Lab 10 - Networking

Due: November 24, 2019 at 10:00 PM

1 Introduction **1**

2 Assignment **2**

3 TCP Connection **3**

3.1 Data to the Server 3

3.2 Data from the Server 4

4 UDP Connection **6**

5 User Interaction **6**

6 Using select() **6**

7 Server and Testing **7**

8 Getting Started **8**

9 Tips **8**

10 Finishing Up **8**

1 Introduction

Daphne the Dolphin wants to use here sonar capabilities to pick up radio signals and listen to her favorite pop singer. She picked up a signal for a great radio server but couldn't find the corresponding client! She has now turned to you, an experienced radio communicator, with writing the radio client code so she can swim to her favorite singer Adam Levine.

Note: All of the labs in CS33 will be partner labs. Remember to fill out the partner form--you must work with a different partner for each lab and can either choose your partner or go random.

2 Assignment

Before you begin working on the project, make sure you review the lecture materials on network programming, *particularly the code demos provided on the website*, as well as any relevant `man` pages for new or unfamiliar functions. This lab is often the first exposure students have to network programming, and working with a guide will be very helpful.

For this assignment, you will be implementing a client for an internet radio station server. The radio station server maintains several radio stations, each of which loops a single song continuously. After starting up, the radio station server streams information about a station and the mp3 encoding of the song being played on that station to each client that connects to it. You will write one such client in this lab.

A radio client communicates with the radio station server using two ports and two different *protocols*. One port communicates using TCP, handling control data for the server. The other port communicates using UDP, receiving song data from the server.

There are two main reasons you will use UDP in this project to deliver music data:

- Unlike TCP, which guarantees packet delivery, UDP does not guarantee that the packets sent will reach their destination. UDP receivers send no acknowledgements to the sender that the data has been received. If a packet fails to be successfully delivered, it will simply be skipped over; the server never checks that the clients get the data.
- Because there is no error checking (recall that UDP does not check for missing packets or duplicates of received packets), UDP can send packets more quickly than TCP. This makes UDP well suited to real-time applications such as streaming music.

Your client will manage input and output from the two ports passed to it, as well as from **`stdin`**, using a **`select()`** event loop. Be sure to set up your sockets with the correct protocols.

When it is complete, your radio client should take three arguments:

```
./client <hostname> <serverport> <udpport>
```

`hostname` is the name of the machine that is running the radio station server. Host names are the names of servers, like **`www.facebook.com`** or **`cs.brown.edu`**. If you are working from a department machine, you can find out the hostname of your computer by looking at its name; for instance, if you were working from **`cslab5e`**, your hostname would be **`cslab5e.cs.brown.edu`**. If you are running the server on the same machine as you are running the client, you can use localhost as your host name. **`localhost`** is a reserved hostname that always represents the computer you are currently using.

`serverport` and **`udpport`** are the ports you will be using to connect to the server and for the server to send your client data, respectively. You may choose these to be whatever you would like, other than the ports 0-1023, which are reserved for system use, and will cause an error if you try to bind a socket to them (for instance, web servers generally use port 80 and ssh servers use port 22 by default). **`serverport`** should match what you choose when setting up the server, and **`udpport`** can be anything not in the reserved range.

3 TCP Connection

The TCP part of your client handles the server control data. It will both send data to the server and receive data from the server.

3.1 Data to the Server

Your client should be able to send the following information to the server through its TCP port:

For `Hello` you will want to send:
`uint8_t command_type = 0;`
 	
and
`uint16_t udp_port;`
	`command_type` indicates to the server which type of command it is being sent; in this case, 0 corresponds to a **`Hello`** command. `udp_port` contains your client's UDP port, to which the server will write music data.
	
	Your client should send this command exactly once, when it first connects to the server, and the information should be sent as a sequence of bytes. For instance, if the UDP port is 3333, the client should first send the number 0 as a **`uint8_t`** to the server, followed by the number 3333 as a **`uint16_t`**.

