Project Snowcast

Due: 11:59pm, 16 Feb 2022

1 Introduction

You will be implementing a simple Internet Radio Station. The purpose of this assignment is to become familiar with sockets and threads, and to get you used to thinking about network protocols.

You should be invited to a Github Classroom environment, where you can access your workspace for this assignment. After you set up your github account, you can start the project at https://classroom.github.com/a/_gRtd5z0

As always, feel free to post a question on Ed, come to one of our office hours, or email us at cs1680tas@lists.brown.edu if you have questions. We’re here to help!

2 Getting Started

This assignment is designed to introduce you to socket programming and re-familiarize you with systems programming and building multi-threaded applications.

In lecture 2, we will introduce socket programming in detail by building some programs during class.

In addition, we recommend consulting the online resources listed on the Documents and Resources page of the course website, specifically the Programming Help section.

In particular, we recommend checking out Beej’s Guide to Network Programming, which is an excellent resource on using socket primitives, including example programs. Chapters 2-3 provide a good overview of the mechanics of sockets in Linux, Chapter 5 describes the most important socket system calls in detail, and Chapter 6 provides some full program examples.

3 Protocol

This assignment has two parts: the server, which streams songs, and a pair of clients for connecting to the server and receiving songs.

This section describes the protocol the client and server use to communicate. Some of the terms used here (eg. “MUST”, “MUST NOT”, “MAY”) describe required or optional features, similar to RFC network standards—these terms are described in detail in the Conventions document on the website

https://cs.brown.edu/courses/csci1680/s22/content/conventions.pdf
There are two kinds of data being sent between the server and the client. One is the control data. The client uses this data to specify which station to listen to and the server uses it to give the client song information. The other kind is the song data, which the server reads from song files and streams to the client. You will be using TCP for the control data and UDP for the song data.

The client and server communicate control data by sending each other messages over the TCP connection.

### 3.1 Client to Server Commands

The client sends the server messages called commands. There are two commands the client can send the server, in the following format.

**Hello:**
```
uint8_t commandType = 0;
uint16_t udpPort;
```
**SetStation:**
```
uint8_t commandType = 1;
uint16_t stationNumber;
```

A `uint8_t` is an unsigned 8-bit integer. A `uint16_t` is an unsigned 16-bit integer. Your programs MUST use network byte order. So, to send a `Hello` command, your client would send exactly three bytes to the server: one for the command type and two for the port.

The `Hello` command is sent when the client connects to the server. It tells the server what UDP port the server should be streaming song data to.

The `SetStation` command is sent to pick an initial station or to change stations. `stationNumber` identifies the station.

### 3.2 Server to Client Replies

There are three possible messages called replies the server may send to the client:

**Welcome:**
```
uint8_t replyType = 0;
uint16_t numStations;
```
**Announce:**
```
uint8_t replyType = 1;
uint8_t songnameSize;
char songname[songnameSize];
```

---

2 You can use these types from C if you `#include <inttypes.h>`.

3 Use the functions `htons`, `htonl`, `ntohs` and `ntohl` to convert from network to host byte order and back.
InvalidCommand:
    uint8_t replyType = 2;
    uint8_t replyStringSize;
    char replyString[replyStringSize];

A Welcome reply is sent in response to a Hello command. Stations are numbered sequentially from 0, so a numStations of 30 means 0 through 29 are valid. A Hello command, followed by a Welcome reply, is called a **handshake**.

An Announce reply is sent on two occasions: after a client sends a SetStation command, or when the station a client is listening to changes its song. songnameSize represents the length, in bytes, of the filename, while songname contains the filename itself. The string must be formatted in ASCII and must **not be null-terminated**. So, to announce a song called **Gimme Shelter**, your client must send the replyType byte, followed by a byte whose value is 13, followed by the 13 bytes whose values are the ASCII character values of **Gimme Shelter**.

### 3.3 Invalid Conditions

Since neither the client nor the server may assume that the program with which it is communicating is compliant with this specification, they must both be able to behave correctly when the protocol is used incorrectly.

