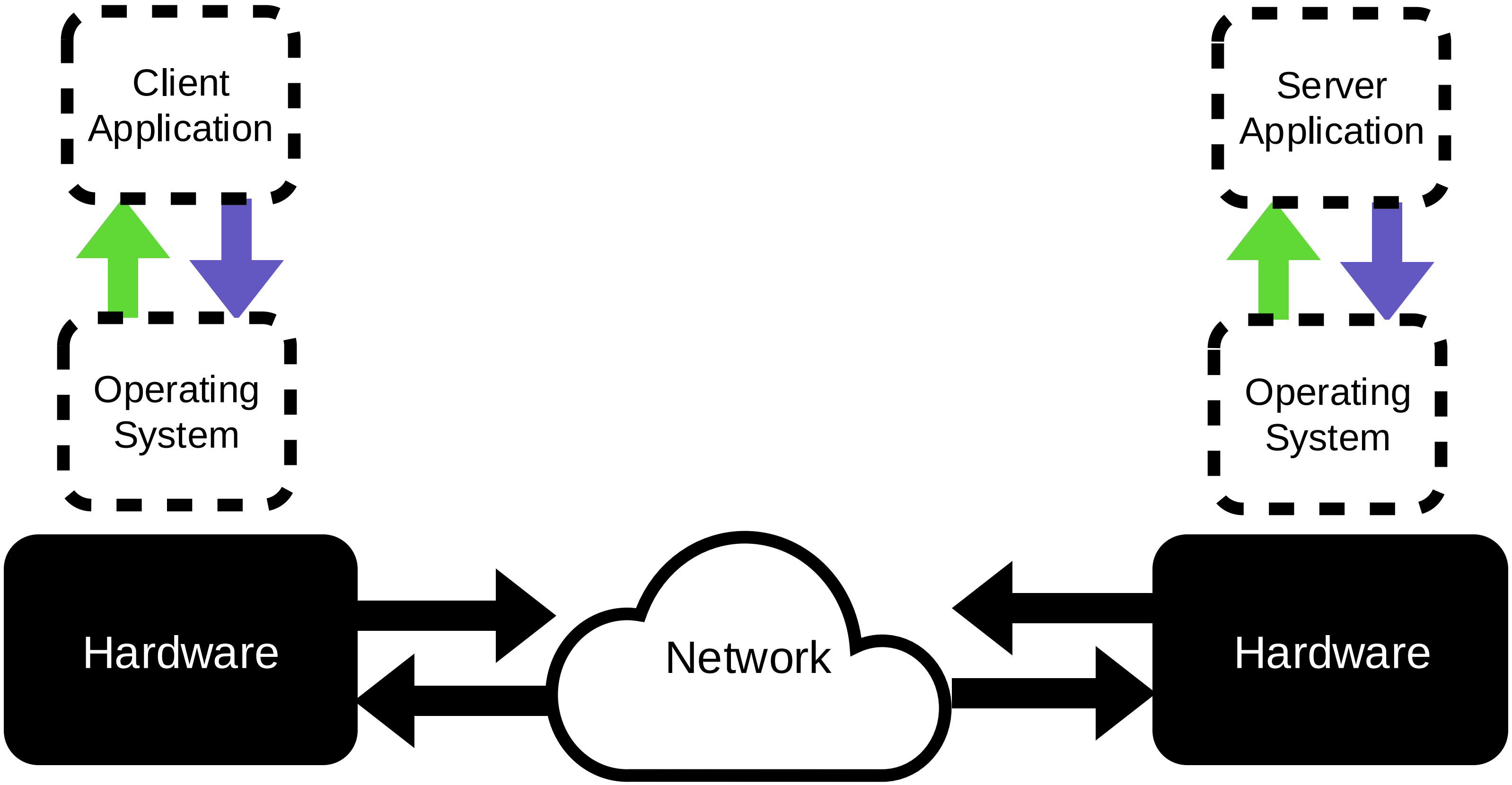
Lecture 24: Networking (cont'd)

