 50Mbps, in 200ms?

50Mbps * 200ms = 1.25 MB

We call this the

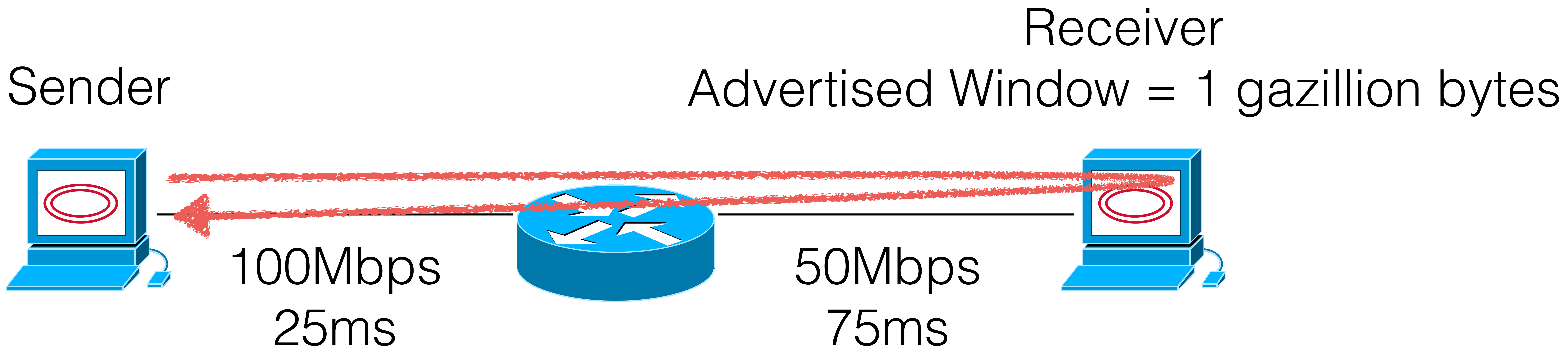
bandwidth-delay product.

Pipe Model



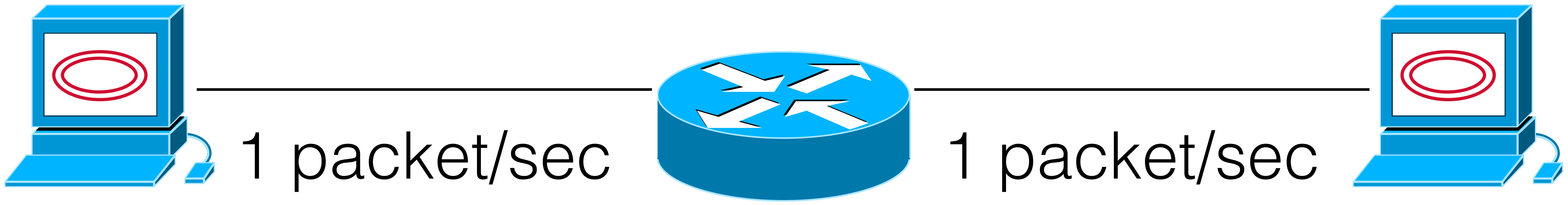
- Bandwidth-Delay Product (BDP): “volume” of the link
  - amount of data that can be “in flight” at any time
  - propagation delay × bits/time = total bits in link

When we set our window to the BDP, we get into a very convenient loop called "ACK Clocking"

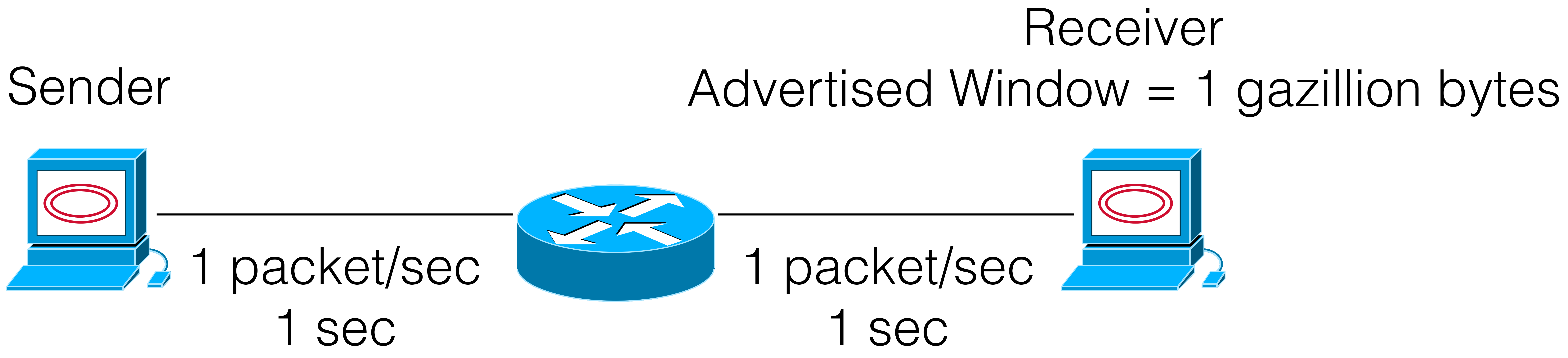


One round-trip-time (RTT) = 200 milliseconds

Yes, yet another demo….



I receive new ACKs back at *just* the right rate so that I can keep transmitting at 1 packet/sec.



How big should we make the window?

- Window should be:
    - Less than or equal to the advertised window so that we do not overload the receiver.
        - This is called Flow Control.
    - Less than or equal to the bandwidth-delay product so that we do not overload the network.
        - This is called Congestion Control.
- (That's it).

What are we missing?

How do we actually figure out the BDP?!?!

Today’s Agenda

- #1: How big should we size the window?
- **#2: How should we determine the BDP?**
- #3: How does “plain” TCP work?

Problem Constraints

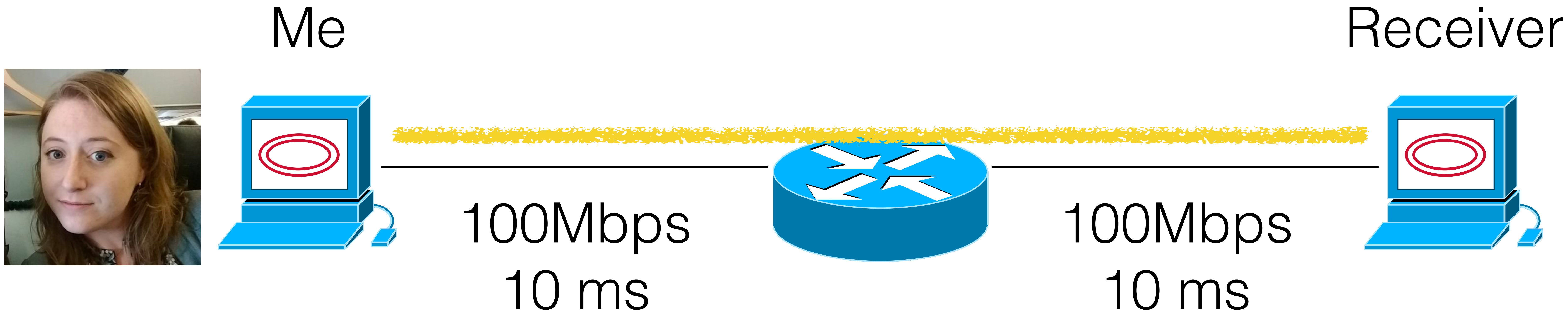
- The network does not tell us the bandwidth or the round trip time.
    - *Implication: Need to infer appropriate window size from the transmitted packets.*

Let’s make it harder…

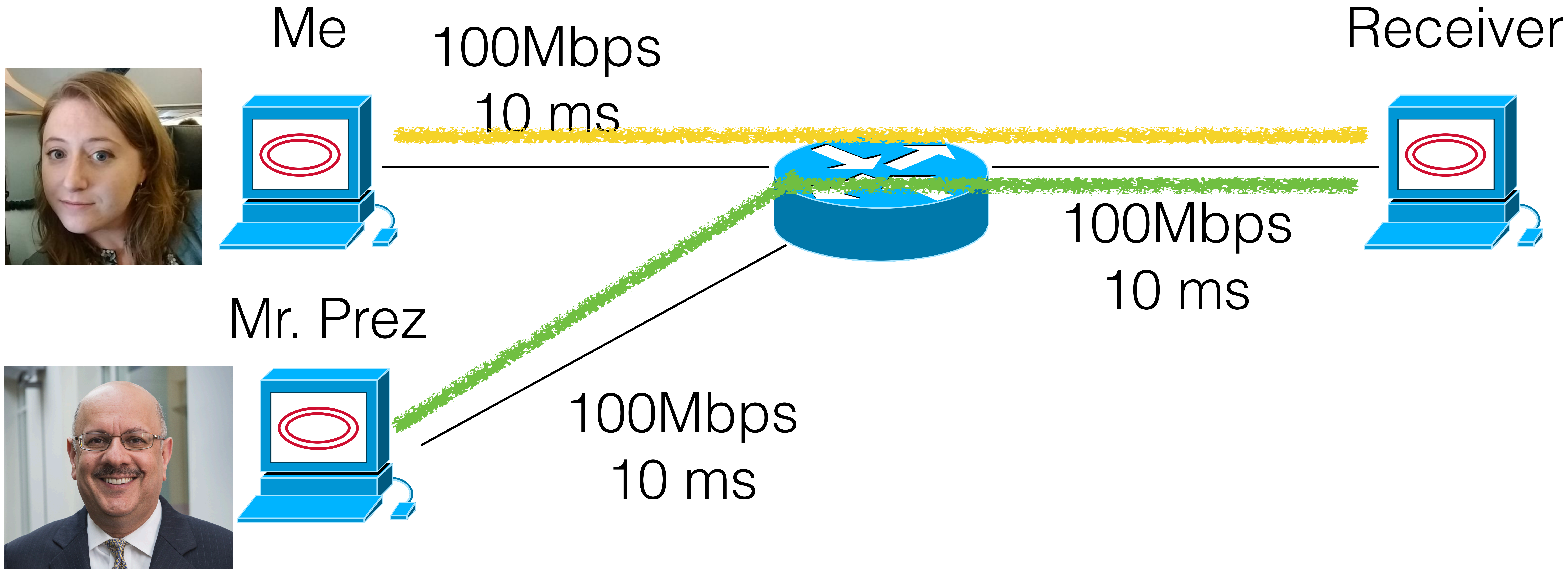
Problem Constraints

- The network does not tell us the bandwidth or the round trip time.
- My share of bandwidth is dependent on the other users on the network.

