Socket Programming

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevin Bernat	Jialin Cai
Mati Davis	Donglun He
Chandravaran Kunjeti	Heyi Liu
Shufan Liu	Eddy Yang

Poll Everywhere

pollev.com/tqm

- Approximately how many internet connected devices do you own?

Poll Everywhere

pollev.com/tqm

- Which layer handles this problem?

- Host A tries to send a long message to Host B in another city, broken up into many packets. A packet in the middle does not arrive, so Host A sends it again.

Poll Everywhere

pollev.com/tqm

- Which layer handles this problem?

- Host A tries to send a message to Host B, but Host C and Host D are also trying to communicate on the same network, so Host A has to avoid interfering

Logistics

- HW3 Posted Due Thursday 3/30 @ 11:59
  - Should have everything you need
  - Should be on the shorter side theoretically
  - Auto-grader to be released today
- Project Partner Sign up to be released TODAY
  - Project Spec will be released with it to help make a decision
  - Due Wednesday at midnight next week 4/5
- Final Exam Scheduling form out now, complete by tonight @ midnight

Lecture Outline

- **Network Programming**
  - **Sockets API**
  - **Network Addresses**
  - **DNS Lookup**

Files and File Descriptors

- Remember `open()`, `read()`, `write()`, and `close()`?
  - POSIX system calls for interacting with files
  - `open()` returns a file descriptor
    - An integer that represents an open file
    - This file descriptor is then passed to `read()`, `write()`, and `close()`
  - Inside the OS, the file descriptor is used to index into a table that keeps track of any OS-level state associated with the file, such as the file position

Parameters to

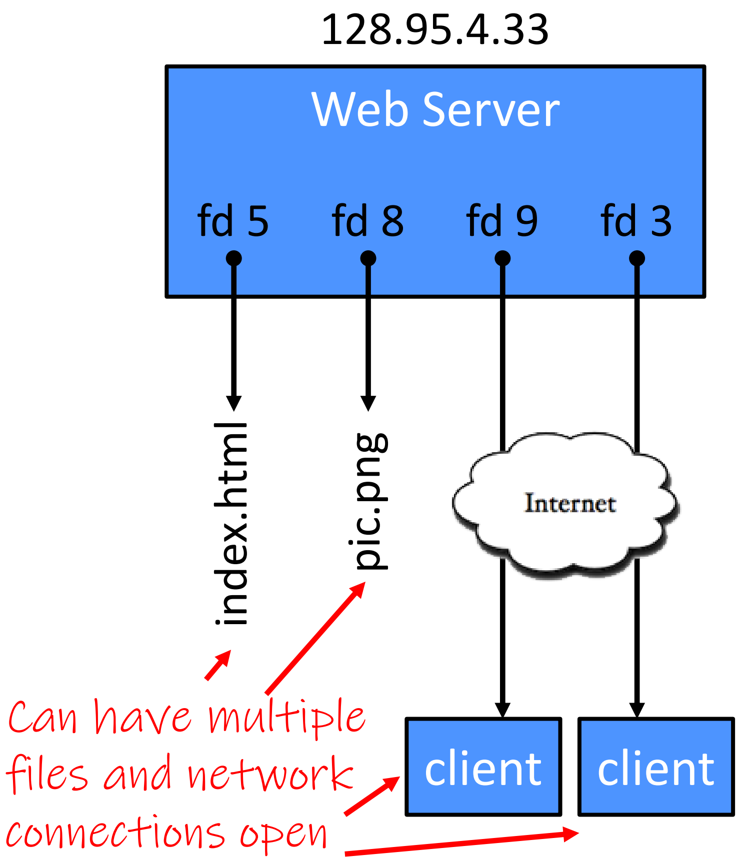
Can't be a pointer, don't want to give address to kernel

Networks and Sockets

- UNIX likes to make *all* I/O look like file I/O
  - You use **`read`**`()` and **`write`**`()` to communicate with remote computers over the network!
  - A file descriptor use for network communications is called a socket
  - Just like with files:
    - Your program can have multiple network channels open at once
    - You need to pass a file descriptor to **`read`**`()` and **`write`**`()` to let the OS know which network channel to use

In other words, we specify the socket to read/write on

File Descriptor Table



OS's File Descriptor Table for the Process

File Descriptor	Type	Connection
0	pipe	stdin (console)
1	pipe	stdout (console)
2	pipe	stderr (console)
3	TCP socket	local: 128.95.4.33:80 remote: 44.1.19.32:7113
5	file	index.html
8	file	pic.png
9	TCP socket	local: 128.95.4.33:80 remote: 102.12.3.4:5544

0,1,2 always start as stdin, stdout & stderr.

