Socket Programming

Intermediate Computer Systems Programming, Fall 2025

Instructor: Travis McGaha

TAs: Theodor Bulacovschi Ash Fujiyama

Administrivia

- Final Project Partner Sign-up
  - If you want to work solo, let me know or you will be randomly assigned a partner
  - Sign-ups posted tonight
- HW9 posted after class today or early tomorrow
  - implementing (simplified) TCP
- Check-in posted same time as HW9

Lecture Outline

- **IP Addresses && SocketAddress**
- UDPSocket
- TCPStream
- TCPListener

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
    - And ipv6
  - Is usable for getting something working. Doing things properly, error checking & supporting both IPV4 and IPV6 sucks

The nixnet Sockets API

- Something I wrote in C++
  - Same style as the Rust net module
  - Uses C++/Rust idioms
- Less painful to handle IP Addresses and works well (I think, but I wrote it :P)
- In the nixnet namespace
- Posted attached to this lecture as “lecture code”
- Read the README.md about installing clang++-19

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 ≈ 8-8.2 billion people in the world (Nov 2025)

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 ☹

Quick Raise Hands

- Do you think we can have multiple connections on a computer to the same port?
  - Example Ports:
    - 80 for HTTP (web traffic)
    - 443 for HTTPS (Secure web traffic)

Socket

- A socket is a network file descriptor
- Each socket is uniquely identified by:
  - Protocol (either TCP or UDP)
  - IP Address
  - Port

- Socket Address is technically the combination of all three of these things but often “Socket Address” just refers to the IP Address & Port pair

Aside: std::expected

- Added in C++23, very similar to std::optional
  - **`std::optional<T>`** can have the type T or be nullopt
  - **`std::expected<T, U>`** can have the type T or the error (unexpected) type U
  - Some member functions
    - has_value()
    - has_error()
    - value()
    - error()
- Example:

```
std::expected<int, std::string> sqrt(int n) {
  if (n < 0) {
    return std::unexpected("can't square root a negative number");
  }
  int res = ..... math ......;
  return res;
}

auto res = sqrt(n);
if (!res) {
  std::cerr << res.error() << std::endl;
}
int x = res.value();
```

Aside: nixnet::errno_t

- nixnet::errno_t
  - Errors in posix are usually indicated by setting a global (thread local) int called errno
  - posix errno is really easy to accidentally mess it up and overwrite the value:
  - This type encapsulates it into a type that is harder to mess up.
- What's the issue here?
  - **`socket()`** returns -1 on error and sets errno to indicate what error occurred

```
int res = socket(AF_INET, SOCK_STREAM, 0);

if (res == -1) { // Error!
  std::cerr << "ERROR ENCOUNTERED CREATING A SOCKET" << std::endl;

  // use strerror to get the string explaining the error
  std::cerr << strerror(errno) << std::endl;
  exit(EXIT_FAILURE);
}
```

← This print will overwrite errno

You can do almost nothing between when the error occurs and evaluating errno.

Aside: nixnet::result

- A std::expected that has the error type set to `ninxnet::erno_t`

- Has a `expect()` function
  - If there is an error, the specified error message + the error held is printed and it throws an exception.
  - If there is a value, it is returned.
  - Rust design pattern, std::expected doesn't have this

```
nixnet::SocketAddr addr = nixnet::SocketAddr::v6_from_str(argv[1], argv[2])
                              .expect("Invalid port or IPv6 address passed in");
```

nixnet::SocketAddr

- Struct that represents either an IPv4 or IPv6 address and a port number

- Follows rust pattern, instead of constructing, you call a “from” function that can return either the thing or an error code.

```
static result<SocketAddr> SocketAddr::v6_from_str(std::string_view string_rep,
                                                  std::string_view port_str);

static result<SocketAddr> SocketAddr::v6_from_str(std::string_view string_rep,
                                                  uint16_t port_num);


SocketAddr home = nixnet::SocketAddr::v4_from_str("127.0.0.1", 3400).expect("this bad msg");
```

Lecture Outline

- IP Addresses && SocketAddress
- **UDPSocket**
- TCPStream
- TCPListener

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

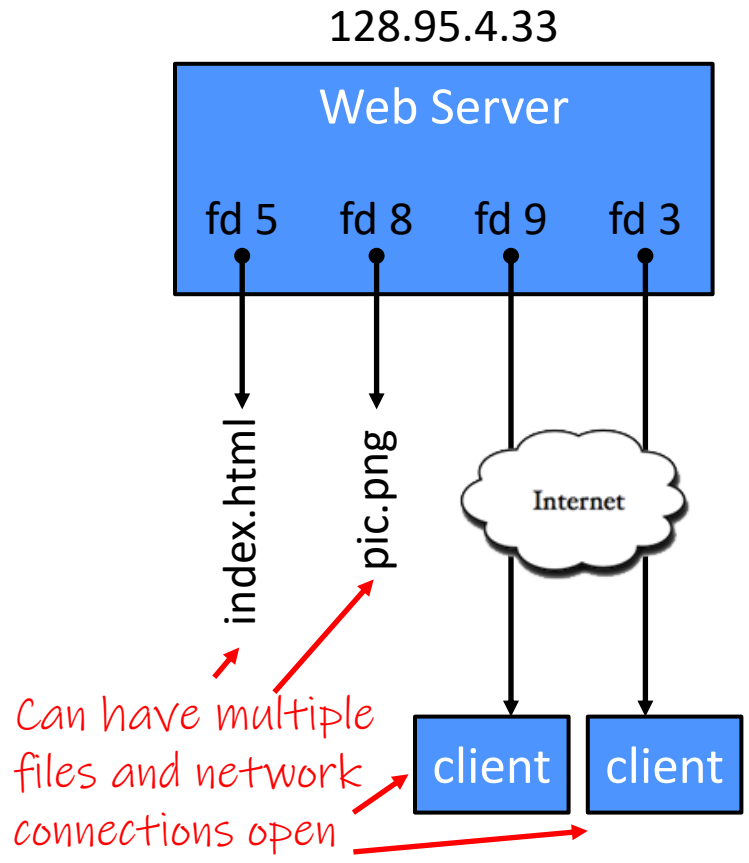
Can't be a pointer, don't want to give address to kernel

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
  - A Socket is an endpoint for network communication
  - 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.

