CSE 333 Section 8

Client-side Networking & netcat

UNIVERSITY of WASHINGTON

Logistics

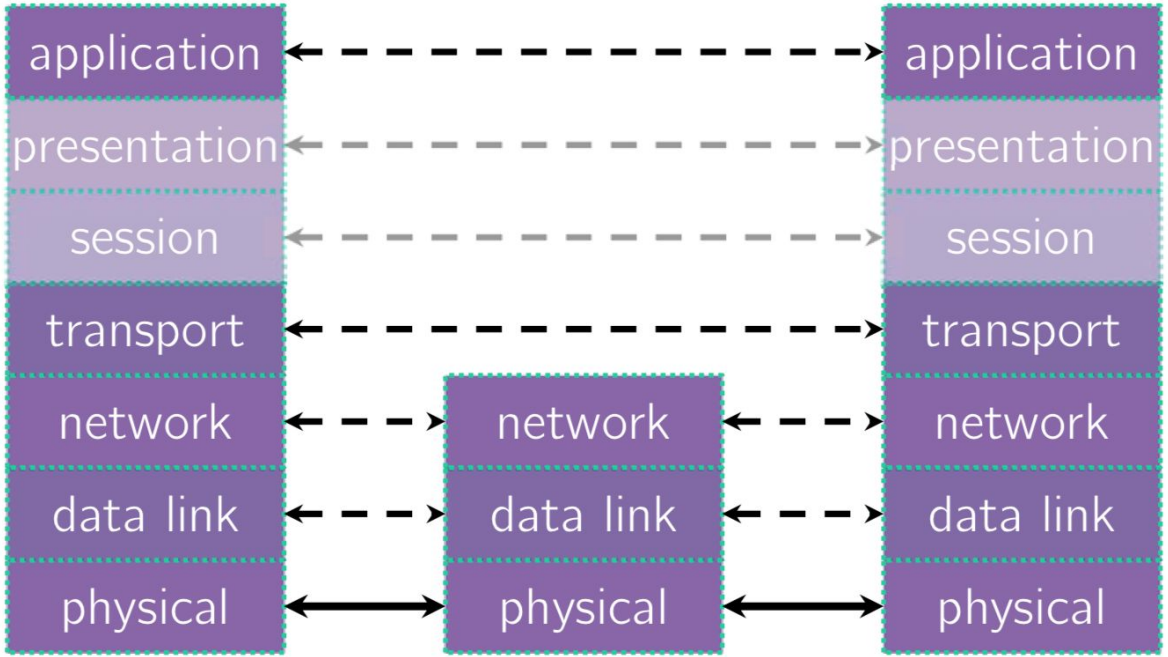
Homework 3 due today @ 11:59 pm

Released tomorrow:

- Exercise 10: client side networking @ due Wed 3/2 11 am
- Exercise 11: server side networking @ due Fri 3/4 11 am

