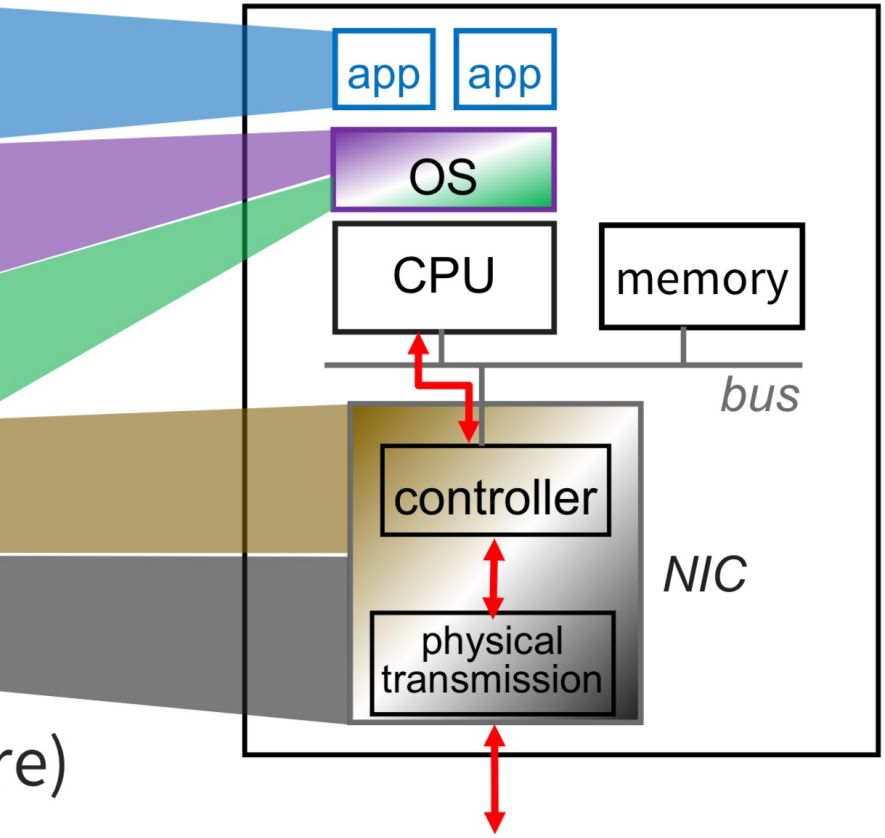
Lecture 26: TCP

CS 105

Spring 2024

OSI Network Model

Application	HTTP, FTP, DNS (*these^* are usually in libraries)
Transport	TCP, UDP
Network	IP, ICMP (ping)
Link	Ethernet, WiFi
Physical	wires, signal encoding



(Hard to draw firm lines here)

Transport Layer Protocols

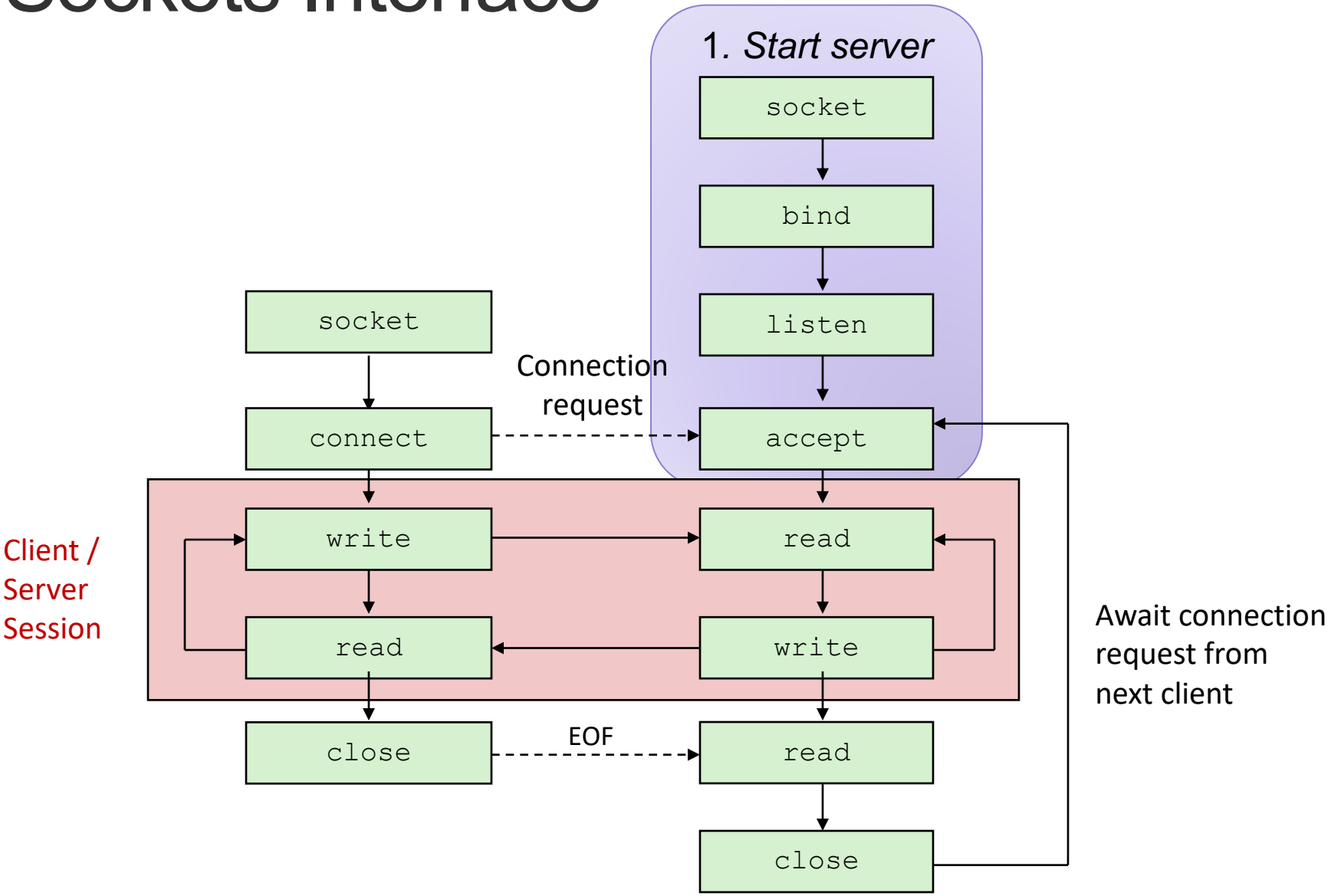
User Datagram Protocol (UDP)

- **unreliable, unordered delivery**
- connectionless
- best-effort, segments might be lost, delivered out-of-order, duplicated
- reliability (if required) is the responsibility of the app

