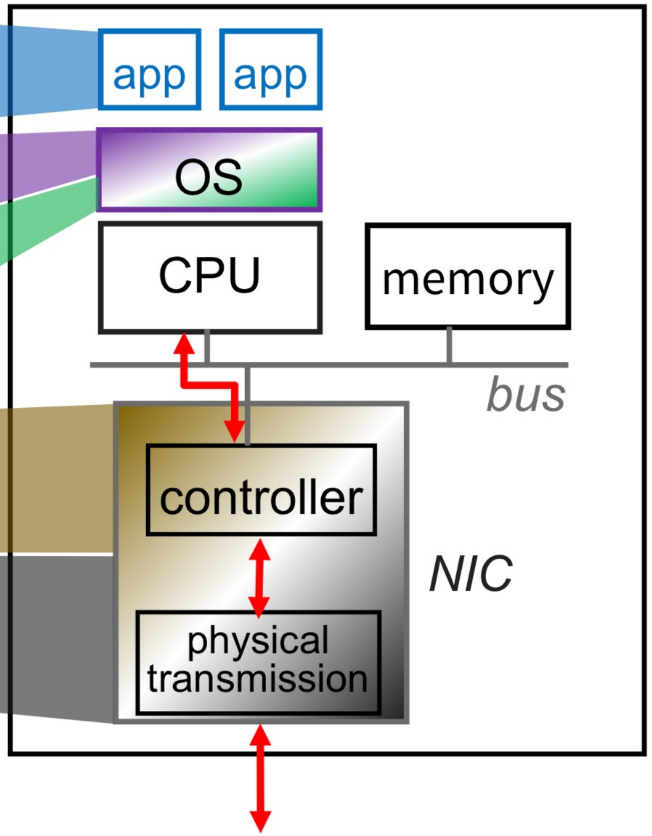
TCP

Transmission Control Protocol

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

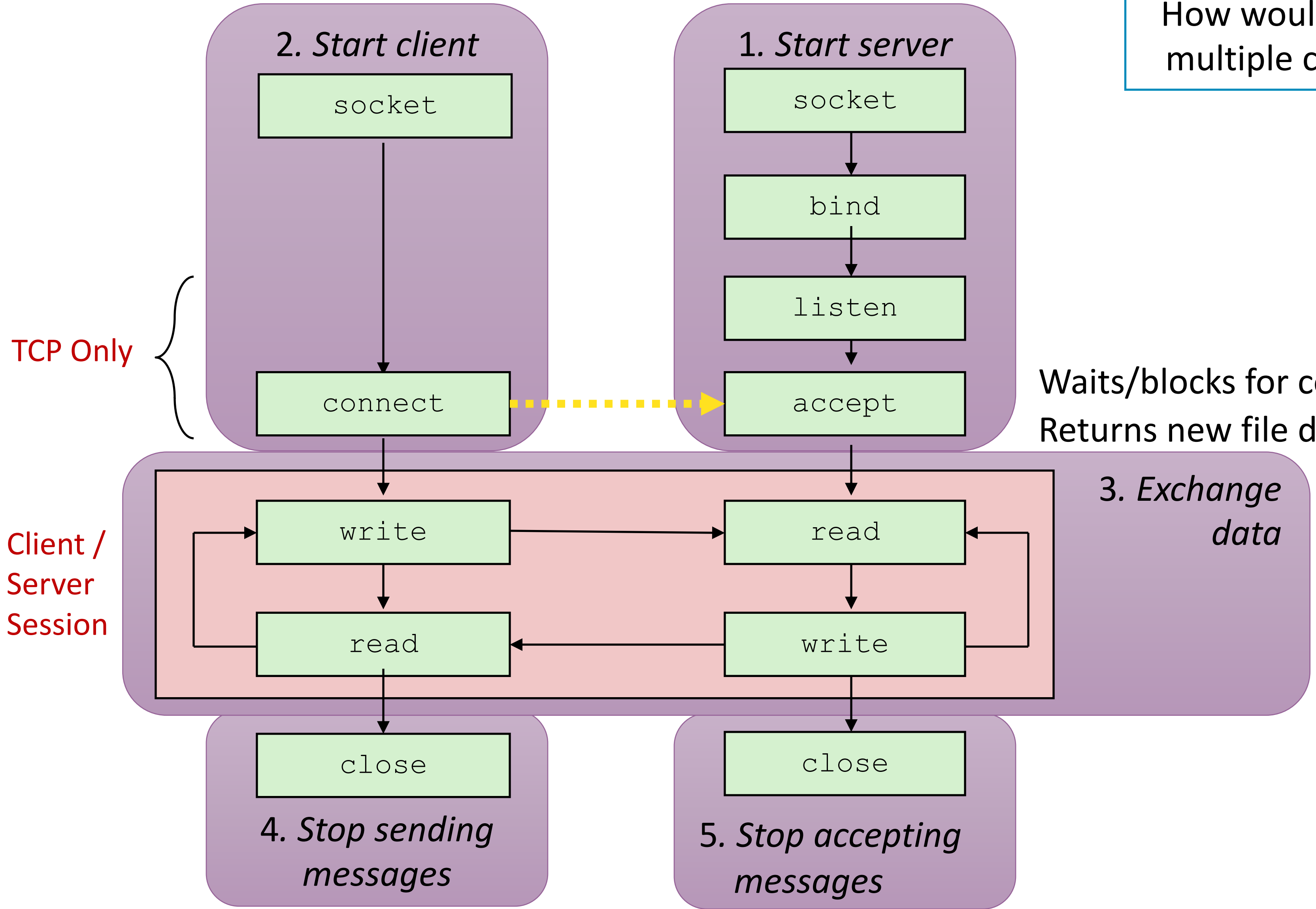
Transmission Control Protocol (TCP)

- Reliable, inorder delivery
- Connection setup
- Flow control
- Congestion control

Note: neither guarantees latency or bandwidth

Sockets Interface

How would you accept multiple connections?



Waits/blocks for connection
Returns new file descriptor for read/write

Sockets Interface: `socket`

- Clients and servers use the `socket` function to create a *socket descriptor*:

```
int socket(int domain, int type, int protocol)
```

- Example:

```
int clientfd = socket(AF_INET, SOCK_DGRAM, 0);
```

```
int clientfd = socket(AF_INET, SOCK_STREAM, 0);
```

Indicates that we are using 32-bit IPV4 addresses

Indicates transport protocol

Protocol specific! Best practice is to use `getaddrinfo` to generate the parameters automatically, so that code is protocol independent.

```
getaddrinfo("www.example.com", "http", &hints, &res);
int s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

Sockets Interface: `bind`

- A server uses `bind` to ask the kernel to associate the server's socket address with a socket descriptor:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

- The process can read bytes that arrive on the connection whose endpoint is `addr` by reading from descriptor `sockfd`.
- Similarly, writes to `sockfd` are transferred along connection whose endpoint is `addr`.
- Protocol specific! Best practice is to use `getaddrinfo` to generate the parameters automatically, so that code is protocol independent.

Sockets Interface: `listen`

- By default, kernel assumes that descriptor from socket function is an active socket that will be on the client end of a connection.

- A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client:

```
int listen(int sockfd, int backlog);
```

- Converts `sockfd` from an active socket to a **listening socket** that can accept connection requests from clients.

- `backlog` is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests.

Sockets Interface: `accept`

- Servers wait for connection requests from clients by calling `accept`:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- Waits for connection request to arrive on the connection bound to `listenfd`, then fills in client's socket address in `addr` and size of the socket address in `addrlen`.

- Returns a **connected file descriptor** that can be used to communicate with the client via Unix I/O routines (`fwrite`, etc.).

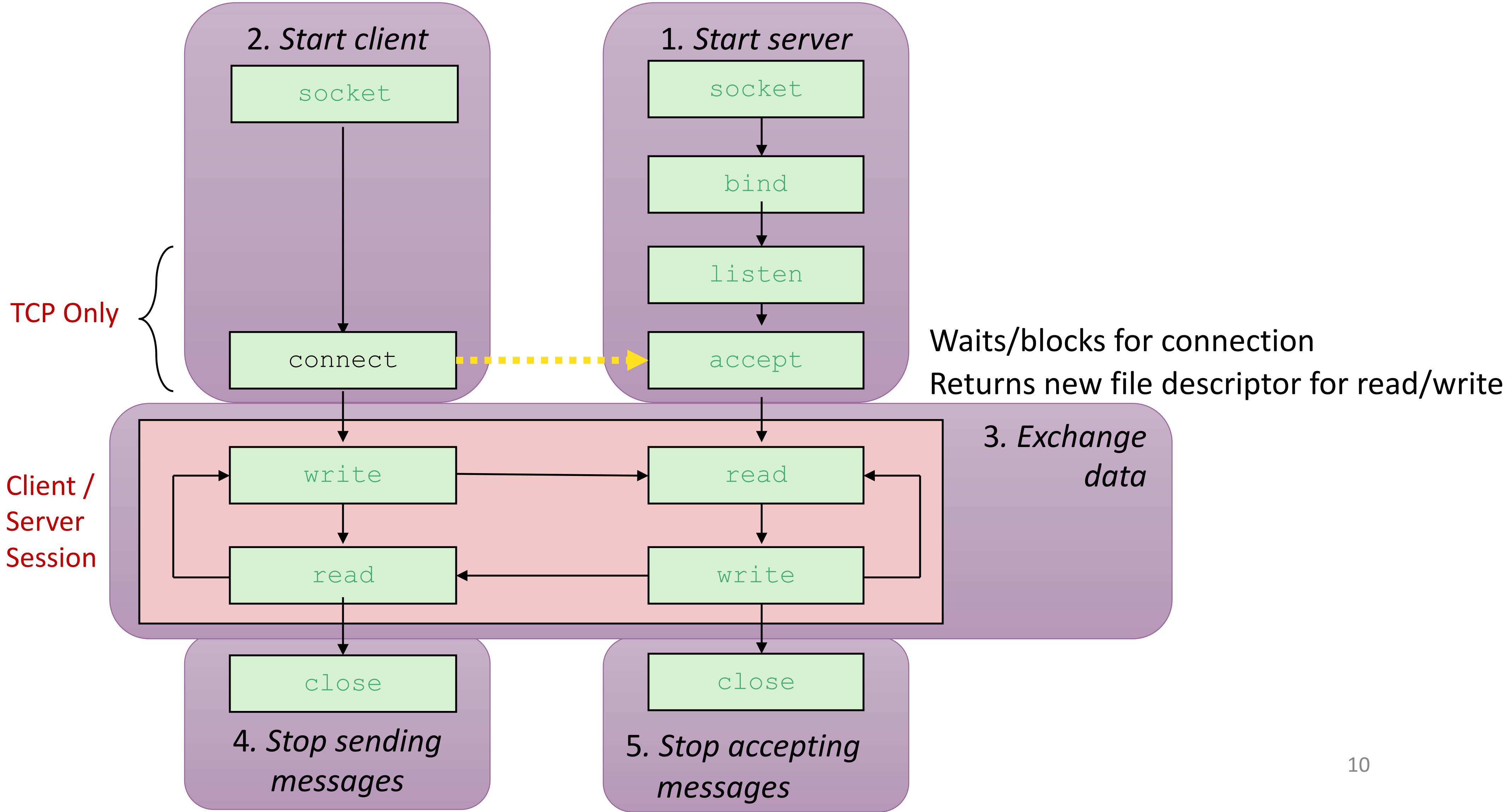
Connected vs. Listening Descriptors

- Listening descriptor
  - End point for client connection requests
  - Created once and exists for lifetime of the server

- Connected descriptor
