Networking

CS 4410
Operating Systems

[R. Agarwal, L. Alvisi, A. Bracy, M. George, Kurose, Ross, E. Sirer, R. Van Renesse]

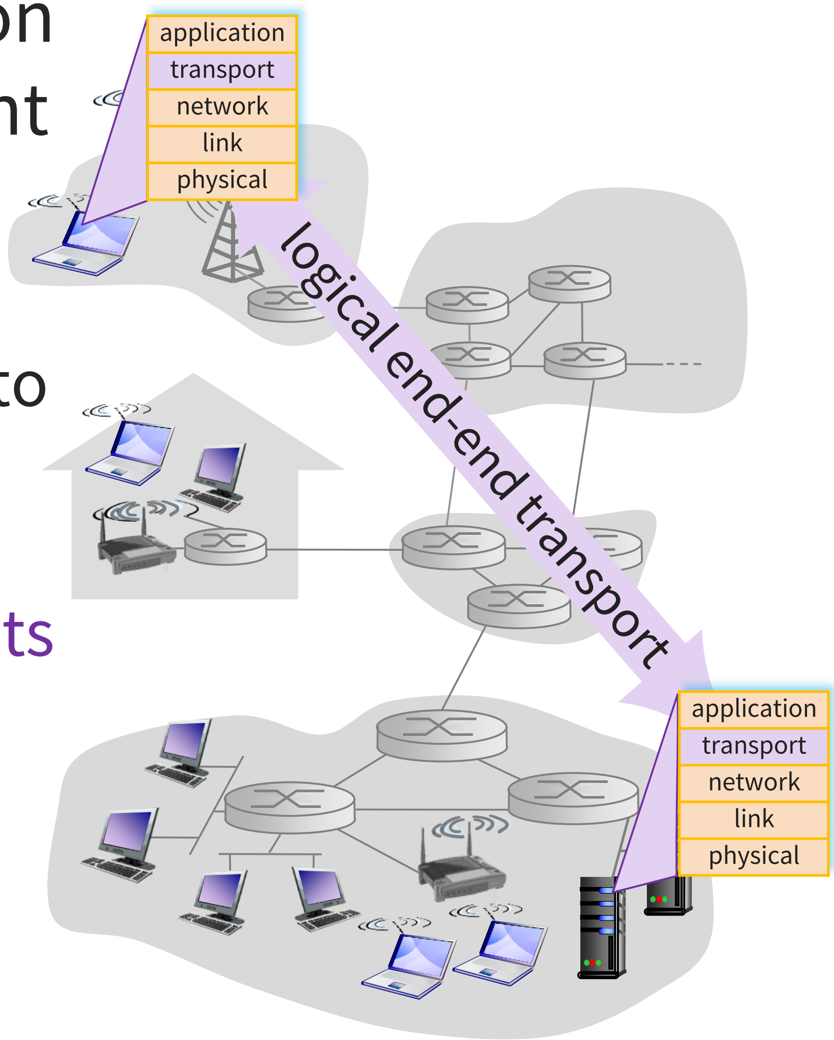
Application
Transport
Network
Link
Physical

Transport Layer: UDP & TCP

Several figures in this section come from
“Computer Networking: A Top Down Approach”
by Jim Kurose, Keith Ross

Transport services and protocols

- Provide **logical** communication between processes on different hosts
- Run in end systems
  - **Sender:** packages messages into segments, passes to network layer
  - **Receiver:** reassembles segments into messages, passes to application layer



App chooses protocol it wants (*e.g.*, TCP or UDP)

Transport services and protocols

User Datagram Protocol (UDP)

- **unreliable, unordered delivery**
- no-frills extension of best-effort IP

“**Unreliable** Datagram Protocol”

Transmission Control Protocol (TCP)

- **reliable, in-order delivery**
- congestion control
- flow control
- connection setup

“**Trusty** Control Protocol”

Services **not** available:

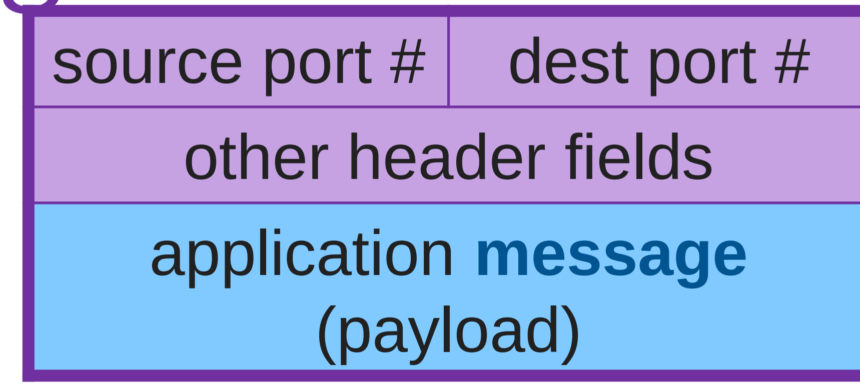
- delay guarantees
- bandwidth guarantees

How to create a segment

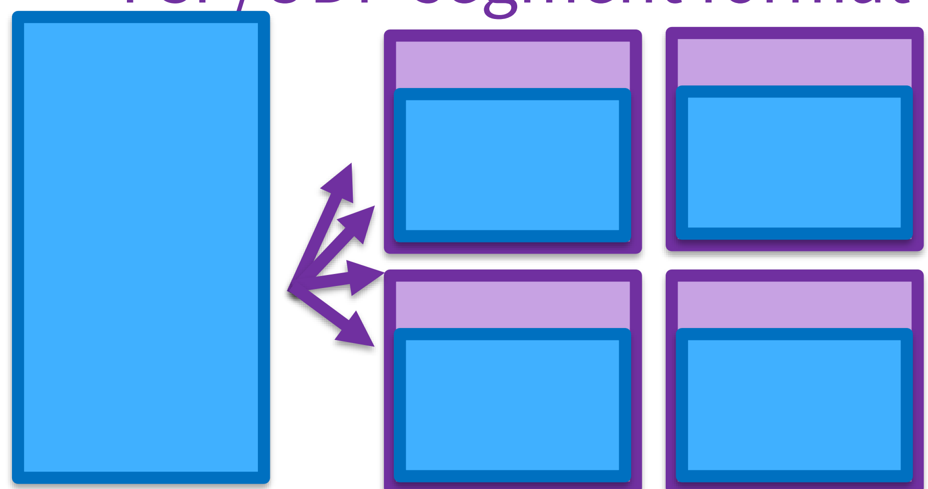
Sending application:
- specifies IP address and destination port
- uses socket bound to a source port

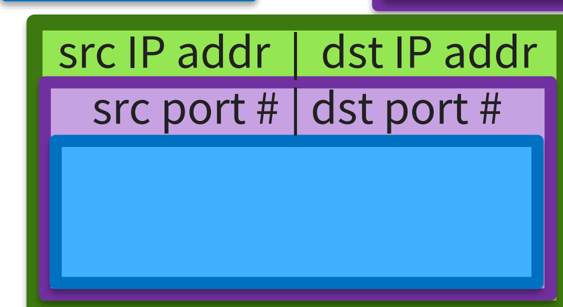
Transport Layer:
- breaks application message into smaller chunks
- adds transport-layer header to each

Network Layer:
- adds network-layer header (with IP address)



