CSE333 Systems Programming

Networking: Client Sockets

Instructors:

Naomi Alterman

Teaching Assistants:

Ann Baturytski	Barbora Buzkova	Mendel Carroll
Camden Harris	Hannah Hempstead	Rishabh Jain
Irene Lau	Jonathan Nister	Advay Patil
Aviel Wood	Angela Wu	Jennifer Xu

Administrivia

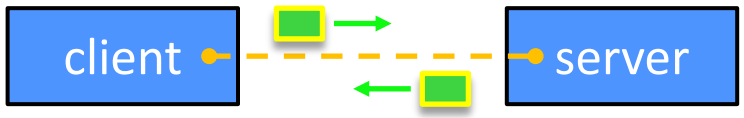
- HW3 due this Thursday
- Ex 10 out today, due in a week (next Wednesday 5/27)
  - Client that sends a message over a stream socket and then exits
- Ex 11 out Friday, due in a week (next Friday 5/29)
  - Server that receives messages from a stream socket, prints it to the screen, and then exits

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:

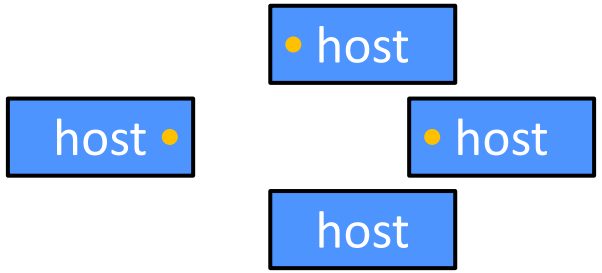


3) Close connection:

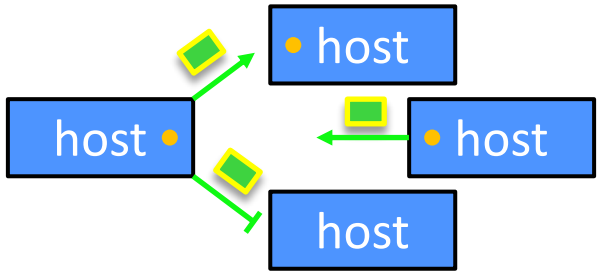
Datagram Sockets

- 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:

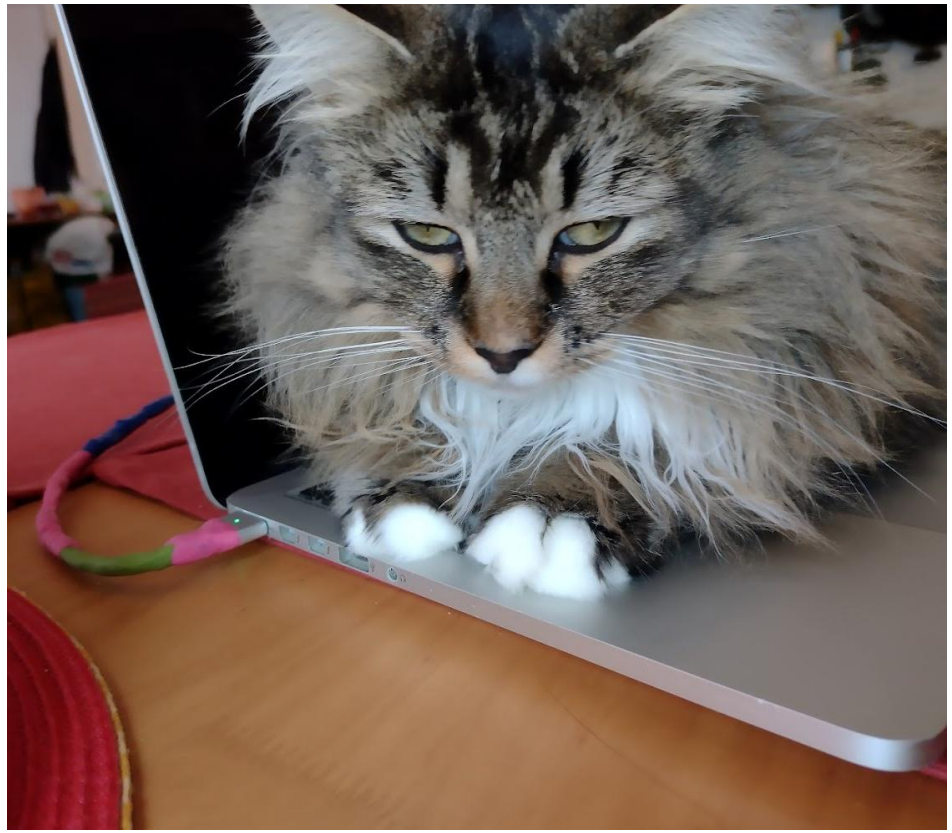


2) Communicate:



Our networking multitool

- Netcat ("`nc`")
  - Does what "`cat`" does, but over the net
  - Specifically:
    - Opens a socket
    - Writes bytes from `stdin` to the socket
    - Prints bytes read from the socket to `stdout`
- Usage (TCP):
  - Client: `nc HOST PORT`
  - Server: `nc -k -l PORT`
  - To exit either of them: `Ctrl+C`



Our mission this week: send bytes over TCP using sockets

- 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
  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 1: Figure Out IP Address and Port

IPv4 Network Addresses

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

- Each of those bytes has information about "which part" of the network the machine is on
  - Read left to right, the machine's location gets more "network local"
  - IPs with the same prefix are on the same "subnet," which represents ***some abstratct form of locality***

IPv6 Network Addresses

- An IPv6 address is a **16-byte** tuple
  - Typically written in "hextets" (groups of 4 hex digits)
    - Can omit leading zeros in hextets
    - 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

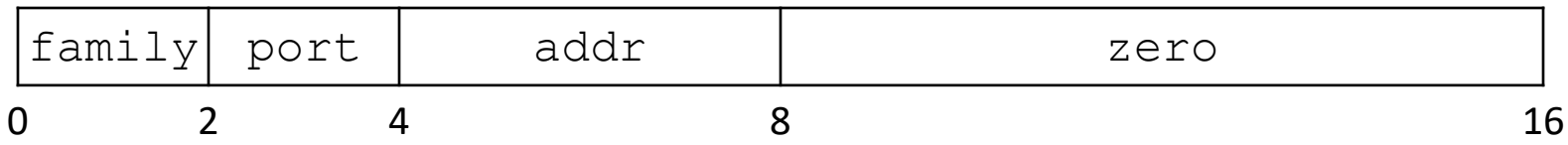
- Structures, constants, and helper functions available in:
  `#include <arpa/inet.h>`
- Addresses stored in network byte order (big endian)
- Converting between host and network byte orders:
  - `uint32_t htonl(uint32_t hostlong);`
  - `uint32_t ntohl(uint32_t netlong);`
    - 'h' for host byte order and 'n' for network byte order
    - Also versions with 's' for short (`uint16_t`)
- 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
    - (Crude way to fake inheritance-like behavior in languages without it)

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
};
```

`struct sockaddr_in`:

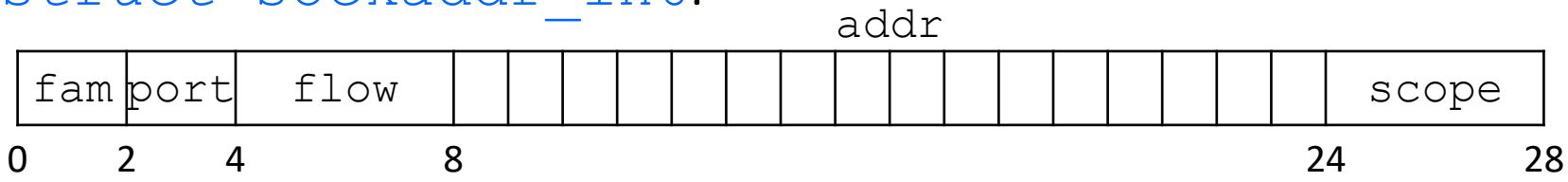


IPv6 Address Structures

```
// 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
};
```

`struct sockaddr_in6`:



Generic Address Structures

```
// 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];
};
```

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

Address Conversion

- `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

- ```
  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`

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;

  return EXIT_SUCCESS;
}
```
```

Poll Everywhere

pollev.com/naomila

Write out the bytes of the following `struct sockaddr_in` in memory:

- **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`**

0								
8								

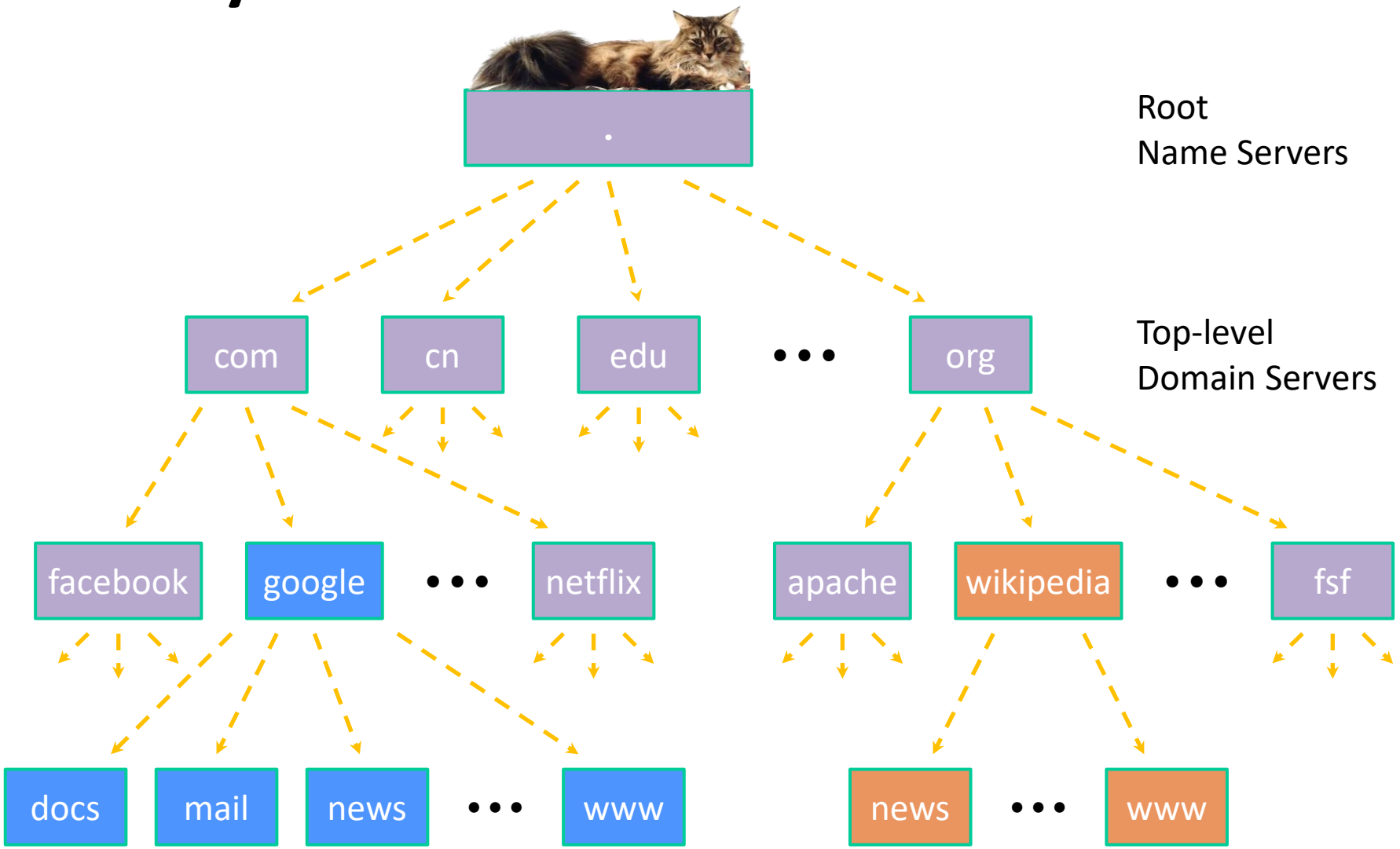
```
struct in_addr {
  uint32_t s_addr;
};

struct sockaddr_in {
  sa_family_t    sin_family;
  in_port_t      sin_port;
  struct in_addr sin_addr;
  unsigned char  sin_zero[8];
};
```

Domain Name System

- People tend to go to “google.com”, not 142.251.152.119
  - This is a **domain name** in the **DNS** (“Domain Name System”), a world-wide, distributed, decentralized database of names
  - Domain names serve as “nicknames” for IP addresses (and other things)
  - Figuring out the IP address for a domain name is called **resolving** that name
  - And that’s hard, because, well…

DNS Hierarchy



Resolving DNS names

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

- In C: …it's complicated :\
  - A given DNS name can have many IP addresses
  - Many different IP addresses can map to the same DNS name
  - A DNS lookup may require interacting with many DNS servers

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);S
```

getaddrinfo

- **`getaddrinfo()` arguments:**
  - `hostname` – domain name or IP address string
  - `service` – port # (*e.g.* `"80"`), service name (*e.g.* `"www"`) or `NULL`/`nullptr`
  - `hints` (and `res` output data) -

```
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

Step 2: Creating a 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) {
     std::cerr << strerror(errno) << std::endl;
     return EXIT_FAILURE;
  }
  close(socket_fd);
  return EXIT_SUCCESS;
}
```

Step 3: Connecting to a Server

Step 3: Connect to the Server

- The `connect()` system call establishes a connection to the remote server:

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

  - `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
  - The network stack within the OS will communicate with the remote host to establish a TCP connection to it
    - This involves ~2 *round trips* across the network

Full Connect Example

- See `connect.cc`

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

// 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: Exchanging data

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
  - This might cause *deadlock*!
  - Can `read()` return `0`?

Step 4: `write()`

- `write()` enqueues 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*

Read/Write Example

```
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;
}
```

- See `sendreceive.cc`
  - Demo

Step 5: Cleaning up

Step 5: `close()`

- `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