

# CS168 Programming Assignment 1: Snowcast

---

<i>Assignment Out:</i>	9/6
<i>Assignment Due:</i>	9/17 (11:59pm)
<i>Helpsession:</i>	TBA

---

## 1 Introduction

The purpose of this assignment is to become familiar with the sockets API and refresh your threads programming skills. If you're unfamiliar with threads, you should read the pages on threads linked from our course webpage.

You will be implementing a simplified version of an Internet Radio Station. We have a simplified protocol that you are going to be using, and you will build a client and server that interact using this protocol. You will then be able to listen to music streaming from the server over the CS (or other) network.

In a streaming radio station (like iTunes radio stations), the client connects to the server and plays the music streamed from the server as long as there is music to play. In this project, you will build both the server and the client. However, there are implications that you will learn over the next week or so that govern which type of connection you will want to use to implement this capability. For example, TCP, a reliable<sup>1</sup> protocol, has extra overhead as a result of this reliability. Streaming music has the property that getting newer data is more important than getting all of the data, as opposed to a file transfer. As a result, we'll be using UDP, an unreliable protocol, to transfer the media. We'd like to have all of the data, but if we can't, then we'd much rather get the newer data than retransmit the old.

The motivation for this assignment is that you want to be able to play some streaming media file (like an mp3 file) in real time as it is retrieved from a remote server. Your client will need to stay attached to the server in order to continue the music output. Your radio station should present the illusion that there is an unlimited supply of music. Further, like a real radio station, you will need to support multiple clients attached to the server at any given time.

## 2 Requirements

You can usually run `mpg123 file.mp3` to hear a mp3 file. When your project is complete, you should be able to run `./snowcast <port number> | mpg123` - and get the same output as playing the file directly. Note the hyphen in the command; it is not a typo.

---

<sup>1</sup>You will learn more about what types of guarantees this reliability refers to later in the course

Note: You can use `/usr/bin/aumix` to control the volume and which outputs are enabled. You should bring headphones to the sunlab and connect them to the computer. If any sunlab machine has speakers, please do not use them for this assignment.

There is a useful Makefile that you can copy from `/course/cs168/asgn/snowcast/`.

## 2.1 Server

The server must be able to handle multiple simultaneous connections, using the POSIX threads library. You should be able to connect to it and send a message indicating what port number it should send to, and the station identifier that you would like to listen to. The server should use multiple threads to manage each station in use at one time. The server will have a control port open that will allow clients to disconnect from the server, change stations, or begin a session. This control connection will be implemented using a TCP connection.

## 2.2 Client

There are two client binaries that you will have to write. One is a controller program which will communicate to the server on a TCP port. The other is the actual radio client itself, which will be responsible for outputting the streaming media to stdout, which should then be piped into the media player.

The controller should be able to connect to the server, and indicate that the UDP client it represents is interested in subscribing to the radio station. It should then sit and wait for input from the command line, telling it when to quit and shutdown all of the connections cleanly.

The UDP listener will simply be responsible for reading the data from the datagram (UDP) socket and write it to standard output. This can be a single threaded program.

## 3 Implementation

The details of how to use the sockets API are explained in this excellent guide<sup>2</sup>. There is only one change to the socket library that we are using. If you use the linux machines with the new socket libraries, note that the prototype for `accept(2)` takes a `socklen_t*` as its third argument. For all intent and purposes, you can simply cast an `int*` to this non-struct. Linus Torvalds has an interesting comment in `accept(2)` regarding this silly change.

We recommend that you use C for this and future assignments. However, if you are familiar with C++ and its standard libraries /STL, you may use it, but it will not be supported by the TAs. You will need to use the C-style POSIX threads and sockets for this assignment. You may *not* use C++ wrapper classes in place of these (unless you wrote the wrappers).

You will be using TCP and UDP to implement this program. One thing to keep in mind is that you can't assume that `recv()` will completely fill your buffer. For example, if you are trying to send a 32-bit integer across the wire, you can't assume that the entire integer was received on

---

<sup>2</sup><http://beej.us/guide/bgnet/>

the other end. You must **EXPLICITLY** check the return values of `recv` until you receive the proper amount of data you are expecting. Nor, with `send()`, can you ensure that the entire quantity of data made it to the kernel for transmission. Remember, TCP is a stream-oriented protocol, not packet-oriented.

UDP, on the other hand, is packet oriented, and either the entire packet will make it across, or none of it will.

Another note: You *must* protect access to data shared by multiple threads. Reading / writing to `ints` is not guaranteed to be atomic by C or C++, so you should use pthread mutexes and such to protect shared data. This will be very useful in maintaining a list of available radio stations and the number of clients who are subscribed to said station.

Finally, try to keep your code simple and easy to understand. It will help you when you're debugging, and it will help us when we're grading.

