Server-Side Networking

CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:

Dao Yi	Farrell Fileas	Lukas Joswiak
Nathan Lipiarski	Renshu Gu	Travis McGaha
Yibo Cao	Yifan Bai	Yifan Xu

Poll Everywhere

pollev.com/cse333

About how long did Exercise 15 take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I didn't finish / I prefer not to say

Administrivia

- Exercise 16 out today
  -  Second-to-last exercise 
- HW4 posted
  - Due last Thursday of the quarter (12/5)
  - Only 1 late day allowed for HW4 (hard deadline of 12/6)
- Canvas updated with late days and HW1 + HW2 grades
  - Let Hannah know if you can't access

Lecture Outline

- **Roadmap**
- Server-side Networking
  - … ?

Review: Client-side Networking

- Step 1: Figure out the IP/Port
- Step 2: Create a Socket
- Step 3: Connect the Socket
- Step 4: `read`() and `write`() Data
- Step 5: Close the Socket

Socket API: Server TCP Connection

- Similar structure to clients:
  1) Figure out the IP address and port on which to listen
  2) Create a socket
  3) `bind()` the socket to the address(es) and port
  4) Tell the socket to `listen()` for incoming clients
  5) In a loop: `accept()` a client connection
  6) In a loop: `read()` and `write()` to that connection
  7) `close()` the client socket

analogue to connect() in client-side networking

server's read/write loop

this is where you'll plug in your application logic

Server Networking: Lecture Objectives

- Know what each of the 7 steps of server-side networking does and why it is important

- *Non-objective*: be able to write server-side networking code from scratch after this lecture
  - You'll have plenty of code to practice with at home ☺
  - Copy and paste is not necessarily a bad thing here – but make sure you *understand* it well enough to modify it if you have to

Lecture Outline

- Roadmap
- **Server-side Networking:**
  - Figure out the IP address / port
  - Create a socket
  - **`bind`**`()` the socket
  - **`listen`**`()` for incoming clients
  - In a loop: ← this loop is the only place we have network I/O
    - **`accept`**`()` a client connection
    - **`read`**`()` and **`write`**`()` to that connection
  - **`close`**`()` the client socket

Servers != Clients

- 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/flags

best practice

  - Can request listen on all local IP addresses by passing `NULL`/`nullptr` as `hostname` and setting `AI_PASSIVE` in `hints.ai_flags`
    - Effect is to use address `0.0.0.0` (IPv4) or `::` (IPv6)

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

(no I/O yet)

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

(still no I/O)

- `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

Pseudocode Time

- Assume we have set up `struct addrinfo hints` to get both IPv4 and IPv6 addresses
  - Write pseudocode to bind to and listen on the first socket that works

- Pieces you can use:
  - `retval = getaddrinfo(..., &res);`
  - `freeaddrinfo(res);`
  - `fd = socket(...);`
  - `retval = bind(fd, ...);`
  - `retval = listen(fd, SOMAXCONN);`
  - `close(fd);`

Demo #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` is ALSO a normal parameter: initially 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

Demo #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:
  - Creates a listening socket that accepts connections from clients
  - Reads a line of text from the client
  - Parses the line of text as a DNS name
  - Does a DNS lookup on the name
  - Writes back to the client the list of IP addresses associated with the DNS name
  - Closes the connection to the client