  - End point of the connection between client and server
  - A new descriptor is created each time the server accepts a connection request from a client
  - Exists only as long as it takes to service client

- Why the distinction?
  - Allows for concurrent servers that can communicate over many client connections simultaneously
    - E.g., Each time we receive a new request, we `fork` a child to handle the request

Sockets Interface



Sockets Interface: `connect`

- A client establishes a connection with a server by calling connect:

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

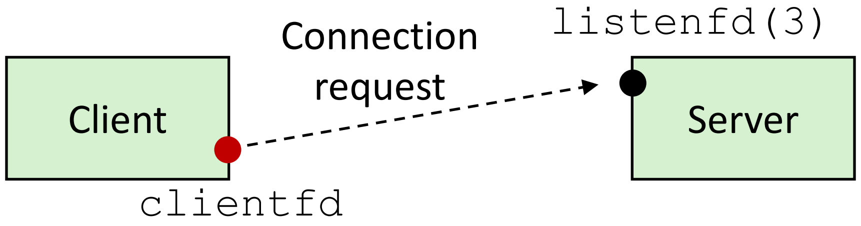
- Attempts to establish a connection with server at socket address `addr`
  - If successful, then `clientfd` (returned value) is now ready for reading and writing.
  - Resulting connection is characterized by socket pair

    `(x:y, addr.sin_addr:addr.sin_port)`
    - `x` is client address
    - `y` is ephemeral port that uniquely identifies client process on client host