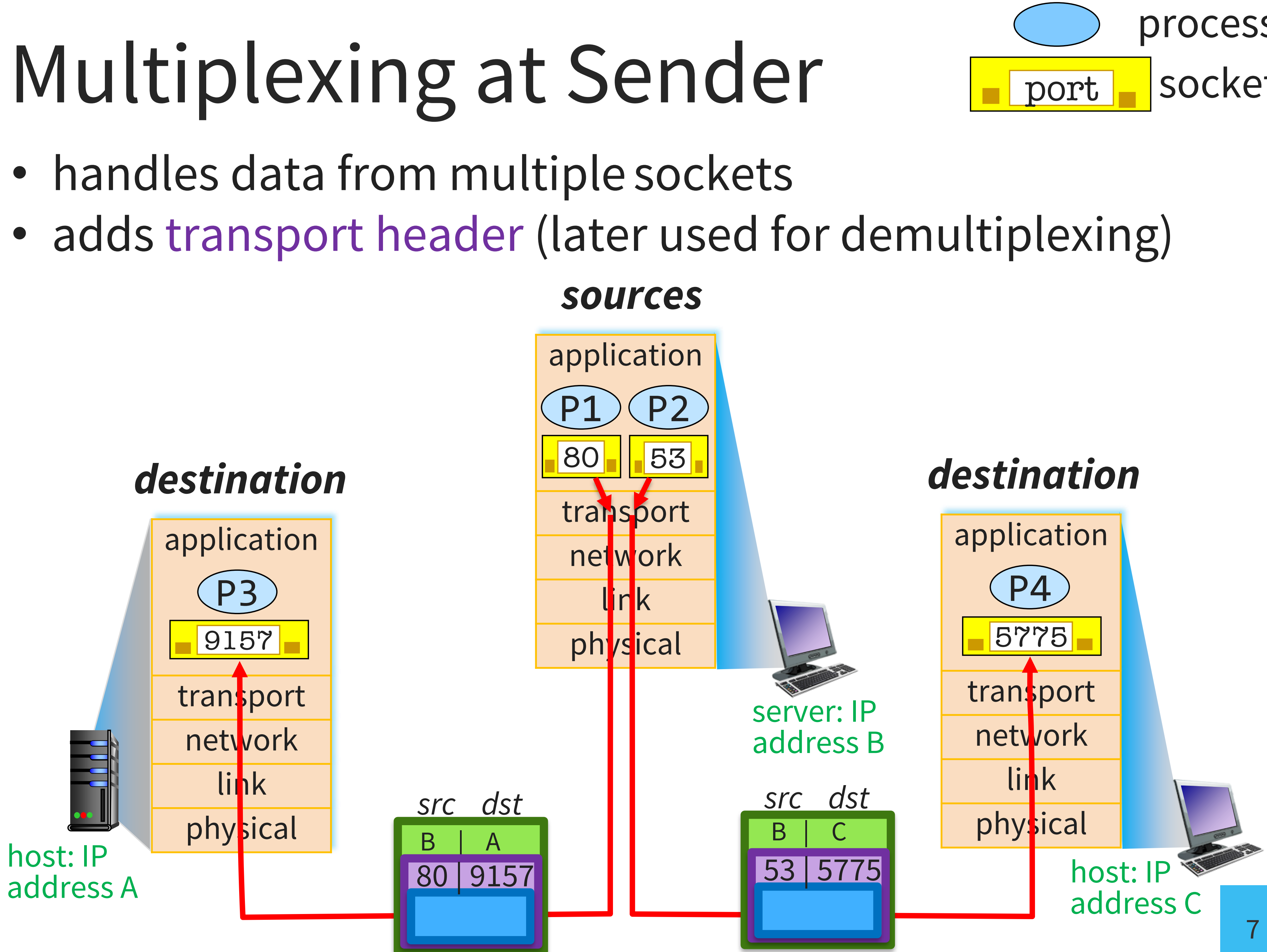
TCP/UDP segment format





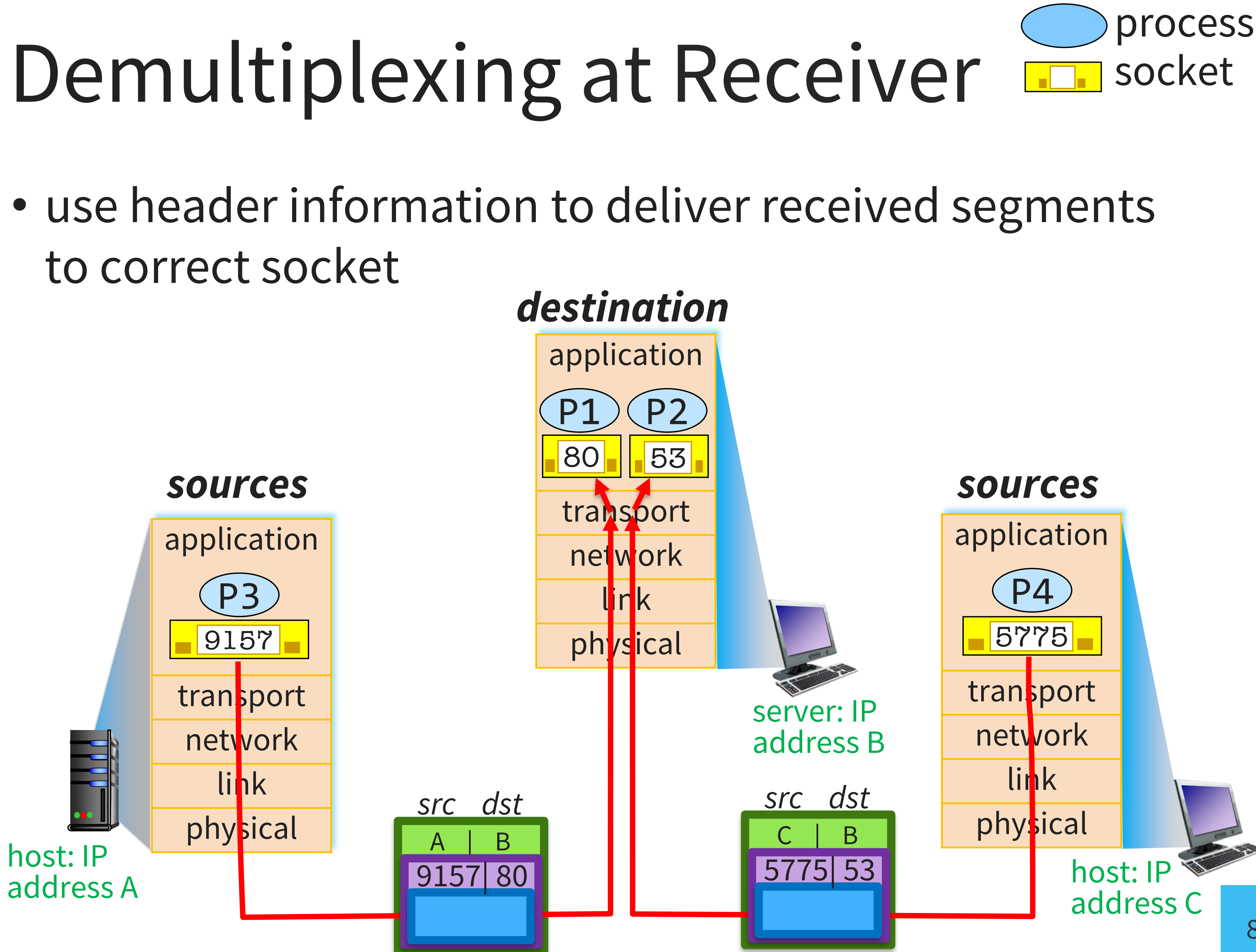
Multiplexing at Sender

- handles data from multiple sockets
- adds transport header (later used for demultiplexing)



Demultiplexing at Receiver

- use header information to deliver received segments to correct socket



User Datagram Protocol (UDP)

- no frills, bare bones transport protocol
- **best effort** service, UDP segments may be:
  - lost
  - delivered out-of-order, duplicated to app
- **connectionless:**
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others
- reliable transfer still possible:
  - **add reliability at application layer**
  - application-specific error recovery!

