Week 7 - Monday

# COMP 3400

# Last time

- What did we talk about last time?
- Socket programming
- IPv4 and IPv6 addressing

# Questions?

# Assignment 4

# Project 2

# Back to Sockets

# Getting addresses from a host name

- DNS converts a host name to an IP address
- The **`getaddrinfo()`** function lets us get a linked list of matching addresses

```
int getaddrinfo (const char *name, const char *service,
const struct addrinfo *hints, struct addrinfo **results)
```

- The only annoying bit is that we have to fill out a hints structure
- A utility function **`freeaddrinfo()`** is provided to free the linked list structure when done with it

```
void freeaddrinfo (struct addrinfo *info);
```

# The addrinfo struct

- The result of **`getaddrinfo()`** is stored into the pointer given by the last argument

```
struct addrinfo {
  int ai_flags;
  int ai_family;
  int ai_socktype;
  int ai_protocol;
  socklen_t ai_addrlen;
  char *ai_canonname;
  struct sockaddr *ai_addr;   // Pointer to address we need
  struct addrinfo *ai_next;   // Pointer to next addrinfo in linked list
};
```

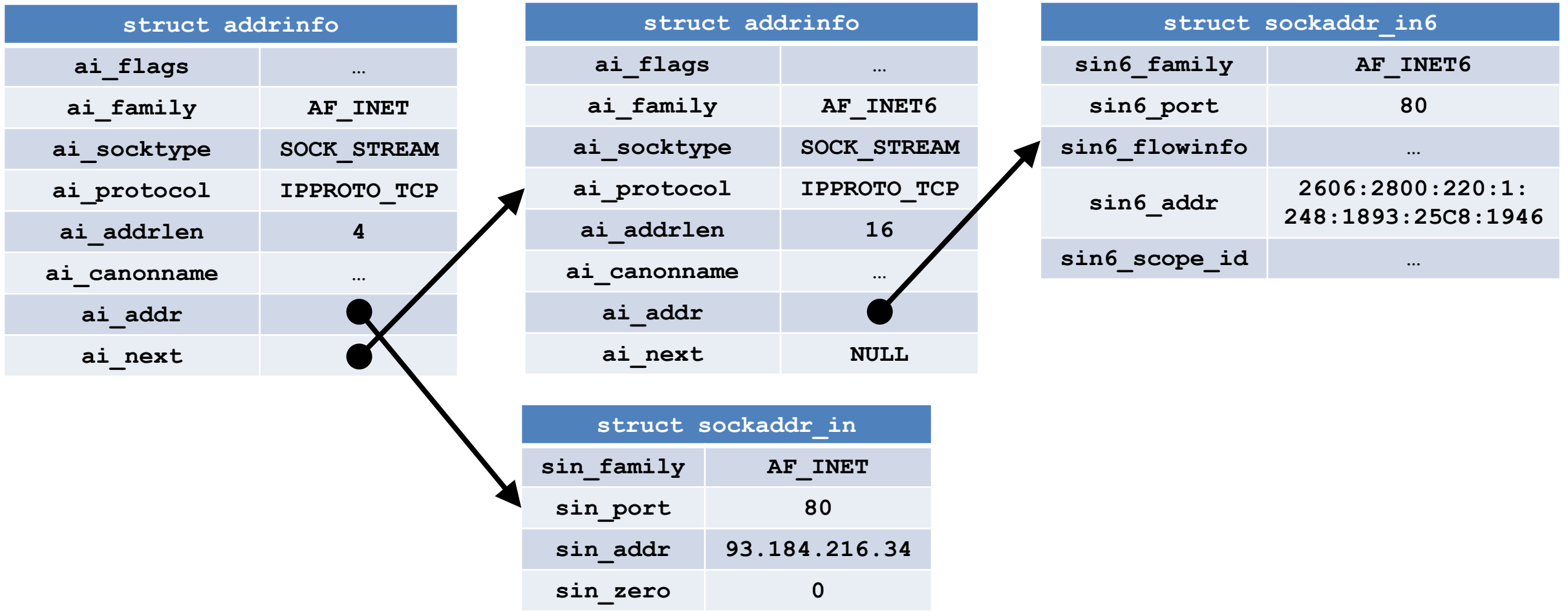
# Getting address example

```
struct addrinfo hints, *server_list = NULL, *server = NULL;
memset (&hints, 0, sizeof (hints));
hints.ai_family = AF_INET;        // IPv4
hints.ai_socktype = SOCK_STREAM; // Byte-streams (TCP)
hints.ai_protocol = IPPROTO_TCP; // TCP
assert (getaddrinfo (hostname, "http", &hints, &server_list) == 0); // Get addresses

for (server = server_list; server != NULL; server = server->ai_next)
 {
    if (server->ai_family == AF_INET) // Only take IPv4
      {
        // Cast to IPv4 socket
        struct sockaddr_in *addr = (struct sockaddr_in *)server->ai_addr;
        printf ("IPv4 address: %s\n", inet_ntoa (addr->sin_addr));
      }
  }
freeaddrinfo (server_list);
```

