

CSE 333

Section 7

Client-side Networking
& ex10-11 demo



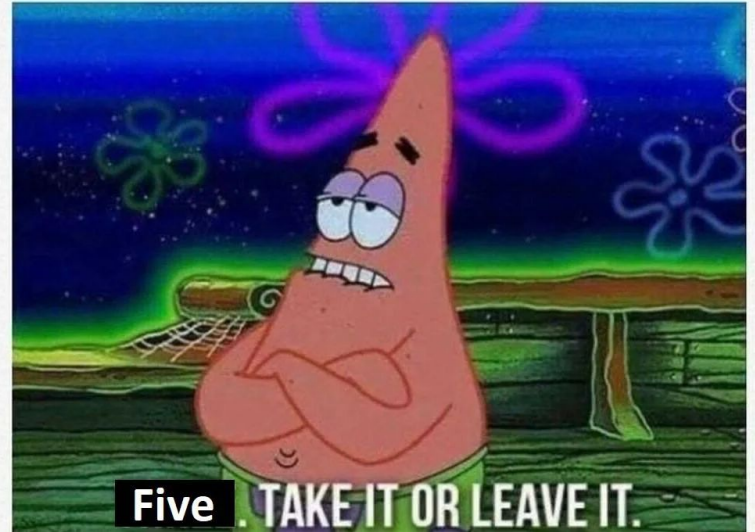
Logistics

- Homework 3:
 - Due **tonight (8/3) @ 11:59pm**
 - Late deadline is Sunday @ 11:59 PM if you use 2 late days
- Exercise 10:
 - Due **Monday (8/7) @ 1 pm**
- Exercise 11:
 - Due **Thursday (8/10) @ 1 pm**
- Quiz 3:
 - Releases Monday (8/7) @ 2 pm, due **Wednesday (8/9) @ 11:59 PM**
- Homework 4:
 - Releases tomorrow, due **Wednesday (8/16) @ 11:59 PM**

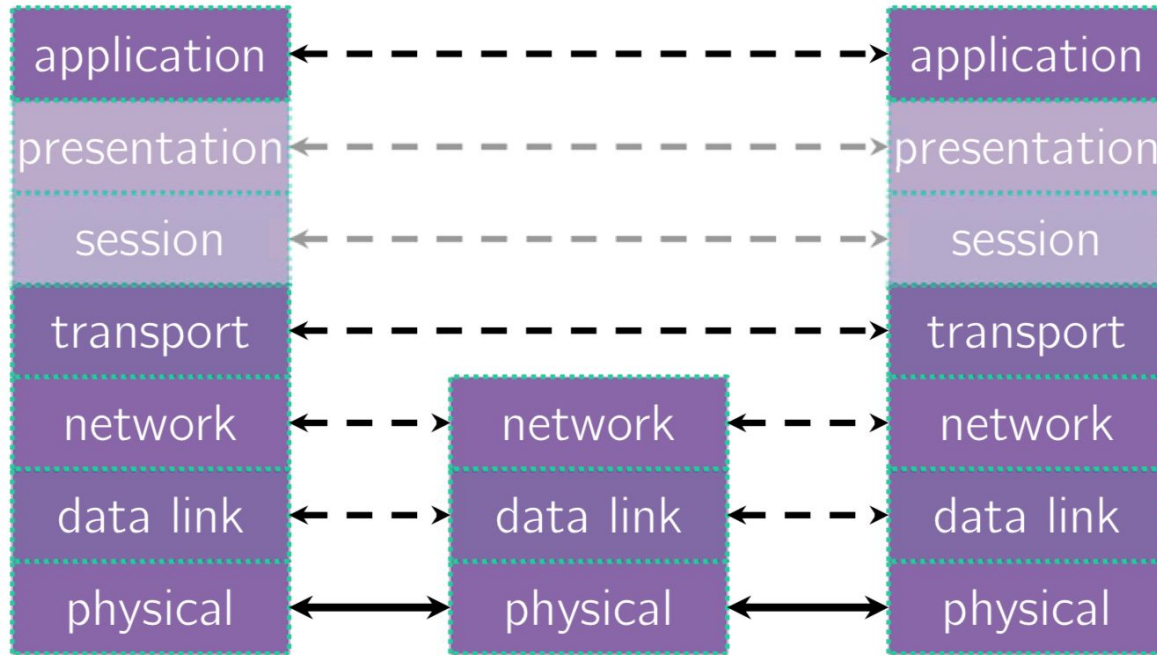
Computer Networking - At a High Level

Interviewer: this role requires knowledge in the
7 layer internet model

Me:



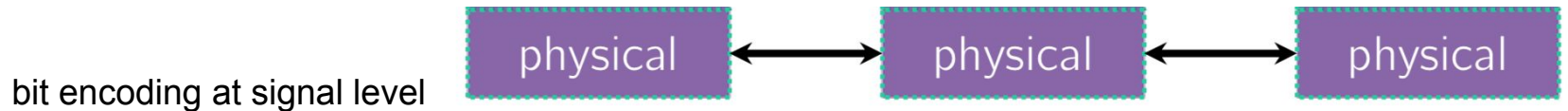
Computer Networks: A 7-ish Layer Cake



Computer Networks: A 7-ish Layer Cake

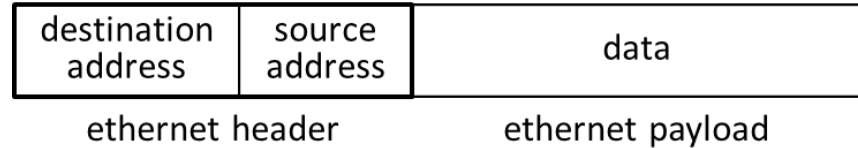
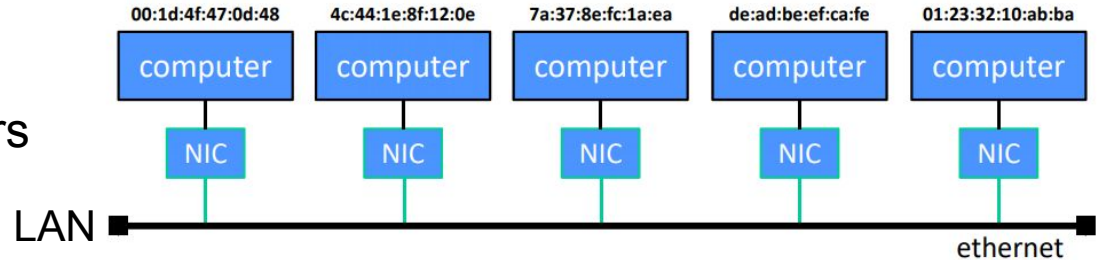


