CS519: Computer Networks
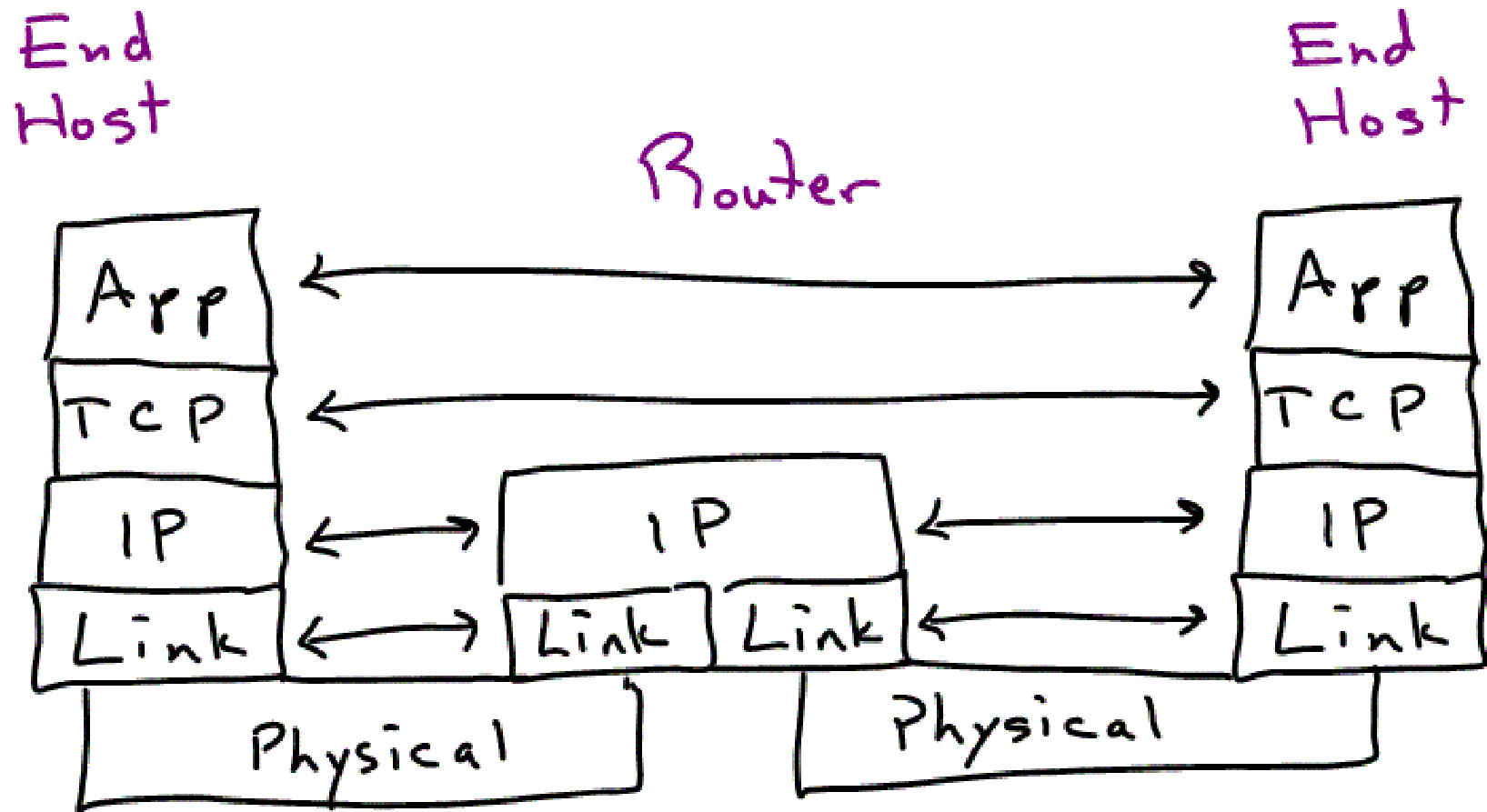
Lecture 5, Part 1: Mar 3, 2004

Transport: UDP/TCP demux and flow control / sequencing
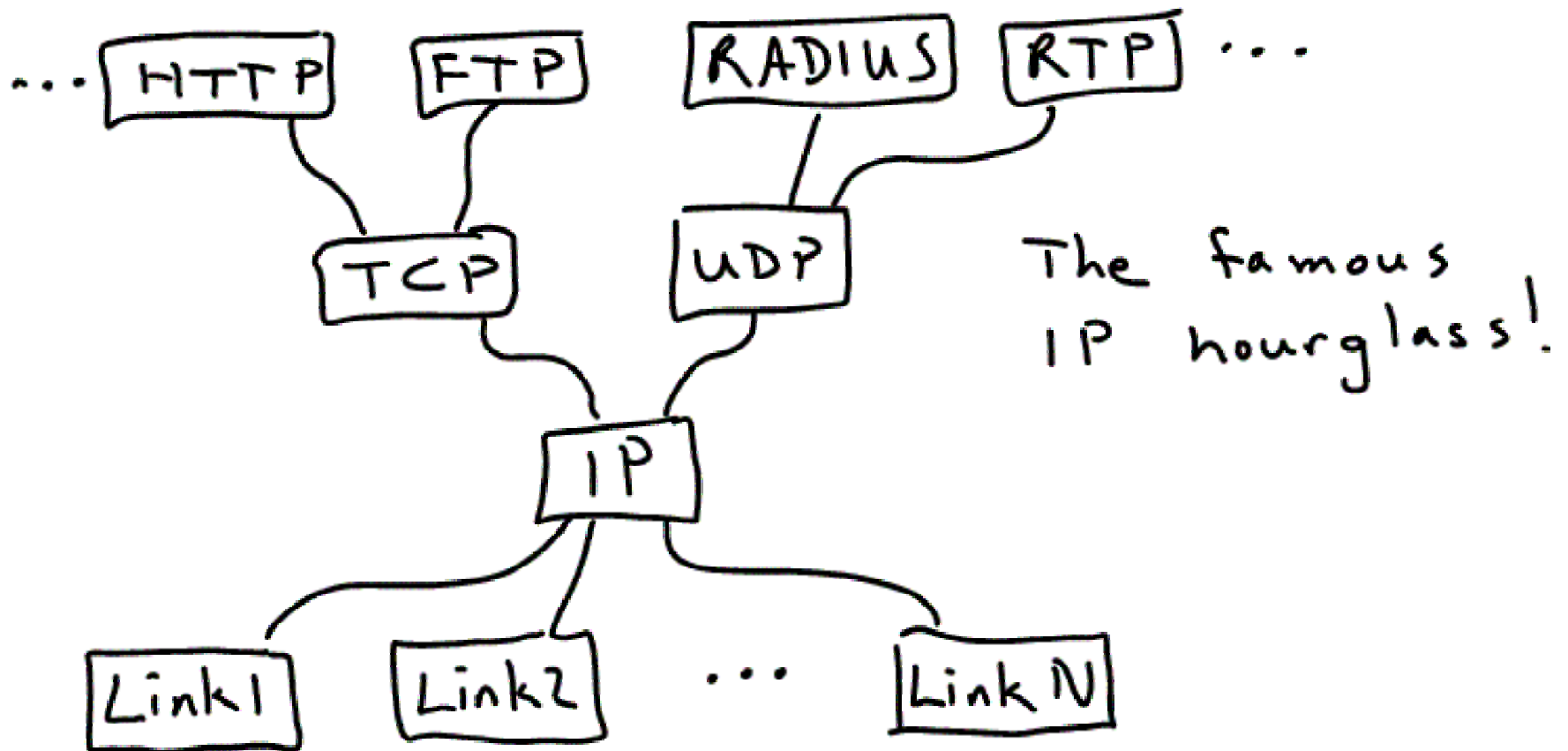
Recall our protocol layers . . .

