Networking
Sockets

Adapted by Joseph Lunderville
from slides by Dr. Brian Fraser
and course material by Dr. Steve Ko

Topics

- How does software do something complicated like networking?
  - Layers!
- What are the two types of sockets?
- What syscalls can we use to work with sockets?

Networking

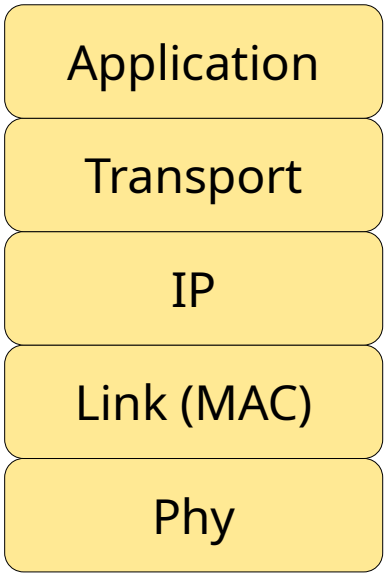
- *Programs can communicate with each other via a network*
  - Can be across a network (wifi, wired, ...)
  - Can be on the same computer!
- *More Resources*
  - Beej's Guide to Network Programming is popular:

    https://beej.us/guide/bgnet/
  - The Linux Programming Interface (our recommended text) is also great

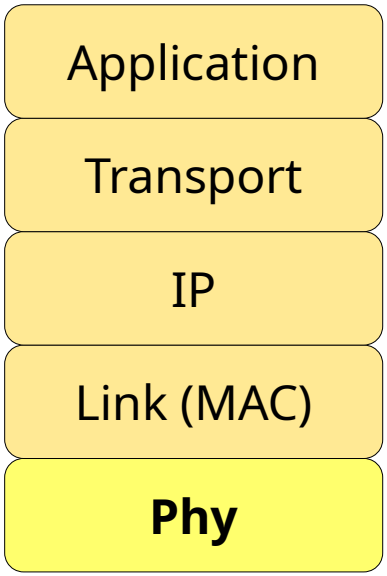
Basics of the Networking Stack

Networking Stack

- *Layer Stack*
  - Software uses a “network stack” to organize responsibilities into layers
  - Each layer provides a service to the layer above it
  - (This is exactly the same kind of *vague* architecture diagram as the virtualization diagram)

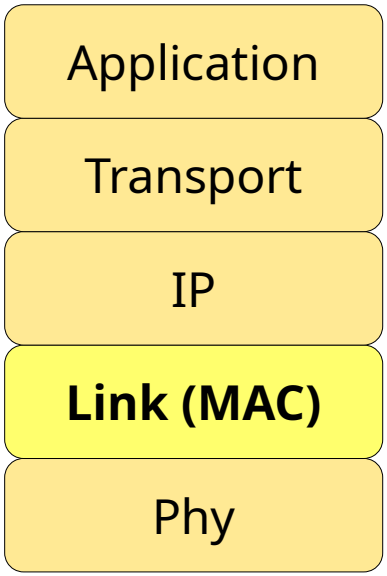


Physical Layer



- *Phy (Physical) Layer*
  - Hardware generates and receives signals
  - Need to know how to physically send and receive data
  - Standards focus on voltage, signalling
- *Analogy to package delivery*
  - truck and driver on a particular route, getting stuff from A to B

Link Layer

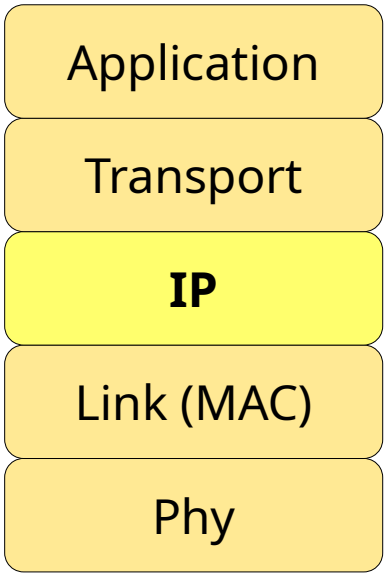


- *Link (MAC) Layer*
  - Does local network (“last hop”) addressing and routing
  - This is only for a (small area) local network
  - I.e., wired or wireless local network