Types of Sockets

- **Stream sockets** What we will focus on
  - For connection-oriented, point-to-point, reliable byte streams
    - Using TCP, SCTP, or other stream transports
- **Datagram sockets**
  - For connection-less, one-to-many, unreliable packets
    - Using UDP or other packet transports
- Raw sockets
  - For layer-3 communication (raw IP packet manipulation)

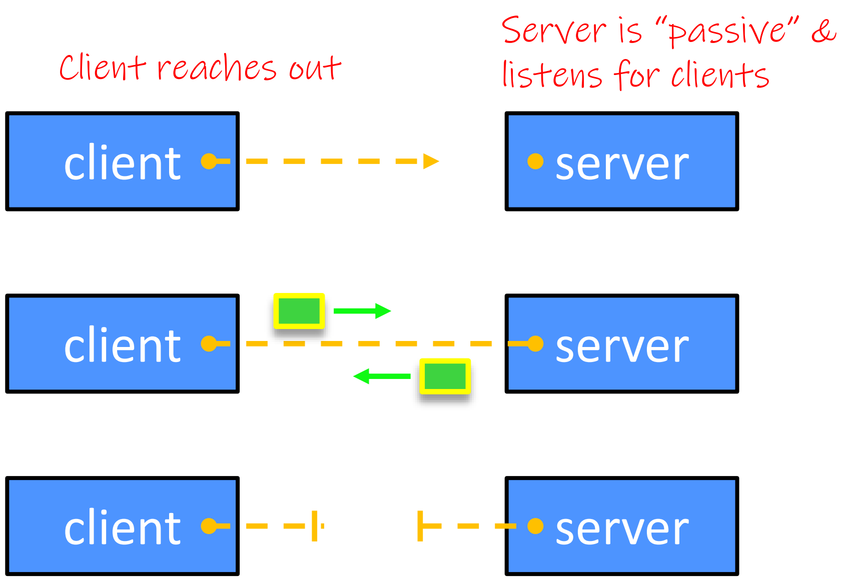
Stream Sockets

- Typically used for client-server communications
  - Client: An application that establishes a connection to a server
  - Server: An application that receives connections from clients
  - Can also be used for other forms of communication like peer-to-peer

1) Establish connection:

2) Communicate:

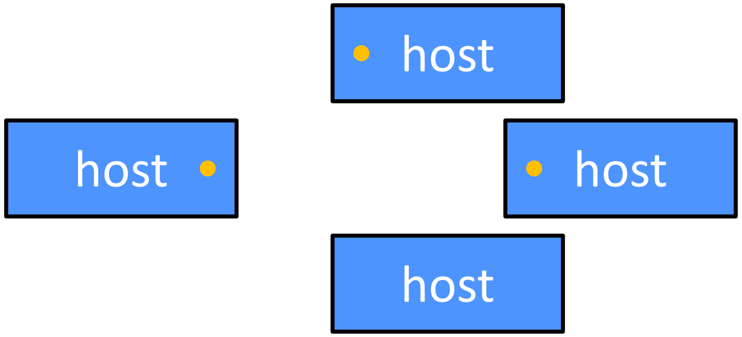
3) Close connection:



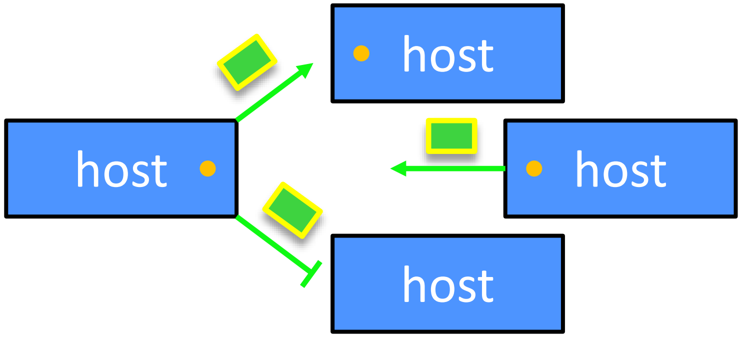
Datagram Sockets

- Often used as a building block
  - No flow control, ordering, or reliability, so used less frequently
  - *e.g.* streaming media applications or DNS lookups

1) Create sockets:



2) Communicate:



The Sockets API

- Berkeley sockets originated in 4.2BSD Unix (1983)
  - It is the standard API for network programming
    - Available on most OSs
  - Written in C

Can still use these in C++ code
You'll see some C-idioms and design practices.

- POSIX Socket API
  - A slight update of the Berkeley sockets API
    - A few functions were deprecated or replaced
    - Better support for multi-threading was added

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:
  1) Figure out the IP address and port to which to connect ** Today **
  2) Create a socket
  3) Connect the socket to the remote server
  4) `read()` and `write()` data using the socket
  5) Close the socket

New stuff (steps 1–3)

Same as file I/O (steps 4–5)

Good Breakdown of this entire process in section tomorrow

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

IPv4 Network Addresses

- An IPv4 address is a **4-byte** tuple (2^32 addresses)
  - For humans, written in "dotted-decimal notation"
  - *e.g.* 128.95.4.1 (`80:5f:04:01` in hex)

- IPv4 address exhaustion
  - There are $2^{32} \approx 4.3$ billion IPv4 addresses
  - There are ≈ 7.77 billion people in the world (February 2020)

How many internet connected devices do each of us have?