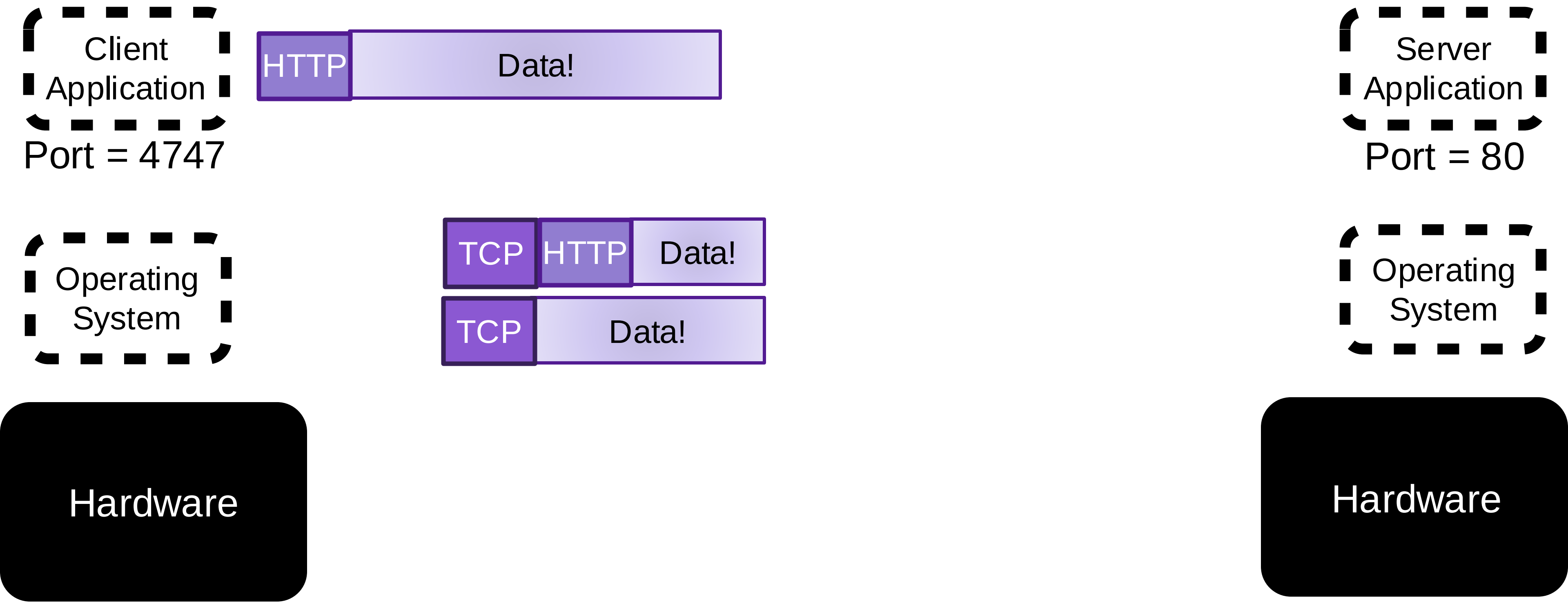
CS 105

Fall 2024

Review: Networked Systems



Review: Encapsulation



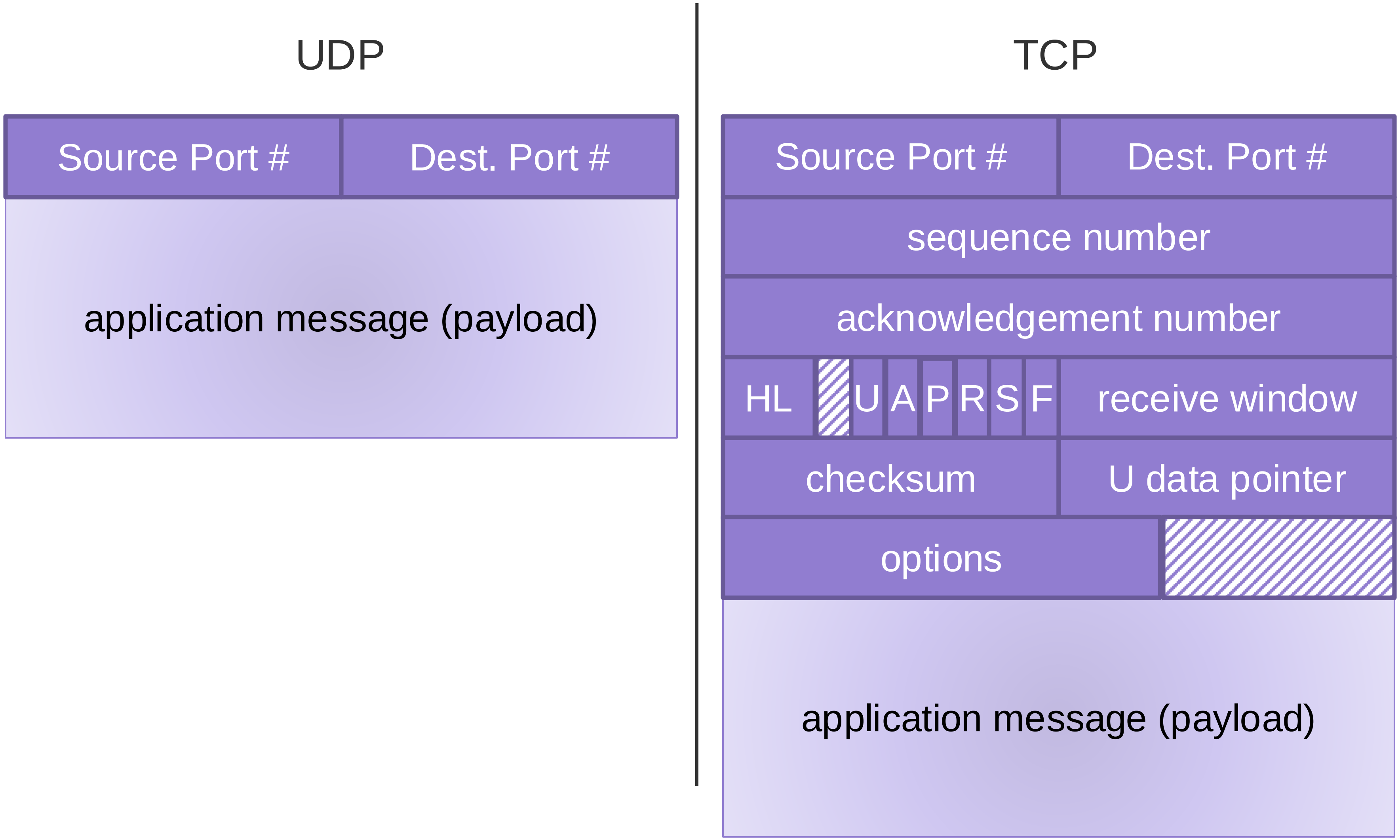
Ports

- A **port** is a 16-bit integer that identifies a process
  - Ephemeral port: Assigned automatically by client kernel when client makes a connection request.
  - Well-known port: Associated with some type of service on server

