Version 1 – 19/12/2016

Final Project: Internet Radio application

First part Due date: Jan 5, 2017

Second part Due date: Jan 27, 2017

Contents

1 Introduction

You will be implementing a simple Internet Radio Station that streams songs using a multicast in a single AS. The purpose of this assignment is to become familiar with sockets and threads, and to get you used to thinking about network protocols.

2 Protocol

This assignment has two parts:
1. The server, which handle connected clients and streams songs to multicast groups.
2. The client (will implemented as a pair of programs), which connecting to the server and receiving the songs. The client programs will be referred as "*Client Control*" and "*Client Listener*".

There are two kinds of data being sent between the server and the client.
One is the control data. The client (Control) uses this data to learn about the stations, 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 multicast group.

There is a connection between the pair of client programs. The data passing between them, is intended to informing the Listener (by the Control) about the proper station to listen on.

You will be using TCP for the control data, UDP for the song data and TCP for the connection between the client programs.

The client-server control communication and the client-client control communication is done by message passing over TCP connections.

2.1 Client (Control) to Server Commands

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

Hello:
uint8_t commandType = 0;
uint16_t reserved = 0;

AskSong:
uint8_t commandType = 1;
uint16_t stationNumber;

AskList:
uint8_t commandType = 2;

A ***uint8_t***[1] is an unsigned 8-bit integer. A ***uint16_t*** is an unsigned 16-bit integer.

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

For each integer field in the ***commands***, your programs **MUST** use network byte order[2].
Your programs **MUST** send the exactly amount of bytes as in the command format. 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.

The ***AskSong*** command is sent by the client to inquire which song is played now in a station. ***stationNumber*** identifies the station the client inquire about.

The ***AskList*** command is sent by the client to inquire which clients is connected to the server and what station they are listen to.

2.2 Server to Client (Control) Replies

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

Welcome:

```
uint8_t     replyType = 0;
uint16_t    numStations;
uint32_t    multicastGroup;
uint16_t    portNumber;
```

Announce:

```
uint8_t     replyType = 1;
uint8_t     songnameSize;
char        songname[songnameSize];
```

ListenersList:

```
uint8_t     replyType = 2;
uint8_t     dataSize;
uint8_t     listenerEntries[dataSize];
```

While each listener entry:

```
uint32_t    clientIP;
uint16_t    clientTCPPort;
uint16_t    stationNumber;
```

InvalidCommand:

```
uint8_t     replyType = 3;
uint8_t     replyStringSize;
char        replyString[replyStringSize];
```

For each integer field in the ***commands***, your programs **MUST** use network byte order.
Your programs **MUST** send the exactly amount of bytes as in the command format.
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.
multicastGroup indicates the multicast IP address used for station 0. Station 1 will use the multicast group IP +1, etc. ***portNumber*** indicates the port to listen to. All stations are using the same port.
A ***Hello*** command, followed by a ***Welcome*** reply, is called a *handshake*.

An ***Announce*** reply is sent after a client sends a ***AskSong*** command about a station ID.

---

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

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 would 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"*.

A ***ListenersList*** reply is sent in response to a ***AskList*** command.
dataSize represents the length, in bytes, of the ***listenerEntries*** field. The ***listenerEntries*** is composed from sequence of listener entries with three fields each.

An ***InvalidCommand*** reply is sent in response to invalid conditions from the client.
replyStringSize represents the length, in bytes, of the ***replyString***, while ***replyString*** contains the error string itself. The string must be formatted in ASCII and must not be null-terminated.

2.3 Client (Control) to client (Listener) Commands

InformStation:

uint8_t replyType = 0;
uint32_t multicastGroup;
uint16_t portNumber;

The ***InformStation*** command is sent by the *client_control* in order to inform the *client_listener* which station to listen on.

The ***multicastGroup*** and ***portNumber*** are the multicast IP and the port of the desired station.

2.4 Client (Listener) to client (Control) Replies

Ack:

uint8_t replyType = 1;
uint32_t multicastGroup;
uint16_t portNumber;

A ***Ack*** reply is sent in response to an ***InformStation*** command.
The ***multicastGroup*** and ***portNumber*** are stay the same as in the ***InformStation***. It indicates watch command is acknowledged.