IPv6 Network Addresses

- An IPv6 address is a **16-byte** tuple (2^{128} addresses ~about 3.4×10^{38})
  - Typically written in "hextets" (groups of 4 hex digits)

2 rules for human readability

    - 1 Can omit leading zeros in hextets
    - 2 Double-colon replaces consecutive sections of zeros
  - *e.g.* `2d01:0db8:f188:0000:0000:0000:0000:1f33`
    - Shorthand: `2d01:db8:f188::1f33`
  - Transition is still ongoing
    - IPv4-mapped IPv6 addresses
      - 128.95.4.1 mapped to `::ffff:`128.95.4.1 or `::ffff:805f:401`
    - This unfortunately makes network programming more of a headache ☹

Linux Socket Addresses

- Structures, constants, and helper functions available in `#include <arpa/inet.h>`
- Addresses stored in network byte order (big endian)
- Converting between host and network byte orders:
  - `uint32_t htonl(uint32_t hostlong);`
  - `uint32_t ntohl(uint32_t netlong);`
    - ‘`h`’ for host byte order and ‘`n`’ for network byte order
    - Also versions with ‘`s`’ for short (`uint16_t` instead)
- How to handle both IPv4 and IPv6?
  - Use C structs for each, but make them somewhat similar
  - Use defined constants to differentiate when to use each: `AF_INET` for IPv4 and `AF_INET6` for IPv6 (other types of sockets exist, not just ipv4 & ipv6)

First field in a struct is always an ID

"AF" = Address Family

IPv4 Address Structures

```
// IPv4 4-byte address
struct in_addr {
  uint32_t s_addr;                  // Address in network byte order
};

// An IPv4-specific address structure
struct sockaddr_in {
  sa_family_t    sin_family;    // Address family: AF_INET
  in_port_t      sin_port;      // Port in network byte order
  struct in_addr sin_addr;      // IPv4 address
  unsigned char  sin_zero[8];   // Pad out to 16 bytes
};
```

Always big endian

should always be AF_INET

(2 bytes)

`struct sockaddr_in`:

family	port	addr	zero
0–2	2–4	4–8	8–16

Practice Question

- Assume we have a `struct sockaddr_in` that represents a socket connected to 198.35.26.96 (c6:23:1a:60) on port 80 (0x50) stored on a little-endian machine.
  - `AF_INET = 2`
  - Fill in the bytes in memory below (in hex):

sin_family (host) | sin_port (network) | sin_addr (network)

0	02	00	00	50	C6	23	1A	60
8	00	00	00	00	00	00	00	00

zeroes (host)

IPv6 Address Structures

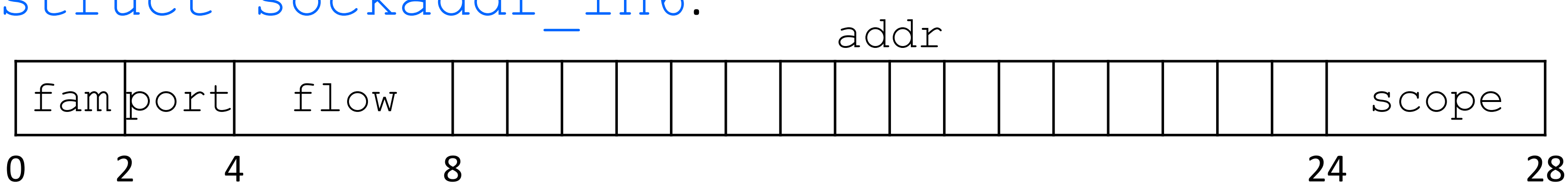
```
// IPv6 16-byte address
struct in6_addr {
  uint8_t s6_addr[16];          // Address in network byte order
};

// An IPv6-specific address structure
struct sockaddr_in6 {
  sa_family_t     sin6_family;    // Address family: AF_INET6
  in_port_t       sin6_port;      // Port number
  uint32_t        sin6_flowinfo;  // IPv6 flow information
  struct in6_addr sin6_addr;      // IPv6 address
  uint32_t        sin6_scope_id;  // Scope ID
};
```

should always be AF_INET6

Can ignore

`struct sockaddr_in6`:



Generic Address Structures

struct sockaddr*

```
// A mostly-protocol-independent address structure.
// Pointer to this is parameter type for socket system calls.
struct sockaddr {
  sa_family_t sa_family;    // Address family (AF_* constants)
  char        sa_data[14];  // Socket address (size varies
                            // according to socket domain)
};

// A structure big enough to hold either IPv4 or IPv6 structs
struct sockaddr_storage {
  sa_family_t ss_family;    // Address family

  // padding and alignment; don't worry about the details
  char __ss_pad1[_SS_PAD1SIZE];
  int64_t __ss_align;
  char __ss_pad2[_SS_PAD2SIZE];
};
```