. . . and our protocol graph

The famous IP hourglass!

CS519

- IP gets the packet to the host
  - Really the interface
- Now how do we get the packet from the interface to the right process?
- Well, you've kinda seen this already, but lets cover again

TCP and UDP ports

- The ports serve to "demux" the packet
  - Get it from the interface to the right process

UDP Header

SRC port	DST port
checksum	length
DATA	

TCP and UDP ports

- Some ports are "well-known"
  - HTTP is by default TCP port 80
  - DNS is UDP or TCP port 53
  - Etc.
- Servers listen at these ports
- Other ports are dynamically assigned
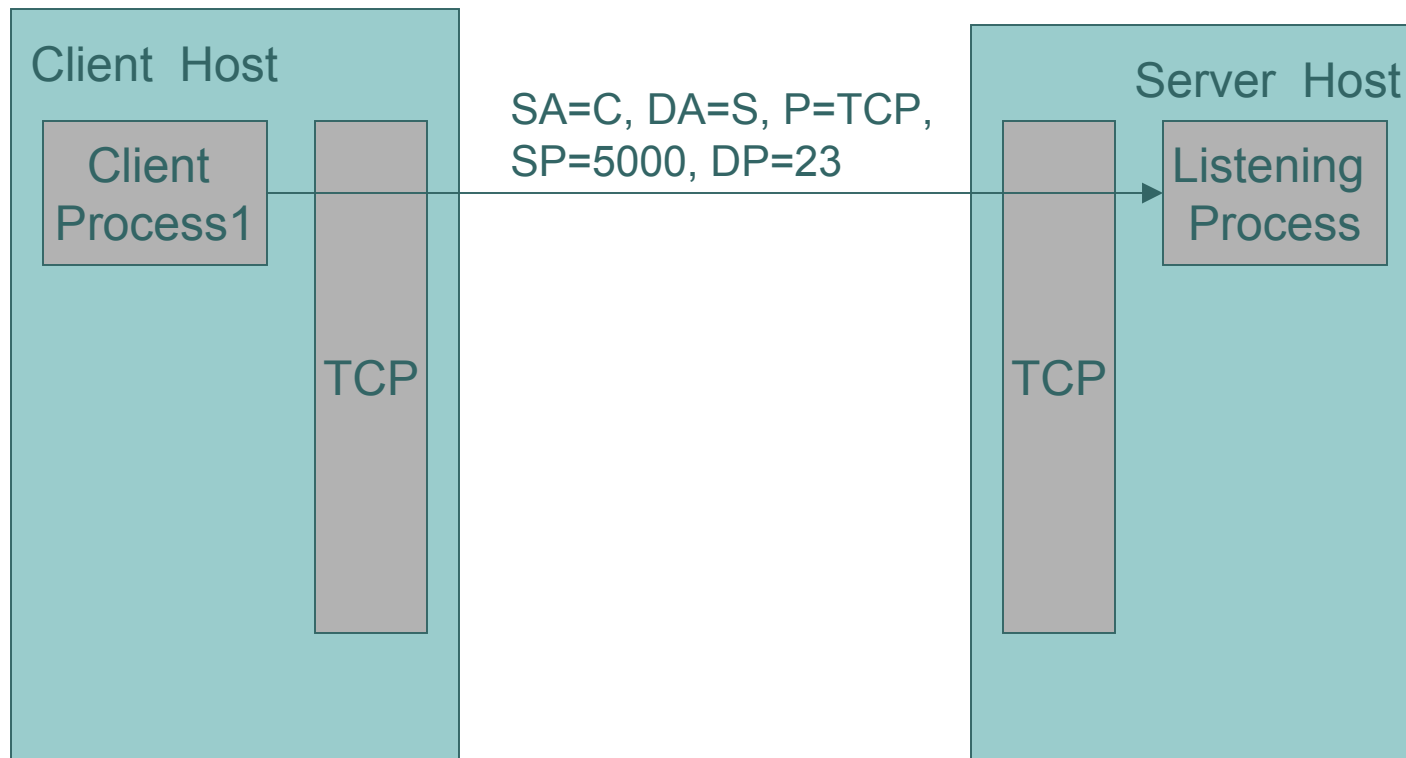 - Clients usually dynamically assign ports