# Confusing structs!

- Here's a visualization of the **addrinfo** and **sockaddr** structs that might come back from **getaddrinfo()**



# Programming practice

- Adapt the code on the previous slide:
  - Read a host or IP address from the user
  - Read a service or port name from the user
  - Print out the resulting IP addresses

Note the following common port names and services:

| Port | Name | Service |
|---|---|---|
| 21 | FTP | Insecure file transfer |
| 22 | SSH | Secure shell |
| 23 | Telnet | Insecure remote access |
| 25 | SMTP | Email delivery |
| 53 | DNS | IP address lookup |
| 67 | DHCP | IP address assignment |
| 68 | DHCP | IP address assignment |
| 80 | HTTP | Web page |
| 88 | Kerberos | Authentication |

| Port | Name | Service |
|---|---|---|
| 110 | POP3 | POP email access |
| 123 | NTP | Time synchronization |
| 143 | IMAP | IMAP email access |
| 194 | IRC | Internet chat service |
| 389 | LDAP | Authentication |
| 443 | HTTPS | Secure web page |
| 530 | RPC | Remote procedure call |
| 631 | IPP | Internet printing |
| 993 | IMAPS | Secure IMAP access |

# Client side: connecting

- After all the madness is done getting the **`sockaddr`**, a client can connect to a listening server with the **`connect()`** function

```
int connect (int socket, const struct sockaddr *address, socklen_t address_len);
```

- The **`connect()`** function is a blocking call that will eventually succeed or fail to connect the socket file descriptor to an actual network connection
- If successful, we can read and write from that file descriptor

# Server side: options

- The server side is more complicated
- It's useful to set some options on the socket using the (confusing) **`setsockopt()`** function

```
int setsockopt (int socket, int level, int option, const void *value, socklen_t lengeth);
```

  - Reusing the port, allowing reuse of the same port, even after crashing
  - Timing out on read messages
- After creating the socket:

```
//Allow port reuse
int on = 1;
setsockopt (socketfd, SOL_SOCKET, SO_REUSEADDR, (const void *) &on, sizeof (int));
// Set a 5-second timeout when waiting to receive
struct timeval timeout = { 5, 0 };
setsockopt (socketfd, SOL_SOCKET, SO_RCVTIMEO, (const void *) &timeout,
            sizeof (timeout));
```

# Server side: binding and listening

- After creating the server socket (and maybe setting options), the next step is to bind the server to a port

```
int bind (int socket, const struct sockaddr *address, socklen_t address_len);
```

- For UDP, the server is then ready to receive messages
- For TCP, it has to listen on the socket

```
int listen (int socket, int backlog);
```

  - The backlog gives how many clients can queue up when trying to connect to the server