Networking - At a High Level

Computer Networks: A 7-ish Layer Cake



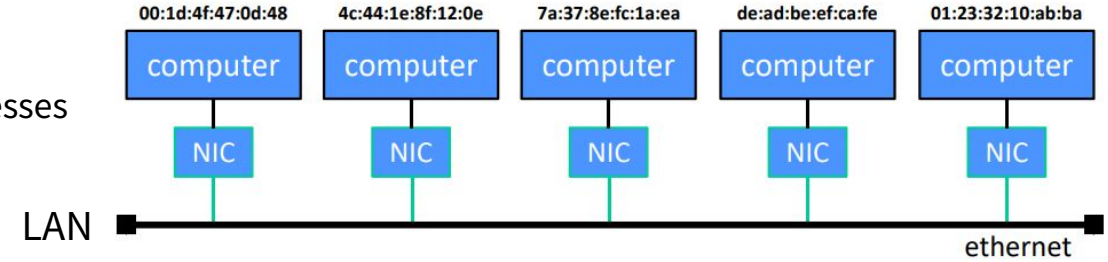
Computer Networks: A 7-ish Layer Cake

Wires, radio signals, fiber optics

bit encoding at signal level

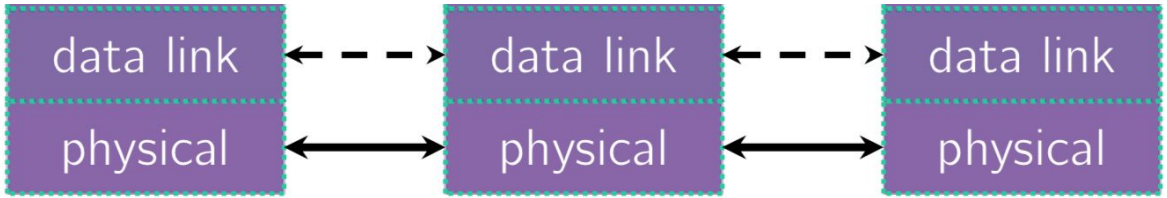
Computer Networks: A 7-ish Layer Cake

WIFI, ethernet. Has to do with MAC addresses

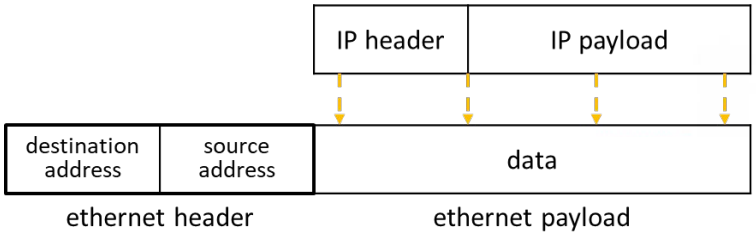


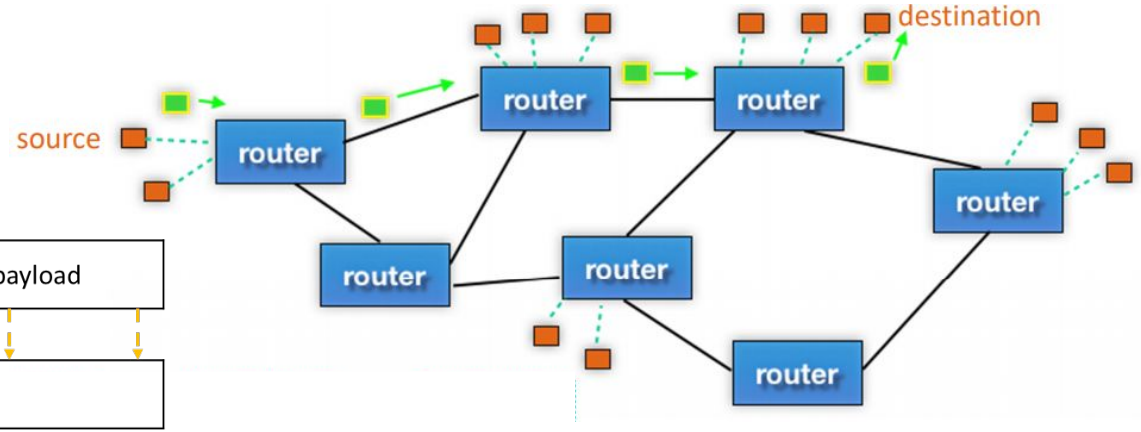
multiple computers on a local network

bit encoding at signal level



