Software Systems

Day 19 - Networking

DNS Cache Poisoning

- When you try to connect to a webpage like softsys.olin.edu, your machine first translates this name into an IP address like 10.0.13.12.
- The system that maps names to IP addresses is called the Domain Name Service (DNS).
- DNS lookups can require multiple packet round trips, and take a long time - so the results are cached (typically for a length of time).
- Unfortunately, this can cause some issues.
- https://youtube.com/watch?v=7MT1F0O3_Yw

Networking: Connections

- Networking involves communication between computers.
- Computers are identified by their IP addresses, but it isn't enough.
  - It would be absurd for a computer to only be able to talk to one other address at a time.
- Communication from one machine to another is thus identified by:
  - Source IP address
  - Source port
  - Destination IP address
  - Destination port
- Ports let you identify specific lines of network communication.

Networking: Connections

- Exercise:
  - You can see all bound ports on your system in /etc/services.
  - Take a look at this file (with cat /etc/services or the program of your choice).
  - What services are running on your system? How many are there?

Networking: Everything is a File

- Remember, in the UNIX paradigm, everything is a file - including network communication channels.
- Specifically, the abstraction used for networking is a socket.
- Sockets can be mapped to file descriptors, which can then be read/written like other files.

Sockets: Files, ish

- Once you have set up a socket, you have a file descriptor that you can read/write like any other file.
- But you don't create a socket with open/close like you do a file.
- Why do you think this is?

Sockets: Files, ish

- To get a file descriptor for a file, you need:
  - Pathname
  - Flags (read/write/etc)
- To get a file descriptor for a socket, you need:
  - Communication domain (protocol family, like IPv4, Bluetooth, AX.25, etc).
  - Socket type (usually stream/reliable or datagram/unreliable)
  - Protocol (specific protocol within a family, if there are multiple)
- And that's just to create the socket.

Sockets: Files, ish

- For communication, files and sockets also differ.
- With a file, you can just open it and start reading/writing.
- A client can attempt to connect to a remote address and port, but it's not guaranteed to succeed, because that would be bad.
  - Imagine being able to just write to a random port at 184.28.197.135.
- So a server receiving connections needs to BLAB:
  - *Bind* a socket to an open port.
  - *Listen* on that port for connection requests.
  - *Accept* a connection on a socket.
  - *Begin* talking to the client in a forked process.

Networking: Connection Types

- There are a fair few types of socket connections, but the two you need to know are SOCK_STREAM and SOCK_DGRAM.
- SOCK_STREAM is a connection-oriented, sequenced, reliable stream.
  - The network protocol for this is typically TCP.
  - Any data you send will be received and ordered correctly by the receiver.
  - If a part of data (a packet) doesn't make it, the sender retries.
- SOCK_DGRAM is a connectionless, unreliable stream.
  - The network protocol for this is typically UDP.
  - Packets are not guaranteed to be received in order, or at all.
  - Best-effort: no retries, so if a packet gets lost, tough luck.