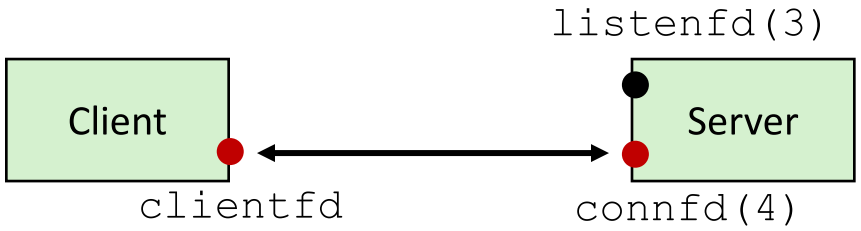
Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.

`accept` Illustrated

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



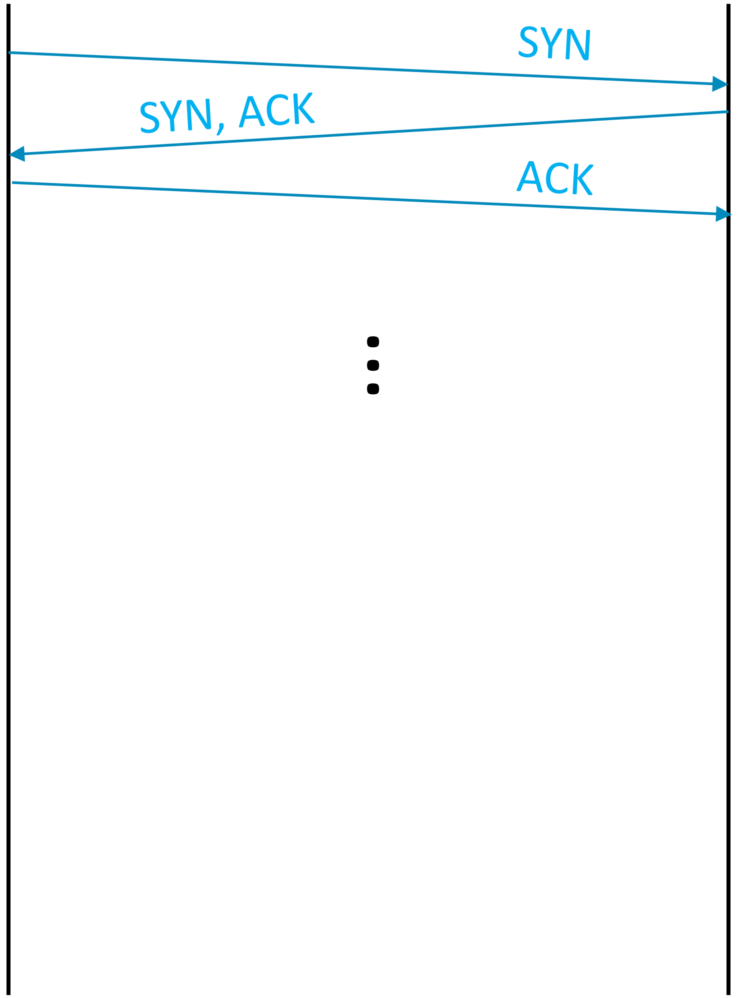
2. Client makes connection request by calling and blocking in `connect`



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

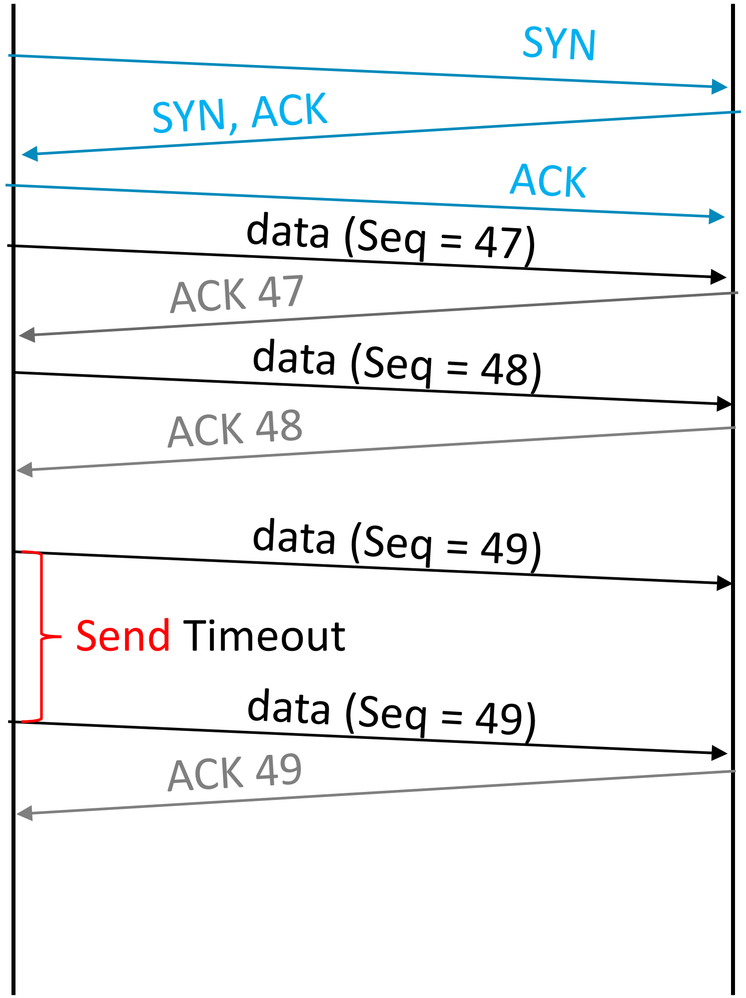
TCP Connections

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



Reliable Transport

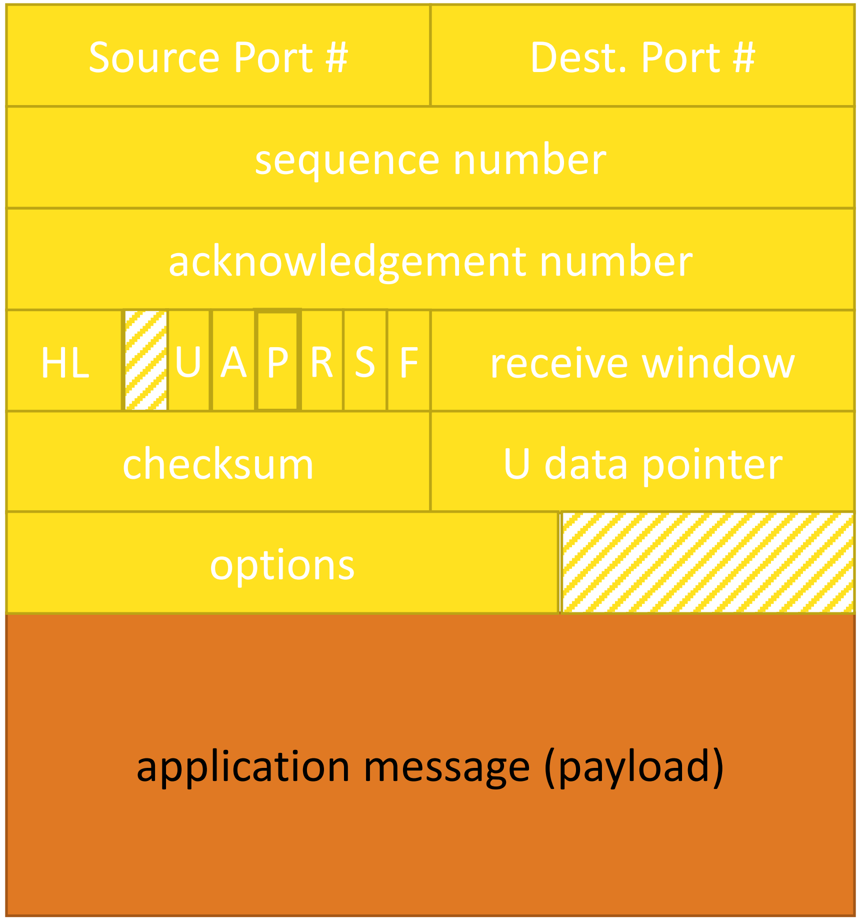
- Each SYN segment will include a randomly chosen sequence number
- Sequence number of each segment is incremented by data length
- Receiver sends ACK segments acknowledging latest sequence number received
- Sender maintains copy of all sent but unacknowledged segments; resends if ACK does not arrive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



Transport-Layer Segment Formats