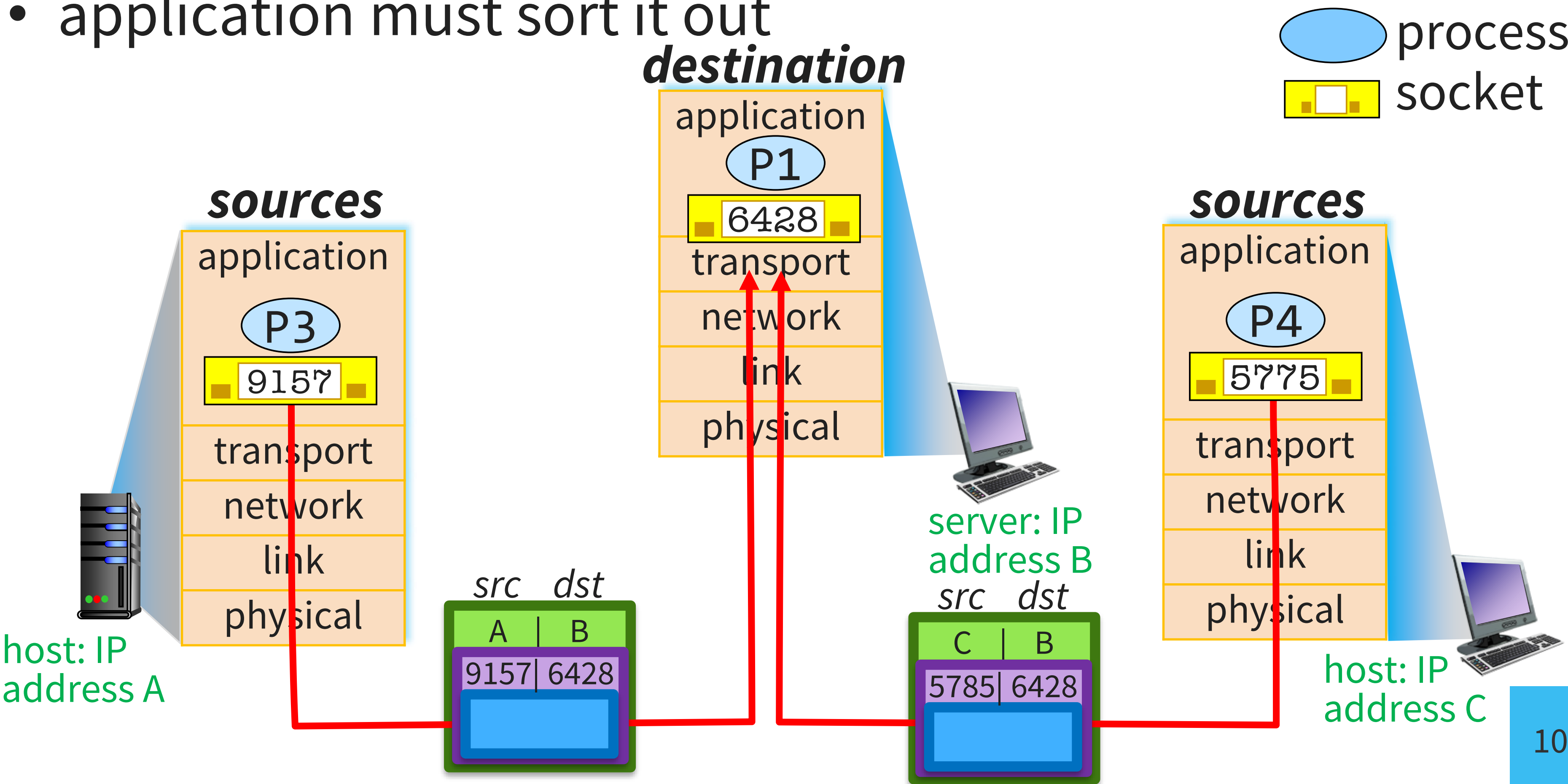
I was gonna tell you guys a joke about UDP…
But you might not get it

I was you guys about UDP might not

Connectionless demux: example

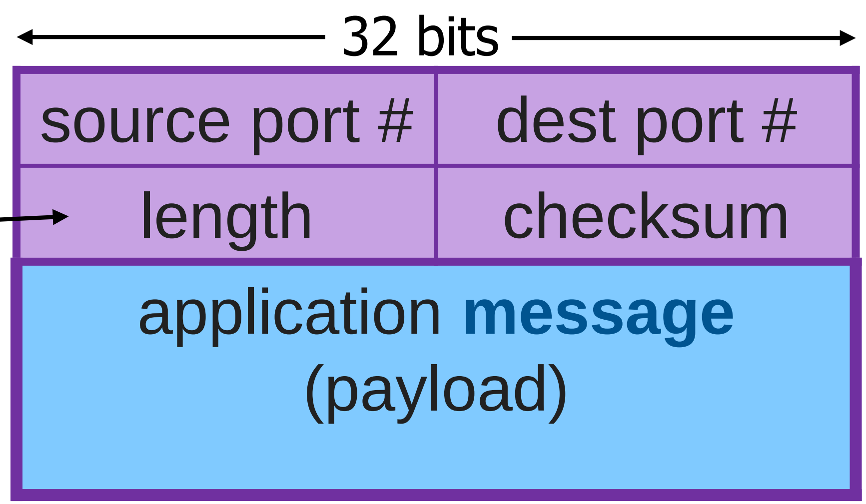
Host receives 2 UDP segments:

- checks dst port, directs segment to socket w/that port
- *different src IP or port* but *same dst port* → *same socket*
- application must sort it out



UDP Segment Format

length (in bytes) of UDP segment, including header



UDP header size: 8 bytes

(IP address will be added when the segment is turned into a datagram/packet at the Network Layer)

UDP Advantages & Disadvantages

Speed:

- no connection establishment (which can add delay)
- no congestion control: UDP can blast away as fast as desired

Simplicity:

- no connection state at sender, receiver
- small header size (8 bytes)

(Possibly) **Extra work for applications**:
Need to handle reordering, duplicate suppression, missing packets

Not all applications will care about these!

Who uses UDP?

Target Users: streaming multimedia apps

- loss tolerant *(occasional packet drop OK)*
- rate sensitive *(want constant, fast speeds)*

UDP is good to build on

Applications & their transport protocols

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	typically proprietary	UDP or TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Routing protocol	RIP	Typically UDP
Name translation	DNS	Typically UDP

Transmission Control Protocol (TCP)

- Reliable, ordered communication
- Standard, adaptive protocol that delivers good-enough performance and deals well with congestion
- All web traffic travels over TCP/IP

- Why? enough applications demand reliable ordered delivery that they should not have to implement their own protocol