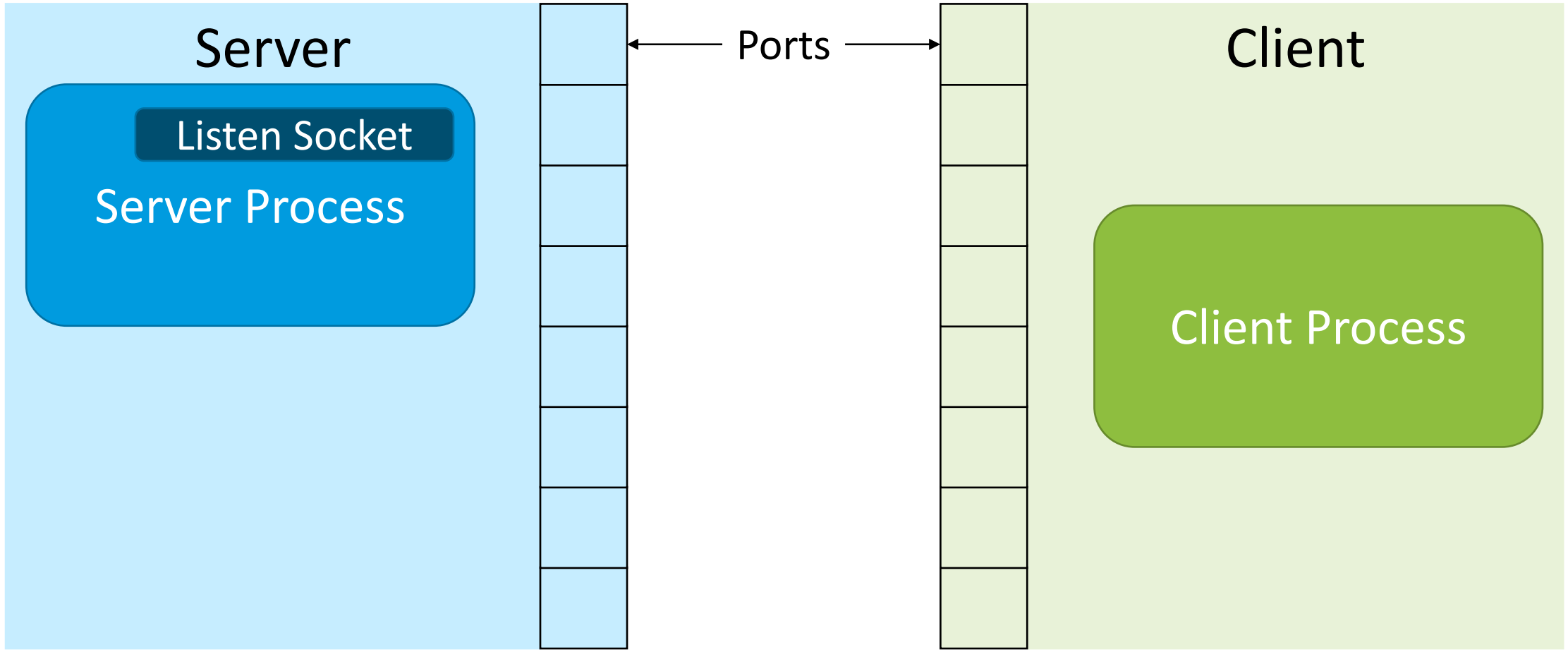
Networking: Connection Types

- Exercise:
  - In what applications would you use each type of connection?
- SOCK_STREAM is a connection-oriented, sequenced, reliable stream.
  - Any data you send will be received and ordered correctly by the receiver.
  - If a part of data (a packet) doesn't make it, the sender retries.
- SOCK_DGRAM is a connectionless, unreliable stream.
  - Packets are not guaranteed to be received in order, or at all.
  - Best-effort: no retries, so if a packet gets lost, tough luck.

Sockets: Connection Process

- A server receiving connections needs to BLAB:
  - *Bind* a socket to an open port.
  - *Listen* on that port for connection requests.
  - *Accept* a connection on a socket.
  - *Begin* talking to the client in a forked process.
- As an extended exercise, we'll walk through how to implement a very simple server and client in C.
  - This can be found in the exercises/ folder for today.

