Version 42.00 – 21/12/2017

Final Project: Internet Radio App

Due date: Jan 17, 2018

Contents

1 Introduction

You will be implementing a "simple" Internet Radio Station that streams songs using multicast in a single AS. You will also implement a simple client that will connect the radio station, play songs receive data from the radio station, and even add new songs to the 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.

The server and client both communicate using the **Protocol** (described in the following section). The protocol is the language that both the client and server need to use to interact with each other. If either of them misbehaves, that is, behaves is a way that is contrary to the protocol (wrong message types, wrong format, wrong timing), the connection will be terminated. Either side might trigger the disconnect. The client will disconnect itself from the server (if the server misbehaved). The server disconnects the client from itself (if the client misbehaved). In either case, the server must remain online always, sending songs and be ready to receive more clients at any time.

The **Server** itself is a multithreaded machine, it is always online, it streams multiple songs and handles multiple clients at the same time. The server will need to handle multiple sources of input and output at the same time (and not crush…):
The Server streams several songs at the same time using multicast, with a single song for each multicast address.
The server handles several clients simultaneously; a client might disconnect or send a request at any time, and the server must handle each client properly, and know which clients are connected to it each moment.
The server also handles simple input from a user. The user may choose to close the server or print the server's database.
Any connected client might also request to upload a new song to the station, when the song is uploaded successfully, the server begins playing it, and any client is able to listen to it.

The **Client** is the one listening to the radio. The Client plays the songs that are streamed from the server. The Client sends queries to the server (what song is played at each station), or request to upload a song to the server. The client also interacts with input from a human user (someone needs to ask it to change the station...).

You will be required to implement both programs using C in Linux over PC5 (your project must compile on PC5), while your programs will be tested on the lab PCs (PC1-PC4). This document doesn't describe all the details of implementation required, some of those are up to you, it does however describe the requirements from your program and especially from the protocol. In the end, we expect that each of the programs you write will interact seamlessly with each other, your server will be able to handle any client, and your client will handle and connect to any server.

2 Protocol

This assignment has two parts:

1. The server, which handle connected clients and streams songs to multicast groups.
2. The client connects to the server and receives songs. The client programs will be referred as "*Client*".

There are two kinds of data being sent between the server and the client:
The first is the "control" data, data that contains information about the songs, connection information, etc.
The other kind is "song" data, which the server reads from MP3 files and streams to a multicast group.

The client program has two parts, one handles the control data and the other handles the song data.
You will be using TCP for the control data and UDP for the song data.

The client-server "control" communication, is done by message passing over a TCP connection, this is basically the protocol.
The following sections describe the messages involved, what is the role of each message in the protocol, and the timeouts between each event and message.

2.1 Client 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.
Note that in each message the first field is either the *commandType* or the ***replyType***. This is a single byte that describes the message type, it is constant in all messages of the same type.

Hello:

uint8_t	*commandType = 0;*
uint16_t	*reserved = 0;*

AskSong:

uint8_t	*commandType = 1;*
uint16_t	*stationNumber;*

UpSong:

uint8_t	*commandType = 2;*
uint32_t	*songSize; //in bytes*
uint8_t	*songNameSize;*
char	*songName[songNameSize];*

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>`.

2.1.1 A Brief Overview for each Message Type in the Client

A ***Hello*** command is sent when the client connects to the server.
reserved*:* This field is always set to zero.

An ***AskSong*** command is sent by a client to inquire about a song that is played in a station.
stationNumber: The number of the station we inquire about.

An ***UpSong*** command is sent by the client when it wants to upload a new song to the sever.
SongSize: This is the exact size of the song file, in bytes.
songNameSize: Represents the length, in bytes, of the filename, that is, the length of the song name string, ***songName***.
songName: A string that contains the filename/song name itself. It is of length "songNameSize" bytes. The string must be formatted in ASCII and must not be null-terminated.

2.2 Server to Client Replies

There are five 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];

PermitSong*:*
uint8_t replyType = 2;
uint8_t permit;

InvalidCommand:
uint8_t replyType = 3;
uint8_t replyStringSize;
char replyString[replyStringSize];

NewStations*:*
uint8_t replyType = 4;
uint16_t newStationNumber;

2.2.1 A Brief Overview for each Message Type in the Server

A ***Welcome*** reply is sent in response to a ***Hello*** command.
numStations: The number of stations at the server. Stations are numbered sequentially from 0, so a "***numStations"*** =30 means that stations from 0 through 29 are valid.
multicastGroup**:** Indicates the multicast IP address used for station 0. Station 1 will use the multicast group ***multicastGroup*** +1, and so on.
portNumber: The multicast port indicates the port number to listen to. All stations will be using the same port!