TCP Segment Format

HL: header len

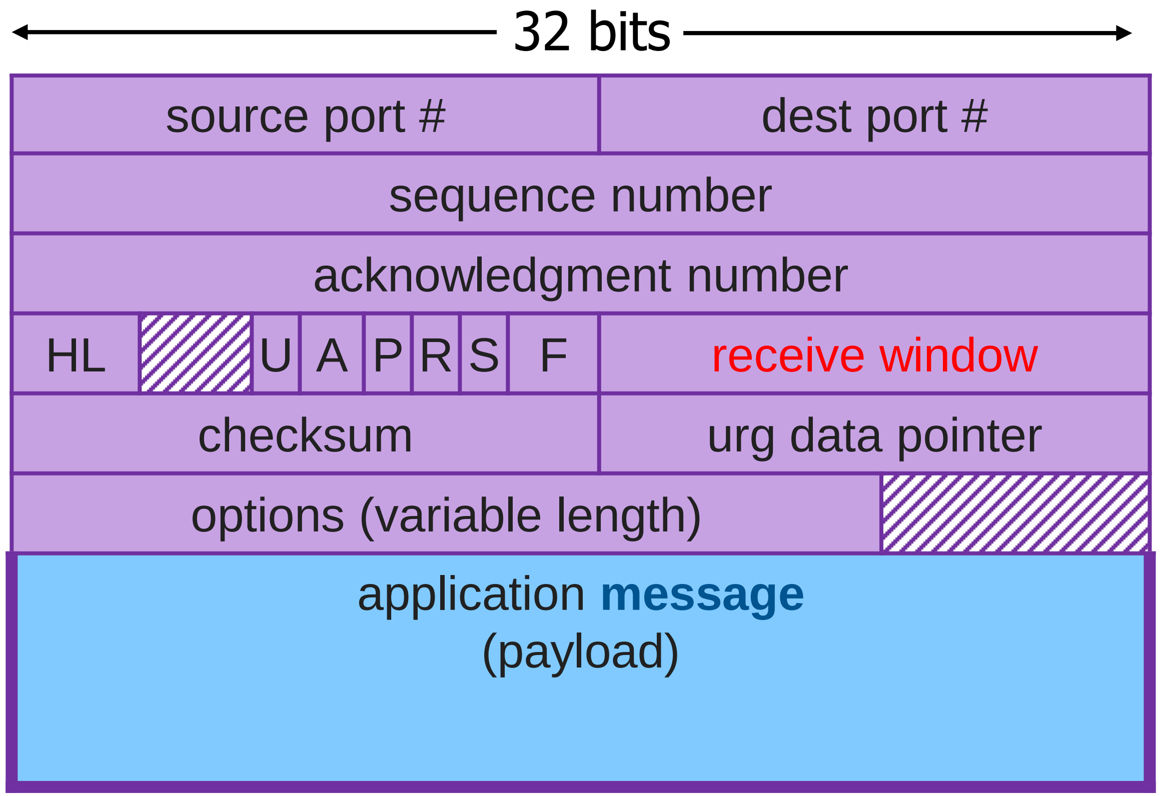
U: urgent data

A: ACK # valid

P: push data now

RST, SYN, FIN: connection commands (setup, teardown)

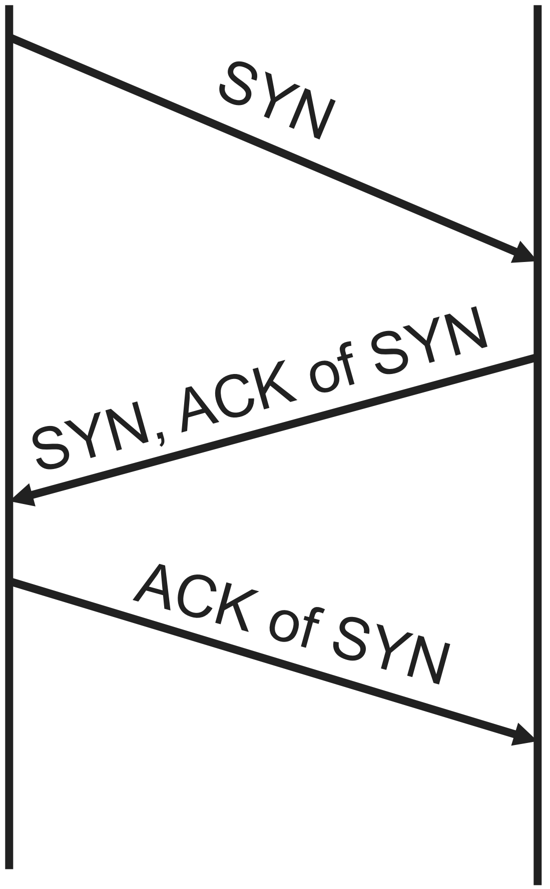
bytes receiver willing to accept



TCP header size: 20-60 bytes

(IP address will be added when the segment is turned into a datagram/packet at the Network Layer)

TCP Connections



- TCP is *connection* oriented
- A connection is initiated with a three-way handshake
- Three-way handshake ensures against duplicate SYN packets
- Takes 3 packets, 1.5 RTT (**R**ound **T**rip **T**ime)

SYN = Synchronize
ACK = Acknowledgment

I would tell you a joke about TCP... If only to be acknowledged 😢

TCP Handshakes

3-way handshake establishes common state on both sides of a connection.

Both sides will:

- have seen one packet from the other side → know what the first seq# ought to be
- know that the other side is ready to receive

Server will typically create a new socket for the client upon connection.

TCP Sockets

Server host may support many simultaneous TCP sockets

Each socket identified by its own 4-tuple

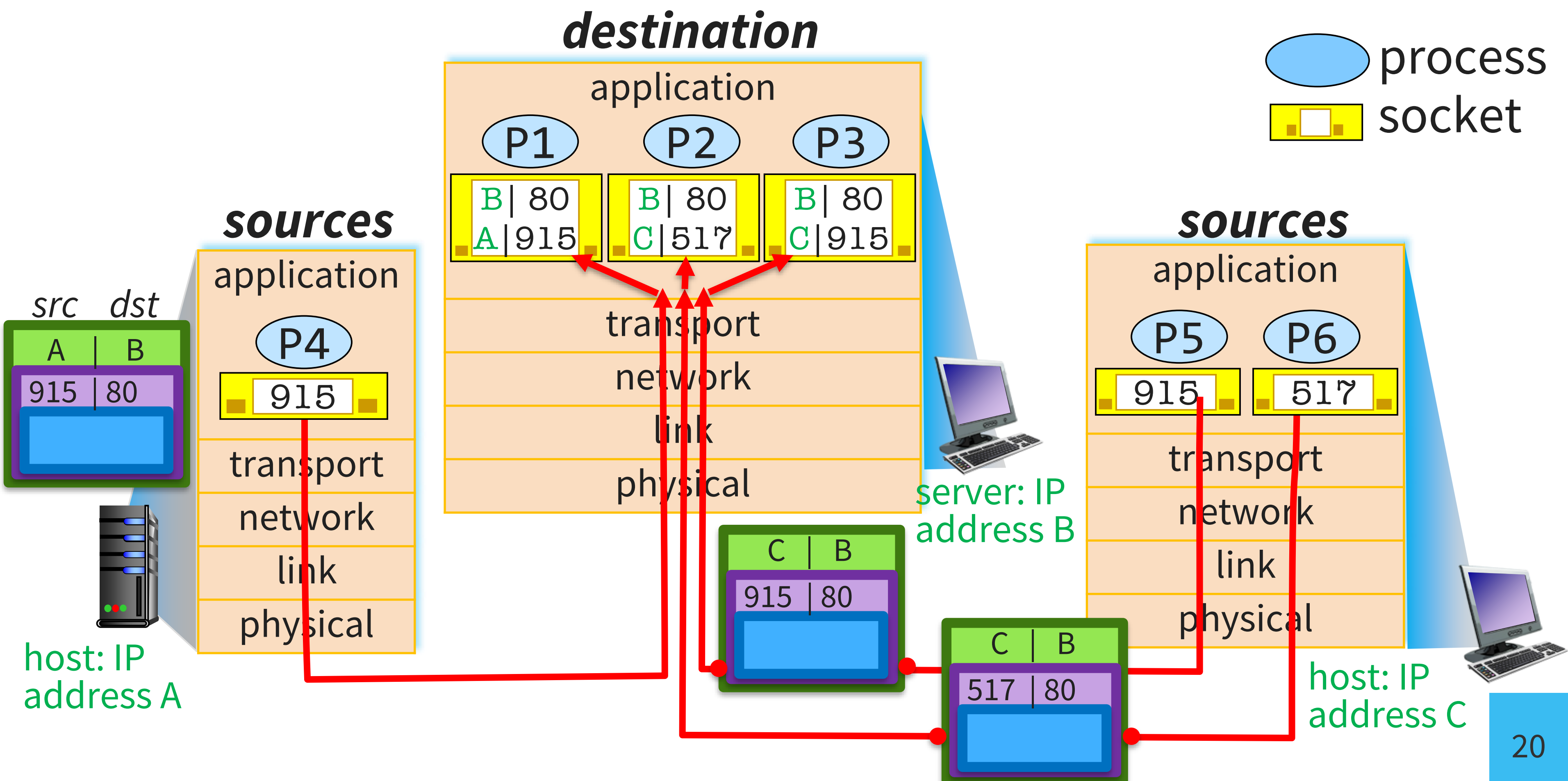
- source IP address
- source port number
- dest IP address
- dest port number

Connection-oriented demux: receiver uses all 4 values to direct segment to appropriate socket

Connection-oriented demux: example

Host receives 3 TCP segments:

- all destined to IP addr B, port 80
- demuxed to different sockets with socket's 4-tuple



TCP Packets

Each packet carries a unique sequence #

- The initial number is chosen randomly
- The SEQ is incremented by the data length