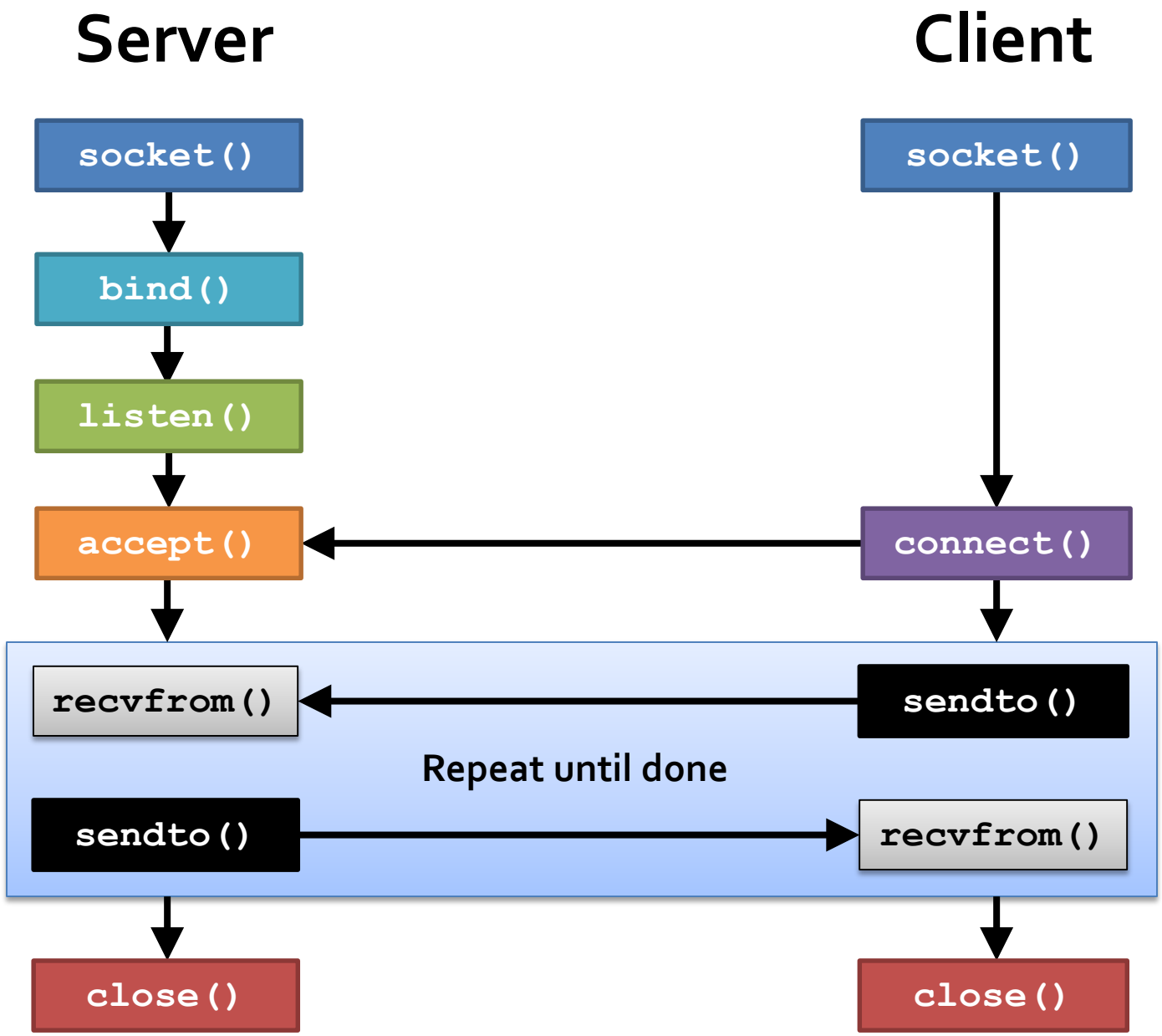
# Server side: accepting

- For TCP connections, after listening, the server can call **`accept()`**