Wires, radio signals, fiber optics



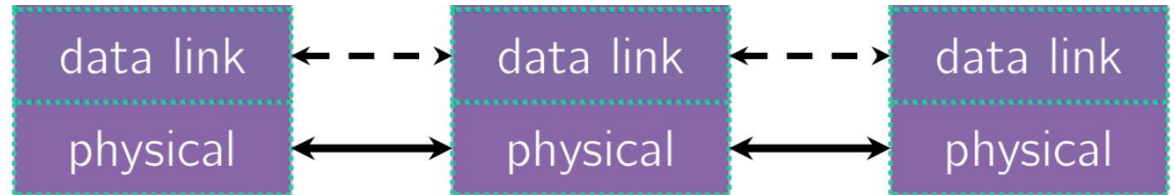
Computer Networks: A 7-ish Layer Cake

WiFi, ethernet.
Connecting multiple computers

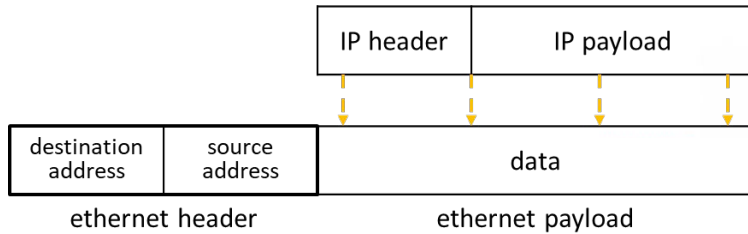
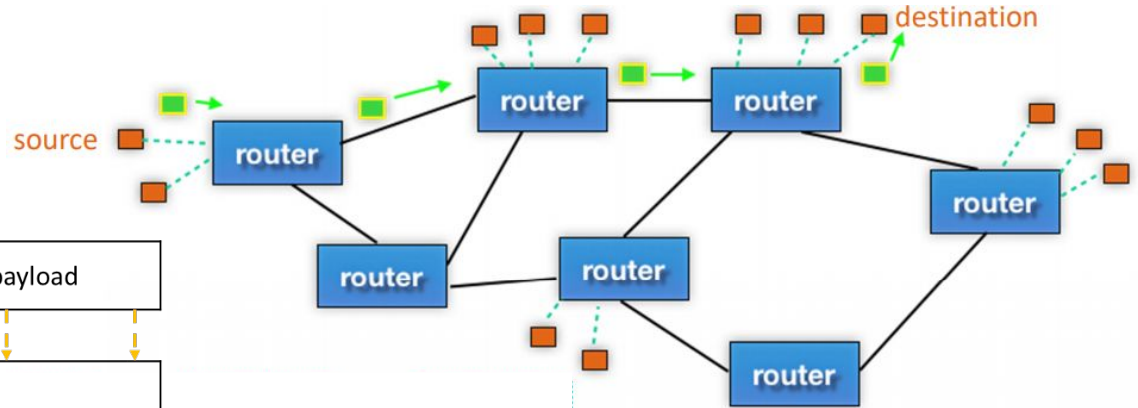