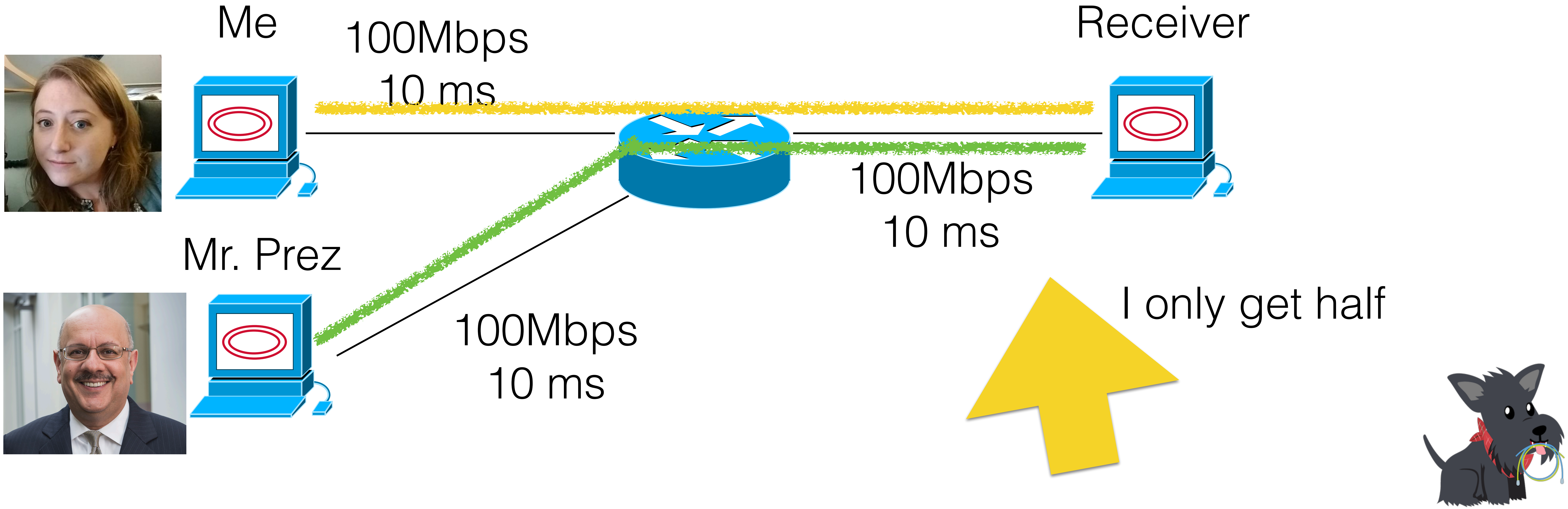
My window size: 100Mbps x 10ms



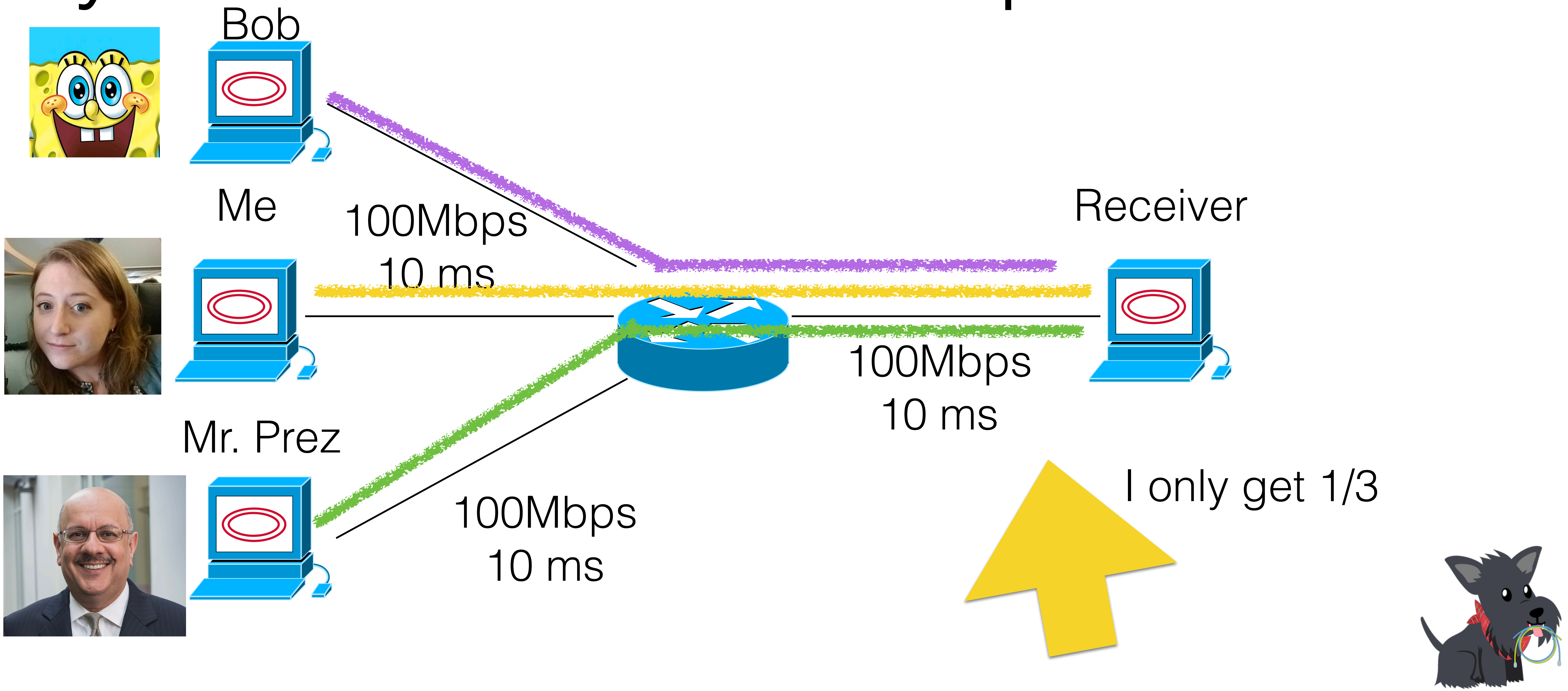
My window size: 50Mbps x 10ms



My window size: 50Mbps x 10ms



My window size: 33Mbps x 10ms



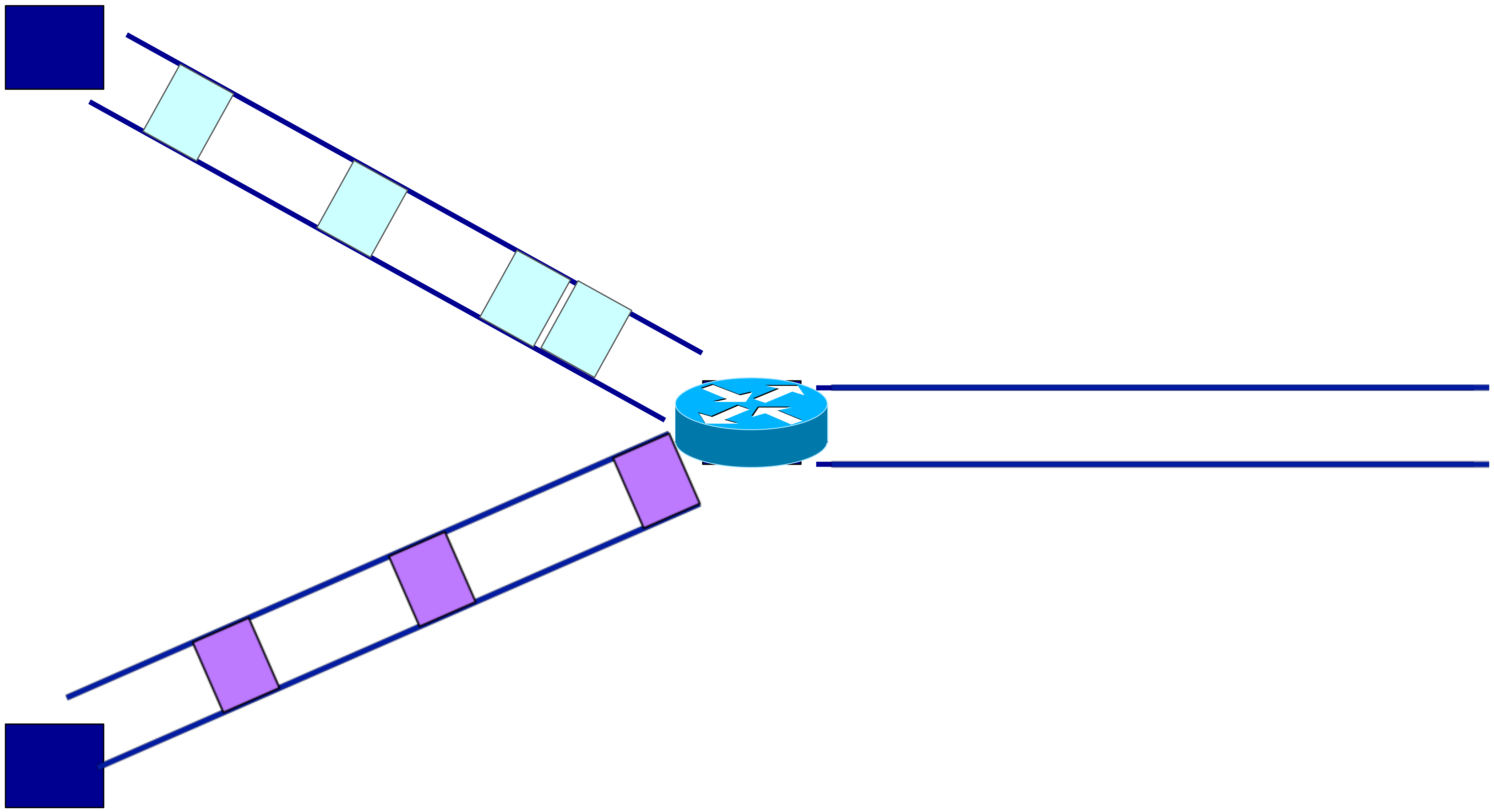
Problem Constraints

- The network does not tell us the bandwidth or the round trip time.
- My share of bandwidth is dependent on the other users on the network.
  - *Implication: my window size will change as other users start or stop sending.*

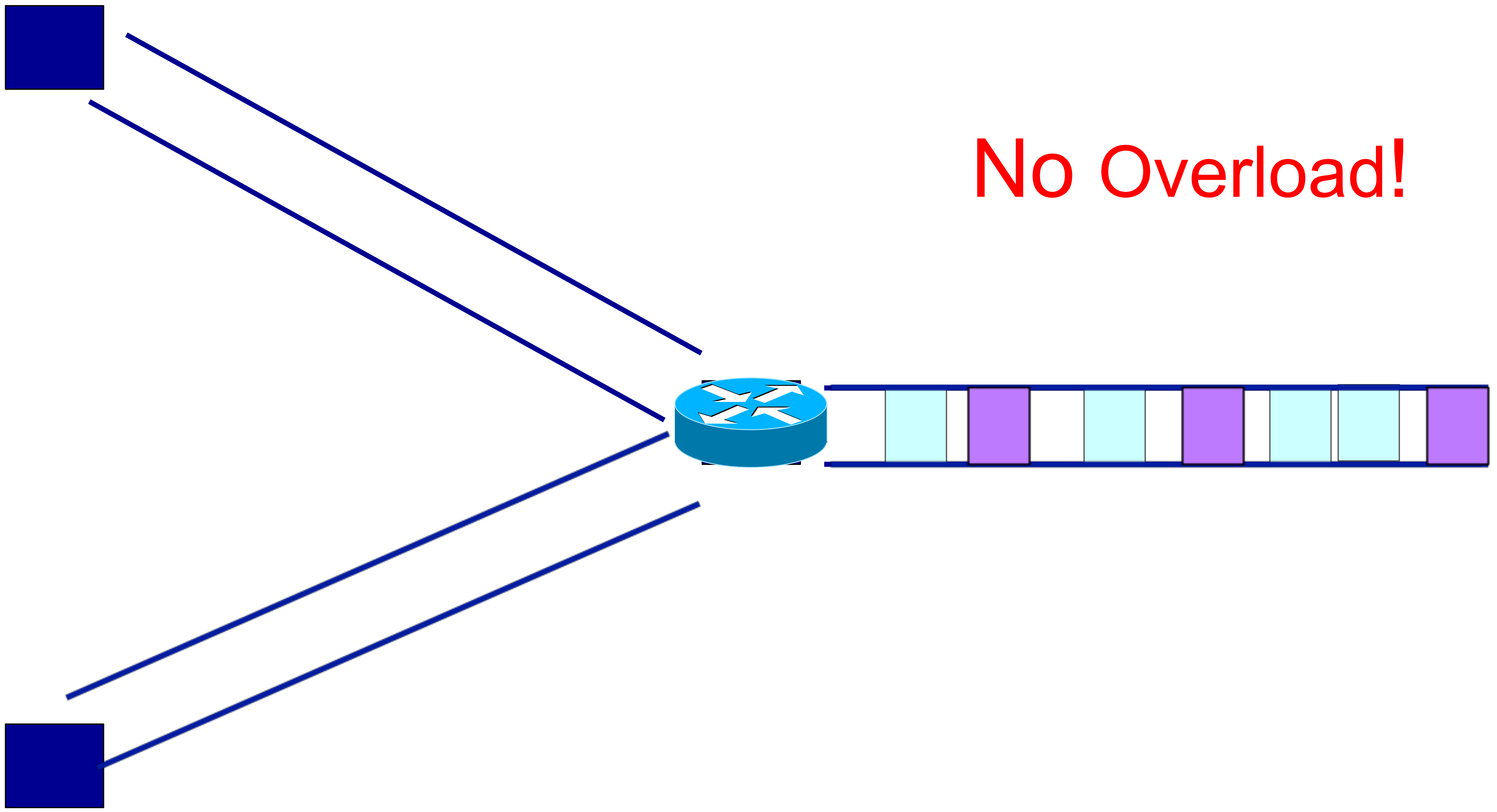
Problem Constraints

- The network does not tell us the bandwidth or the round trip time.
- My share of bandwidth is dependent on the other users on the network.
- Excess packets may not be dropped, but instead stalled in a bottleneck queue.

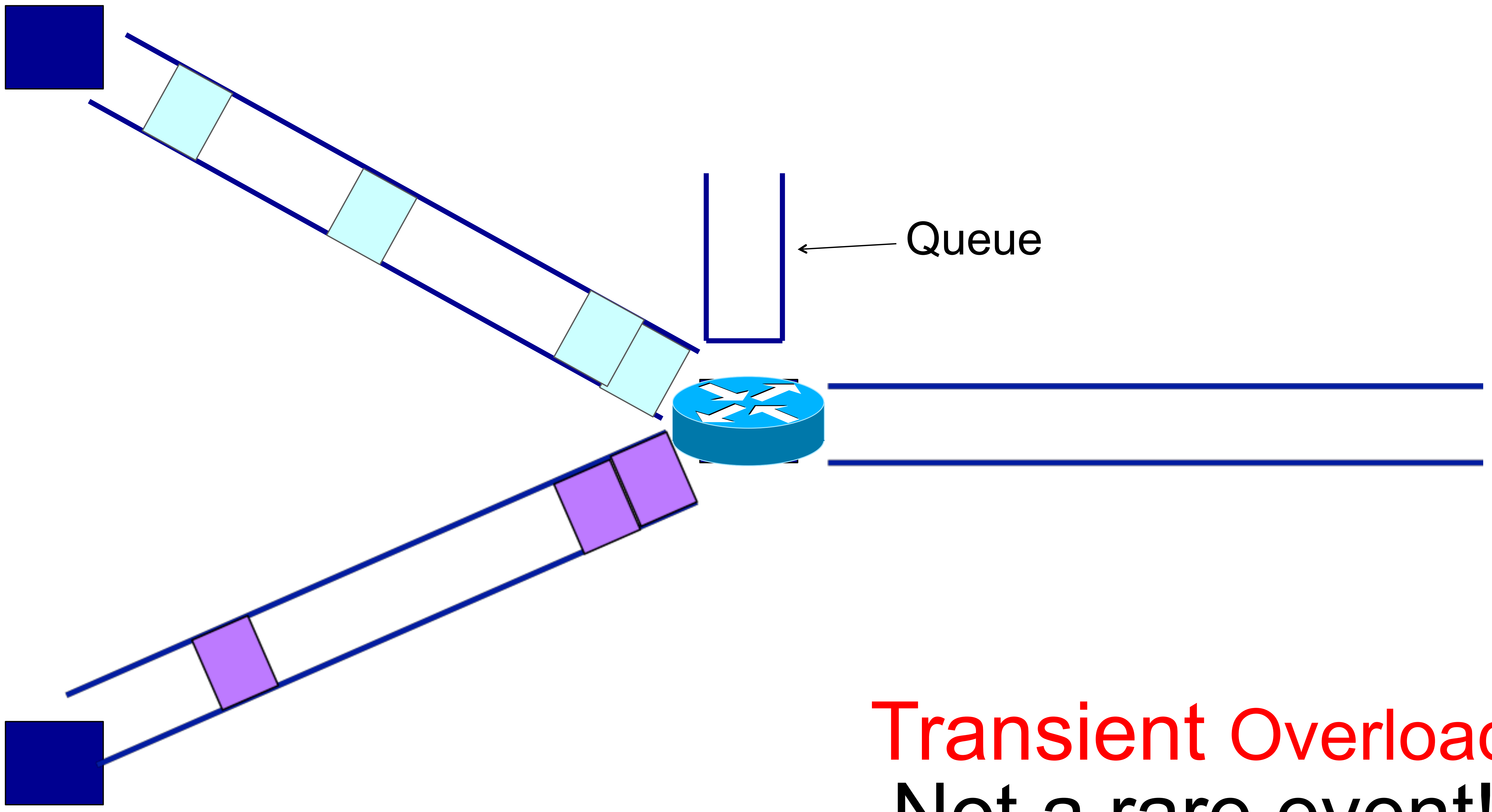
All routers have queues to avoid packet drops.



All routers have queues to avoid packet drops.

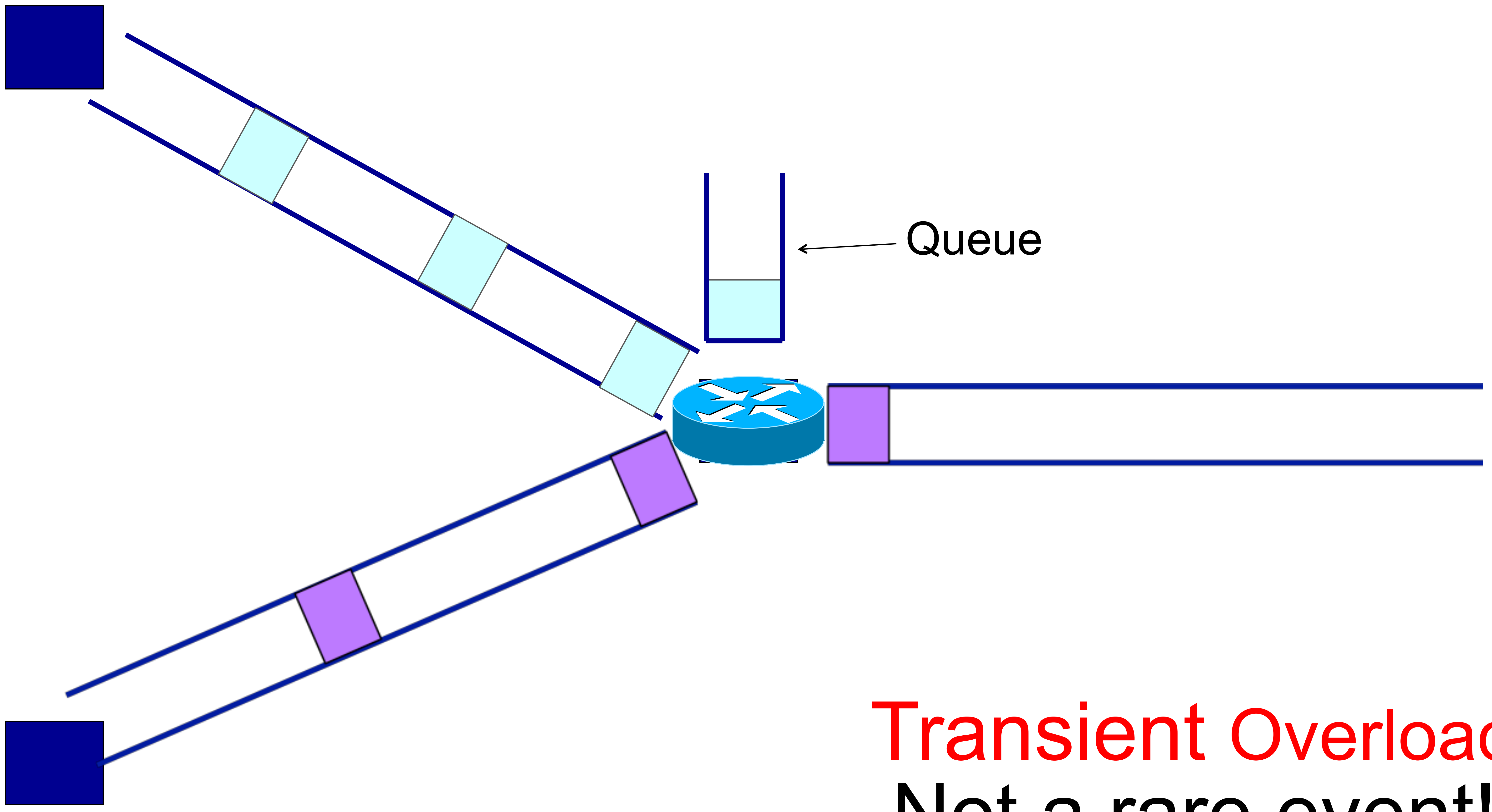


Statistical multiplexing: pipe view



Transient Overload
Not a rare event!

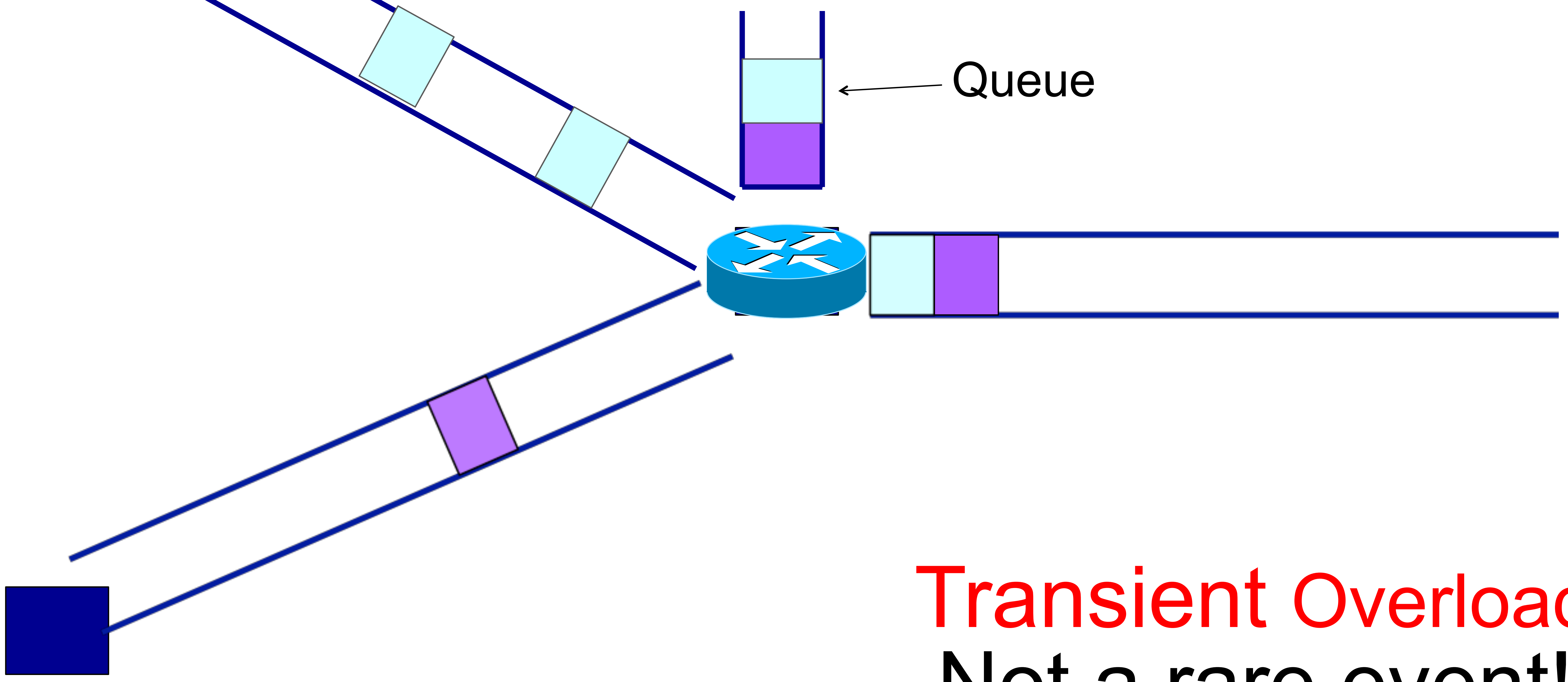
All routers have queues to avoid packet drops.



