Project Snowcast

Due: 11/25

1 Introduction

You are a baby. Sometimes, you have trouble sleeping at night. The scientists at CS033 neonate research facility believe that the soothing melodies of "Vanilla Ice" may help you with your chronic insomnia. They have graciously set up a piece of server software that broadcasts the lullabies, but they are relying on you to create a client for their server. Your task is to create this client so that you and babies everywhere can be lulled gently to sleep.

2 Assignment

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

A Snowcast 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 traffic is uni-directional, so UDP will simply send packets to its destination without waiting for confirmation that the packet was received. If a packet fails to be successfuly delivered, it will simply be skipped over. This makes UDP well suited to real-time applications such as streaming music.
- Because there is no error checking, UDP can send packets more quickly than TCP.

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 Snowcast client should take three arguments:

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

`hostname` is the name of the machine that is running the snowcast 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, your web browser generally uses port 80 and `ssh` uses port 22 by default). `serverport` should match what you chose when setting up the server, and `udpport` can be anything not in the reserved range.

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. Network programming is quite complicated and working with a guide will be very helpful.

Get started by running

```
cs033_install snowcast
```

to acquire the stencil for this project.

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 for the Server

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

- `Hello:`
  ```
  uint8_t command_type = 0;
  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 send 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`.

- `Set Station:`
  ```
  uint8_t command_type = 1;
  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 Snowcast 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 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. 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 Snowcast client will also receive instructions from the server. Prepare to receive the following instructions during the lifetime of your program:

- `Welcome:`
  ```
  uint8_t reply_type = 0;
  uint16_t num_stations;
  ```

  The `Welcome` reply 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*.

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

  ```
  uint8_t reply_type;
  uint16_t num_stations;
  if (read(tcp_fd, &reply_type, sizeof(uint8_t)) != sizeof(uint8_t)) {
      /* error handling */
  };
  if (read(tcp_fd, &num_stations, sizeof(uint16_t)) != sizeof(uint16_t)) {
      /* error handling */
  };
  ```

  Remember that `read()` transfers un-typed bytes into a generic buffer. Here, those buffers are the `command_type` and `udp_port` variables. You can think of the 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.

- `Announce:`
  ```
  uint8_t reply_type = 1;
  uint8_t song_name_size;
  char song_name[song_name_size];
  ```

  Your client will receive `Announce` messages from the server on two occasions: after it changes stations (by sending a `Set Station` command) and whenever the station that it is listening to changes songs. `song_name_size` indicates the length, in bytes, of the new song name; that name will follow 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`, which can be redirected independently of `stdout`.

- `Invalid Command`:

```
uint8_t reply_type = 2;
uint8_t reply_string_size;
char reply_string[reply_string_size];
```

  The `Invalid Command` reply is sent as a response to any invalid command that your client may have sent the server. Similar to the `Announce` command, the `reply_string_size` indicates the number of bytes contained in the reply string itself.

  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.

When you sent commands to the server, 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>`.

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 8).

After your client initiates its TCP connection with the Snowcast 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 to your program as an argument. Do this before your TCP connection first connects to the Snowcast server.

5 User Interaction

The TCP part of your Snowcast 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 Snowcast client will do the same. It should wait for input from `stdin`, handling any input received as follows:

- if the input is an integer on its own line, your program should send a `Set Station` command if the indicated station is valid. If the indicated station is not valid, your program should respond to the user with `"Invalid station."`.
- if the input is the string `"q"` or `"quit"` on their own lines, your program should clean up and exit.

- if the input is a blank line, it should be ignored.
- any other input line should receive the response `"Invalid command."`.

You may use and modify your tokenization code from earlier in the semester to handle whitespace. Some examples of how to handle whitespace:

- `"1 1"` is equivalent to `"1 1"`, which falls in none of the non-error categories above, so is an invalid command.
- `"q uit"` is two separate tokens, which is neither exactly the sting `"q"` nor exactly the string `"quit"`, so this is also an invalid command.
- `"1 "` tokenizes to `"1"`, so is a station change to station 1.
- `" quit"` tokenizes to `"quit"`, so the server must quit.
- `" \t\t\t"` is a series of whitespace characters, which tokenizes to no tokens, so should be ignored.

Note that a line is a string of characters terminated by a newline.

6 Using `select()`

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 while loop. This is necessary because when `select()` returns, it modifies the contents of its sets such that they only contain the ready file descriptors. The other file descriptors will have been cleared and thus must be restored in the next iteration of the loop.
- 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 Getting Started

A good way to start this project 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()` loop and make sure that your event loop is working correctly. Once you've done this, you should be able to configure your program to accept input from `stdin` without much trouble—you've already done this in the previous lab, after all. Your `select()` 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.

8 Server and Testing

A Snowcast server is provided for you in */course/cs033/bin/cs033_snowcast_server*. To run this server, run

```
cs033_snowcast_server <port> <file1 [file2 [file3 [...]]]>
```

where each file is an mp3 file. This will run the server with a number of stations equal to the number of files provided. If you don't have your own, you can find some mp3 files in */course/cs033/pub/snowcast*. You could also run

```
cs033_snowcast_server port /course/cs033/pub/snowcast/*
```

which will create a station for each mp3 file in */course/cs033/pub/snowcast*.

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

- Create a small text file and add it as a station to the Snowcast server. The server just streams its input files - its intended use is with music files, but there's no restriction.

  ```
  /bin/echo "Hello World" > snowcast_test_file.txt
  /course/cs033/bin/cs033_snowcast_server port snowcast_test_file.txt
  ```

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

- Try piping the output of your Snowcast 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). 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. Using the MSLab computers is recommended. If you find a computer with working sound in the Sunlab, please run the command

  ```
  cs033_report_sound_computer <computer name>
  ```

  to add to our list of valid computers. You and your classmates can run

  ```
  cs033_list_sound_computers
  ```

to check the list of all computers that anyone in the class has reported as working. Please use this tool to help make your testing easier.

9 Tips

- Please consider the following: 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`.
- You can use your `colorize` program from lab10 to check what your client is printing to `stderr` and `stdout`. Additionally, a lab demo `cs033_colorize` is provided in the course bin. Run `cs033_colorize <snowcast_client> <args>` in your terminal to check whether your output is written to the correct file descriptor. Don't do this to test music streaming, however - it will corrupt the music data.

10 Handing In

To hand in your Snowcast client, run

```
cs033_handin snowcast
```

from your project working directory. In addition to handing in your code, please also hand in the Makefile used to compile your program (if you used one) and a README documenting the structure of your program, any bugs left unresolved in your code, any extra features you added, and how to compile your program (if you do not use a Makefile).

If you wish to change your handin, you can do so by re-running the handin script. Only your most recent handin will be graded.