multiple computers on a local network

bit encoding at signal level



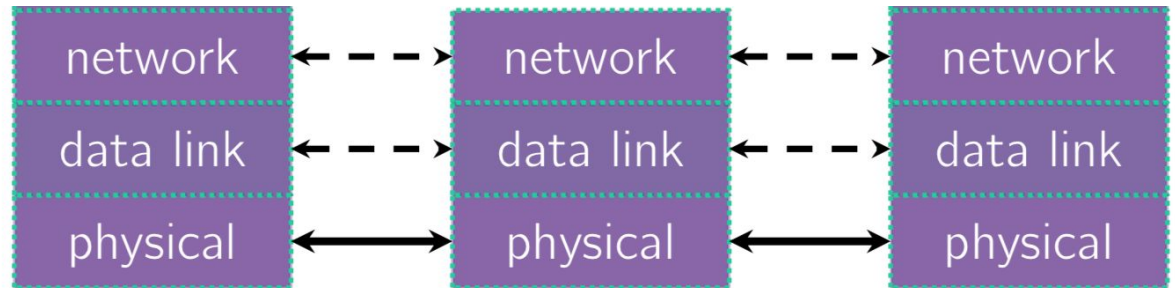
Computer Networks: A 7-ish Layer Cake



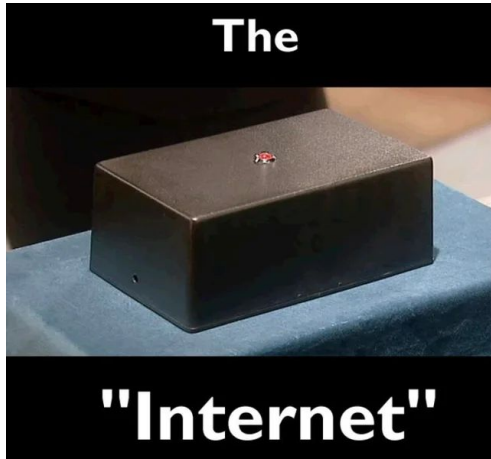
routing of packets across networks

multiple computers on a local network

bit encoding at signal level

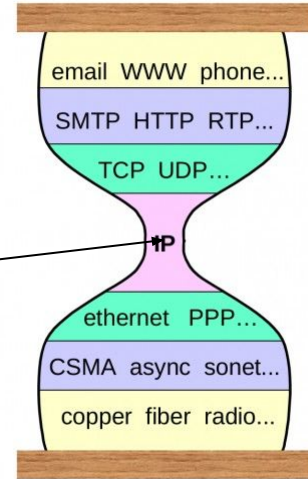


Computer Networks: A 7-ish Layer Cake



on/Interface

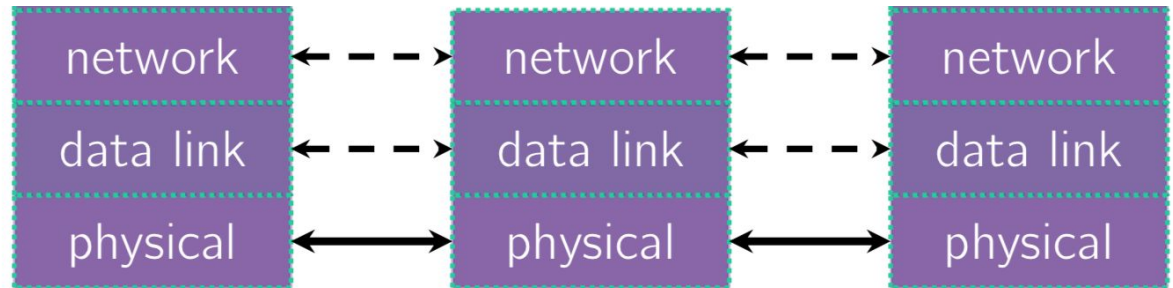
Backbone of the Internet!