UDP

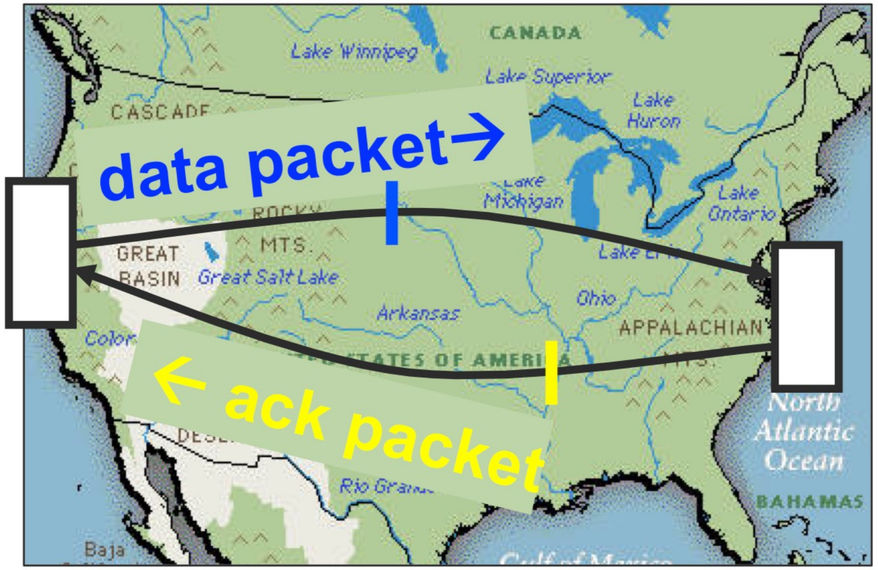
TCP

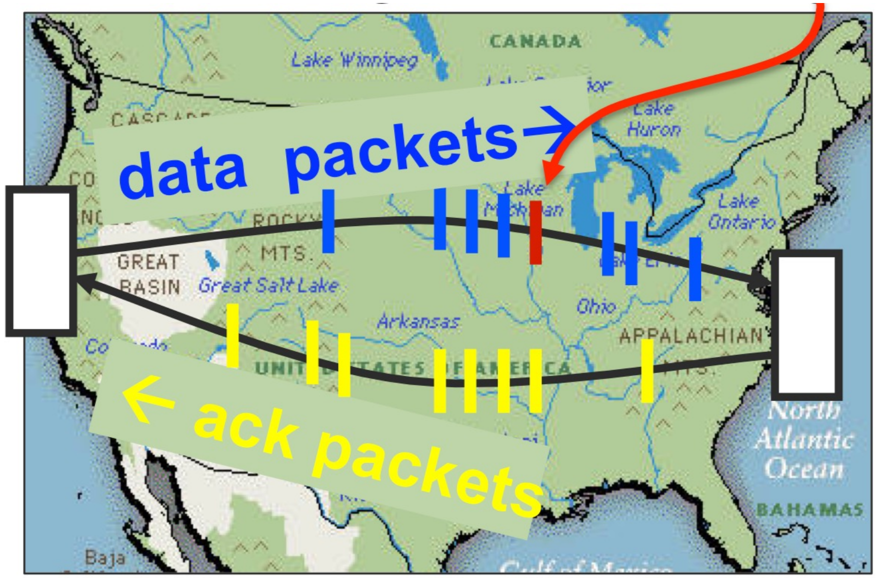


Pipelined Protocols

- Pipelining allows sender to send multiple "in-flight", yet-to-be-acknowledged packets
  - Increases throughput
  - Needs buffering at sender and receiver
- How big should the window be?

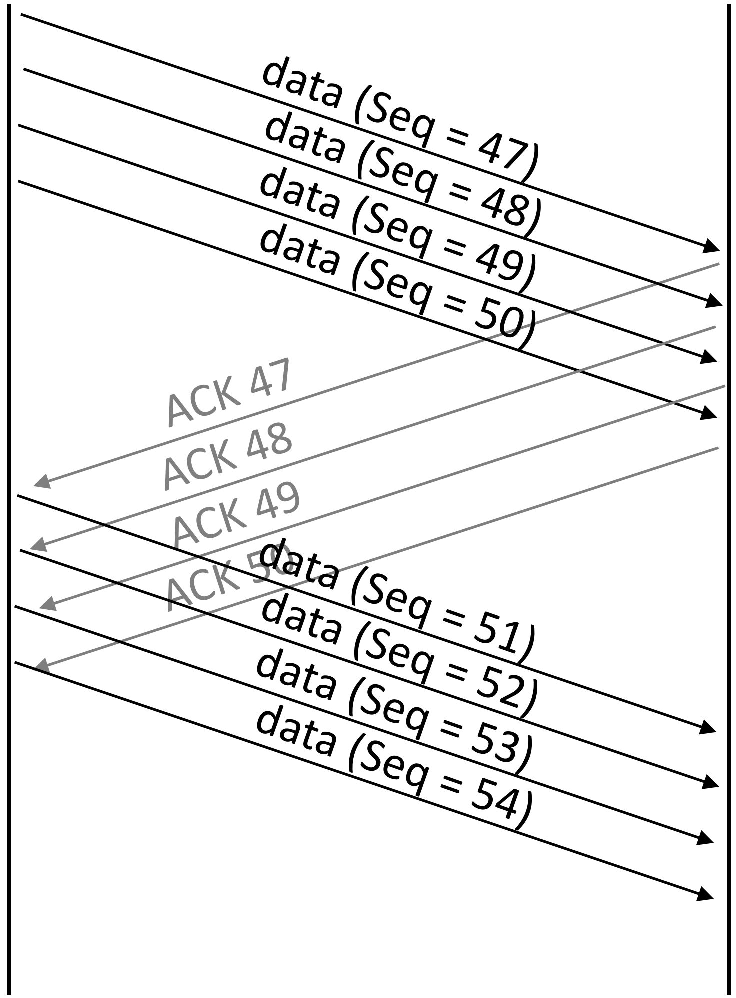
what if a packet in the middle goes missing?





Example

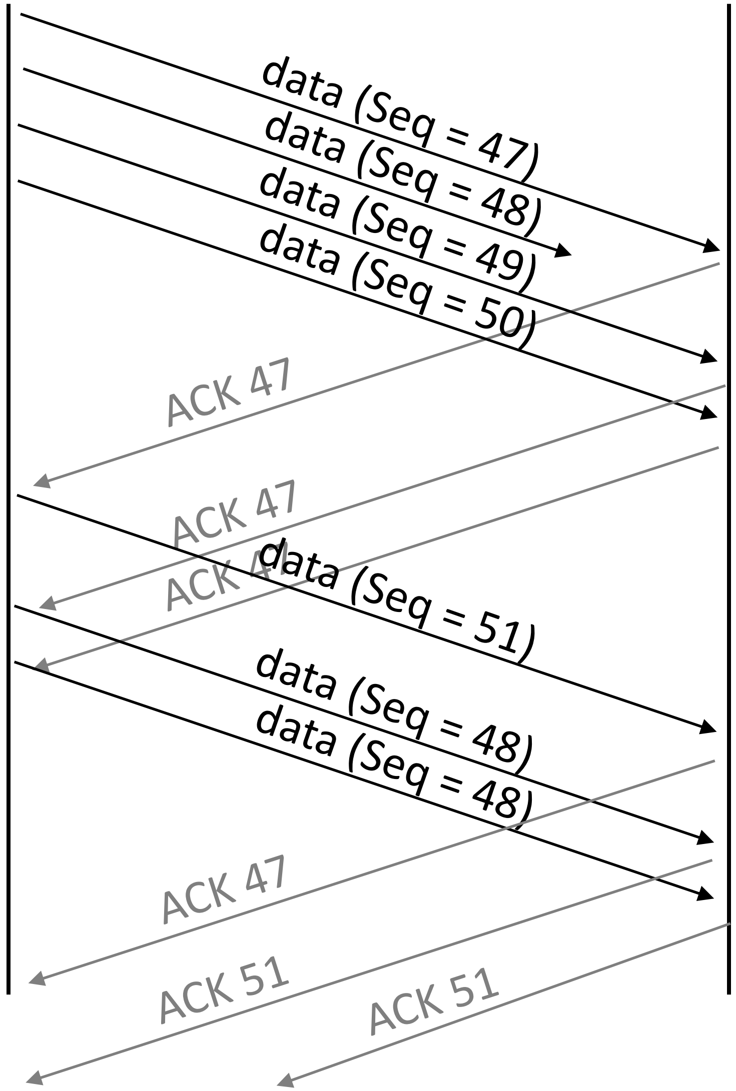
- Window Size = 4
- Sender can have up to 4 unacknowledged messages
- When ACK for first message is received, it can send another message



TCP Fast Retransmit

- Receiver always ACKs the last id it successfully received

- Sender detects loss without waiting for timeout, resends missing packet

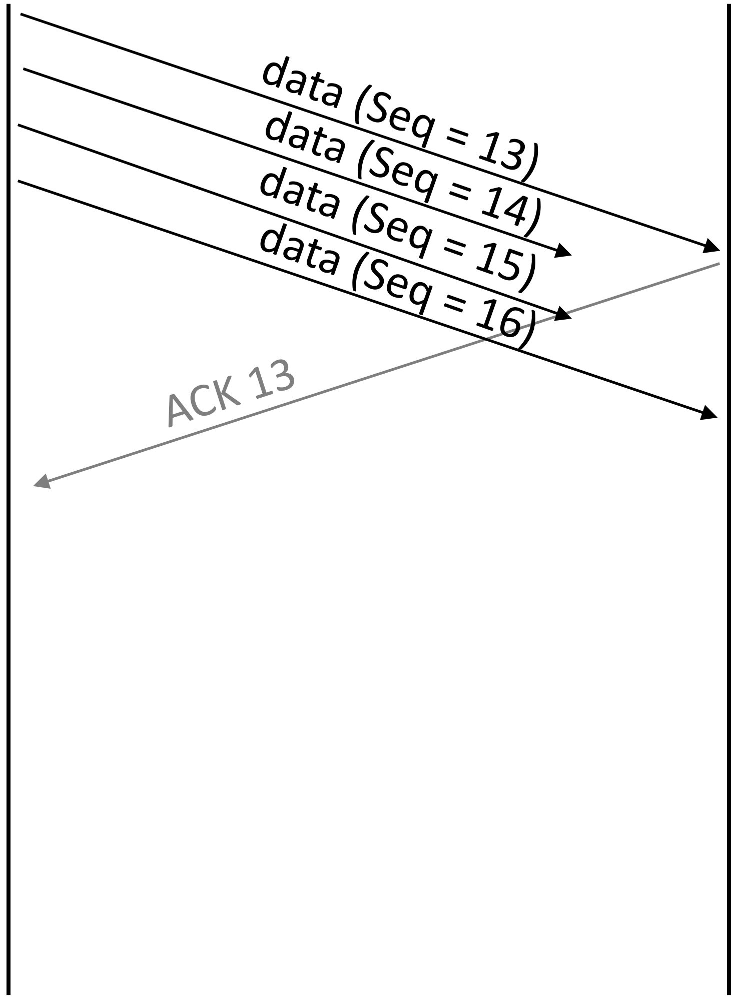


Practice with TCP Sequence Numbers

- Consider the sequence of transmitted messages shown on the right

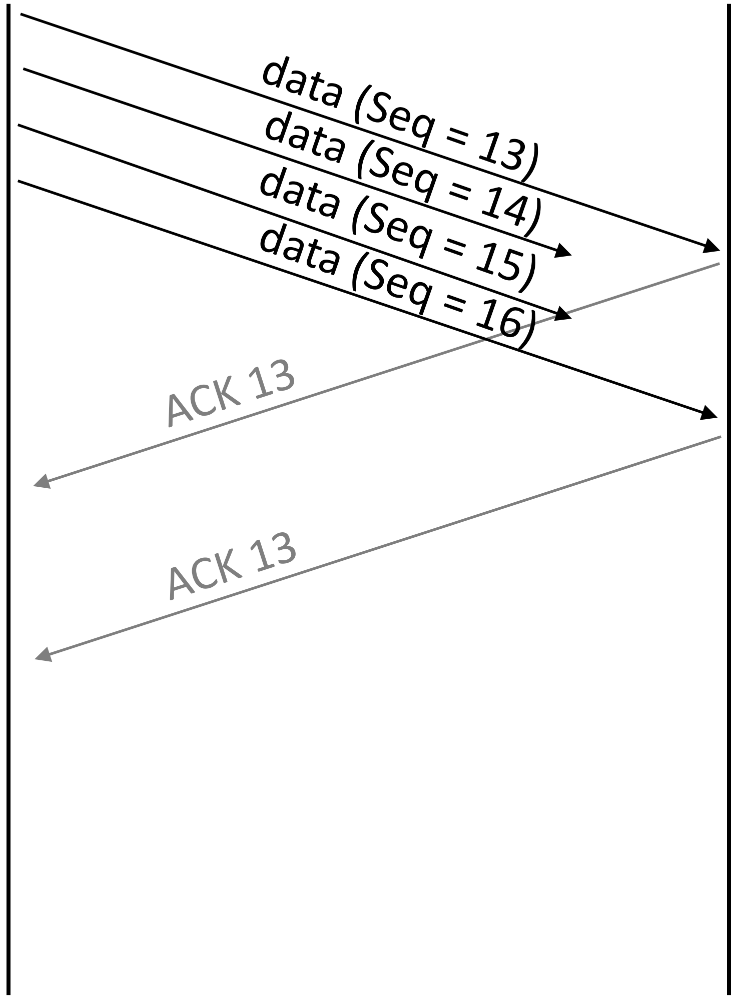