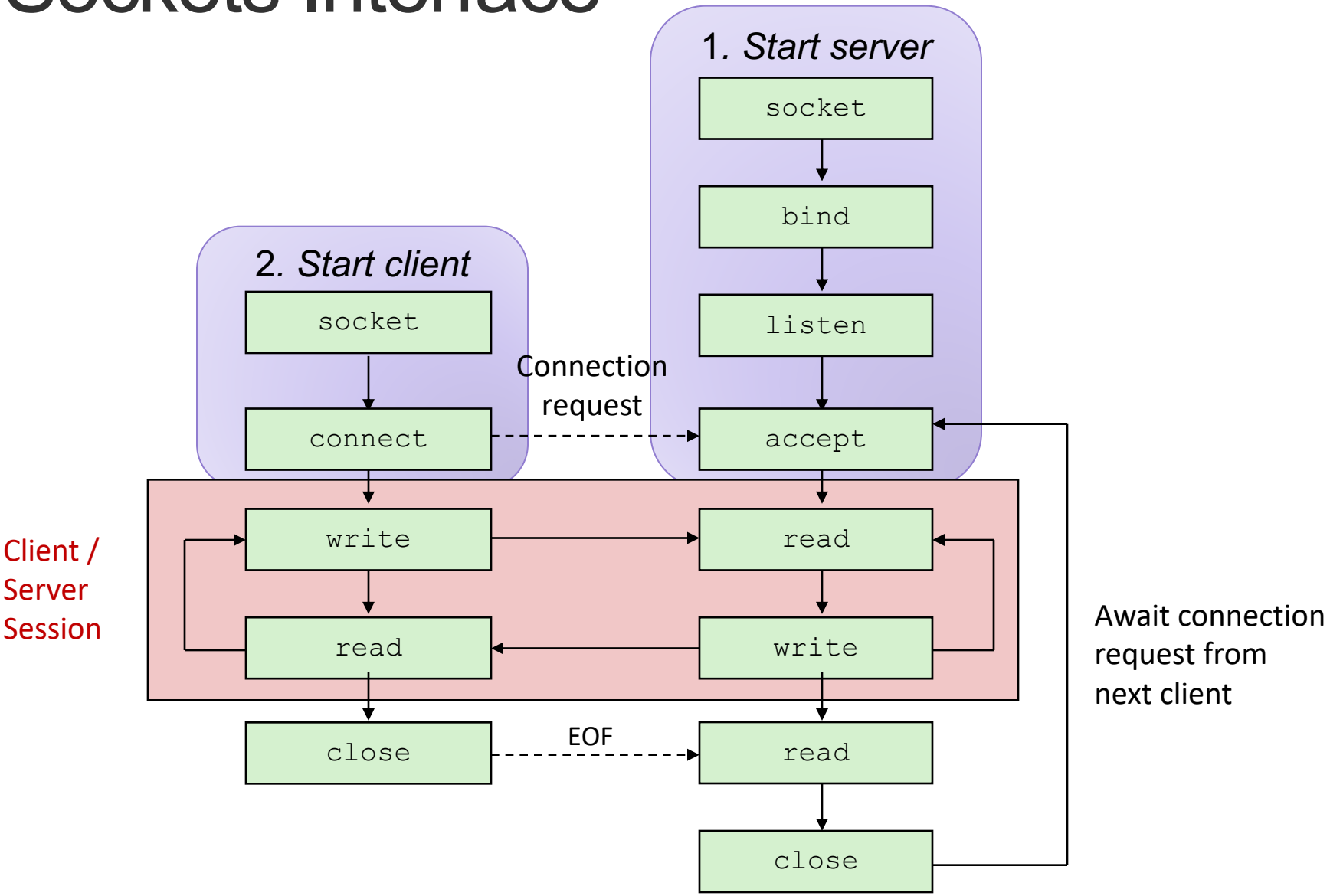
Transmission Control Protocol (TCP)

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

Sockets Interface

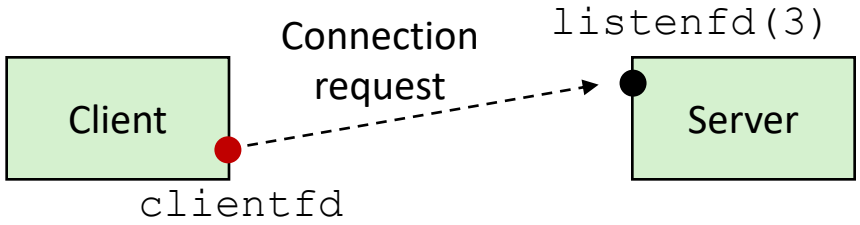


Sockets Interface

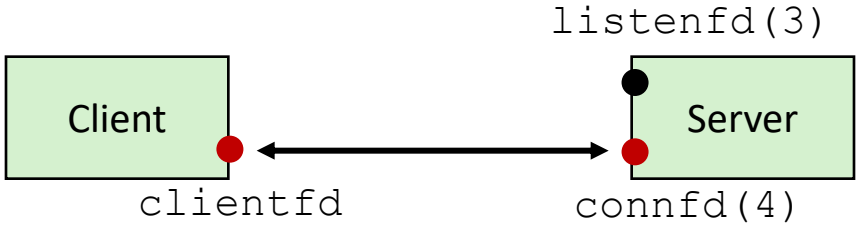


`accept` Illustrated

1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`



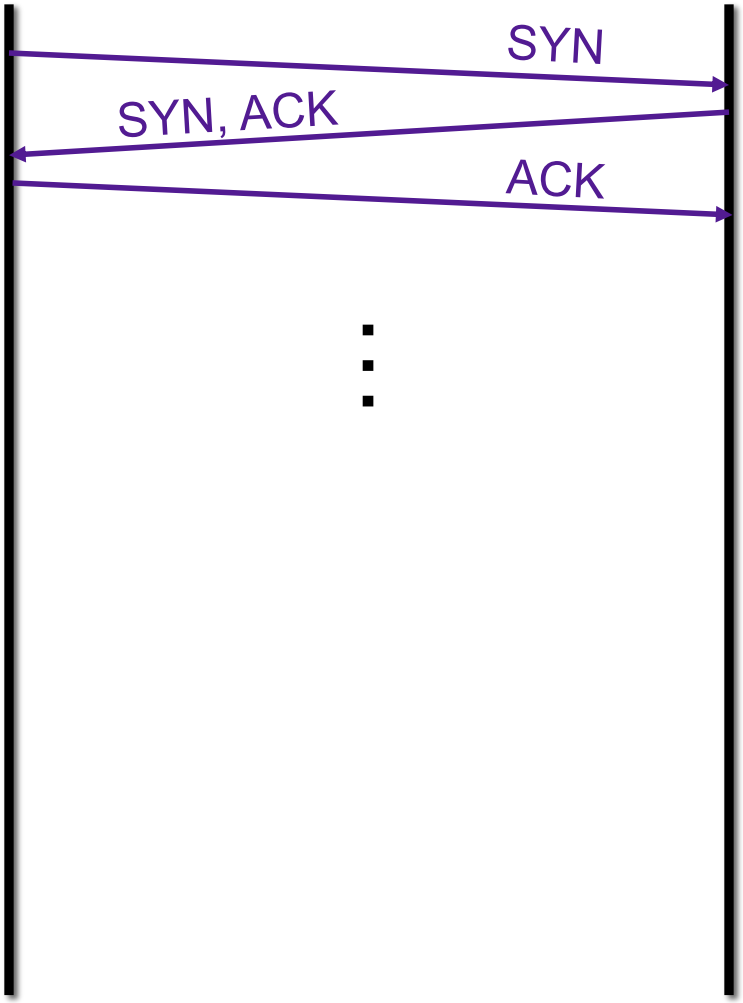
2. Client makes connection request by calling and blocking in `connect`



