# CSE 333
# Section 7

Client-side Networking

UNIVERSITY of WASHINGTON

# Logistics

Due Today:

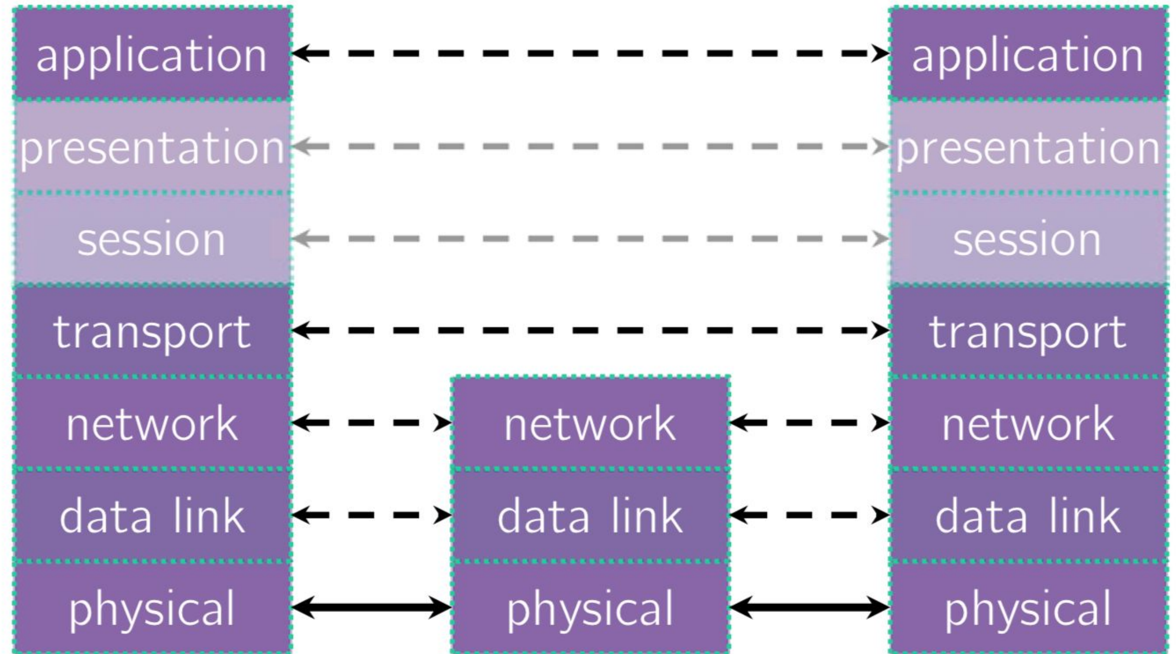
Homework 3 @ 11:59 pm

Due Wednesday:

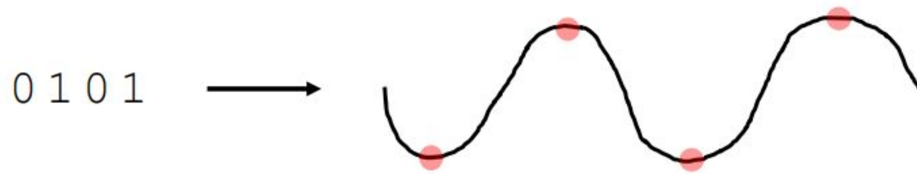
Exercise 10 @ 11 am

# Networking - At a High Level

# Computer Networks: A 7-ish Layer Cake

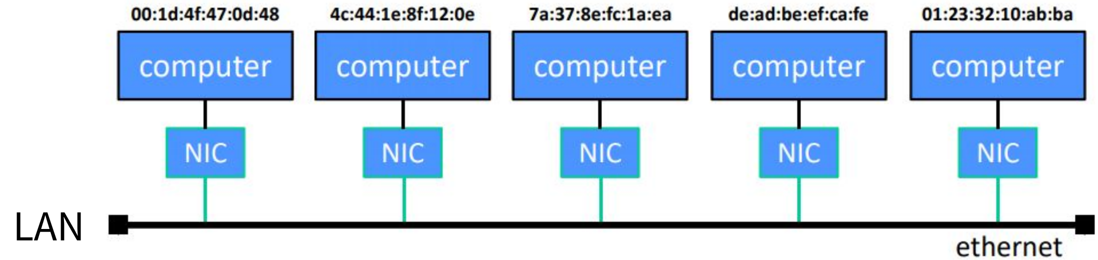


# Computer Networks: A 7-ish Layer Cake



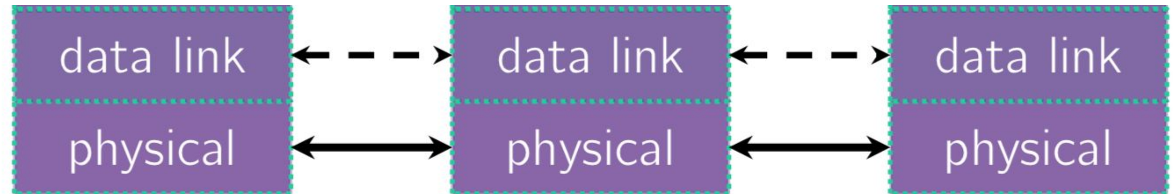

bit encoding at signal level

# Computer Networks: A 7-ish Layer Cake

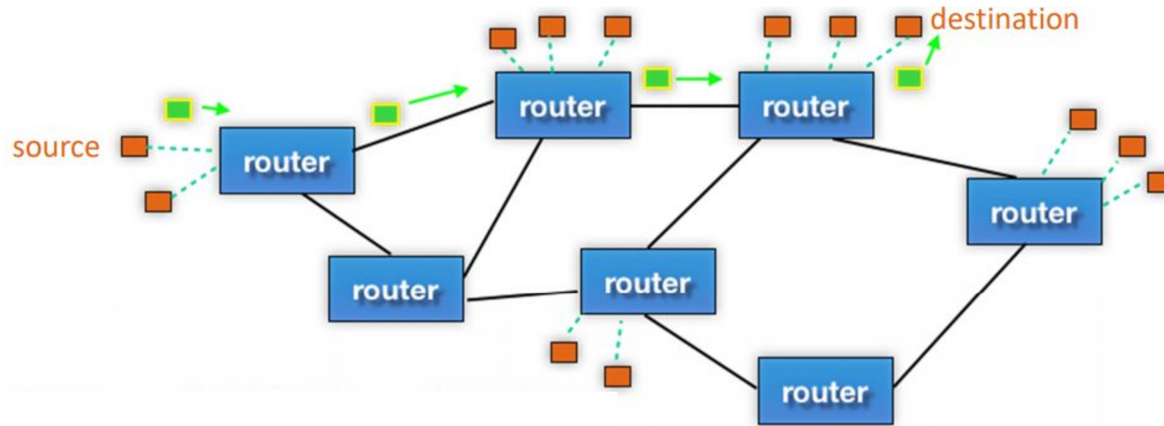



multiple computers on a local network