- What will be the next ACK number sent by the server?

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



Practice with 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?

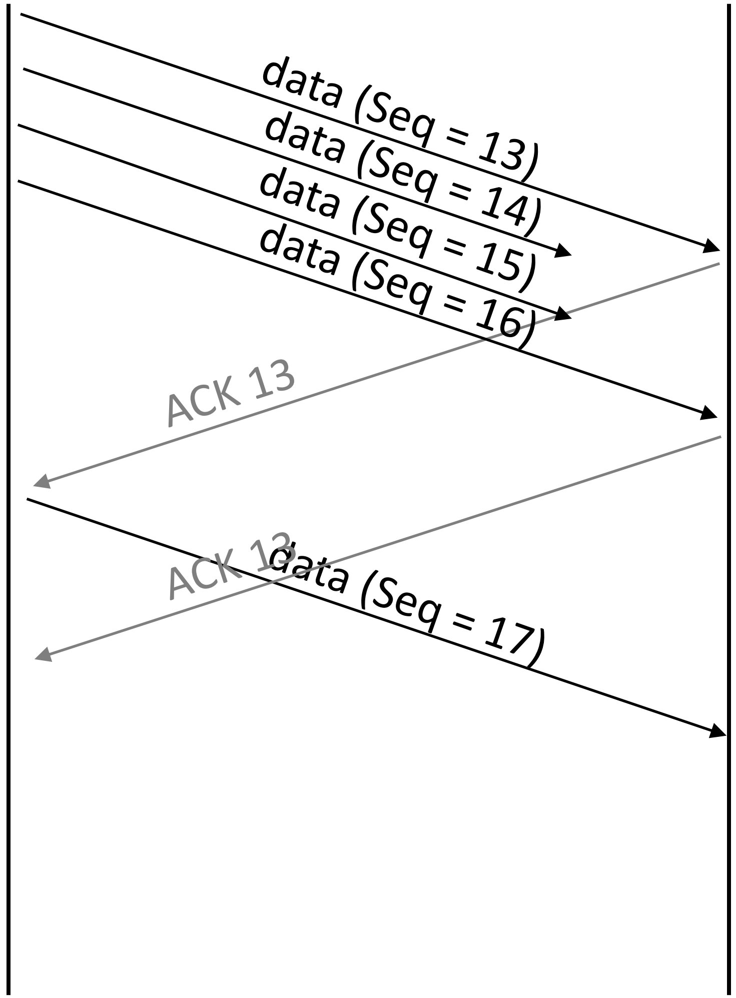


Practice with TCP Sequence Numbers

- Consider the sequence of transmitted messages shown on the right

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

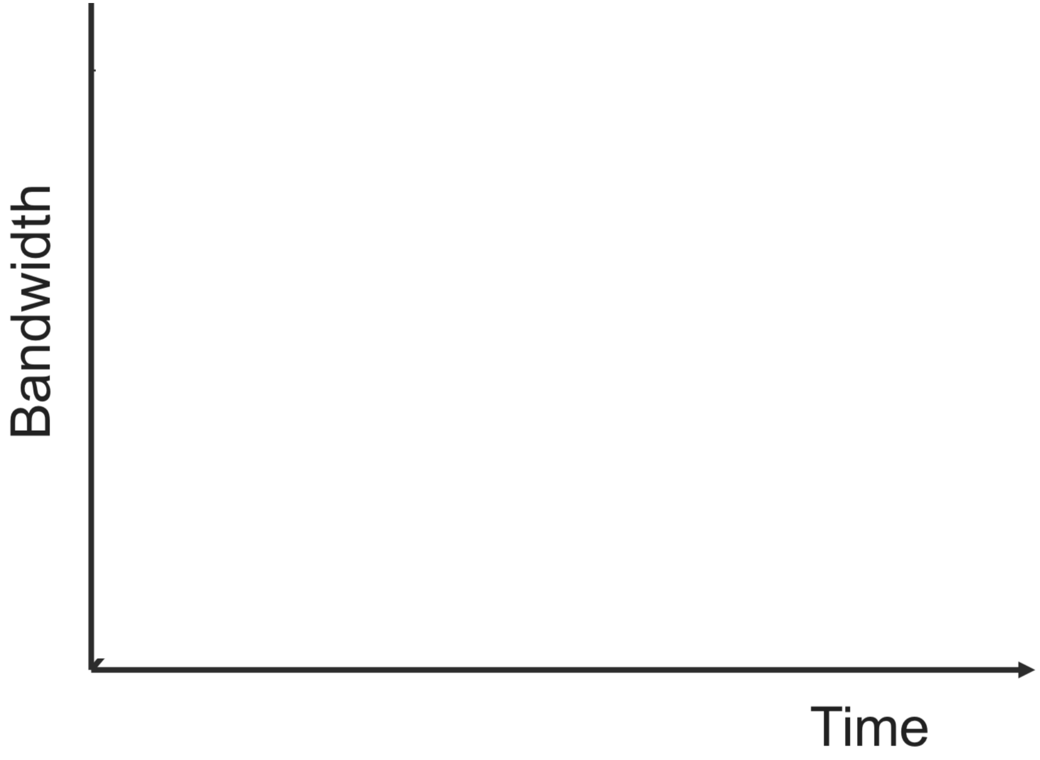
- What will be the next Seq number sent by the client?



TCP Congestion Control

TCP operates under a principle of

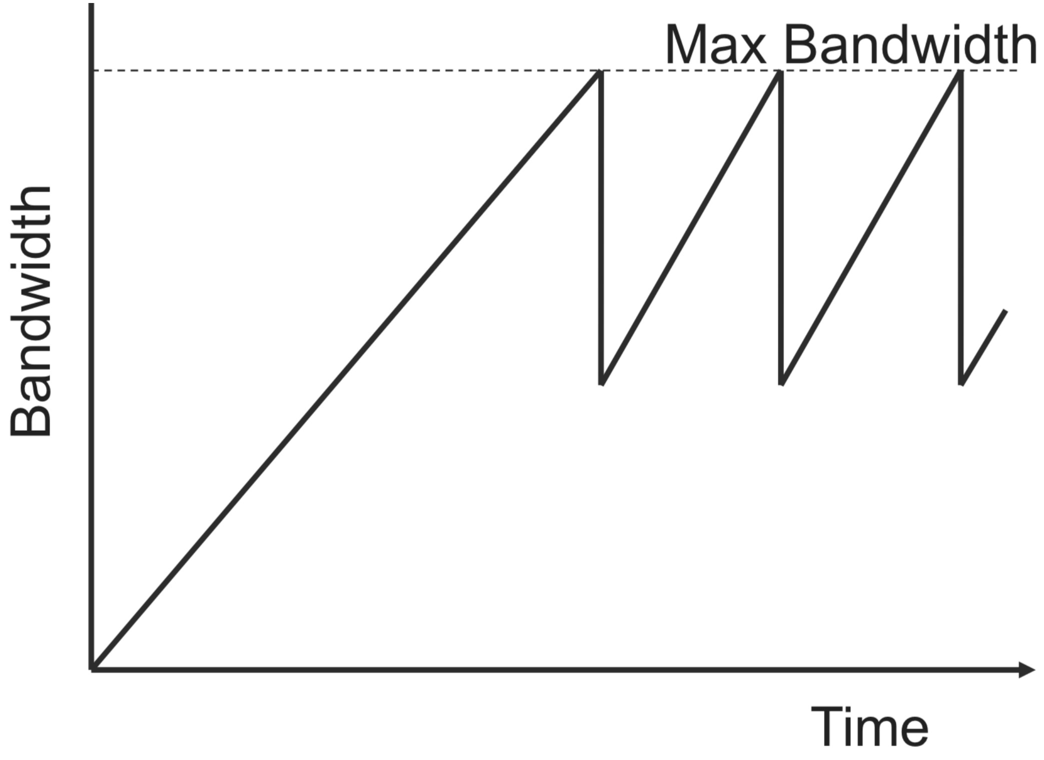
- additive increase
  - window size++ every RTT if no packets lost
- multiplicative decrease
  - window size/2 if a packet is dropped



TCP Congestion Control

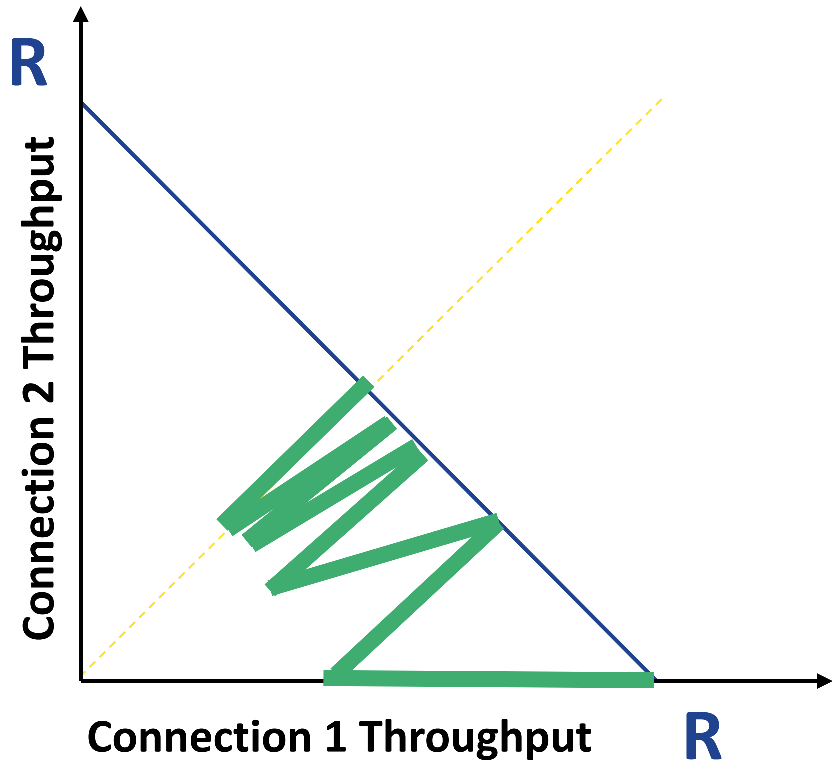
TCP operates under a principle of

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



TCP Fairness

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



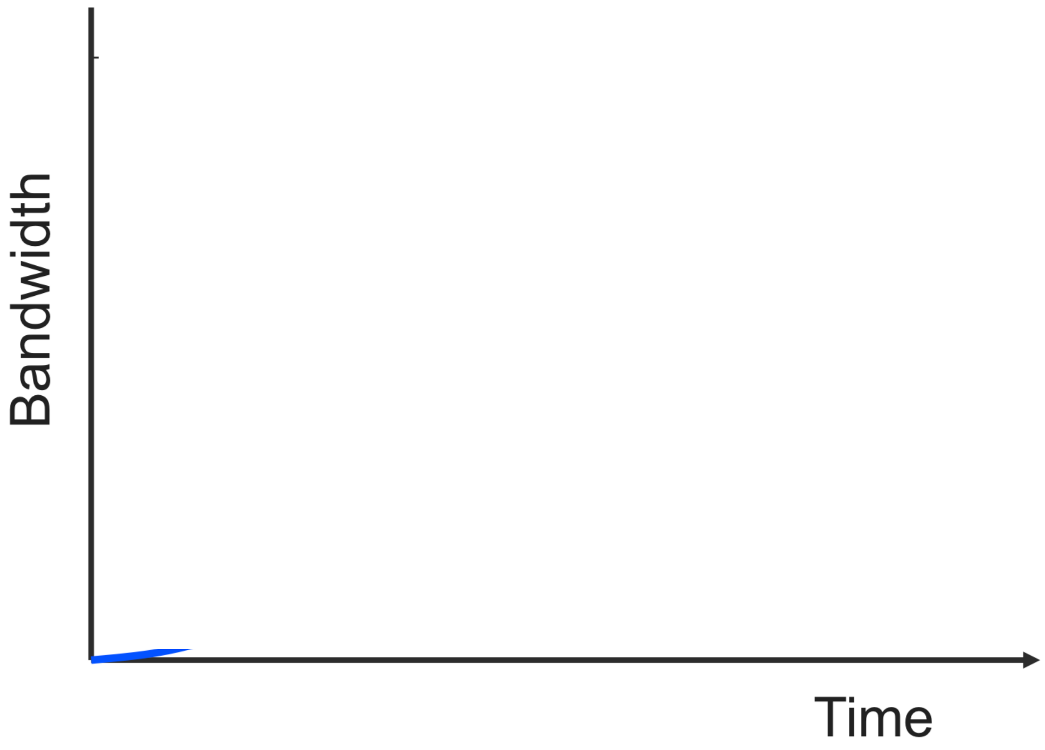