3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`

TCP Connections

- TCP is connection-oriented
- A connection is initiated with a three-way handshake
- Recall: server will typically create a new socket to handle the new connection

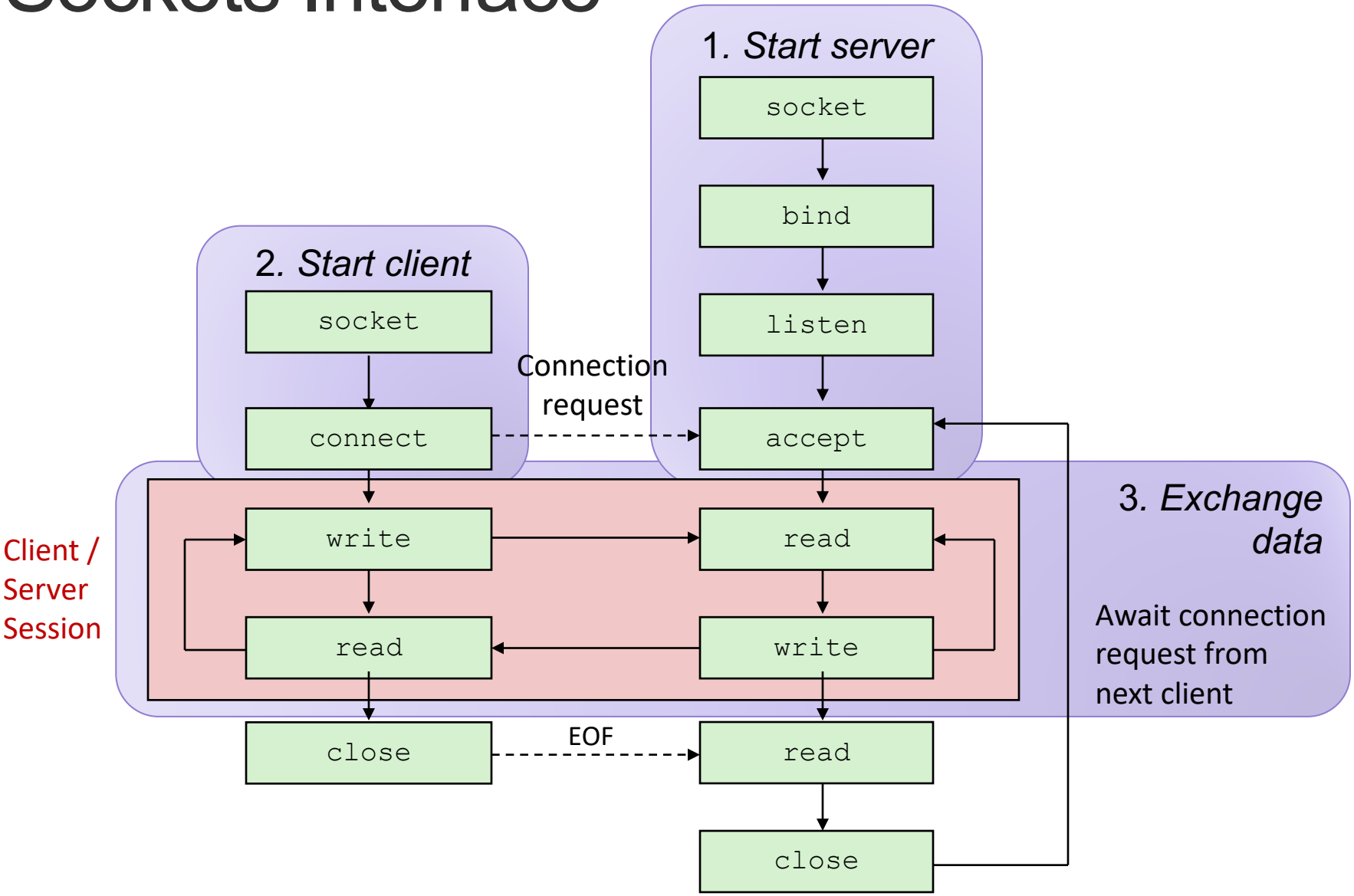


Exercise: Connection Setup

- Consider the network operations we've discussed thus far: socket, bind, listen, accept, connect. What sequence are these operations called in if a client wants to send one message to the server?

client **server**

Sockets Interface



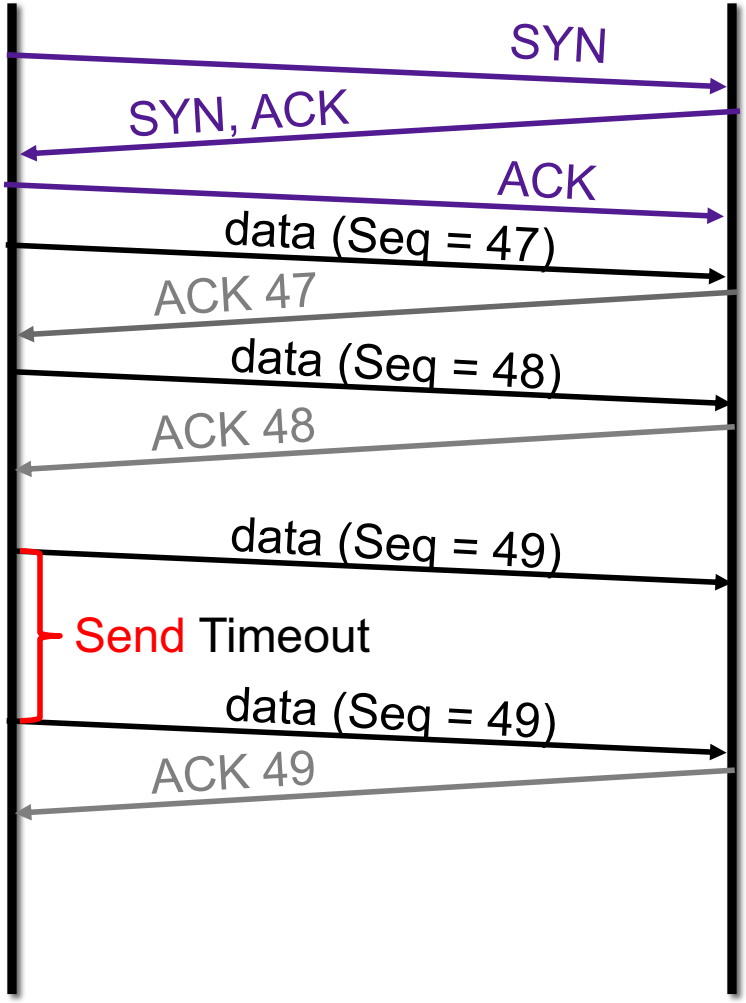
Communicating over a channel

- Consider the network operations we've discussed thus far: socket, bind, listen, accept, connect. What sequence are these operations called in if a client wants to send one message to the server?