bit encoding at signal level



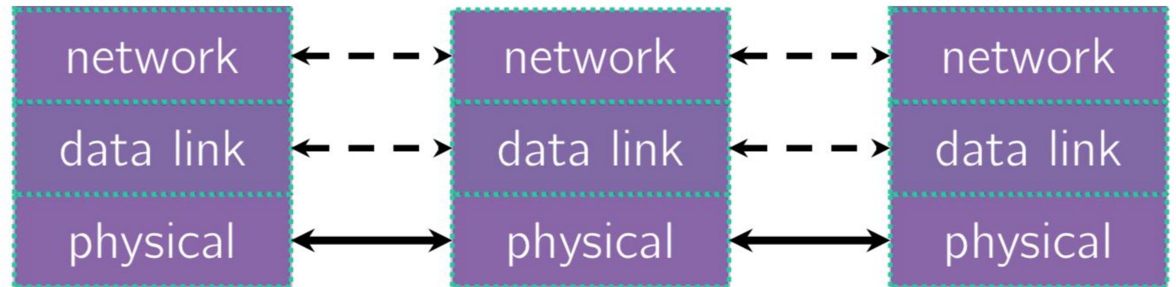
# Computer Networks: A 7-ish Layer Cake



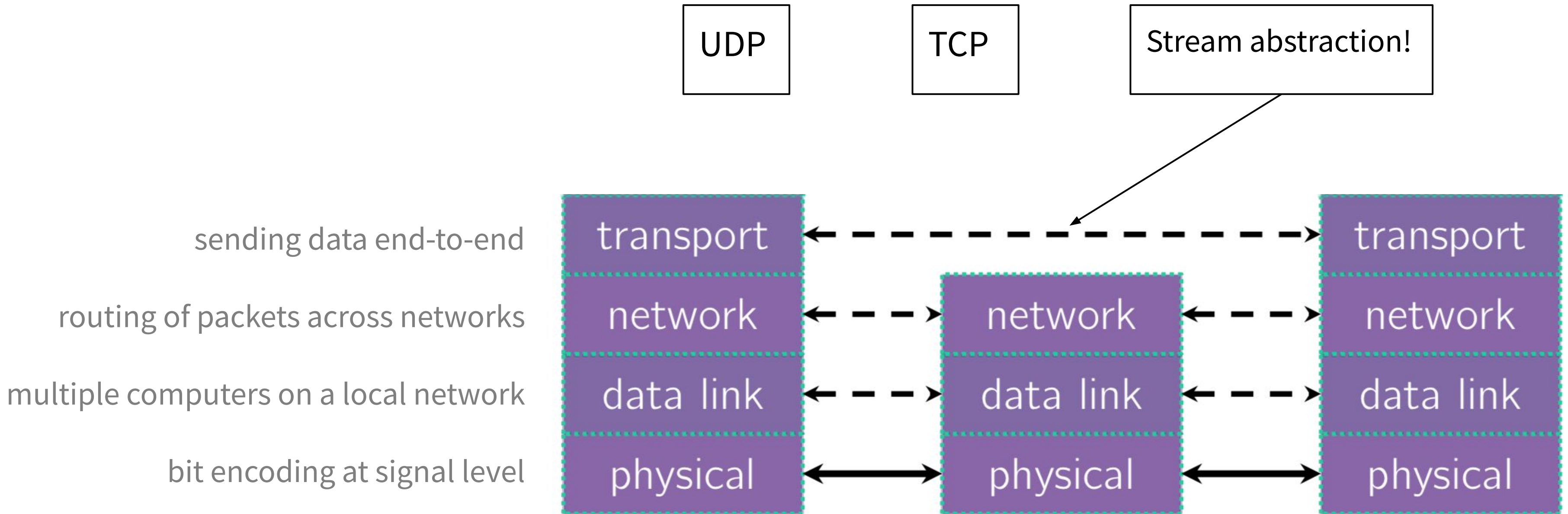

routing of packets across networks

multiple computers on a local network

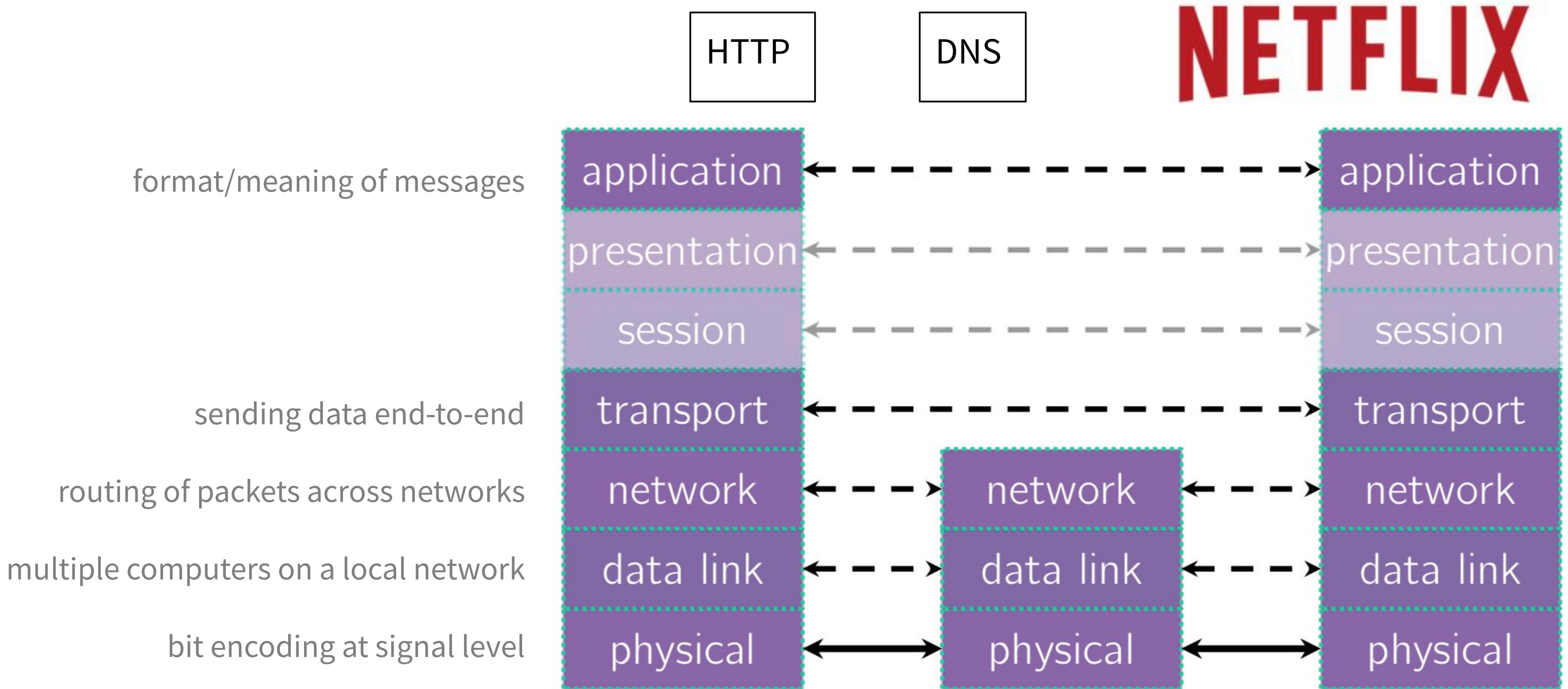
bit encoding at signal level



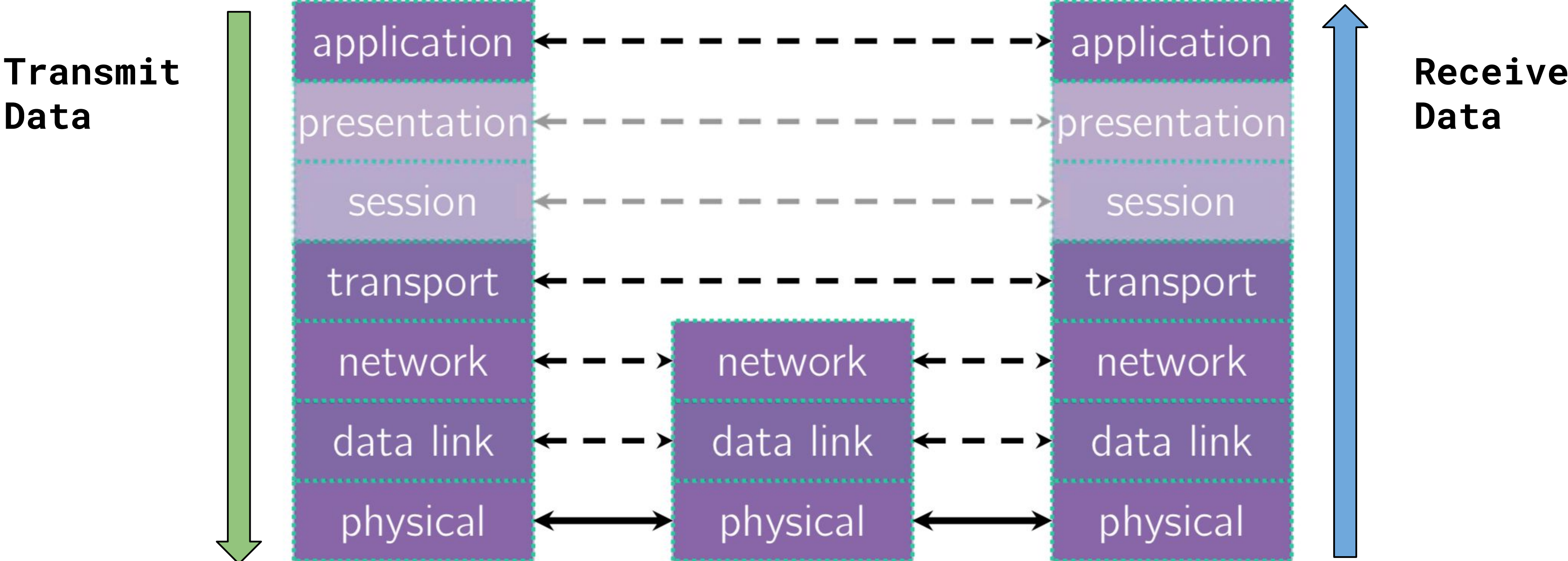
# Computer Networks: A 7-ish Layer Cake



# Computer Networks: A 7-ish Layer Cake



# Data flow

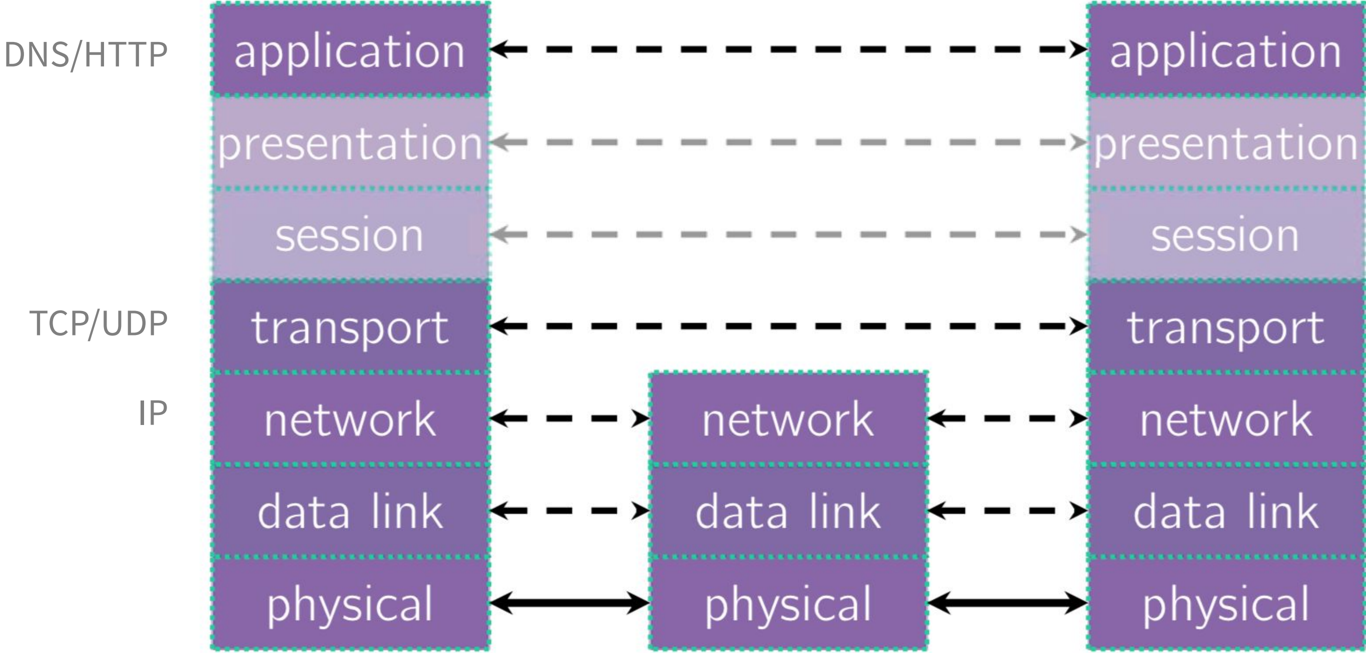