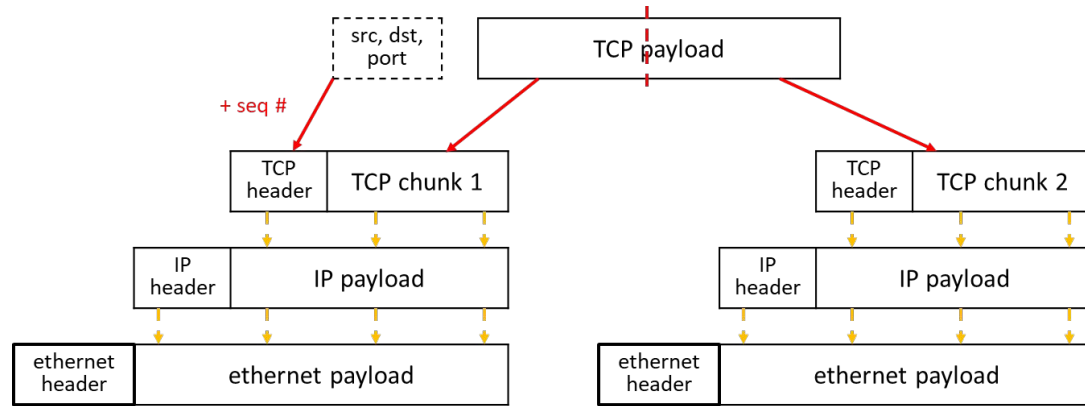
routing of packets across networks

multiple computers on a local network

bit encoding at signal level

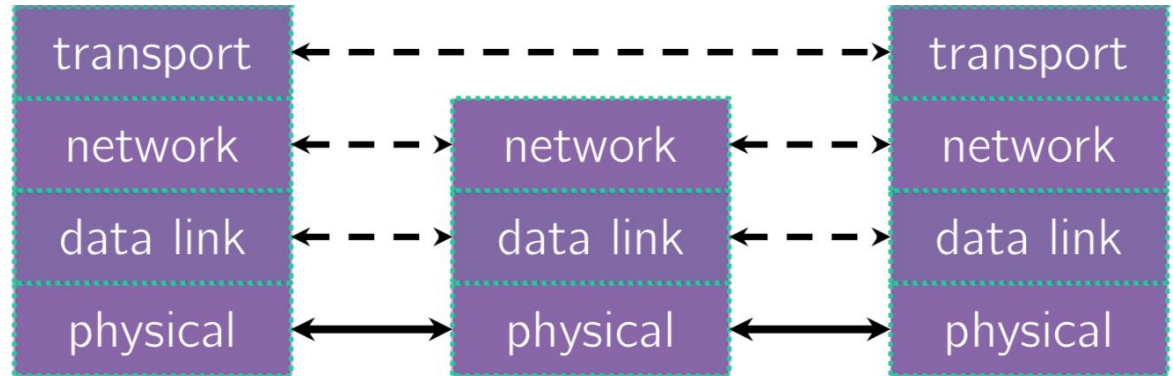


Computer Networks: A 7-ish Layer Cake

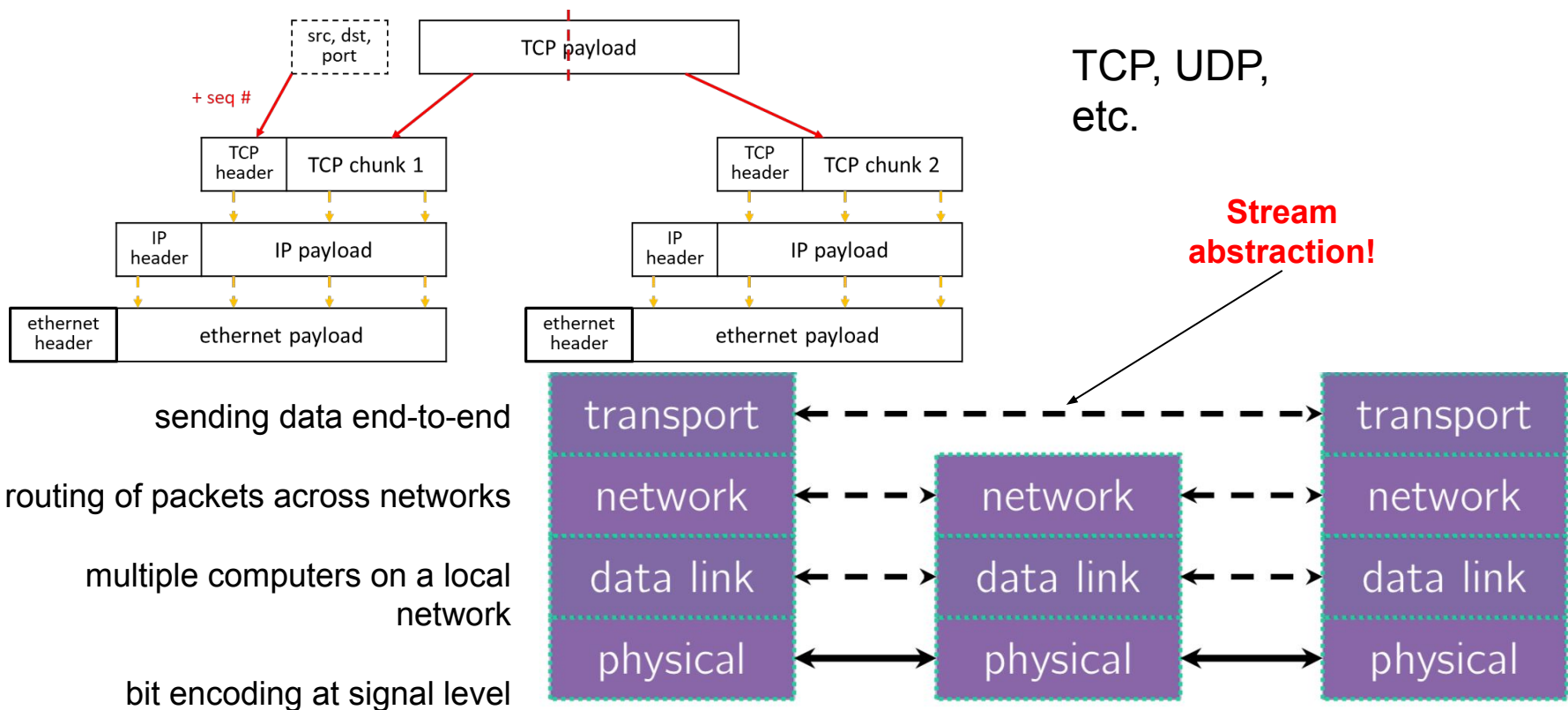


TCP, UDP,
etc.

- sending data end-to-end
- routing of packets across networks
- multiple computers on a local network
- bit encoding at signal level