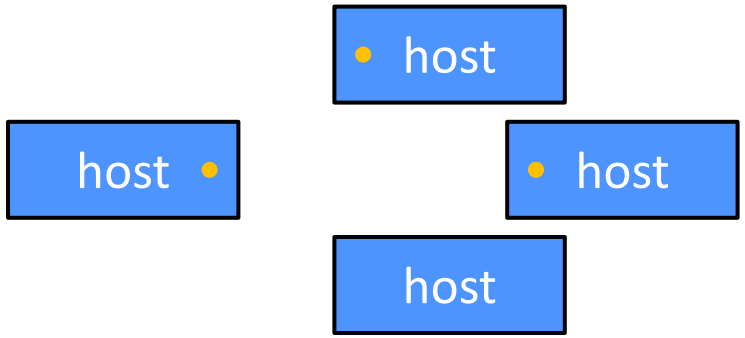
Types of Sockets

- Stream sockets
  - 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)

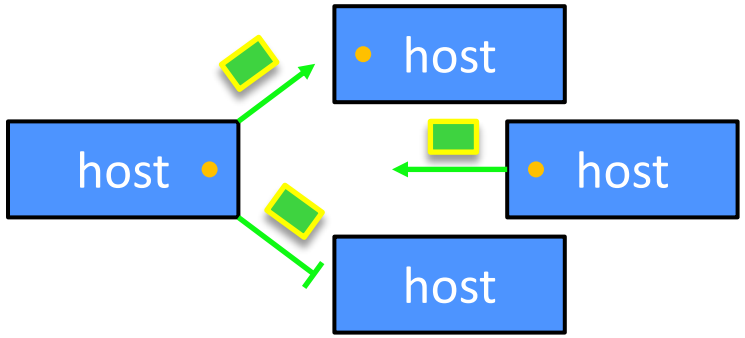
Datagram Sockets

- 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:



UDPSocket

- Demo udp_send
- send_socket
  - Ephemeral port allocated
- Send fails on message to big
- Send doesn’t fail if no one is there to receive it!

Maximum Transmission Unit

- Sometimes called MTU
- Size of the largest unit of data that can be communicated in a single network layer (IP Layer) transaction.
- Size is dynamic and depends on the underlying infrastructure
- Trying to send a UDP message larger than the MTU size will cause an error
- In this class we artificially set the MTU for udp sockets to their minimum value
  - IPv4: 68 bytes
  - IPv6: 1280 bytes

UDPSocket

- Demo udp_send.cpp

- send_socket
  - Ephemeral port allocated
  - Ephemeral ports are “temporary” ports.
  - For the socket “reaching out”, then an exact port number doesn’t matter since the protocol & application will be found out by the port number of who we are connecting to

- Send fails on message to big ☹

- Send doesn’t fail if no one is there to receive it!

UDP connect

- Connect doesn't really do much for UDP sockets
  - IT DOES NOT SETUP A "CONNECTION" LIKE TCP DOES
- When you send() data and don't specify an address, it will go to the connected address as a default
- Data that is recv()'d is only recv'd from the connected address, data from other addresses are ignored
- Other very small things
- Packets still follow UDP, can be dropped, or received out of order

UDPSocket

- Demo udp_recv.cpp
- recv vs recv_from
- Structured binding to get the results nicely

Lecture Outline

- IP Addresses && SocketAddress
- UDPSocket
- **TCPStream**
- TCPListener

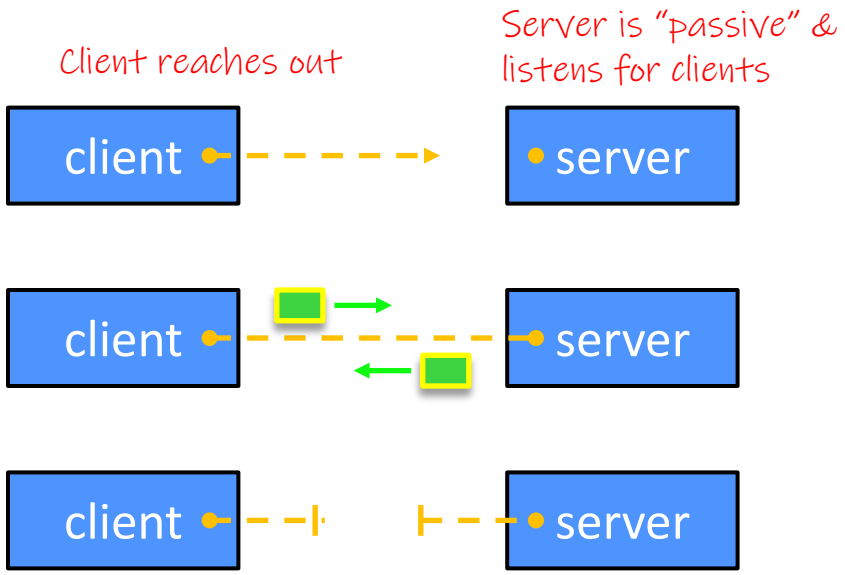
Stream Sockets

- 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:



TCP Connection

- TCP Connections are identified by 4 things
  - Local address
  - Local port
  - Peer address
  - Peer port

TCPStream

- tcp_send.cpp demo / walk through
- TcpStream::connect
- send / recv
  - Can send huge number of bytes fine