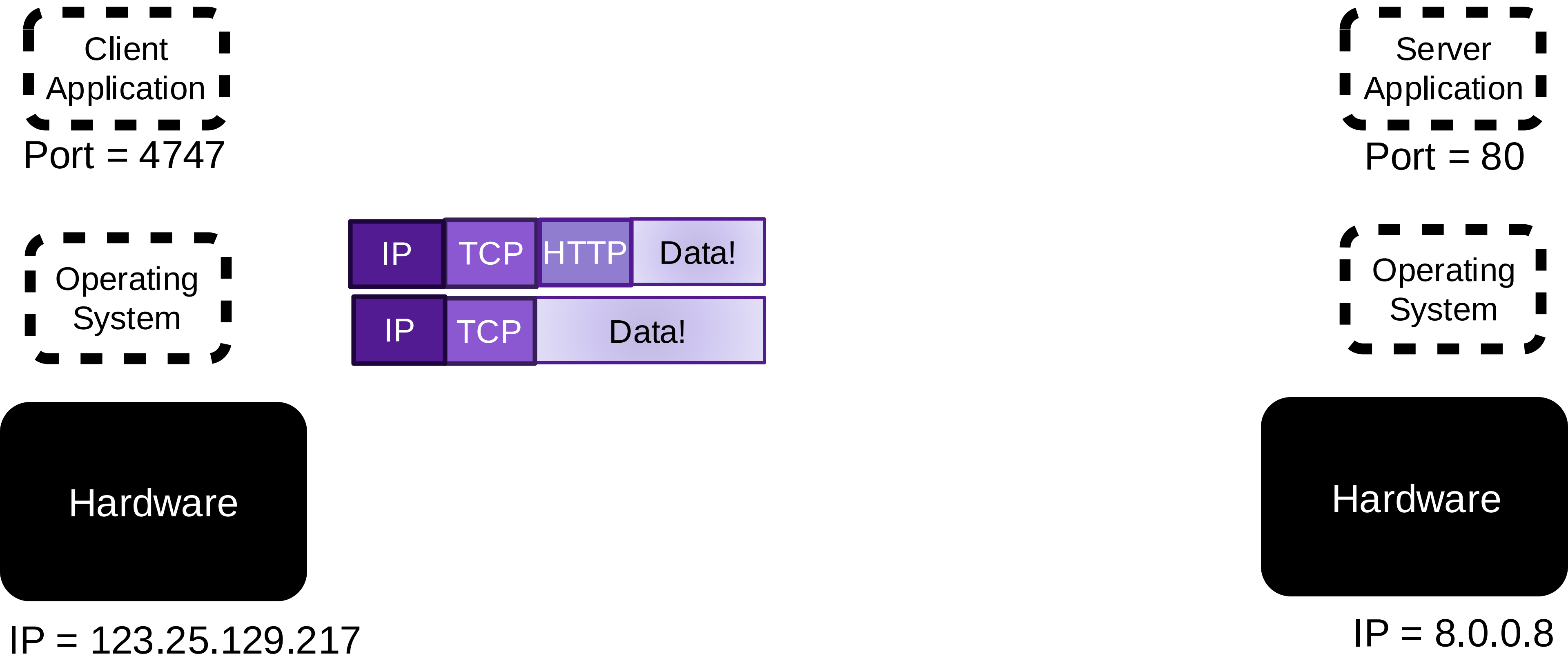
- Example **well-known ports** corresponding services:
  - echo server: 7/echo
  - ssh servers: 22/ssh
  - email server: 25/smtp
  - web servers: 80/http
  - secure web servers: /https

- If you are implementing a networked system, you implement both server code and client code (and hard-code the server port into the client code)

Transport-Layer Header Formats



Review: Encapsulation

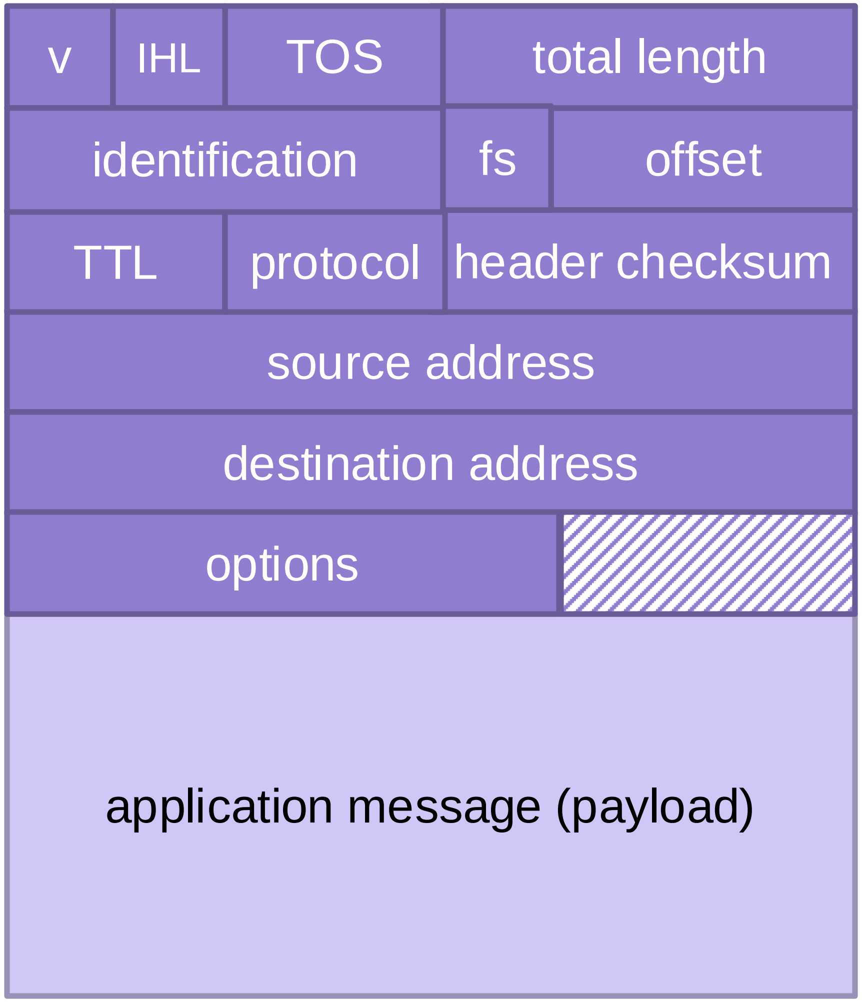


Internet Protocol (IP)

