# Socket Programming (Cont)

## Computer Systems Programming, Spring 2025

**Instructor:** Travis McGaha

**Teaching Assistants**:

Andrew Lukashchuk

Angie Cao

Aniket Ghorpade

Ashwin Alaparthi

Austin Lin

Hassan Rizwan

Lobi Zhao

Pearl Liu

Perrie Quek

# Poll Everywhere

**pollev.com/tqm**

- What questions do you have about sockets?

# Administrivia

- Final Project Details Coming soon-ish
  - Done in pairs
  - Pair signup is due @midnight tomorrow. Random pairs released on Wednesday
  - Details to be posted today
  - SOME of it is auto graded. There is a lot of functionality that is not autograded that you will need to implement
  - Demo in a little bit ☺
- No more HW assignments other than the project and catching up on old assignments!
- TA Application is out! I highly recommend it ☺

# Lecture Outline

- **Final Project Demo**
- Client-Side Socket Programming (Wrap-up)
- Server-Side Socket Programming

# Project demo

- ./searchserver 5950 ./test_tree
  - Run giving a port and a directory containing files to search over
  - “results found” doesn’t show up until you actually do a search
    - Results in order
  - Multi word queries
  - Can click link to open the file
    - Why /static/ in the links?
  - Inspect page to look at HTML (We will give you some sample HTML and HTTP so you know what it looks like)
- Can take a long time to run, so you can run on smaller subdirectories of the test_tree:
  - ./searchserver 5950 ./test_tree/tiny
  - ./searchserver 5950 ./test_tree/books

# Poll Everywhere

pollev.com/tqm

- What questions do you have about the project?

# Lecture Outline

- Final Project Demo
- **Client-Side Socket Programming (Wrap-up)**
- Server-Side Socket Programming

# Socket API: Client TCP Connection

- We'll start by looking at the API from the point of view of a client connecting to a server over TCP

- There are five steps:

** Today **

New stuff

1) Figure out the IP address and port to which to connect
2) Create a socket
3) Connect the socket to the remote server

Same as file I/O

4) `read()` and `write()` data using the socket
5) Close the socket

# Step 1: Figure Out IP Address and Port

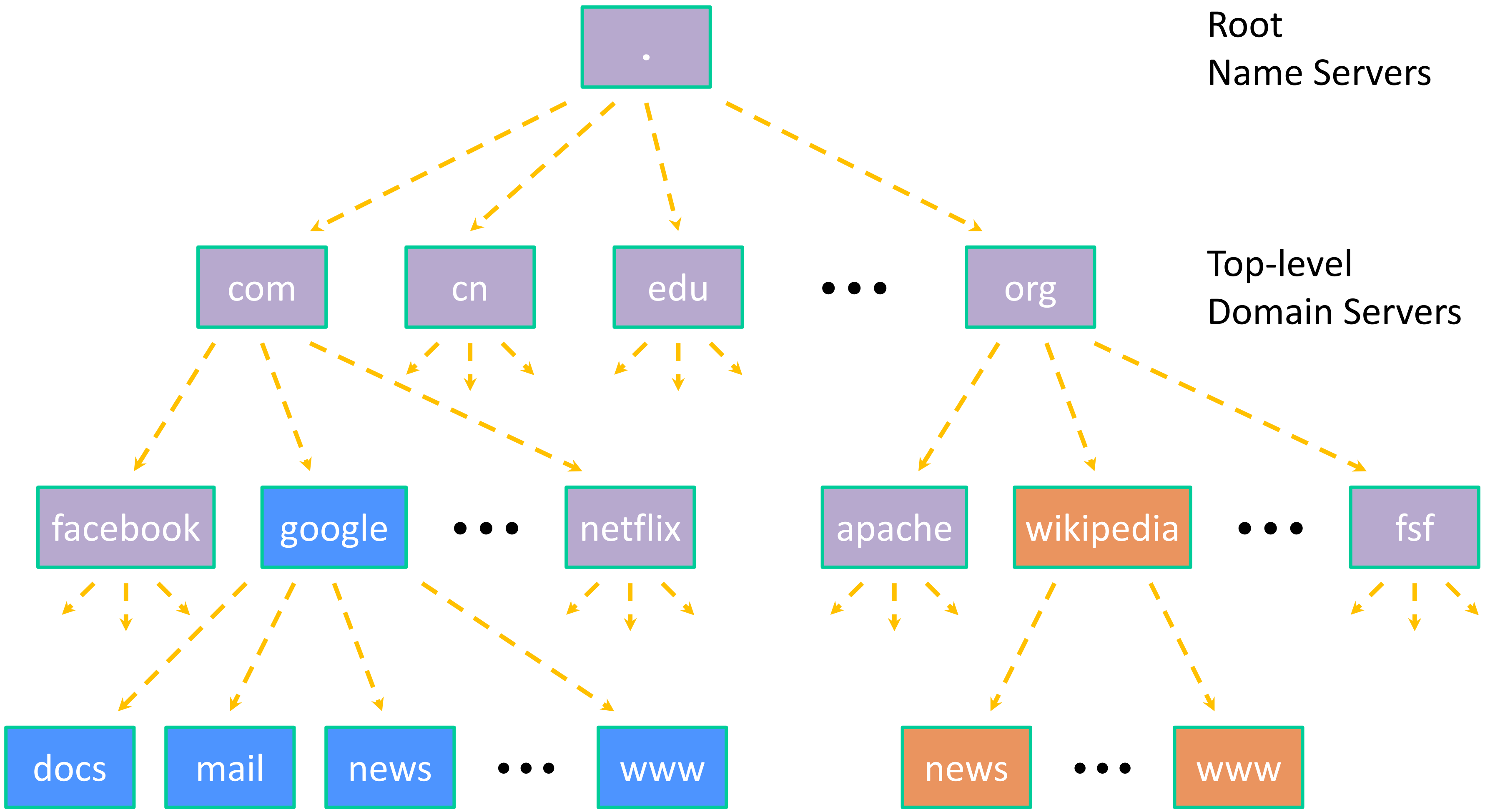
- Several parts:
  - Network addresses
  - Data structures for address info C data structures ☹
  - DNS (Domain Name System) – finding IP addresses

