Project 1: Snowcast

Due: 11:59 PM, Sep 24, 2018

Contents

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/1TqBD0MJ`

If you're unfamiliar with sockets or threads, you should read the pages about them linked from the course webpage. As always, e-mail us at `cs1680tas@lists.brown.edu`, post a question on Piazza, or come to one of our office hours if you have further questions.

2 Protocol

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

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.

2.1 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`[1] is an unsigned 8-bit integer. A `uint16_t` is an unsigned 16-bit integer. Your programs MUST use **network byte order**.[2] So, to send a `Hello` command, your client would send exactly three bytes to the server.

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 inital station or to change stations. `stationNumber` identifies the station.

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

[2] Use the functions `htons`, `htonl`, `ntohs` and `ntohl` to convert from network to host byte order and back.

2.2 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];
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`.

2.3 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.

2.3.1 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 before a `Hello` command was sent. The client must send a `Hello` command before sending any other commands.
  - If the command was sent before the server responded to a previous `SetStation` 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.

- `Hello`
  - More than one `Hello` command was sent. Only one should be sent, at the very beginning.
- An unknown command was sent (one whose `commandType` was not 0 or 1).

2.3.2 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).

2.3.3 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, they must be taken to only affect the connection in question, and not unrelated connections (this is obviously more of a problem for the server than for the client). The requirements related to timeouts are:

- 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.

- A timeout MAY occur in any of the following circumstances:
  - 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 curcumstance 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.

3 Implementation Requirements

We recommend that you implement this project in C; we find it very straightforward to do so. If you are unfamiliar or rusty with C, read through the documentation linked on the course web page or contact the TAs for help. We will offer full language support and help with debugging tools.

If you decide you would like to implement this or future projects in a language other than C, please contact us beforehand to seek approval. This project intends to familiarize you with the Berkeley sockets API, so you must demonstrate that your language provides a sufficiently similar API. Linking to a web page containing the relevant language documentation is sufficient. You must not use high-level socket wrappers unless you write them yourself; we will tell you which libraries are and which are not acceptable.

3.1 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 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.

3.2 Clients

You will write two separate clients.

3.2.1 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`[3].

3.2.2 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.

3.3 Server

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

```
snowcast_server <tcpport> <file1>...
```

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 file1, Station 1 should play file2, 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.

[3]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.

You want the server to stream music, not to send it as fast as possible. Assume that all mp3 files are 128kbps, meaning that the server MUST send data at a rate of 128Kibps (16 KiB/s).[4]

The server must print out any commands it 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 the server should 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.

Additionally:

- 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 (see the Protocol section above).
- The server MUST never crash, even when a misbehaving client connects to it. The connection to *that* client might be terminated, however.
- 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.
- 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.[5]

4 Testing

We've provided a sample Makefile in each of your github repo that you can use as a stencil to get started.

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 `mp3` directory in each of your github repo.
You can pipe the output of your UDP client into mpg123 to listen to the mp3:

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

If you bring headphones to the sunlab, you should hear something.

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 ones. 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!

[4] You can convert an existing MP3 with: `ffmpeg -b 128k -i old.mp3 new.mp3`
[5] For example, you should be able to have `/dev/urandom as a station`

4.1 Monitoring receive rate

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. Instead we ask only that you stream the mp3 at a constant bitrate. `pv` is a linux built-in executable 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
```

You can also pipe the output of `pv` into `mpg123`.

4.2 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 in each of your github repo. Take advantage of these! You can test your adherence to the protocol based on how well your programs interact with them. This is why our protocol is specified so precisely. Your programs are expected to interoperate with ours.

5 Handin

Hand in your project by commiting and pushing before each milestone and final due dates. We will grade based on the version committed before and closest to each due. We should be able to rebuild your programs snowcast_listener, snowcast_control, snowcast_server by running `make clean all`.

6 Grading

6.1 Milestone - 30%

To make sure you're on the right track, 30% of your grade will be a milestone.

You must commit and push onto your repo by **September 13th at 11:59PM** a client that successfully connects to a server, sends a `Hello` command, then waits for and prints the `Welcome` reply.

10% of the milestone is a small demo. We will have interactive meetings to grade the milestone by September 15th at 6PM.

The other 20% is for the design of your server, which is the hardest part of the assignment. You will be graded based on how well you have thought your design through. Make sure you especially think through your threading model. Will you spawn a new thread for every command you receive? How many threads will you have per client? Will you have one thread to handle all the stations, or one or more threads for each one? How will they communicate with each other? What mutexes will you need?

If you're having trouble with the design, please come to our hours, or e-mail us with your questions. We also encourage appointments outside of our hours if you feel you need help in-person.

6.2 Program - 65%

Most of your grade will be based on how well your program conforms to the 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. You can use `perror` to print error messages.
- You can't assume `recv` and `send` will read or write all the bytes you requested. You have to check each return code and re-call them until the entire buffer is read or written.
- You MUST protect access to data shared by multiple threads, even integers.

6.3 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!

6.4 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 him. If a new station is added, you can maybe 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 Hints/Tips

The following hints and tips are 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()`[6]

For the TCP connection, timeouts can be set on the socket using `setsockopt()`.

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. Look at the end of section 5.6 in the networking guide (off the course website). 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 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()`[7] 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()` function.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS168 document by filling out the anonymous feedback form:

`https://piazza.com/brown/fall2018/csci1680`.

[6]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.

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