- Initiated by the DoD in 60s-70s
- Currently transitioning (very slowly) from IPv4 to IPv6

- Example address: 128.84.12.43

- interoperable
- network dynamically routes packets from source to destination



Aside: IPv4 and IPv6

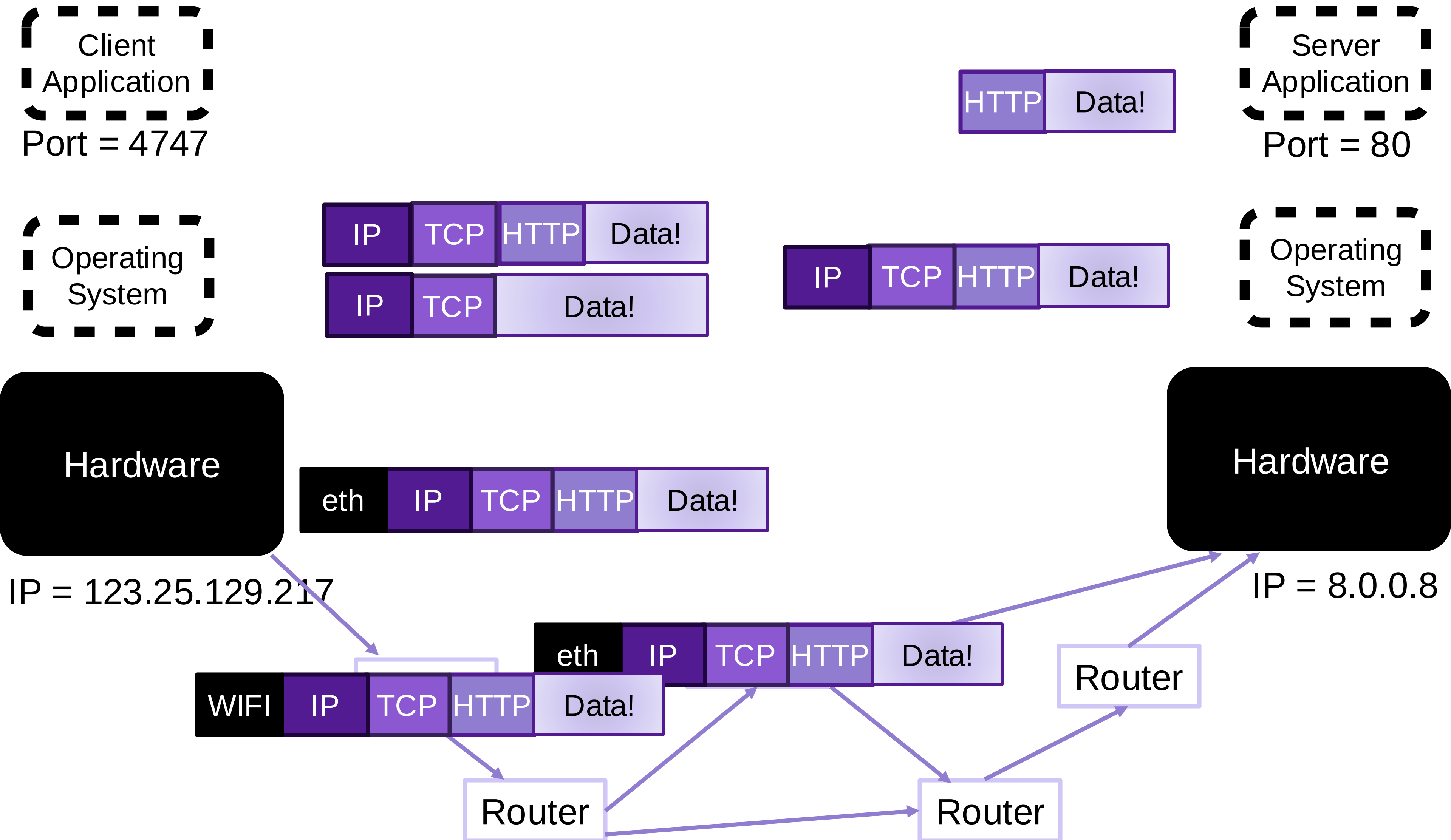
- The original Internet Protocol, with its 32-bit addresses, is known as *Internet Protocol Version 4* (IPv4)

- 1996: Internet Engineering Task Force (IETF) introduced *Internet Protocol Version 6* (IPv6) with 128-bit addresses
  - Intended as the successor to IPv4

- As of April 2023, majority of Internet traffic still carried by IPv4
  - 38-44% of users access Google services using IPv6.

- We will focus on IPv4, but will show you how to write networking code that is protocol-independent.