For `Set Station` you will want to send:
`uint8_t command_type = 1;`
 	
and
`uint16_t station_number;`
	This command changes the station that your client is listening to. As it did with the **`Hello`** command, `command_type` indicates to the radio station server what type of command it is receiving; 1 indicates the **`Set Station`** command. **`station_number`** indicates the new station. Note that the station numbers are zero-indexed.
	
	Similar to the **`Hello`** command, send this command to the server first by writing 1 as a **`uint8_t`**, followed by the station number as a **`uint16_t`**.

A **`uint8_t`** is an unsigned single-byte integer, and a **`uint16_t`** is an unsigned double-byte integer. These data types are declared in **`<inttypes.h>`**, which is already included for you in the stencil code.

The local byte-ordering of the **uint16_t** may be different from the network order, so you must be sure to make the appropriate conversion to *network byte order*. To accomplish this, you'll want to use the functions **htons()** and/or **htonl()** when sending the server the port number.[1] These functions are defined in **<netdb.h>**. Consult the `man` pages for a description of these functions.

3.2 Data from the Server

The TCP port of your radio client will also receive instructions from the server. You can receive the following instructions during the lifetime of your program:

For `Welcome` you will want to send:
`uint8_t reply_type = 0;`
and
`uint16_t num_stations;`
	The **Welcome** reply (**reply_type = 0**) will be sent in response to the **Hello** command, so your client will receive it only once. A **Hello** command followed by a **Welcome** response is called a *handshake*.
	The **num_stations** piece of the message tells the client how many stations the radio station server currently runs. Keep in mind that this is a 0-indexed list, so the highest station index is **num_stations** - 1.

For `Announce` you will want to send:
`uint8_t reply_type = 1;`
`uint8_t song_name_size;`
and
`char song_name[song_name_size];`
	Your client will receive **Announce** messages (**reply_type = 1**) from the server on two types of occasions: after the client changes stations (by sending a **Set Station** command), and whenever current station restarts the song.
	song_name_size indicates the length, in bytes, of the new song name. That name will then be transmitted in the **song_name** array. Note that the song name sent will *not* be a null-terminated string, but merely a sequence of characters.
	Your client should echo the announcements it receives back to the terminal, in real time. However, it must not do so through **stdout**, since doing so would interfere with the mp3 data streamed to the UDP port (see section 4). You can solve this problem by writing announcements to **stderr** instead, which can be redirected independently of **stdout**.

[1] Since the network byte order is always defined to be big-endian, we might need to convert the byte ordering before we send data across the network. **htons()** stands for "host (local machine) to network, short (2 bytes)" and **htonl()** stands for "host to network, long (2 bytes)".

For `Invalid Command` you will want to send:
`uint8_t reply_type = 2;`
`uint8_t reply_string_size;`
and
`char reply_string[reply_string_size];`
	The **`Invalid Command`** reply (**`reply_type = 2`**) is sent as a response to any invalid command that your client sends to the server.
	Similar to the `Announce` command, the **`reply_string_size`** indicates the number of bytes contained in the reply string, and then **`reply_string`** contains the reply characters.
	After sending an **`Invalid Command`** message, the server will close its connection to your client. If this happens, your client will no longer be able to do anything meaningful, so it should exit gracefully.

If **`tcp_fd`** is the file descriptor associated with your TCP port, you can read the instructions from the network as follows:

```
 /* Reading a Welcome Instruction */


uint8_t reply_type;
uint16_t num_stations;
int ret;


if ((ret = read(tcp_fd, &reply_type, sizeof(uint8_t))) == sizeof(uint8_t)) {
    /* use reply_type to do stuff */
} // otherwise, handle errors


if ((ret = read(tcp_fd, &num_stations, sizeof(uint16_t))) == sizeof(uint16_t))
{
    /* use num_stations to do stuff */
} // otherwise, handle errors
```

Remember that **`read()`** transfers untyped bytes into a generic buffer. Here, those buffers are the **`reply_type`** and **`num_stations`** variables. You can think of these variables as arrays of bytes with sizes **`sizeof(uint8_t)`** and **`sizeof(uint16_t)`** respectively, so that reading into their address sets their values.