An ***Announce*** reply is sent after a client sends a ***AskSong*** command about a ***stationNumber***.
songNameSize: Represents the length of the filename, in bytes.
songName: The name of the song itself. It is a string contains the filename itself. The string must be formatted in ASCII and must **not be null-terminated**.
So, to ***announce*** a song called "*Roar*", your client would send the ***replyType*** byte, followed by a byte whose value is 4 and followed by the 4 bytes whose values are the ASCII character values of the song name *" Roar"*.

A ***PermitSong*** is sent as a reply for an ***UpSong*** message. It either approves or disapproves the song upload to the server.
Permit: The answer can either be a positive reply (set to 1) or a negative reply (set to 0). The server first checks whether it is able to receive a new song, if the song transfer is possible then the server replies with a ***Permit*** =1, otherwise the server replies with the ***Permit*** =0. After this message is sent the song transfer begins by the client.

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

A ***NewStations*** message is sent to all the clients who are currently connected to the server and not a part of a song upload process, and to the client who just finished uploading. The message announces the current number of active stations. It is sent when there is a change in the number of active stations due to an upload of a song.
newStationNumber: This is the current new number of stations. This field is set to the new number of active stations. The ***NewStations*** reply is also sent to an uploading client when the song transfer is completed successfully.

2.3 Server-Client Messages, field Specifications

We can't just send bytes to the network in any way we want, we need to agree on how each field is formatted as well.
Each message is sent in a single buffer.
The order in which the fields are set in the message is the same as the order presented in this document.
The length of each field is according to the field size, i.e. uint16_t is two bytes long, etc...
For each of the **integer fields** in each message in your programs you **MUST** use network byte

order2, that is, use function such as htons() and htonl() to change your message fields to the network byte order and back again when you receive them.
Your programs **MUST** send exactly the number of bytes as in the command format. So, to send a ***Hello*** command, your client would send exactly three bytes to the server. Any other number is a violation of the protocol.
For each of the string fields, the string must be formatted in ASCII and must **not be null-terminated**.
For example, to send an ***UpSong*** message using a buffer "buff" we set the first byte to 2(this message type), the next four bytes are filled with the song's length **after** we used *htonl*() on the value. Then the next byte (byte 5), will be the song's name length and the rest of the bytes are the song's name (without the null terminator '\0'). And this message length will be:
1 + 4 + 1 + song_name_size.
We can summarize this briefly in the following pseudo code:

.

```
buff[0]  =2;//message type
buff[1 to 4]  =htonl(song_size);// this field is more than one byte long
buff[5]=song_name_size;
buff[6 to song_name_size]=song_name;
```

2.5 Server-Client Protocol basics

In our project, the server and client program have a basic protocol with which they interact. It operates under a TCP connection and is divided into two phases, the first, *initial connection*, when a client first tries to connect to a server, and the second, *established connection*, after the connection had been established.
The *initial connection* has two messages associated with it; ***Hello*** and ***Welcome***.
The *established connection* also has two messages associated with it; ***AskSong*** and ***Announce***.
In this subsection we will give a simple explanation about the protocol. You will see how and where the messages are used in the protocol under normal conditions, and finally we will see a case when an ***InvalidCommand*** is sent. You can read the rest of this document to get a better picture of the implementation details.
In the *initial connection* phase, or the *handshake*, the client is 'turned on' and sends a ***Hello*** messages to the server, the server adds the client to its client list, and replies to the client with a correct ***Welcome*** message, the client then saves the information gained from the ***Welcome*** message and starts playing the first station. The connection moves to the *established connection* phase.
At the *established connection* phase, when the user types a station number at the client side, the client first starts playing the new station and it then send an ***AskSong*** message to the server. The server replies with a correct ***Announce*** message to the client, the client prints the information to the user and resumes waiting for user or server input.

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

In any case of an error, that is, a message is sent out of order, wrong format or a timeout occurred, the server will send an ***InvalidCommand*** message to the client. Consequently, the client needs to terminate and the server closes the connection with the client and removes it from the server.