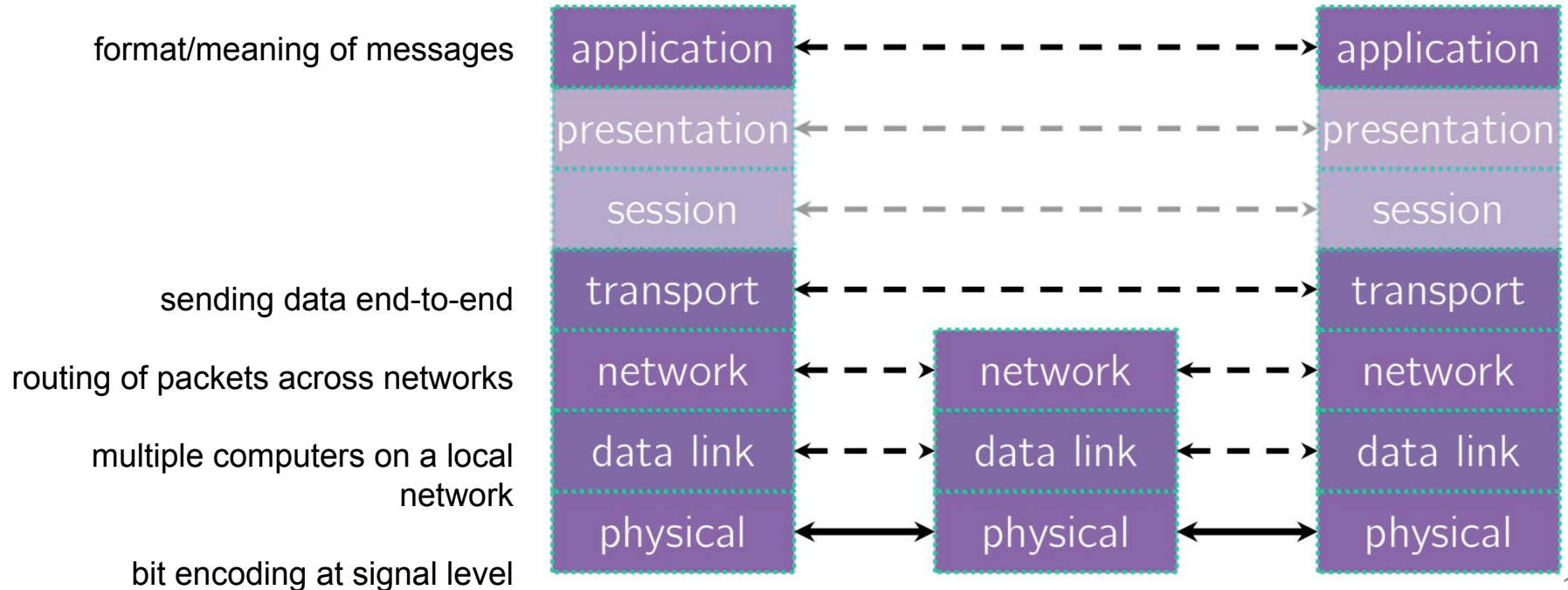


Computer Networks: A 7-ish Layer Cake

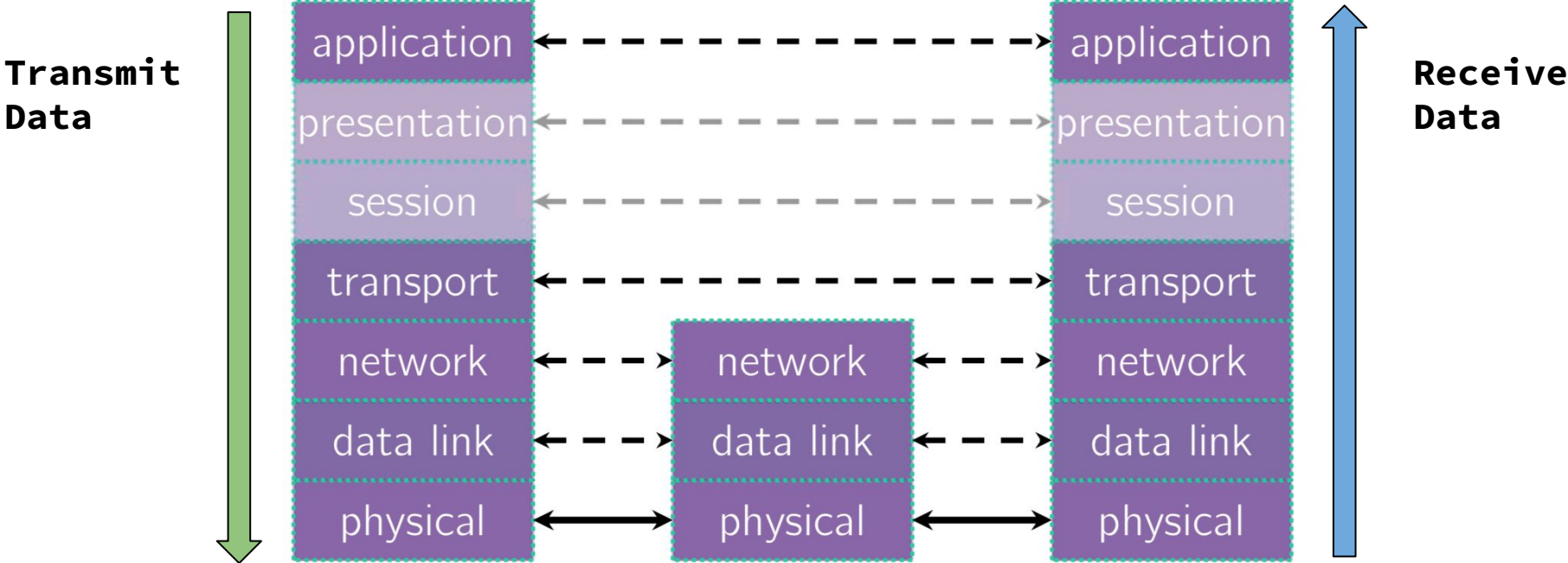


Computer Networks: A 7-ish Layer Cake

HTTP, DNS, anything else?



Data Flow



Exercise 1

Exercise 1

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

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

Remember these functionalities are from the **C standard library**, though we are using them in our C++ programs (but they're coming to C++26 🤖)

*difficulty is subjective

Sockets (Berkeley Sockets)

- Just a file descriptor for network communication
 - Defines a local endpoint for network communication
 - Built on various operating system calls
- Types of Sockets
 - Stream sockets (TCP)
 - Datagram sockets (UDP)
 - There are other types, which we will not discuss
- Each socket is associated with a **port number (`uint16_t`)** and an **IP address**
 - Remember to convert between host order and network order!
https://www.gnu.org/software/libc/manual/html_node/Byte-Order.html
 - `ai_family` will help you to determine what is stored for your socket!



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

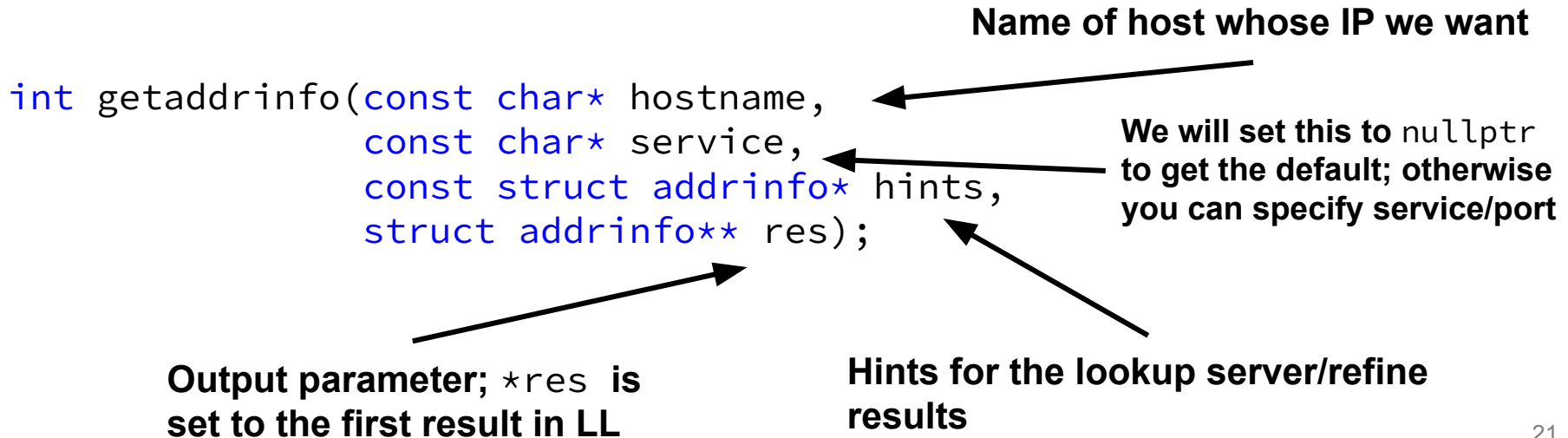
Understanding struct sockaddr*

- 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

Step 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



Step 1: 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

Steps 2 and 3: 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!

Steps 4 and 5: Using your Connection

```
// 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...)

Helpful References

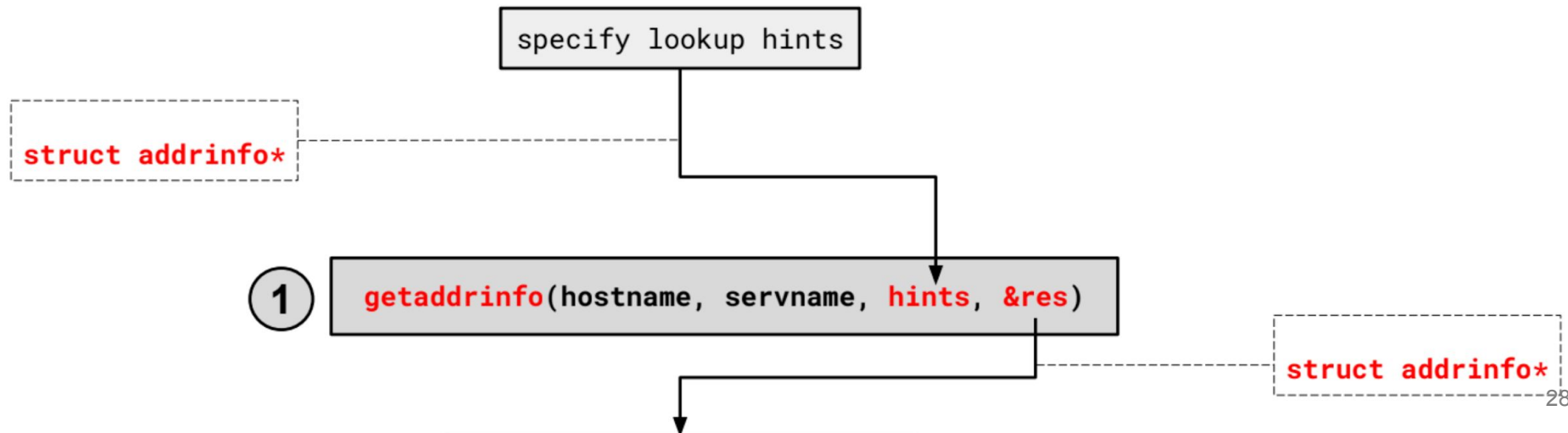
1. Figure out what IP address and port to talk to
 - dnsresolve.cc
2. Build a socket from the client
 - connect.cc
3. Connect to the server using the client socket and server socket
 - sendreceive.cc
4. Read and/or write using the socket
 - sendreceive.cc (same as above)
5. Close the socket connection

Exercise 2

1. getaddrinfo()

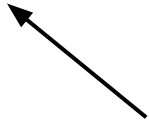
```
int getaddrinfo(const char* hostname,  
               const char* service,  
               const struct addrinfo* hints,  
               struct addrinfo** res);
```

- 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_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 sockaddr for address  
    char* ai_canonname; // canonical name  
    struct addrinfo* ai_next; // can form a linked list  
};
```