You should make sure that **`read()`** is correctly error-checked. In particular, in the event of a server shutdown (which can be done via **`ctrl-c`**), the TCP connections will be closed and all read calls waiting on the TCP ports will return 0. If other errors occur, the syscall will return -1

and set **errno.** For more information, you should refer to the man pages (**man 2 read**). Your client should handle all error cases gracefully.

When you sent commands to the server, recall that you used **htons()** and/or **htonl()**. As you may have guessed, there are corresponding functions to go in the other direction! Use **ntohs()** and **ntohl()**, which are also defined in **<netdb.h>**.[2]

Here is a visual summary of how sockets and TCP work in Unix:

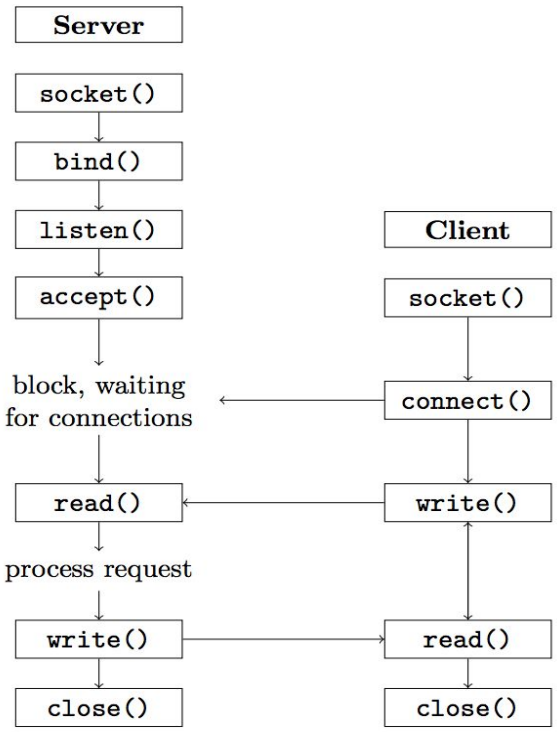

Figure 1: This diagram was inspired by an answer we found on StackOverflow.

4 UDP Connection

The UDP part of your client receives the song's mp3 encoding from the server. It will not send commands to the server; all it needs to do is echo the mp3 data that it receives from the server to **stdout**. You can then play the music by piping the output of your client to another program (see section 7).

After your client initiates its TCP connection with the radio station server and chooses a station, the server will begin streaming data to the UDP port of your client. Consequently, the UDP part of your client need not **connect()** to the server; all it needs to do is **bind()** to the port passed

[2] As you may have guessed, **ntohs()** stands for "network to host, short" and **ntohl()** "network to host, long".

to your program as an argument. Do this before your TCP connection first connects to the radio station server.

When you are filling out the `addrinfo` struct for UDP, you should note that the `ai_flags` field MUST be set to `AI_PASSIVE`. This is because we want the UDP socket to be in a listening mode. For more information, please refer to the man pages.

5 User Interaction

The TCP part of your radio client communicates with the server, establishing the connection and then controlling the station streamed to the UDP part of your client. These elements of your client do not, however, provide any control over when the station should be changed or the connection should be closed—when should your TCP connection send a **`Set Station`** command? When should it close its connection to the server?

A physical radio performs these operations when instructed to do so by its user; your radio client will do the same. We have already written this for you in the stencil code, so don't worry about it!

Once you have finished the parsing section in the **`select()`** loop, you can set the station through the command line simply by typing a number between 0 and one less than the number of files you passed the server and hitting enter.

6 Using select()

The **`select()`** while-loop handles input to the radio client from the udp port, the tcp port, and the user. Be sure to familiarize yourself with the **`select()`** function before starting. Important functions you should be using within your **`select()`** loop include **`FD_ZERO(), FD_SET()`**, and **`FD_ISSET()`**. The **`man`** pages for these functions contain detailed information about what they do and how they should be used.

Some helpful tips:

- **`FD_ZERO()`** and **`FD_SET()`** should be called with every iteration of the **`select()`** while-loop. This is necessary because when **`select()`** returns, it modifies the contents of the input file descriptor set to contain only the ready file descriptors. The other file descriptors will have been cleared and thus must be restored in the next iteration of the loop. For example, if only the udp port becomes ready, then the tcp port and **`stdin`** file descriptors will be removed upon return.