For example, a ***Hello*** message is sent during the *established connection* phase. See the following figure for an example (Fig 2.5.1).
Whenever the server detects that the client terminated, it will close the connection and remove it from the server. On the other hand, when the client detects connection errors/protocol errors, the client closes the connection and terminates, without directly sending any message to the server. The server still need to detect that the client is disconnected.

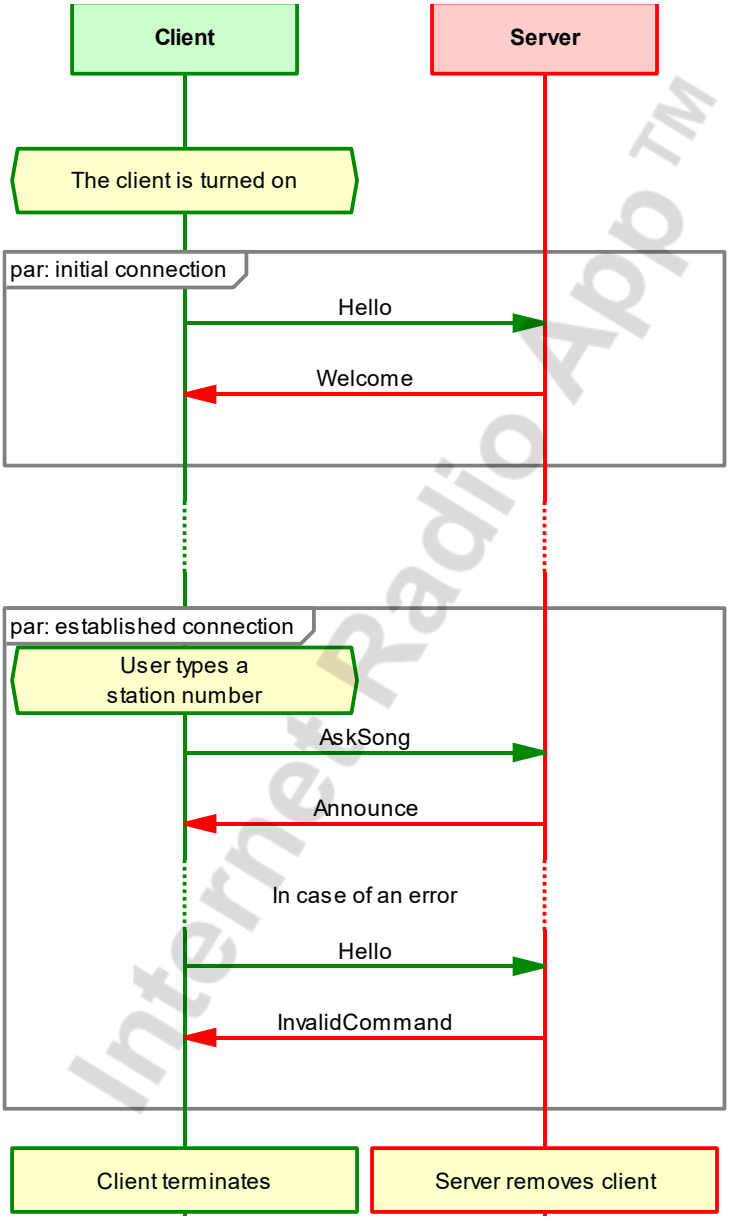

Fig2.5.1: The Server-client protocol (ignore the word “par”).

2.6 Upload Procedure

Each client has the ability to upload a song to the server at any point after a server-client connection has been *established*, once a song has been uploaded successfully, the server opens a new station for the song, and notifies all of its clients. We'll call this the *upload procedure*.
The *upload procedure* has 3 types of messages associated with it; ***UpSong , PermitSong, NewStations.***
In this subsection, we will see how the messages are used in a normal successful procedure and in a one possible case of a failed one.
A **successful *upload procedure*** begins when the user types 's' at the client's side, the user is then prompted to enter a song name, (to exit this sub menu he may merely type a non-existing file name). Once the client program tests that the song file is "uploadable" (The song file can be opened and its size must be in the range of 2000B to 10Mibs.), the client sends an ***UpSong*** message to the server with the correct fields, and the *upload* procedure begins. If no other songs are being uploaded at the moment (only one client can upload a song at a time) the server tests that the song is downloadable (a song file with the same name can be opened and saved, and no such song is being played), if it is, the server sends a positive (***Permit*** =1) ***PermitSong*** reply to the client. The client starts to send data packets to the server. (During the upload the server should not send any messages to the client who is uploading, except an ***InvalidCommand*** in case of an error (the client stopped uploading)). Once the file upload has finished, three things need to happen, the client waits for a ***NewStations*** message from the server, the server opens a new station for the song at the next available slot and begin playing it, and the server sends a ***NewStations*** message to all clients connected to it. Then the *upload procedure* is finished.