    (LAN = Local Area Network)
- *Link layer uses MAC addresses*
  - MAC = Medium Access Control
  - Ethernet MAC addresses are 6 bytes formatted:

    05:35:5a:30:f9:05
- *Analogy to package delivery*
  - Street-level final address and route

IP (Network) Layer



- *IP ("Network") Layer*
  - Does inter-network addressing and routing
  - IP = Internet Protocol
- *What if you want to connect two different local networks?*
  - Still need addressing and routing, but how does traffic know when to cross from one network to the other?
  - How to address machines that aren't on your network?
- IP addresses look like: 192.168.7.53
- Or, IPv6: 2001:db8:3333:4444:5555:6666:7777:8888 (Yuck! No wonder it's still not common)

Wait, But Why?

- It may seem redundant to have multiple addressing methods. If it did not occur to you to wonder, **you can safely ignore this discussion. This material is purely provided for interest.**
- You may ask: sure, we need a common addressing scheme to connect two networks… but why IP?
  - Wireless and and wired Ethernet actually use the same MAC addresses
  - Moreover, it’s normal to bridge them without IP-level routing
  - We could have invented ways to route MAC addresses globally
- Conversely, you may ask: why does the link level have to use MAC addresses?
  - There are links that don’t, like SLIP
  - Why can’t we route IP natively over the local network?

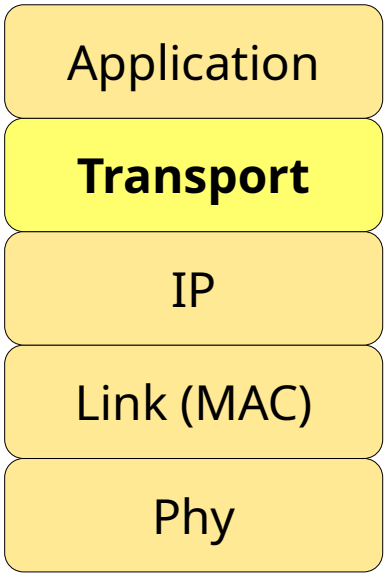
Wait, But Why? (cont'd)

- The reasons are partly historical: Ethernet and IP coexisted with a mix of other media and networking standards, interop was important
- But there is also an *organizational* element, about localizing decisions appropriately
  - IP and the Internet are designed with hierarchical, global-level control over network address assignment and routing
    - You do not want to be forced to deal with ICANN just to set up a LAN!
  - Machines on a LAN need to talk to each other directly to do network management
    - If the link layer is not separate from IP, all the details of technology- and vendor-specific protocols have to be negotiated globally
  - It may seem silly, but Ethernet MAC addresses and IP addresses being obviously different makes it easy to keep straight which level you're in

Wait, But Why? (cont'd)

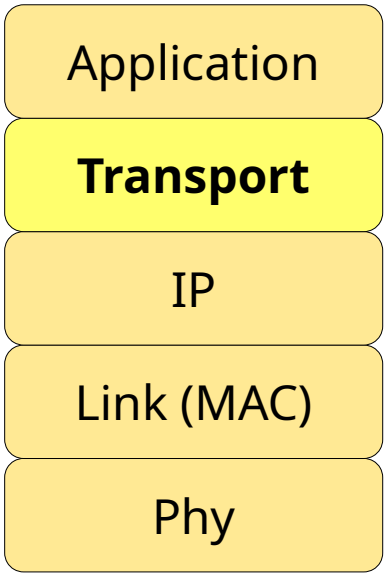
- It should be apparent that this architecture does not represent not a rigorous theoretical taxonomy
  - It's historical and pragmatic, and it elides a lot
  - I know we're sweeping a lot under the rug
- For now, every hardware-level address you encounter probably *will* be an Ethernet-style MAC, and every global address an IPv4 or IPv6 address
- **For practical work, we only need to bother with what *is*, not what *could be* or *might have been***

Transport Layer



- *Transport Layer*
  - Handles packet tracking/retransmission
- *Imagine shipping many packages: problems*
  - Packages can be lost
    - Package lost or stolen in transit, truck crashes
  - Packages can arrive out of order
    - They may be sent on different trucks, be accidentally delayed at a shipping hub
  - Packages can be duplicated
    - If the sender mistakenly thinks the package is lost and re-sends
- Transport layer protocols have to be supported by all the network infrastructure! This layer is intertwined with IP