UDP/TCP application process selection

CS519

- *Unicast* application process is selected by the complete 5-tuple, consisting of:
  - *Source and Dest IP address*
  - *Source and Dest port*
  - *IP protocol*
  - Ex: an FTP server may have concurrent transfers to the same client. Only the source port will differ.
- *Multicast* application process is selected by a 3-tuple: *Dest IP address and UDP port, and IP protocol*
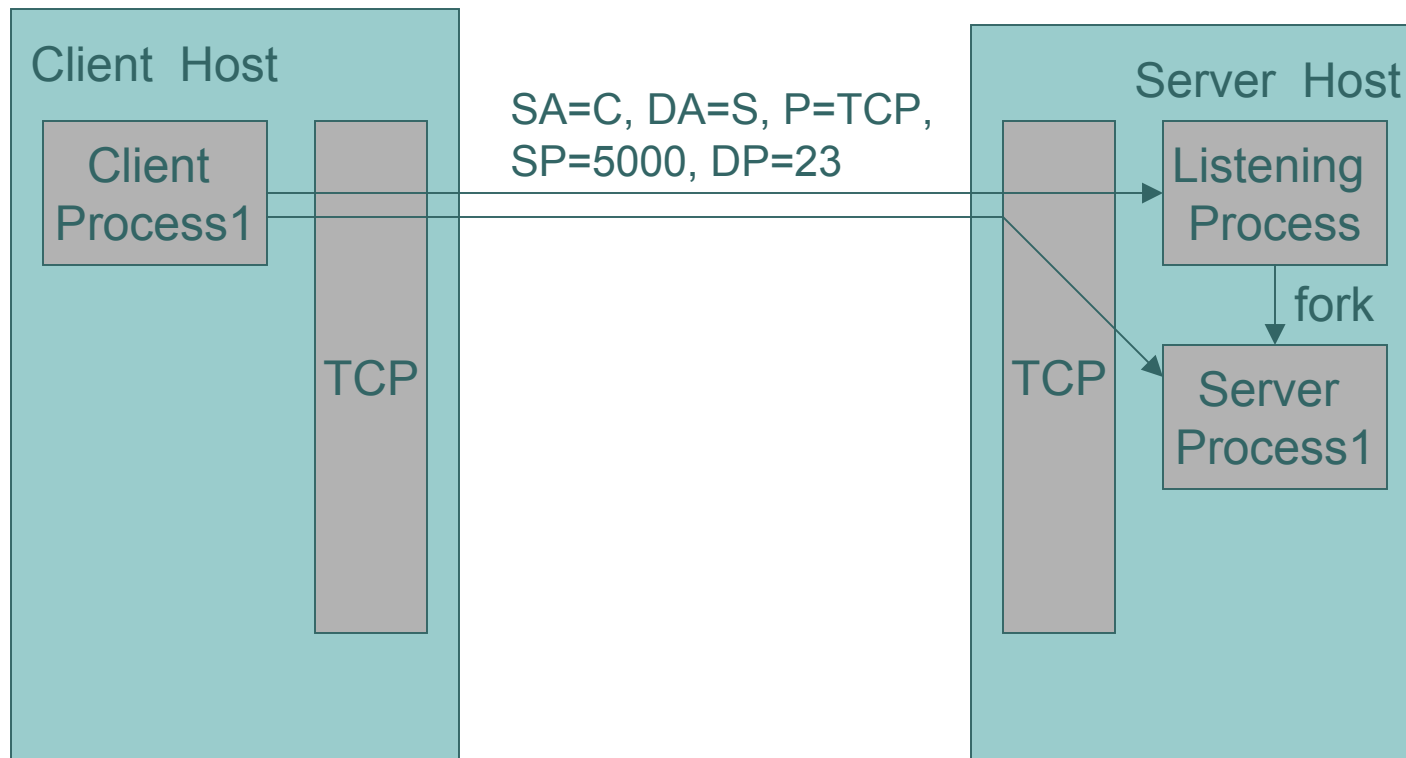 - Because it is multicast, UDP may select multiple processes
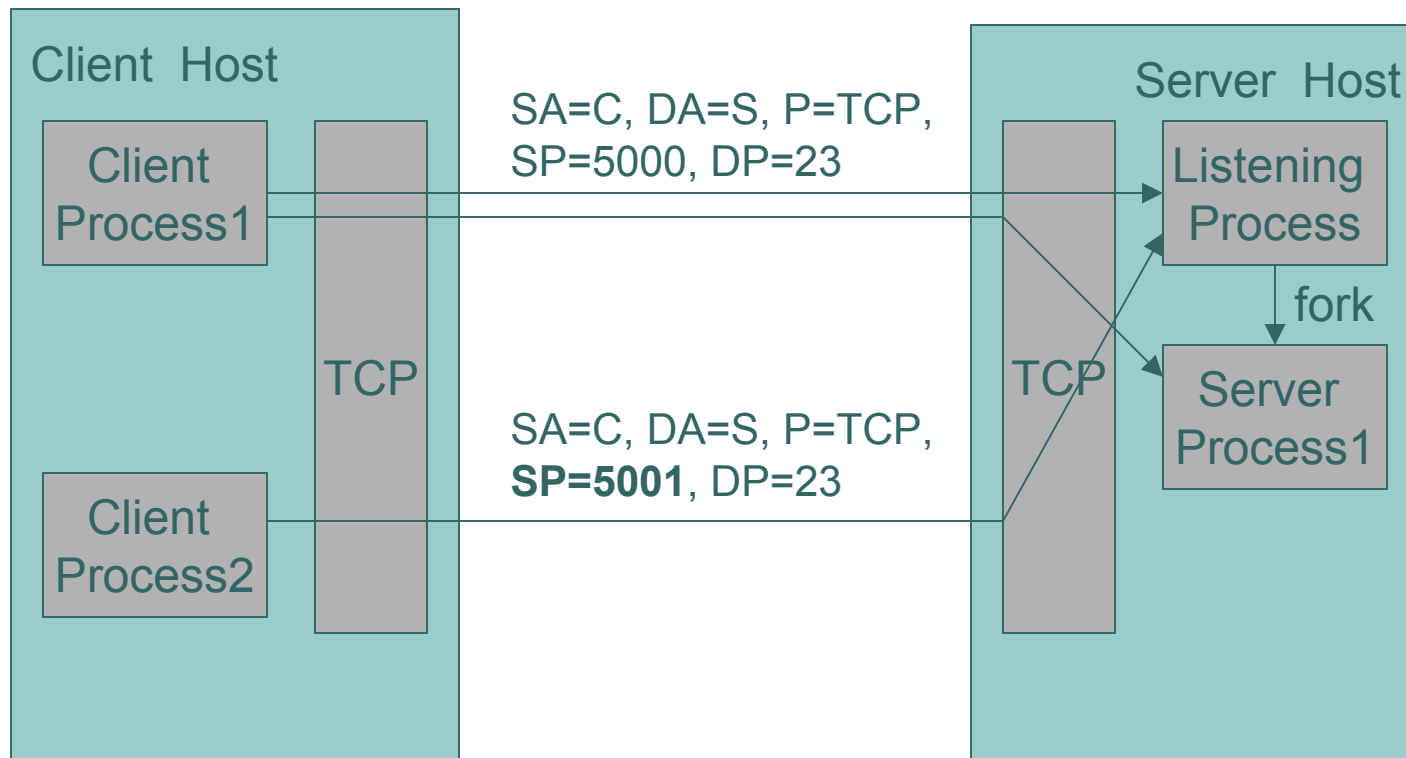
Typical server incoming connection processing