# Exercise 1

# Exercise 1

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

# Exercise 1



# TCP versus UDP

**Transmission Control Protocol (TCP)**

- Connection oriented Service
- Reliable and Ordered
- Flow control

Potential applications:

- Websites
- Email

**User Datagram Protocol (UDP)**

- Connectionless service
- Unreliable packet delivery
- Faster (usually lower latency)
- No feedback

Potential applications:

- VOIP
- Multiplayer gaming

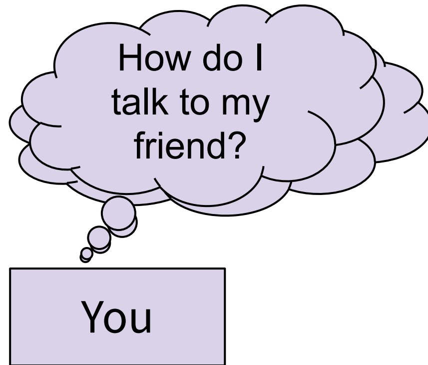
# Client-side Networking Preview

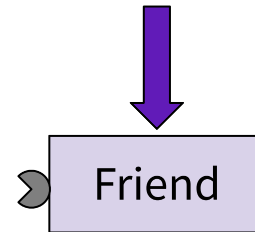
Let’s say you want to talk to your friend, but you’ve both lost your phones so you’ll have to do something a little… old fashioned

You know your friend is waiting somewhere for you to contact them. How can you do that?