Family is always first to identify the socket type

struct sockaddr isn't big enough for ipv6

- Commonly create `struct sockaddr_storage`, then pass pointer cast as `struct sockaddr*` to **`connect`**()

Address Conversion

Address family String representation Addr destination: struct in_addr* // or struct in_6addr*

- `int inet_pton(int af, const char* src, void* dst);`
  - Converts human-readable string representation ("**p**resentation") to **n**etwork byte ordered address
  - Returns 1 (success), 0 (bad `src`), or -1 (error)

genaddr.cc

```
#include <stdlib.h>
#include <arpa/inet.h>

int main(int argc, char **argv) {
  struct sockaddr_in sa;    // IPv4
  struct sockaddr_in6 sa6;  // IPv6

  // IPv4 string to sockaddr_in (192.0.2.1 = C0:00:02:01).
  inet_pton(AF_INET, "192.0.2.1", &(sa.sin_addr));

  // IPv6 string to sockaddr_in6.
  inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr));

  return EXIT_SUCCESS;
}
```

Address Conversion

Address family

Addr src:
struct in_addr*
// or
struct in_6addr*

- ```
  const char* inet_ntop(int af, const void* src,
                        char* dst, socklen_t size);
  ```
  - Converts network addr in `src` into buffer `dst` of size `size`
  - Returns `dst` on success; `NULL` on error

genstring.cc

```
#include <stdlib.h>
#include <arpa/inet.h>

int main(int argc, char **argv) {
  struct sockaddr_in6 sa6;           // IPv6
  char astring[INET6_ADDRSTRLEN];    // IPv6

  // IPv6 string to sockaddr_in6.
  inet_pton(AF_INET6, "2001:0db8:63b3:1::3490", &(sa6.sin6_addr));

  // sockaddr_in6 to IPv6 string.
  inet_ntop(AF_INET6, &(sa6.sin6_addr), astring, INET6_ADDRSTRLEN);
  std::cout << astring << std::endl; // 2001:0db8:63b3:1::3490

  return EXIT_SUCCESS;
}
```

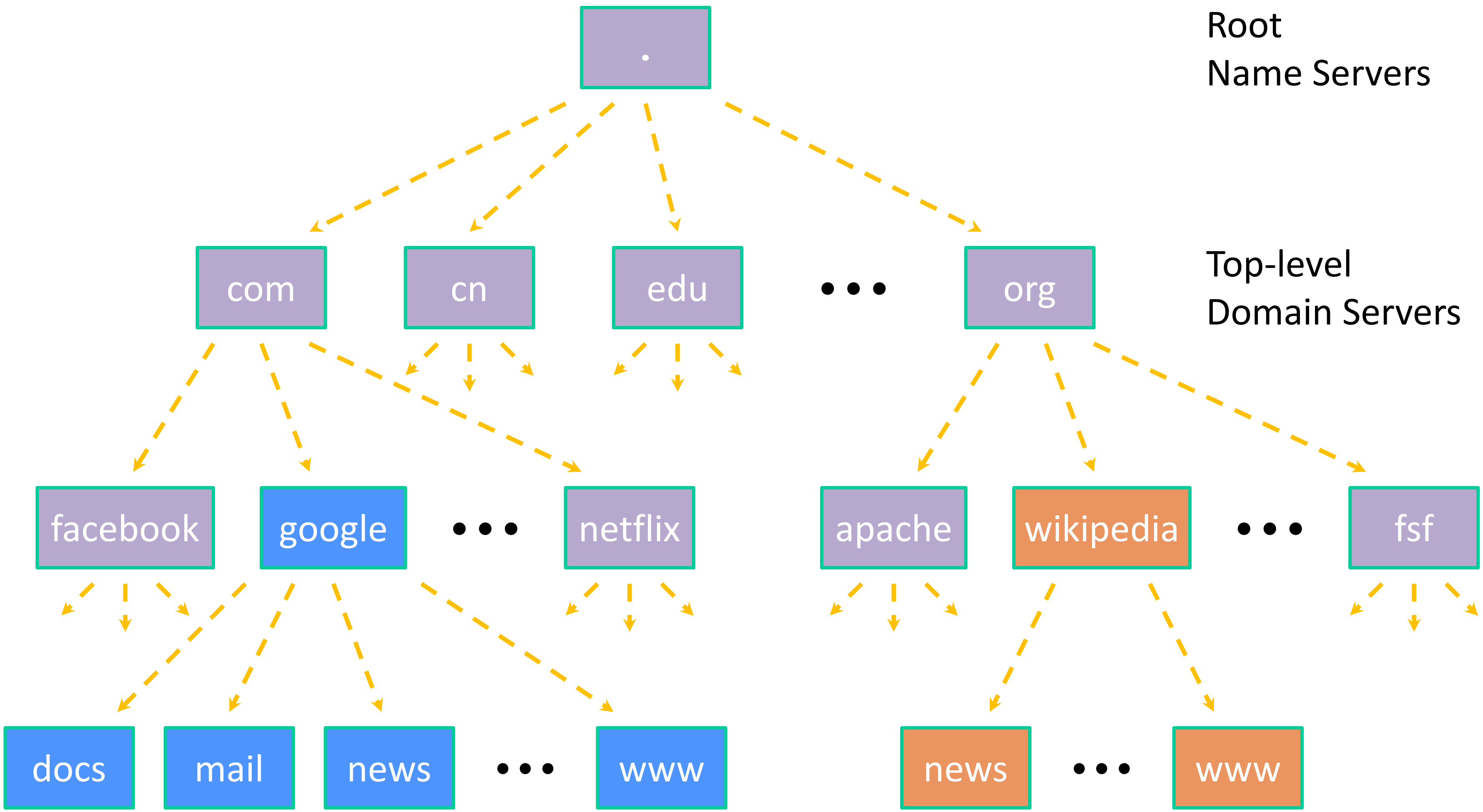
If converting ipv4:
INET_ADDRSTRLEN
```

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>`
  - ```
    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`