Transient Overload

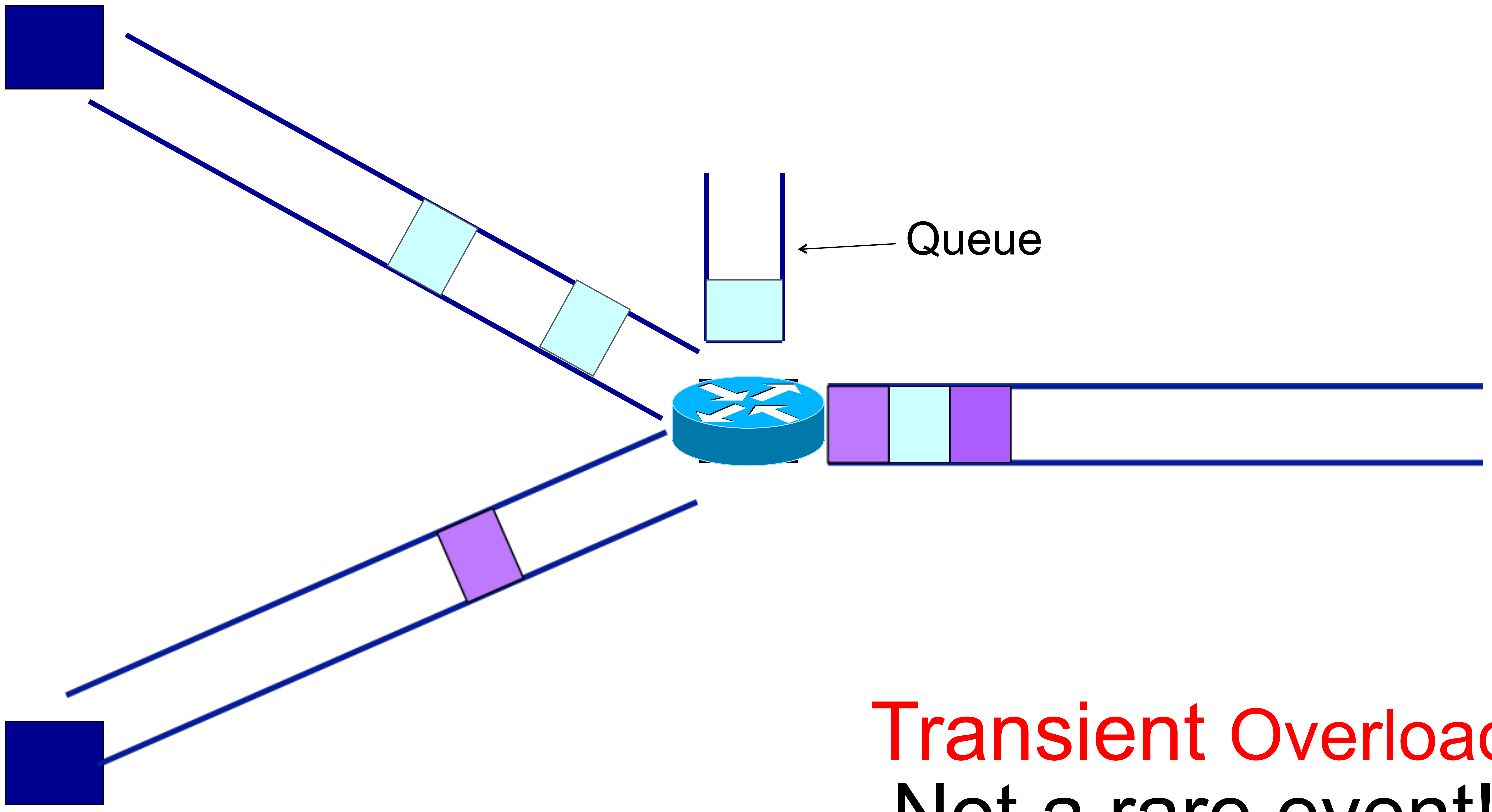
Not a rare event!

All routers have queues to avoid packet drops.



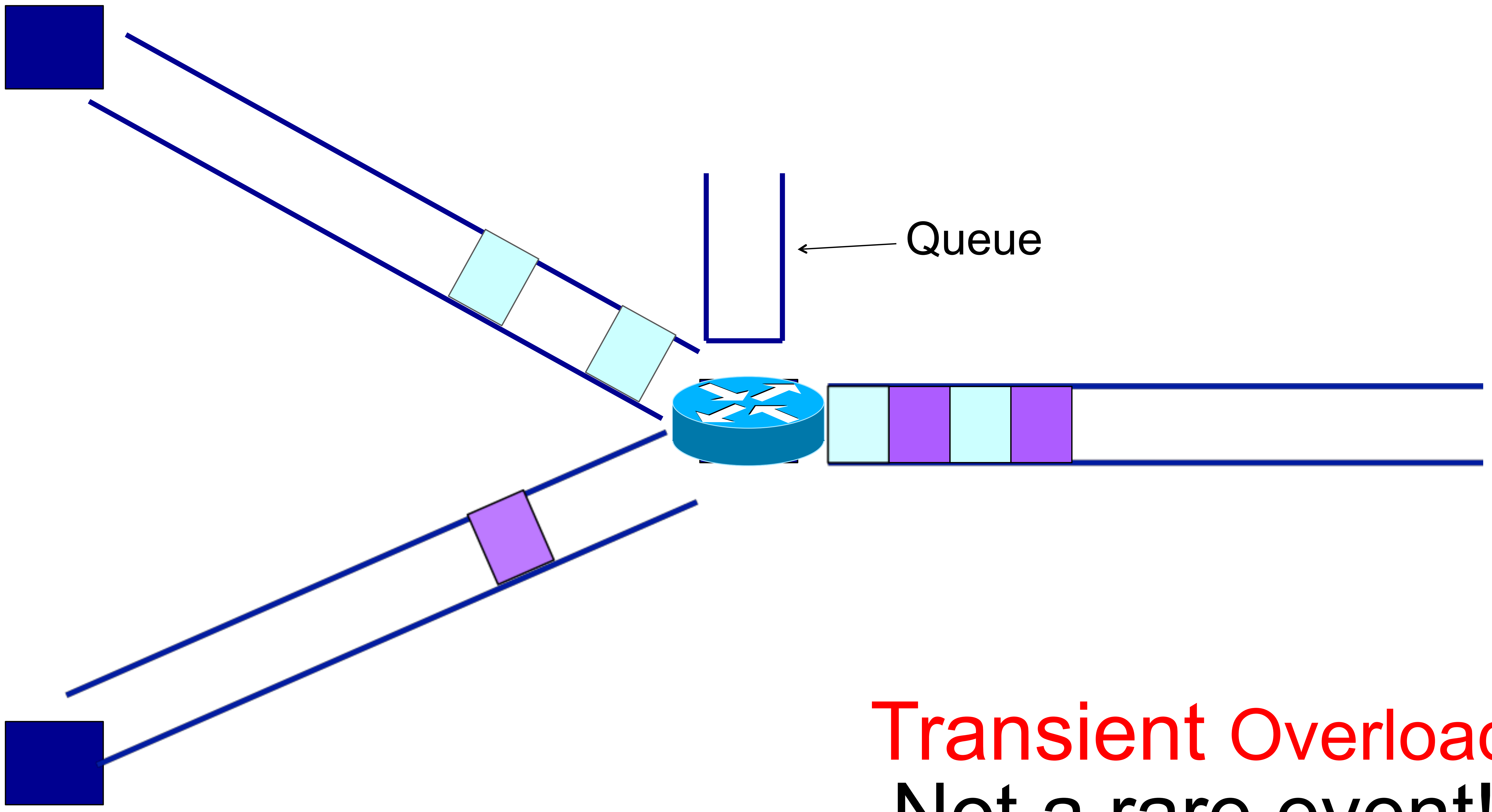
Transient Overload
Not a rare event!

All routers have queues to avoid packet drops.



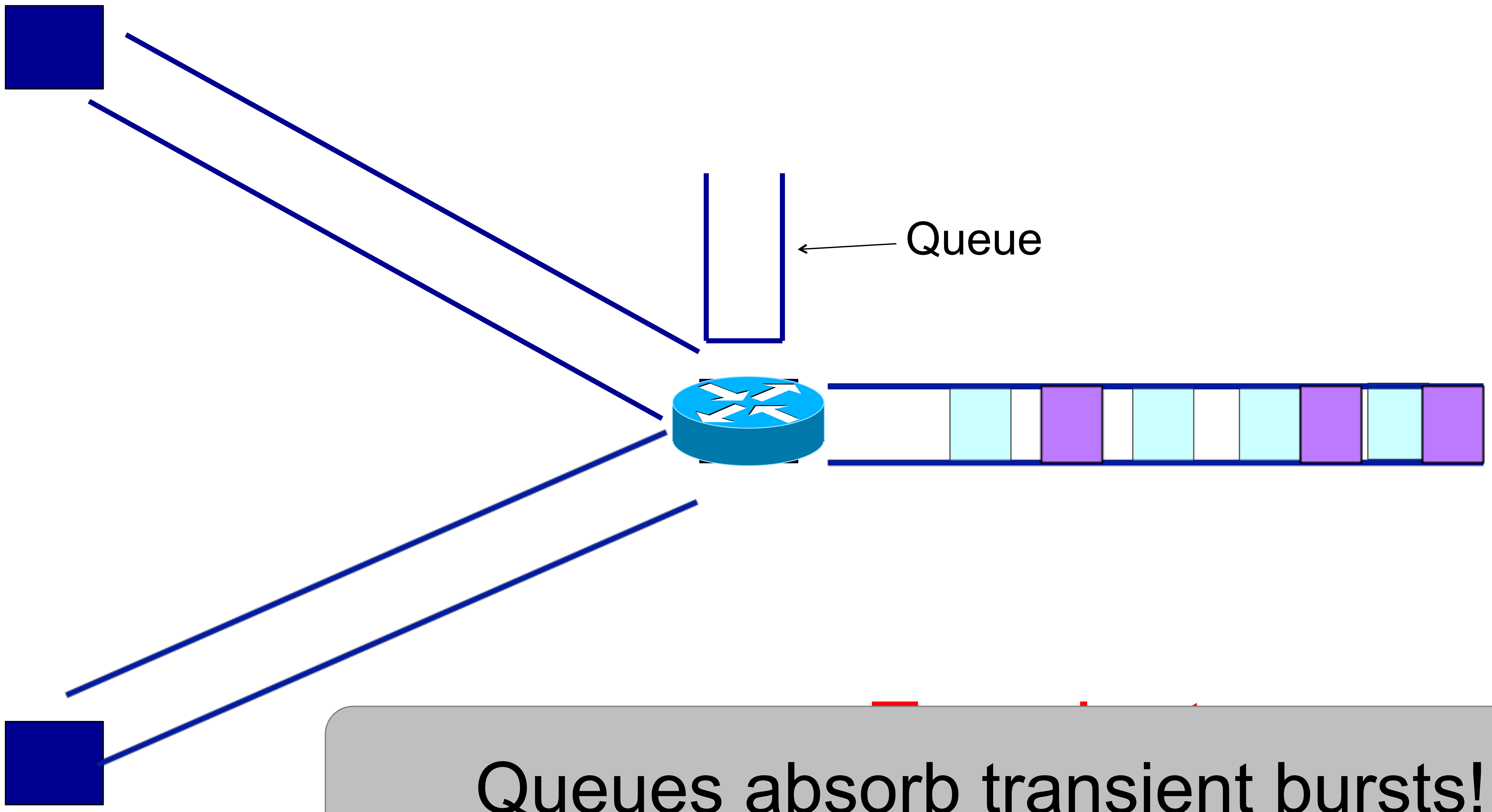
Transient Overload
Not a rare event!

All routers have queues to avoid packet drops.



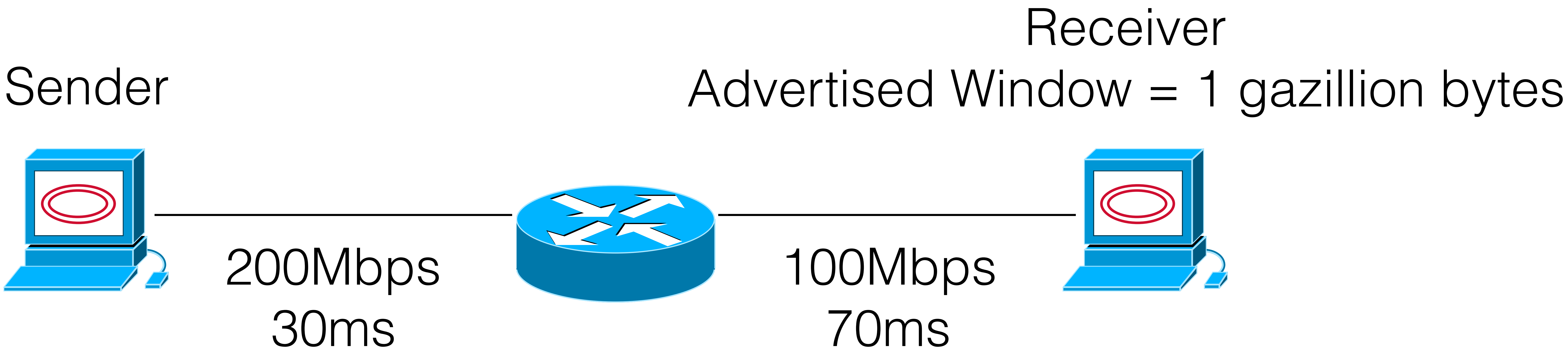
Transient Overload
Not a rare event!

All routers have queues to avoid packet drops.

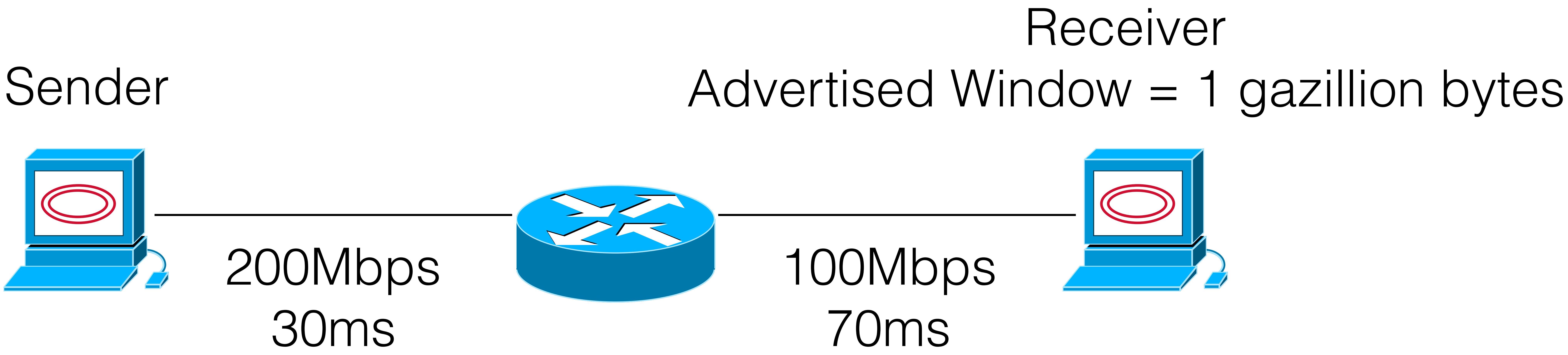


Queues absorb transient bursts!