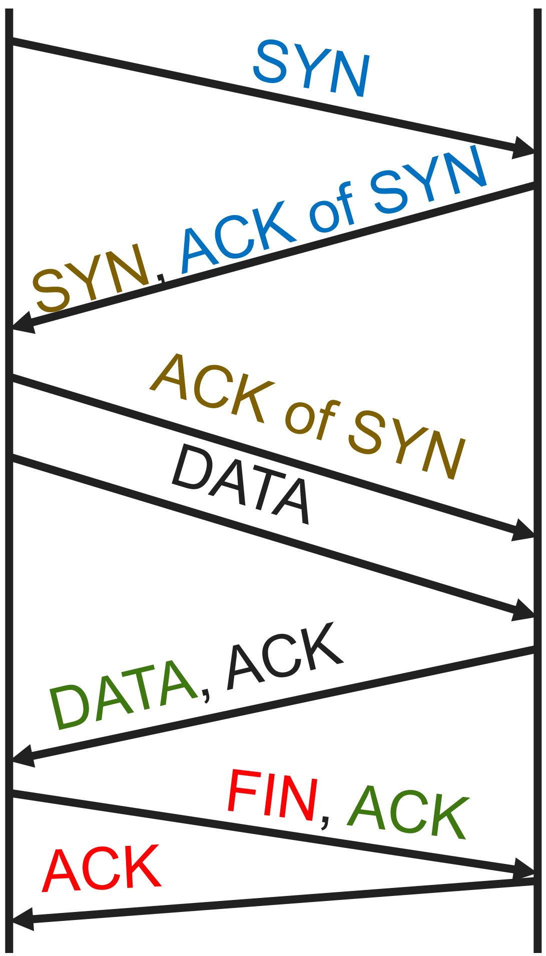
  4410 simplification: just increment by 1

Each packet carries an **ack**nowledgment

- Acknowledge a set of packets by ACK-ing the latest SEQ received

Reliable transport is implemented using these identifiers

TCP Usage Pattern



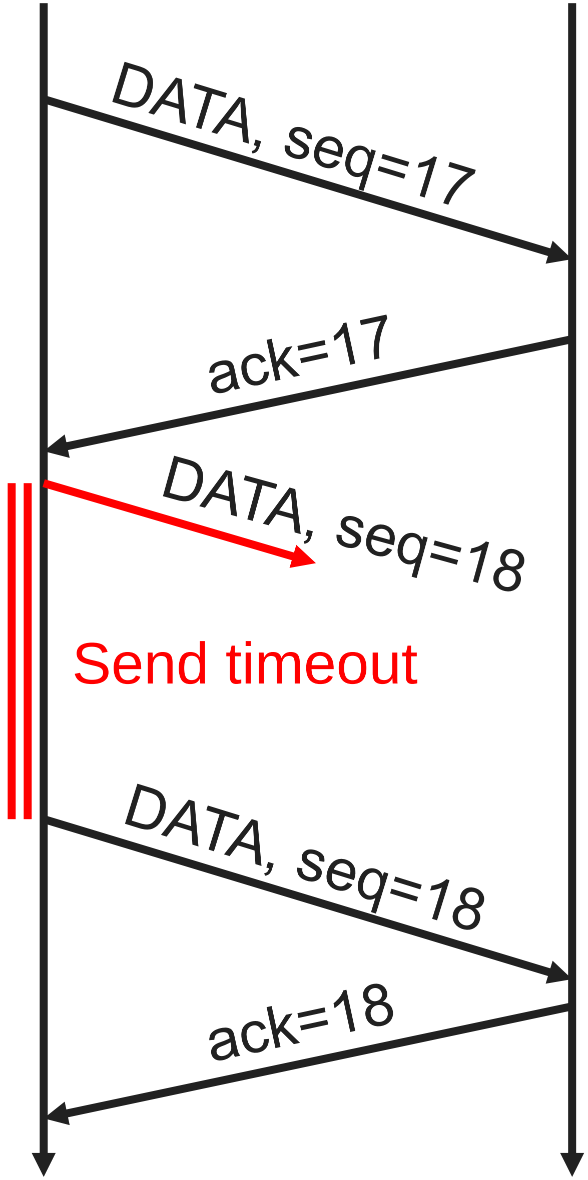
3 round-trips:

1. set up a connection
2. send data & receive a response
3. tear down connection

FINs work (mostly) like SYNs to tear down connection

Need to wait after a FIN for straggling packets

Reliable transport



- Sender-side: TCP keeps a copy of all sent, but unacknowledged packets
- If acknowledgment does not arrive within a “send timeout” period, packet is resent
- Send timeout adjusts to the round-trip delay

Here's a joke about TCP.
Did you get it?
Did you get it?
Did you get it?
Did you get it?

How long does it take to send a segment?

- S: size of segment in bytes
- L: one-way latency in seconds
- B: bandwidth in bytes per second

- Then the time between the start of sending and the completion of receiving is about L + S/B seconds (ignoring headers)
- And another L seconds (total: 2L + S/B) before the acknowledgment is received by the sender
  - assuming ack segments are small
- The resulting end-to-end throughput (without pipelining) would be about S / (2L + S/B) bytes/second

TCP timeouts

What is a good timeout period ?

- Goal: improve throughput without unnecessary transmissions

```
NewAverageRTT = (1 - α) OldAverageRTT + α LatestRTT
NewAverageVar = (1 - β) OldAverageVar + β LatestVar
where LatestRTT = (ack_receive_time – send_time),
           LatestVar  = |LatestRTT – AverageRTT|,
           α = 1/8, β = ¼ typically.
Timeout = AverageRTT + 4*AverageVar
```