You know your friend is waiting somewhere for you to contact them. How can you do that?

1. Figure out their current address



## You know your friend is waiting somewhere for you to contact them. How can you do that?

1. Figure out their current address
2. Get a cup

# You know your friend is waiting somewhere for you to contact them. How can you do that?

1. Figure out their current address
2. Get a cup
3. Connect a string between your two cups

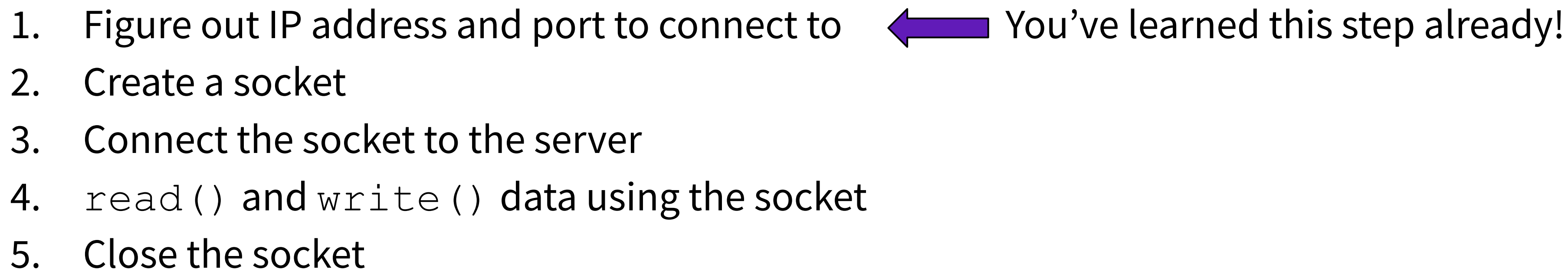
# You know your friend is waiting somewhere for you to contact them. How can you do that?

1. Figure out their current address
2. Get a cup
3. Connect a string between your two cups
4. Talk!

# You know your friend is waiting somewhere for you to contact them. How can you do that?

1. Figure out their current address
2. Get a cup
3. Connect a string between your two cups
4. Talk!
5. Clean up your cup and string

# Client-side Networking Steps



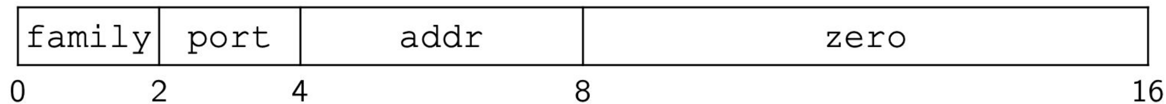

More details in tomorrow's lecture!

# Sockets and DNS Lookup

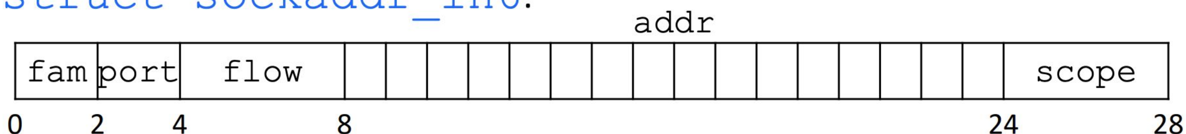
# Sockets

- Just a file descriptor for network communication
- Types of Sockets
  - Stream sockets (TCP)
  - Datagram sockets (UDP)
- Each socket is associated with **a port number** and **an IP address**
  - Both port and address are stored in network byte order (big endian)

`struct sockaddr_in`:

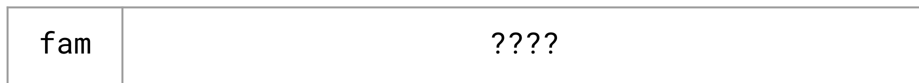



`struct sockaddr_in6`:



# 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)
  - The most significant byte is stored in the highest address
- **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 hostlong);`
  - `uint16_t ntohs (uint16_t hostlong);`

  - `uint32_t htonl (uint32_t hostlong);`
  - `uint32_t ntohl (uint32_t hostlong);`

# getaddrinfo()

```
// Figure out what IP address and port to talk to
// returns 0 on success, negative number on failure
int getaddrinfo(const char *hostname,    // hostname to lookup
        const char *servname,            // service name
        const struct addrinfo *hints,    // desired output (optional)
        struct addrinfo **res);          // results structure


// Frees memory allocated by getaddrinfo()
void freeaddrinfo(struct addrinfo *ai);
```