Review: Encapsulation

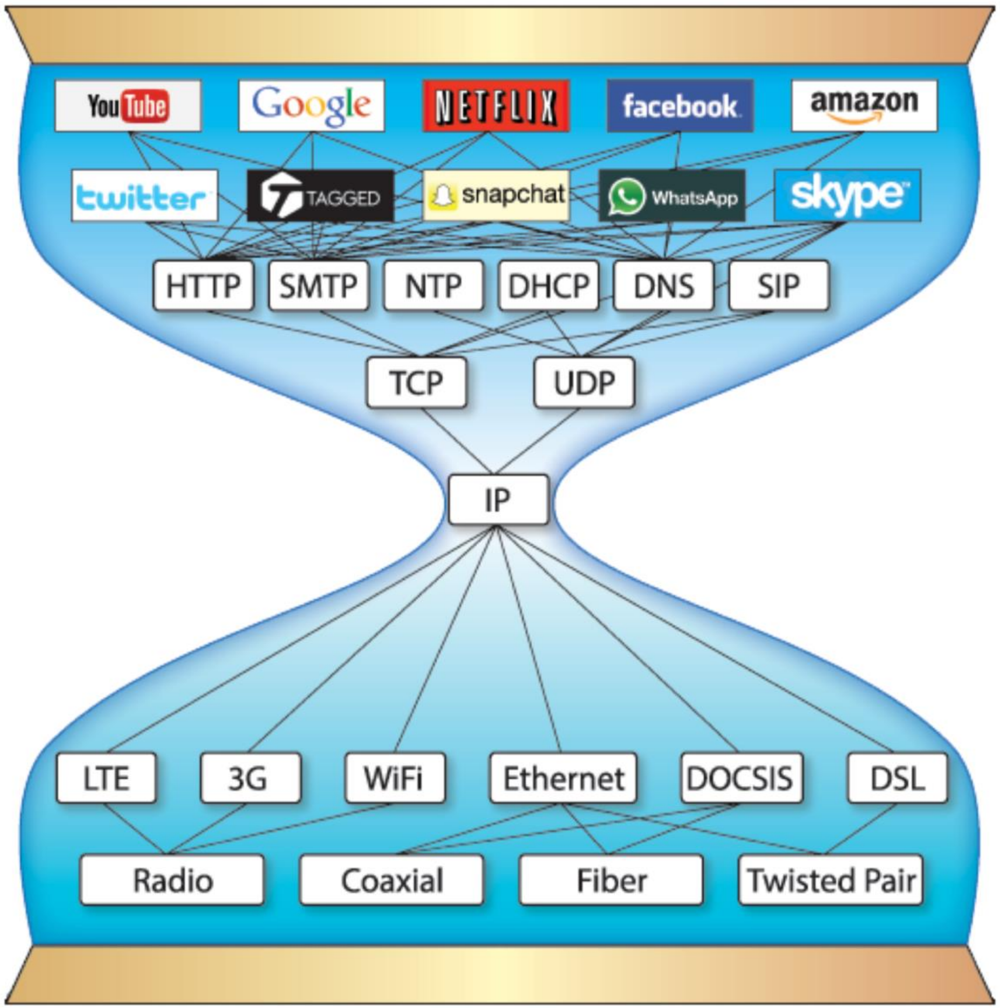


The Network Stack

User-Level Application

Operating System

Hardware (NIC)