Client Host

Client Process1

TCP

SA=C, DA=S, P=TCP, SP=5000, DP=23

Server Host

Listening Process

TCP

Typical server incoming connection processing

Client Host

Client Process1

TCP

SA=C, DA=S, P=TCP, SP=5000, DP=23

Server Host

Listening Process

fork

TCP

Server Process1

Typical server incoming connection processing

Client Host

Client Process1

Client Process2

TCP

SA=C, DA=S, P=TCP, SP=5000, DP=23

SA=C, DA=S, P=TCP, **SP=5001**, DP=23

TCP

Server Host

Listening Process

fork

Server Process1

Typical server incoming connection processing

Client Host

Client Process1

TCP

SA=C, DA=S, P=TCP, SP=5000, DP=23

SA=C, DA=S, P=TCP, **SP=5001**, DP=23

Client Process2

TCP

Server Host

Listening Process

fork

Server Process1

fork

Server Process1
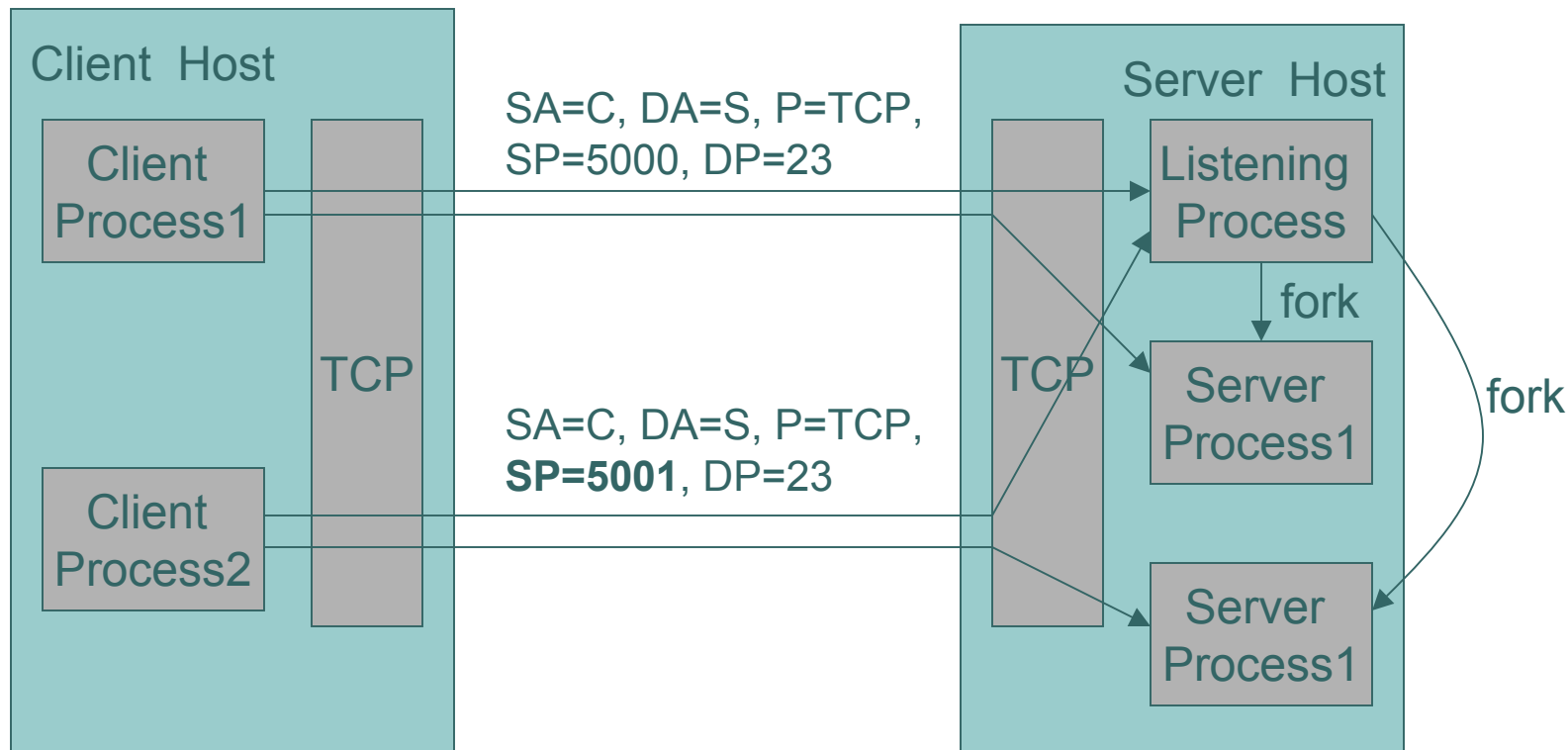
UDP and TCP service

- UDP is connectionless *packet* transport service
  - Like IP, packets can be lost, mis-ordered, duplicated
- A receive() of X bytes corresponds to a previous send() of X bytes
  - And a corresponding packet of X bytes
    - (Ignoring packet loss or other errors like not providing enough receive buffer)
- If sending app sends, but receiving app doesn't receive, packet will be lost
  - Even if no packets are lost in the network!