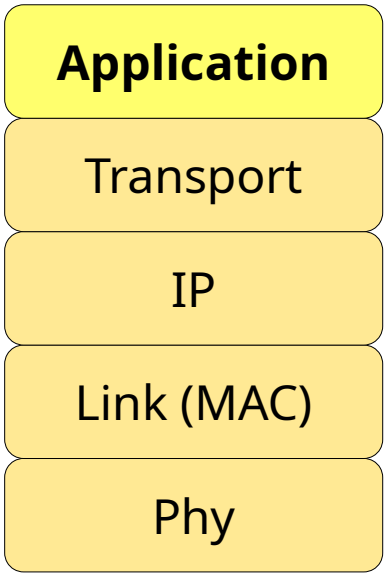
Transport Layer (cont'd)



- *Commonly supported IP transport protocols*
  - TCP (Transmission Control Protocol)
    - Provides a stream-like interface, read and write like a file
    - Guarantees in-order delivery without duplicates
    - Usually you can trust it (but e.g. checksums have non-negligible chance of missing errors during long transmissions!)
  - UDP (User Datagram Protocol)
    - Provides a datagram interface, atomic message blocks sent and received chunk-at-a-time
    - Makes no promises at the user level
    - There is packet fragmentation/reassembly and checksums under the hood (same caveats about errors!)
- *Port numbers*
  - Use a socket port number to distinguish different applications on the same host
  - Servers listen on well-known ports, e.g. port 80 or 443

Application Layer

- *Application Layer*
  - What the application (server or client) implements
  - Often a well-known protocol such as:
    - HTTP
    - SMTP
    - SSH
    - ...



Audience Participation - Spot the Address

- Which of the following is _____?
  1) an IP Address
  2) a MAC Address
  3) a Port Number

a) 8001

b) 19:02:16:08:07:01

c) 153.10.23.103

d) 0xF532 5E85 0005 235F

Sockets Interface

Socket Syscalls

- An application can use a **socket** to communicate with another process (local or remote)
- *There are five key socket-specific syscalls*
  - `socket()`
  - `bind()`
  - `listen()`
  - `accept()`
  - `connect()`