BDP: 100Mbps * 200ms = 2.5MB

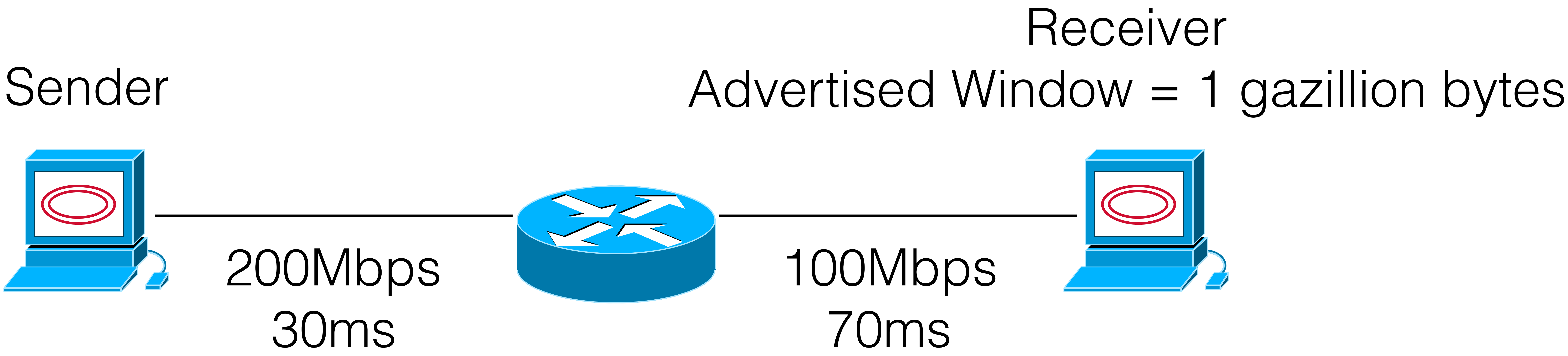


BDP: 100Mbps * 200ms = 2.5MB



If I have 1000B payloads, my window will be 2500 packets.

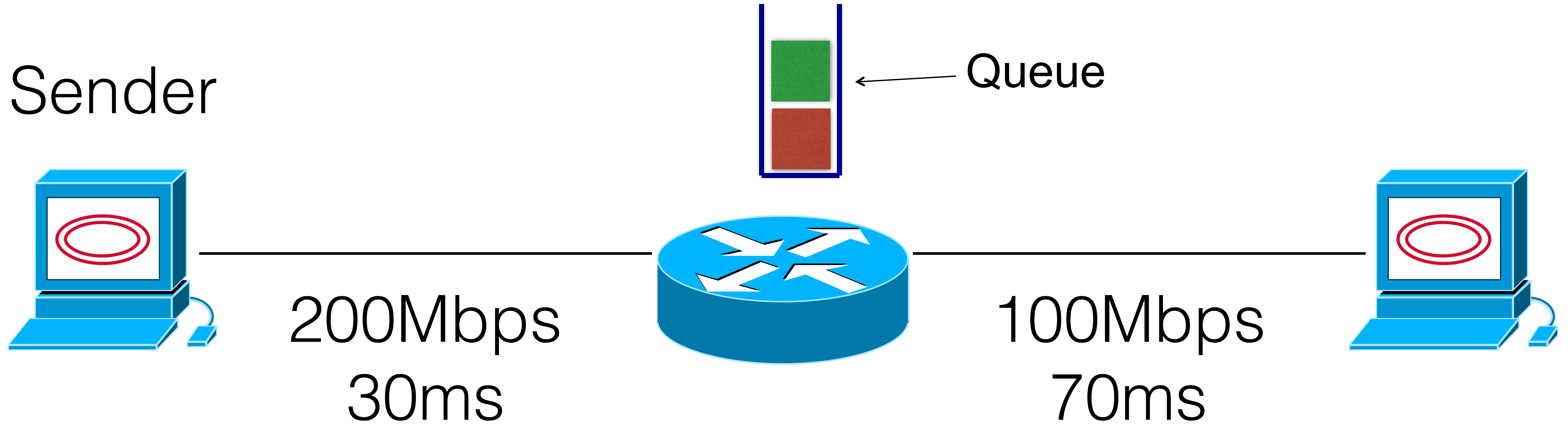
BDP: 100Mbps * 200ms = 2.5MB



Will packets get dropped if I set my window to, say, 2.6MB or 2600 packets?

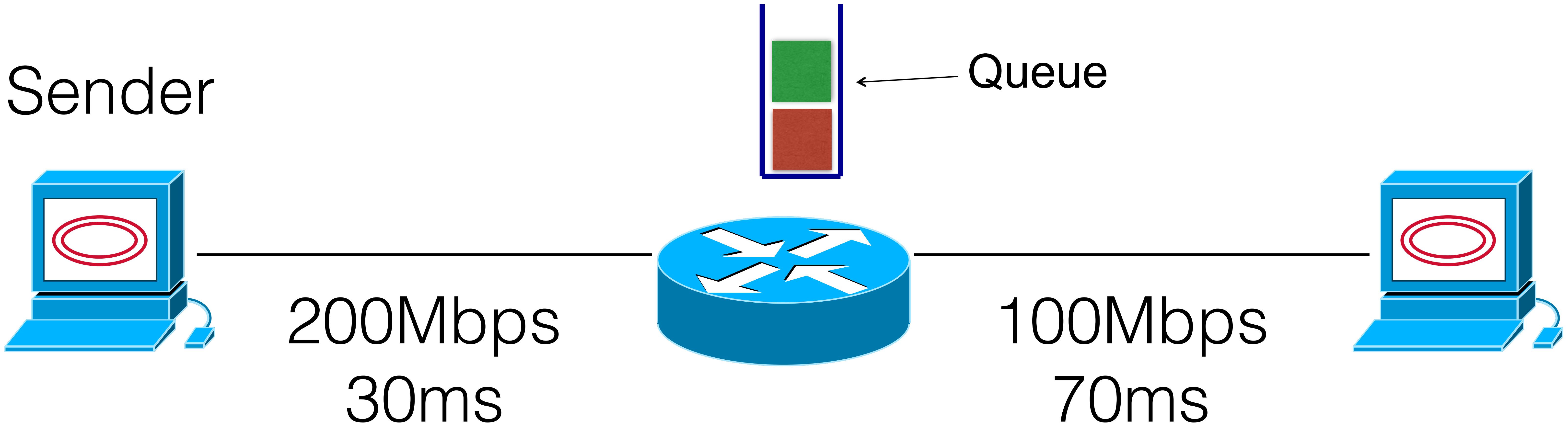
What do you think?

BDP: 100Mbps * 200ms = 2.5MB



If the queue can hold 100 more packets, none will be dropped!

BDP: 100Mbps * 200ms = 2.5MB



If the queue cannot “absorb” the extra packets, they will be dropped.

Problem Constraints

- The network does not tell us the bandwidth or the round trip time.
- My share of bandwidth is dependent on the other users on the network.
- Excess packets may not be dropped, but instead stalled in a bottleneck queue.
  - *Implication: It's okay to "overshoot" the window size, a little bit, and you still won't suffer packet loss.*

Congestion Control Algorithm: An algorithm to determine the appropriate window size, given the prior constraints.

There are *many* congestion control algorithms.

- TCP Reno and NewReno (the OG originals)
- Cubic (Linux, OSX)
- BBR (Google)
- LEDBAT (BitTorrent)
- Compound (Windows)
- FastTCP (Akamai)
- DCTCP (Microsoft Datacenters)
- TIMELY (Google Datacenters)
- Other weird stuff (ask Ranysha on Thursday)

Some History: TCP in the 1980s

- Sending rate only limited by flow control
  - Packet drops → senders (repeatedly!) retransmit a full window's worth of packets
- Led to "congestion collapse" starting Oct. 1986
  - Throughput on the NSF network dropped from 32Kbits/s to 40bits/sec
- "Fixed" by Van Jacobson's development of TCP's congestion control (CC) algorithms

Van Jacobsen

Internet Hall of Fame
Kobayashi Award
SIGCOMM Lifetime Achievement Award

- Inventor of TCP Congestion Control
- “TCP Tahoe”
- More recently, one of the co-inventors of Google’s BBR
- Author of many networking tools (traceroute, tcpdump)

LITERALLY SAVED THE INTERNET FROM COLLAPSE

Jacobson's Approach

- Extend TCP's existing window-based protocol but adapt the window size in response to congestion
    - required no upgrades to routers or applications!
    - patch of a few lines of code to TCP implementations

- A pragmatic and effective solution
    - but many other approaches exist

- Extensively improved upon
    - topic now sees less activity in ISP contexts
    - but is making a comeback in datacenter environments

The default TCP everyone teaches is TCP Reno, so that is what we will teach in this class.

* Even though Reno isn’t what Jacobsen invented.

** Even though our research at CMU suggests that it’s extinct — no one uses it anymore

*** On Thursday you’ll learn about “living” TCPs

TCP Reno: General Blueprint

- If a packet is lost, slow down! The packet is a signal that you are sending *too fast.*

- If you have been sending for a while and no packets are lost, speed up! No loss is a signal that you are probably are sending less than the link capacity.

How much should we slow down? Speed up?

- AIAD: Additive Increase, Additive Decrease
  - Every RTT, I increase my window by one. Every time I have a loss, I decrease my window by one.
- MIAD: Multiplicative Increase, Additive Decrease
  - Every RTT, I increase my window by 2x. Every time I have a loss, I decrease my window by one.
- AIMD: Additive Increase, Multiplicative Decrease
  - Every RTT, I increase my window by 1. Every time I have a loss, I decrease my window by 2x.
- MIMD: Additive Increase, Multiplicative Decrease
  - Every RTT, I increase my window by 2x. Every time I have a loss, I decrease my window by 2x.

Let’s Try It

- Turn to a partner. One of you will be “the network”, the other will be “the sender.”

- Network:
    - Choose a random number between 1 and 30. This is your BDP.
    - Every time your partner guesses, tell them “drop” if they overshoot, or “no drop” if they undershoot.
    - On a piece of paper, keep track of how many times your partner guessed, and keep track of how many packets are “lost”
        - If my partner guesses 40, and my secret number is 28, we “lost” 12 packets and transmitted 28.

- Sender:
    - Choose an algorithm (AIMD, MIMD, MIAD, or AIAD) and an *initial window size* — a random number from 1-30 that is your first window size.
    - Tell your partner “I transmit $windowsize packets”
        - Your partner will tell you whether there were dropped packets or no dropped packets.
    - Adjust your window according to the algorithm and then make another guess.

Who thinks they had a good algorithm/initial window size?

- What algorithm did you choose?
    - Why is it a good algorithm?
- What initial window size did you choose?
    - Why is it a good initial window size?

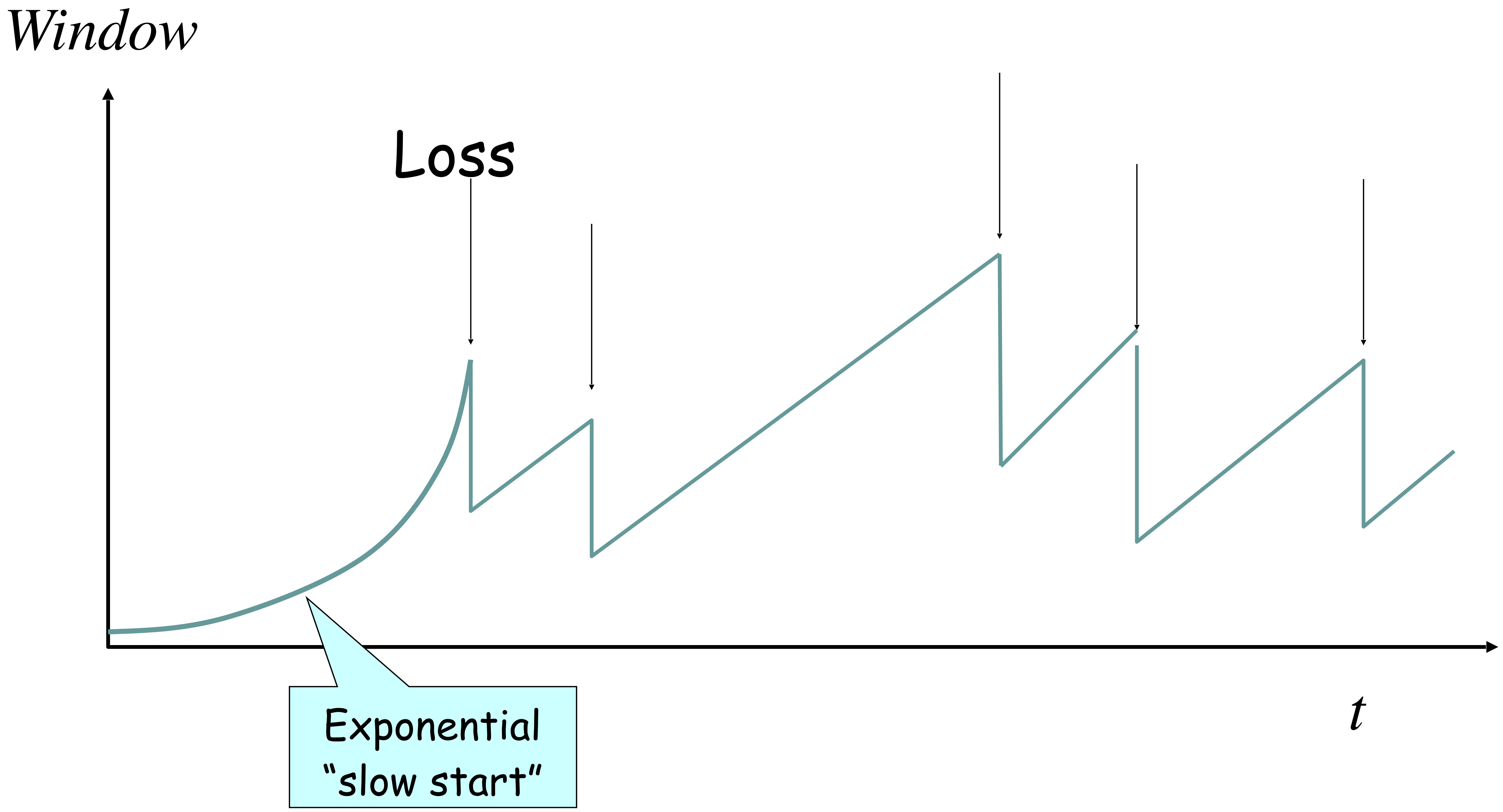
Challenges

- If you overshoot, lots of packets can be lost — for you and anyone else sharing the link!
    - Wastes network resources
    - Slows down transmission overall (have to wait for timers to go off)
    - Wastes CPU time (complicates book-keeping at sender and receiver)
- If you undershoot your transmission is slower than it could be…. :(

TCP Reno

- Uses Multiplicative Increase at startup to find the “right” sending rate quickly. Initial window size is set to 4.
    - For historical reasons this is called “slow start” — senders used to just pick an insane high initial window size and this was “slower” than that.
- Under normal operation, uses Additive Increase/Multiplicative Decrease (AIMD) to adjust the sending rate over time.

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

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

  - Or if a drop happens.

Why AIMD?

- Key idea:
  - Be cautious in consuming new resources
    - So we don’t cause another congestion collapse!
  - Be aggressive in slowing down at packet drops.
    - So we don’t cause another congestion collapse!
- Other nice properties: AIMD is guaranteed to converge to a *fair share* between two senders sharing the same link with the same RTT.
  - More on this on Thursday.