UDP packet loss

app

os

UDP
IP

packets can
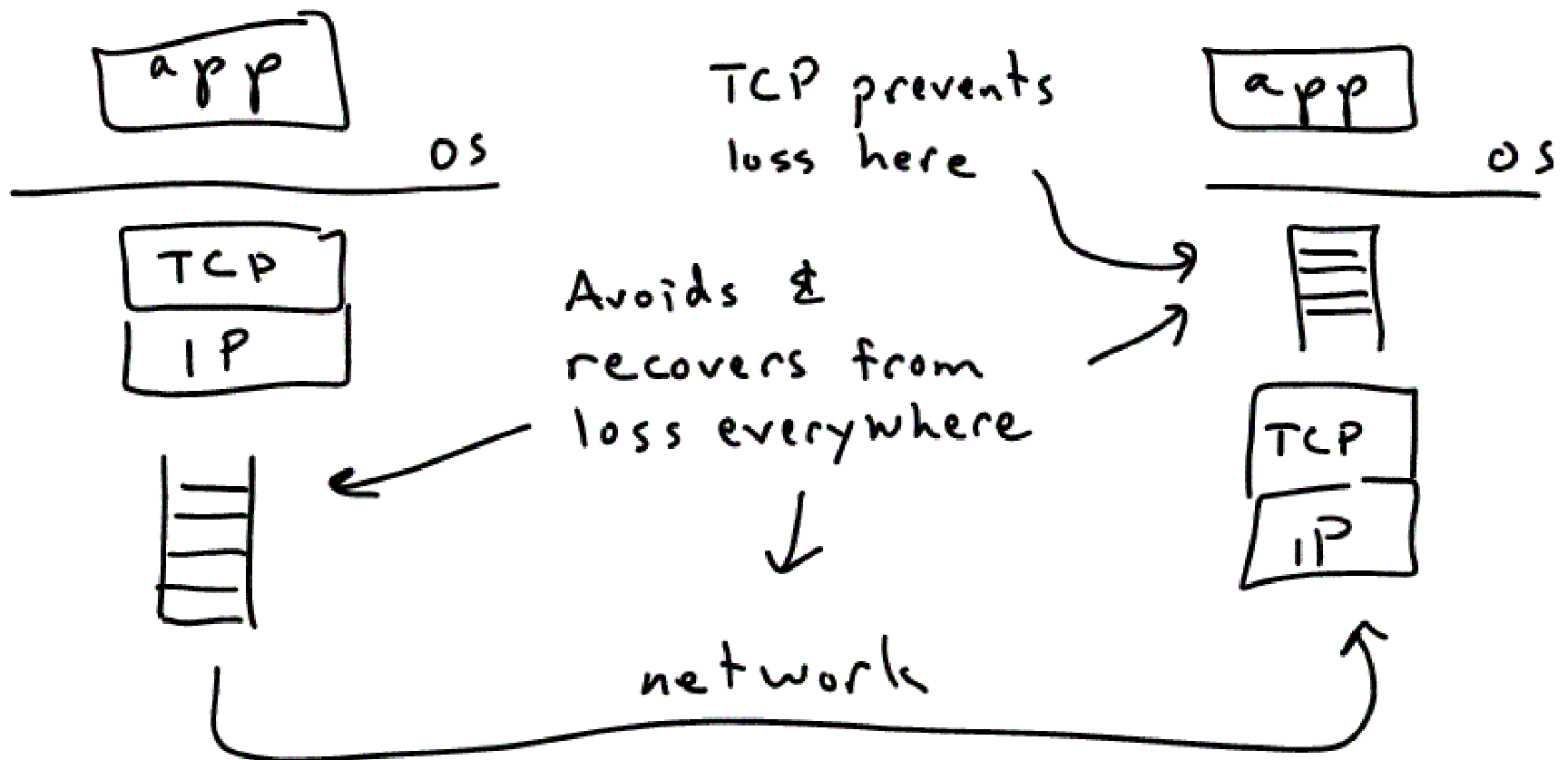be lost

app

os

UDP
IP

network
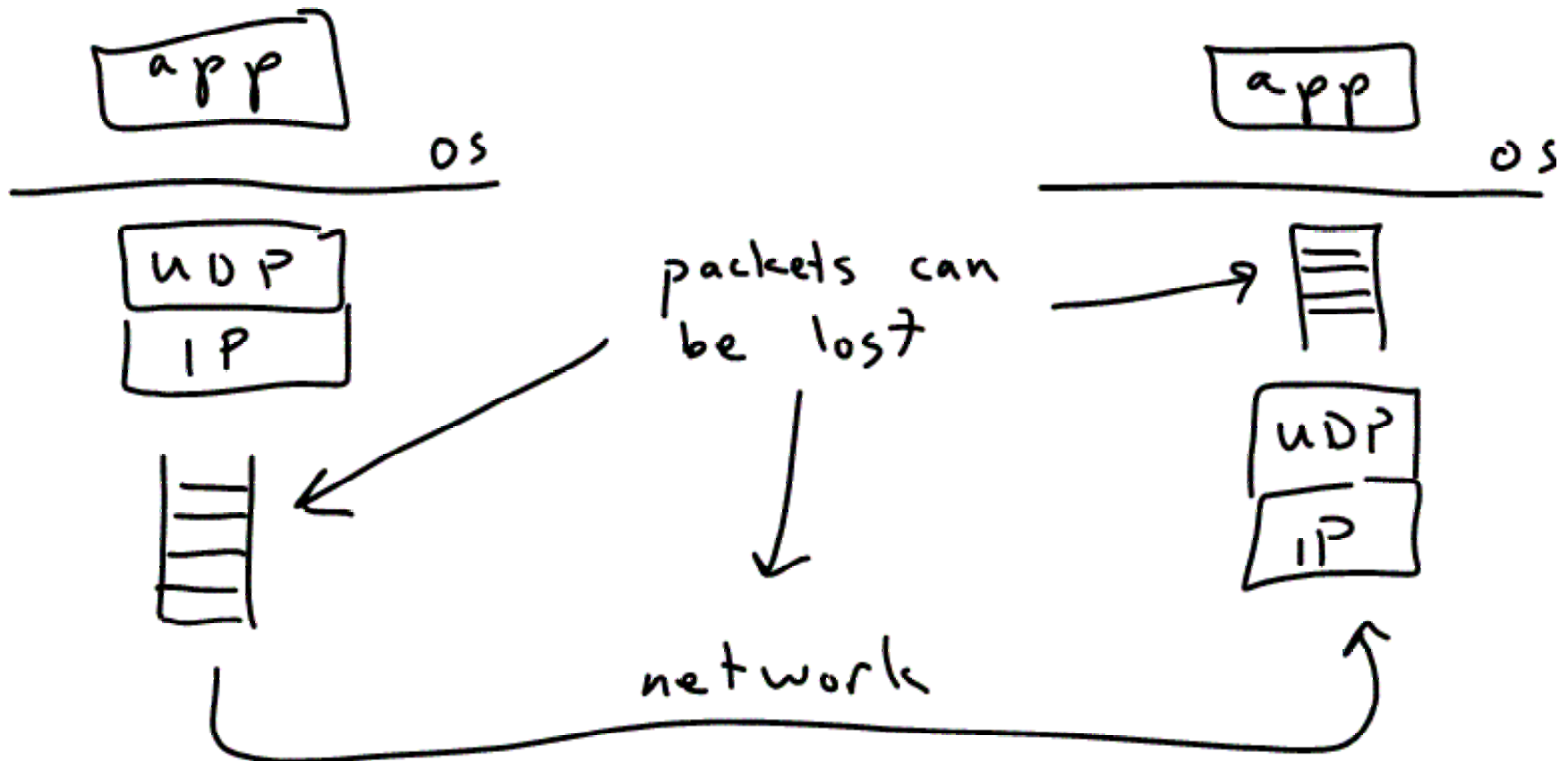
UDP and TCP service

- TCP is a reliable *byte-stream* transport service
  - As long as the TCP connection is established, bytes arrive in the order they were sent
- But, a send() of X bytes doesn't imply a receive() of X bytes
  - Sender can send 500 bytes, and receiver can read 1 byte 500 times (and it could have been transmitted as 2 250-byte packets)
  - And vice versa
- TCP provides flow control