# Domain Name System

- People tend to use DNS names, not IP addresses
  - The Sockets API lets you convert between the two
  - It's a complicated process, though:
    - A given DNS name can have many IP addresses
    - Many different IP addresses can map to the same DNS name
      - An IP address will reverse map into at most one DNS name
    - A DNS lookup may require interacting with many DNS servers

- You can use the Linux program "`dig`" to explore DNS
  - `dig @server name type (+short)`
    - `server`: specific name server to query
    - `type`: `A` (IPv4), `AAAA` (IPv6), `ANY` (includes all types)

# DNS Hierarchy



# Resolving DNS Names

- The POSIX way is to use **getaddrinfo**()
  - A complicated system call found in #include <netdb.h>
  - Basic idea:

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

Output param

- Tell **getaddrinfo**() which host and port you want resolved
  - String representation for host: DNS name or IP address
- Set up a "hints" structure with constraints you want respected
- **getaddrinfo**() gives you a list of results packed into an "addrinfo" structure/linked list
  - Returns 0 on success; returns *negative number* on failure
- Free the struct addrinfo later using **freeaddrinfo**()

# getaddrinfo

- getaddrinfo() arguments:
  - hostname – domain name or IP address string
  - service – port # (*e.g.* "80") or service name (*e.g.* "www") or NULL/nullptr
  - Hints Parameter

```
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
};
```

Can use 0 or nullptr to indicate you don't want to filter results on that characteristic

# DNS Lookup Procedure

```
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
};
```

1) Create a `struct addrinfo hints`
2) Zero out `hints` for “defaults”
3) Set specific fields of `hints` as desired
4) Call **`getaddrinfo`**`()` using `&hints`
5) Resulting linked list `res` will have all fields appropriately set

- See dnsresolve.cpp

# Socket API: Client TCP Connection

- There are five steps:
  1) Figure out the IP address and port to connect to
  2) Create a socket
  3) Connect the socket to the remote server
  4) `read()` and `write()` data using the socket
  5) Close the socket

# Step 2: Creating a Socket

- `int socket(int domain, int type, int protocol);`
  - Creating a socket doesn't bind it to a local address or port yet
  - Returns file descriptor or -1 on error

socket.cpp

```
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <iostream>

int main(int argc, char** argv) {
  int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
  if (socket_fd == -1) { // check for error
     std::cerr << strerror(errno) << std::endl;
     return EXIT_FAILURE;
  }
  close(socket_fd); // clean up
  return EXIT_SUCCESS;
}
```

# Step 3: Connect to the Server

- The `connect()` system call establishes a connection to a remote host result from `socket()`

  -
    ```
    int connect(int sockfd, const struct sockaddr* addr,
                socklen_t addrlen);
    ```
    result from `getaddrinfo()`
    - `sockfd`: Socket file description from Step 2
    - `addr` and `addrlen`: Usually from one of the address structures returned by `getaddrinfo` in Step 1 (DNS lookup)
    - Returns `0` on success and `-1` on error
- `connect()` may take some time to return
  - It is a *blocking* call by default Waits on an event before returning
  - The network stack within the OS will communicate with the remote host to establish a TCP connection to it Performs a "Handshake" with the server
    - This involves ~2 *round trips* across the network

# Connect Example

- See connect.cpp