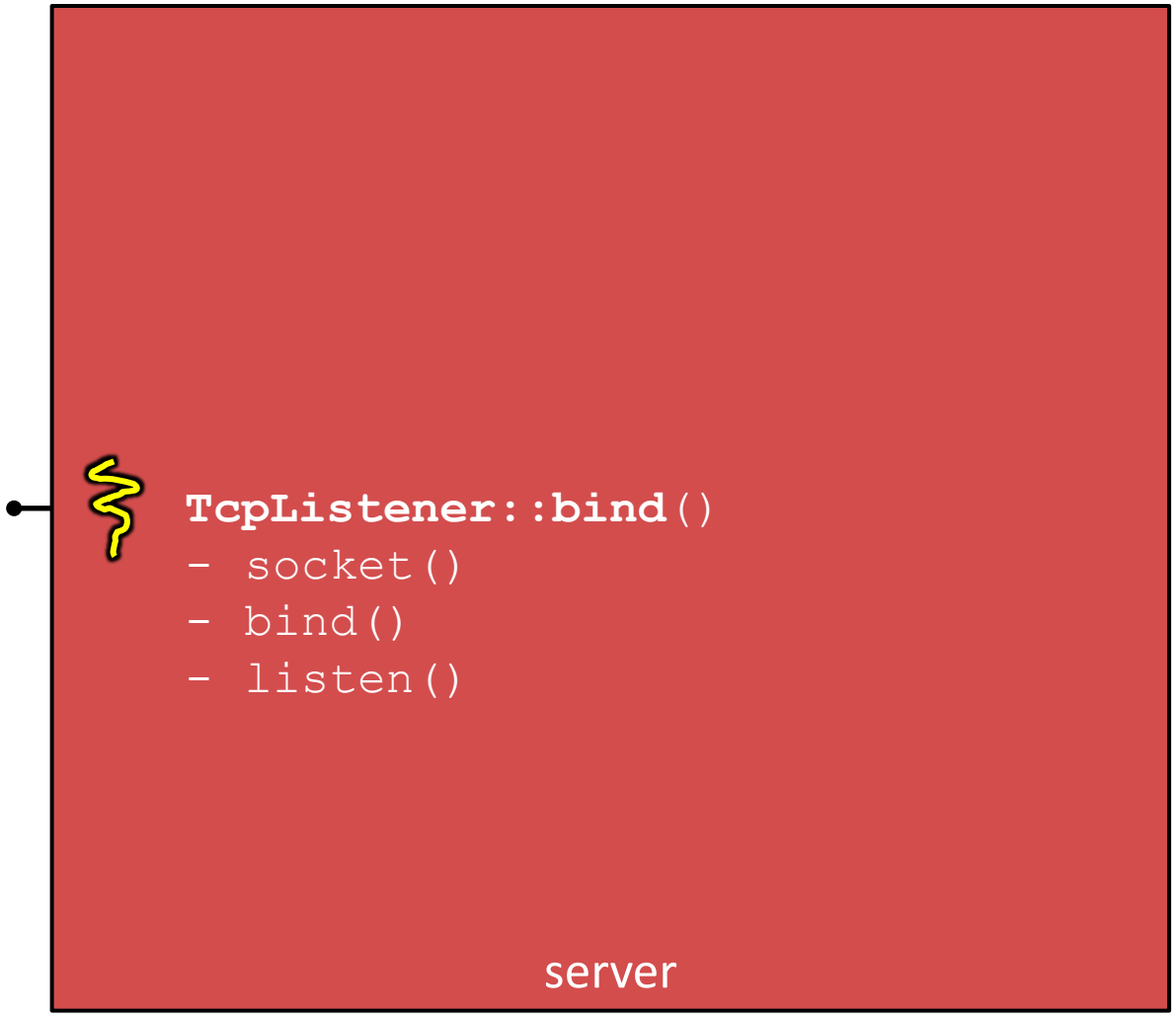
Lecture Outline

- IP Addresses && SocketAddress
- UDPSocket
- TCPStream
- **TCPListener**

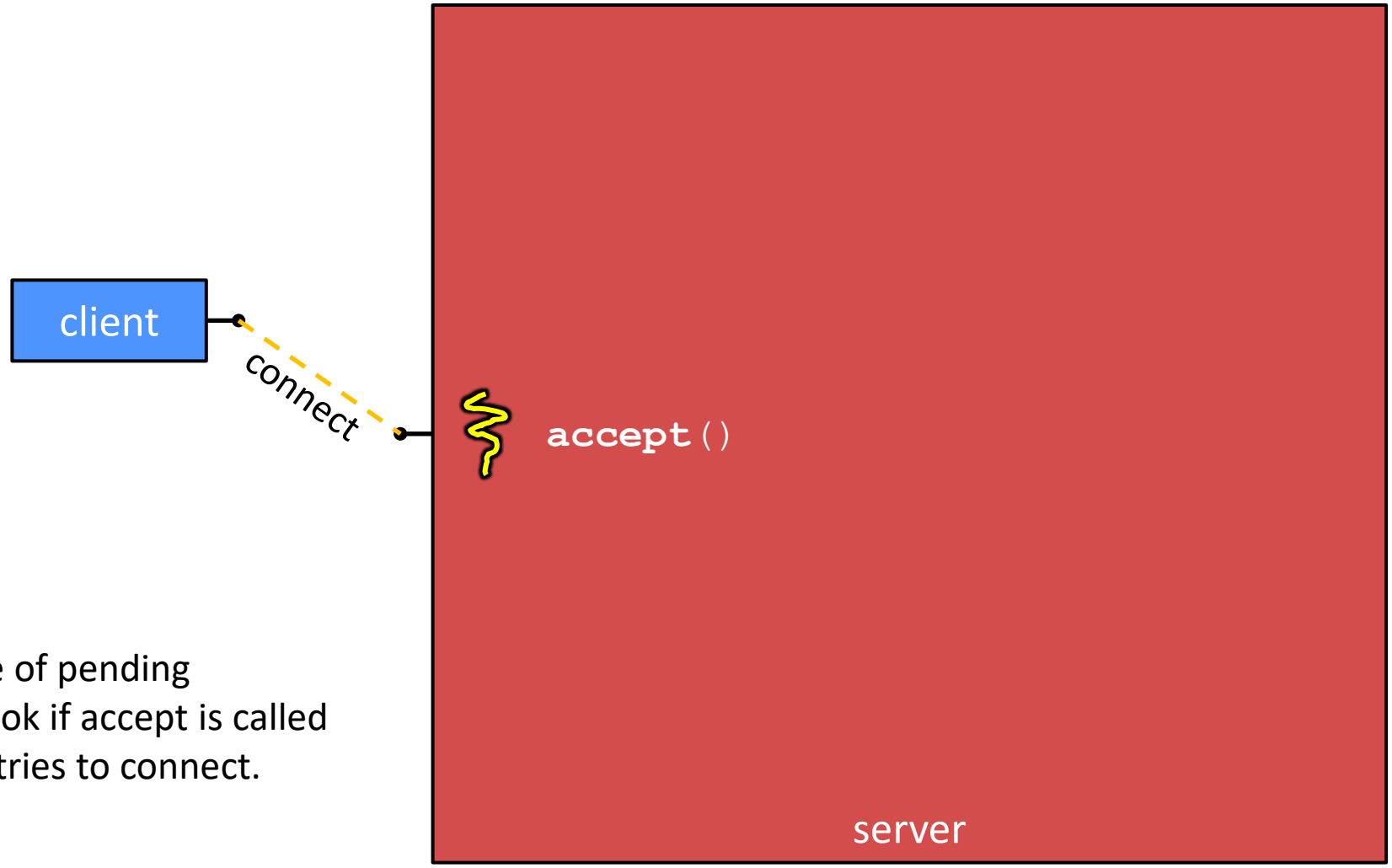
Server Pattern

- Create a listener socket bound to a local addr & port
- Waits for incoming connections
- On new connection, creates a new TCPStream socket to handle that connection