#### 3.3.1 Server

On the server side, an InvalidCommand reply is sent in response to any invalid command. replyString should contain a brief error message explaining what went wrong. Give helpful strings stating the reason for failure. If a SetStation command was sent with 1729 as the stationNumber, a bad replyString is “Error, closing connection.”, while a good one is “Station 1729 does not exist.”. To simplify the protocol, whenever the server receives an invalid command, it MUST reply with an InvalidCommand and then **close the connection** to the client that sent it.

Invalid commands happen in the following situations:

- **SetStation**
  - The station given does not exist.
  - The command was sent by a client before they’ve sent a Hello command. Clients must send a Hello command before sending any other commands.
  - If the command was sent by a client before the server responded to a previous SetStation from the client by sending an Announce reply, then your server MAY reply to this with an InvalidCommand. This means that your client should be careful and wait for an Announce before sending another SetStation, but your server can be lax about this.
3.3.2 Client

On the client side, invalid uses of the protocol MUST be handled simply by disconnecting. This happens in the following situations:

- Announce
  - The server sends an Announce before the client has sent a SetStation

- Welcome
  - The server sends a Welcome before the client has sent a Hello
  - The server sends more than one Welcome at any point (not necessarily directly following the first Welcome).

- InvalidCommand
  - The server sends an InvalidCommand. This may indicate that the client itself is incorrect, or the server may have sent it out of error. In either case, the client MUST disconnect.

- An unknown response was sent (one whose replyType was not 0, 1 or 2).

3.3.3 Timeouts

Sometimes, a host you’re connected to may misbehave in such a way that it simply doesn’t send any data. In such cases, it’s imperative that you are able to detect such errors and reclaim the resources consumed by that connection. In light of this, there are a few cases in which you will be required to time out a connection if data isn’t received after a certain amount of time.

These timeouts should be treated as errors just like any other I/O error you might have, and handled accordingly. In particular, a timeout must only affect the connection in question, and not unrelated connections (this is obviously more of a problem for the server than for the client).

You are required to handle one timeout condition:

- If a client or server receives some of the bytes of a message, if it does not subsequently receive all of the bytes of the message within 100 milliseconds, the client or server MUST time out that connection. See the Hints section for details.
There are several other conditions that may trigger timeouts. Handling these are optional:

- If a client connects to a server, and the server does not receive a Hello command within some preset amount of time, the server MAY time out that connection. If this happens, the timeout MUST NOT be less than 100 milliseconds.

- If a client connects to a server and sends a Hello command, and the server does not respond with a Welcome reply within some preset amount of time, the client MAY time out that connection. If this happens, the timeout MUST NOT be less than 100 milliseconds.

- If a client has completed a handshake with a server, and has sent a SetStation command, and the server does not respond with an Announce reply within some preset amount of time, the client MAY time out that connection. If this happens, the timeout MUST NOT be less than 100 milliseconds.

A timeout MUST NOT occur in any circumstance not listed above.

Note that while we specify precise times for these timeouts, we don’t expect your program to behave with absolute precision. Processing delays and constraints of running in a multi-threaded environment, among other concerns, make such precision guarantees impossible. We simply expect that you make an effort to come reasonably close—don’t be off by wide margins when you could make obvious improvements, but also don’t bother trying to finely tune it.

4 Implementation Requirements

You may implement this project in C, C++, Go, or Rust if you’d like to use another language, please ask on Ed to seek approval. Generally, you should try to stick closely to the Berkeley socket API for this project. In particular, you may use sys/socket.h in C/C++, net in Go, or std::net in Rust. You may not use RPC libraries or libraries for marshalling/unmarshalling structures to/from bytes such as protobuf. If you want to use any other libraries related to networking or byte-packing, please ask on Ed first.

If you are using C, the util directory of the repository contains a small library for a linked list (list.h). You do not need to use this, but it is available for you if you want. We will provide a similar library for later projects, so you may just want to get acquainted with it now. See the file list.h for usage information.