```
// Get an appropriate sockaddr structure.
struct sockaddr_storage addr;
size_t addrlen;
LookupName(argv[1], port, &addr, &addrlen);// Helper function that calls
                                           // getaddrinfo()

// Create the socket.
int socket_fd = socket(addr.ss_family, SOCK_STREAM, 0);
if (socket_fd == -1) {
  cerr << "socket() failed: " << strerror(errno) << endl;
  return EXIT_FAILURE;
}

// Connect the socket to the remote host.
int res = connect(socket_fd,
                  reinterpret_cast<sockaddr*>(&addr),
                  addrlen);
if (res == -1) {
  cerr << "connect() failed: " << strerror(errno) << endl;
}
```

# Sockets are sort of like files

- From this point it just turns into
  - Read/write
  - Close

- Looks like a file right?

- But this isn’t a file, it’s a network connection. It just looks like one
  - File
  - Terminal Input/Output
  - Pipe
  - Network Connection  (More similar to reading/writing terminal or pipe than a file)

# Sockets are sort of like files

- When dealing with stream sockets (TCP) Sockets, the TCP part is done for us. We can deal with the stream ABSTRACTION
  - Stream: That the bytes show up in order reliably

**pollev.com/tqm**

- How do you think a network connection may behave differently from a file?
  - If it helps you can compare a file to reading/writing into a book and reading/writing a socket to texting/messaging a friend.

# Step 4: `read()`

- If there is data that has already been received by the network stack, then read will return immediately with it
  - `read()` might return with *less* data than you asked for

- If there is no data waiting for you, by default `read()` will *block* until something arrives

  **pollev.com/tqm**

  - How might this cause *deadlock*?
  - Can `read()` return `0`? (EOF)

# Step 4: `write()`

- `write()` queues your data in a send buffer in the OS and then returns
  - The OS transmits the data over the network in the background
  - When `write()` returns, the receiver probably has not yet received the data!

- If there is no more space left in the send buffer, by default `write()` will *block*

# Poll Everywhere

**pollev.com/tqm**

- When we call `write()`, what data do we need to pass to it when writing over the network?

A. **Any data our application needs to send**

B. **All of the above + TCP info (sequence number, port, …)**

C. **All of the above + IP info (source & dest IP addresses…)**

D. **All of the above + Ethernet info (source & dest MAC addresses)**

E. **We're lost…**

# Read/Write Example

- See sendreceive.cpp

```
while (1) {
  int wres = write(socket_fd, readbuf, res);
  if (wres == 0) {
    cerr << "socket closed prematurely" << endl;
    close(socket_fd);
    return EXIT_FAILURE;
  }
  if (wres == -1) {
    if (errno == EINTR)
      continue;
    cerr << "socket write failure: " << strerror(errno) << endl;
    close(socket_fd);
    return EXIT_FAILURE;
  }
  break;
}
```

# Step 5: close()

-
  ```c
  int close(int fd);
  ```
  - Nothing special here – it's the same function as with file I/O
  - Shuts down the socket and frees resources and file descriptors associated with it on both ends of the connection

# Demo: sendreceive.cpp

- Demo, use `netcat -l <port>` to listen on a port and use `./sendreceive localhost <port>` to connect

- Code Walkthrough
  - What hints are we looking for when we LookupName?
    What do you think they mean?
  - What abstraction layer of the OSI model does this program exist in?
  - What if we wanted to make this code read and respond more than once?
    What if we wanted it to keep going until the connection is closed by the server?

# Lecture Outline

- Final Project Demo
- Client-Side Socket Programming (Wrap-up)
- **Server-Side Socket Programming**

# Socket API: Server TCP Connection

Analogy: opening a (boba) shop!

- Pretty similar to clients, but with additional steps:
  1) Figure out the IP address and port on which to listen* Finding a good location Sometimes the location is already known, so this may not be a step.
  2) Create a socket Building the store
  3) `bind()` the socket to the address(es) and port Advertising the store
  4) Tell the socket to `listen()` for incoming clients Open shop!
  5) `accept()` a client connection Next customer in line, Please!
  6) `read()` and `write()` to that connection Transaction occurs
  7) `close()` the client socket Customer leaves shop or refuse service

# Servers

- Servers can have multiple IP addresses ("*multihoming*")
  - Usually have at least one externally-visible IP address, as well as a local-only address (127.0.0.1)

- The goals of a server socket are different than a client socket
  - Want to bind the socket to a particular *port* of one or more IP addresses of the server
  - Want to allow multiple clients to connect to the same port
    - OS uses client IP address and port numbers to direct I/O to the correct server file descriptor

# Step 1: Figure out IP address(es) & Port

- Step 1: `getaddrinfo()` invocation may or may not be needed (but we'll use it)
  - Do you know your IP address(es) already?
    - Static vs. dynamic IP address allocation
    - Even if the machine has a static IP address, don't wire it into the code – either look it up dynamically or use a configuration file
  - Can request listen on all local IP addresses by passing `NULL` as `hostname` and setting `AI_PASSIVE` in `hints.ai_flags`
    - Effect is to use address `0.0.0.0` (IPv4) or `::` (IPv6)

Common and hard to find bug is forgetting to set this☹

Not needed for project tho!

# Step 2: Create a Socket

- Step 2: `socket()` call is same as before
  - Can directly use constants or fields from result of `getaddrinfo()`
  - Recall that this just returns a file descriptor – IP address and port are not associated with socket yet

# Step 3: Bind the socket