Computer Networks: A 7-ish Layer Cake

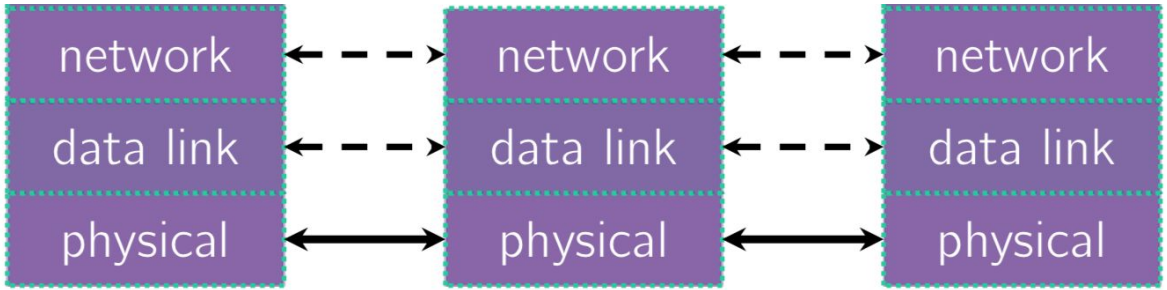




routing of packets across networks

multiple computers on a local network

bit encoding at signal level



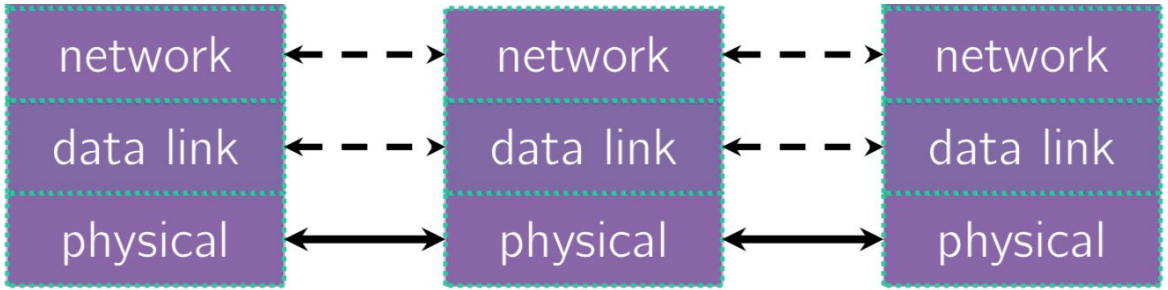
Computer Networks: A 7-ish Layer Cake

Abstraction/Interface

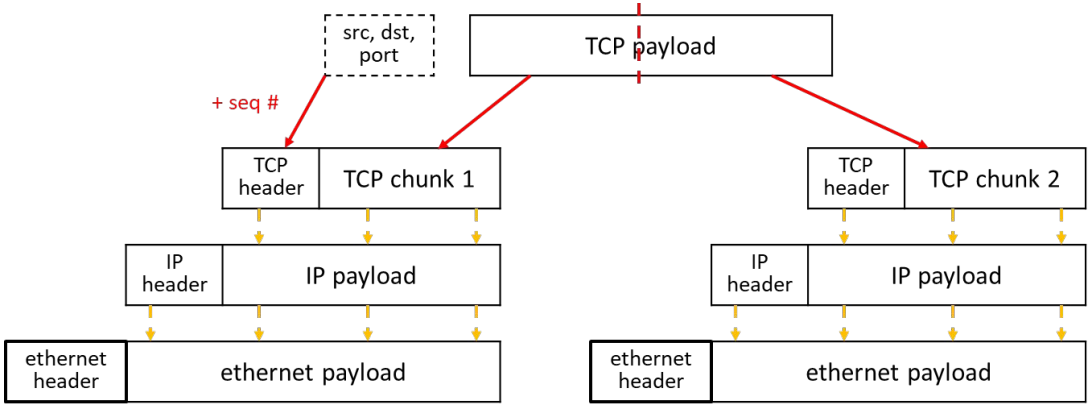
routing of packets across networks

multiple computers on a local network

bit encoding at signal level



Computer Networks: A 7-ish Layer Cake



TCP, UDP.

