

# CSE 333

## Section 8

Client-side Networking

Netcat is listening on  
port 80

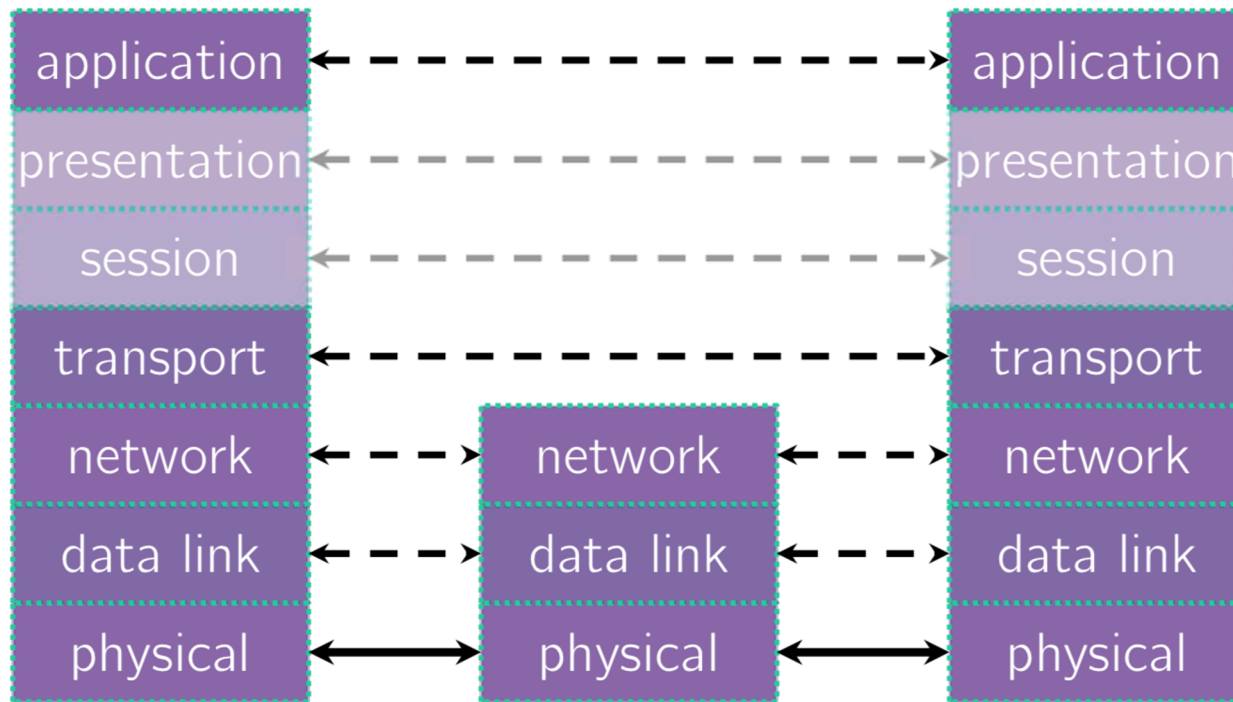


# Logistics

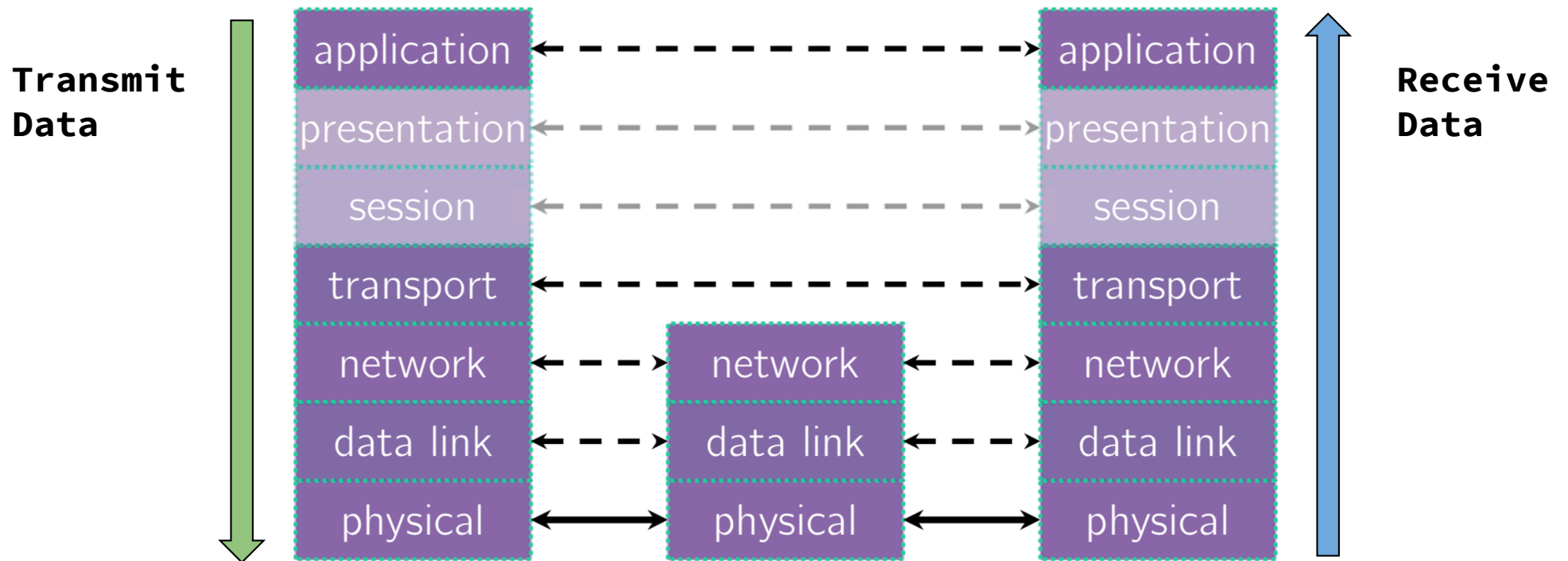
- Exercise 15:
  - Out later today
  - Due **Monday** (5/23) @ 10:00am
- Exercise 16:
  - Out after lecture tomorrow
