CS168 Programming Assignment 1: Snowcast

Assignment Out:	January 26, 2012
Milestone:	February 3, 2012
Assignment Due:	February 10, 2012, 10pm

---

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.

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 `cs168tas@cs.brown.edu` or to 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.

2.1 Client to Server Commands

The client sends the server **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 should use **network byte order**.[2] So, to send a `Hello` command, your client must 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.

2.2 Server to Client Replies

There are three possible **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 `Beat It`, your client must send the `replyType` byte, followed by a byte whose value is 7, followed by the 7 bytes whose values are the ASCII character values of `Beat It`.

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`

[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.

  - 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).

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. Thus far, we have approved requests for C++ and Scheme (but still contact us if you want to use one of these). Note that for the later projects, you will be responsible for finding or writing your own IP and TCP packet headers (these can be included directly from the Linux kernel headers for C students), the TAs will offer limited language support, and your partner for the project must approve of your choice of language.

3.1 Clients

You will write two separate clients.

3.1.1 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]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.

3.1.2 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 should 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.

If the client gets an invalid reply from the server (one whose `replyType` is not 0, 1, or 2), then it should 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.

3.2 Server

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

```
snowcast_server tcpport file1 [file2 [file3 [...]]]
```

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 should loop its song indefinitely.

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

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 should send data at a rate of 128kpbs (16 kilobytes/s).

The server must print out any commands it receives and any replies it sends to stdout. It should also have a simple command-line interface: 'p' followed by a newline should 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 should cause the server to close all connections, free any resources it's using, and quit.

Additionally:

- The server has to support multiple clients simultaneously.
- In order to introduce you to event-based programming, we are requiring that your server utilize only a single thread for accepting new client connections and responding to client commands. This means you are **not** allowed to spawn a new thread for each connected client. In order to accomplish this, you will utilize `select()` or `epoll()`. For more guidance, consult the Intro to Asynchronous Programming guide on the course web page.

- There should 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 should 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 should 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 should still progress, without sending any data. The radio doesn't stop when no one is listening.
- The server should **not** read the entire song file into memory.

4 Testing

We've provided a sample Makefile in `/course/cs168/pub/snowcast/Makefile` 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 in `/course/cs168/pub/snowcast/mp3`. 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.

4.1 Rate Monitor

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. We've created a rate monitoring program available in `/course/cs168/pub/snowcast/rate_monitor`. This takes data from stdin, outputs it 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 kilobytes/second. You can run it as follows:

```
./snowcast_listener port |
 /course/cs168/pub/snowcast/rate_monitor > /dev/null
```

You can also pipe the rate monitor's output 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. They're in `/course/cs168/pub/snowcast`. 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 typing

```
cs168_handin snowcast
```

from *inside* the directory where your work is located. To reduce clutter, the handin script removes .o files and binary executable files, and runs `make clean` before handing in your assignment. You can handin more than once - the new handin will replace the older one. We should be able to rebuild your programs by running `make`.

6 Grading

6.1 Milestone - 20%

To make sure you're on the right track, 20% of your grade will be a milestone.

You must meet with a TA by Friday the 3rd. More information on milestone signups will be provided. 10% of the milestone is a small demo. You must demo a client to us that successfully connects to a server, sends a `Hello` command, then waits for and prints the `Welcome` reply.

The other 10% 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 through your design. Make sure you think through your threading model. How many threads will you have on the server? Will you have one thread to handle all of the stations, or a thread for each one? What mutexes will be needed to ensure data integrity?

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 - 75%

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

We recommend that you use `#define` or const ints (in C++) for protocol-type constants. Your code will be much more readable - you won't be checking to see whether `replyType` is 2, you'll be checking to see that it's `REPLY_INVALID_COMMAND`.

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()`[4]

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 4.2 in the networking guide (off the course website). 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()` 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 `gethostbyname()`.

[4]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.