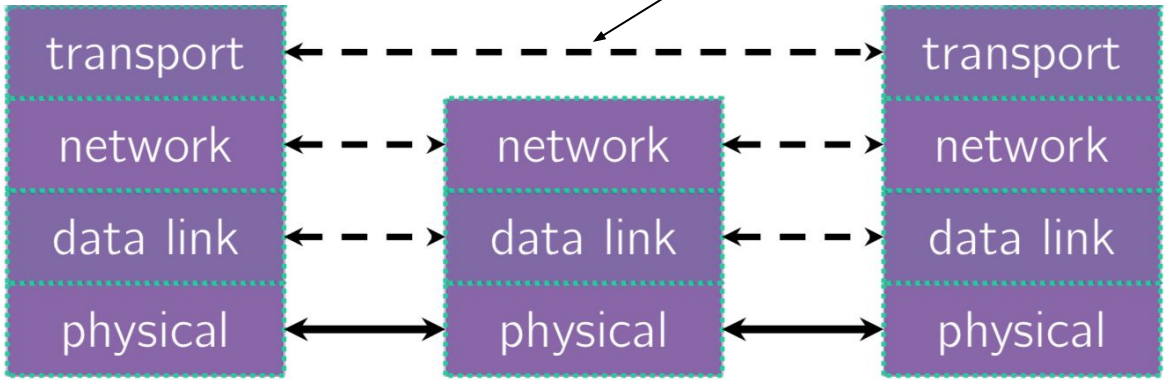
Stream abstraction!