Server Listener Model

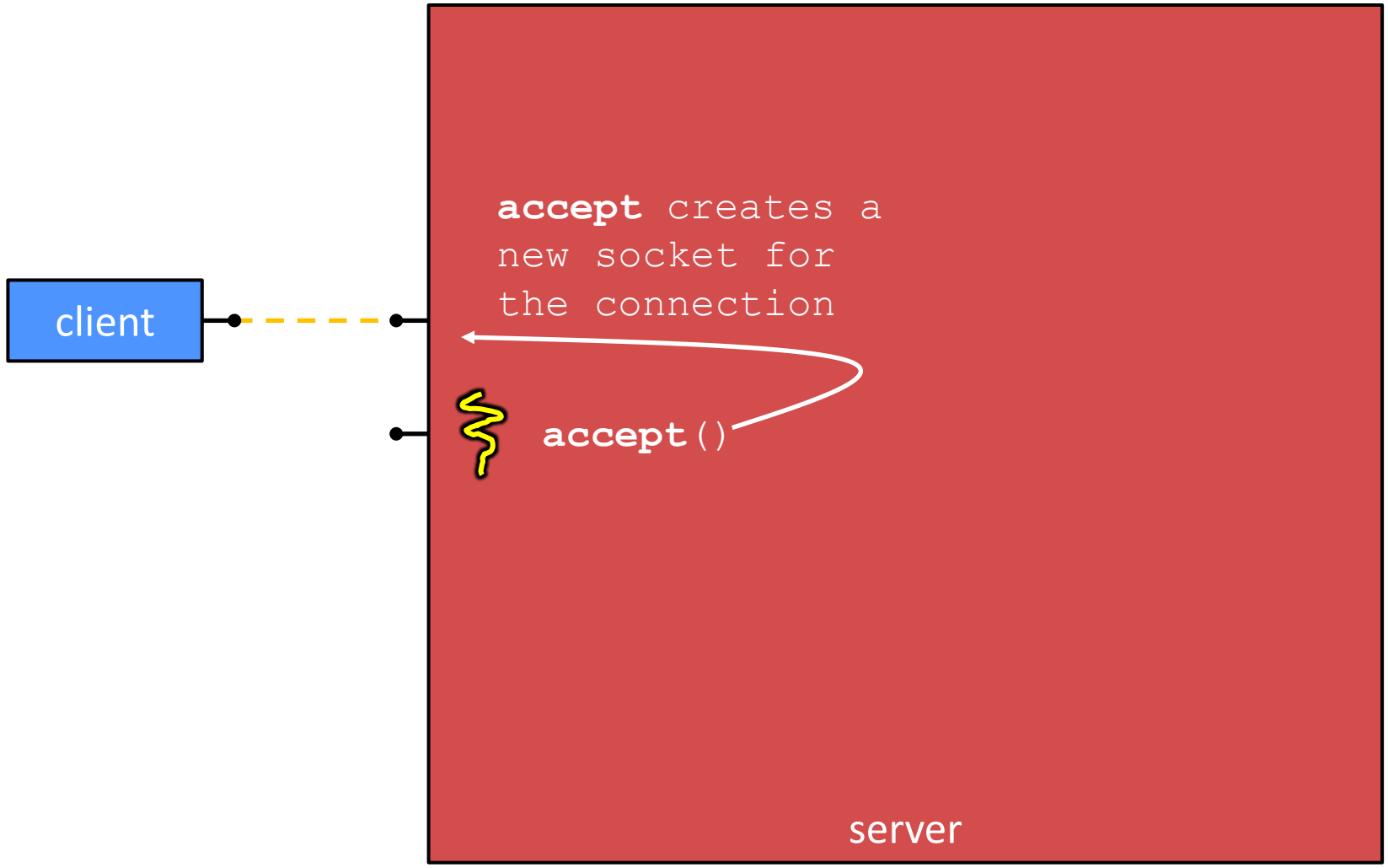


Server Listener Model



Accept maintains a queue of pending connection requests. It is ok if accept is called before or after the client tries to connect.

Server Listener Model



TCPListener

- tcp_echo_server.cpp demo / walk through

- bind()

- accept()

- The TcpListener does not read/write to the client itself!!!!