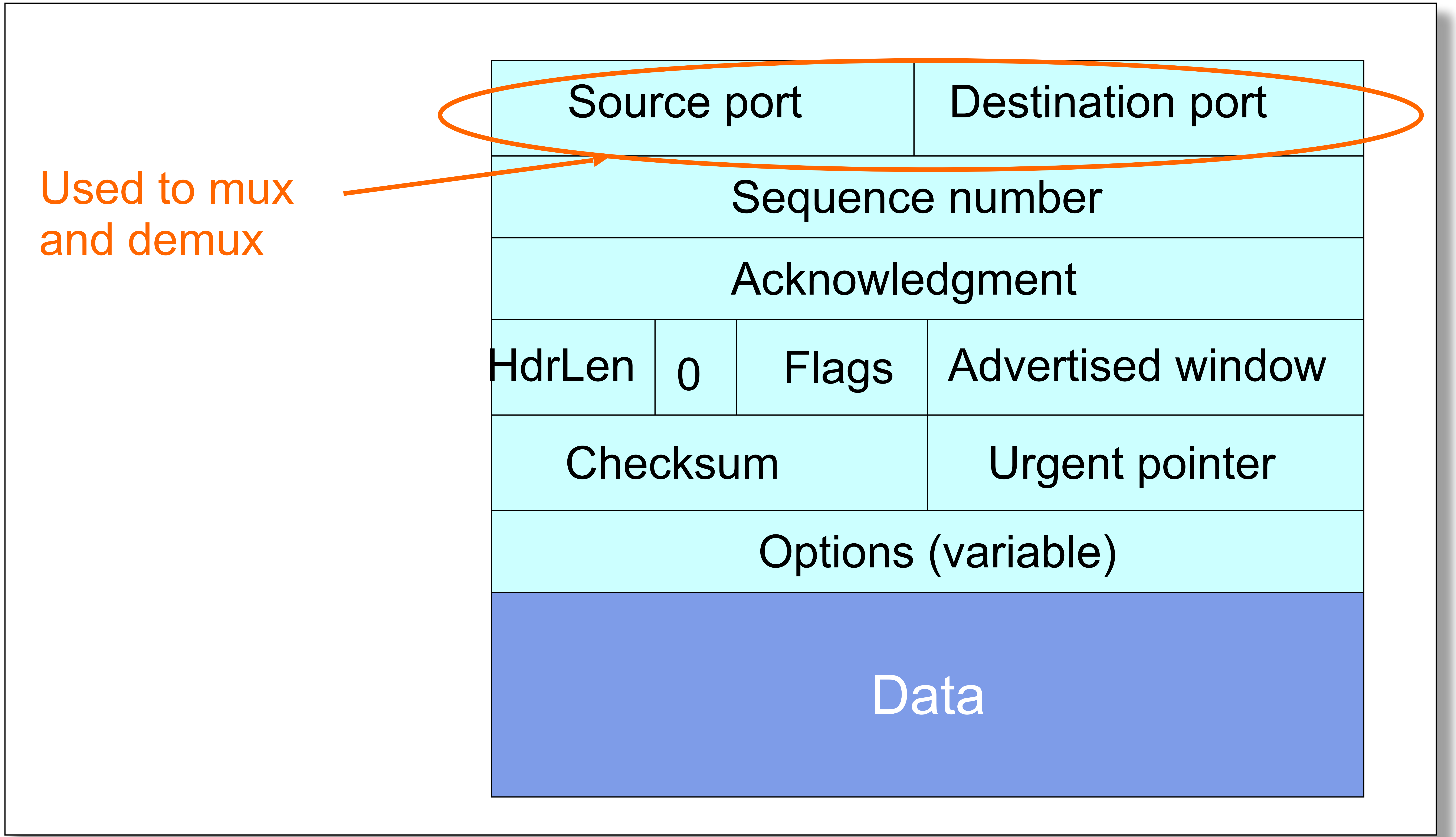
Today's Agenda

- #1: How big should we size the window?
- #2: How should we determine the BDP?
- **#3: How does "plain" TCP work?**

Today's Agenda

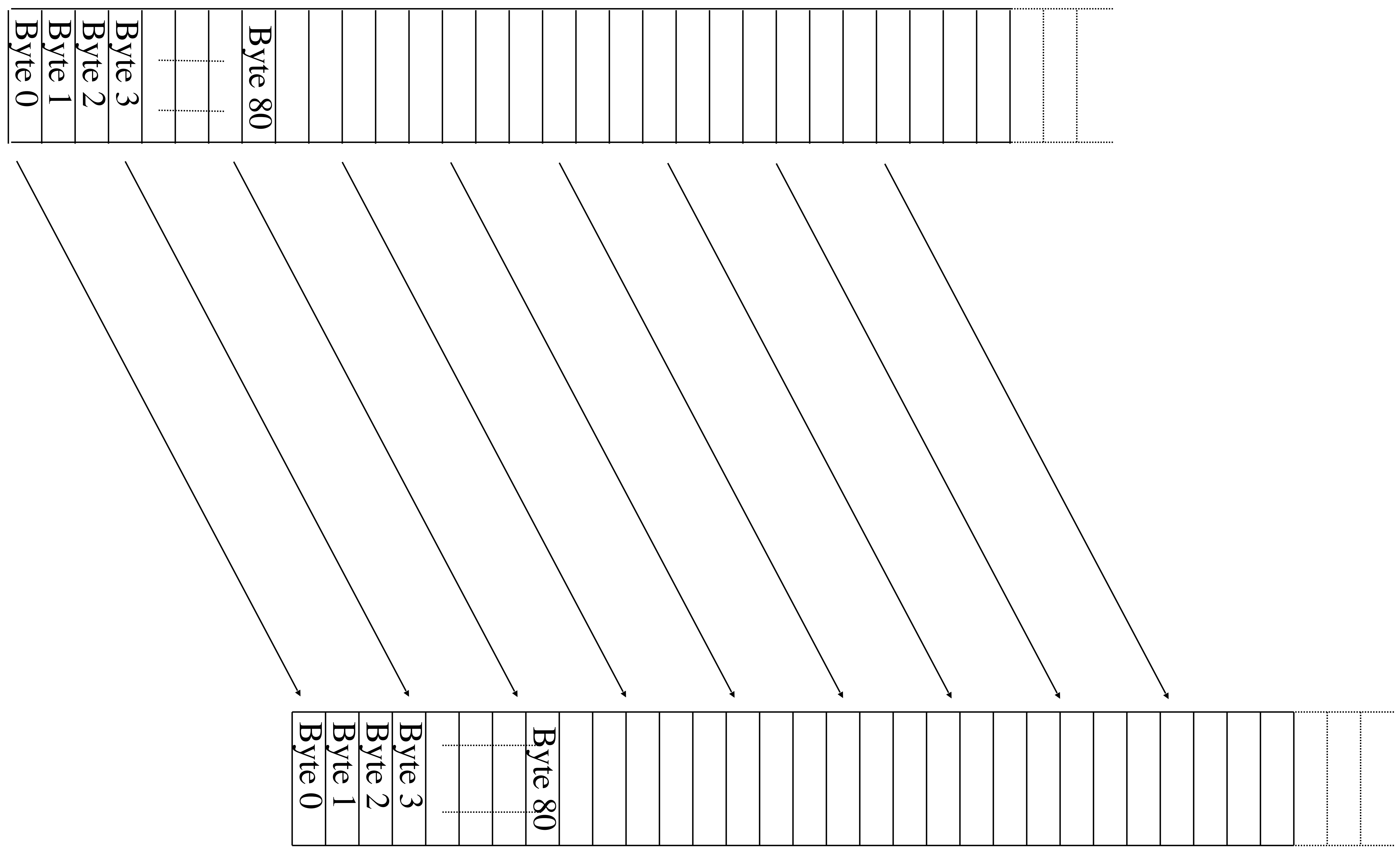
- #1: How big should we size the window?
- #2: How should we determine the BDP?
- **#3: How does "~~plain~~" TCP Reno work?**

TCP Header


Used to mux
and demux
Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

TCP “Stream of Bytes” Service…

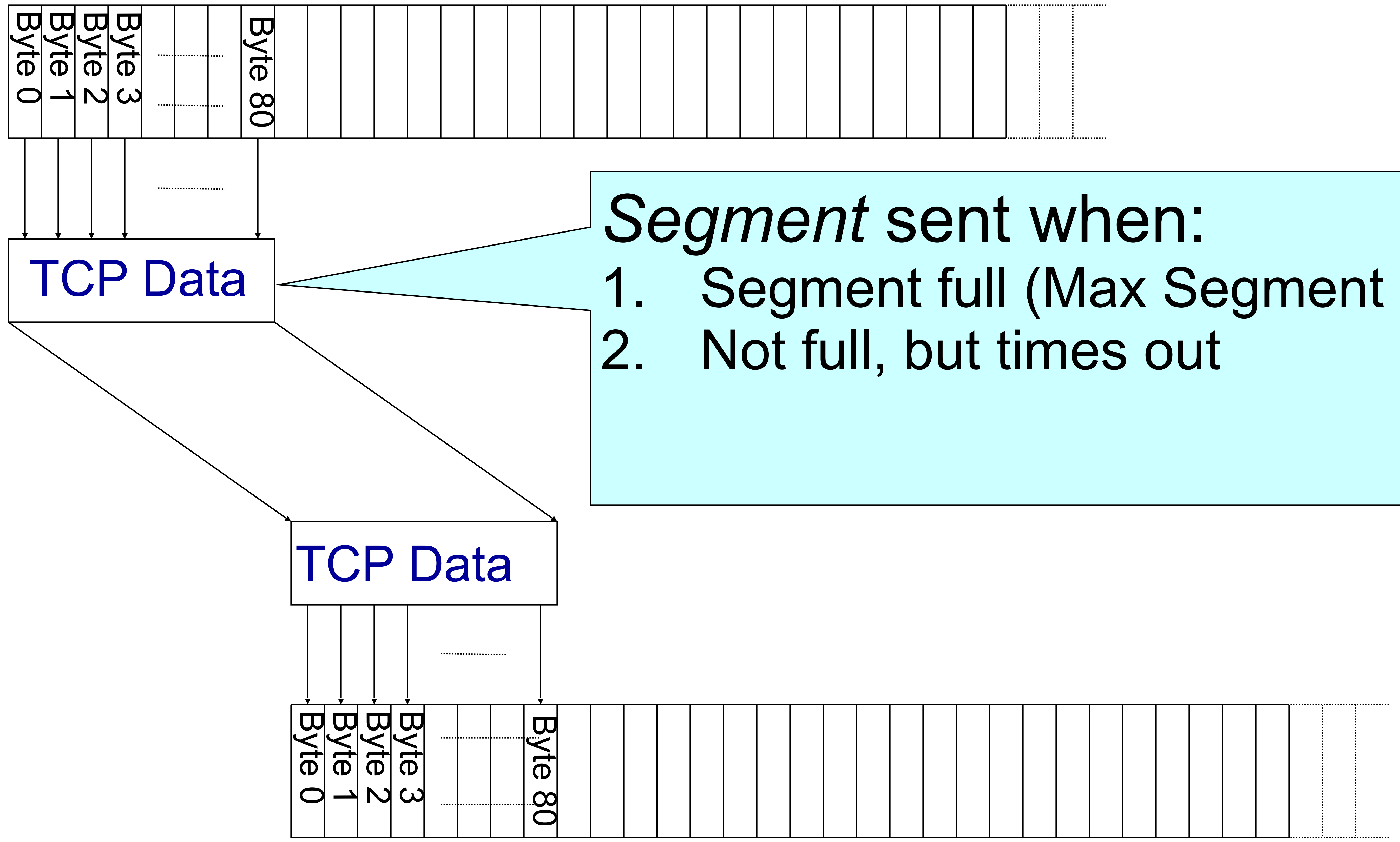
Application @ Host A



Application @ Host B

… Provided Using TCP "Segments"

Host A

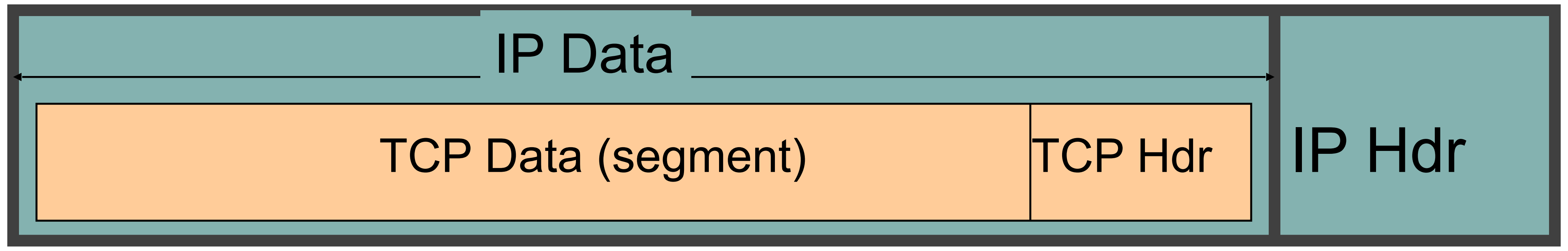


Segment sent when:

1. Segment full (Max Segment Size),
2. Not full, but times out

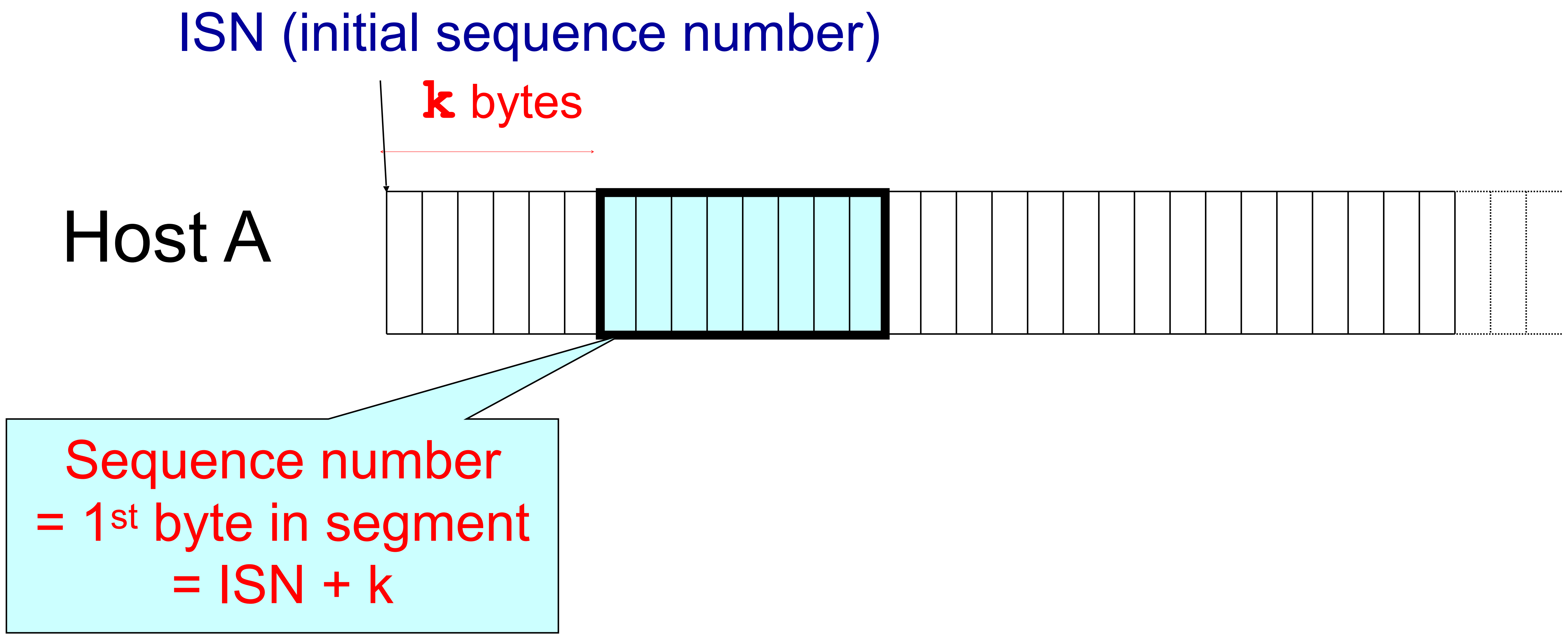
Host B

TCP Segment

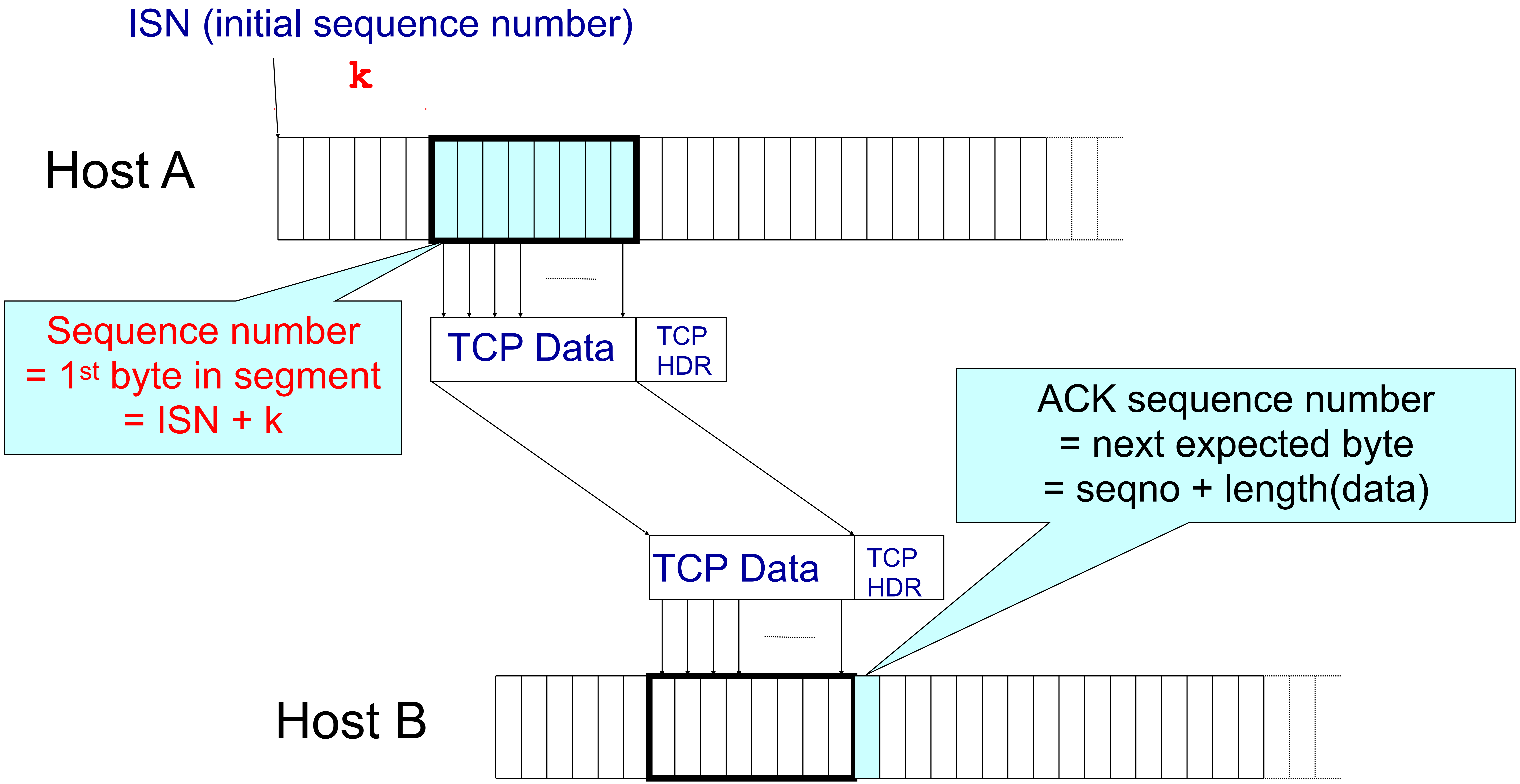


- IP packet
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes with Ethernet
- TCP packet
  - IP packet with a TCP header and data inside
  - TCP header ≥ 20 bytes long
- TCP **segment**
  - No more than Maximum Segment Size (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream
  - MSS = MTU – (IP header) – (TCP header)