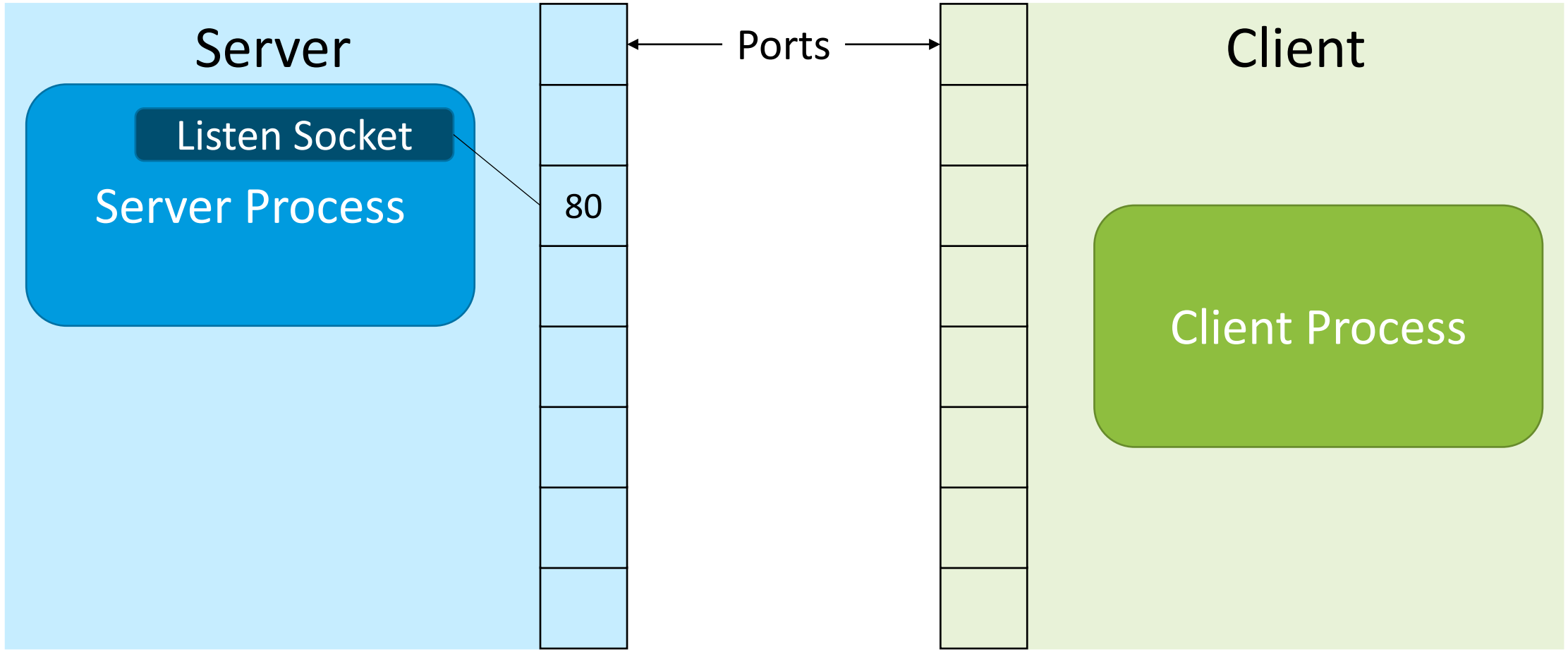
Sockets: socket()



Sockets: socket()

- Exercise:
  - In server.c, write lines of code to create a socket with the IPv4 protocol and a reliable byte stream.
  - Look up the relevant documentation with `man socket`.
  - If you have time, add an error message with a call to `error()` (see the manpage for more details).

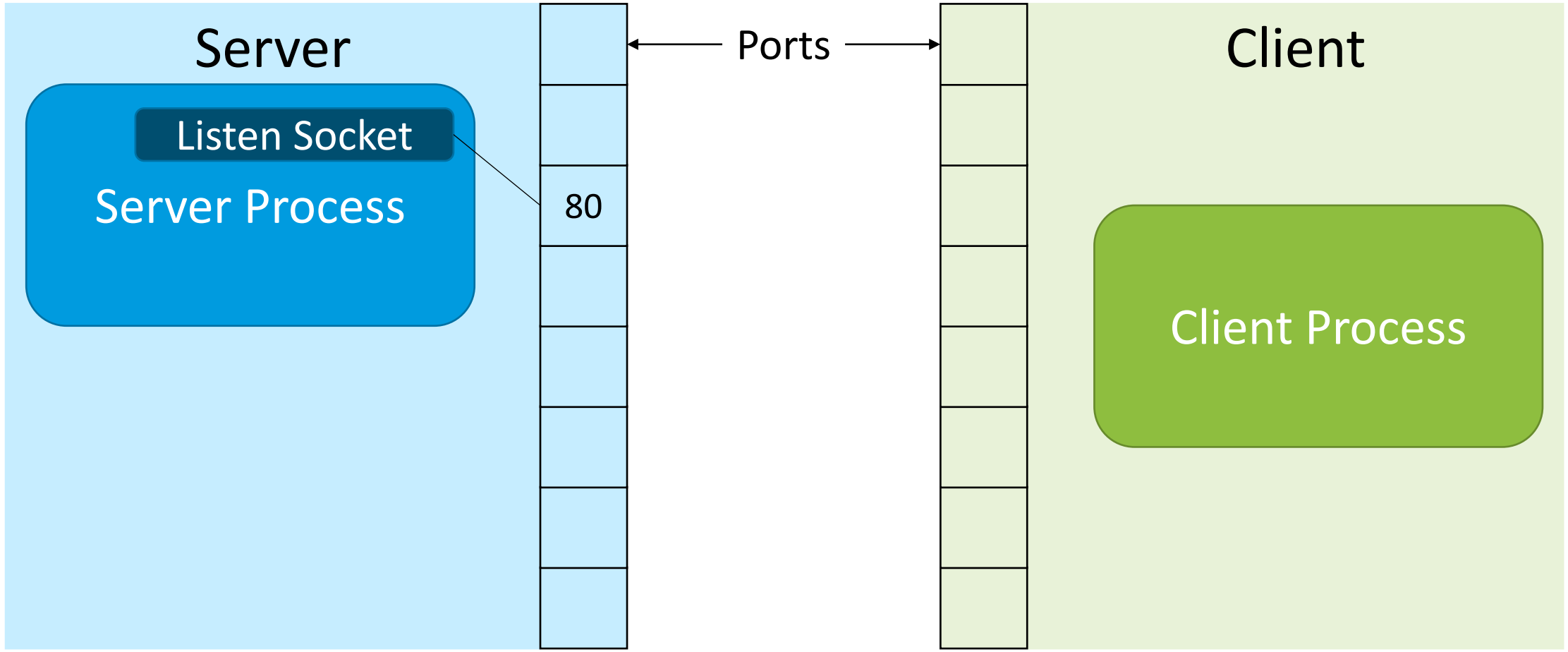
Sockets: bind()