*Note that we get a linked list of results

1. getaddrinfo() - Interpreting Results

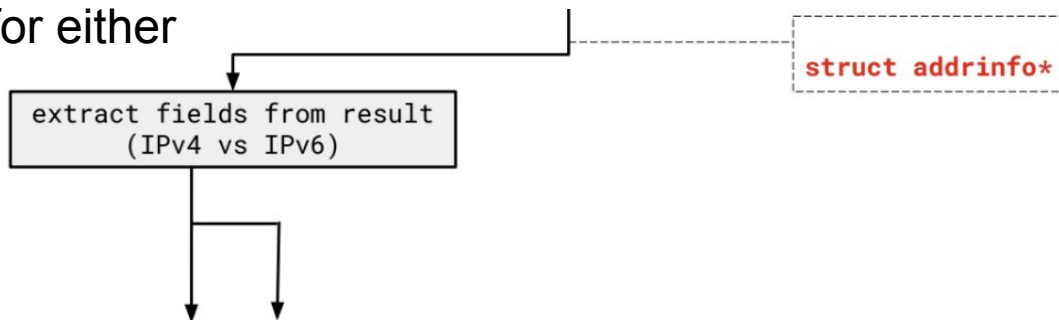
```
struct addrinfo {  
    int ai_family; // AF_INET, AF_INET6, AF_UNSPEC  
    struct sockaddr* ai_addr; // pointer to socket addr  
    ...  
};
```

- 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*`:

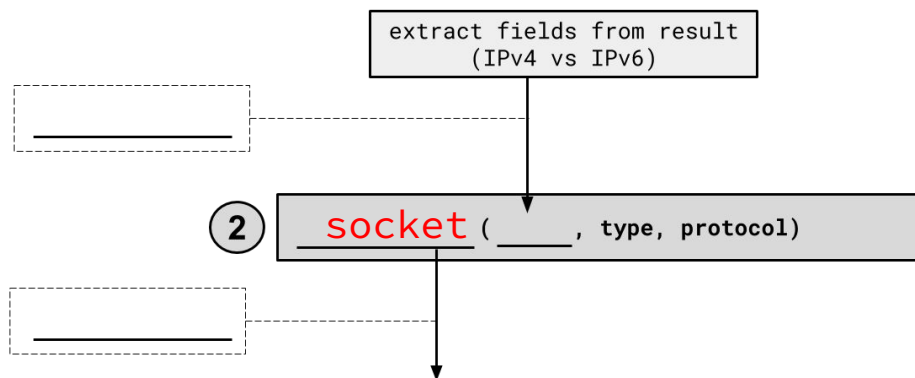
- The field `sa_family` describes if it is IPv4 or IPv6
- Cast to `struct sockaddr_in*` (v4) or `struct sockaddr_in6*` (v6) to access/modify specific fields (i.e. ports)
- Store results in a `struct sockaddr_storage` to have a space big enough for either