TCP flow control

app

os

TCP
IP

TCP prevents
loss here

Avoids &
recovers from
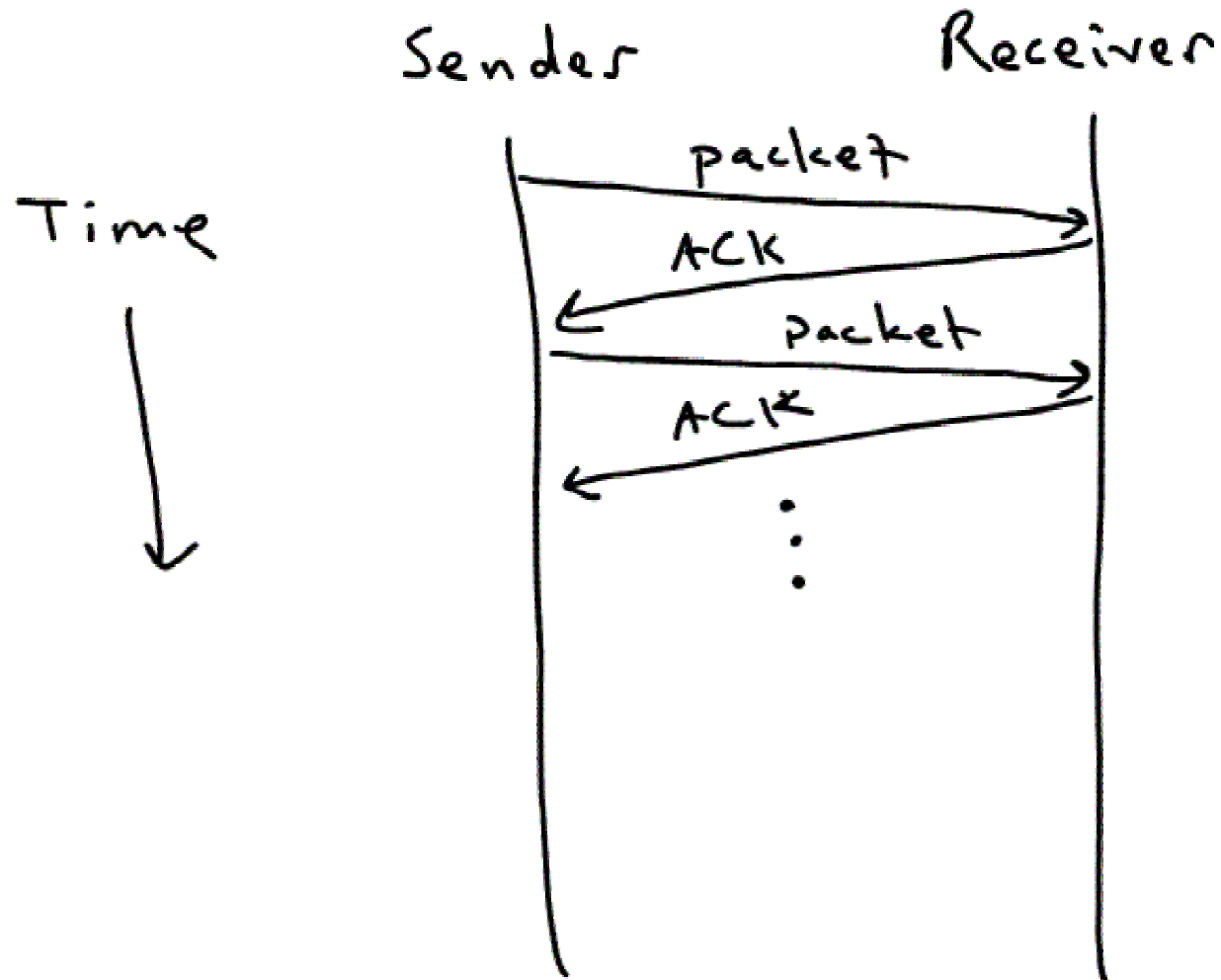loss everywhere

app

os

TCP
IP

network

Stop-and-wait

- Before looking at TCP in its full glory, lets look at simpler sequencing / flow control algorithms

- Stop-and-wait is about as simple as it can get

- Sender sends packet, waits for ack, sends another packet, . . .

- Receiver receives packet, acks it . . .

Stop-and-wait

Sender   Receiver
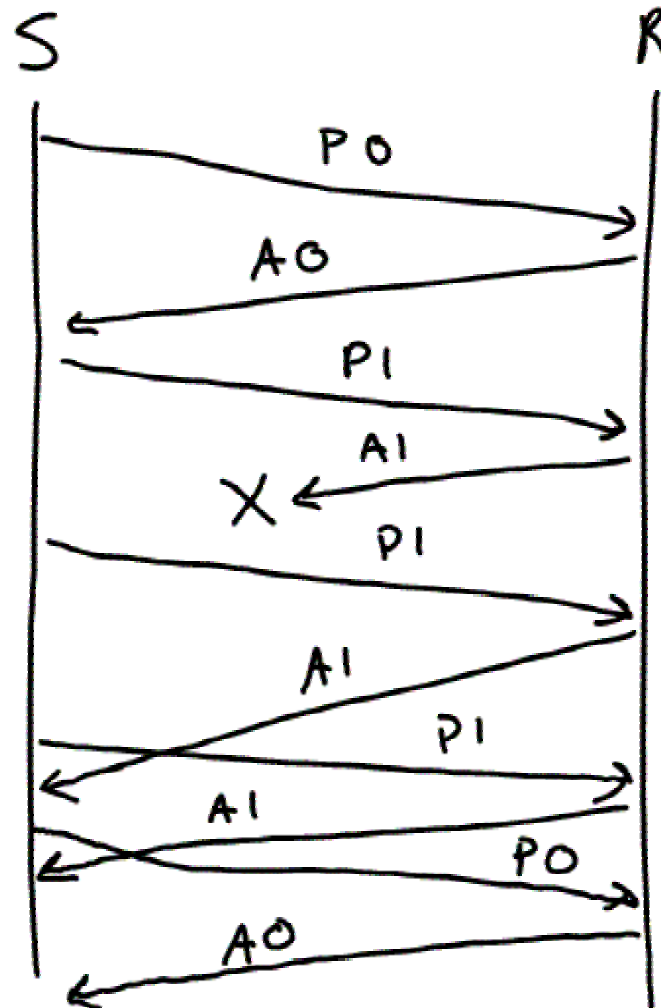
Time

packet

ACK

packet

ACK

Stop-and-wait

- Receiver only needs one packet's worth of receive buffer
  - Only send ACK after received packet is processed
- Sender only needs one packet's worth of send buffer
  - Save packet until get ACK, then save the next packet

Even stop-and-wait not quite this simple!