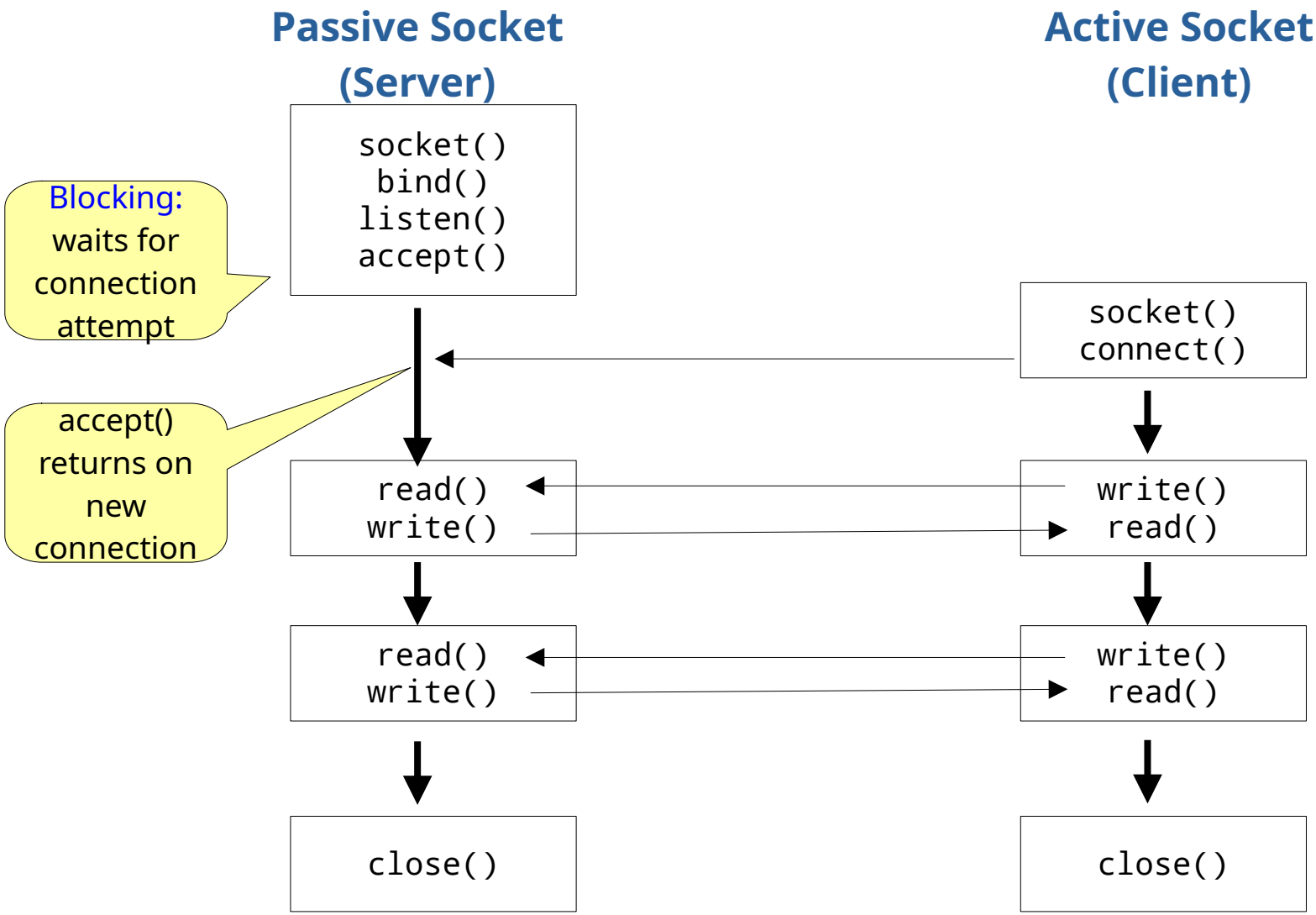
socket()

- `int socket(int domain, int type, int protocol)`
  - Returns a file descriptor
  - Functions to send/receive
    - socket-specific calls: `send()`, `recv()`, `sendto()`, `recvfrom()`
    - file I/O calls: `read()`, `write()`
- `int domain`
  - Wait, what?
    - We have been talking about TCP/IP, sockets is meant to be generic
    - In practice if you try to support many “domains” with the same code, you will be able to share some parts and not others
  - Domain examples
    - `AF_UNIX`: Local communication (filesystem paths as addresses)
    - `AF_INET`: IPv4 Internet protocols
    - `AF_INET6`: IPv6 Internet protocols

socket( ) (cont'd)

- `int socket(int domain, int type, int protocol)`
- `int type`
  - *SOCK_STREAM (TCP)*
    - Sequenced, reliable, two-way, connection-based byte stream
    - **Connection-based/connection-oriented**: will explain later
  - *SOCK_DGRAM (UDP)*
    - Datagrams (connectionless, unreliable packets of a max length)
    - **Connectionless**: will explain later.
- `int protocol`
  - *Always 0 for us;* not used for `AF_UNIX`, `AF_INET`, and `AF_INET6`
  - Some domains allow different protocols

Stream Socket Sequence (TCP)



TCP Explanation - `bind()`

- `socket()` creates a socket
- `bind()` binds the socket to an address
  - Digression: but what is *“binding”?*
    - One of those fuzzy CS terms like “virtual”
    - In general: associating a specific parameter value with an object, e.g. function parameter binding – in Python:

```
from functools import partial
f = partial(print, “hello”)
f(“world”)   # prints “hello world”
```

  - We are associating an *address* or a *port* with the socket
  - What this association *does* depends on the kind of socket

TCP Explanation - `bind()`

- `socket()` creates a socket
- `bind()` binds the socket to an address
- Uses a generic address struct
  - Different protocols use different structs (with different-yet-similar names, and different fields)
  - …`sockaddr` is not big enough for Unix domain or IPv6 addresses; can use `sockaddr_storage`, or if you know the address type for your domain, e.g. sockaddr_un or sockaddr_in6 you can use that)
  - You may be better off imagining that takes a `bind()` takes a `void*`

```
struct sockaddr {
    sa_family_t sa_family;
    char        sa_data[14];
    // size varies.
    // bind() given struct size.
};
```