Sequence Numbers

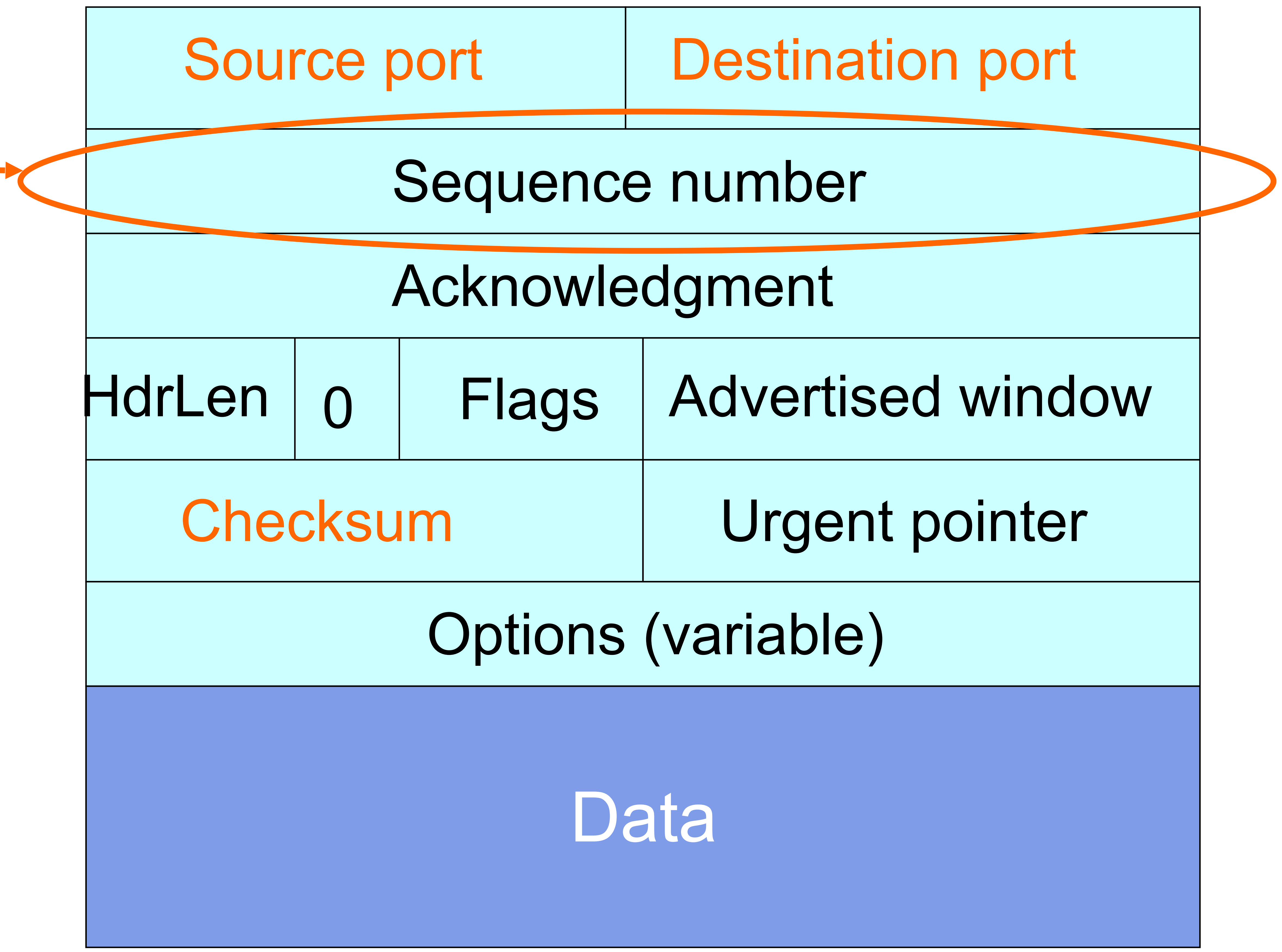


Sequence Numbers



TCP Header

Starting byte offset of data carried in this segment



TCP Header

Acknowledgment gives seqno just beyond highest seqno received **in order**
("What Byte is Next")

Remember: CUMULATIVE — this means I have every byte before this sequence number

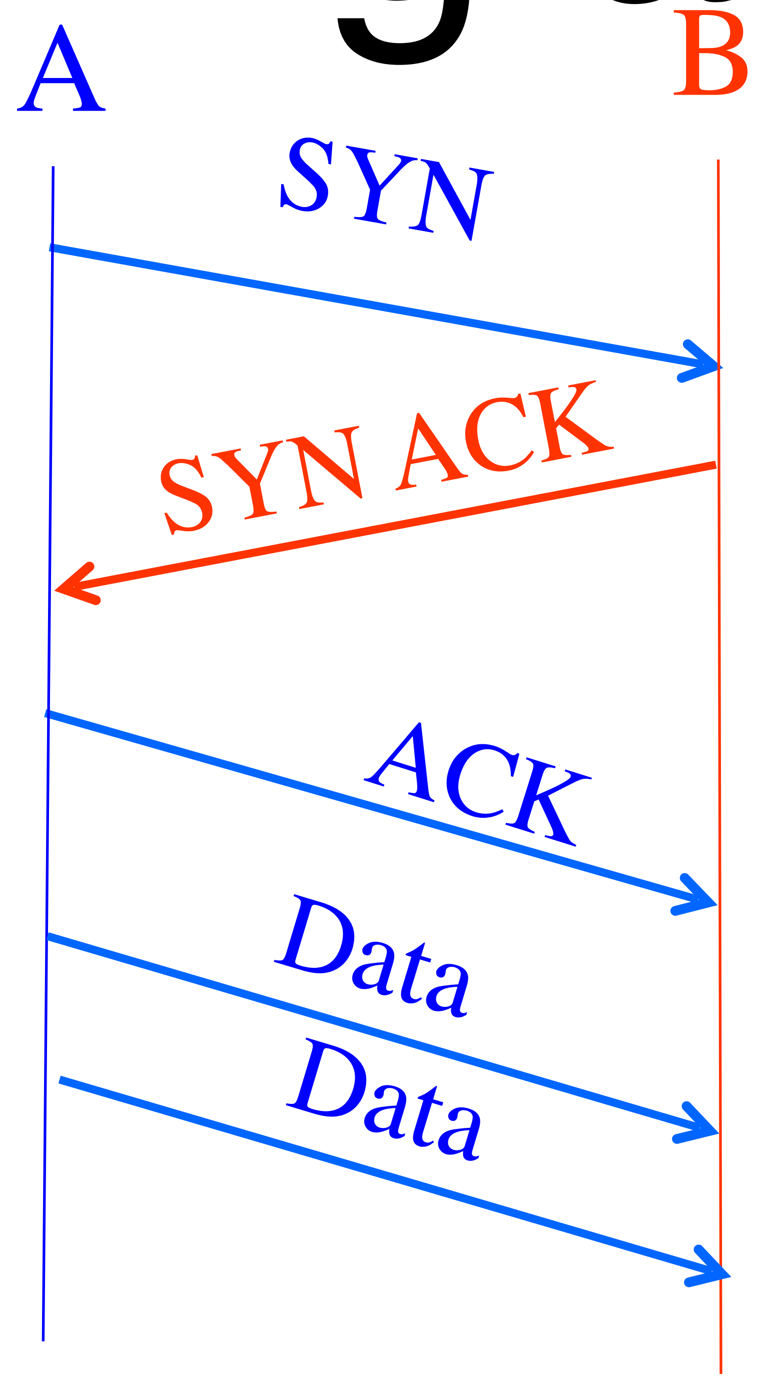
Source port		Destination port	
Sequence number			
Acknowledgment			
HdrLen	0	Flags	Advertised window
Checksum		Urgent pointer	
Options (variable)			
Data			

TCP Connection Establishment and Initial Sequence Numbers

Initial Sequence Number (ISN)

- Sequence number for the very first byte
- Why not just use ISN = 0?
- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - … small chance an old packet is still in flight
- TCP therefore requires changing ISN
- Hosts exchange ISNs when they establish a connection

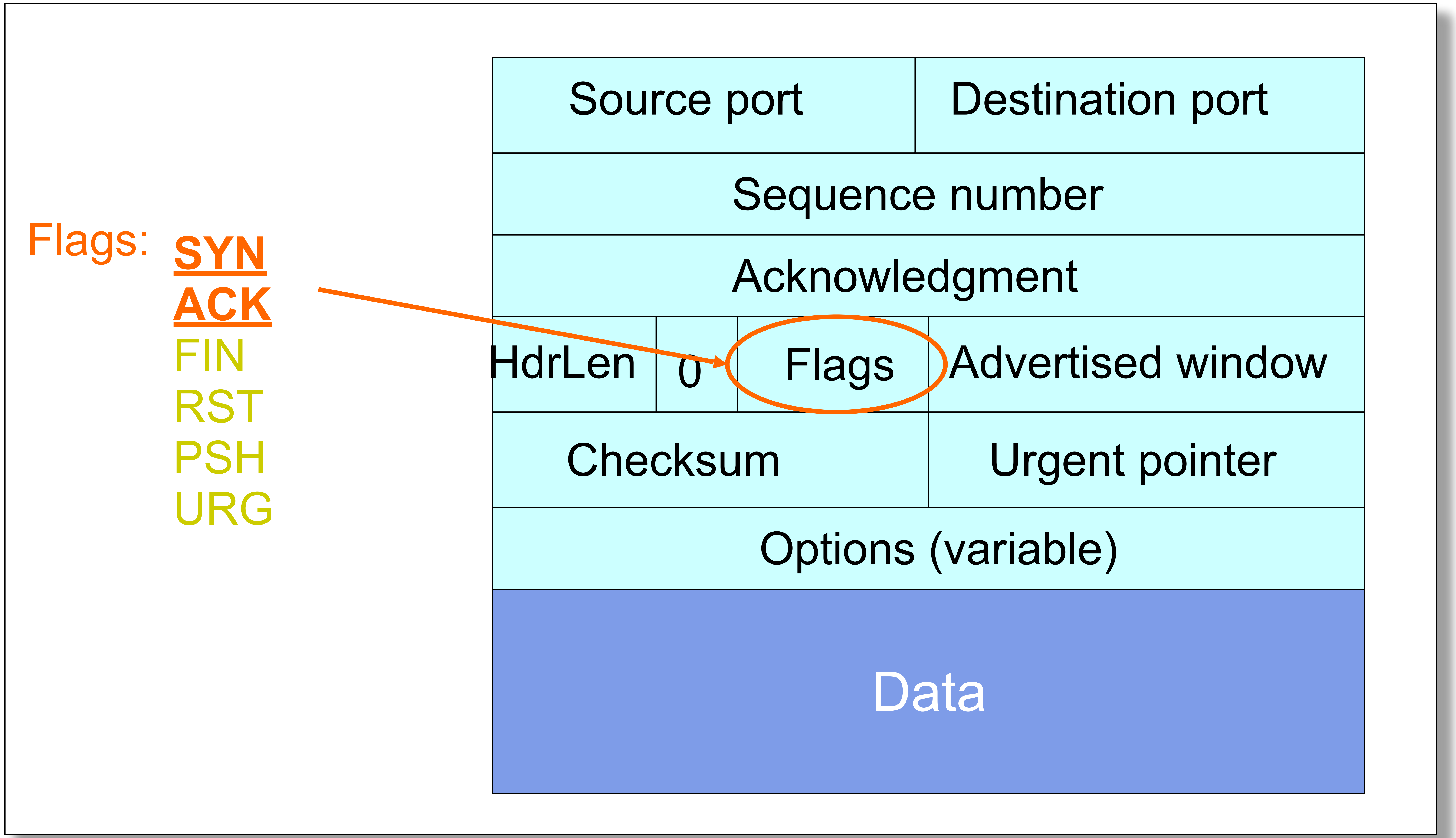
Establishing a TCP Connection



Each host tells its ISN to the other host.

- Three-way handshake to establish connection
  - Host A sends a **SYN** (open; “synchronize sequence numbers”) to host B
  - Host B returns a SYN acknowledgment (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK

TCP Header


Flags: SYN
ACK
FIN
RST
PSH
URG
Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

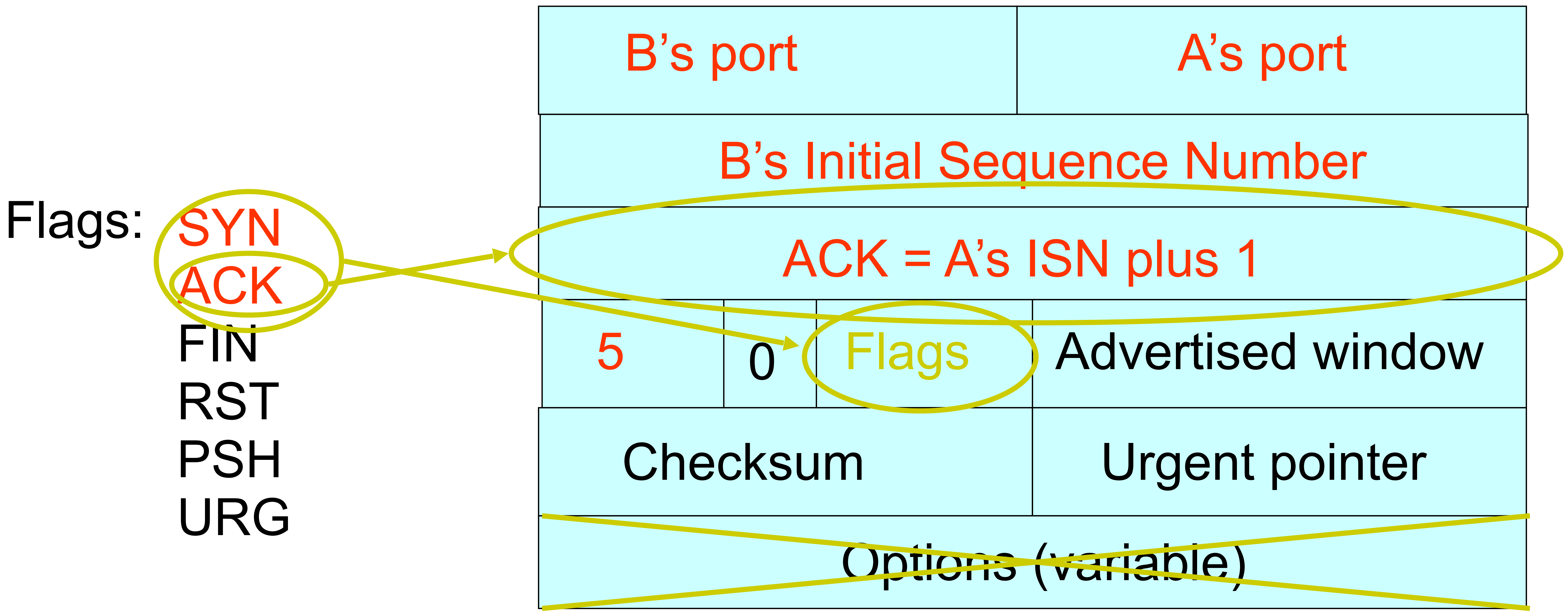
Step 1: A's Initial SYN Packet

Flags: SYN
ACK
FIN
RST
PSH
URG

A's port		B's port	
A's Initial Sequence Number			
(Irrelevant since ACK not set)			
5	0	Flags	Advertised window
Checksum		Urgent pointer	
~~Options (variable)~~			