→ Timeout is a function of RTT and variance

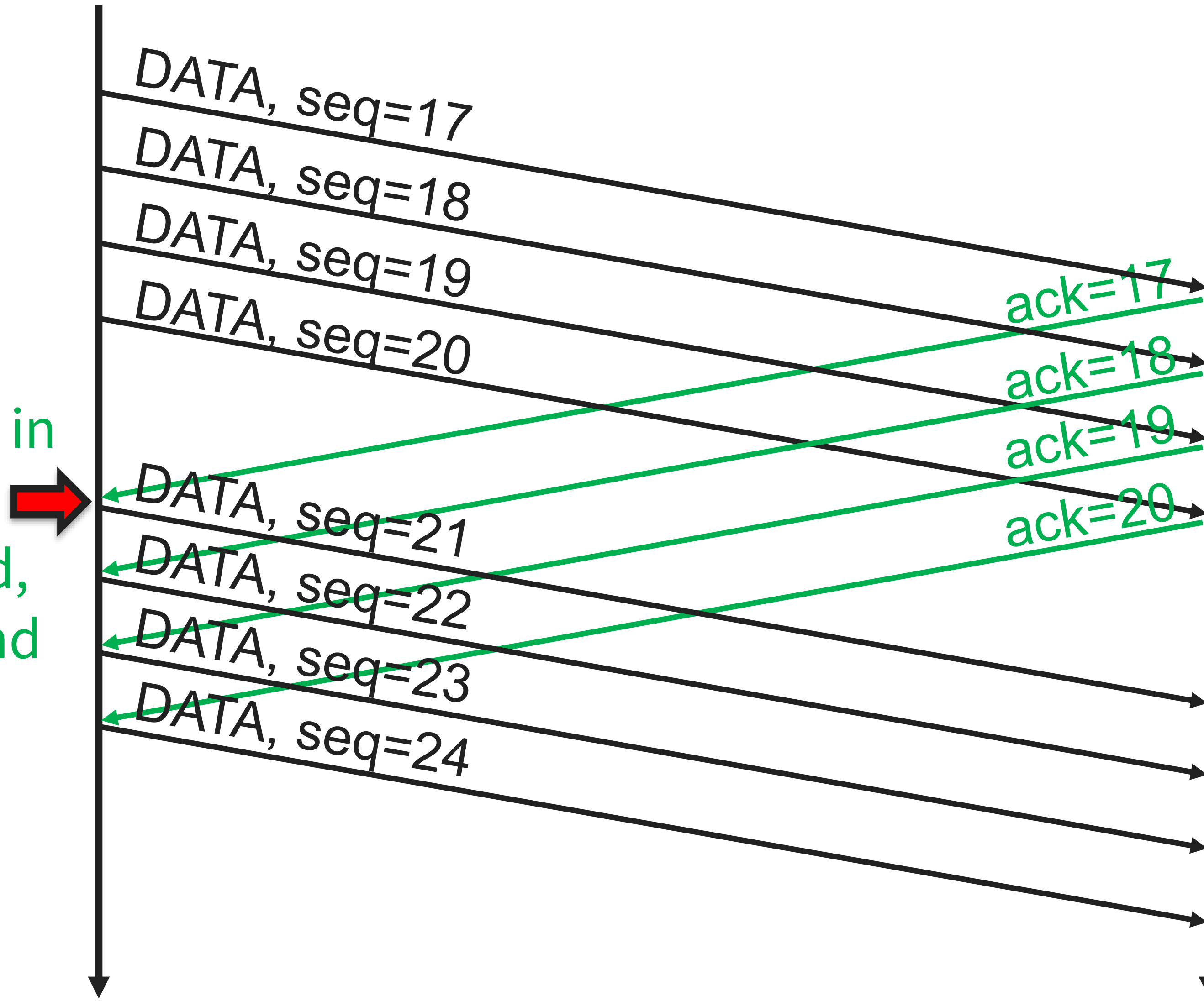
Pipelined Protocols

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

- increases throughput
- need buffering at sender and receiver
- How big should the window be?
- What if a packet in the middle goes missing?

Example: TCP Window Size = 4

When first item in window is acknowledged, sender can send the 5th item.



How much data "fits" in a pipe?

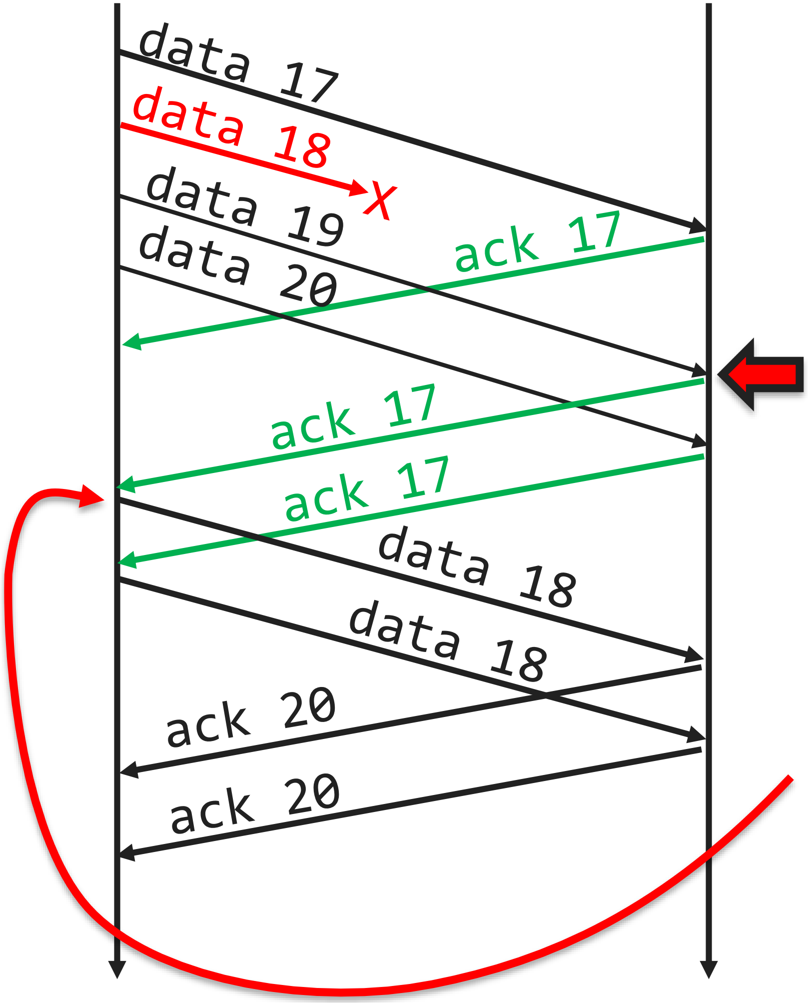
Suppose:

- b/w is b bytes / second
- RTT is r seconds
- ACK is a small message

→ you can send `b*r` bytes before receiving an ACK for the first byte

(but b/w and RTT are both variable…)

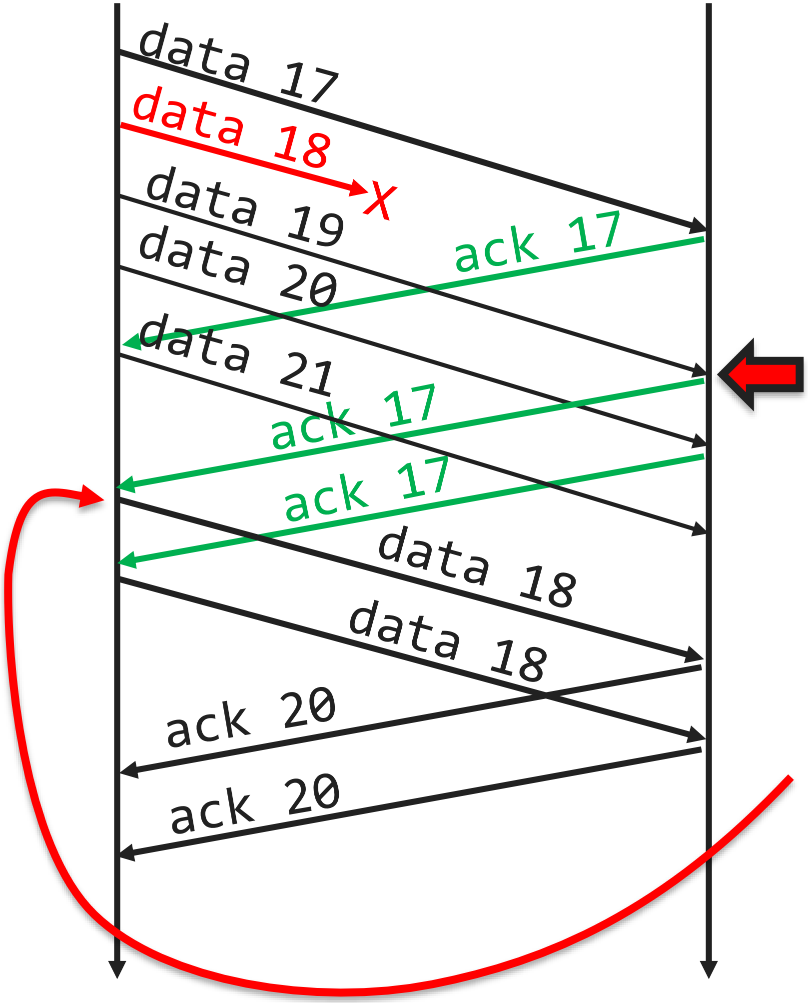
TCP **Fast Retransmit**



Receiver detects a lost packet (*i.e.*, a missing seq), ACKs the last id it successfully received

Sender can detect the loss without waiting for timeout

TCP **Fast Retransmit**



Receiver detects a lost packet (*i.e.*, a missing seq), ACKs the last id it successfully received

Sender can detect the loss without waiting for timeout

TCP Congestion Control

Additive-Increase/Multiplicative-Decrease (**AIMD**):

- window size++ every RTT if no packets dropped
- window size/2 if packet is dropped
  - drop evident from the acknowledgments