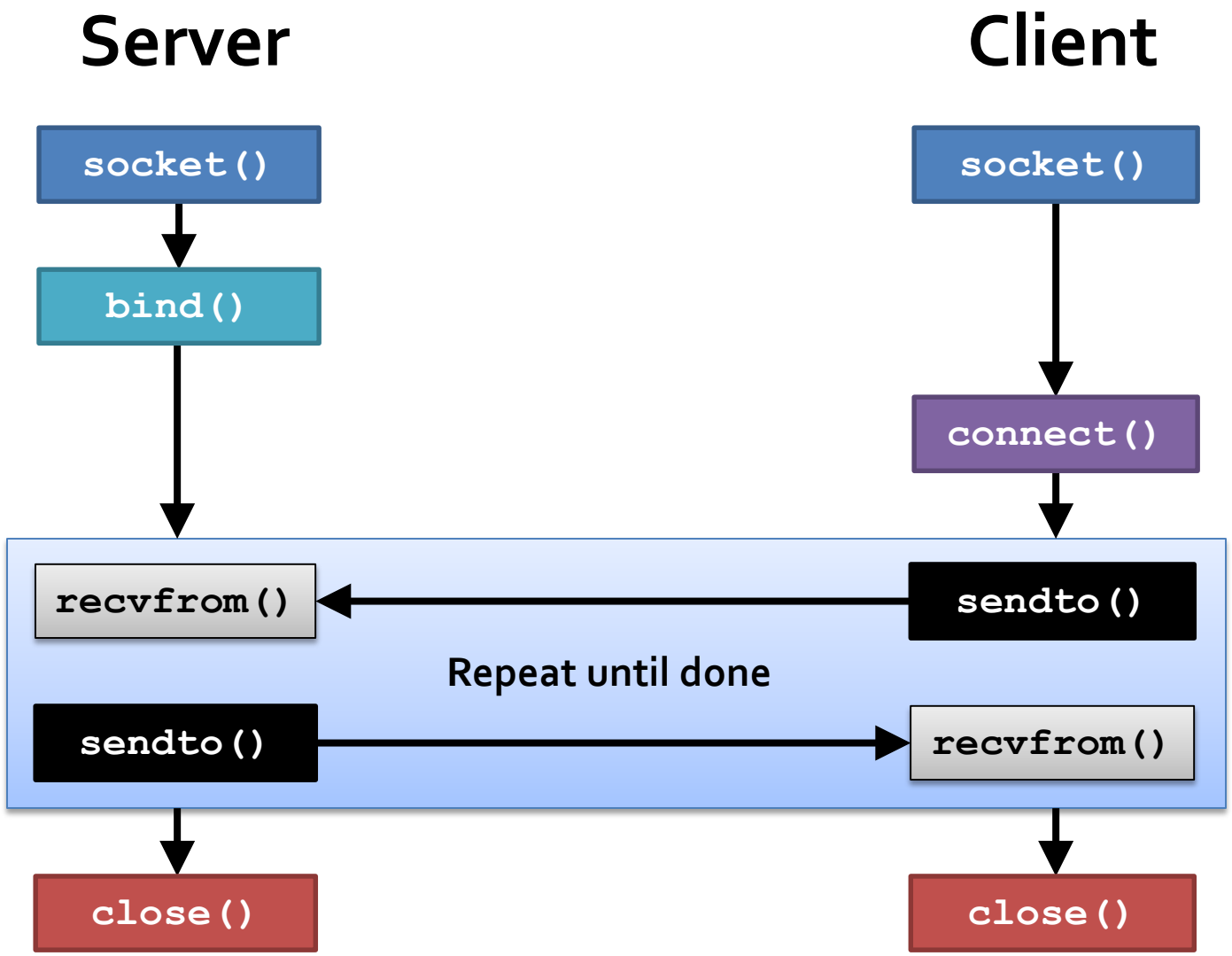
```
int accept (int socket, struct sockaddr *address, socklen_t *address_len);
```