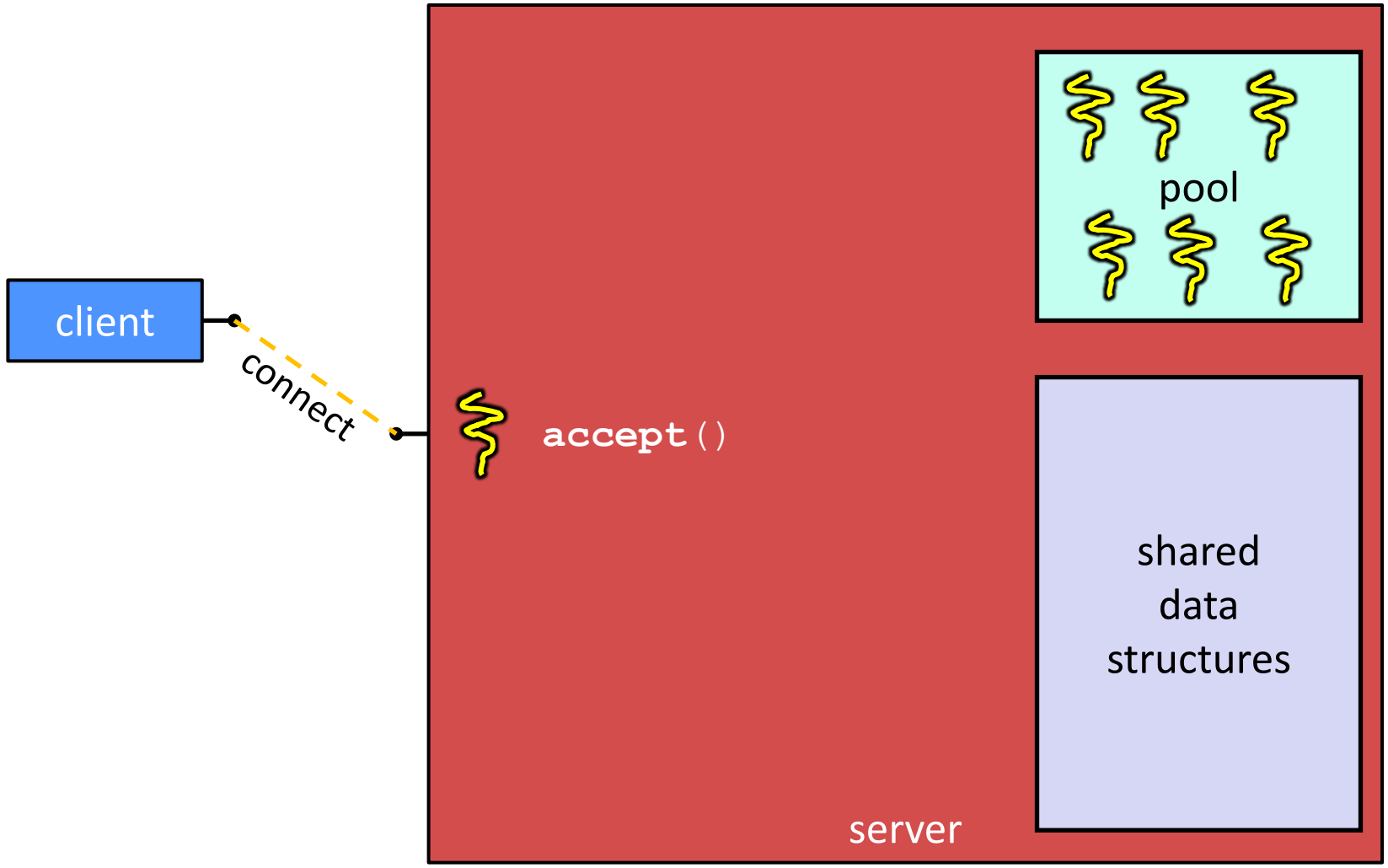
- Demo: how are sockets different than files? Maybe more similar to pipes?
  - What if client is changed to read first before sending?

Something to Note

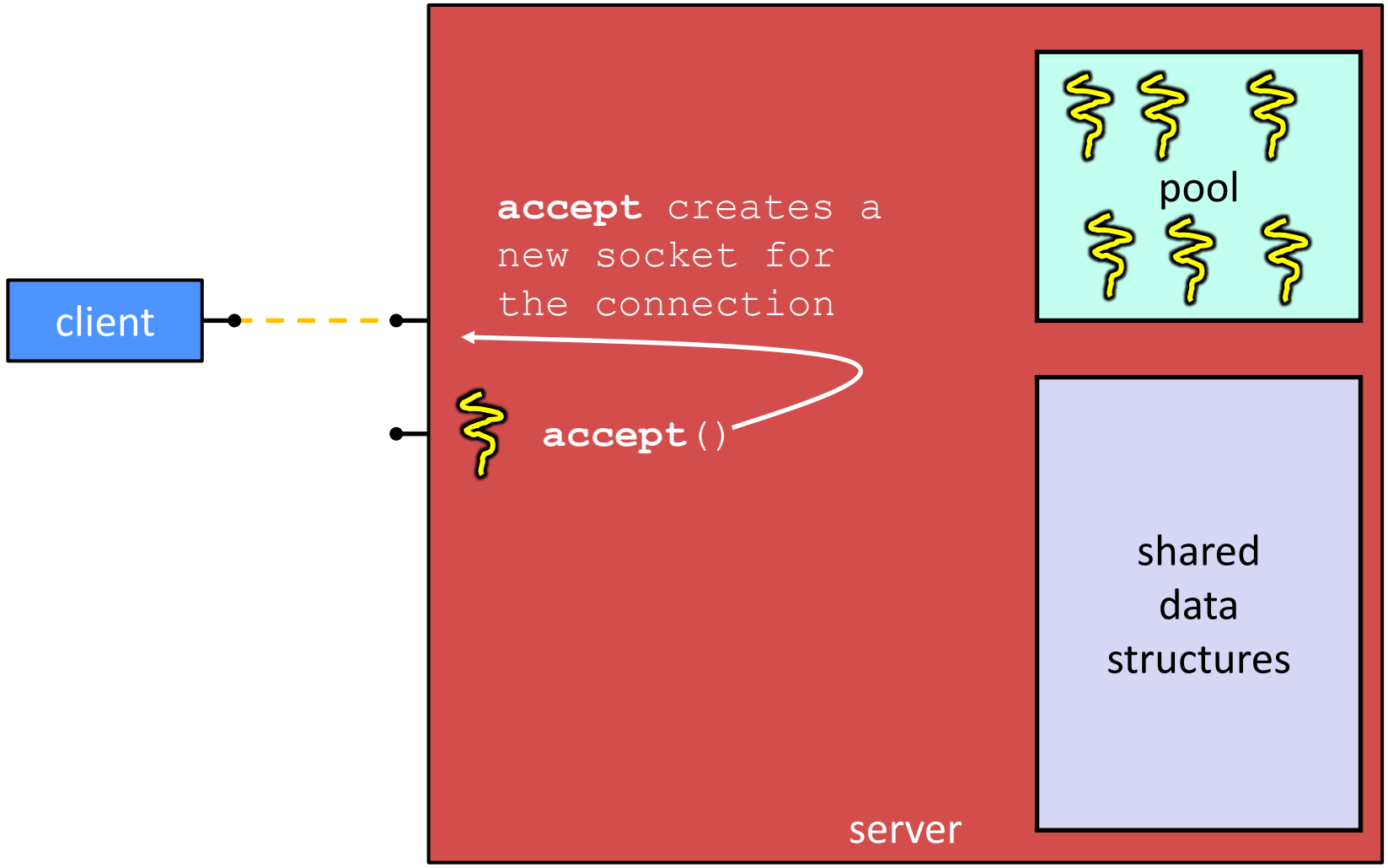
- Our server code is not concurrent
  - Single thread of execution
  - The thread blocks while waiting for the next connection
  - The thread blocks waiting for the next message from the connection