Application

Transport

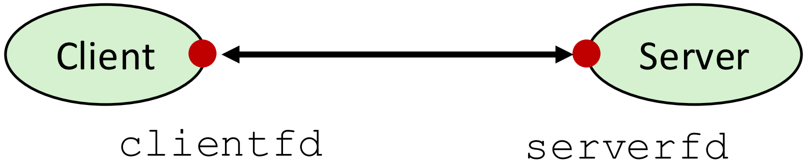
Network

Data Link

Physical

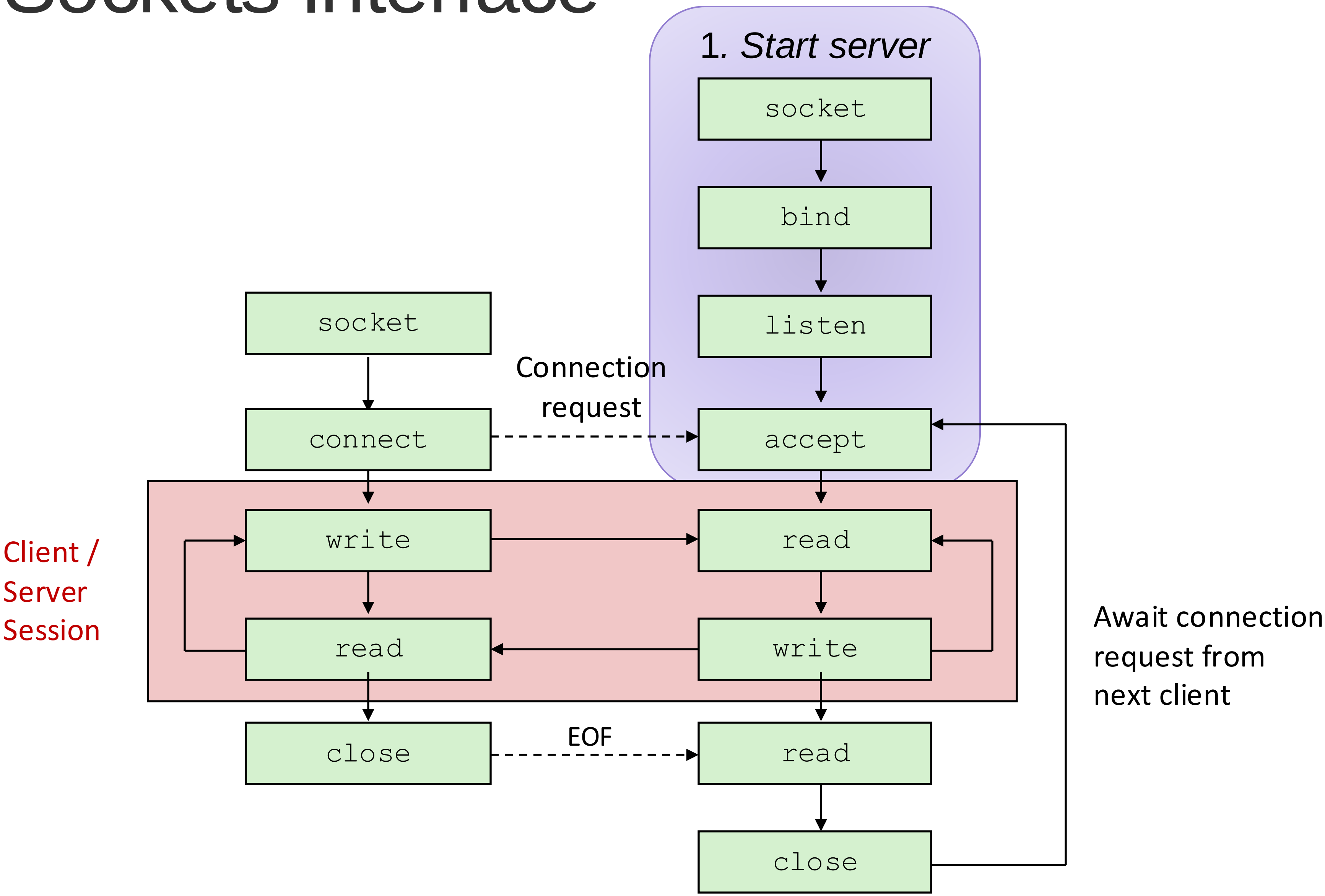
Sockets

- What is a socket?
  - IP address + port
  - To the operating system, a socket is an endpoint of communication
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network
    - **Recall:** All Unix I/O devices, including networks, are modeled as files
- Clients and servers communicate with each other by reading from and writing to socket descriptors



- The main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors

Sockets Interface



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_STREAM, 0);
```

Indicates that we are using 32-bit IPV4 addresses (AF_INET)

Indicates that the socket will be the end point of a TCP connection (SOCK_STREAM)

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

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, SA* addr, socklen_t addrlen);
```

- Clients don't have to do this

- The process can then 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`.

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

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 listenfd, SA *addr, int *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 descriptor** that can be used to communicate with the client via Unix I/O routines.
- Process can read and write to this connected descriptor to get/send messages over the network

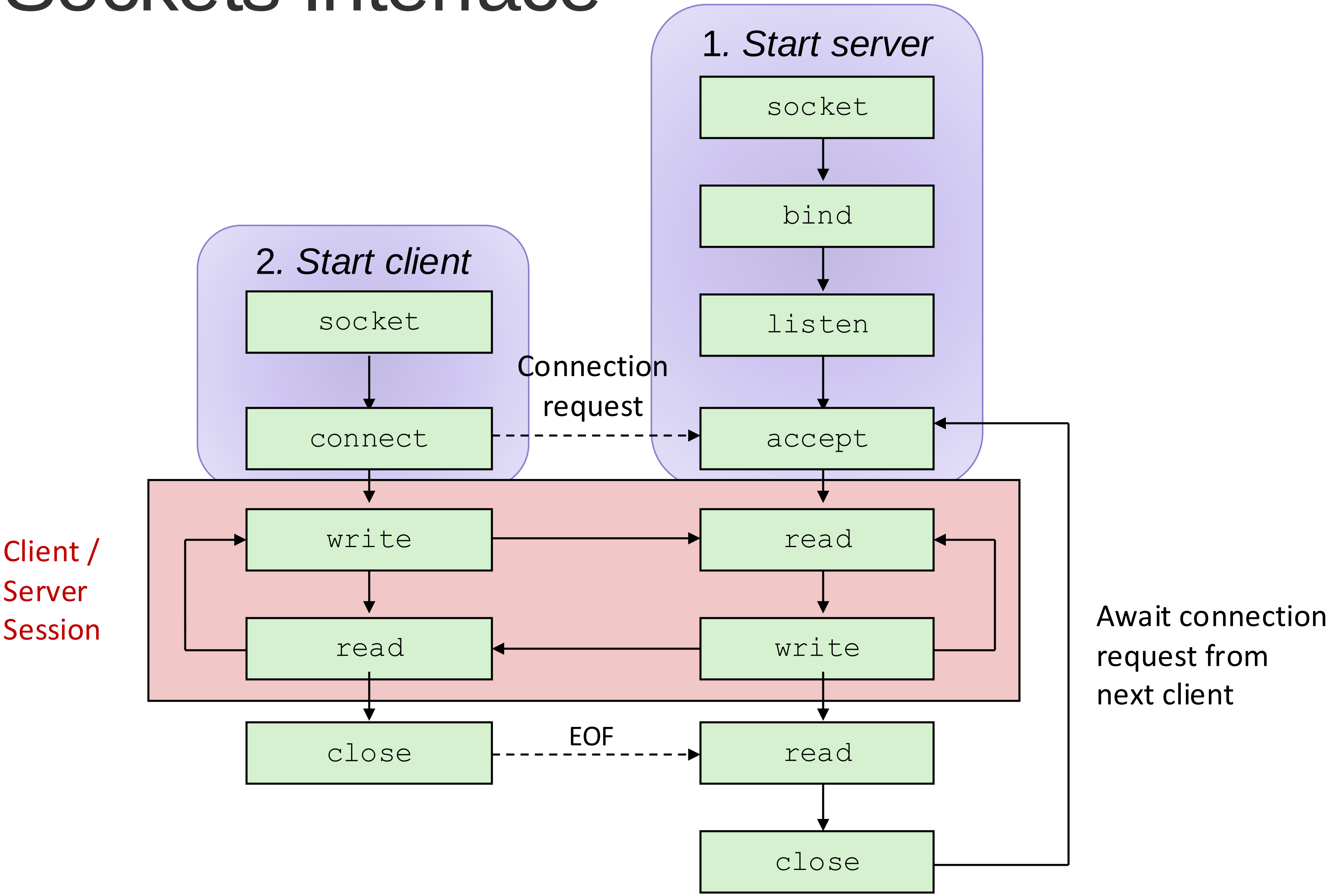
Connected vs. Listening Descriptors

- 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:

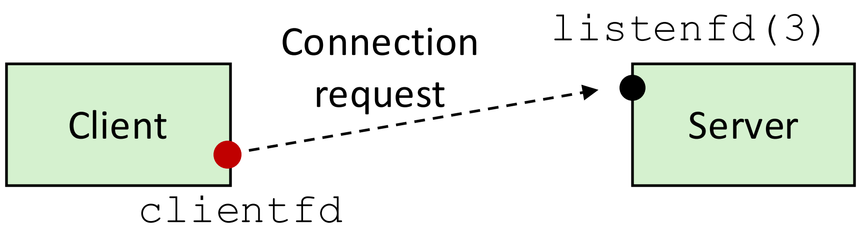
```
int connect(int clientfd, SA* addr, socklen_t addrlen);
```