- 
```
int bind(int sockfd, const struct sockaddr* addr,
         socklen_t addrlen);
```
  - Looks nearly identical to **connect()**!
  - Returns **0** on success, **-1** on error

We'll just pass in results from getaddrinfo() & socket()

- Some specifics for addr:
  - **Address family:** AF_INET or AF_INET6
    - What type of IP connections can we accept?
    - POSIX systems can handle IPv4 clients via IPv6 ☺
  - **Port:** port in network byte order (**htons()** is handy)
  - **Address:** specify *particular* IP address or *any* IP address
    - "Wildcard address" – INADDR_ANY (IPv4), in6addr_any (IPv6)

# Step 4: Listen for Incoming Clients

- 
  ```
  int listen(int sockfd, int backlog);
  ```
  - Tells the OS that the socket is a listening socket that clients can connect to
  - `backlog`: maximum length of connection queue
    - Gets truncated, if necessary, to defined constant `SOMAXCONN`
    - The OS will refuse new connections once queue is full until server `accept()`s them (removing them from the queue)
  - Returns `0` on success, `-1` on error

- Clients can start connecting to the socket as soon as `listen()` returns
  - Server can't use a connection until you `accept()` it

# Example #1

- See server_bind_listen.cpp
  - Takes in a port number from the command line
  - Opens a server socket, prints info, then listens for connections for 20 seconds
    - Can connect to it using netcat (`nc`)

- Questions:
  - Why do we have a for loop over line 52?
    - What are we looping over?
    - Why can’t we just use the first thing?
    - Why do we call socket and bind in the loop and not after?

# Step 5: Accept a Client Connection

❖