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

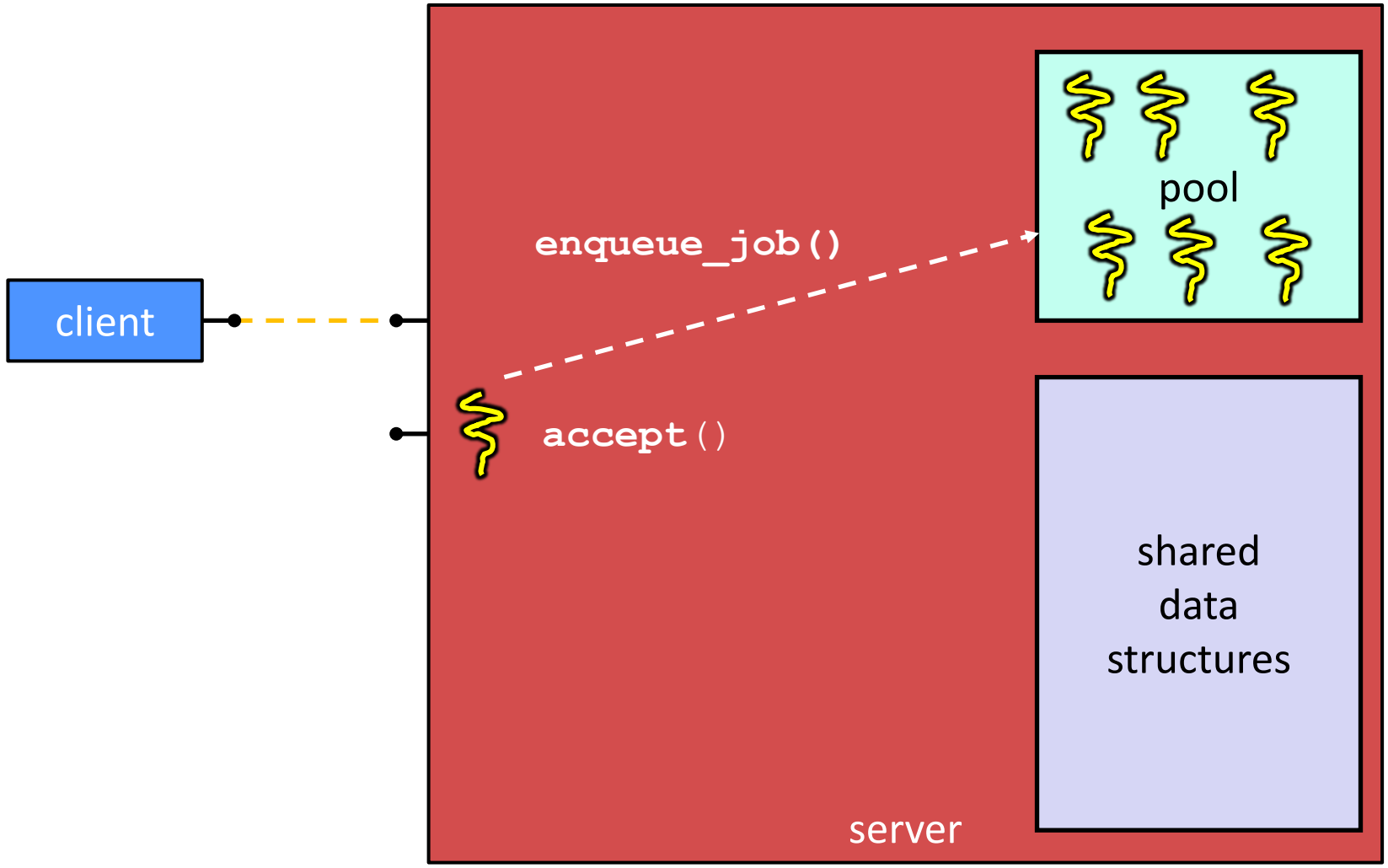
Multithreaded Server: Thread Pool



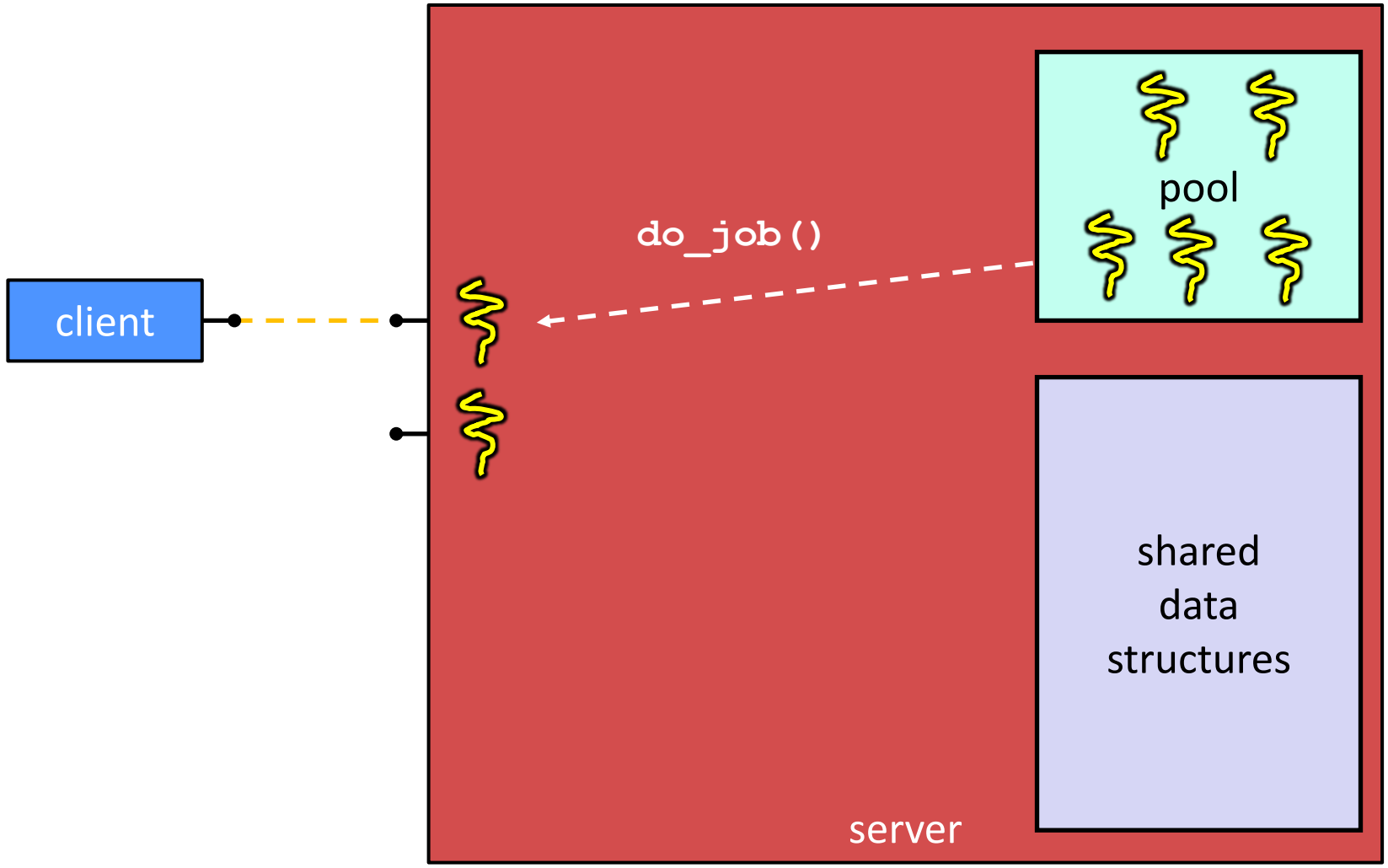