client **server**

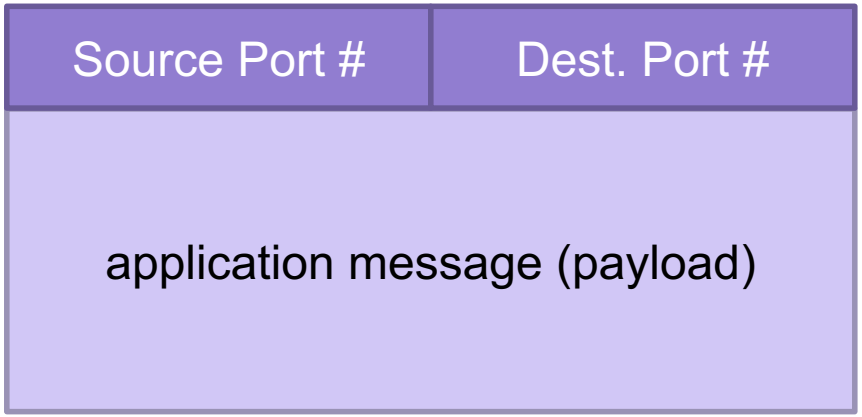
Reliable Transport

- Each SYN segment will include a randomly chosen sequence number
- Sequence number of each segment is incremented by data length
- Receiver sends ACK segments acknowledging latest sequence number received
- Sender maintains copy of all sent but unacknowledged segments; resends if ACK does not arrive within timeout
- Timeout is dynamically adjusted to account for round-trip delay