Stop-and-wait requires a 1-bit sequence number space
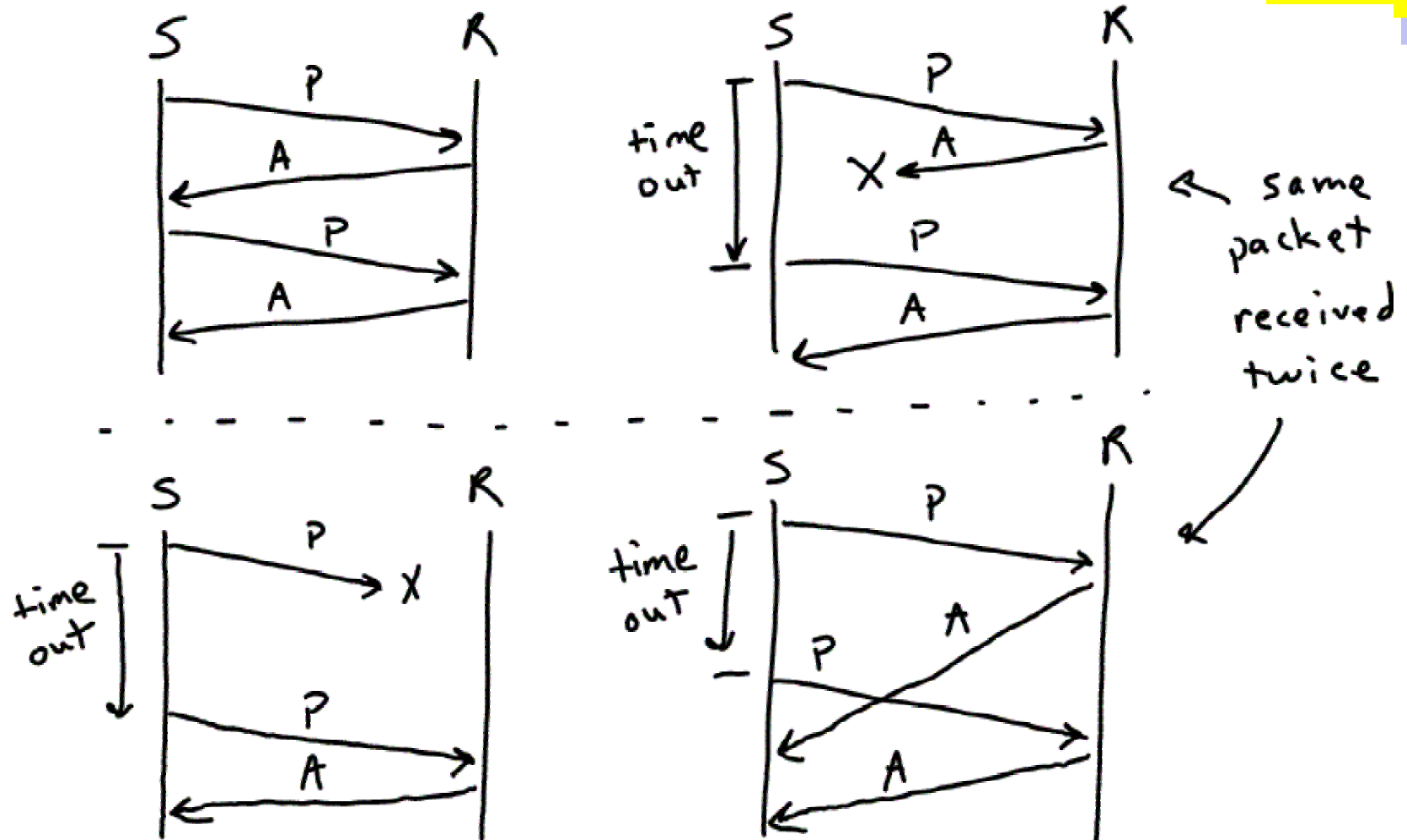
Problem with stop-and-wait

- Fine on a short-skinny pipe
  - Low bandwidth, low distance
- Wasteful on a long-fat pipe
  - High delay x bandwidth product
- 1.5 Mbps link, 45ms round-trip delay
  - Approx. 8KB BW x delay
- Eight 1KB packets can be sent in one RTT, but stop-and-wait only sends one packet in one RTT

Sliding window

- Sender can send multiple bytes before getting an ACK for the first byte
  - Number of bytes is the *send window*
  - Sender must buffer these bytes in case it has to retransmit
- Receiver can buffer multiple bytes before delivering any to the application
  - Number of bytes is the *receive window*
  - Receiver must buffer these bytes in case application doesn't read them on time
  - Or in case some bytes not received