  - Due **Wednesday** (5/25) @ 10:00am
- Homework 3:
  - Due **Tonight** (5/19) @ 11:00pm

# Computer Networking Review

# Computer Networks: A 7-ish Layer Cake

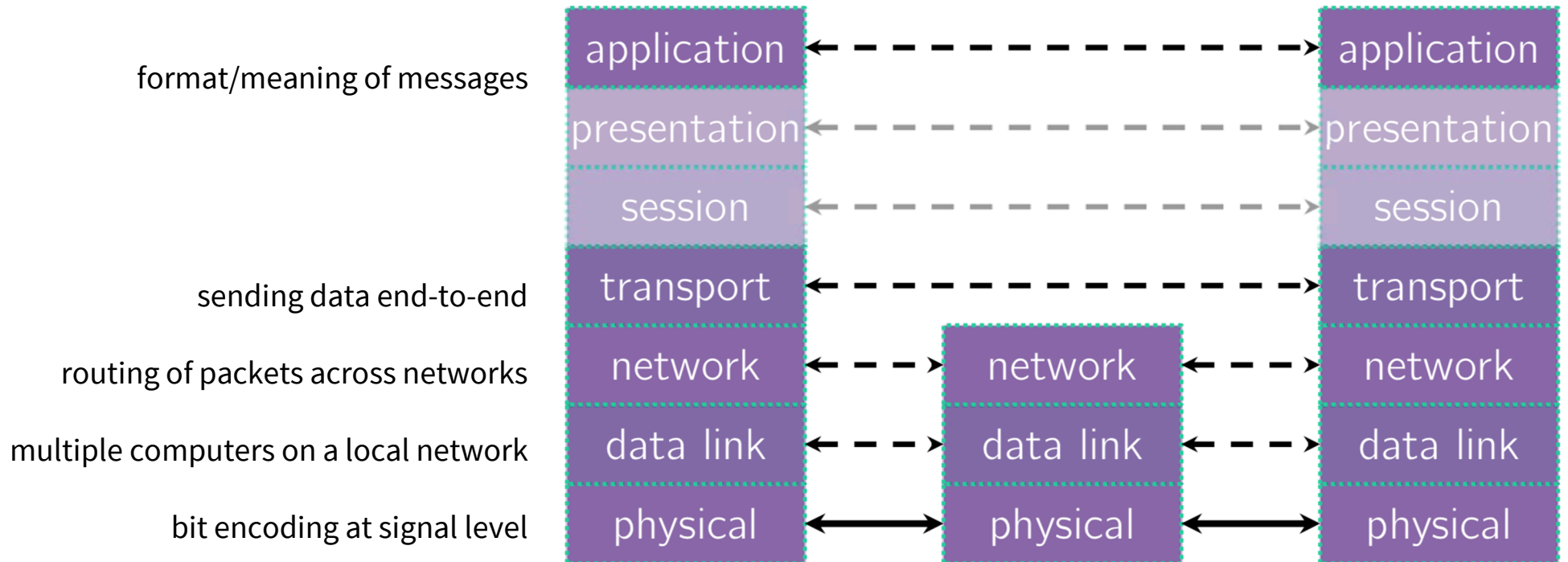


# Data Flow



# Computer Networks: A 7-ish Layer Cake

HTTP, DNS, anything else?