# getaddrinfo()

- 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);
```

# getaddrinfo() - Interpreting Results

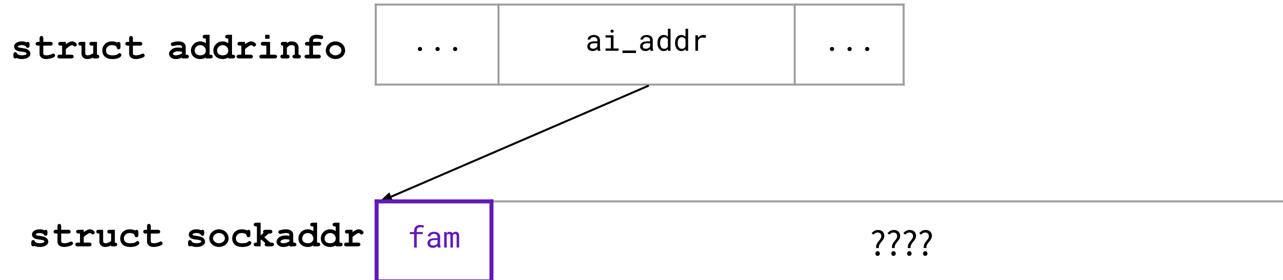
```
struct addrinfo {
    int ai_flags; // additional flags
    int ai_family; // AF_INET, AF_INET6, AF_UNSPEC
    int ai_socktype; // SOCK_STREAM, SOCK_DGRAM, 0
    int ai_protocol; // IPPROTO_TCP, IPPROTO_UDP, 0
    size_t ai_addrlen; // length of socket addr in bytes
    struct sockaddr* ai_addr;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

- `ai_addr` points to a `struct sockaddr` describing the socket address

# getaddrinfo() - Interpreting Results

With a `struct sockaddr*`:

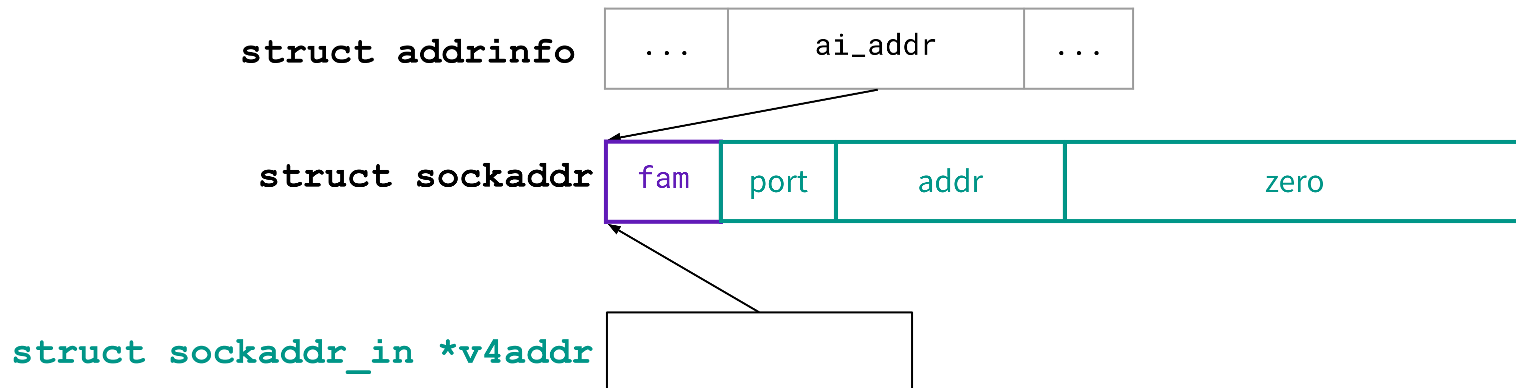
- The field `sa_family` describes if it is IPv4 or IPv6



# getaddrinfo() - Interpreting Results

With a `struct sockaddr*`:

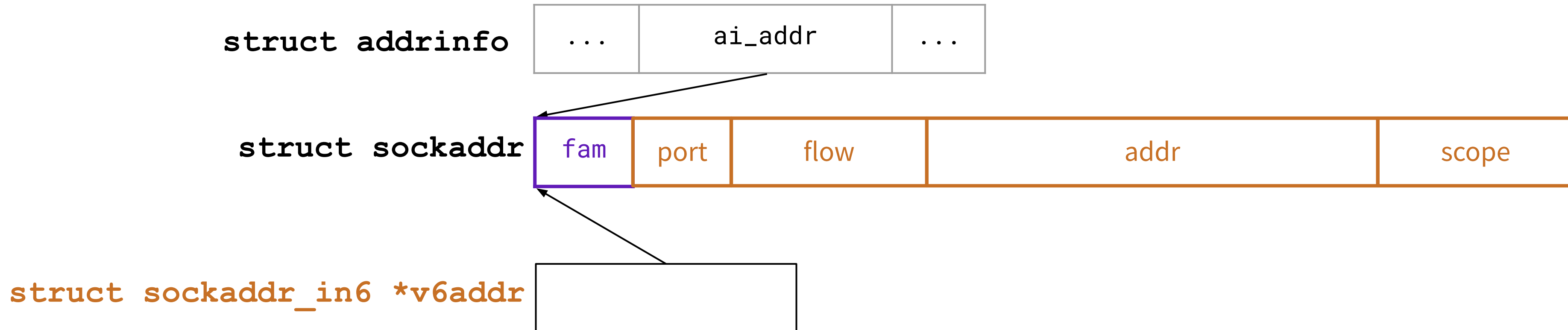
- The field `sa_family` describes if it is IPv4 or IPv6
- Cast to `struct sockaddr_in*` (v4) or `struct sockaddr_in6*` (v6) to access/modify specific fields



# 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



# 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

**`struct sockaddr_storage`**

# POSIX Review

# read()

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

What might a return value of 0 for `read()` mean in the context of networking?

If `read()` returns 0 when reading from a socket this indicates that the network connection was closed. Note this is different from a failure occurring!

Where will `int fd` come from when dealing with the network?

When reading (or writing) across the network we use a socket to abstract away the networking details. This socket is identified by an integer value that we can use with POSIX functions.

# write()

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

What values of errno are recoverable?

`EINTR` and `EAGAIN`

Can `write()` return 0?

In general `write()` will not return 0 but it is possible in a few cases. For networking, a return value of less than `count` such as 0 can be an indicator that the connection was closed.

# POSIX Exercise

```
int main(int argc, char **argv) {
  int socket_fd;
  char readbuf[512];
  int res;
  ... // Assume code to set up the network connection is included
  // Read data from the server until the connection is closed
  while (_________________________________ != 0) {
    if (res == -1) {
      if(______________________________) {continue;}
      ____________________
      return EXIT_FAILURE;
    }
    // Write the data we read to stdout
    int pos = 0;
    while (res > 0) {
      int write_res = _______________________________________
      if (write_res == -1) {
        if (_______________________________) {continue;}
        ____________________
        return EXIT_FAILURE;
      }
      res -= write_res;
      pos += write_res;
  }
  ____________________
  return EXIT_SUCCESS;
}
```

```
int main(int argc, char **argv) {
  int socket_fd;
  char readbuf[512];
  int res;
  ... // Assume code to set up the network connection is included
  // Read data from the server until the connection is closed
  while ((res = read(socket_fd, readbuf, 512))  != 0) {
    if (res == -1) {
      if(errno == EINTR || errno == EAGAIN ) {continue;}
      close(socket_fd);
      return EXIT_FAILURE;
    }
    // Write the data we read to stdout
    int pos = 0;
    while (res > 0) {
      int write_res =  write(STDOUT_FILENO, readbuf + pos, res);
      if (write_res == -1) {
        if (errno == EINTR || errno == EAGAIN ) {continue;}
          close(socket_fd);
          return EXIT_FAILURE;
      }
      res -= write_res;
      pos += write_res;
  }
  close(socket_fd)
  return EXIT_SUCCESS;
}
```

# Exercise 10 Demo

# Steps for Testing

Set up server:

- `$nc -l <port num> > <output file name>`
- I.e., `$nc -l 2048 > output.bytes`

Connect to server:

- `$./ex10 <server name> <port num> <local file>`
- I.e., `$./ex10 attu7 2048 test.txt`

# Testing Notes

- Pick your favorite 4-digit number for the port - don’t use the one from the spec as you may run into conflicts if someone else is using it!
- Have to start server over each time you test
- Can use `localhost` as the server name if running on same machine (i.e., attu7)