Multithreaded Server: Thread Pool



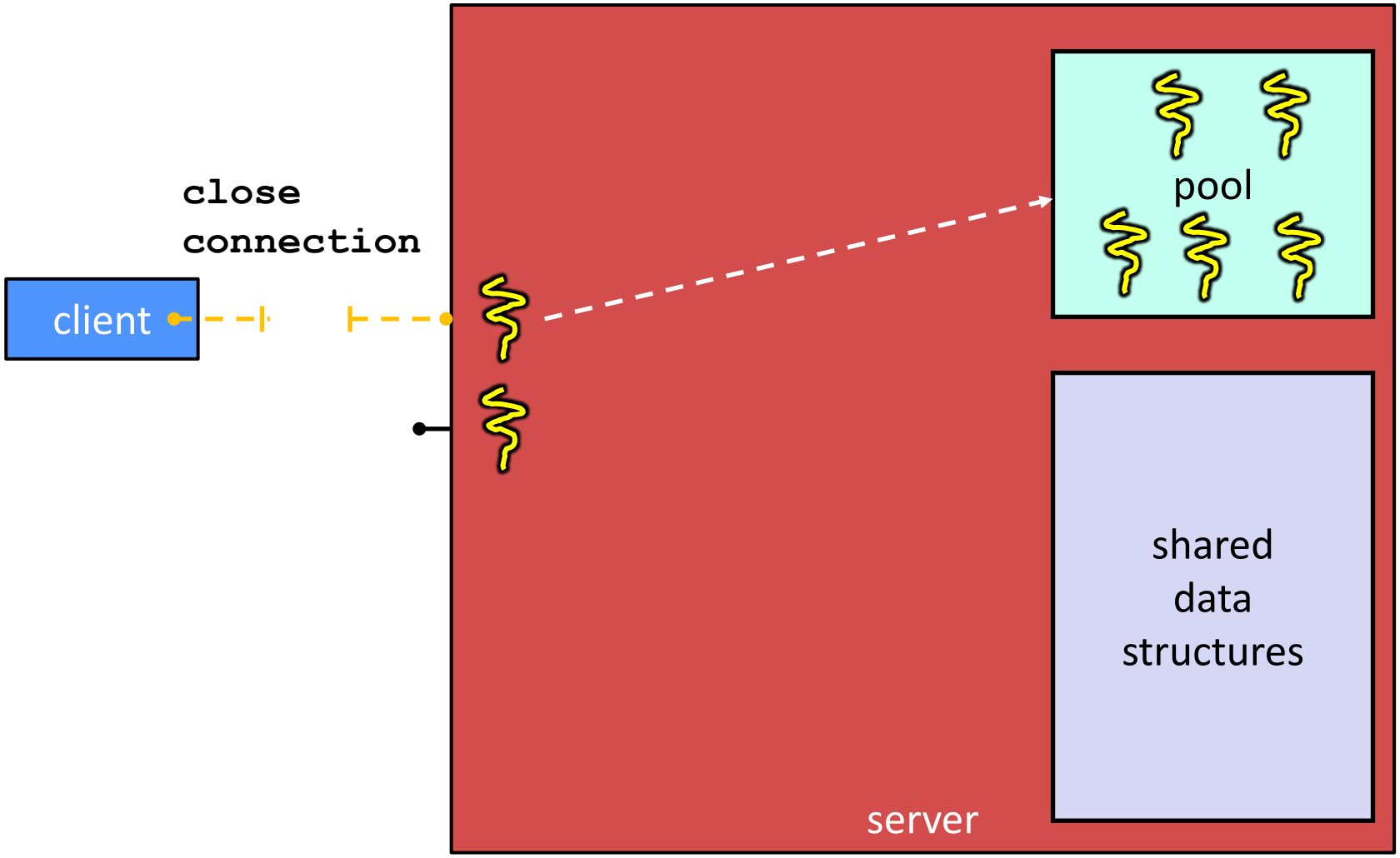
Multithreaded Server: Thread Pool



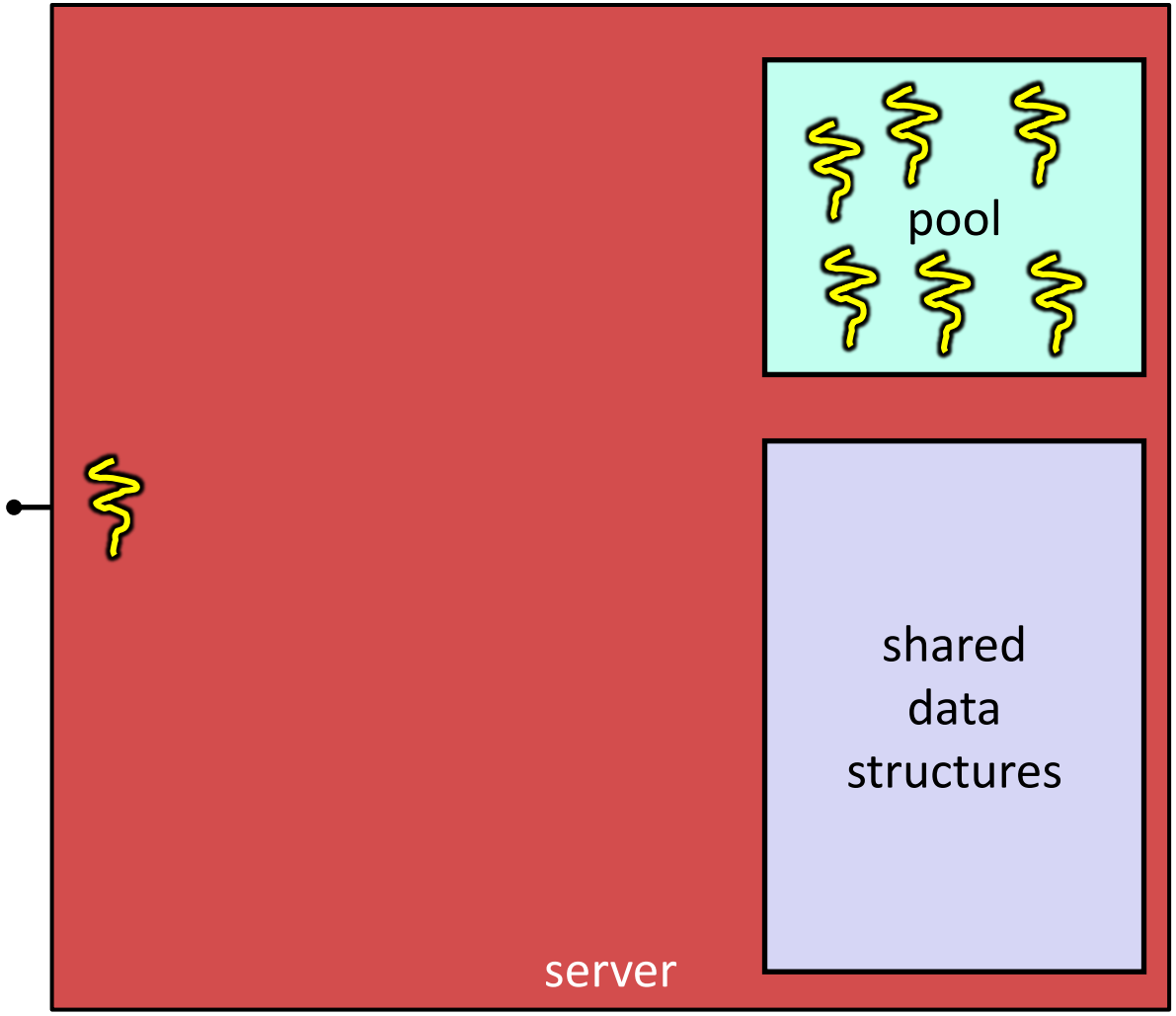
Multithreaded Server: Thread Pool