# Exercise 1

# Terminology Review

- DNS: Translating between IP addresses and host names. (Application Layer)
- IP: Routing packets across the Internet. (Network Layer)
- TCP: Reliable transport protocol on top of IP. (Transport Layer)
- UDP: Unreliable transport protocol on top of IP. (Transport Layer)



# TCP versus UDP

## Transmission Control Protocol (TCP):

- Connection-oriented Service
- Reliable and Ordered
- Flow control

## User Datagram Protocol (UDP):

- “Connectionless” service
- Unreliable packet delivery
- High speed, no feedback

TCP guarantees reliability for things like messaging or data transfers. UDP has less overhead since it doesn't make those guarantees, but is often fine for streaming applications (e.g., YouTube or Netflix) or other applications that manage packets on their own or do not want occasional pauses for packet retransmission or recovery.

# Client-Side Networking

# Client-Side Networking in 5 Easy\* Steps!

1. Figure out what IP address and port to talk to
2. Build a socket from the client
3. Connect to the server using the client socket and server socket
4. Read and/or write using the socket
5. Close the socket connection

\*difficulty is subjective

# 1. Figuring out the port and IP

- Performs a **DNS Lookup** for a hostname
- Use “hints” to specify constraints (struct addrinfo\*)
- Get back a linked list of struct addrinfo results

```
int getaddrinfo(const char* hostname,
               const char* service,
               const struct addrinfo* hints,
               struct addrinfo** res);
```

**Name of host whose IP we want**

**We will set this to nullptr to get the default; otherwise you can specify service/port**

**Output parameter; \*res is set to the first result in LL**

**Hints for the lookup server/refine results**

# Obtaining your server's socket address

```
struct addrinfo {  
    int ai_flags;           // additional flags  
    int ai_family;         // AF_INET, AF_INET6, AF_UNSPEC  
    int ai_socktype;       // SOCK_STREAM, SOCK_DGRAM, 0  
    int ai_protocol;       // IPPROTO_TCP, IPPROTO_UDP, 0  
    size_t ai_addrlen;     // length of socket addr in bytes  
    struct sockaddr* ai_addr; // pointer to socket addr  
    char* ai_canonname;     // canonical name  
    struct addrinfo* ai_next; // can have linked list of records  
}
```

- `ai_addr` points to a struct `sockaddr` describing a socket address, can be IPv4 or IPv6

# Understanding struct sockaddr\*

- It's just a pointer. To use it, we're going to have to dereference it and cast it to the right type (Very strange C "inheritance")
  - It is the endpoint your connection refers to
- Convert to a struct sockaddr\_storage
  - Read the sa\_family to determine whether it is IPv4 or IPv6
  - IPv4: AF\_INET (macro) → cast to struct sockaddr\_in
  - IPv6: AF\_INET6 (macro) → cast to struct sockaddr\_in6

# Understanding Socket Addresses

**struct sockaddr** (pointer to this struct is used as parameter type in system calls)

fam	????
-----	------

....

**struct sockaddr\_in** (IPv4)

fam	port	addr	zero
-----	------	------	------

16

**struct sockaddr\_in6** (IPv6)

fam	port	flow	addr	scope
-----	------	------	------	-------

28

**struct sockaddr\_storage**

fam	
-----	--

Big enough to hold either

# Sockets (Berkeley Sockets)

- Defines a local endpoint communication between server and client
  - Similar to a file descriptor for network communication
  - Built on various operating system calls
- Each socket is associated with **a port number (uint16\_t)** and **an IP address**
  - Both port and address are stored in network byte order (big endian)
  - `ai_family` will help you to determine what is stored for your socket!