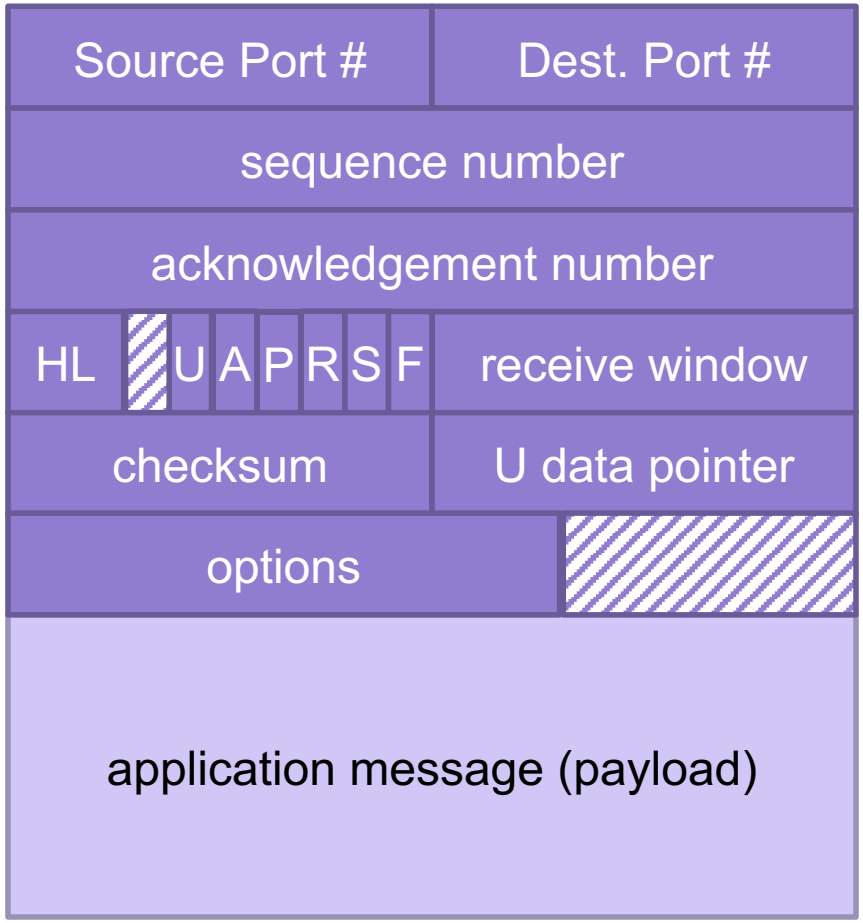


Transport-Layer Segment Formats

UDP

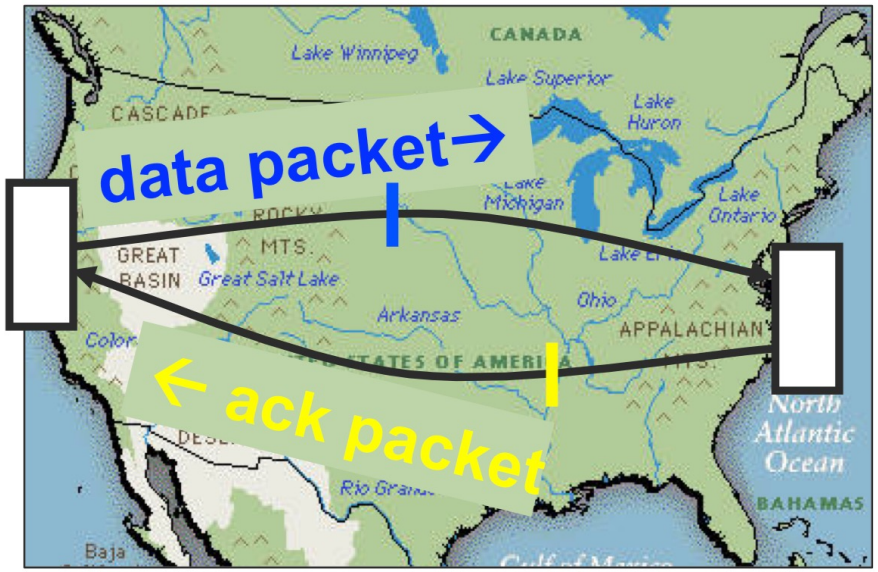


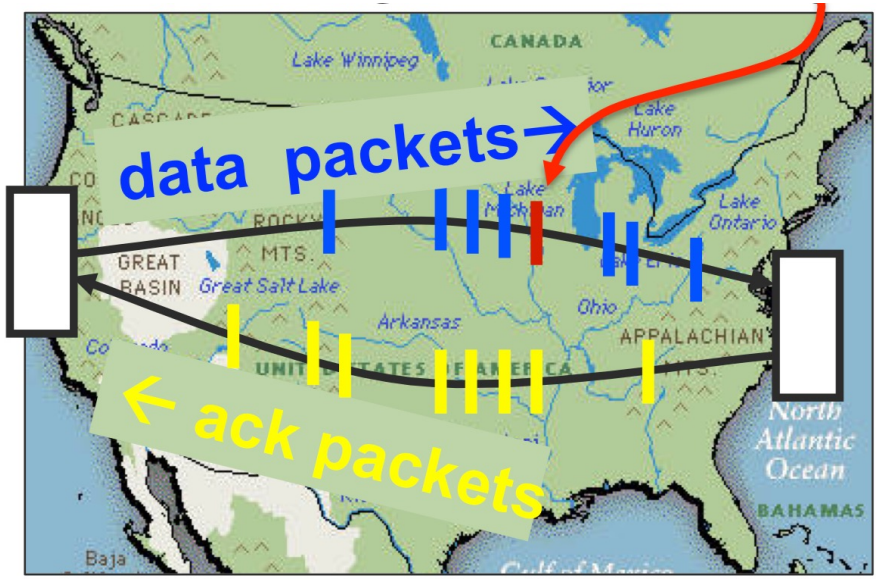
TCP



Pipelined Protocols

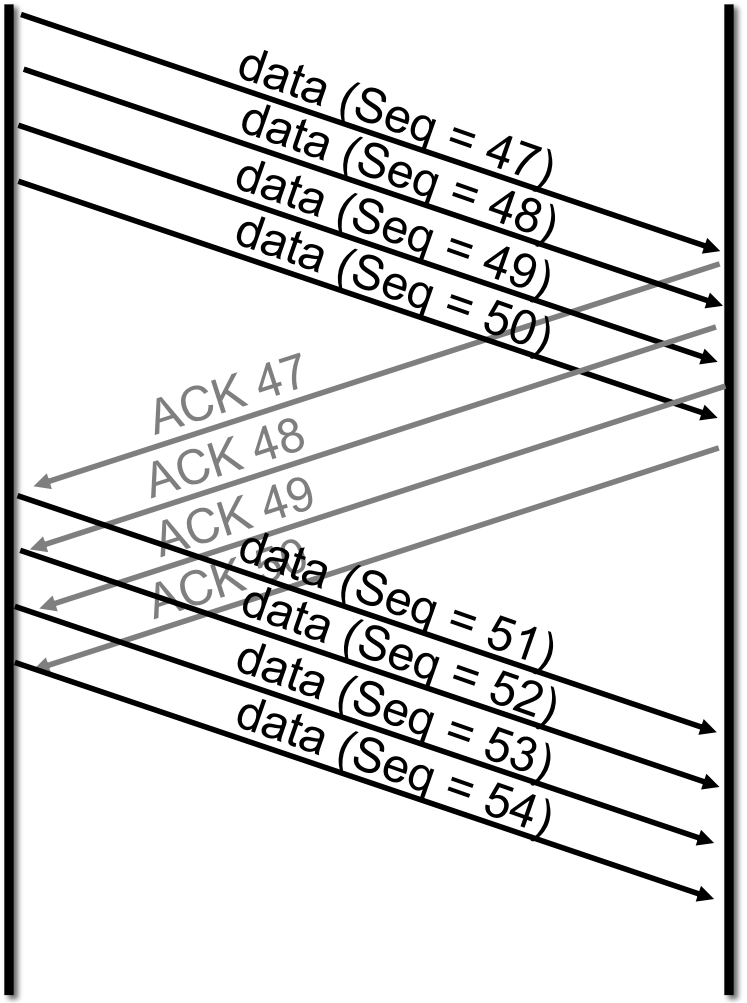
- Pipelining allows sender to send multiple "in-flight", yet-to-be-acknowledged packets
  - increases throughput
  - needs buffering at sender and receiver





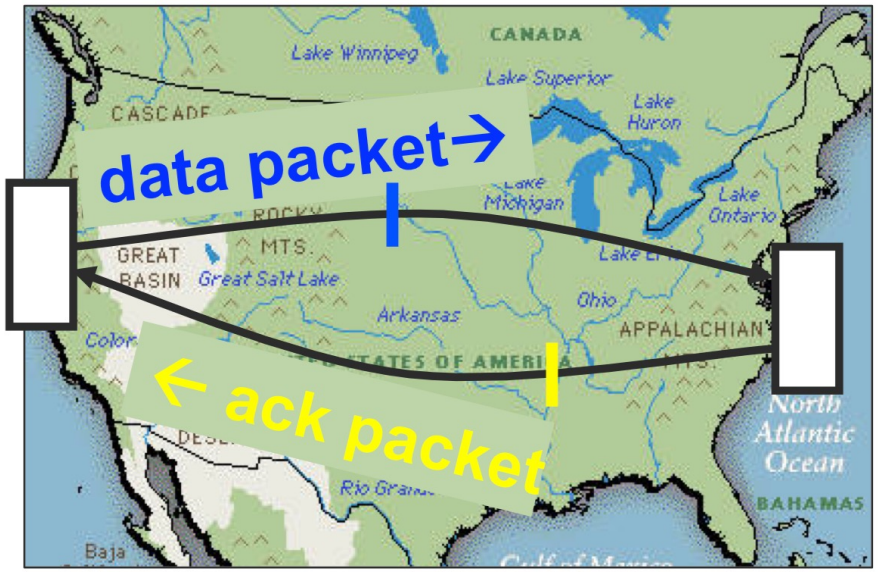
Example: Window Size = 4

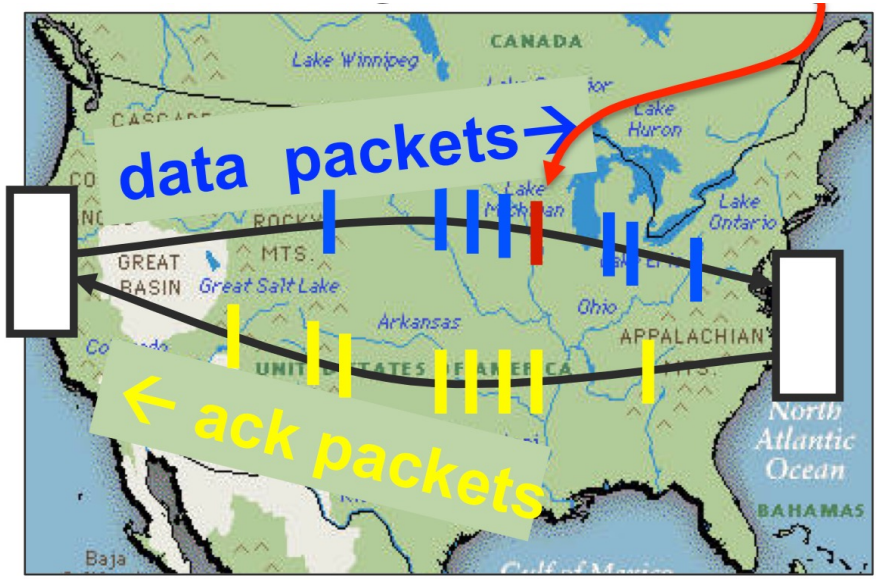
- sender can have up to 4 unacknowledged messages
- when ACK for first message is received, it can send another message



Pipelined Protocols

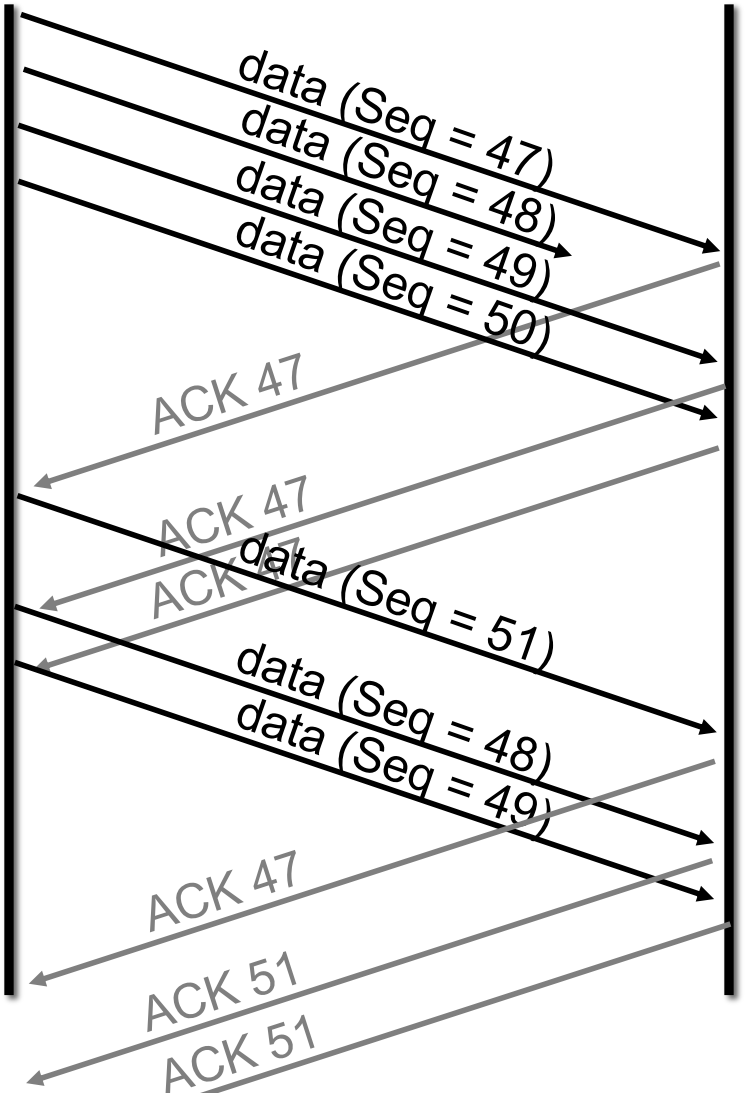
- Pipelining allows sender to send multiple "in-flight", yet-to-be-acknowledged packets
  - increases throughput
  - needs buffering at sender and receiver
- what should we do if a packet goes missing in the middle?