In a possible failed case of an *upload procedure* the client stops uploading the song to the server half-way, the server waits for a timeout of **3 seconds** and then sends an ***InvalidCommand*** message to the client, the client needs to terminate and the server closes the connection with the client and removes it from the server.
Please see both cases in the figures below (Figs 2.6.1 and 2.6.2).

During an *upload procedure* with one client the server should still be available for communication with other clients with the same latency. To make this possible we would want to make each client to upload a song at a slower rate then what is possible to avoid flooding the network and slowing down response time. See rate control section for further details.
In case that the client/server behaves in a manner not according to the protocol (wrong messages are sent, bad order, too slow (see Timeouts section and Invalid Conditions section for more information)), it is your responsibility to implement a correct behavior, that is, the server will send an ***InvalidCommand*** message to the client, the client needs to terminate and the server closes the connection with the client and removes it from the server.

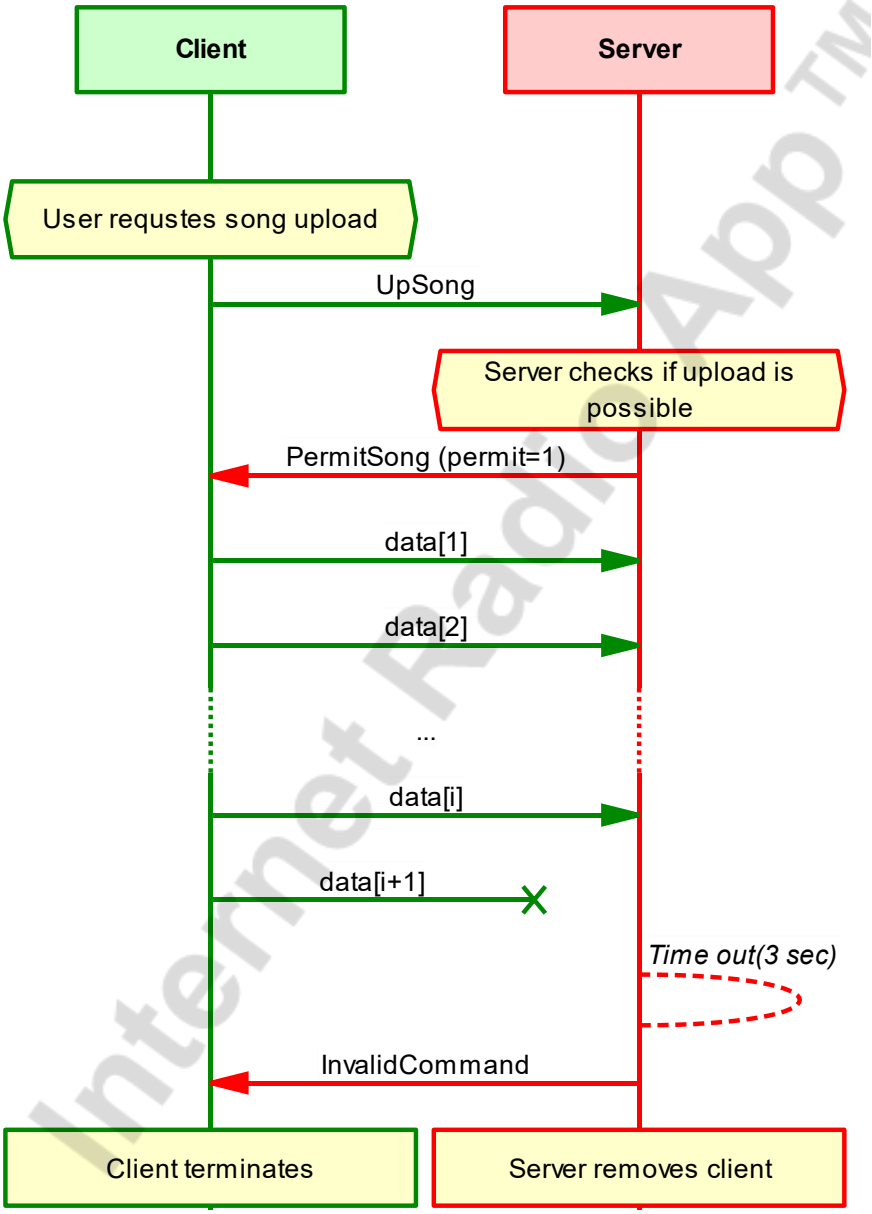

Fig2.6.1: A failed *upload procedure* .