# Building a Connection

2. Create a client socket to manage (returns an integer file descriptor, just like POSIX open)

```
// returns file descriptor on success, -1 on failure (errno set)
int socket(int domain,           // AF_INET, AF_INET6, etc.
           int type,            // SOCK_STREAM, SOCK_DGRAM, etc.
           int protocol);       // just put 0 (network abstraction)
```

3. Use that created client socket to connect to the server socket

```
// Connects to the server
// returns 0 on success, -1 on failure (errno set)
int connect(int sockfd,          // socket file descriptor
            struct sockaddr *serv_addr, // socket addr of server
            socklen_t addrlen);   // size of serv_addr
```

Usually from `getaddrinfo!`

## Using your Connection (Steps 4 and 5)

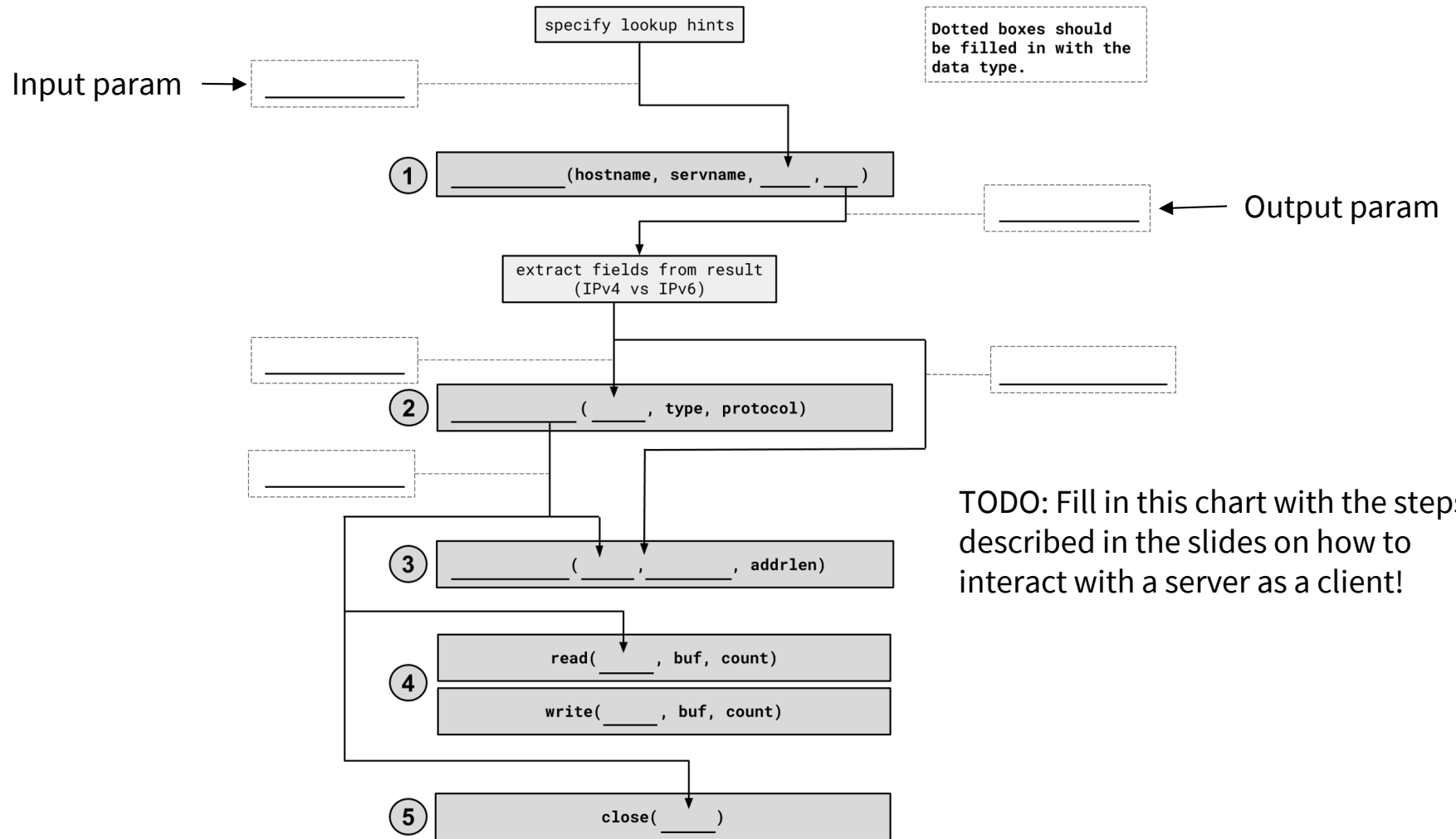
```
// returns amount read, 0 for EOF, -1 on failure (errno set)
ssize_t read(int fd, void *buf, size_t count);
```

```
// returns amount written, -1 on failure (errno set)
ssize_t write(int fd, void *buf, size_t count);
```

```
// returns 0 for success, -1 on failure (errno set)
int close(int fd);
```

- Same POSIX methods we used for file I/O!  
(so they require the same error checking...)