TCP Fast Retransmit

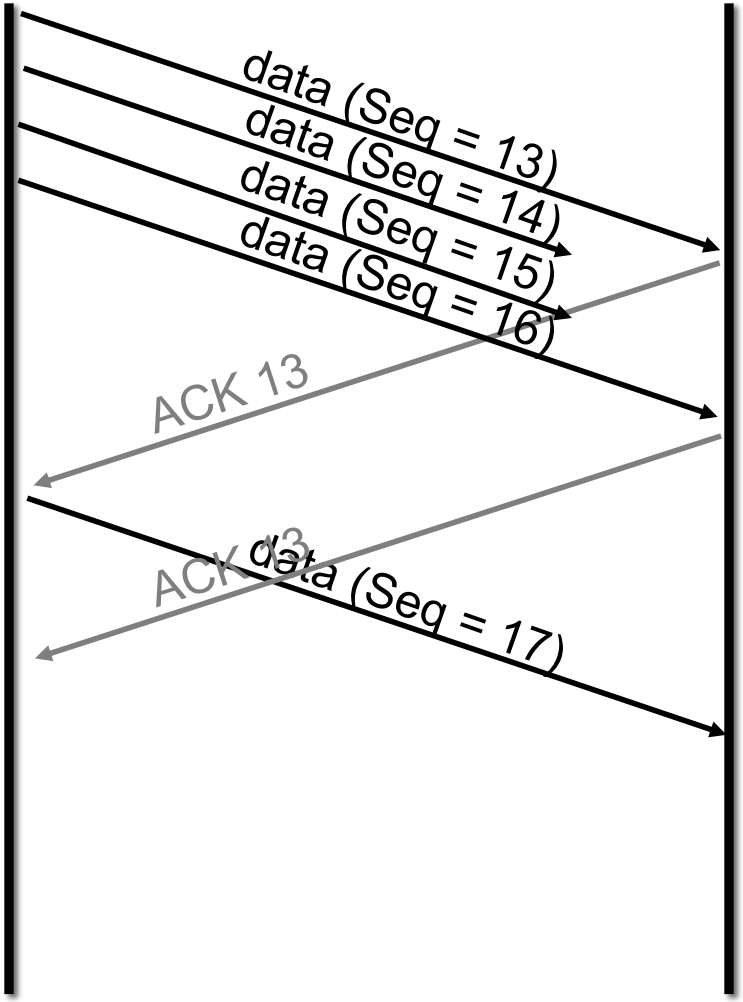
- Receiver always acks the last id it successfully received
- Sender detects loss without waiting for timeout, resends missing packet



Exercise: TCP Sequence Numbers

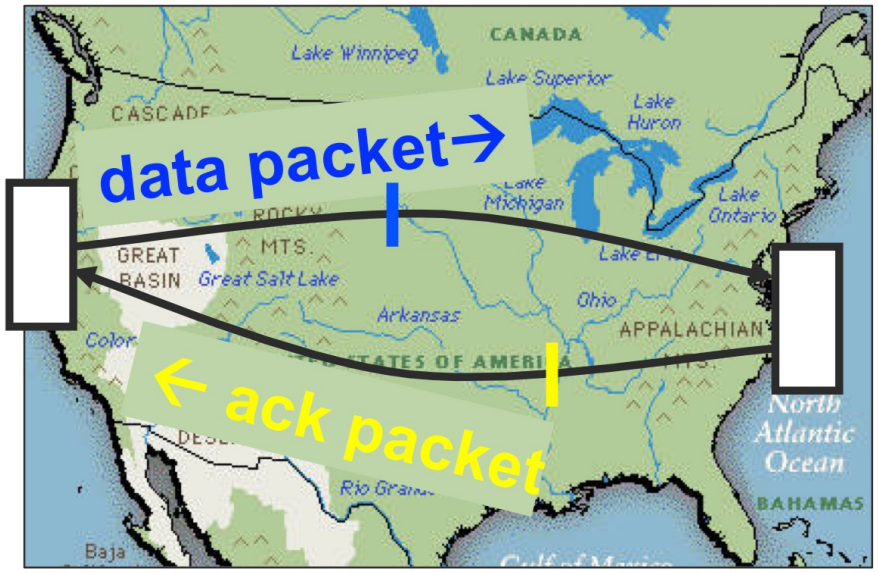
Consider the sequence of transmitted messages shown on the right

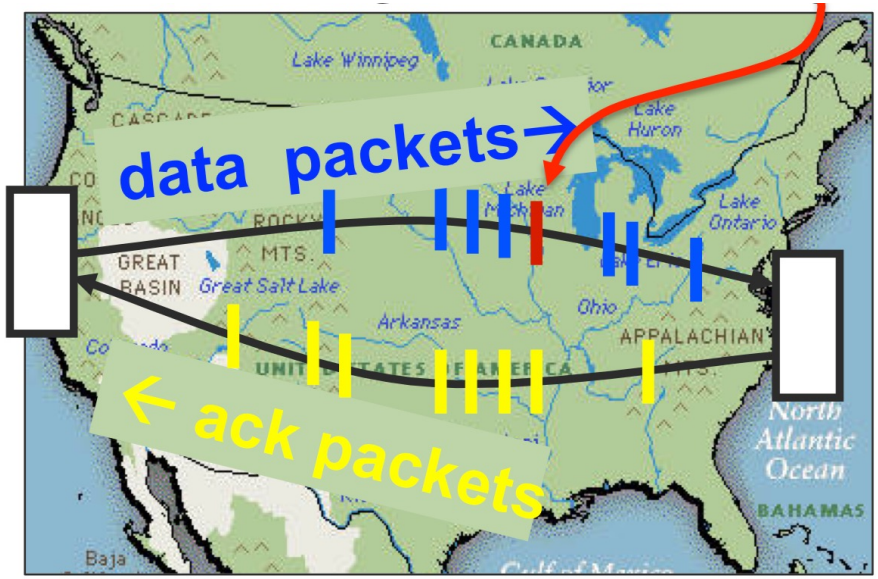
- What will be the next ACK number sent by the server?
- What will be the next Seq number sent by the client?