2.5 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.

2.5.1 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 ***AskSong*** 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:

- ***AskSong***
  - 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.
- ***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, 1 or 2).
- A command that sent with unexpected length (not in the format we mentioned before).

2.5.2 Client (Control)

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

- ***Announce***
  - The server sends an ***Announce*** before the client has sent an ***AskSong***.
- ***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***).
- ***ListenersList***
  - The server sends a ***ListenersList*** before the client has sent an ***AskList***.
  - The server sends a ***ListenersList*** in unexpected format (not in the format we mentioned before).
- ***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** print the ***replyString*** and disconnect.
- ***Ack***
  - The *client_listener* sends an ***Ack*** before the *client_control* has sent an ***InformStation***.
- An unknown response was sent (one whose ***replyType*** was not 0, 1, 2 or 3).

- A command that sent with unexpected length (not in the format we mentioned before).

2.5.3 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 or protocol errors 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:

- 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 SHOULD 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 SHOULD 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 an ***AskSong*** command, and the server does not respond with an ***AskSong*** reply within some preset amount of time, the client SHOULD 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 an ***AskList*** command, and the server does not respond with an ***ListenersList*** reply within some preset amount of time, the client SHOULD time out that connection. If this happens, the timeout MUST NOT be less than 100 milliseconds.
  - If the *client_control* connects to the *client_listener* and sends an ***InformStation*** command, and the *client_listener* does not respond with an ***Ack*** reply within some preset amount of time, the client SHOULD 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.

3 Implementation Requirements

You **MUST** implement this project in **C** to help you become familiar with the Berkeley sockets API.
The project **MUST** work on the laboratory computers using your multicast topology (at GNS3) and using the virtual box machines (pc1, … , pc4).
Your programs **MUST** to work well with the executable programs, which we will supply.

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, "*Client Listener*" and "*Client Control*".

3.2.1 Client Listener

The *client_listener* handles song data and control messages. The executable **MUST** be called radio_listener. Its command line **MUST** be:
radio_listener

The *client_listener* is waiting for one TCP connection on port 12701 on the LOOPBACK INTERFACE. It is waiting for the *client_control* to establish a TCP connection at startup.
For each arriving ***InformStation*** command, the *client_listener* should start listening for the appropriate station. Listening for a station is done by binding a UDP socket to the given multicast group IP and port.
After each receiving ***InformStation*** command, the *client_listener* should reply with an ***Ack***.
The *client_listener* **MUST** print all data received on the listening UDP socket to stdout3.
The *client_listener* stop running when the *client_control* stops.

Remember to close all sockets upon termination,

[3] There's no need for the client_listener 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.

3.2.2 Client Control

The *client_control* handles the control data from the server and from the *client_listener*. The executable MUST be called radio_control. Its command line MUST be:

radio_control <servername> <serverport>

<servername> represents the IP address (e.g. 132.72.38.158) or hostname (e.g. localhost) which the control client should connect to, and <serverport> is the port to connect to.

First, the *clien_control* MUST connect to the *client_listener* and to the server, and communicate with them according to the protocol. After the handshake, it MUST show a prompt and wait for input from stdin. If the user types in 'q' followed by a newline, the client MUST quit (Don't forget to close the socket). If the user types in a number followed by a newline, the control MUST send an ***AskSong*** 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. Following this, it MUST send an ***InformStation*** to the *client_listener*. If the user types in 'l' followed by a newline, the client MUST send an ***AskList*** command to the server. Upon receiving, the *client_control* is should pars the arriving listeners list and print it.

The *client_control* has to read input from three sources at the same time - stdin, *client_listener* and the server. You **MUST** use select() to handle these tasks in a single thread without blocking. No parallelism is allowed for the *client_control*.

If the client gets an invalid reply from the server (one whose ***replyType*** is not 0, 1, 2 or 3), then it MUST close the connection, print
appropriate message 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.

Remember to close all sockets upon termination.