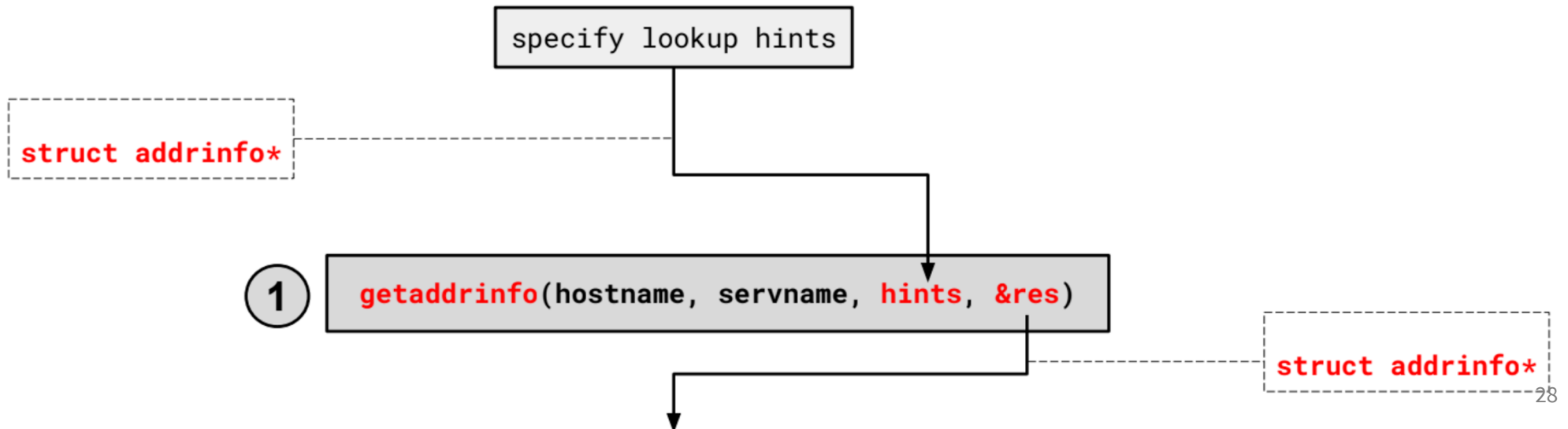
## Exercise 2



# 1. getaddrinfo()

```
int getaddrinfo(const char* hostname,  
               const char* service,  
               const struct addrinfo* hints,  
               struct addrinfo** res);
```

- Performs a **DNS Lookup** for a hostname
- Use “hints” to specify constraints (struct addrinfo\*)
- Get back a linked list of struct addrinfo results



# 1. getaddrinfo() - Interpreting Results

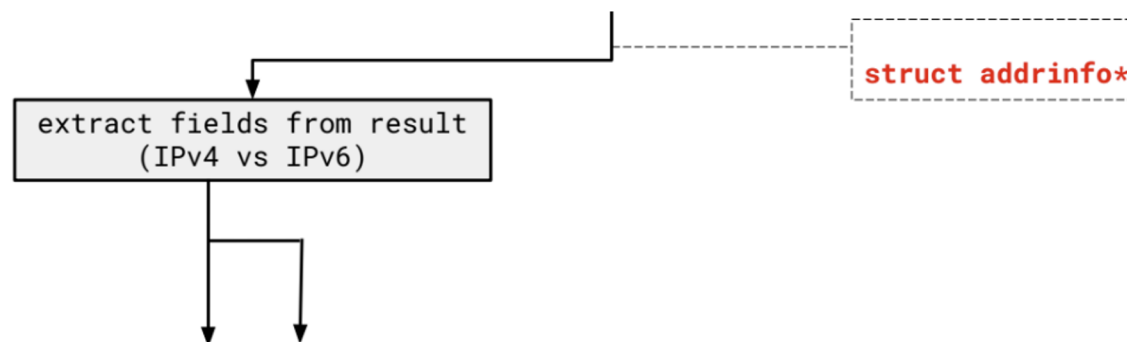
```
struct addrinfo {  
    int ai_family; // AF_INET, AF_INET6, AF_UNSPEC  
    struct sockaddr* ai_addr; // pointer to socket addr  
    ...  
};
```

- These records are dynamically allocated; you should pass the head of the linked list to `freeaddrinfo()`
- The field `ai_family` describes if it is IPv4 or IPv6
- `ai_addr` points to a `struct sockaddr` describing the socket address

# 1. getaddrinfo() - Interpreting Results

With a struct `sockaddr*`:

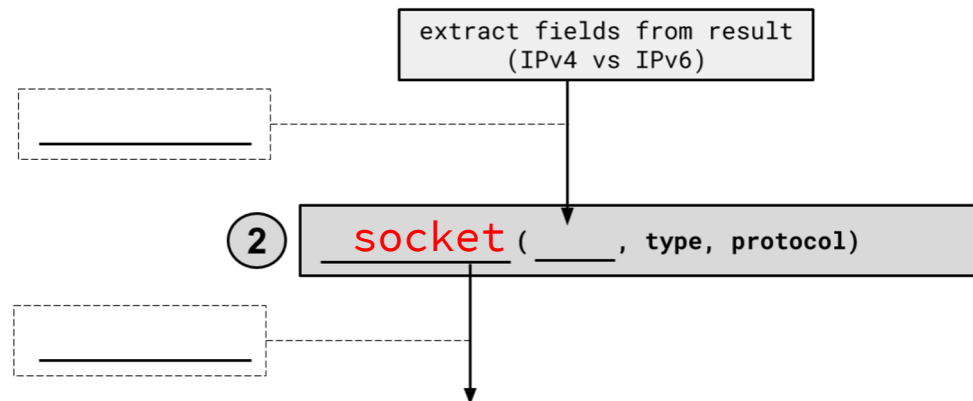
- The field `sa_family` describes if it is IPv4 or IPv6
- Cast to `struct sockaddr_in*` (v4) or `struct sockaddr_in6*` (v6) to access/modify specific fields
- Store results in a `struct sockaddr_storage` to have a space big enough for either



## 2. Build client side socket

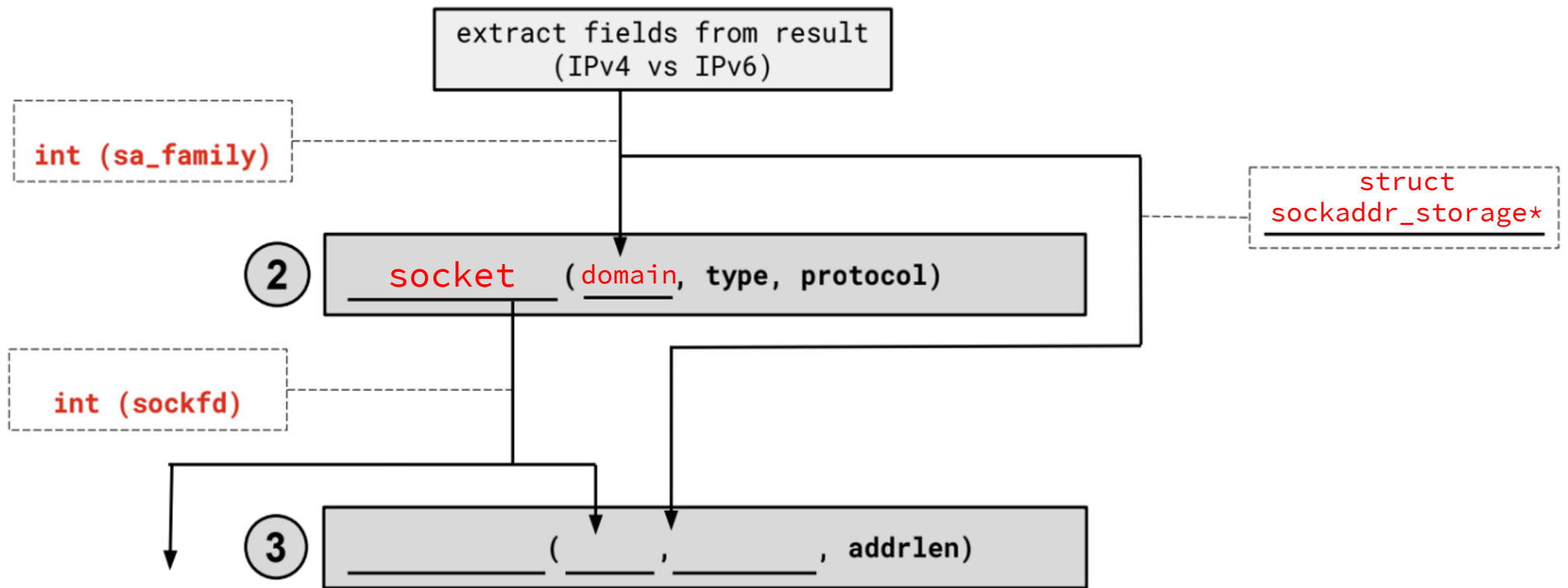
```
int socket(int domain,    // AF_INET, AF_INET6
           int type,     // SOCK_STREAM (for TCP)
           int protocol); // 0 for the default
```