Sockets: bind()

- Exercise:
  - In server.c, write lines of code to set up a sockaddr_in struct with the appropriate values and bind it to the socket.
  - Look up the relevant documentation with `man ip` and `man bind`.
  - The port number and address can be found in constants.h/constants.c.
  - If you have time, add an error message with a call to `error()` (see the manpage for more details).

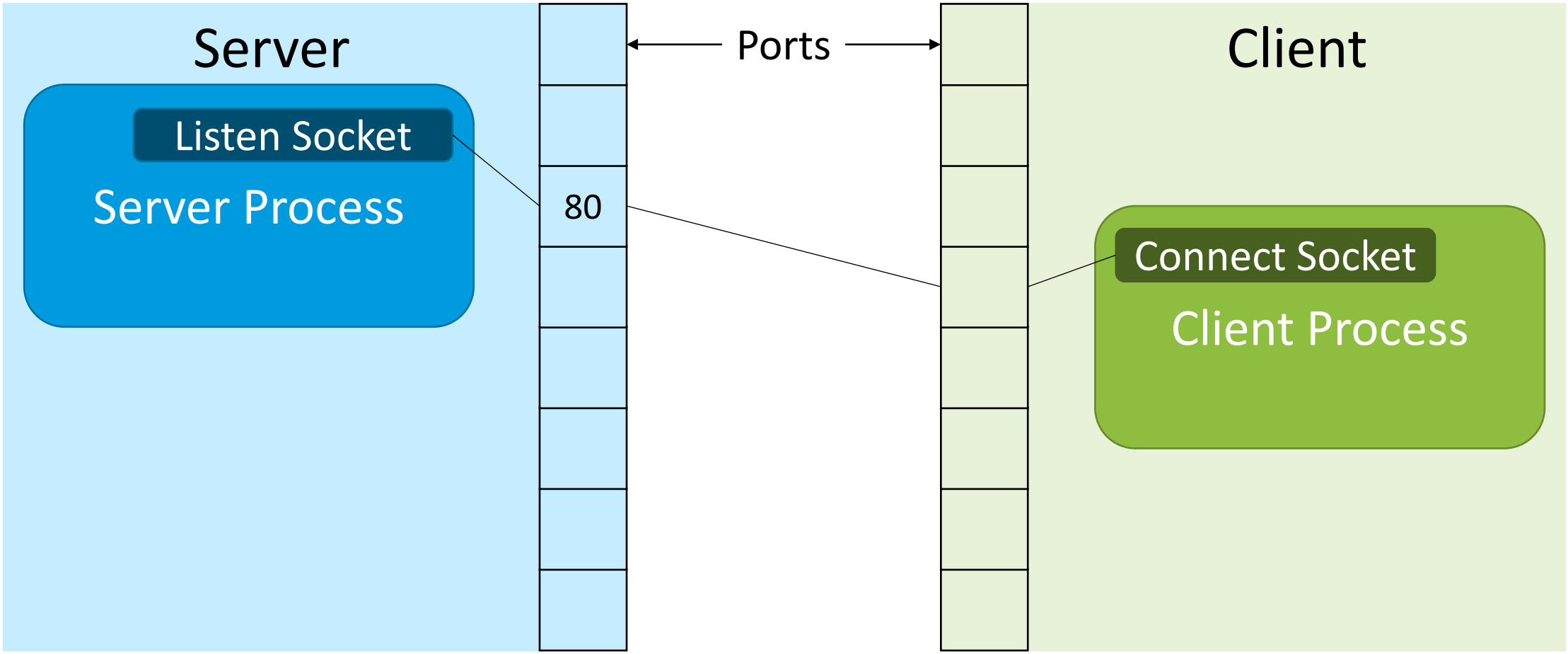
Sockets: listen()