TCP Explanation: `listen()`, `accept()`

- `listen()`: marks the socket as passive
  - i.e., it's used to wait for a connection to come (a server)
  - By default, a socket is active
- `accept()`: server (passive socket) accepts a new connection
  - Returns a separate (new) socket object for the specific connection
  - The original socket is only used to accept new connections
    - Yes, this is a slightly confusing overloading
- `connect()`: client (active socket) connects to a passive socket
  - Needed for “connection-oriented” protocols
  - *Can* be used with connectionless protocols, but not needed
    - Abstractly, it’s a kind of binding, but we don’t use that term to avoid confusion with `bind()`

Audience Participation -TCP Call Sequence

- Which of the following is the most likely sequence of calls for a TCP server?

a)

```
socket()
bind()
listen()
read()
accept()
write()
close()
```

b)

```
socket()
bind()
listen()
accept()
read()
write()
close()
```

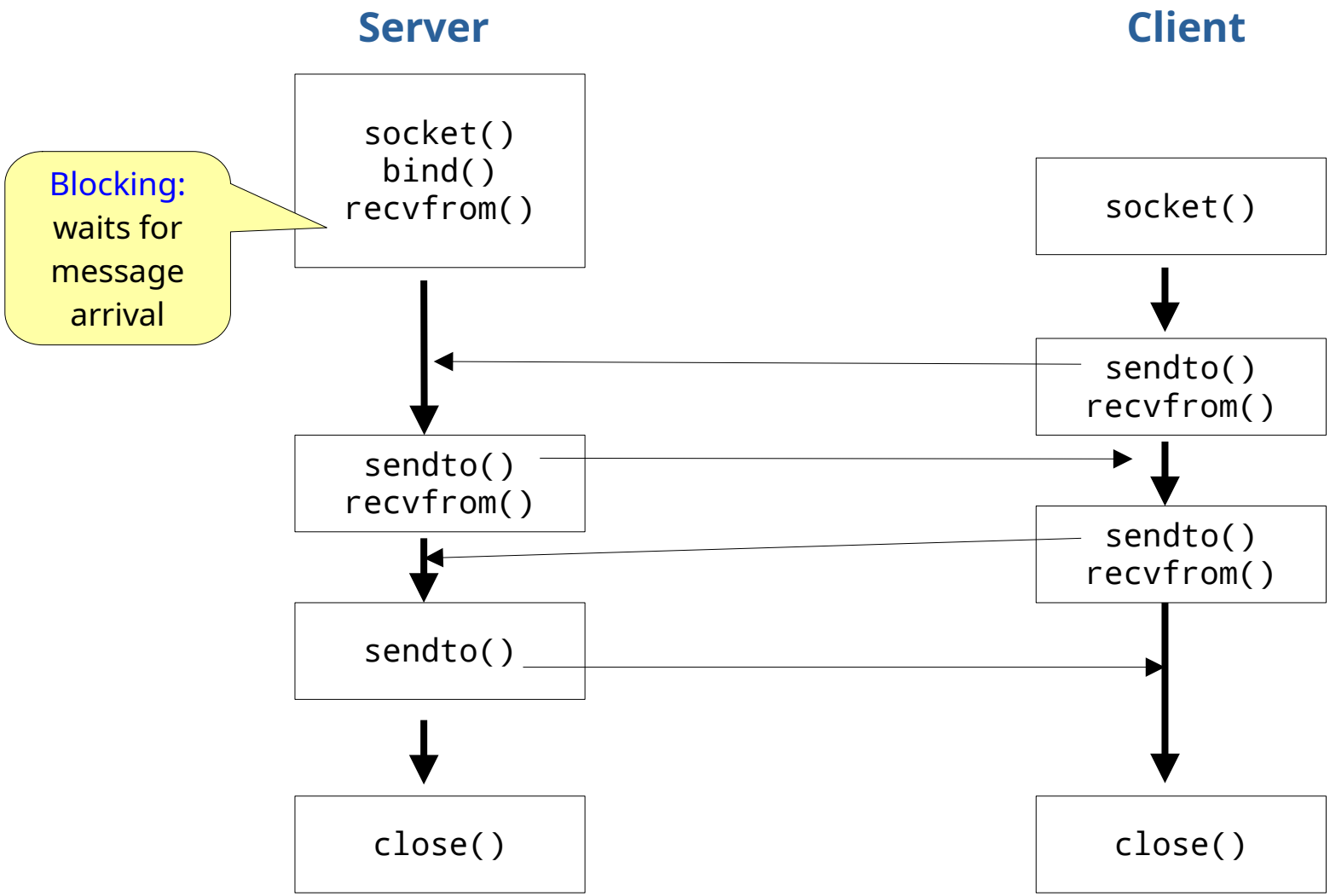
c)

```
socket()
bind()
listen()
accept()
write()
read()
close()
```