```
int accept(int sockfd, struct sockaddr* addr,
           socklen_t* addrlen);
```

- Returns an active, ready-to-use socket file descriptor connected to a client (or **-1** on error)
  - `sockfd` must have been created, bound, *and* listening
  - Pulls a queued connection or waits for an incoming one
- `addr` and `addrlen` are output parameters
  - `*addrlen` should initially be set to `sizeof(*addr)`, gets overwritten with the size of the client address
  - Address information of client is written into `*addr`
    - Use **`inet_ntop`**`()` to get the client's printable IP address
    - Use **`getnameinfo`**`()` to do a *reverse DNS lookup* on the client

# Example #2

- See server_accept_rw_close.cpp
  - *Takes in a port number from the command line*
  - *Opens a server socket, prints info, then listens for connections*
    - *Can connect to it using netcat (`nc`)*
  - *Previous example is pretty much just the Listen() function in this code*
  - Accepts connections as they come
  - Echoes any data the client sends to it on `stdout` and also sends it back to the client

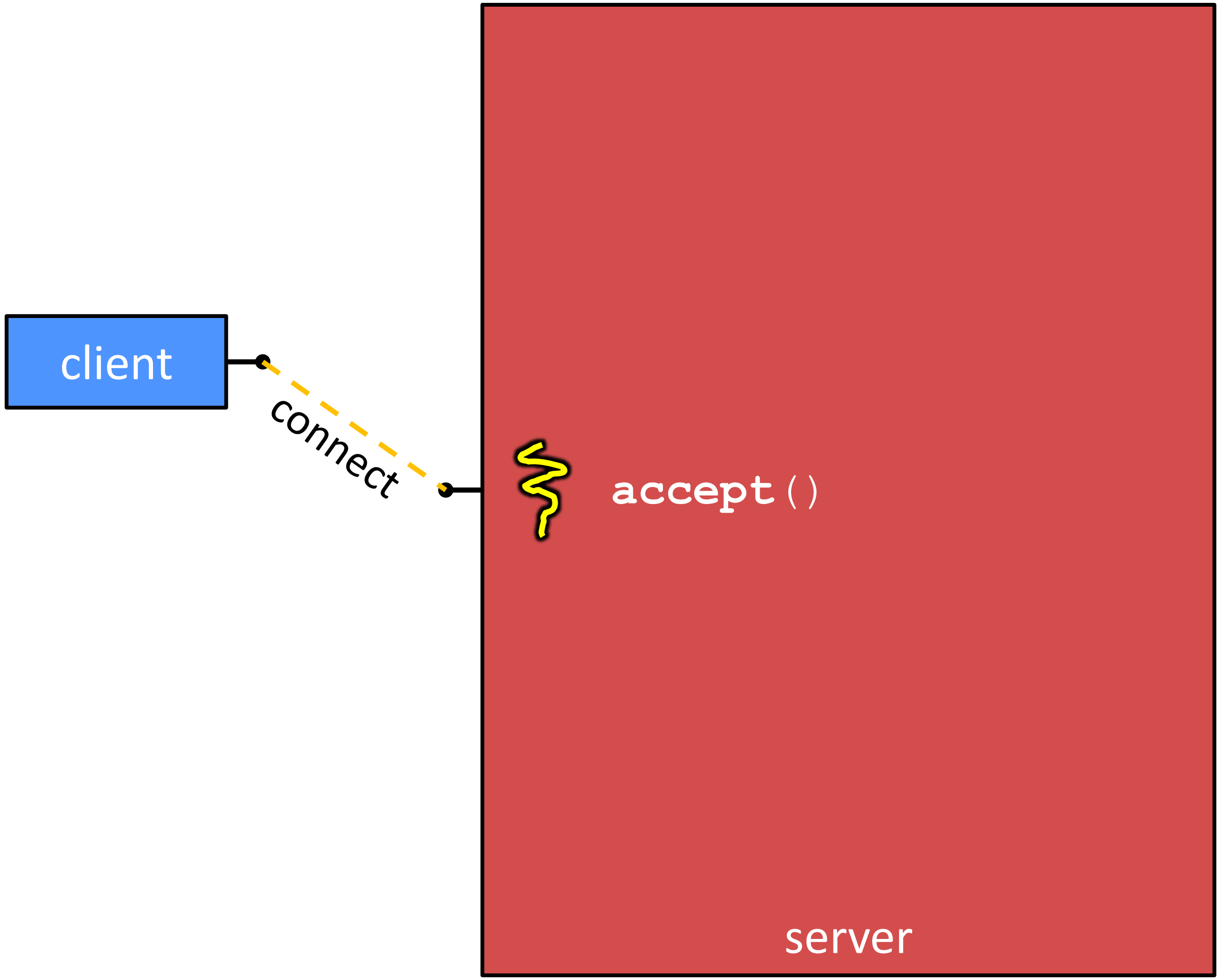
- Question:
  - Why is accept in a while(true) loop?
  - Why doesn’t listen need to be in the loop with accept?
  - Does this handle multiple client? If so, how?

# Something to Note

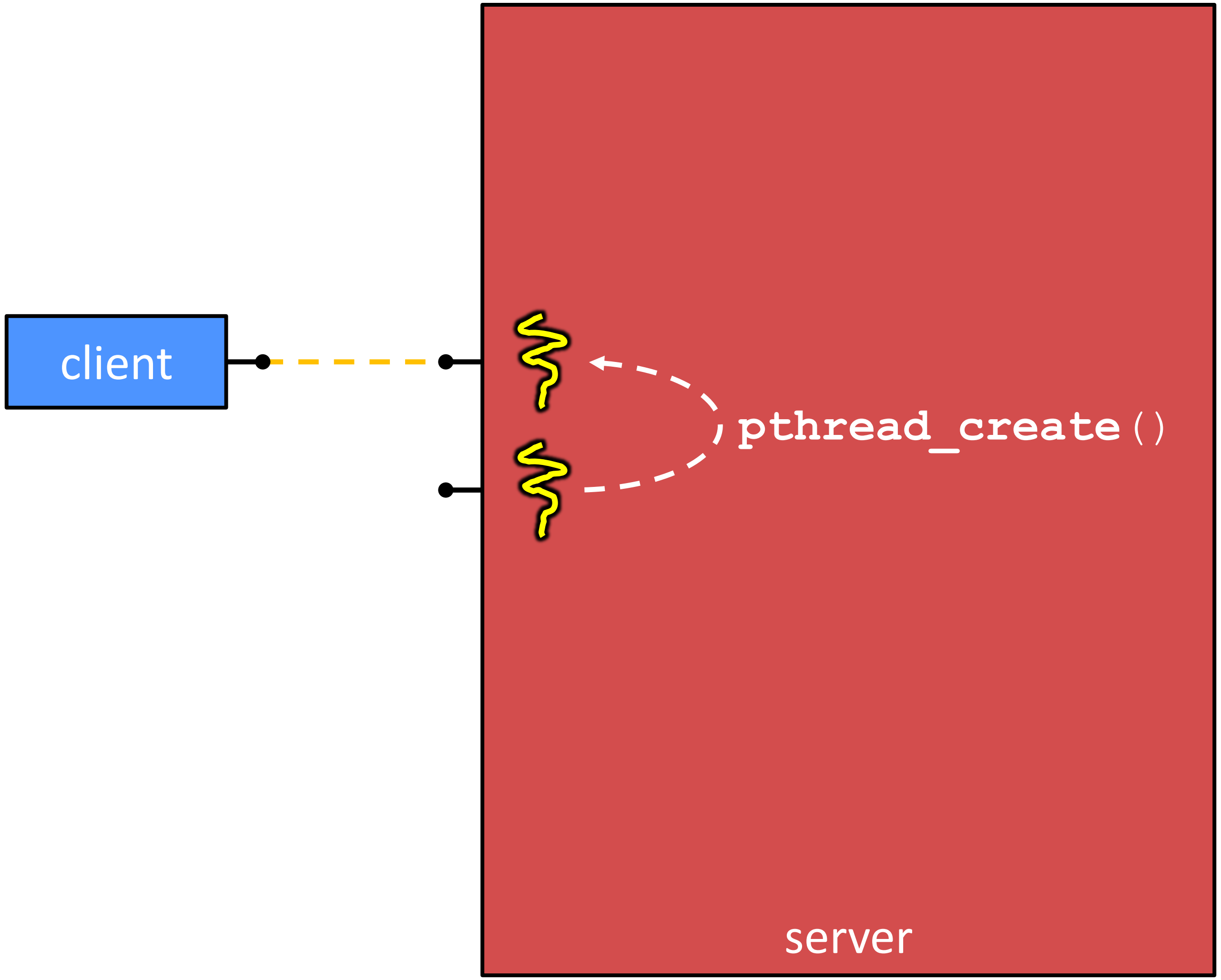
- Our server code is not concurrent
  - Single thread of execution
  - The thread blocks while waiting for the next connection