sending data end-to-end

routing of packets across networks

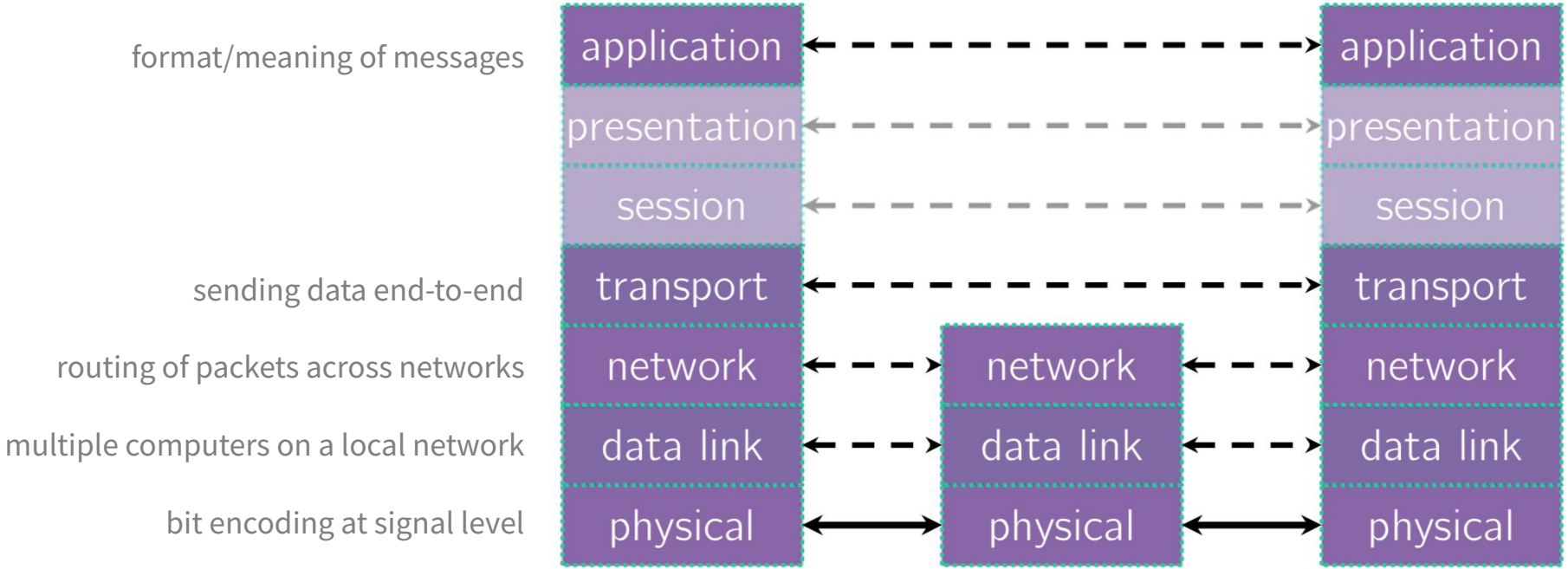
multiple computers on a local network

bit encoding at signal level

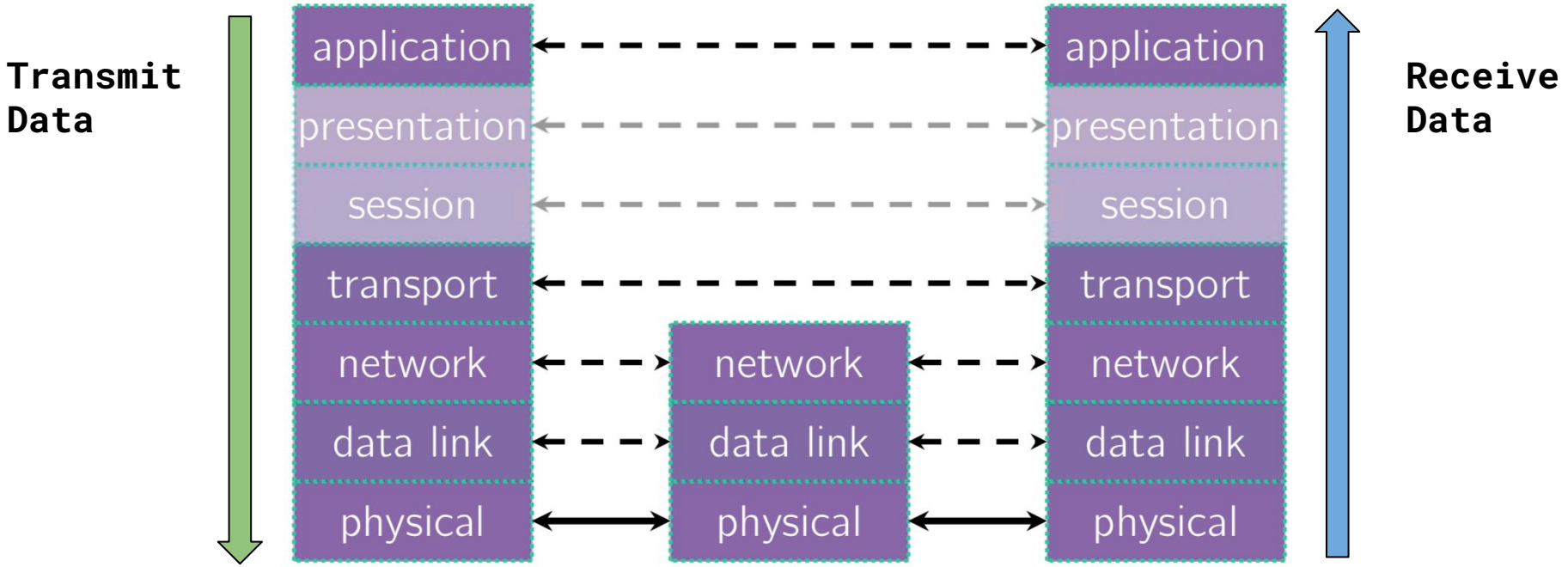


Computer Networks: A 7-ish Layer Cake

HTTP, DNS, anything else?

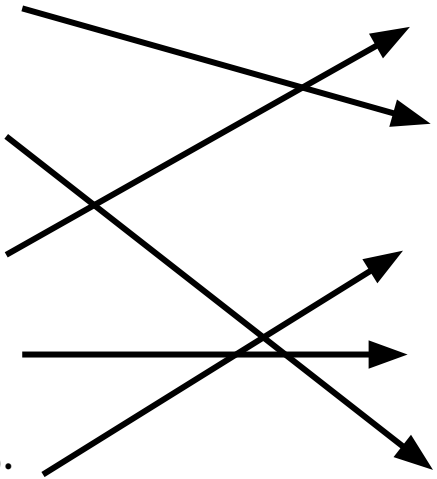


Data flow


Transmit Data
application
presentation
session
transport
network
data link
physical
application
presentation
session
transport
network
data link
physical
network
data link
physical
Receive Data

Terminology Review

- DNS:
- IP:
- TCP:
- UDP:
- HTTP:



Reliable transport protocol on top of IP.