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

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
};
```

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.cc

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.cc

```
#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.cc

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

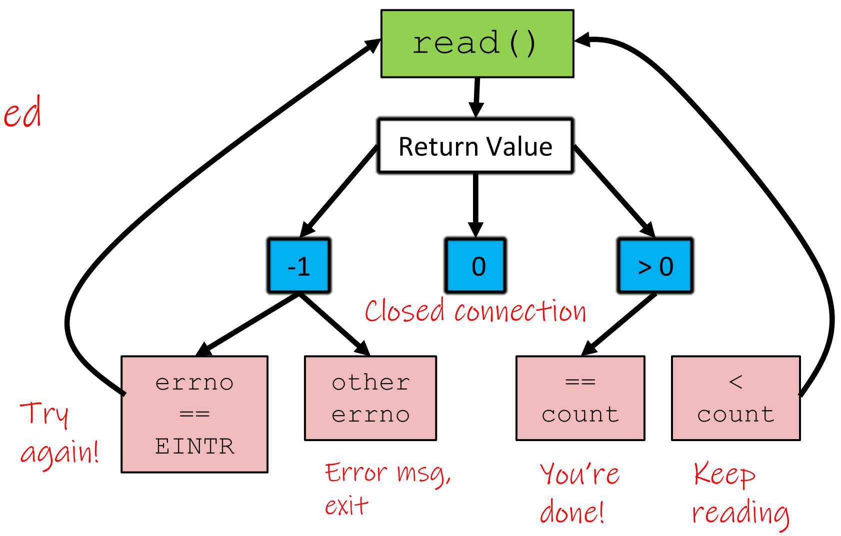
// Create the socket.
int socket_fd = socket(addr.ss_family, SOCK_STREAM, 0);
if (socket_fd == -1) {
  cerr << "socket() failed: " << strerror(errno) << endl;
  return EXIT_FAILURE;
}

// 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;
}
```

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
  - How might this cause *deadlock*? If both server and client try to read with no data sent
  - Can `read()` return 0?
    Yes, connection could close and 0 is returned



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…

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…**

POSIX Sockets is an interface for using the transport layer.

Information about transport layer + below are abstracted away & handled for us.

Read/Write Example

- See sendreceive.cc