- The first argument of **`select()`** is the highest file descriptor in any of the sets, plus 1.

- For its last argument, **`select()`** takes a timeout value. If you set this value to **`NULL`**, **`select()`** will block indefinitely.

7 Server and Testing

A radio station server is provided in **`/course/cs0330/bin/cs0330_networking_server`**, and there are some .txt and .mp3 files available in **`/course/cs0330/pub/networking`**. There are two ways to run this server. The first is to run:

```
cs0330_networking_server <port> <file1 [file2 [file3 [...]]]>
```

where each argument after the port is the path to a file. This will run the server with a number of stations equal to the number of files provided.

The second way is to run:

```
cs0330_networking_server <port> <folder_of_files/*>
```

Which will create a station for each file in the folder specified. To create a server with the files we provide, run:

```
cs0330_networking_server <port> /course/cs0330/pub/networking/*
```

Here are two ways to test your program once you have a server running:

- As is, the client will stream input to `stdout`. This means that tuning into an mp3 file station will stream random characters to `stdout`, and tuning into a text file station will stream the contents of the text file to `stdout`. One way to make a simple test is to create a small text file (or use one of the text files in **`/course/cs0330/pub/networking`**) and add it as a station to the radio station server.

  ```
  /bin/echo "Hello World" > networking_test_file.txt
  /course/cs0330/bin/cs0330_networking_server <port> networking_test_file.txt
  ```

  If you listen to this text file station with a radio client, you should see the text file's contents streamed to `stdout`.

- To listen to an mp3 file, you can try piping the output of your radio client to the mpg123 program:

  ```
  ./client <hostname> <serverport> <udpport> | mpg123 -
  ```

  Bring your headphones to the CIT and have a listen (mpg123 won't work remotely). After entering the above command, enter the station index as you would when running the client REPL normally. Make sure you use the headphone jack in the back of the machine - the jack in the front likely won't work. *Many machines in the CIT have non-functional*

sound. For those that have functional sound, the volume may need to be adjusted. To check, run `amixer sset 'Master' 100%` . If the “Front Left” and “Front Right” speakers say [off], run `amixer sset 'Master' toggle`. Note that trying to listen to a .txt file with this method will terminate the server with an issue.

8 Getting Started

A good way to start this lab is by doing the following:

- Set up your TCP port and test your ability to “handshake” with the server. You can do this without setting up the UDP part of your client, so starting here will help you ensure that you are connecting to the server correctly before moving on to other parts of the project.

- Set up your **`select()`** while-loop so that it can listen for user input, and make sure that the loop is working properly. We have provided a function, **`handle_input`**, which will act on user input; you should be reading in user input from `stdin`, null-terminating it, and passing it into **`handle_input`**. Your **`select()`** while-loop will ultimately handle reading from `stdin`, the TCP connection, and the UDP connection.

These two steps are essentially independent of each other, but it will be difficult to proceed until both are done. We have provided template code for you; all you have to do is fill in the TODOs.

9 Tips

- Please keep in mind that music files contain binary data, not character data. Thus they contain zeroes that do not indicate the end of strings (since they do not contain strings). Make sure to use **`write`** instead of **`printf`** or **`fprintf`** to write their contents to `stdout`.
- Note that **`getaddrinfo()`** *allocates memory*. You should make sure to free it to avoid leaks (see the manual for **`freeaddrinfo()`**).
- Reiterating from the beginning of this write up: before you begin working on the project, make sure you review the lecture materials on network programming, *particularly the code demos provided on the website*, as well as any relevant **`man`** pages for new or unfamiliar functions. This lab is often the first exposure students have to network programming, and working with a guide will be very helpful.
- For this lab, you will have to write your own makefile, which can be modeled after previous labs’ makefiles.

10 Finishing Up

When you are done, run **`33lab_checkoff lab10`** to submit and check off your lab. Both you and your partner must get checked off individually. You may run this command as many times as you want; we will use the most recent grade and handin time recorded by this program.