- Attempts to establish a connection with server at socket address `addr`
  - If successful, then `clientfd` 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`

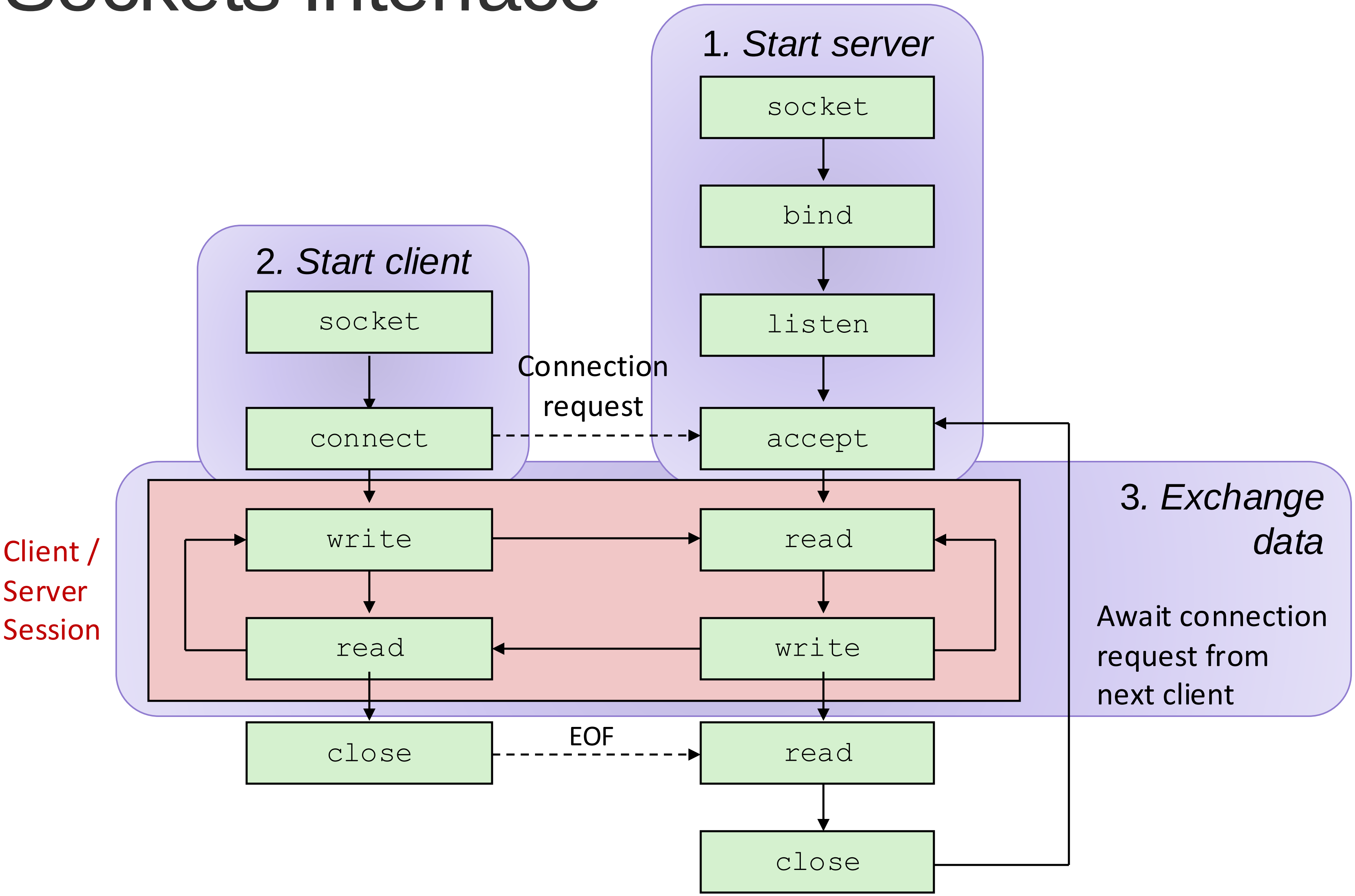
3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`