- This gives us an unbound socket that's not connected to anywhere in particular
- Returns a socket file descriptor (we can use it everywhere we can use any other file descriptor as well as in socket specific system calls)





## 2. Build client side socket



## 3. connect()

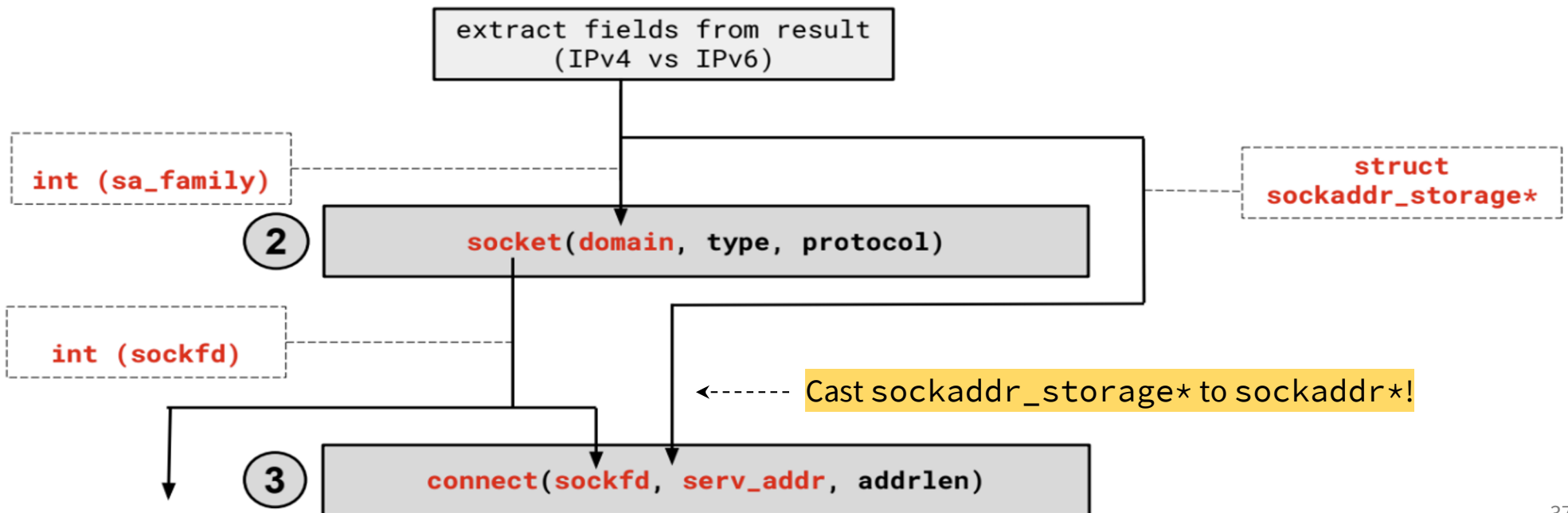
```
int connect(int socket,                // socket fd
            const struct sockaddr *addr, // address to connect to
            socklen_t addr_len);       // length of *addr
```

- This takes our unbound socket and connects it to the host at addr
- Returns 0 on success, -1 on error with errno set appropriately
- After this call completes, we can actually use our socket for communication!

### 3. connect()

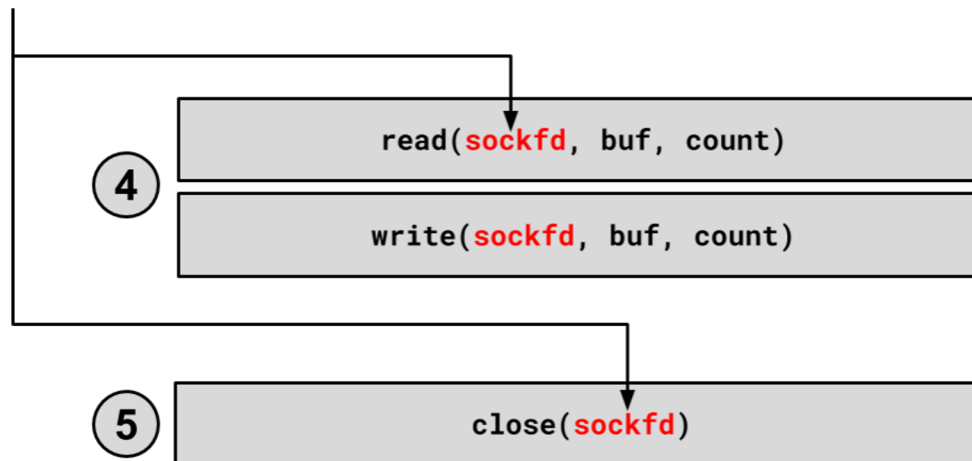
```
int connect(int socket, // from 1
            const struct sockaddr *addr, // from 2
            socklen_t addr_len); // size of serv_addr
```