4.1 Correctness

You will write three separate programs, each of which will interact with the user to varying degrees. It is your responsibility to sanitize all input. In particular, your programs MUST NOT do anything

---

4If you’re using Rust, use version 1.58.1 (the latest as of this project’s release). If you’re using department machines, add /course/cs1680/contrib/rust/bin to your path e.g. by appending export PATH="/course/cs1680/contrib/rust/bin:$PATH" to ~/.bash_profile.
which is disallowed by this specification, even if the user asks for it. The choice of how you deal with this (for example, displaying an error message to the user) is yours, but an implementation which behaves incorrectly, even if only when given incorrect input by the user, will be considered incorrect.

4.2 Clients

You will write two separate clients.

4.2.1 UDP Client

The UDP client handles song data. The executable must be called `snowcast_listener`. Its command line must be:

```
snowcast_listener <udpport>
```

The UDP client must print all data received on the specified UDP port to `stdout`.

4.2.2 TCP Client

The TCP client handles the control data. The executable must be called `snowcast_control`. Its command line must be:

```
snowcast_control <servername> <serverport> <udpport>
```

<servername> represents the IP address (e.g. 128.148.38.158) or hostname (e.g. localhost, cslab6c) which the control client should connect to, and <serverport> is the port to connect to. <udpport> is the port on which the local UDP client is watching for song data.

The control client MUST connect to the server and communicate with it according to the protocol. After the handshake, it should show a prompt and wait for input from stdin. If the user types in ‘q’ followed by a newline, the client should quit. If the user types in a number followed by a newline, the control should send a `SetStation` command with the user-provided station number, unless that station number is outside the range given by the server; you may choose how to handle this situation.

If the client gets an invalid reply from the server (one whose replyType is not 0, 1, or 2), then it MUST close the connection and exit.

The client MUST print whatever information the server sends it (e.g. the numStations in a Welcome). It MUST print replies in real time.

---

5 There’s no need for the UDP client to play the data it receives itself, since you can just pipe its output into another program which plays the music instead. More on this later.
4.3 Server

The server executable must be called `snowcast_server`. Its command line MUST be:

```
snowcast_server <tcpport> <file0> [file 1] [file 2] ...
```

That is, a port number on which the server will listen, followed by a list of files. To make things easy, each station will contain just one song. Station 0 should play file0, Station 1 should play file1, etc... Each station MUST loop its song indefinitely.

When the server starts, it MUST begin listening for connections. When a client connects, it MUST interact with it as specified by the Protocol. Additionally, it MUST send an `Announce` whenever a song repeats.

**Streaming music** To stream music, you should send data to each client at a fixed rate, rather than sending the song data as fast as possible. To do this, assume that all mp3 files are 128kbps, ie, the server MUST send data at a rate of 128Kibps (16 KiB/s).

If multiple clients are connected to one station, they MUST all be listening to the same part of the song, even if they connected at different times. If no clients are connected to a station, the current position in the song MUST still progress, without sending any data. The radio doesn’t stop when no one is listening.

Finally, the server MUST NOT read the entire song file into memory at once. It MAY read the entire file in for some sizes, but there must be a size beyond which it will read data in chunks.

**Server CLI** When the server receives a command, it MUST print out a log of the command receives and any replies it sends to stdout.

It will also have a simple command-line interface: ‘p’ followed by a newline MUST cause the server to print out a list of its stations along with the clients that are connected to each one, and ‘q’ followed by a newline MUST cause the server to close all connections, free any resources it’s using, and quit.

**Robustness** Additionally keep in mind the following requirements for robustness:

- The server MUST support multiple clients simultaneously.
- There MUST be no hard-coded limit to the number of stations your server can support or to the number of clients connected to a station.
- Remember to properly handle invalid commands by sending an `InvalidCommand` message (see the Protocol section above).

---

6 You can convert an existing MP3 with: `ffmpeg -b 128k -i old.mp3 new.mp3`
7 For example, you should be able to have `/dev/urandom` as a station
• The server MUST not crash, even when a misbehaving client connects to it. In other words, in the event of an error handling a client, the server should gracefully handle the error by terminating the connection to that client, rather than crashing the whole server.

5 Testing

A good way to test your code at the beginning is to stream text files instead of mp3s. Once you’re more confident of your code, you can test your program using the mp3 files in the mp3 directory your github repo.