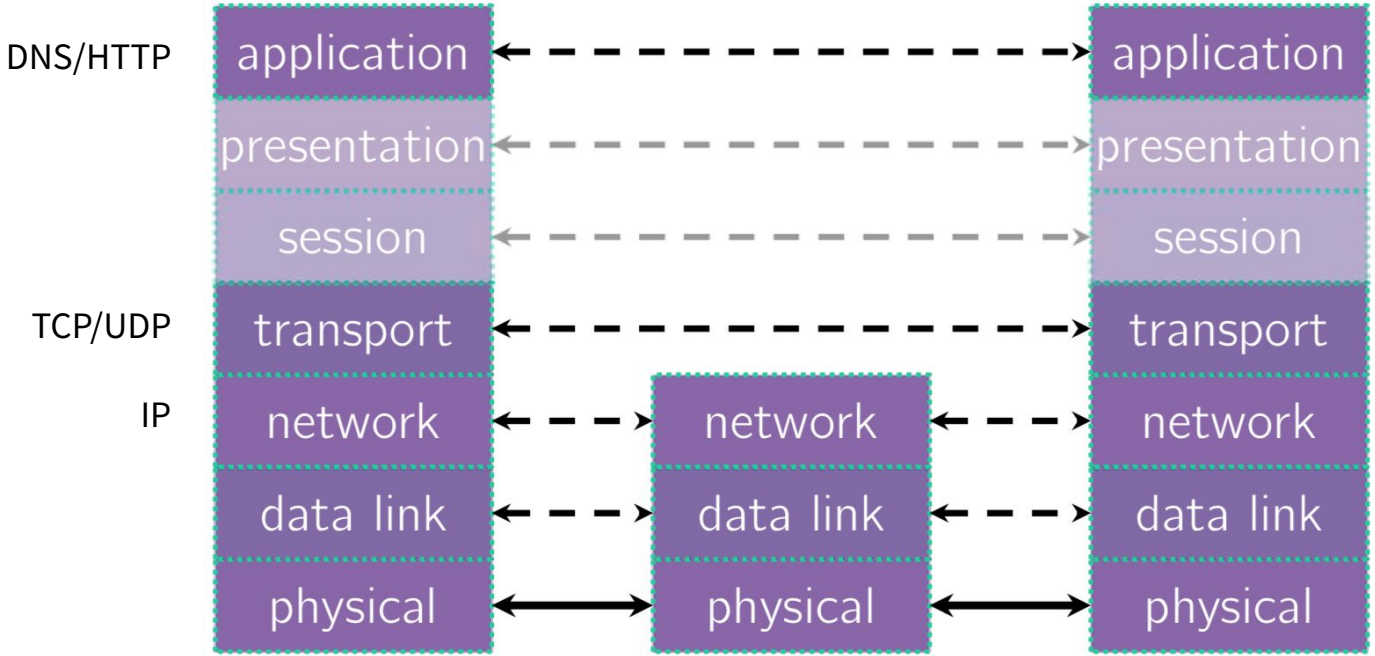
Translating between IP addresses and host names.

Sending websites and data over the Internet.

Unreliable transport protocol on top of IP.

Routing packets across the Internet.

Exercise 1


DNS/HTTP
application
presentation
session
TCP/UDP
transport
IP
network
data link
physical
network
data link
physical
application
presentation
session
transport
network
data link
physical

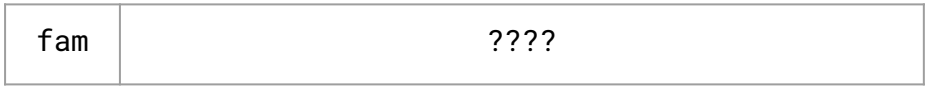
Client-Side Networking

Sockets

- Just a file descriptor for network communication
  - Where else have we used file descriptors? Foreshadowing perhaps?
- 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**
  - Both port and address are stored in network byte order (big endian)

Sockets

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



....

struct sockaddr_in (IPv4)

16

struct sockaddr_in6 (IPv6)