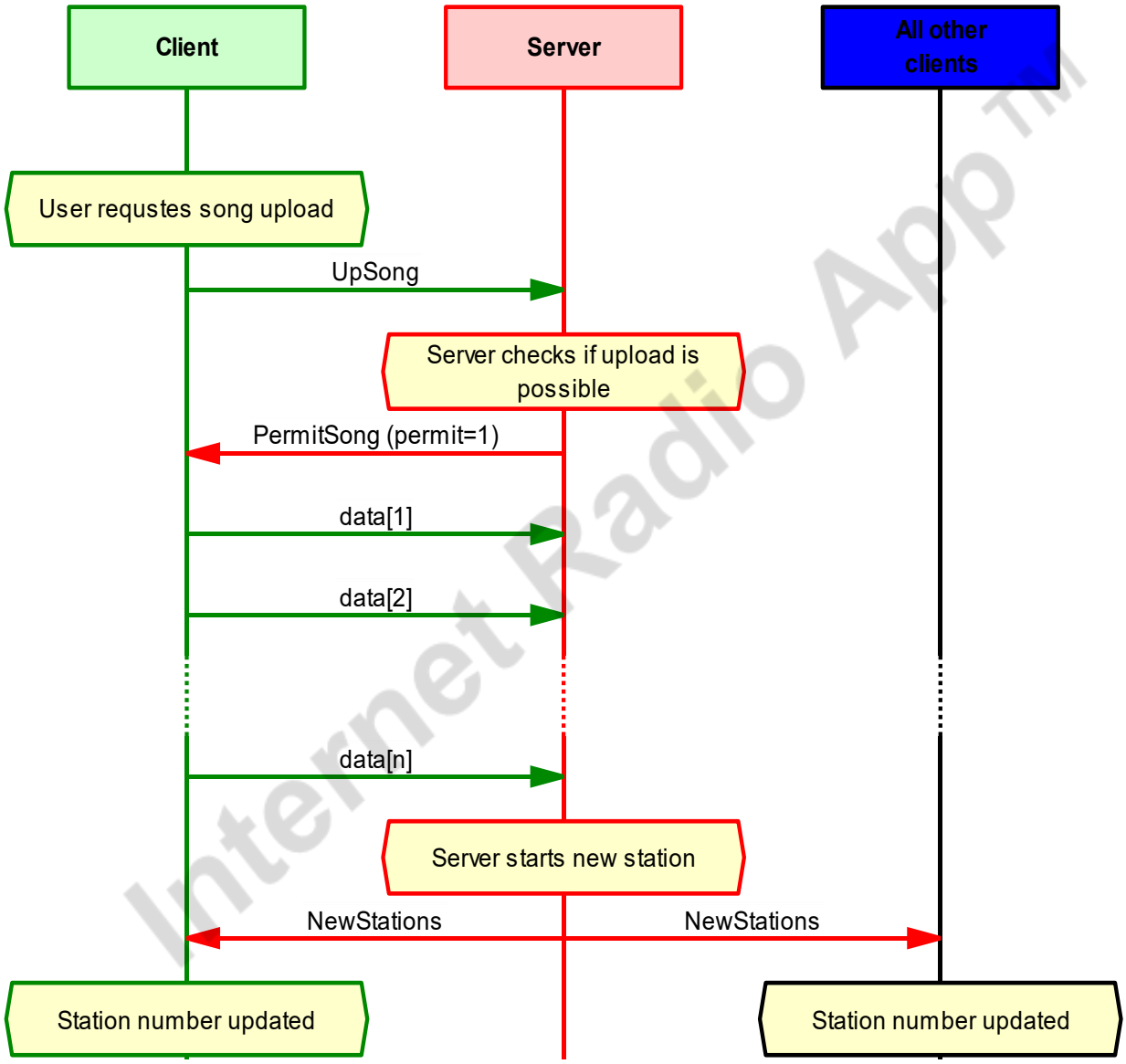

Fig2.6.2: A normal successful *upload procedure* .