Sockets: listen()

- Exercise:
  - In server.c, write lines of code to listen on the bound socket.
  - Look up the relevant documentation with `man listen`.
  - The backlog size can be found in constants.h.
  - If you have time, add an error message with a call to `error()` (see the manpage for more details).

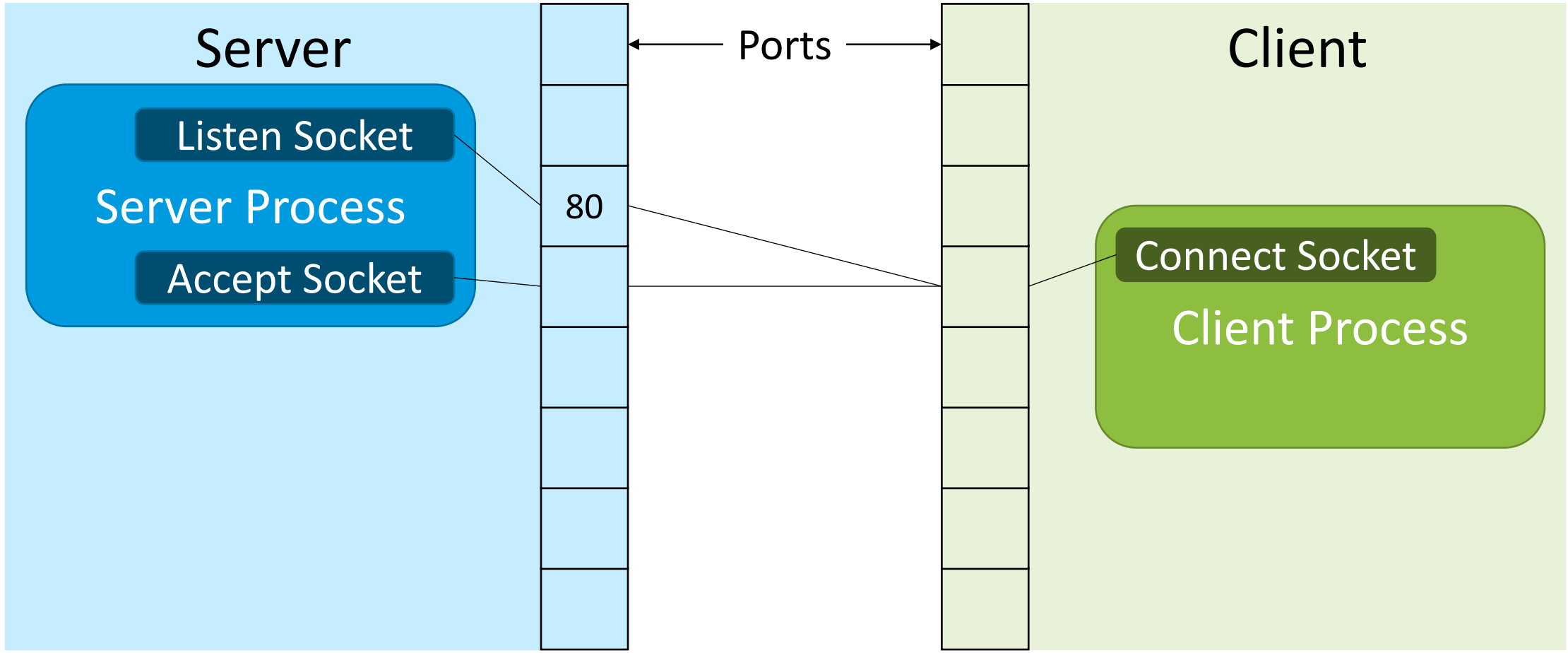
Sockets: connect()



Sockets: connect()

- Exercise:
  - In client.c, write lines of code to set up a socket and sockaddr_in struct with the same values as before, then connect to the server on this socket.
  - Look up the relevant documentation with `man ip` and `man connect`.
  - The port number and address can be found in constants.h/constants.c.
  - If you have time, add an error message with a call to `error()` (see the manpage for more details).