  - Blocking function
  - Will wait until a client tries to connect
  - Then, messages can be sent and received
  - Doing so sets up a TCP session, expecting a series of packets from the connecting client

# TCP Communication



# UDP Communication



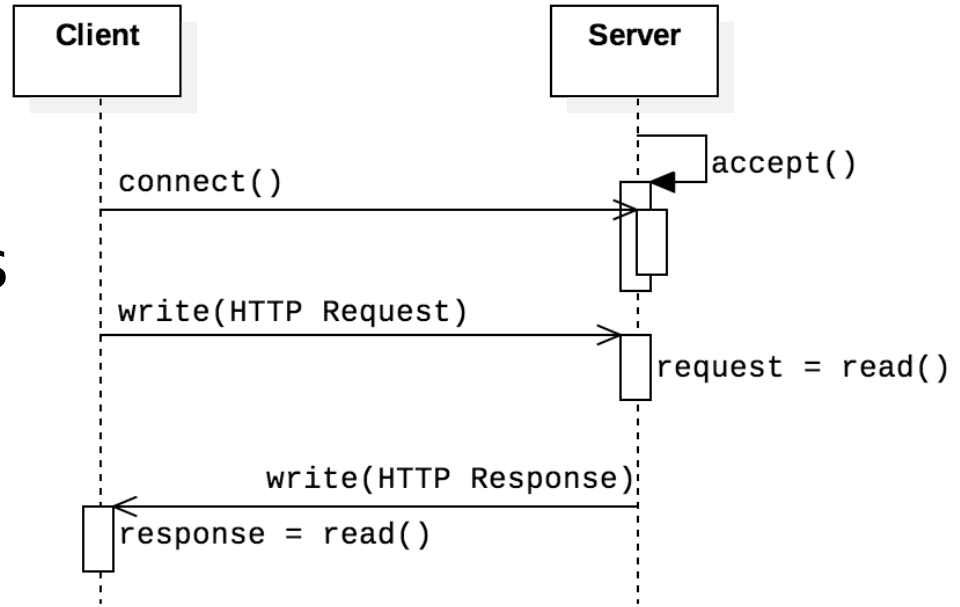
# TCP Socket Programming

# TCP communication

- The biggest differences between single-machine and networked IPC:
  - Networked IPC typically employs **protocols** so that machines agree on how data should be formatted
  - Networked IPC is less reliable
- It's hard to talk about TCP communication without examples that use some particular application layer protocol
- We're going to use HTTP because:
  - It's easy to understand
  - It's really important
  - There are lots of servers in the world we can talk to without any credentials

# HTTP

- **Hypertext Transfer Protocol** (HTTP) is the protocol for (non-encrypted) web page communication
- It's a request-response protocol
  - Shown in the sequence diagram on the right
- HTTP itself is stateless: no information is preserved between requests
- Other features built around HTTP (cookies, server-side scripting, and databases) overcome this stateless limitation



# Sample request

- HTTP requests and responses start with header lines
  - Each ends with CRLF (**\r\n**), with an extra CRLF after all headers
  - Each **\r\n** would simply look like a newline, but we show them below for clarity
- The most common client request is GET
- It must have a line like the following:

```
GET /path HTTP/version\r\n
```

  - **path** is the file being requested
  - **version** is the HTTP version, usually 1.0, 1.1, or 2

```
GET /index.html HTTP/1.0\r\n
Accept: text/html\r\n
Accept-Encoding: gzip, deflate, br\r\n
Accept-Language: en-US,en;q=0.5\r\n
User-Agent: Mozilla/5.0\r\n
\r\n
```

# Upcoming

# Next time…

- Finish TCP socket programming
- UDP socket programming

# Reminders

- Finish Assignment 4
  - Due tonight by midnight!
- Start on Project 2!
- Read section 4.5 and 4.6