28

struct sockaddr_storage

Big enough to hold either

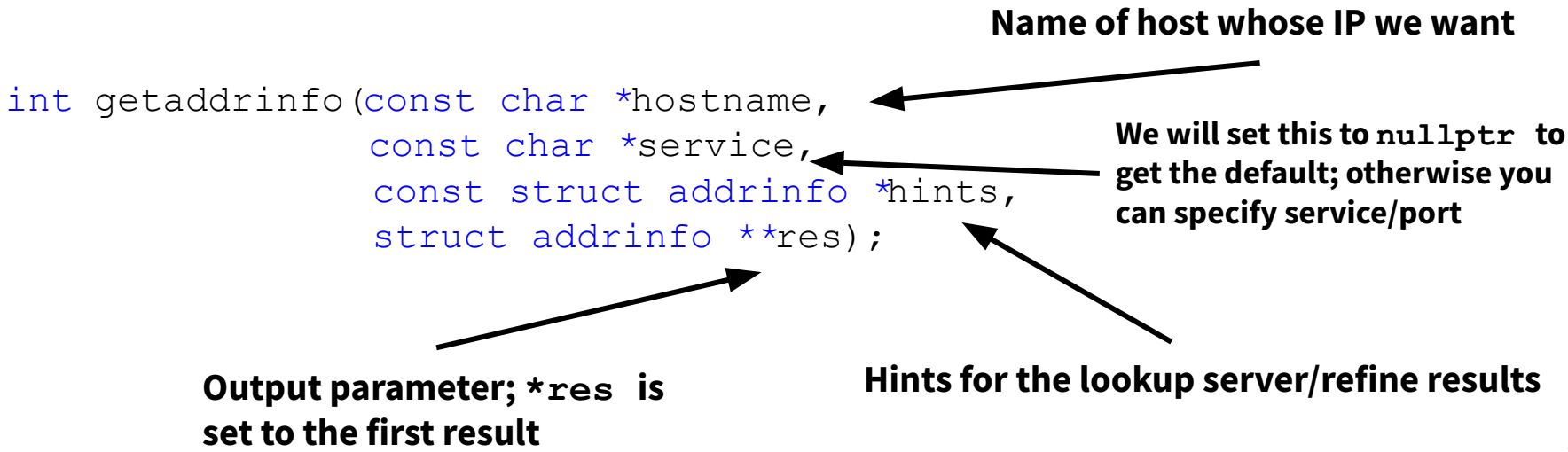
Byte Ordering and Endianness

- **N**etwork Byte Order (Big Endian)
- **H**ost byte order - Might be big or little endian, depending on the hardware
- To convert between orderings, we can use

```
uint16_t htons(uint16_t hostshort);
// htons -> host to network short
uint16_t ntohs(uint16_t netshort);
// ntohs -> network to host short
uint32_t htonl(uint32_t hostlong);
// htonl -> host to network long
uint32_t ntohl(uint32_t netlong);
// ntohl -> network to host long
```

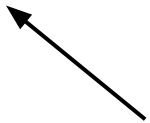
getaddrinfo()

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



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

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

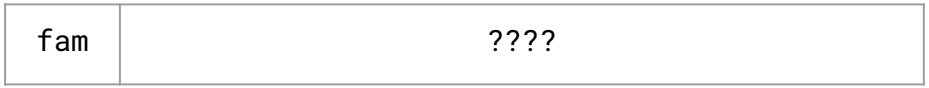
getaddrinfo() - Interpreting Results

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

- A `struct sockaddr*` can point to *either* a `struct sockaddr_in` or a `struct sockaddr_in6`
  - What does this remind us of?
- All of the `struct sockaddr_*` structs have a field called `family` that lets us figure out what kind of address it is at runtime
- We can pass either a `struct sockaddr_in` or a `struct sockaddr_in6` to system calls as needed

1. getaddrinfo() - Interpreting Results

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



....

struct sockaddr_in (IPv4)

16

struct sockaddr_in6 (IPv6)

28

struct sockaddr_storage

Big enough to hold either

2. 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 a normal file descriptor as well as in socket specific system calls)

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!

netcat