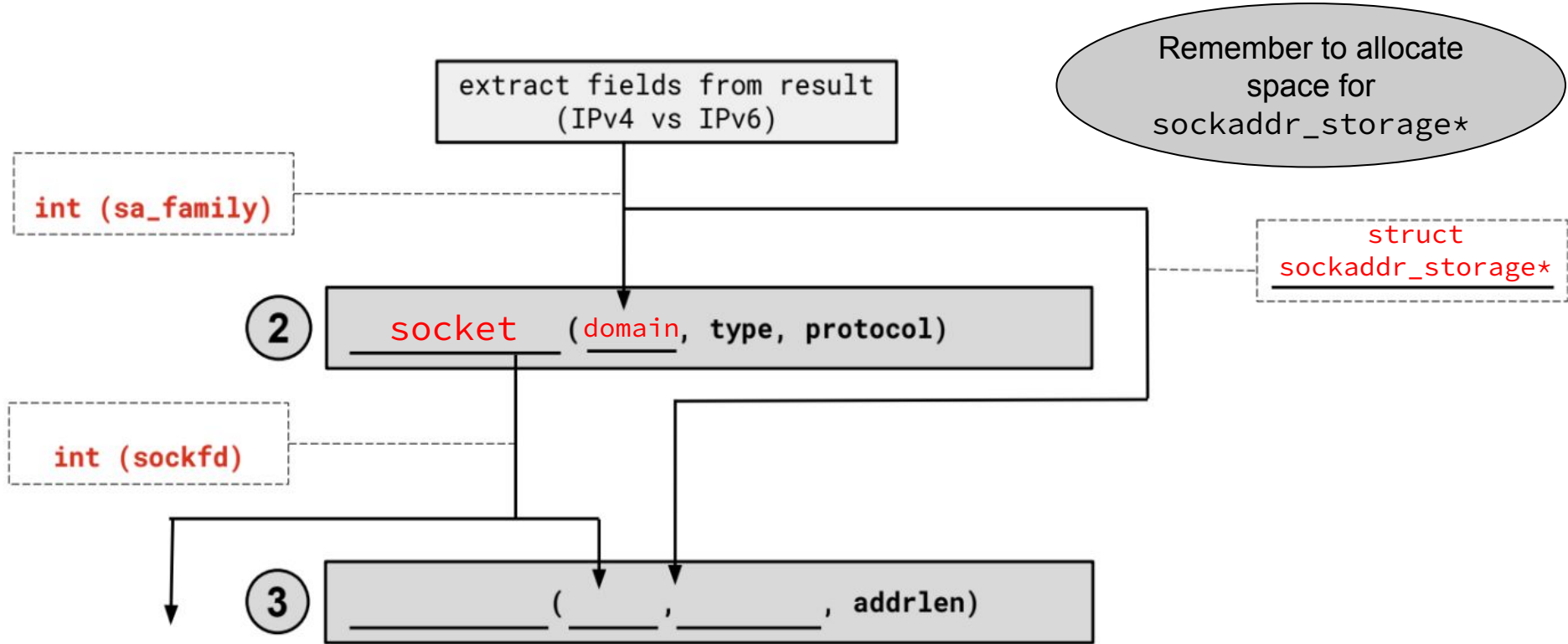
2. Build client side socket

```
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()

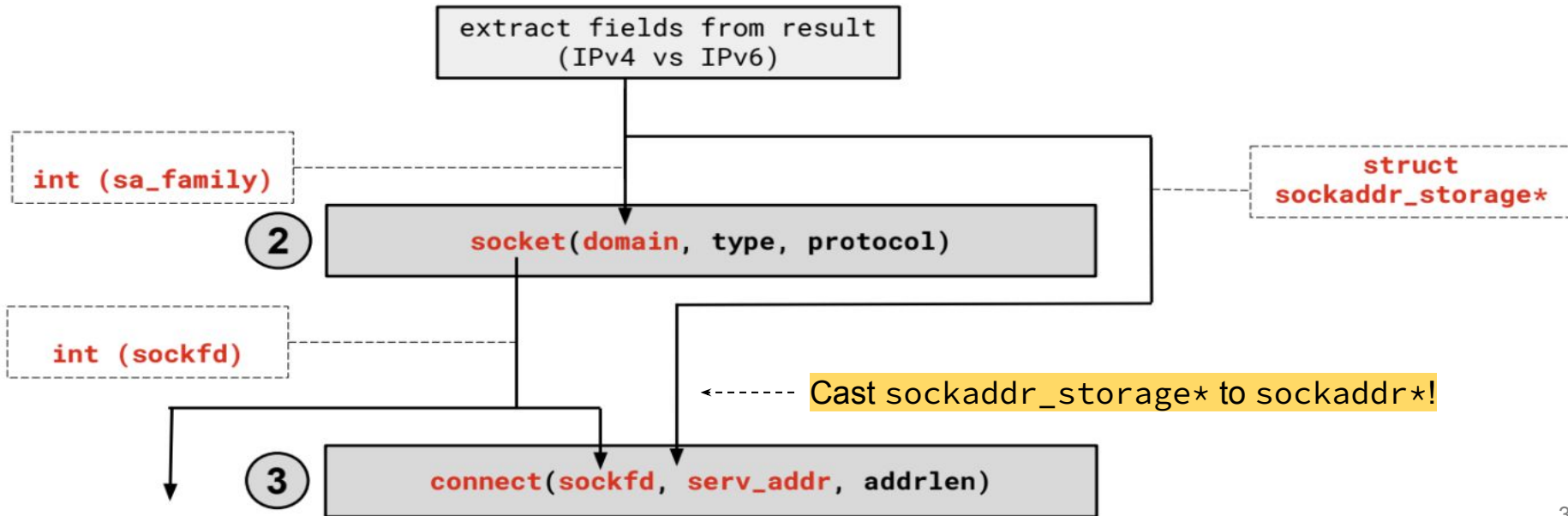
```
int connect(int socket,           // socket fd
            const struct sockaddr *addr, // address to connect to
            socklen_t addr_len);      // length of *addr
```

- 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()

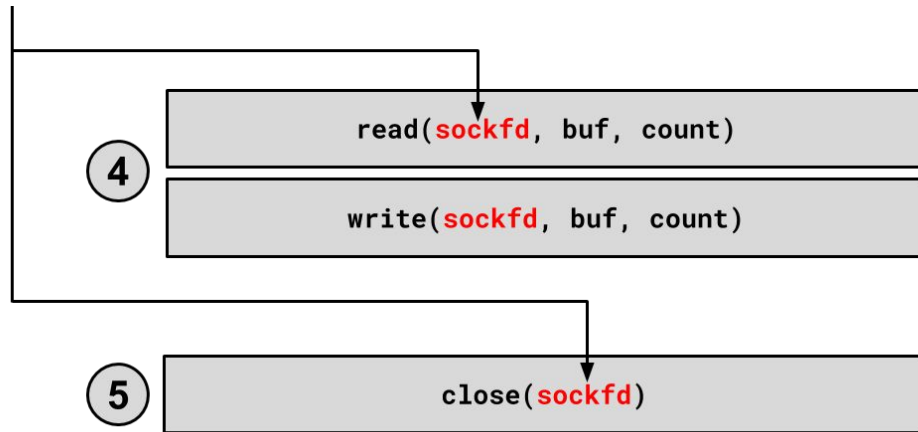
```
int connect(int socket, // from 1
            const struct sockaddr *addr, // from 2
            socklen_t addr_len); // size of serv_addr
```

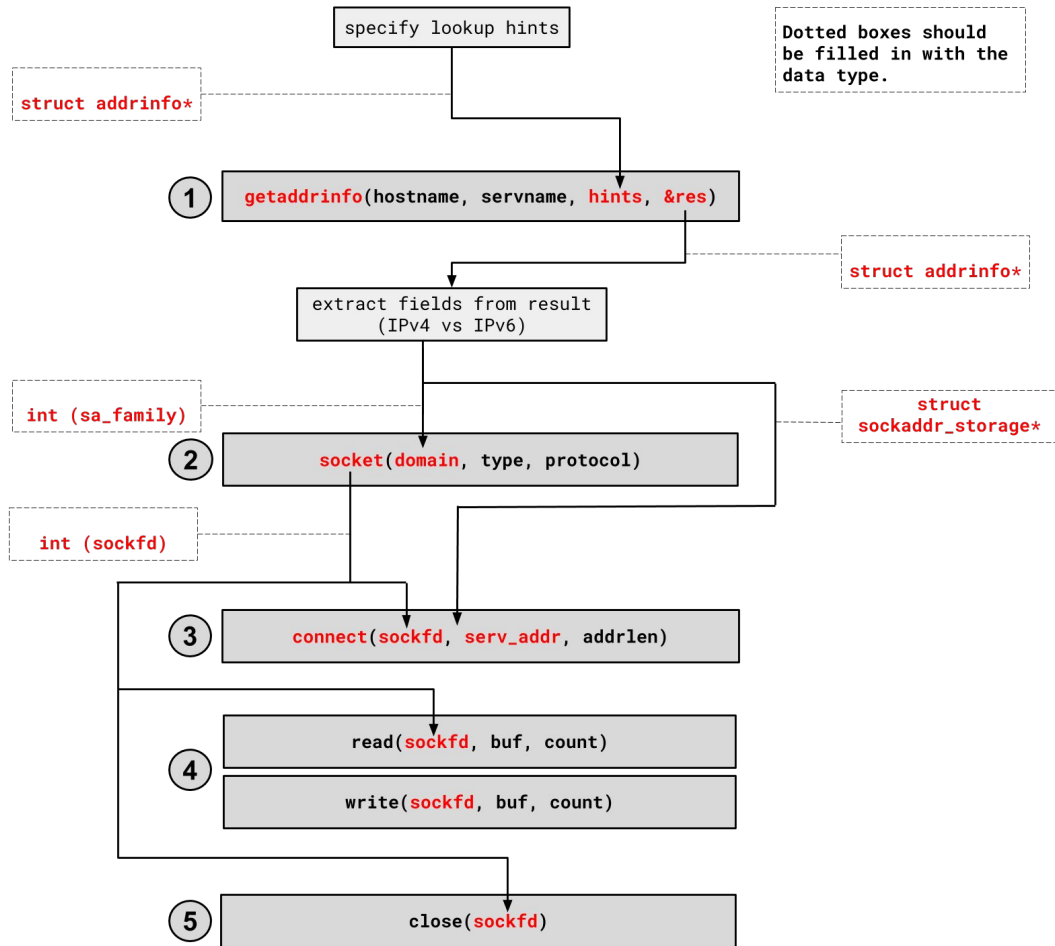
- Connects an available socket to a specified address
- Returns 0 on success, -1 on failure



4. read/write and 5. close

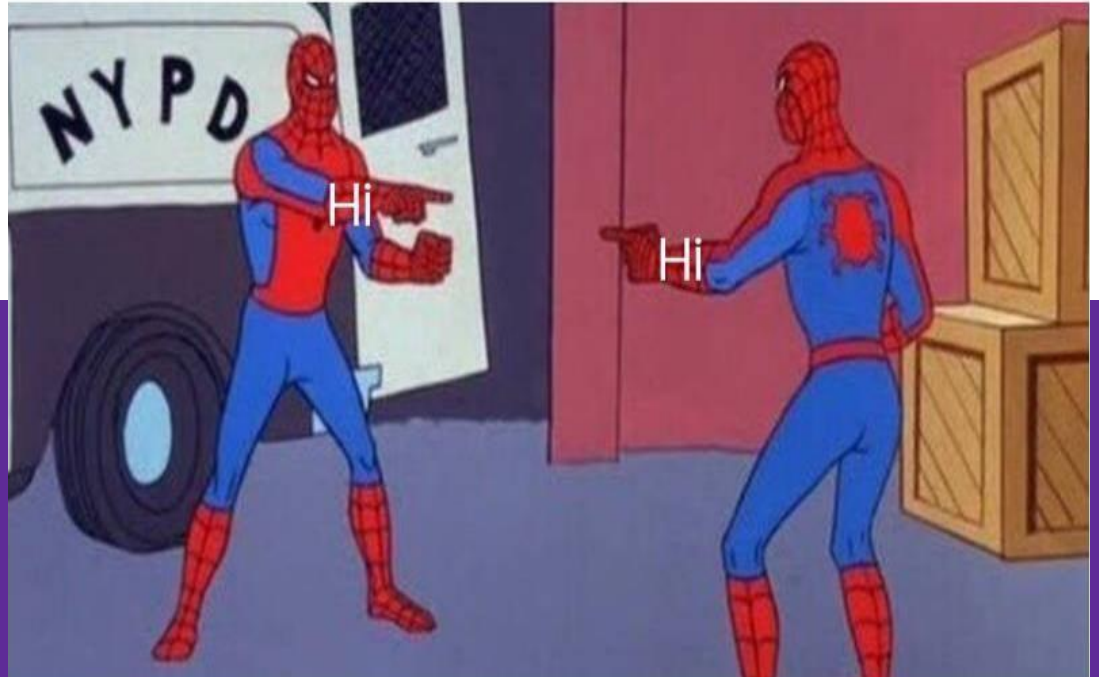
- Thanks to the file descriptor abstraction, use as normal!
- read from and write to a buffer, the OS will take care of sending/receiving data across the network
- Make sure to close the fd afterward





Using Netcat for the first time

Netcat and Ex10-11 demo



netcat

- Command-line utility to setup a TCP/UDP connection to read/write data
 - Man page: <https://www.commandlinux.com/man-page/man1/nc.1.html>
- To start a server:
 - `nc -l <hostname> <port>`
- To connect to that server (as a client):
 - `nc <hostname> <port>`
- `<hostname>` can be:
 - `localhost`
 - `attu#.cs.washington.edu`

Exercise Overviews and Demo

- Ex10: build a client that can send bytes to a server
 - Send the contents of a file over the network
 - Test with netcat server
- Ex11: build a server that listens for incoming client connections
 - Prints out the received data/file contents
 - Test with Ex10 (your own or sample solution) or netcat client
- File comparison (need to make sure that input and output files match exactly bitwise!)
 - Redirect server output to `output.bytes`
 - If both files on the same machine, use: `diff -s file1 file2`
 - If files are on different machines, manually compare md5sum outputs