  - The thread blocks waiting for the next message from the connection

- A crowd of clients is, by nature, concurrent
  - While our server is handling the next client, all other clients are stuck waiting for it ☹

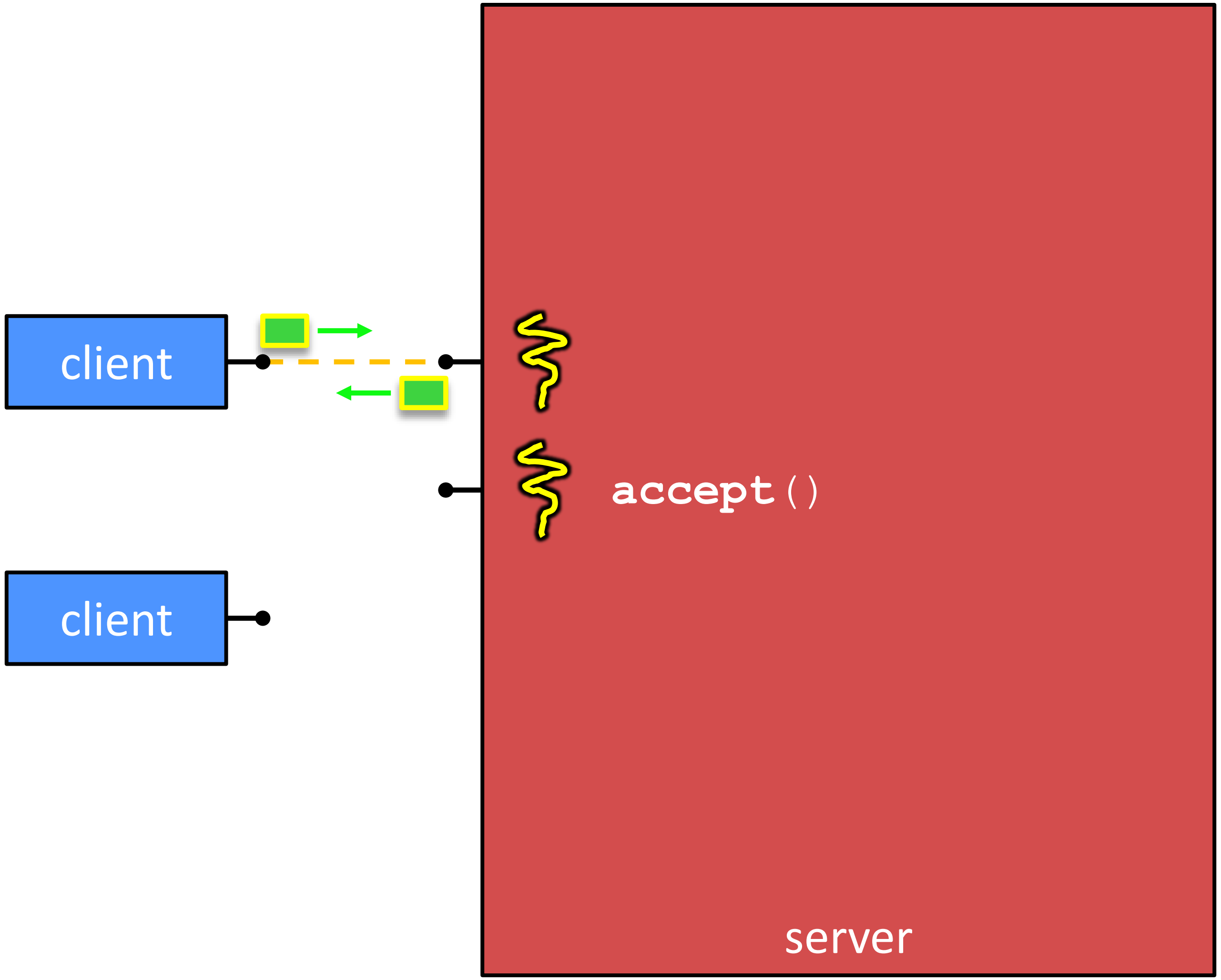
# Multithreaded Server



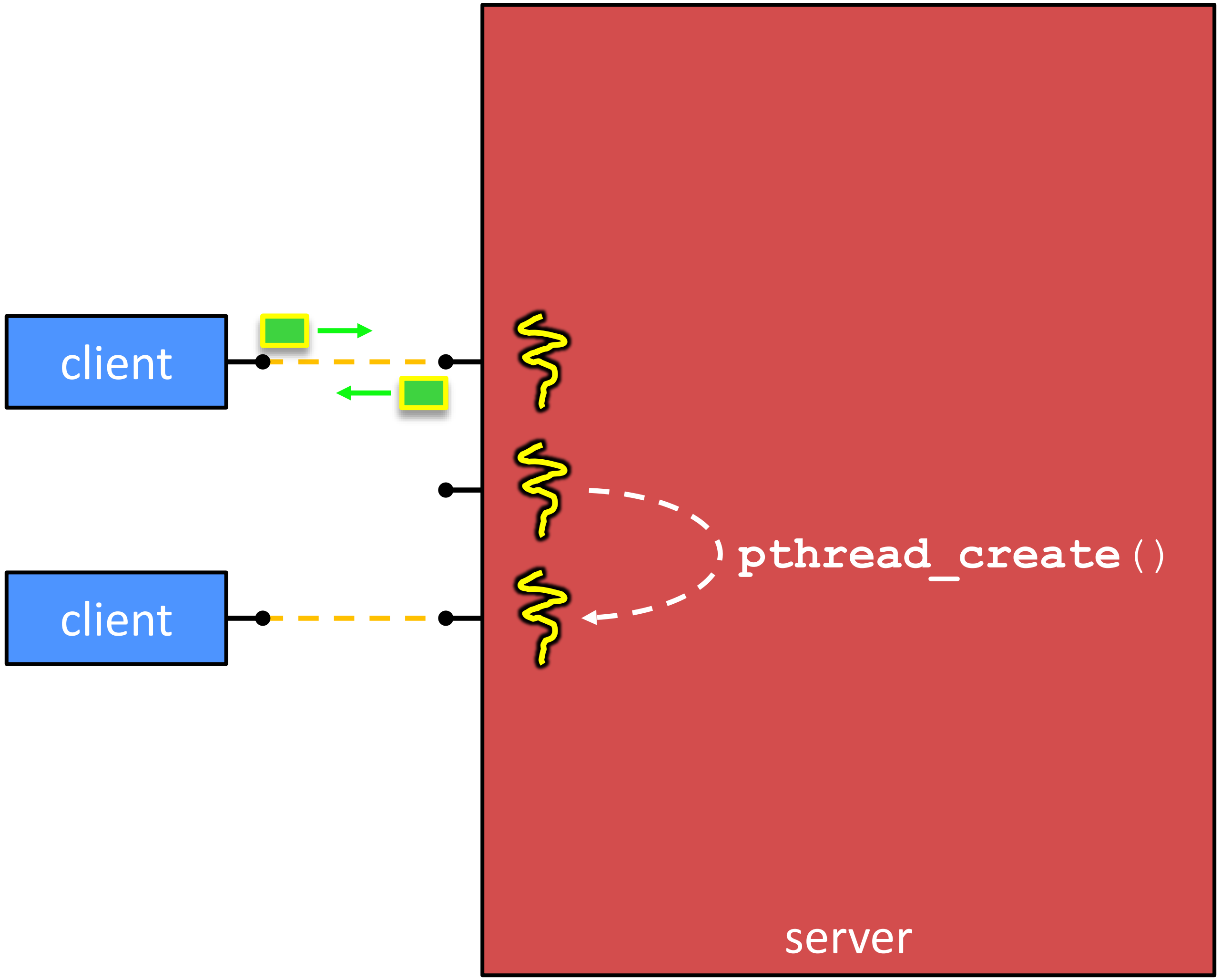
# Multithreaded Server



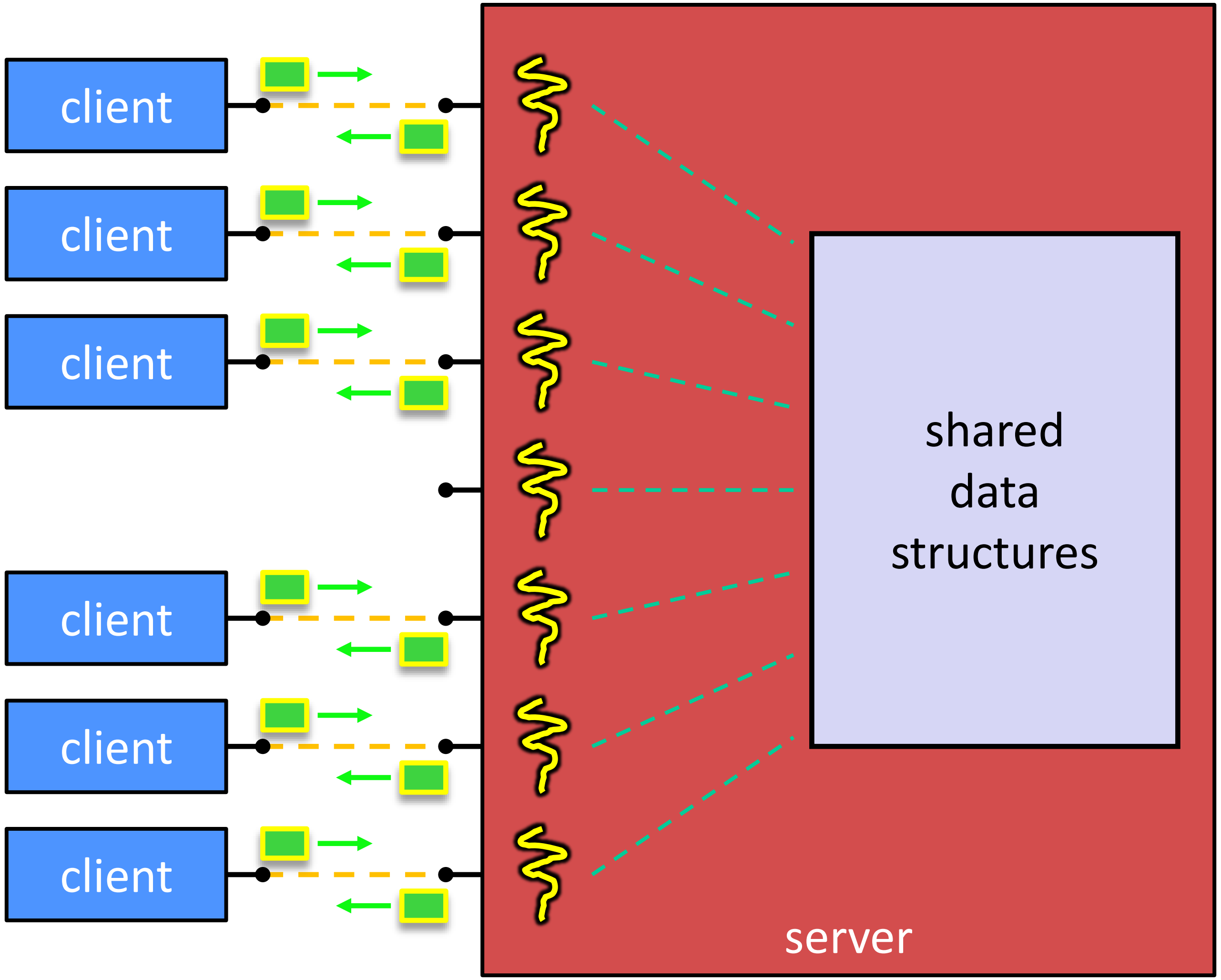
# Multithreaded Server



# Multithreaded Server



# Multithreaded Server



# That’s all!

- Next Lecture:
  - Http ☺