Pipelined Protocols

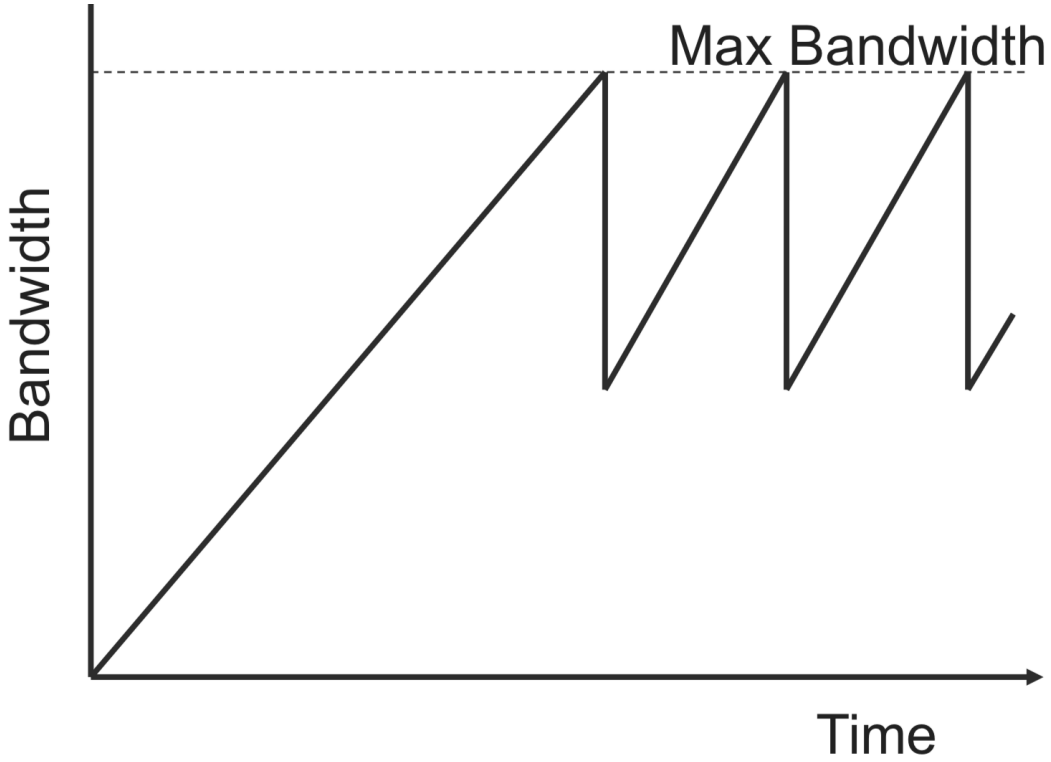
- Pipelining allows sender to send multiple "in-flight", yet-to-be-acknowledged packets
  - increases throughput
  - needs buffering at sender and receiver
- what should we do if a packet goes missing in the middle?
- how big should the window be?





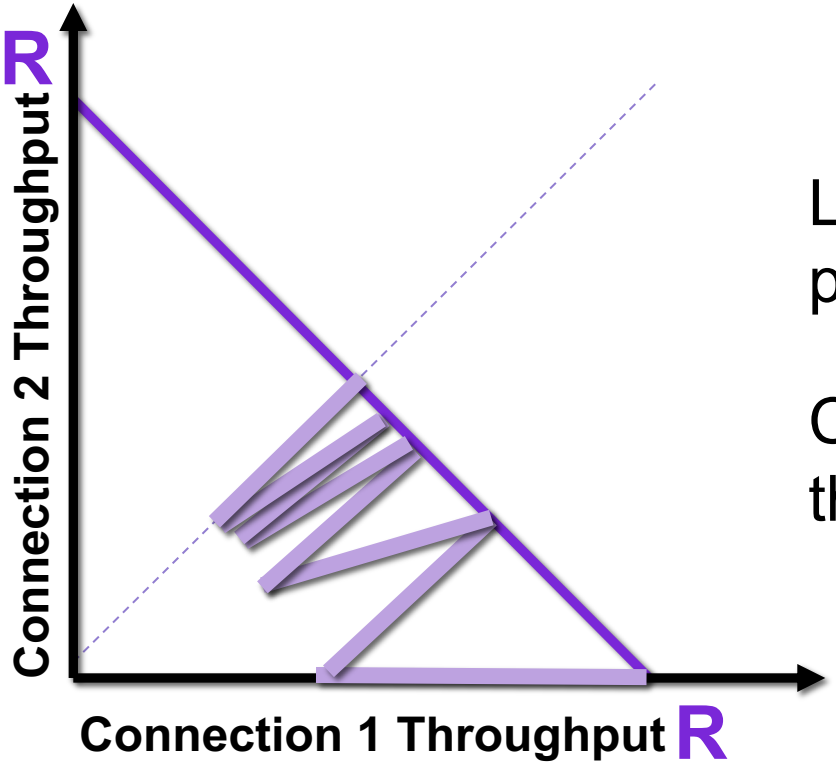
TCP Congestion Control

- TCP operates under a principle of additive increase-multiplicative decrease
  - window size++ every RTT if no packets lost
  - window size/2 if a packet is dropped



TCP Fairness

- Goal: if k TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/k



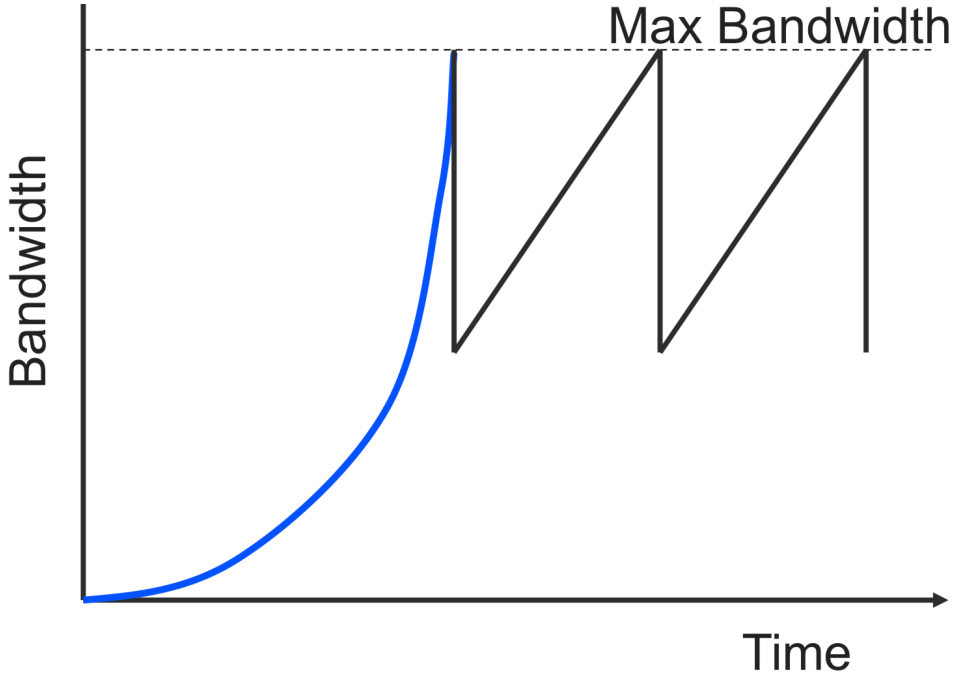
Loss: decreases throughput proportional to current bandwidth