```
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
  ```

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
  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☹

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.cc
  - 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`)

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

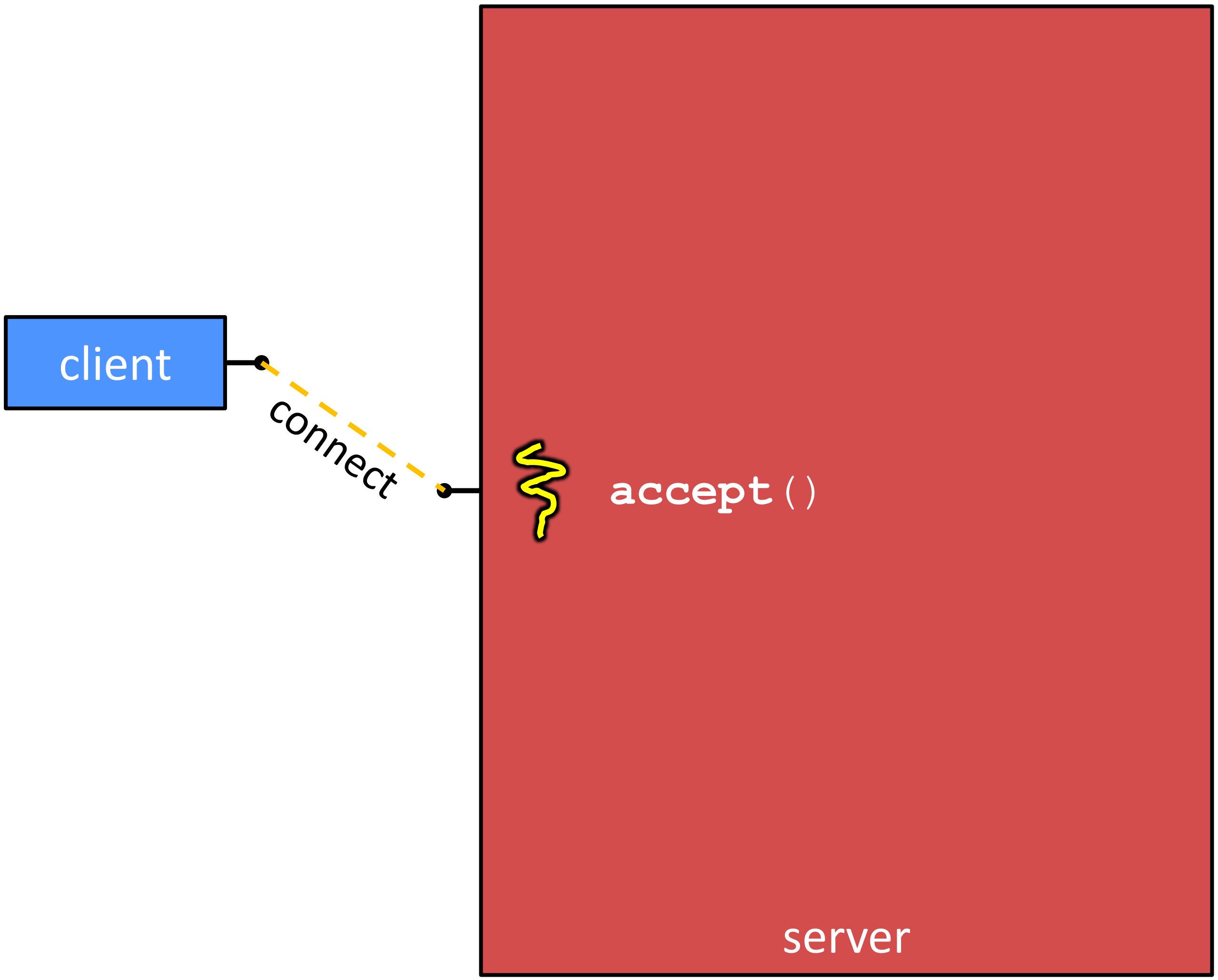
- See server_accept_rw_close.cc