d)

```
socket()
bind()
write()
listen()
accept()
read()
close()
```

Datagram Socket Sequence (UDP)



UDP Explanation

- “connectionless” means we do not establish a connection first
  - “Datagrams” are like text messages, each one is sent independently
  - Each time we receive a message we are told who sent it
- UDP has no active or passive sockets
  - `sendto()` needs to specify the receiver's address
    - Caveat: `connect()` can set a *default*, but this is part of the API, not the actual network protocol
  - `recvfrom()` tells you who sent the message

Audience Participation - UDP Call Sequence

- Which of the following is the most likely sequence of calls for a UDP server?

a)

```
socket()
bind()
sendto()
recvfrom()
close()
```

b)

```
socket()
bind()
listen()
sendto()
close()
```

c)

```
socket()
bind()
read()
write()
close()
```

d)

```
socket()
bind()
recvfrom()
sendto()
close()
```

Audience Participation - Whose Call Is It?

- Which of the options on the right is most likely to use all of the following calls (not in order):

connect()
close()
read()
socket()
write()

a) UDP Client

b) UDP Server

c) TCP Client

d) TCP Server

Audience Participation - Whose Call Is It?

- Which of the options on the right is most likely to use all of the following calls (not in order):

bind()
close()
recvfrom()
sendto()
socket()

a) UDP Client

b) UDP Server

c) TCP Client

d) TCP Server

Audience Participation - Whose Call Is It?

- Which of the options on the right is most likely to use all of the following calls (not in order):

accept()
bind()
close()
listen()
read()
socket()
write()

a) UDP Client

b) UDP Server

c) TCP Client

d) TCP Server

TCP Activity

- *Create two TCP programs: server and client (15m)*
  - Implement the socket sequence using `AF_UNIX`
    - I know it's not technically TCP: quiet, pedant
  - The client should be able to send messages typed on the terminal to the server
  - The server should be able to print out the messages
  - `man unix` for detailed info for `AF_UNIX`
  - An AF_UNIX address uses `struct sockaddr_un`:

```
struct sockaddr_un {
    sa_family_t sun_family;         /* AF_UNIX */
    char        sun_path[108];      /* Pathname = "tmp" */
};
```

UDP Activity

- *Create two UDP programs: server and client (15m)*
  - Implement the socket sequence using `AF_UNIX`
    - This is technically not UDP either
  - The client should be able to send messages typed on the terminal to the server
  - The server should be able to print out the messages
  - `man unix` for detailed info for `AF_UNIX`
  - An AF_UNIX address uses `struct sockaddr_un`:

```
struct sockaddr_un {
    sa_family_t sun_family;         /* AF_UNIX */
    char        sun_path[108];      /* Pathname = “tmp” */
};
```

Summary

- *Network Stack has layers (from bottom to top)*
  - phy, link, IP, transport, application
- ***Socket***
  - OS object used to communicate across network, manage connections
- *Transport-layer protocols*
  - **TCP**
    - Connection-oriented, in-order delivery
    - Server: `socket()`, `bind()`, `listen()`, `accept()`, `read()`, `write()`… `close()`
    - Client: `socket()`, `connect()`, `write()`, `read()`… `close()`
  - **UDP**
    - Connectionless
    - Server: `socket()`, `bind()`, `recvfrom()`, `sendto()`… `close()`
    - Client: `socket()`, `sendto()`, `recvfrom()`… `close()`