3.3 Server

The server executable MUST be called radio_server. Its command line MUST be:

radio_server <tcpport> <mulitcastip> <udpport> <file1 > <file2 > ...

<tcpport> is a port number on which the server will listen. <mulitcastip> is the IP on which the server send station number 0. < udpport > is a port number on which the server will stream the music, followed by a list of one or more files. To make things easy, each station will contain just one song. Station 0 plays the first file, Station 1 plays the second file, etc... Each station MUST loop its song indefinitely. The server MUST play the files that given at the program arguments.

When the server starts, it MUST begin listening for connections. When a client connects, it MUST interact with it as specified by the Protocol.

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

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 server to print out a list of its stations along with the song each station is currently playing and a list of clients that are connected to it via the control channel (tcp). ‘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.
- The minimum number of clients that can connect simultaneously will not be less than 100.
- 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 MAY 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, the current position in the songs MUST still progress, without sending any data. The radio doesn’t stop when no one is listening.
- The server MUST NOT simply 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 be read in chunks.
- ***Make sure*** you close the socket whenever a client connection is closed, or the welcome socket when the program terminates. You MUST implement it both on the server and client.

4 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 client using the executable files provided to you in the moodle site.
You can pipe the output of your UDP client into the program "play" to listen to the mp3.

```
./radio_listener  | play –t mp3 -
```

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 bit rate. We’ll provide a rate monitoring program in course site.
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 KiB/second. You can run it as follows:

```
./radio_listener | ./rate_monitor > /dev/null
```

You can also pipe the rate monitor's output into "play".

5 Handin

- ✓ Hand in your project in the following format:
  radio_<ID1>_<ID2>.zip
- ✓ Submission via moodle.
- ✓ We should be able to rebuild your programs by running the make command in the terminal.
- ✓ The program must run well on your multicast topology (at GNS3) and using the virtual box machines (pc1, … , pc4) .
- ✓ Your programs must to work well with the executable programs, which we will supply.
- ✓ File names must be as defined in this document.
- ✓ **Due dates:**

  Milestone Due Date: Dec 28-29, 2016

  First part Due date: Jan 5, 2017

  Second part Due date: Jan 27, 2017
- ✓ About the requirements for the Milestone due date, will notify you later.
- ✓ At the first due date, you will be required to submit a working client program.
- ✓ At the second due date, you will be required to submit all parts of the program.
  You can also submit improved client program.

6 Grading

Radio Listener: (10%)

- Working and implemented according to guidelines. (100%)

Radio Control: (40%)

- Creating and maintaining TCP connections + user input. (45%)
- Proper use of "select". (15%)
- Treatment for messages outside the protocol scope (error message). (15%)
- Handling of timeouts. (15%)
- Simple input check from the user. (5%)
- Closing the TCP connections. (5%)

Server: (50%)

- Design paper. (10%)
- Maintaining multiple connections with clients. (35%)
- Proper management implementation of stations, according to guidelines. (20%)
- Treatment for message outside the protocol scope (error message) and outside station range. (10%)
- Implementing command keys. (5%)
- Handling of timeouts. (5%)
- Closing the TCP connections. (10%)
- Dynamic number of stations. (5%)

7 Useful Hints/Tips

- ✓ 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].

- ✓ For the TCP connection, timeouts can be set on the socket using select() or setsockopt().

- ✓ To run two programs at the same time on the virtual PCs (pc1-4) use the program "screen", as explained in the Useful_Commands.

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

- ✓ To implement hostname lookup (e.g. localhost to 127.0.0.1), use gethostbyname().

- ✓ Don't send a struct as it is. Compilers might insert padding bytes between structure members (invisible to you program, but they take space in the structure) to conform some alignment rules. Put each individual field of a structure to a raw byte-buffer manually.

- ✓ If you implement the program using more than one .c file (recommended), we encourage the use of *extern* declaration.

Please let us know if you find any mistakes, inconsistencies, or confusing language in the feedback section in the moodle site course.

---

[4] This is because the MTU of Ethernet is 1440 bytes, and we don't want our UDP packets to be fragmented.