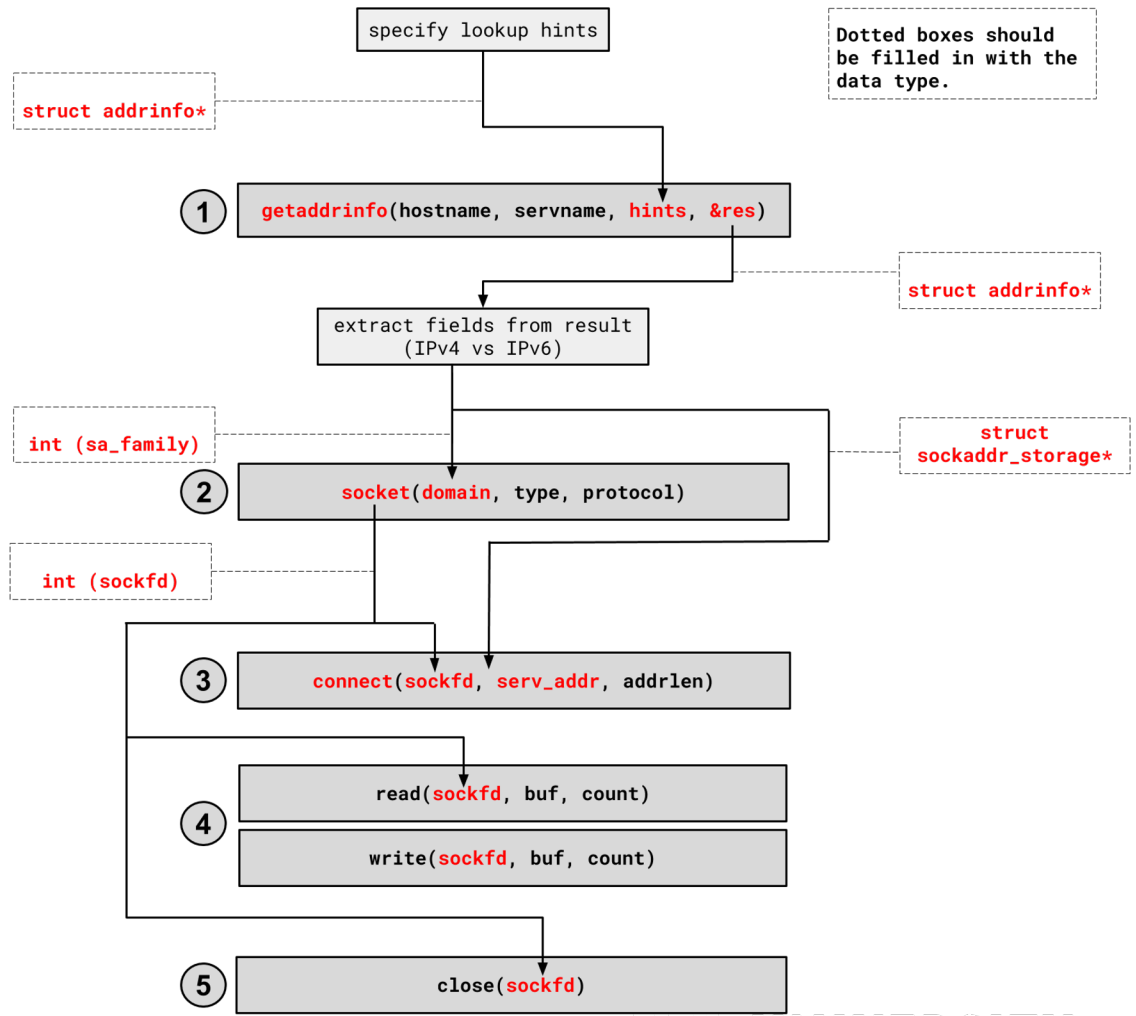
- Connects an available socket to a specified address
- Returns 0 on success, -1 on failure



## 4. read/write and 5. close

- Thanks to the file descriptor abstraction, use as normal!
- read from and write to a buffer, the OS will take care of sending/receiving data across the network
- Make sure to close the fd afterward





# Using Netcat for the first time

**netcat**