Loss: decreases throughput proportional to current bandwidth

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

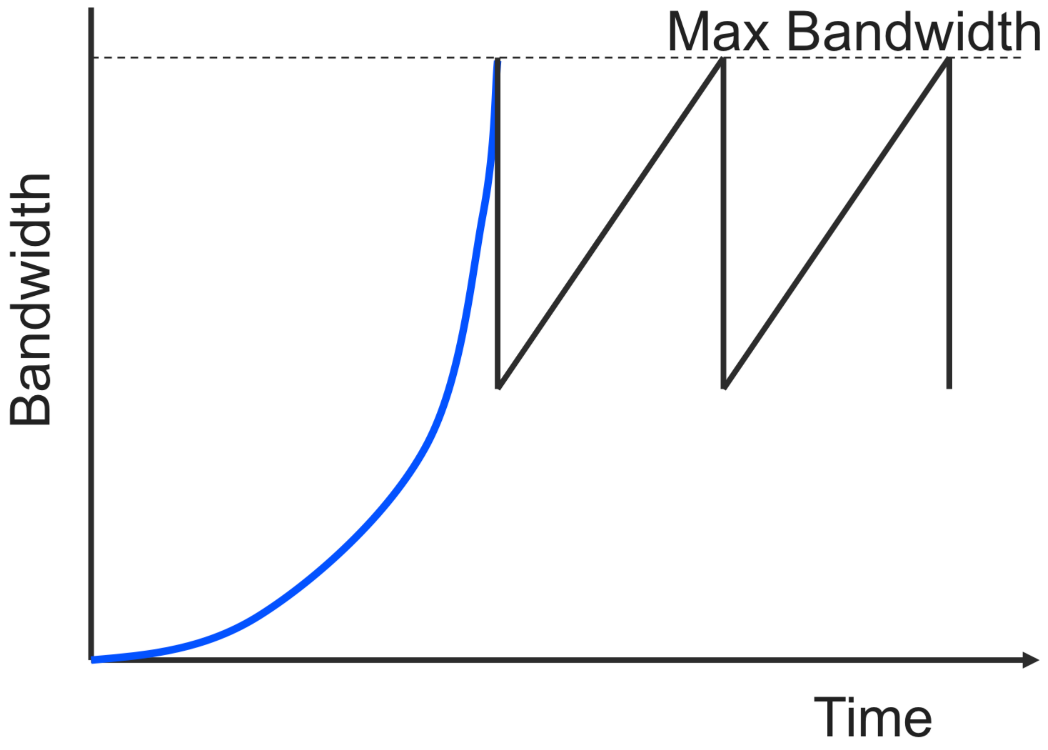
- Solution: allow window size to increase exponentially until first loss



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



Practice with TCP Window Size

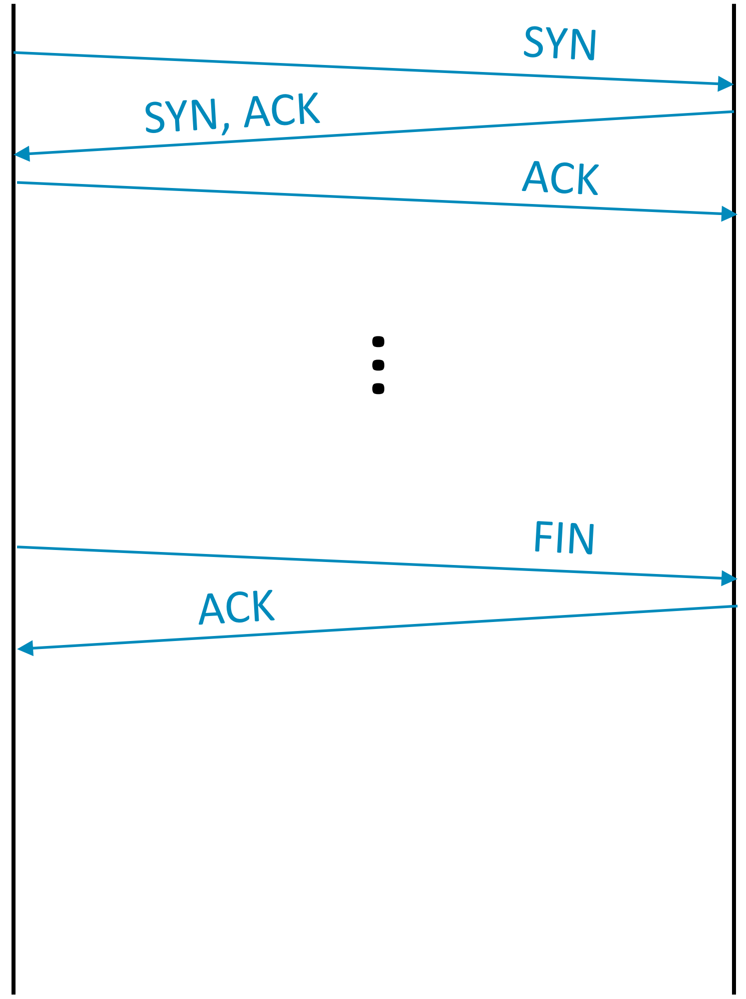
Assume someone changes the code of their TCP client by modifying the congestion avoidance as follows:

- 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?

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



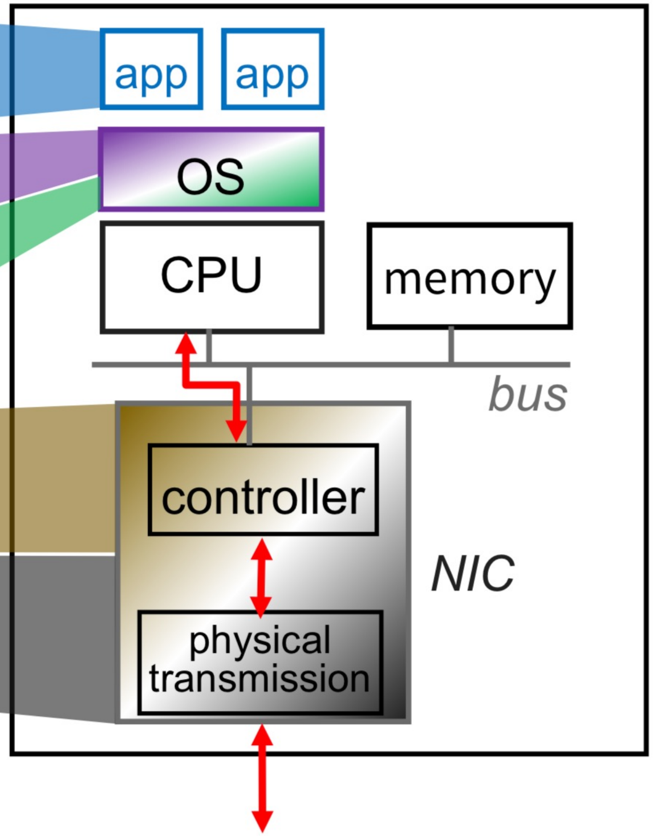
TCP Summary

- 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

Network Model

Web up to HTTP/2		
HTTP	**Application**	HTTP, FTP, DNS (*these*^ are usually in libraries)
TCP+TLS	**Transport**	TCP, UDP
IP	**Network**	IP, ICMP (ping)
Any	**Link**	Ethernet, WiFi
Any	**Physical**	wires, signal encoding