Multithreaded Server: Thread Pool

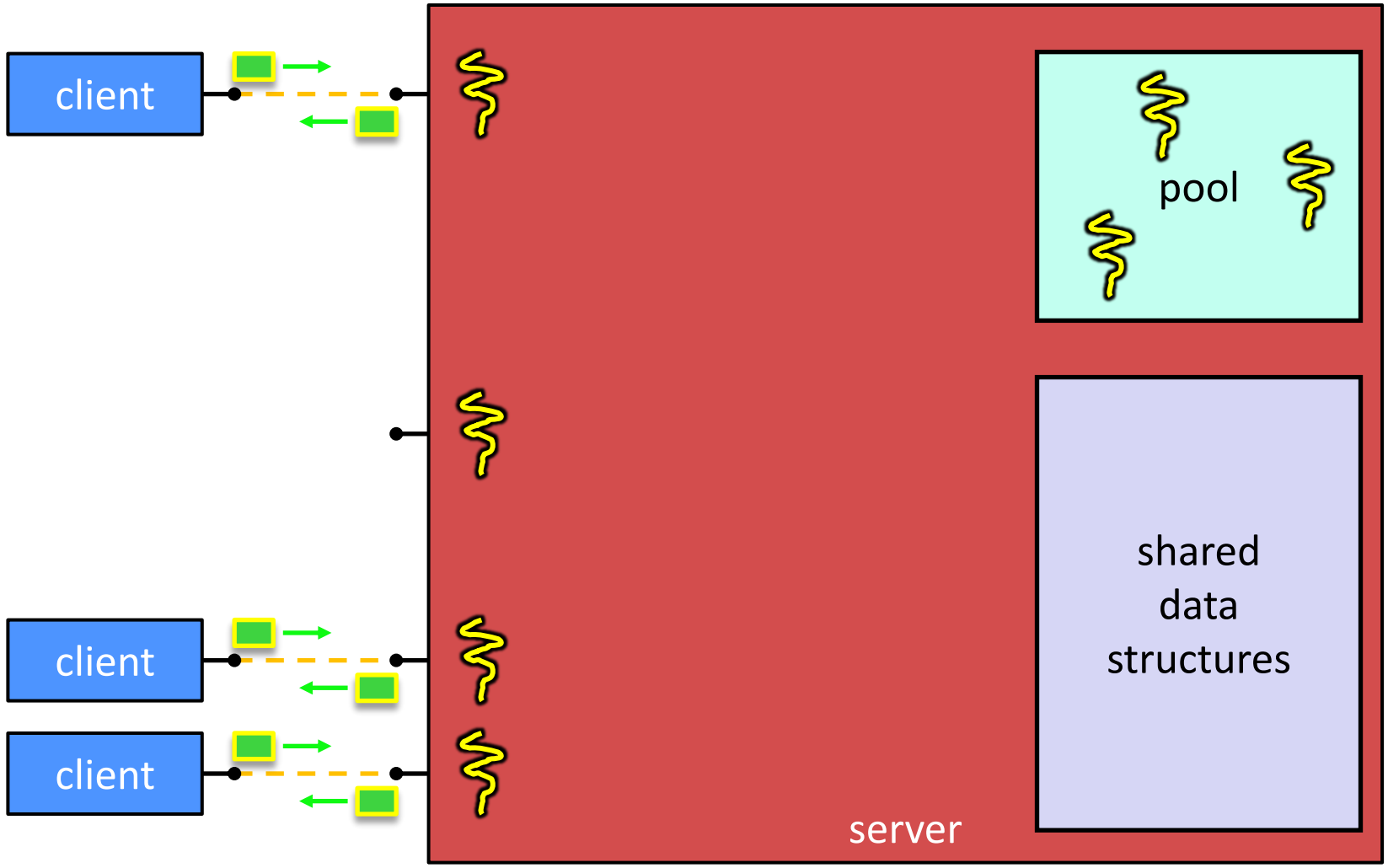


Multithreaded Server: Thread Pool



Multithreaded Server: Thread Pool

Later…



Next Lecture

- HTTP & RESTful!