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

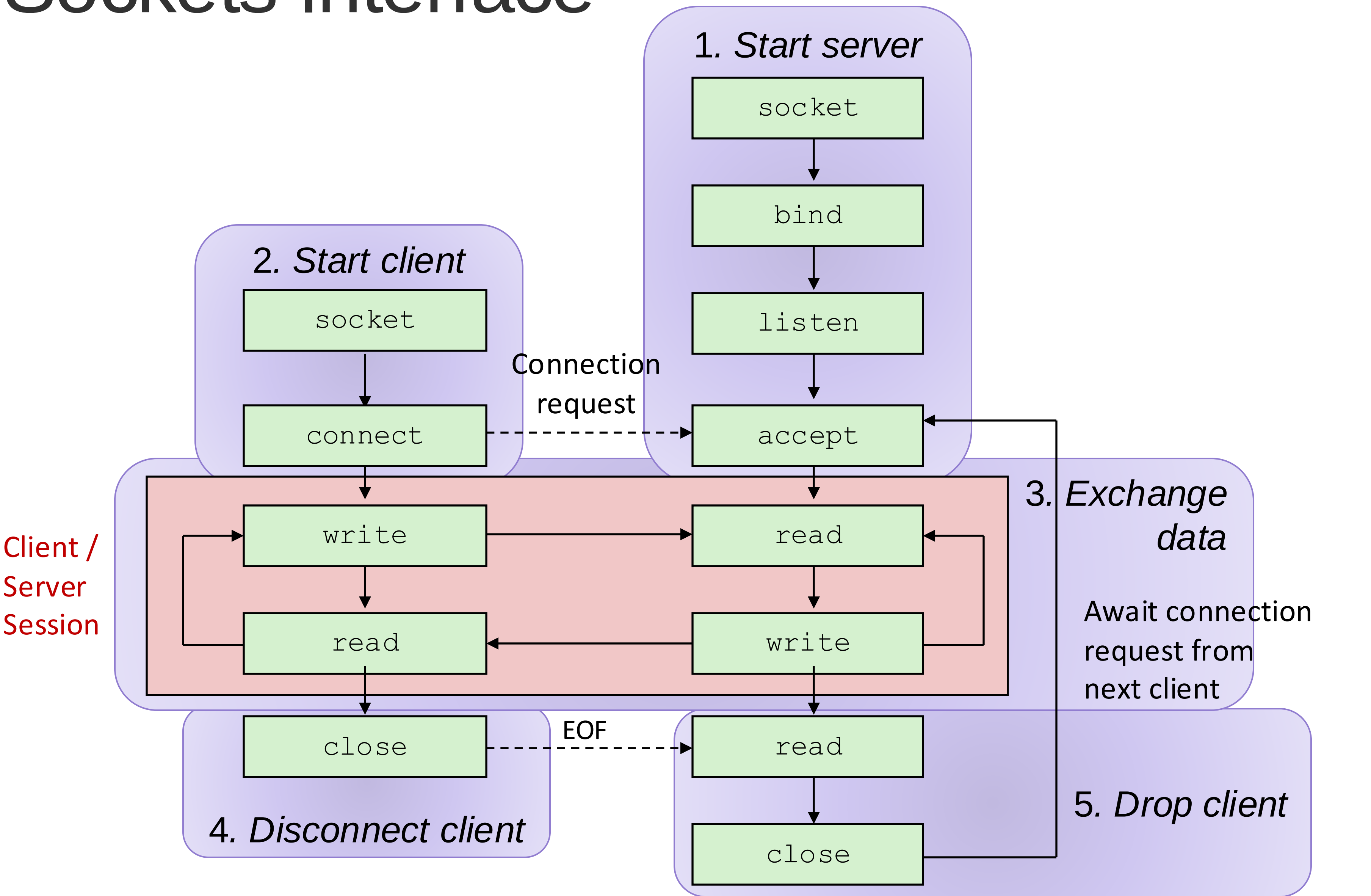


Communicating over a channel

- 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**

Sockets Interface

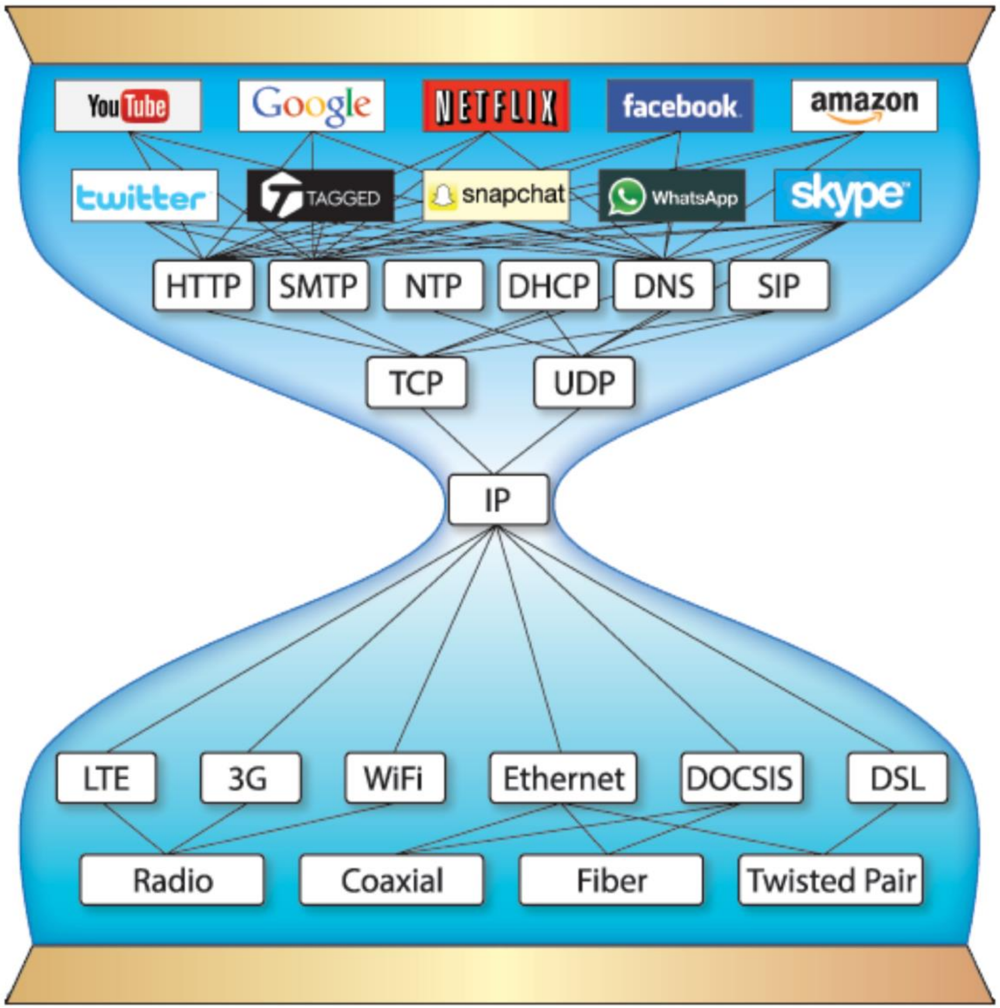


The Network Stack

User-Level Application

Operating System

Hardware (NIC)



Application

Transport

Network

Data Link

Physical