  - *Takes in a port number from the command line*
  - *Opens a server socket, prints info, then listens for connections*
    - *Can connect to it using netcat (`nc`)*
  - Accepts connections as they come
  - Echoes any data the client sends to it on `stdout` and also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

Something to Note

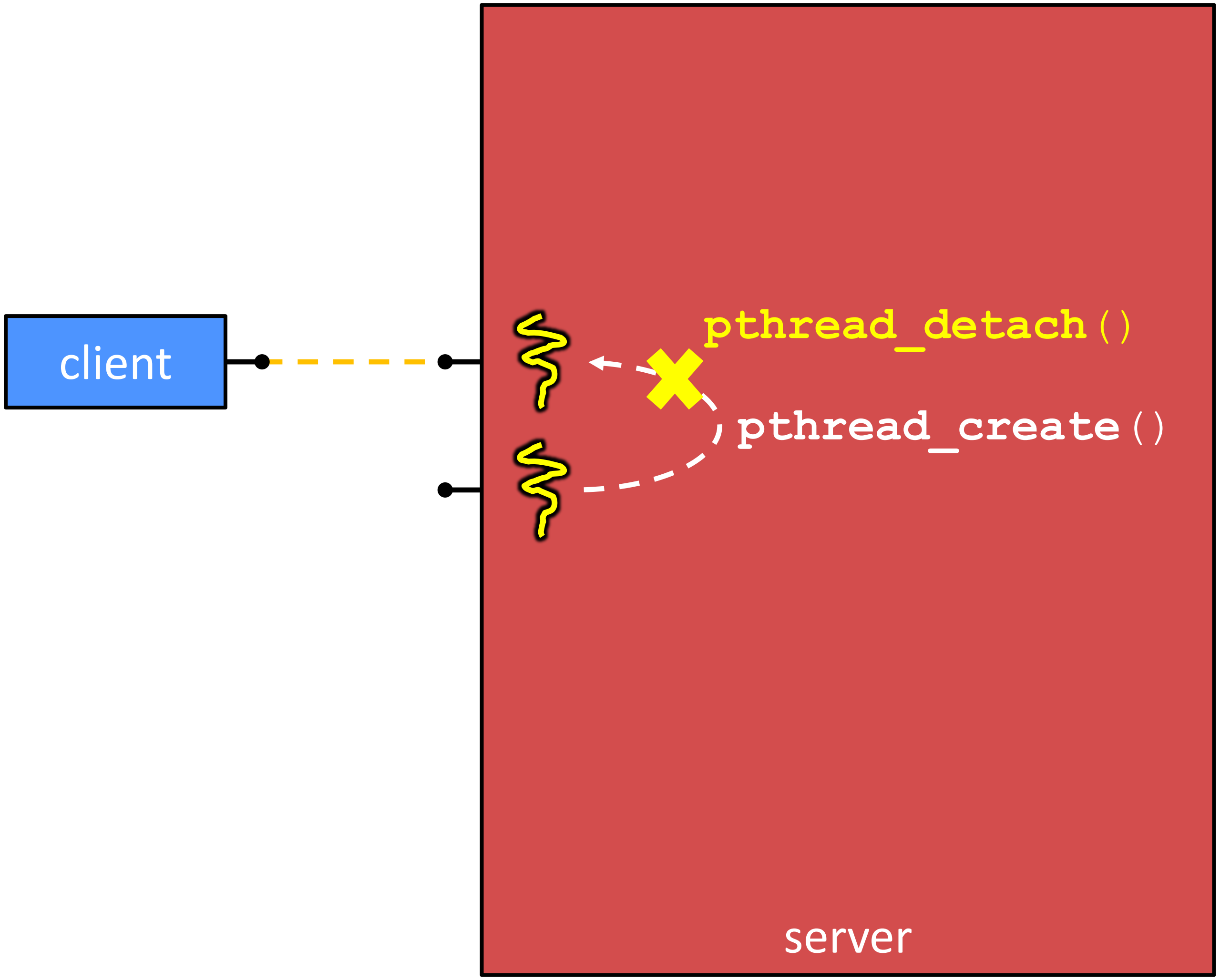
- Our server code is not concurrent