Sliding window

S   R
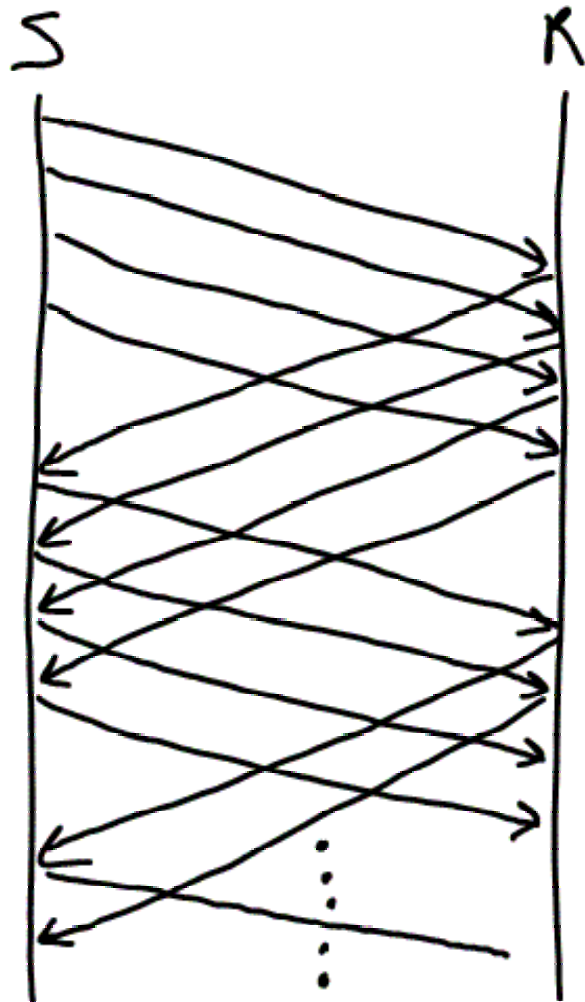
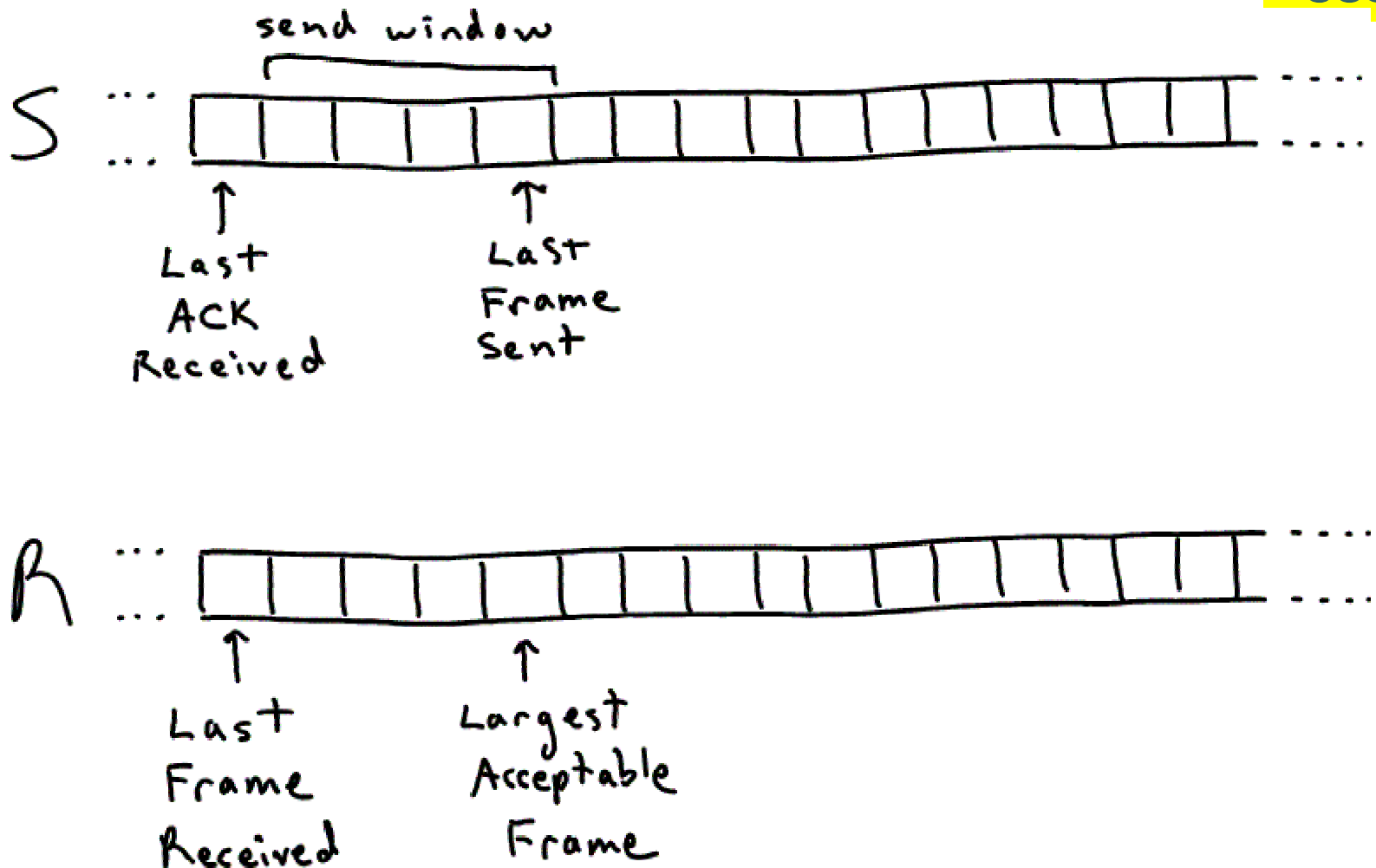Send window of four "packets".

Still not big enough to "fill the pipe"

Send and receive window sizes

- Send window should be big enough to fill the pipe
- Receive window can be smaller than send window
  - As long as receiver can keep up with sender
  - But packet loss can result in more retransmits than necessary
- No point in making receive window bigger than send window
  - Unless congestion in network a concern

Sliding window examples

send window

S ... | | | | | | | | | | | | | | | | | | ...

↑ Last ACK Received

↑ Last Frame Sent

R ... | | | | | | | | | | | | | | | | | | ...

↑ Last Frame Received
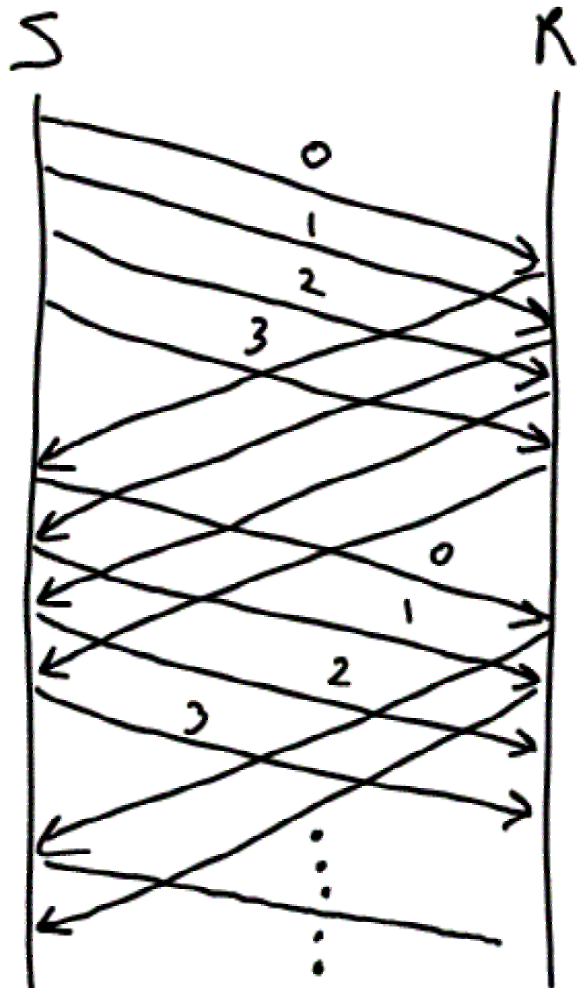
↑ Largest Acceptable Frame

Sliding window examples

- Normal operation
- Receive app delays reading
- Packet lost
- Cumulative ACK
- NACK
- Selective ACK

Seq number space must be at least two times window size

S   R

If seq num space = 4

Receiver doesn't know
if these are retransmits
(cause ACKs lost) or
new packets

Next lecture

- Next few lectures will be all about TCP