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

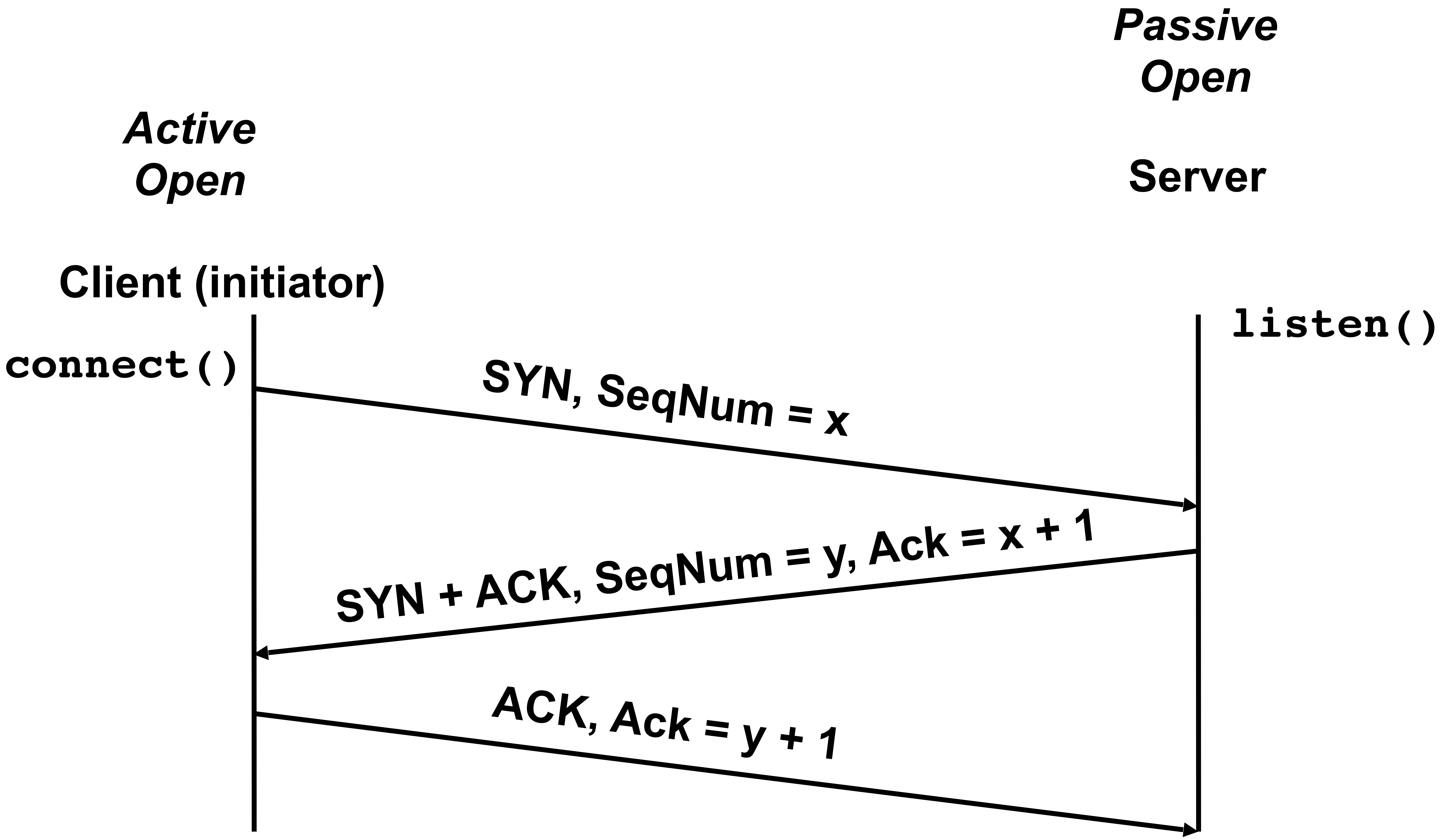
Flags: SYN
ACK
FIN
RST
PSH
URG

A's port			B's port
A's Initial Sequence Number			
B's ISN plus 1			
20B	0	Flags	Advertised window
Checksum			Urgent pointer
~~Options (variable)~~			

A tells B it's likewise okay to start sending

… upon receiving this packet, B can start sending data

Timing Diagram: 3-Way Handshaking



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

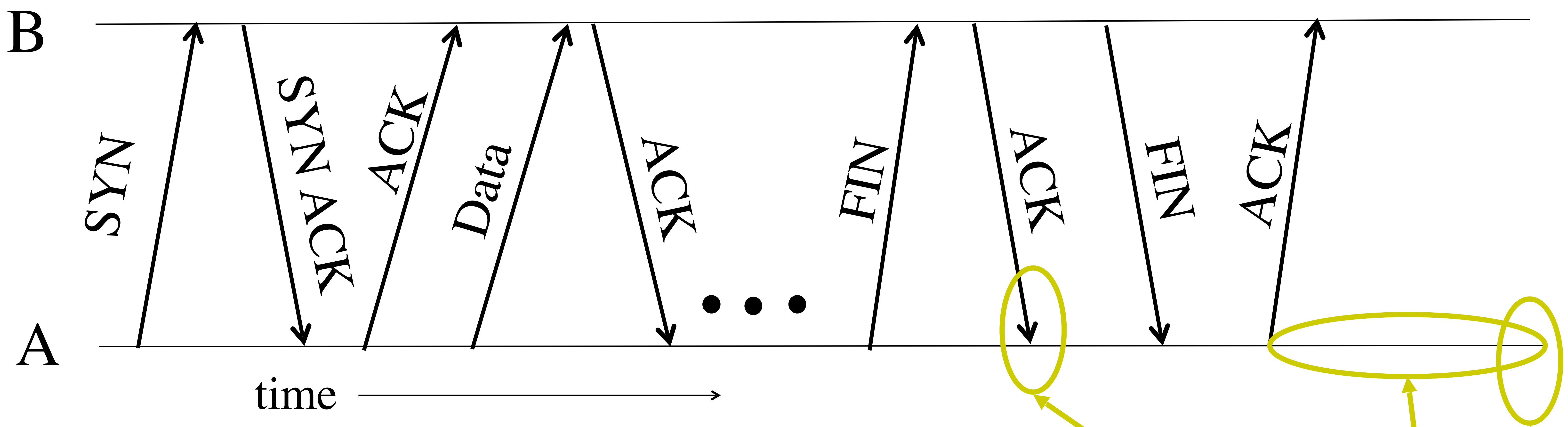
- How should the TCP sender set the timer?
  - Sender has no idea how far away the receiver is
  - Hard to guess a reasonable length of time to wait
  - **SHOULD** (RFCs 1122 & 2988) use default of 3 seconds
    - Some implementations instead use 6 seconds

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…
  - 3-6 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

Tearing Down the Connection

Normal Termination, One Side At A Time



- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host acks the byte to confirm
- Closes A's side of the connection, but not B's
  - Until B likewise sends a **FIN**
  - Which A then acks

Connection now **closed**

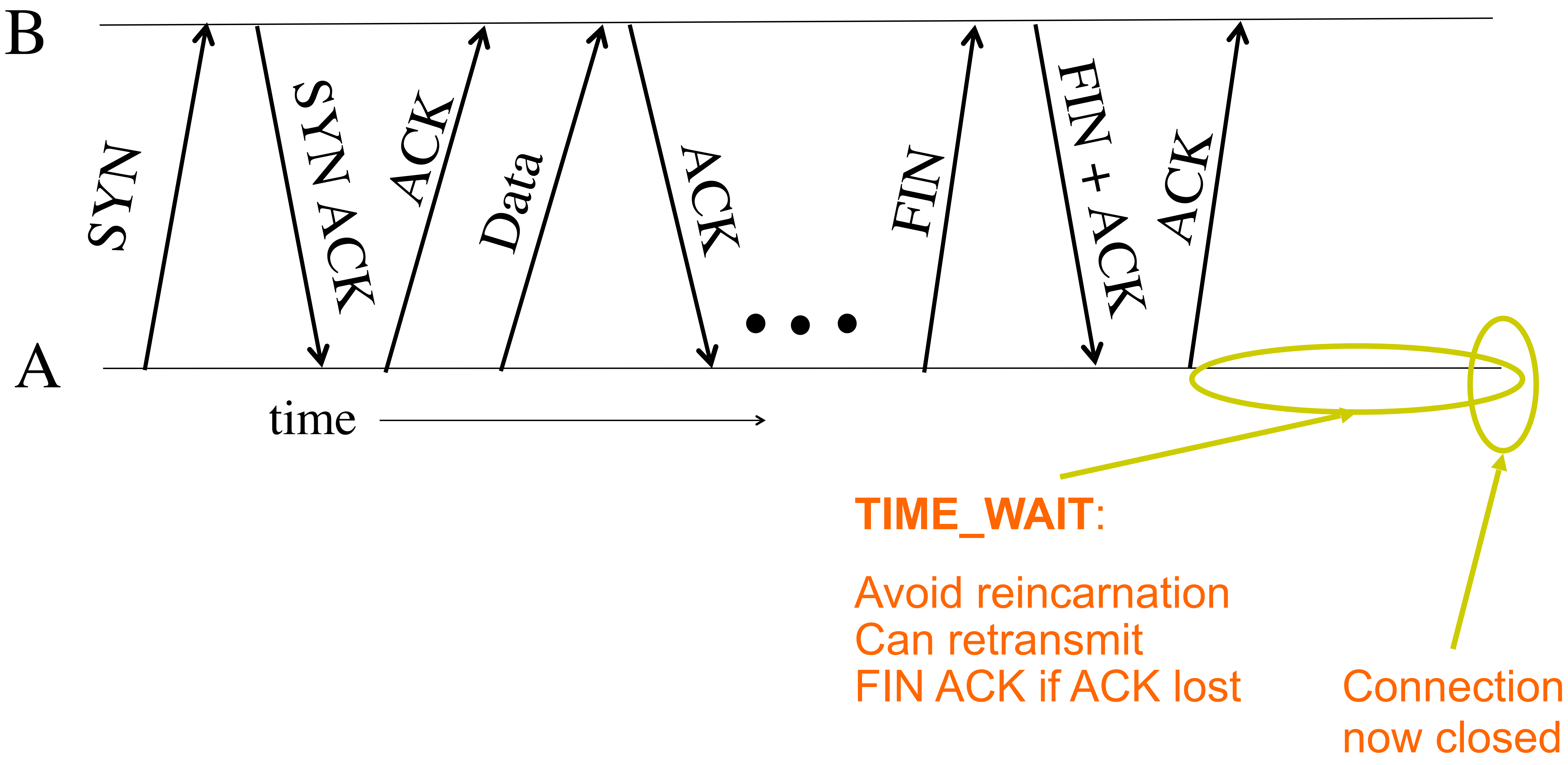
Connection now **half-closed**

TIME_WAIT:

Avoid reincarnation

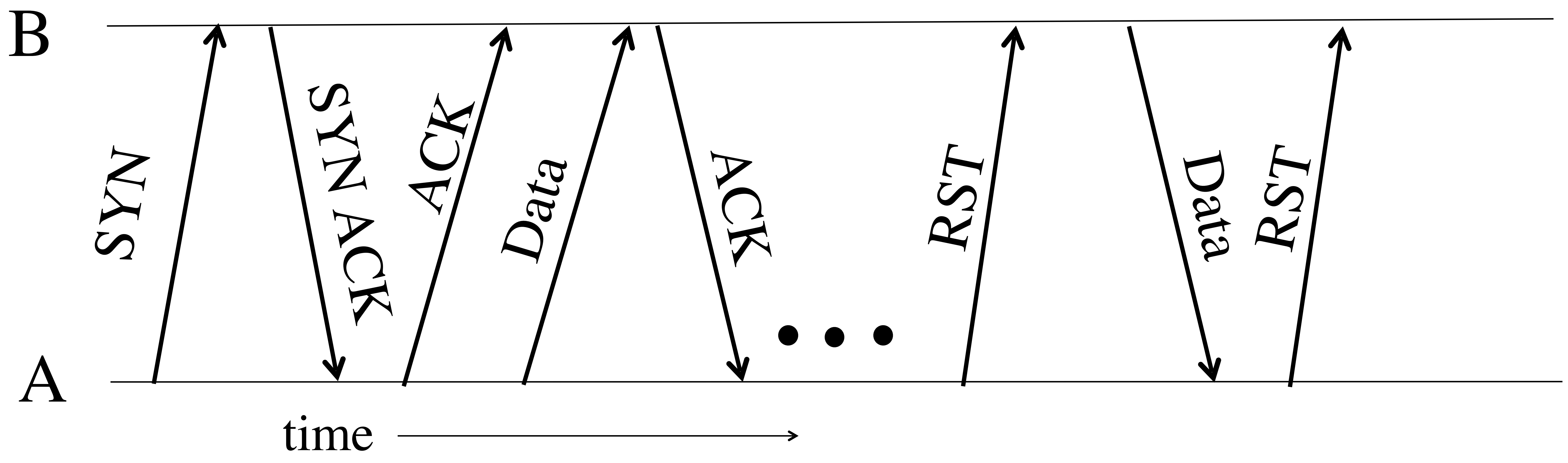
B will retransmit FIN if ACK is lost

Normal Termination, Both Together



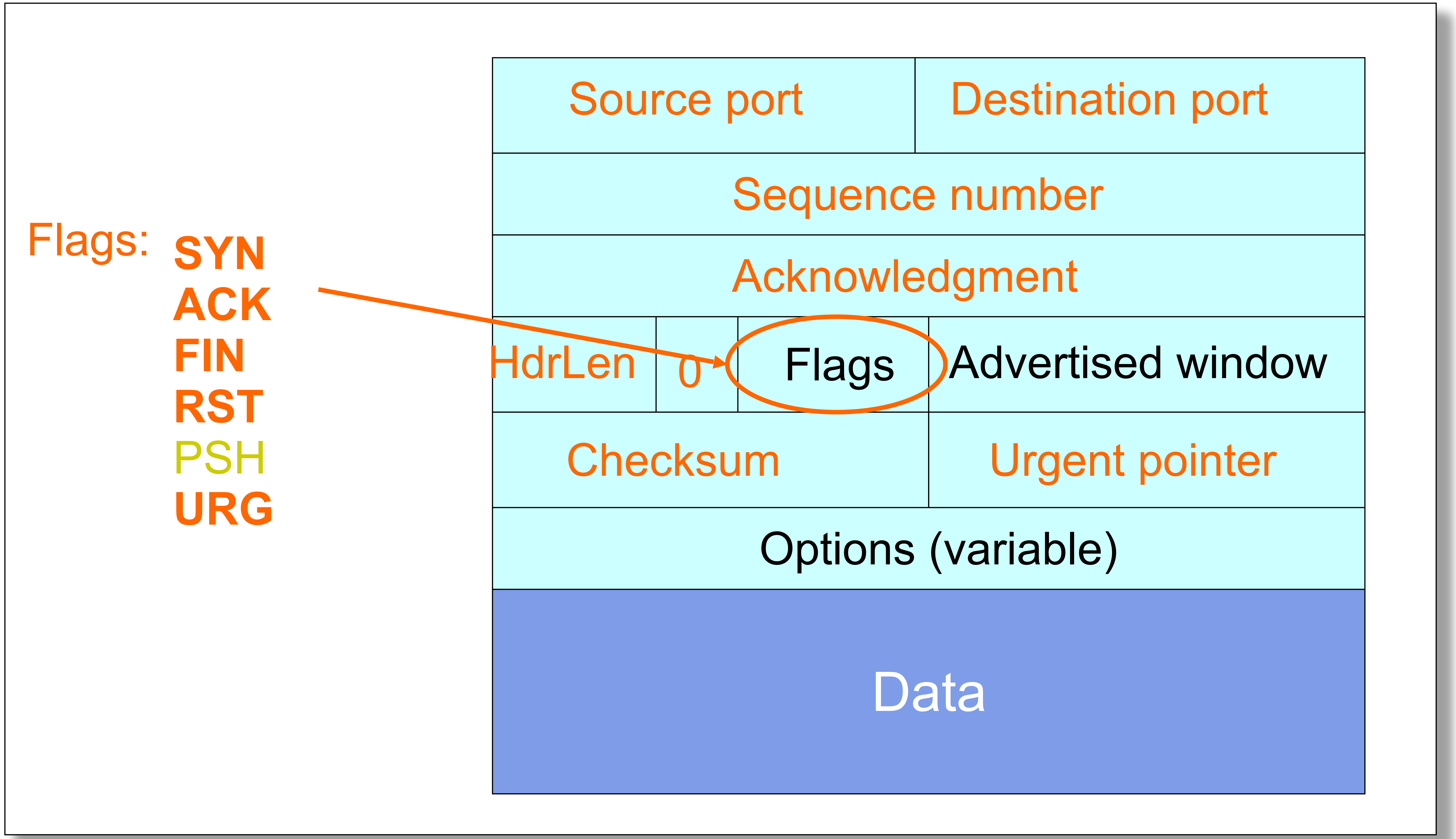
- Same as before, but B sets **FIN** with their ack of A's **FIN**