(Hard to draw firm lines here)

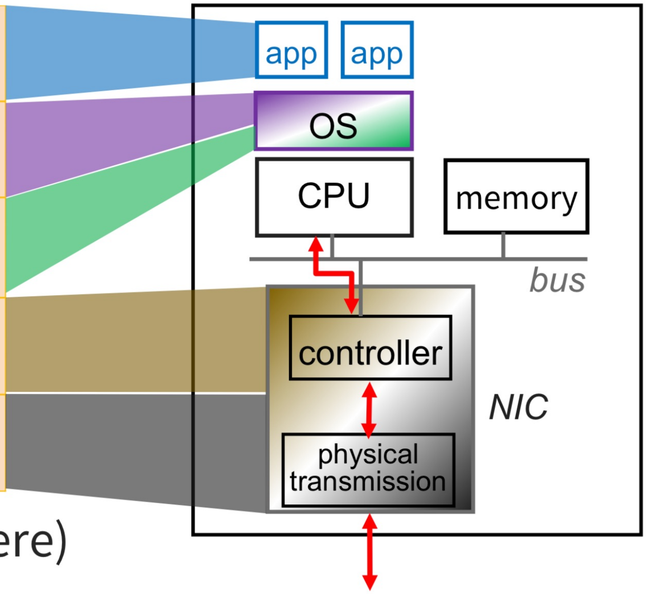


HTTP/3

- HTTP/1.1: TCP (+ optional TLS or SSL) transmitted on IPv4 or IPv6
- HTTP/2: TCP + TLS transmitted on IPv4 or IPv6
- HTTP/3: QUIC (built on UDP) transmitted on IPv4 or IPv6

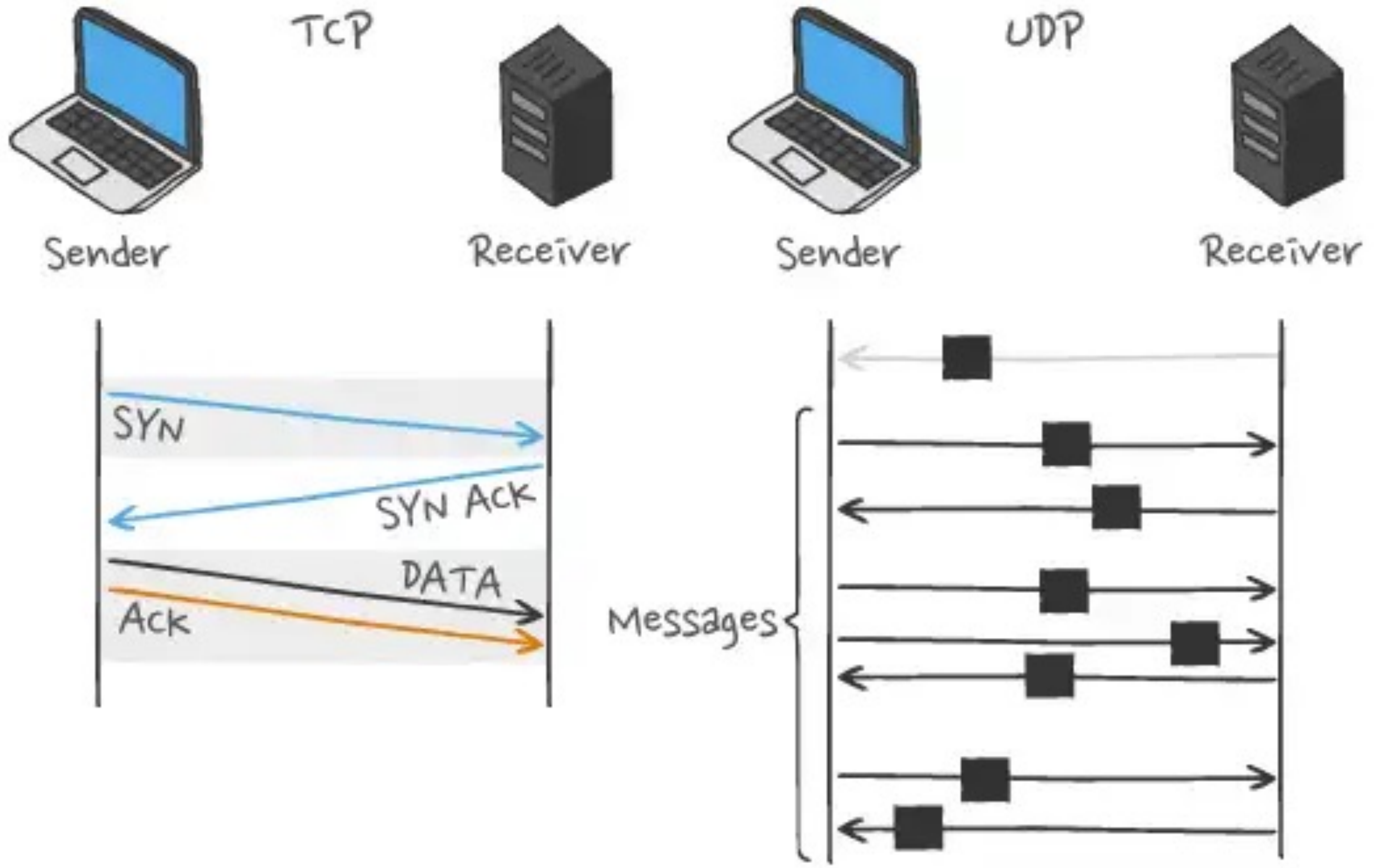
- HTTP/3 Availability
  - Chrome since April 2020
  - Edge since April 2020
  - Firefox since April 2021
  - Safari (not yet enabled by default)

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

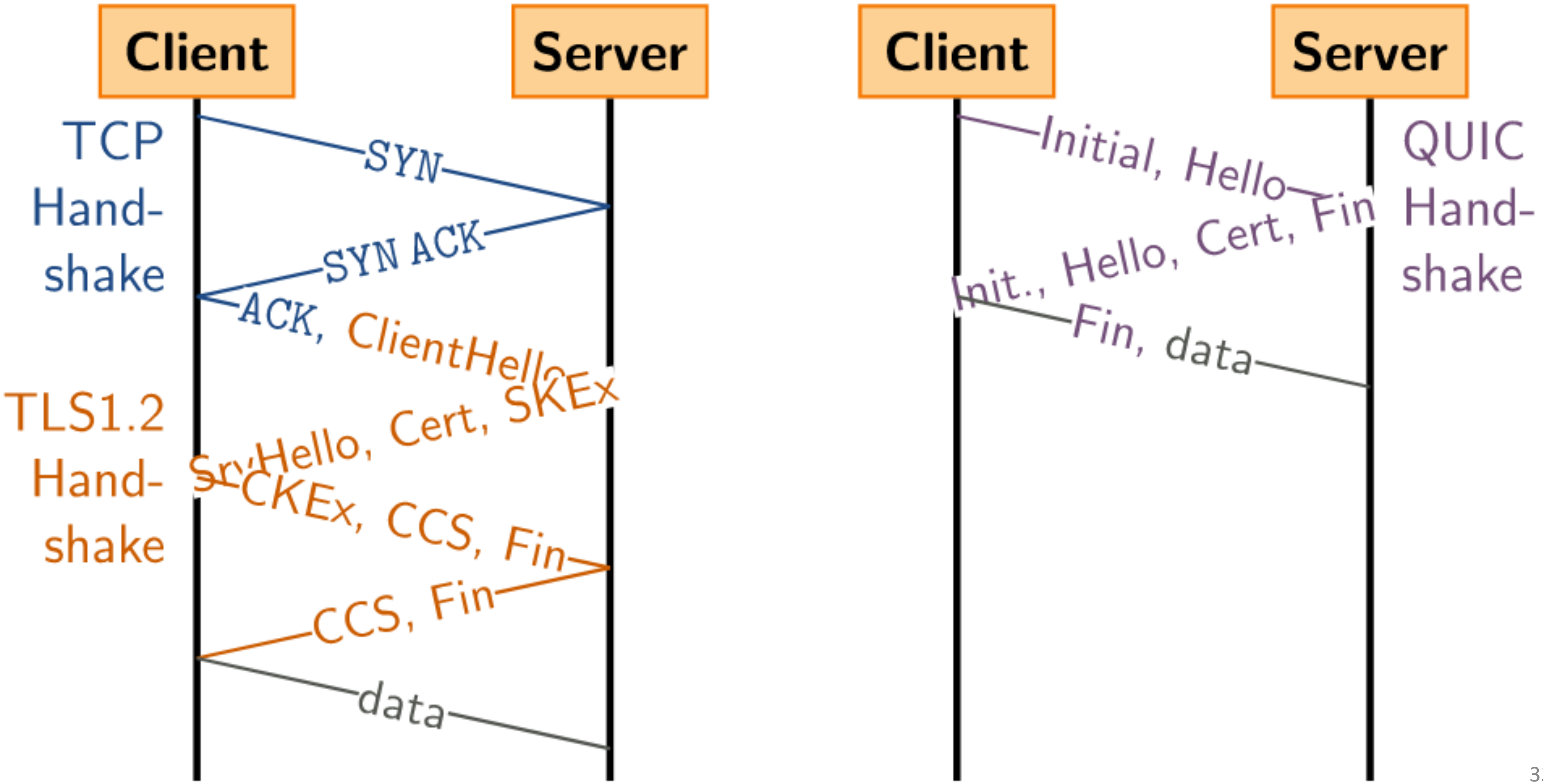


(Hard to draw firm lines here)

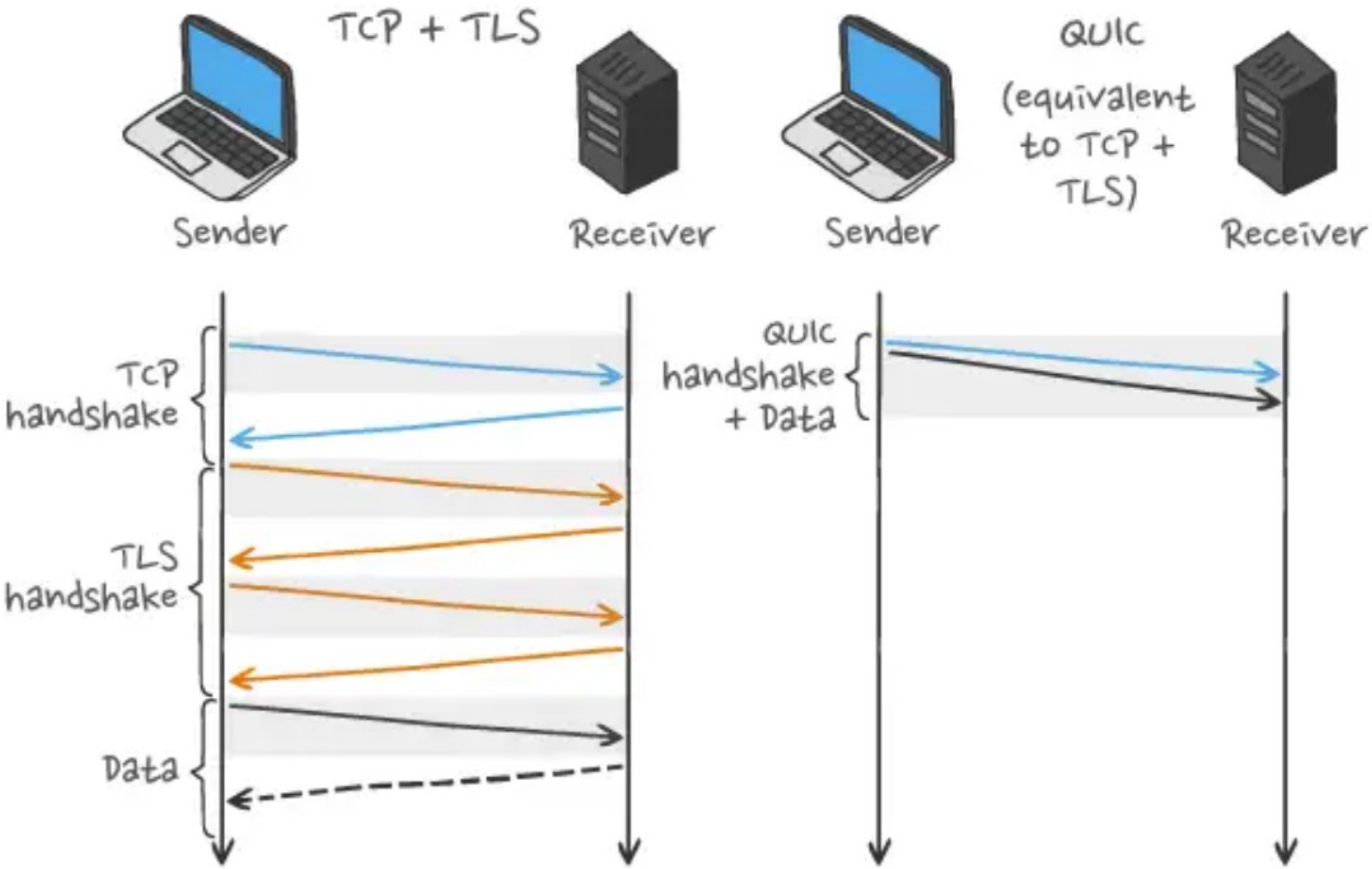
TCP vs UDP


TCP
Sender
Receiver
SYN
SYN Ack
DATA
Ack
UDP
Sender
Receiver
Messages

TCP vs QUIC


Client
Server
TCP Hand-shake
SYN
SYN ACK
ACK, ClientHello
TLS1.2 Hand-shake
SrvHello, Cert, SKEx
CKEx, CCS, Fin
CCS, Fin
data
Client
Server
Initial, Hello
QUIC Hand-shake
Init., Hello, Cert, Fin
Fin, data

TCP + TLS vs QUIC



HTTP/3 Layers