Client-side and Server-side Network Programming

CSE 333 Winter 2023

Instructor: Justin Hsia

Teaching Assistants:

Adina Tung
Danny Agustinus
Edward Zhang
James Froelich
Lahari Nidadavolu
Mitchell Levy
Noa Ferman
Patrick Ho
Paul Han
Saket Gollapudi
Sara Deutscher
Tim Mandzyuk
Timmy Yang
Wei Wu
Yiqing Wang
Zhuochun Liu

Relevant Course Information

- Homework 3 was due yesterday
- Homework 4 released today
  - Due last Thursday of quarter (3/9)
  - Can still use 2 late days for hw4 (hard deadline of 3/12)
  - Demo next lecture

- Exercise 10 released today, due Wednesday (3/1)
  - Client-side TCP connection
- Exercise 11 released today, due Friday (3/3)
  - Server-side programming

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);   // close when done
  return EXIT_SUCCESS;
}
```

Step 3: Connect to the Server

- The **`connect`**() system call establishes a connection to a remote host
  - usually: `struct sockaddr_storage ss;`
    `reinterpret_cast<sockaddr*>(&ss)`

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

    - `sockfd`: Socket file description from Step 2 (`socket()`)
    - `addr` and `addrlen`: Usually from one of the address structures returned by **`getaddrinfo`** in Step 1 (DNS lookup) (`getaddrinfo()`, `struct addrinfo`)
    - 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
    - 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);

// 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;
}
```

// does the getaddrinfo() call

Poll Everywhere

pollev.com/cse333

How do we *error* check `read()` and `write()`?

return # of bytes read/written

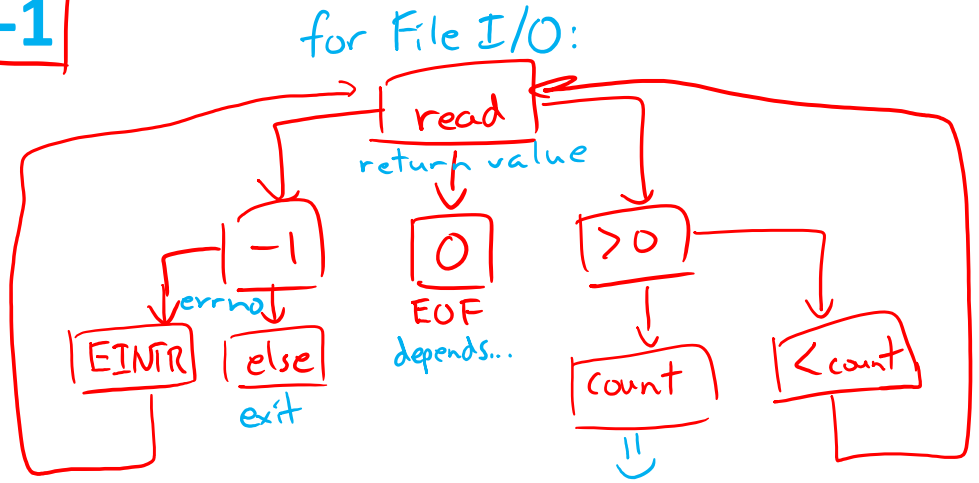
A. `ferror()` for fread()/fwrite()

B. Return value less than expected happens, but not on error

C. Return value of 0 or NULL valid return value (means EOF for read())

D. Return value of -1

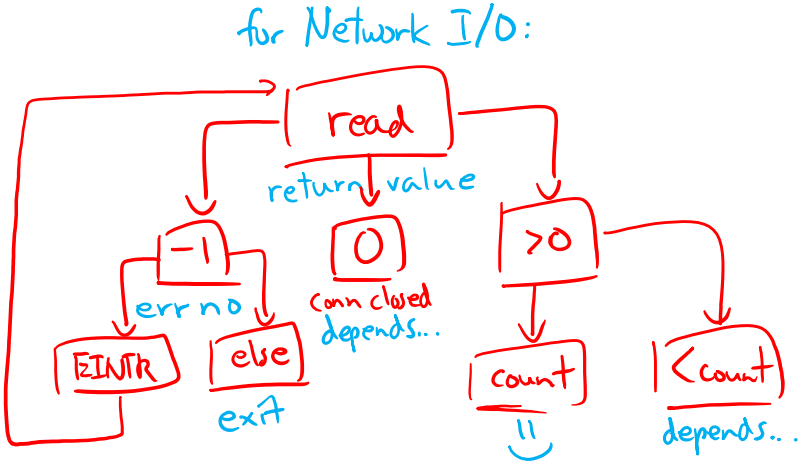
E. We're lost…



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*? server & client have no data to read, but both call read()
  - Can `read()` return 0? Yes, if connection is closed



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*

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()`

- `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

- Pretty similar to clients, but with additional steps:

Analogy:

1) Figure out the IP address and port on which to listen — ① find a location/buy land
2) Create a socket — ② build the structure
3) `bind()` the socket to the address(es) and port — ③ prep work & advertising
4) Tell the socket to `listen()` for incoming clients — ④ open the door (customers queue)
5) `accept()` a client connection — ⑤ "next customer in line!"
6) `read()` and `write()` to that connection — ⑥ transaction occurs
7) `close()` the client socket — ⑦ customer leaves

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

from socket()

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

from getaddrinfo()

- Looks nearly identical to **`connect()`**!
- Returns **`0`** on success, **`-1`** on error

- 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 ☹

Extra Exercise #1

- Write a program that:
  - Reads DNS names, one per line, from `stdin`
  - Translates each name to one or more IP addresses
  - Prints out each IP address to `stdout`, one per line