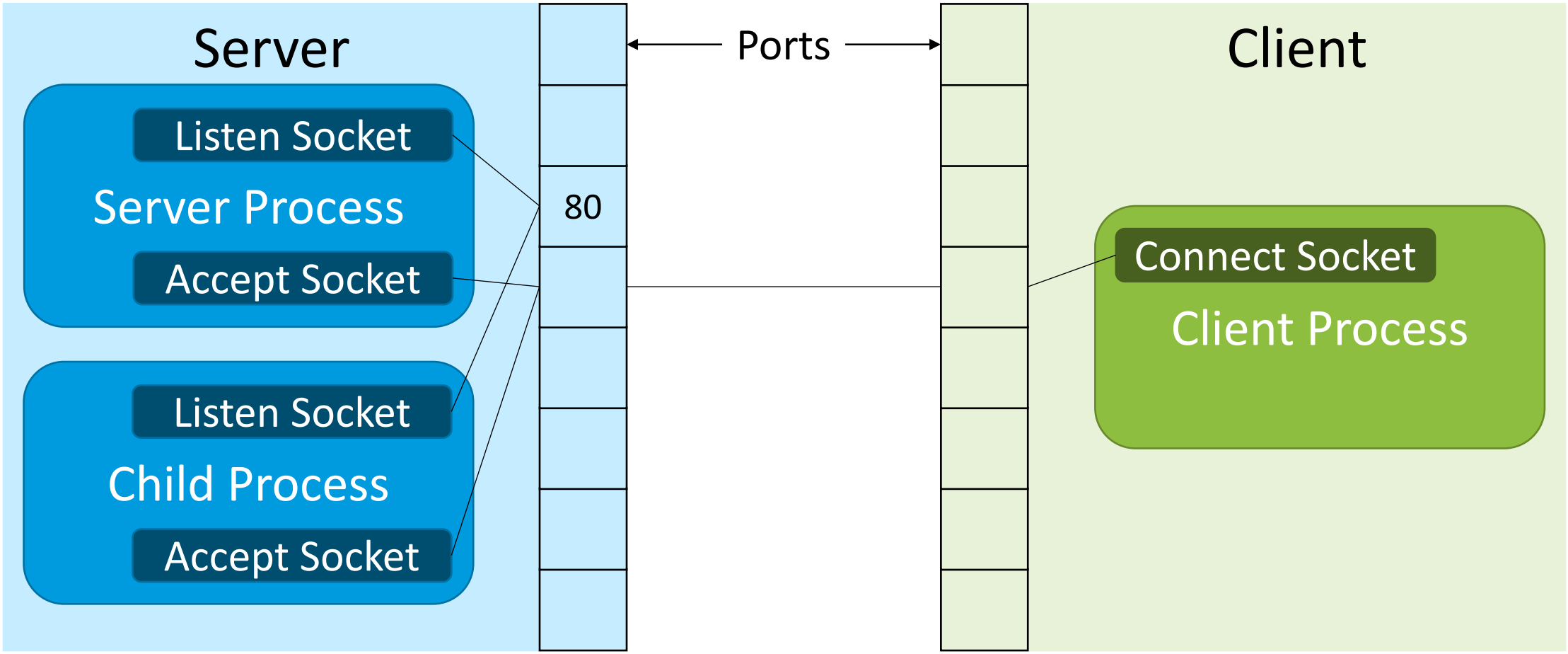
Sockets: accept()



Sockets: accept()

- Exercise:
  - In server.c, write lines of code to accept a connection on the socket.
  - Look up the relevant documentation with `man accept`.
  - If you have time, add an error message with a call to `error()` (see the manpage for more details).
  - If successful, accept() returns a new socket descriptor. Why does it create a new socket rather than reusing the one it was listening on?

Sockets Overview: fork()



Sockets: fork()

- Exercise:
  - In server.c, write lines of code to fork the process after accepting a connection.
  - The child process should print "Server Hello World!" to standard output, immediately close the new socket descriptor and return 0.
  - Look up the relevant documentation with `man accept`.

Sockets Overview: Communication