If you are developing locally, you can pipe the output of your listener to an mp3 player to hear sound:

```
./snowcast_listener port | mpg123 -
```

If the output is stuttery consider trying mplayer instead of mpg123.

Unfortunately, there are many details to streaming mp3s well that would require understanding the mp3 file format in detail to do a really good job, so it’s okay if you can’t test by playing sound or if the output doesn’t sound perfect.

Instead we ask only that you stream the mp3 at a constant bitrate, measured using pv. pv is a built-in Linux tool that passes input from stdin to stdout, and prints statistics about the rate at which it is receiving data to stderr. We’ll be testing to see that your rate is consistently 16 KiB/second. You can run it as follows:

```
./snowcast_listener port | pv > /dev/null
```

If you are able to play sound, you can also pipe the output of pv into mpg123 or mplayer to play the sound while measuring the bitrate.

Once your project is functional, make sure to also run more rigorous tests. Try running your server with many different stations. Connect multiple clients to your server, both listening to the same station and several different stations. Make sure that any misbehavior on the part of the clients, the server, or the human users is met with a proper response from your program(s), and not a segfault!

5.1 Reference Implementations

For your convenience, we have provided binaries of reference implementations of the client and the server that follow the protocol and meet all the requirements, these are located in the reference directory of your repository.

You should take advantage of these programs to help make sure you understand the protocol, and make sure that your programs interoperate with the reference. You can test your adherence to the
protocol based on how well your programs interact with them, i.e. run your client with reference server, and vice versa. This is why our protocol, and all network protocols, in general, are specified so precisely, so that different implementations can coexist together. Your programs are expected to interoperate with ours.

6 Handin

Hand in your project by committing and pushing your work to your git repository before each of the milestone and final due dates. We will grade based on the version committed before and closest to each due date.

Regardless of what language you use, please include a Makefile such that running `make` will compile all of your programs and place them in the main directory, and that `make clean` will clean up any compiled programs and object files.

For languages like Go or Rust that may require external dependencies, make sure that `make` also installs these. For Go, assume that we will grade your project starting from an empty `GOPATH`.

If you have any questions about how your project should be packaged for grading, please make a post on Ed.

7 Grading

7.1 Milestone - 30%

To make sure you’re on the right track, 30% of your grade will be a milestone, which is due Friday, February 4 by 11:59pm. Your submission will have two parts:

- Your code for a partially-implemented client and server (requirements below), which you can submit by simply pushing your work to your repository before the deadline
- A design document describing your planned server design, which you should submit on Gradescope before the deadline

Partial client and server  For the milestone deadline, you should have a client that successfully connects to your server, sends a `Hello` command, then waits for and prints the `Welcome` reply.

In addition, your server should listen for new connections, spawn a new thread to handle the request, and (in the new thread) send a `Welcome` reply. For the milestone, it is sufficient to just send any non-zero number for the number of stations in the `Welcome` reply (i.e., you don’t have to process input files and create stations yet).

To ensure you are implementing the protocol correctly, your client and server should be capable of connecting to the reference version of the server and client, respectively.
We will review socket programming in detail during Lecture 2, which should help you get started with these steps.

**Server design document**  The other 20% of your milestone grade is for the design of your server, which is the most challenging part of this project. Submit a document on Gradescope (text file or PDF) that describes how you intend to design your server. Your design should answer at least the following questions:

1. How will your server use threads? In other words, when are threads created?
2. What data does your server need to store for each *client*?
3. What data does your server need to store for each *station*?
4. What synchronization primitives (ie, mutexes, condition variables, etc.) will you need to protect the shared data and have the threads communicate with each other?
5. What happens on the server (ie, how does the stored server state change) when a client changes stations? (Thinking about this may help you answer the previous question.)

If you’re having trouble with the design, please come to our hours, or post on Ed. If you would like to talk in real-time (in person or remote) and are unable to attend any scheduled TA hours, please let us know and we will do our best to accommodate you.

We will provide feedback on your design and implementation by end of day on Monday, February 7 in order to help inform your ongoing implementation.