→ slowly builds up to max bandwidth, and hover there

- Does not achieve the max possible

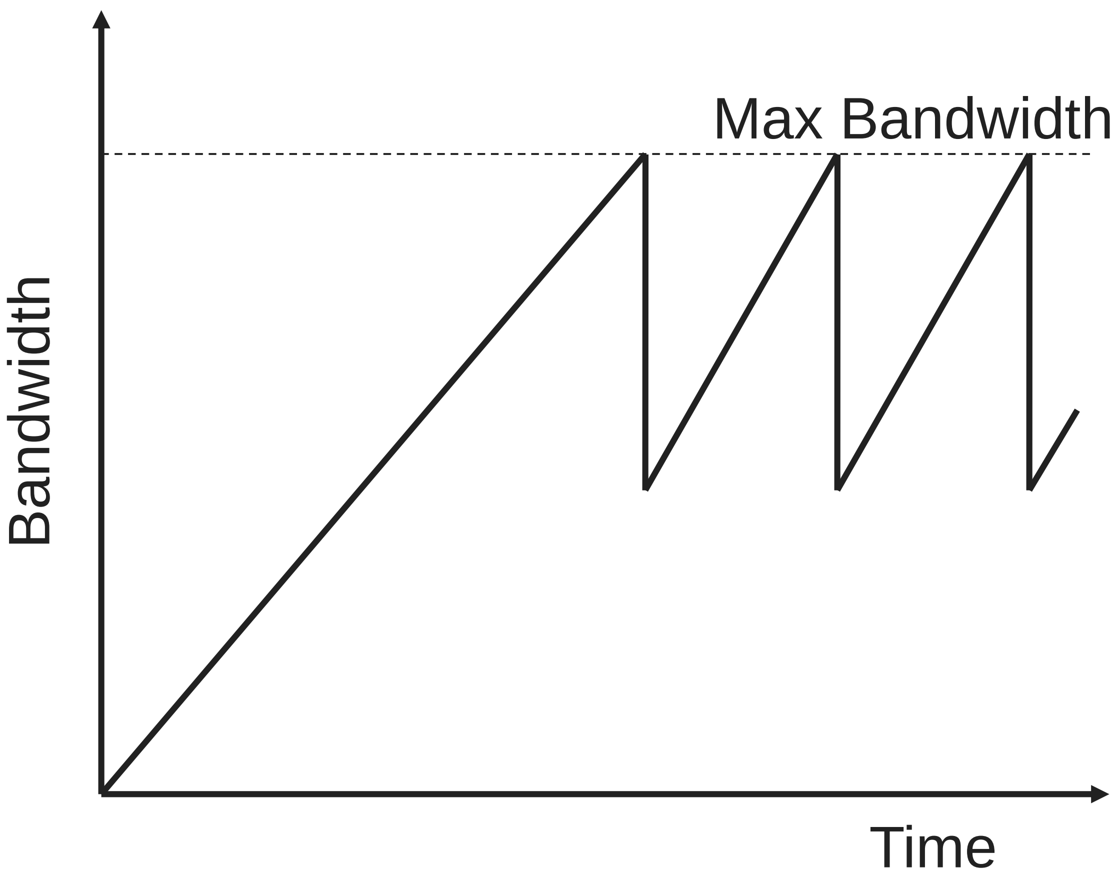
+ Shares bandwidth well with other TCP connections

This linear-increase, exponential backoff in the face of congestion is termed ***TCP-friendliness***

TCP Window Size

- Linear increase
- Exponential backoff

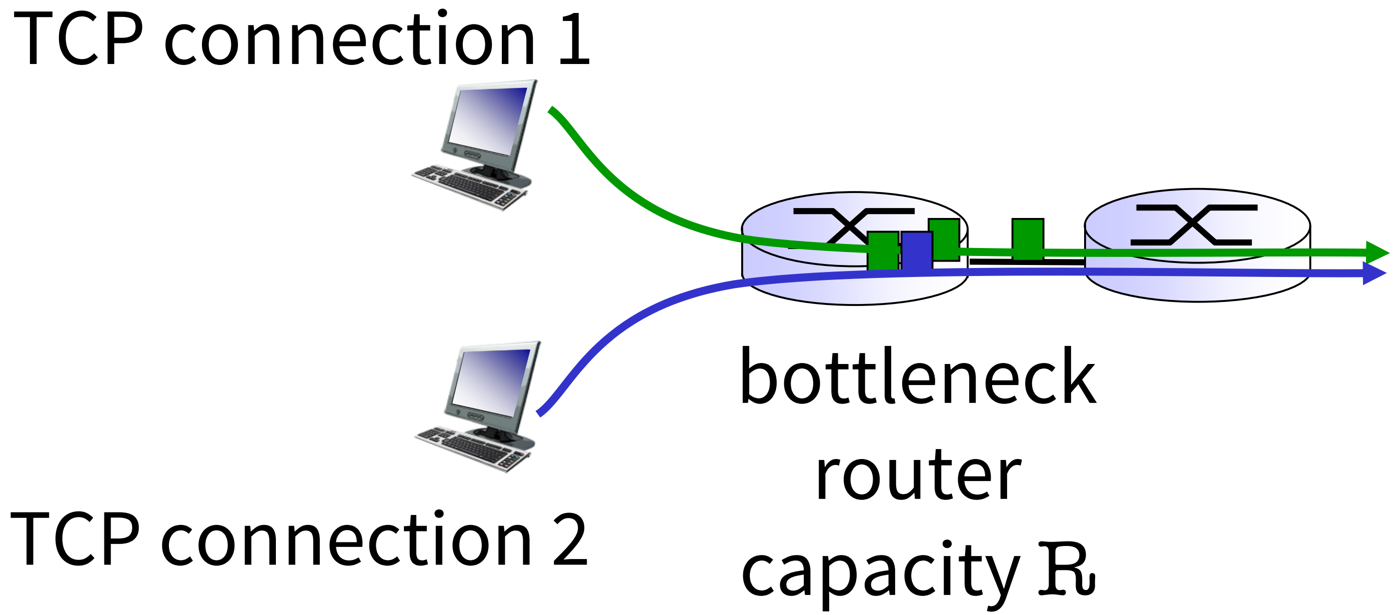
(Assuming no other losses in the network except those due to bandwidth)



Window Sizes:
1,2,3,4,5,6,7,8,9,10,
5,6,7,8,9,10,
5,6,7,8,9,10,
. . .

TCP Fairness

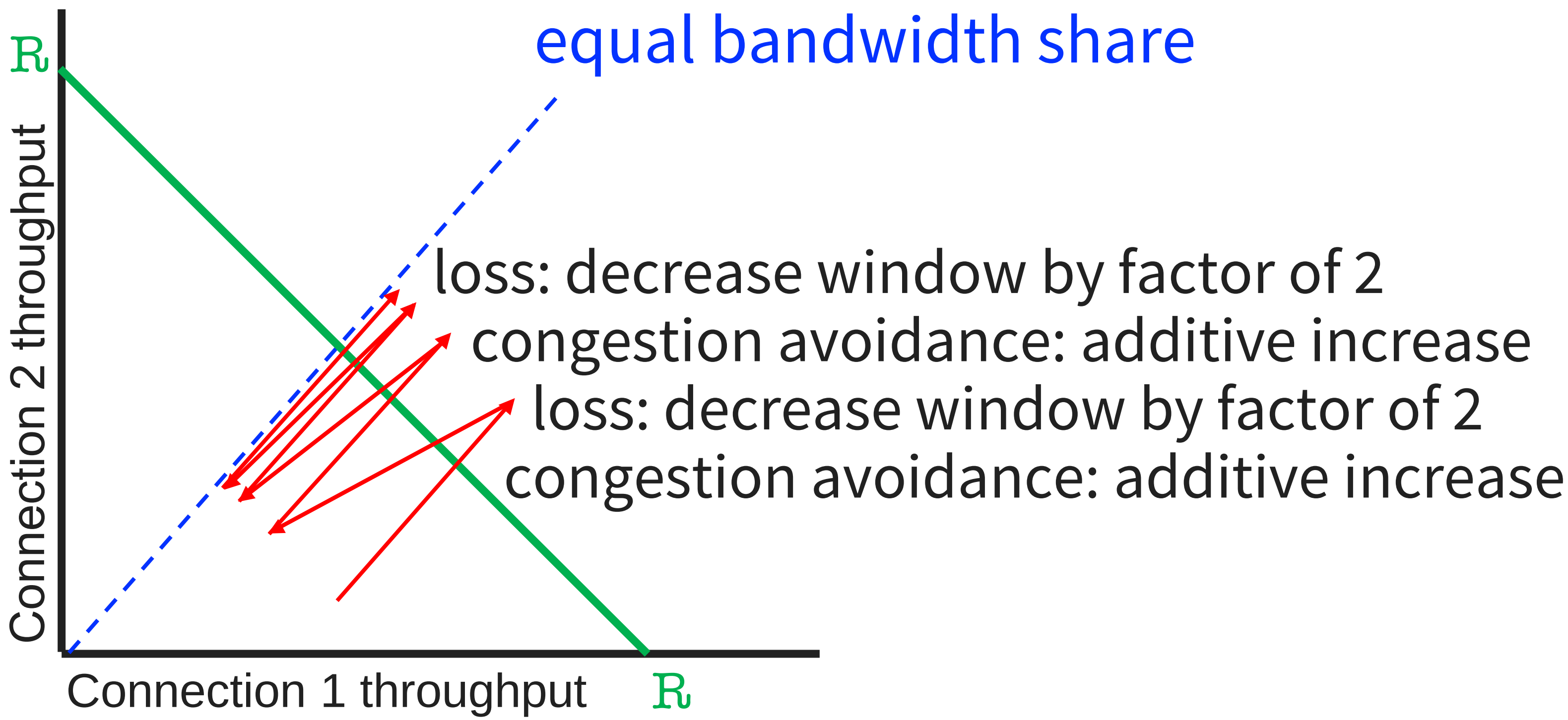
Fairness goal: if `k` TCP sessions share same bottleneck link of bandwidth `R`, each should have average rate of `R/k`