  - Single thread of execution
  - The thread blocks while waiting for the next connection
  - The thread blocks waiting for the next message from the connection

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

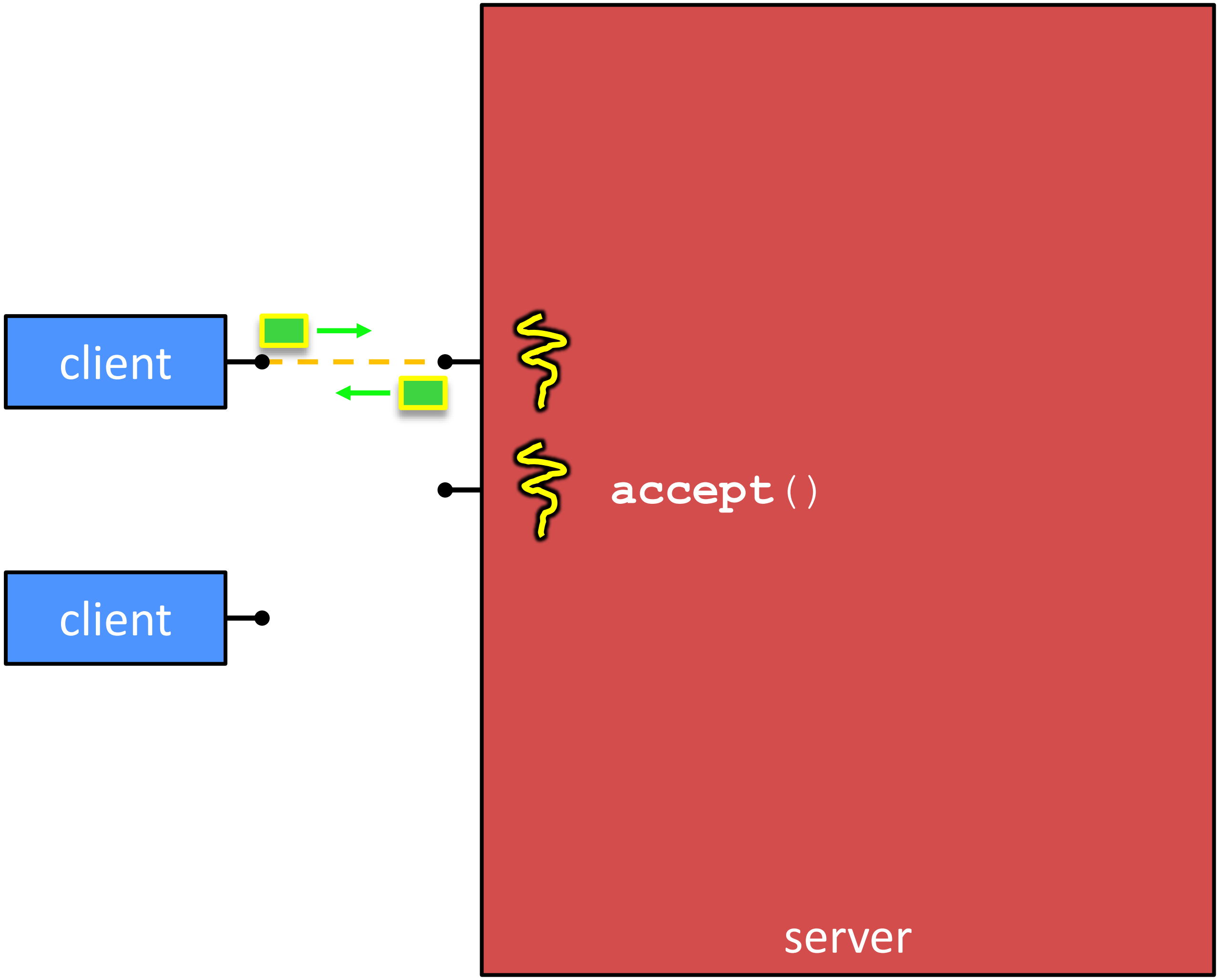
Multithreaded Server



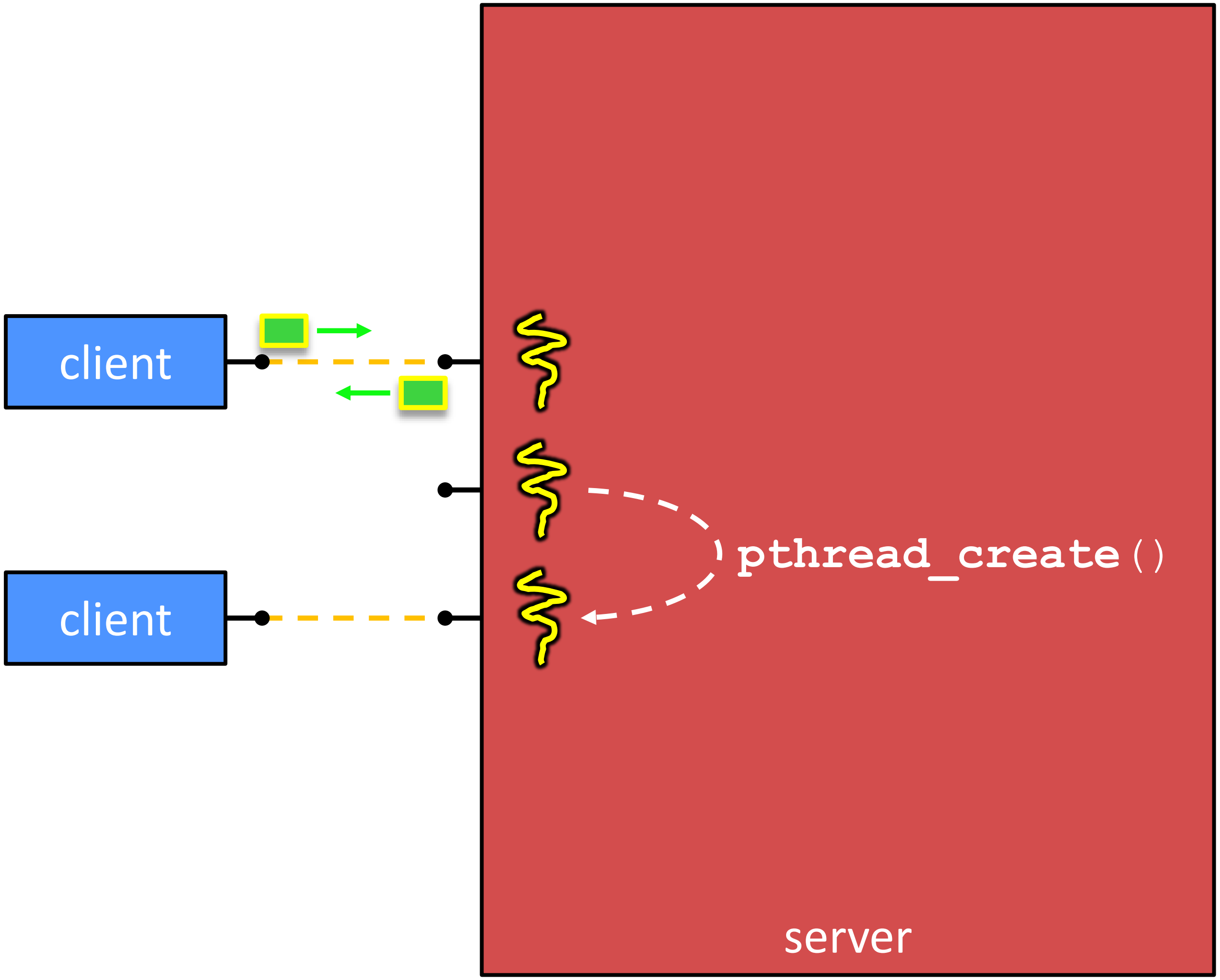
Multithreaded Server



Multithreaded Server



Multithreaded Server



Multithreaded Server