Congestion avoidance: increases throughput linearly (evenly)

TCP Slow Start

- Problem: linear increase takes a long time to build up a decent window size, and most transactions are small

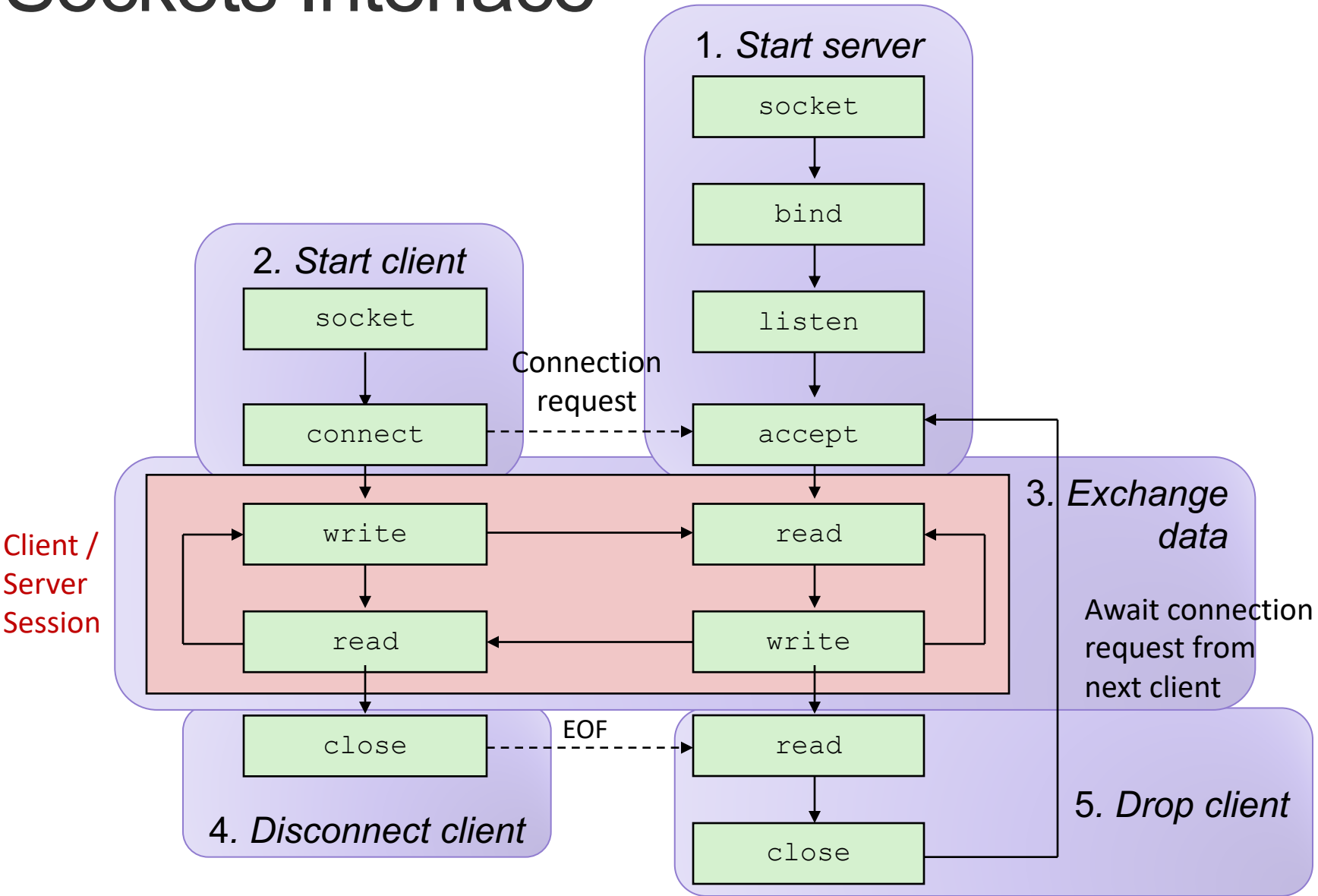
- Solution: allow window size to increase exponentially until first loss



Exercise: TCP Window Size

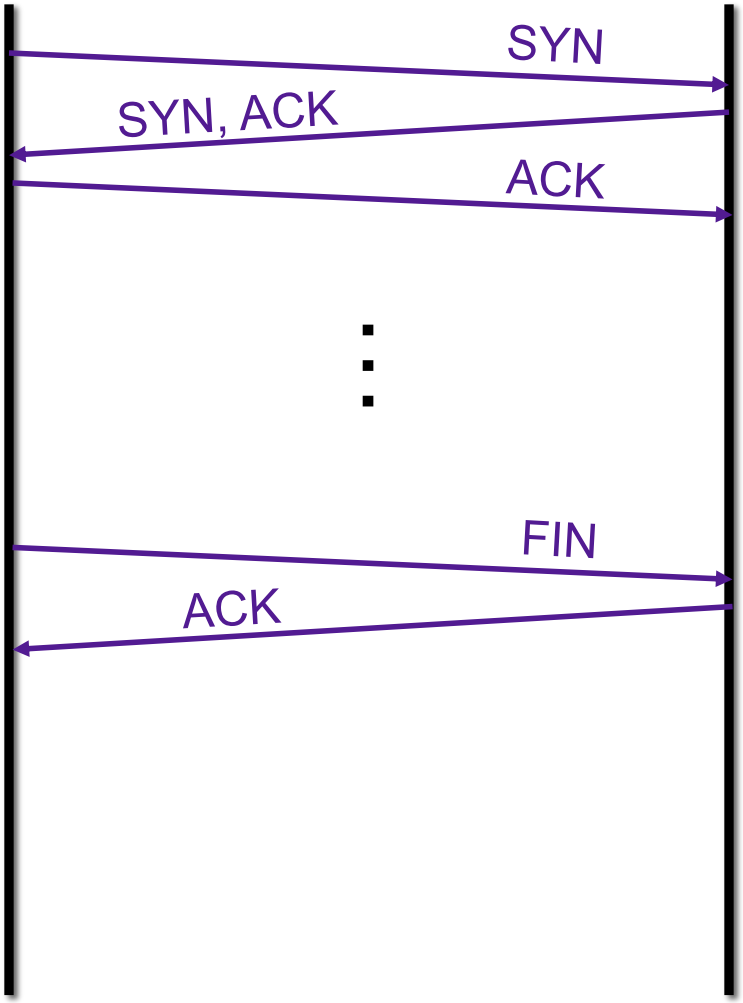
- Assume someone changes the code of their TCP client by modifying the congestion avoidance as follows: instead of increasing the window size by 1 each time an ACK is received, they double the window size each time an ACK is received (like in the slow-start phase).
- What would be the pros and cons of this modification?

Sockets Interface



TCP Connections

- TCP is connection-oriented
- A connection is initiated with a three-way handshake
- Recall: server will typically create a new socket to handle the new connection
- FIN works (mostly) like SYN but to teardown a connection



TCP Summary

- Reliable, in-order message delivery
- Connection-oriented, three-way handshake
- Transmission window for better throughput
  - timeouts based on link parameters (e.g., RTT, variance)
- Congestion control
  - Linear increase, exponential backoff
- Fast adaptation
  - Exponential increase in the initial phase