**7.2 Program - 65%**

Most of your grade will be based on how well your program conforms to the protocol and specification. This includes how well it interacts with the reference implementations, as well as with each other’s projects. Furthermore:

- You MUST check return codes for all system calls you make. In C, you can you can use `perror` to print error messages.
- Do not assume `recv` and `send` will read or write all the bytes you requested. You will need to have to check each return code and re-call them until the entire buffer is read or written. An easy way to do this is to make a wrapper function (eg. `send_all` or `recv_all` that calls `send()` or `recv()` in a loop until you are done, or a an error/timeout is reached. (See the **Hints** section for details.)
- You MUST protect access to data shared by multiple threads, even integers.
7.3 README - 5%

Please include a README file with your program. Describe design decisions, such as how your server is structured in terms of threading, how it handles announces, how it handles multiple clients, etc. List any bugs that you know your program has. We’ll take off less points for any bugs you list than if we had to find them ourselves!

7.4 Extra Credit - up to 20%

The protocol we’ve defined is extremely limited. We’ll consider any addition to the protocol for extra credit. You can also augment the server or client in a non-trivial way. Here are some ideas:

- Add a command which requests a listing of what each of the stations is currently playing (it is acceptable for the TA binary to respond to this with InvalidCommand).
- Add support for multiple songs per station.
- Add a command to retrieve a station’s playlist (maybe the next 5 items or so).
- Add support for adding and removing stations while the server is running through the command line interface. If you remove a station while a client is listening to it, send a StationShutdown packet, or something along those lines, to inform them. If a new station is added, you could send a NewStation packet to all currently connected clients to inform them.

Feel free to ask what we think about your addition. Also note that we’ve awarded extra credit in the past just for particularly innovative or elegant solutions, so feel motivated to do your best in your design and implementation.

A Useful Implementation Tips

Keep in mind the following tips on specific implementation details. These tips are listed for the C programming language. If you choose to use another language, it will probably have similar features to those discussed here.

- For the TCP connection, use recv() and send() (or read() and write()). For the UDP connection, use sendto() and recvfrom(). Don’t send more than 1400 bytes with one call to sendto().

This is because the MTU of Ethernet is 1440 bytes, and we don’t want our UDP packets to be fragmented. You’ll learn more about this later.
• For the TCP connection, one way to configure timeouts is to use `setsockopt()` using the `SO_SNDBTIMEO` or `SO_RCVTIMEO` options, for send and recv operations, respectively. A good place to do this is in a function like `recv_all` that repeatedly calls `recv()` until a certain amount of data is received, or a timeout or other error occurs.

• You will want to permit the server to reuse its port, so that you can kill it and restart it without waiting a few minutes. You can do this by setting `SO_REUSEADDR` on the socket with `setsockopt()`. Consult section 5.3 of Beej’s guide for an example.

• To handle multiple connections on the server, you should have a thread which calls `accept()` in a loop. When it accepts a connection, it should start one or more threads to handle that connection, and then continue `accept()`ing.

• To control the rate that the server sends song data at, use the `nanosleep()` and `gettimeofday()` functions.

• The TCP client has to read input from two sources at the same time - stdin, and the server. You might do this with a thread for the server and a thread for standard input, or you might use `select()`\(^9\) to handle both tasks in a single thread without blocking.

• To implement hostname lookup (e.g. localhost to 127.0.0.1 or cslab6e to 128.148.31.38), use the `gethostbyname()` or `getaddrinfo()` functions.

In general, to find information about how to use specific socket functions, check the following:

• Using your language’s documentation (for C, consult your system man pages, eg. `man socket`). These pages often have details about individual parameters and often examples

• Chapter 5 of Beej’s guide contains descriptions and examples of many socket functions

• You can find lots of examples online on places like StackOverflow. It’s fine to use these, so long as you read about what what you find (eg. in manual pages, Beej’s guide) so you understand what is happening. If you use any examples that you feel are particularly nuanced, include a link in your code or README.

\(^9\)See [http://www.lowtek.com/sockets/select.html](http://www.lowtek.com/sockets/select.html) for a guide on `select()`