### 3.1 Server

To handle multiple connections, you should have one thread which sets up the server communication/control socket and then calls `accept(2)` in a loop. When it accepts a connection, start a new thread to handle it. To avoid the overhead of creating and destroying a thread for each connection, you may optionally create a pool of threads, and select an idle thread from the pool to handle each incoming connection.

It would be nice<sup>3</sup> if the server restricted access to a certain subtree of the filesystem, so, for example, all playlists are held in a certain subdirectory on the local filesystem, as well as the mp3 files themselves.

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 Linux socket guide at the link above.

The clients need not worry about file sizes; They perceive the stream as an unlimited source of data until they tell the server that they are no longer subscribing. However, the server will need to worry about this. You should be using the `read(2)` system call in order to retrieve file data from the mp3 that you are currently streaming. Alternatively, you can use `mmap`, but you may *\*NOT\** map the entire file into memory at once. We will be looking at your code carefully for this. You should read parts of the file in chunks, then write those segments out to the UDP socket pointing to the clients who have subscribed.

Note: To determine the size of a file, use `stat(2)` or `fstat(2)`. See their man pages for details.

### 3.2 Packet Timing

You may need to slow down the rate at which the server sends packets to the clients, so as to prevent the clients from being overwhelmed by too much data. You can take into account the bitrate of mp3 you're currently sending to know how much to slow down transmission. You do not need to check the bitrate programmatically - you can assume all mp3s are at some known bitrate, but please document your decision.

---

<sup>3</sup>In other words, this is not required functionality.

### 3.3 UDP Listener Client

The listener is responsible for monitoring incoming packets on a specified port and writing the data it receives to stdout. You can use a global buffer to copy data from the kernel. Every time data is received from the kernel, you should write it back out to standard out. Ensure that you clear any old data from the buffer before you get more.

You should be using `recvfrom(2)` in this client tool. This program should use a specified port which can be passed into the controller as a command line argument. All data should be sent to the specified port.

### 3.4 TCP Controller Client

The TCP Controller is responsible for subscribing and unsubscribing to specified server media stations. The TCP Controller knows the control TCP port on the server, as well as the UDP port for incoming data on the client. It connects to the server, sends initialization data. The server should then begin streaming media to the client as soon as the specified station is active.

The TCP Controller should have separate threads for sending data and for reading data.

### 3.5 TCP Packet Types

This section is a little bit misleading because "packet" should really only be used to refer to the UDP datagrams that we are sending. However, we have messages that we use with our TCP sockets that we are calling packets as well.

In the TCP world, you need to differentiate between four kinds of messages for this internet radio model. There are two packets that are sent from client to server, and two packets that can be sent from the server to the client. They are defined as follows:

#### Initialization Packet

```
unsigned short flags
unsigned short station
unsigned short port
```

#### Close Packet

```
unsigned short flags
unsigned short station
```

The initialization packet is used to acknowledge that a UDP listener on the client machine is ready to receive streaming music from the server. It specifies the station and the UDP port number that they are listening on. Don't forget to convert the fields into NETWORK BYTE ORDER!

The Close Packet is used to indicate that the UDP listener is unsubscribing from the radio station. If this is the last client on the station, it should be removed and the streaming audio should stop.

There are two packets that can be sent from the server to the client. They are defined as follows:

**Song Packet**

```
unsigned short flags
char[256] song_title
char[256] artist
```

**Unsubscribe Packet**

```
unsigned short flags
unsigned short station
```

The Song Packet is sent to all new subscribing clients to inform them of what song is currently being displayed. It is also sent to all clients when a song changes.

The unsubscribe Packet is sent to unsubscribe clients, to inform them that a radio station is being torn down, and that they should take appropriate measure to close their open resources.

In all cases, the flags should be used to differentiate between the appropriate pair of possible messages being sent. Also remember that you must call `recv(2)` until you are sure that you have the right number of bytes. Further recall that you should not read the integers straight off of the wire; they should be run through the network byte order macros.

### 3.6 UDP Packets

You should simply send the song data in UDP Packets. Allocate a buffer, memset it, and send a specified number of bytes in this buffer to the opposite side. Remember that you need to bind the socket and use `sendto` and `recvfrom`

## 4 Conclusion

If you need any clarification on the assignment, please post to the mailing list, [cs168@list.cs.brown.edu](mailto:cs168@list.cs.brown.edu). If you have specific questions which are not appropriate for the mailing list due to the collaboration policy, please come to TA hours, or email the TAs at [cs168tas@cs.brown.edu](mailto:cs168tas@cs.brown.edu).

You can find some freely distributable mp3s in `/course/cs168/mp3`.

To hand in, run `make clean; /course/cs168/bin/cs168_handin snowcast` from the directory containing your source files.