Abrupt Termination

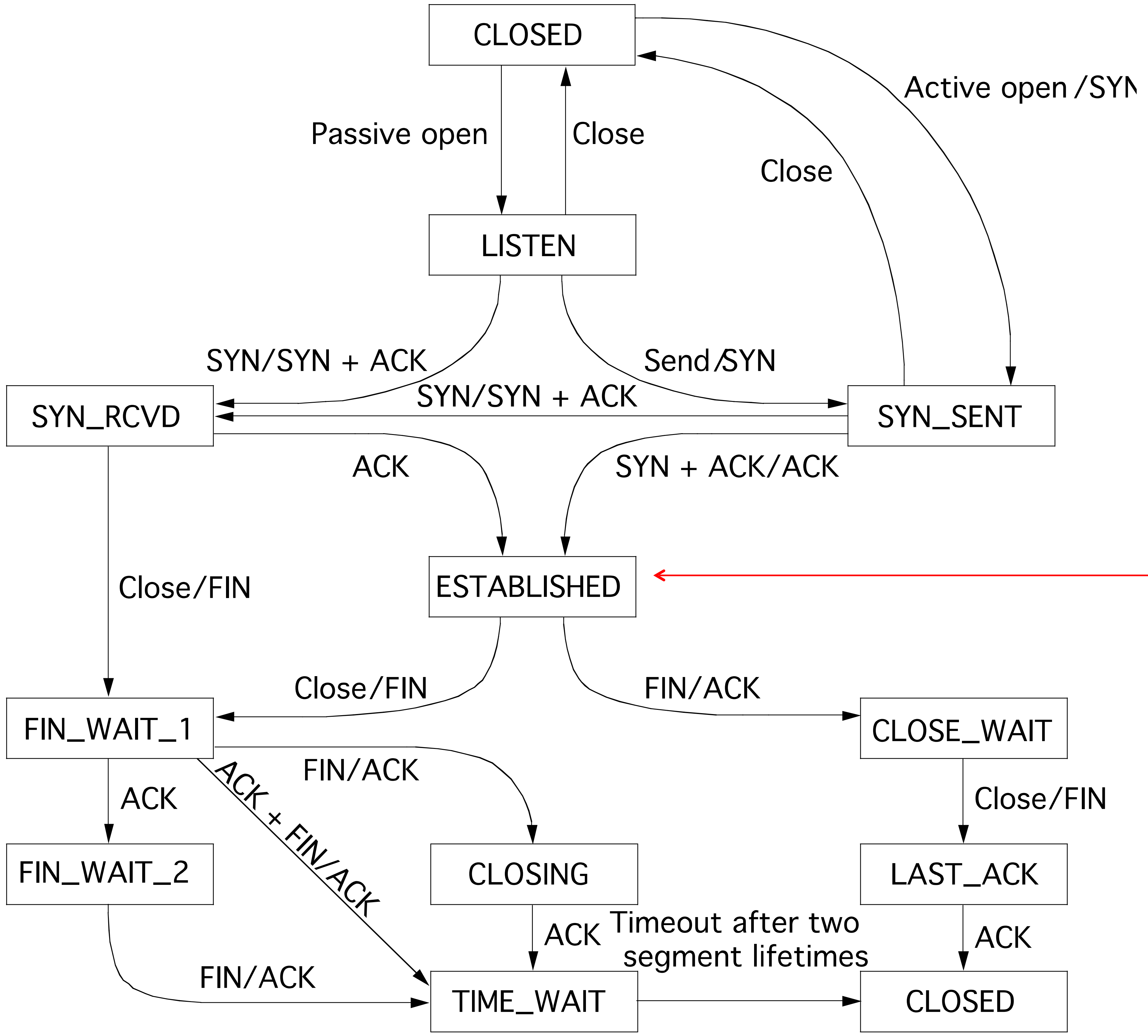


- A sends a RESET (**RST**) to B
  - E.g., because application process on A crashed
- That's it
  - B does not ack the **RST**
  - Thus, **RST** is not delivered reliably
  - And: any data in flight is lost
  - But: if B sends anything more, will elicit another **RST**

TCP Header


Flags:
SYN
ACK
FIN
RST
PSH
URG
Source port
Destination port
Sequence number
Acknowledgment
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

TCP State Transitions



Data, ACK exchanges are in here

After all that work…

- ESTABLISHED is the part where we transmit data.
- When our congestion control algorithm runs.

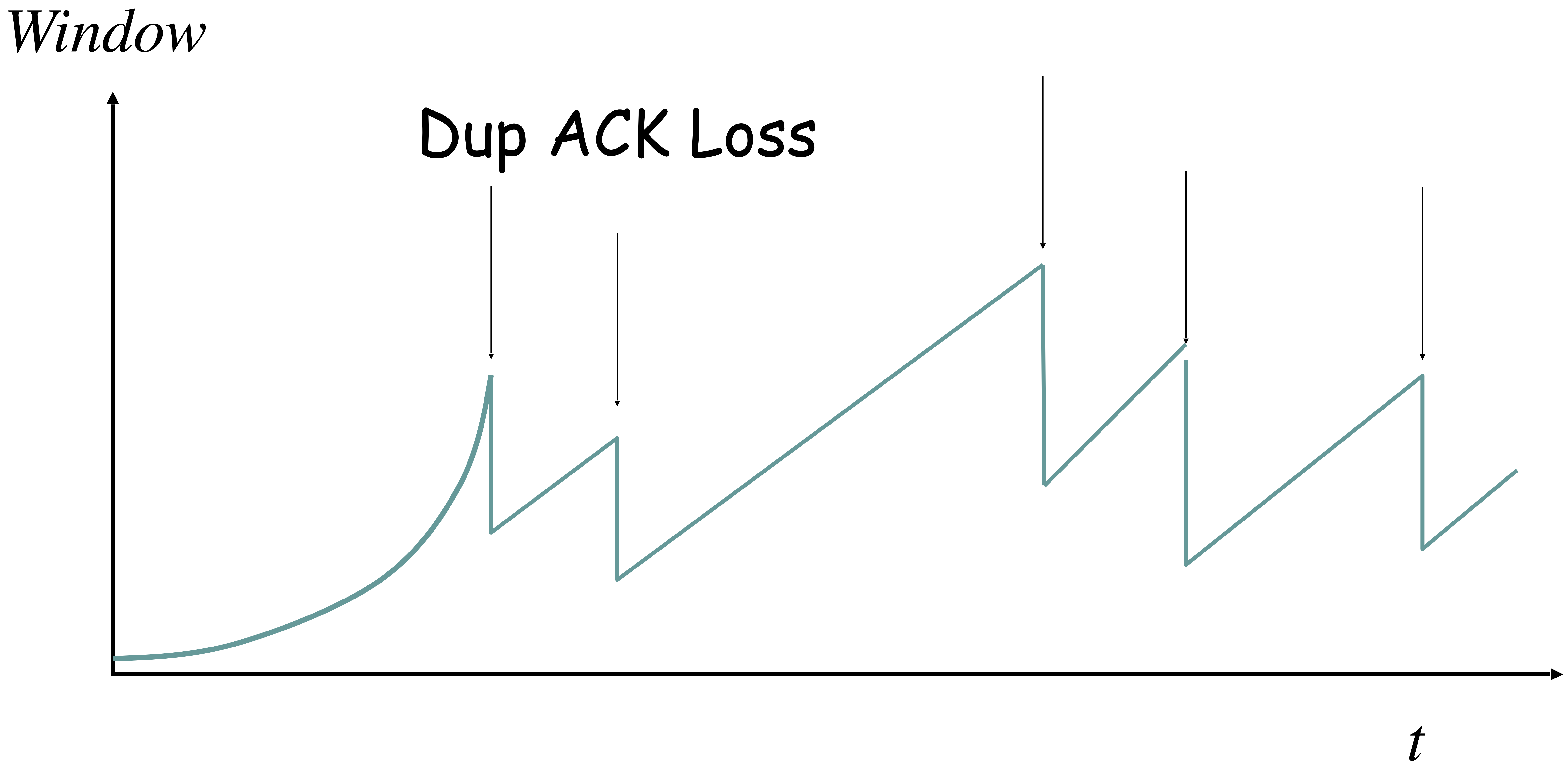
AIMD Mechanics in Reno

- "CWND" is the measured "congestion window"
  - Sending window is min(CWND, Advertised Window)
- Reno follows three key stages to determine CWND:
  - (1) Slow start, where it uses multiplicative increase
  - (2) Congestion avoidance, where it uses additive increase
  - (3) Fast recovery, where it "recovers" from "easy" packet losses.
    - *What do you mean, Easy Packet Losses?*

Duplicate ACKs

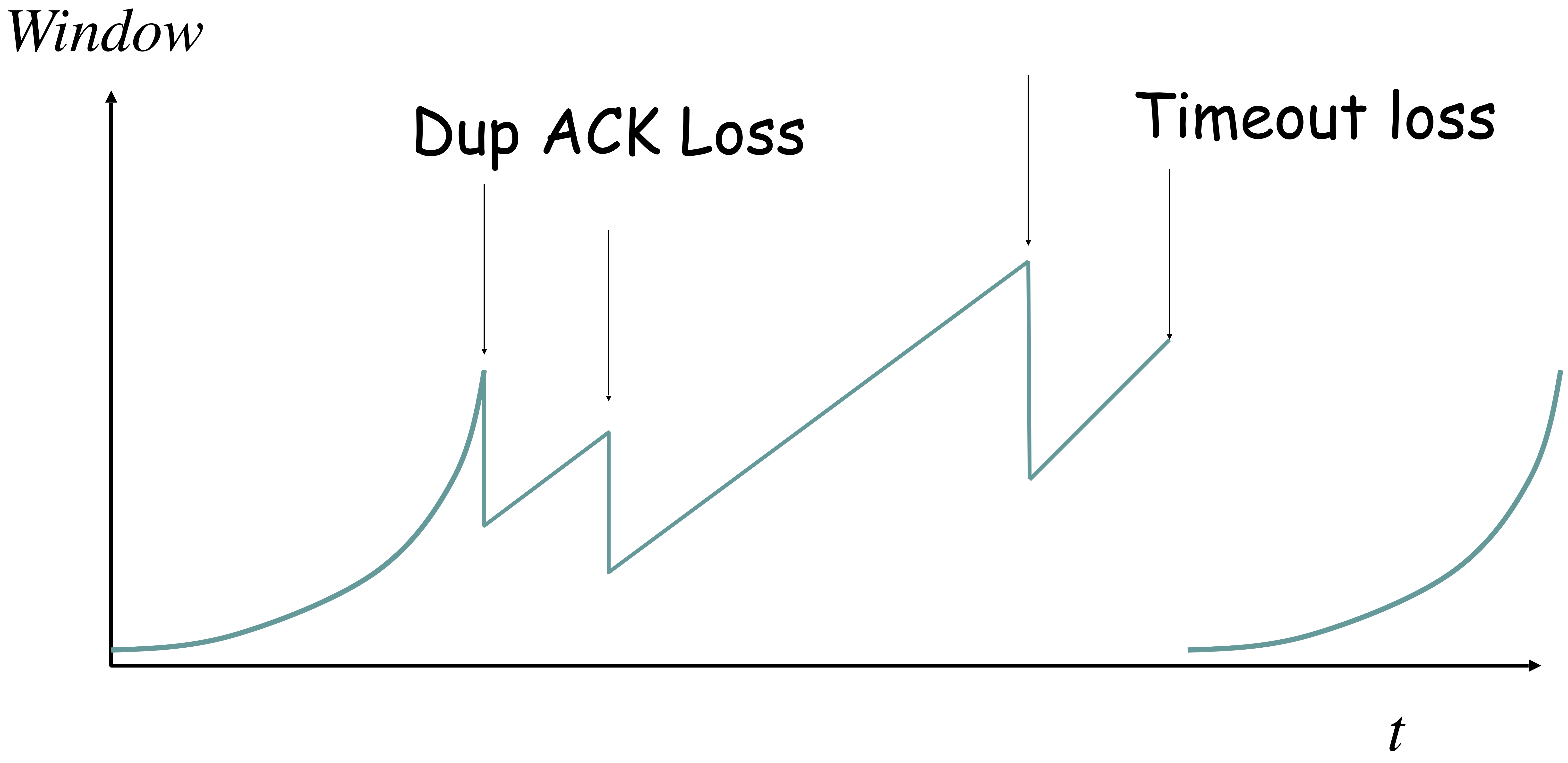
- I can pre-emptively figure out that loss has happened without a timer going off.
- How?
    - Say I receive packets with MSS 1000, sequence numbers 1000, 2000, 4000, 5000, 6000….
    - I know I missed 3000!
- Recall that TCP uses cumulative ACKs — I ACK the next byte such that I have the data for all bytes lower than that.
    - If I see the same “dup” ACK three times, I determine there is a loss.

Leads to the TCP "Sawtooth"

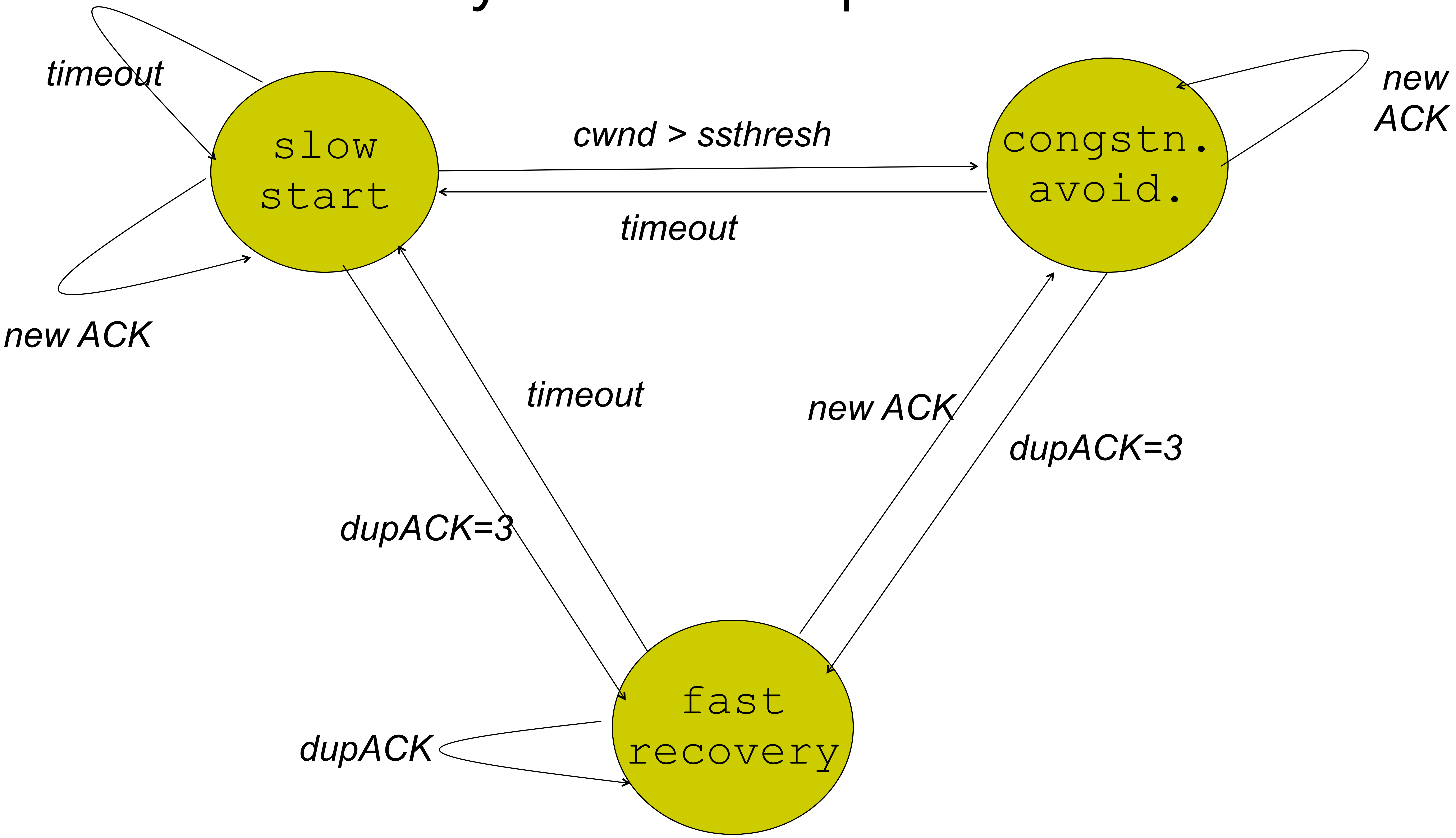


Assumption: Timeout Losses are Worse

- Timeout *can mean* (but not always) that lots of packets were lost and I have severely overshot.
- So I should react more severely to a timeout.
- Instead of halving my window, I will go all the way back to slow start and start over again!


Window
Dup ACK Loss
Timeout loss
t

Print this out and tape it above your bed. This is what you will implement for P2!



Summary

- All TCP connections use the same handshake, initial sequence number exchange, etc.
- But determining the right window size is *hard* because the network does not tell us directly how much capacity is available to us!
  - There are lots of algorithms to measure “CWND”
  - Reno is the classic algorithm, and it uses AIMD.

Thursday

- Why AIMD converges to fairness
- Calculating TCP throughput with loss
- Problems with TCP Reno
- New TCPs: Cubic, BBR
- Is the Internet fair?