Why is TCP fair?

Two competing sessions:

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

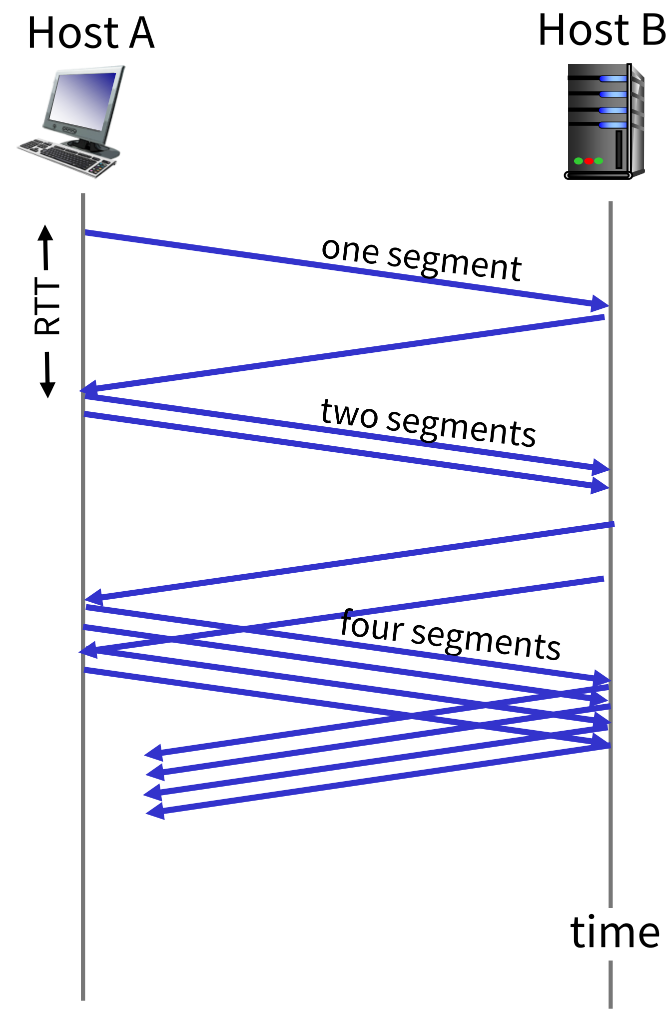


TCP Slow Start

(horrible name)

Problem:
- linear increase takes a long time to build up a window size that matches the link bandwidth*delay
- most file transactions are short

→ TCP spends a lot of time with small windows, never reaching large window size

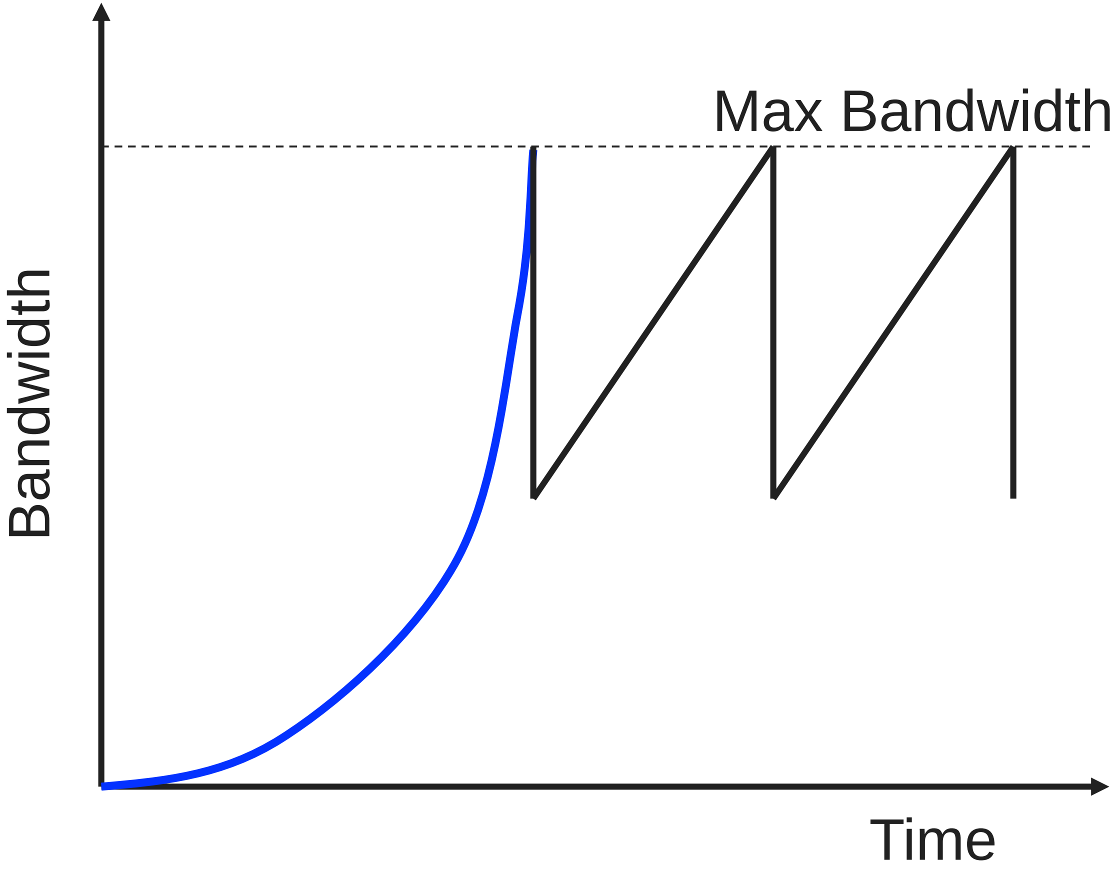


Solution: Allow TCP to increase window size by doubling *until first loss*

Initial rate is **slow** but **ramps up exponentially fast**

TCP Slow Start

- Initial phase: **exponential increase**
- Assuming no other losses in the network except those due to bandwidth



TCP Summary

- Reliable ordered message delivery
  - Connection oriented, 3-way handshake
- Transmission window for better throughput
  - Timeouts based on link parameters
- Congestion control
  - Linear increase, exponential backoff
- Fast adaptation
  - Exponential increase in the initial phase