Notes:

- Once the *upload procedure* begins the user is "locked" from any further interactions with the client program until the procedure is finished.

-At the server side, the *upload procedure* begins when a positive ***PermitSong*** message has been sent. The server will not send any message (except error messages) to the uploading client.

-If a client receives a negative ***PermitSong*** message (***Permit*** =0), the client program prints an informative message to the user, the *upload procedure* is finished and the client returns to the established connection phase.

2.7 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.

2.7.1 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.

An Invalid command happens 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.
- ***UpSong***
  - The ***songSize*** is out of range (less than 2000B or more than 10MB).
  - The command was sent before a ***Hello*** command was sent.
  - 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).
  - Note that the *best practice* is to disconnect the client on any error, that is, disconnect the client whenever any message is sent out of place, however, you will only be tested on these situations.

2.7.2 Client

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

- ***Announce***
  - The server sends an ***Announce*** before the client has sent an ***AskSong***.
  - The server answers an ***AskSong*** with a ***PermitSong*** message.
- ***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** print the ***replyString*** and disconnect.
- ***PermitSong***
  - The server sends a ***PermitSong*** to the client before the client has sent an ***UpSong*** to the server.
  - The server answers a ***PermitSong*** with an ***Announce*** message.
- ***NewStations***
  - The reply was sent before a ***Welcome*** reply was sent.
- An unknown response was sent (one whose ***replyType*** was not 0, 1, 2,3 or 4).
- A command that sent with unexpected length (not in the format we mentioned before).

- Note that the *best practice* is to exit on any error, that is exit whenever any message is sent out of place, however, you will only be tested on these situations.

2.8 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** be 300 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** be 300 milliseconds.
  - If a client has completed a handshake (the initial connection phase) with a server, and has sent an ***AskSong*** command, and the server does not respond with an ***Announce***

reply within some preset amount of time, the client SHOULD time out that connection. If this happens, the timeout **MUST** be 300 milliseconds.

- For an established connection between a client and a server, if the client sends an ***UpSong*** message and the server does not respond within 300 milliseconds.
- For an established connection between a client and a server, where the client initiated a song upload and is already in the process of uploading a file. When the client has finished uploading the song successfully, the server needs to respond with a ***NewStations*** message within **2 seconds**.
- While a song is being uploaded the server stops receiving data messages from the client transfer (before the entire song was sent), for more than **3 seconds**, the server sends an ***InvalidCommand*** to the client, it removes it from the client list and closes the connection.

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

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** work well with the programs which we will supply.
To upload the files, you **MUST** use sockets with send/recv functions. You must not use external programs or special functions to upload a file.

3.1 Correctness

You will write two 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 is yours, but you must display an error message to the user and an implementation which behaves incorrectly, even if only when given incorrect input by the user, will be considered **incorrect**.
It is highly recommended that your programs contain as many informative prints as possible (you can use the example programs and see some examples) this will allow us to better evaluate your code, when your code is graded at the 'defense, it is harder to give a good grade for a program that is uninformative.

3.2 Clients

You will write one client program, called "*Client Control*".
The *Client Control* handles the control and song data from the server. 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.

The client is logically divided into three parts, the first, *control*, handles a TCP connection with the server, the second, *listener*, handles the song data and the UDP connection, the third, *user*, handles the user's input.
First, the *control* **MUST** connect to the server, and communicate with it according to the protocol. After the *handshake*, it **MUST** show a prompt on screen (regrading data from the server, and regarding the fact the client is now ready to interact with the user), start the *listener* (start playing the song) and wait for input from user(stdin). If the user types in 'q' followed by a newline, the client **MUST** quit (Don't forget to close the socket and free any resources), except when input is locked during song upload. If the user types in a number followed by a newline('\n'), the client **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 (don't send the message to the server).
Following this, the *listener* **MUST** then change the station according to the station number given by the user.
Upload procedure: If the user types in 's' followed by a newline, the user is prompted to input the song file name, the song **MUST** exist in the program's directory, if however, it is not found, the client returns to the established connection phase and waits for further input from the user. An ***UpSong*** will **ONLY** be sent after the song file is found, opened, and its size is established. Once the song upload procedure starts, the client is locked for all user input, any input is discarded. The song file size must be in the range of 2000B to 10Mibs. The client then waits for a conformation from the server, only after the server confirms the transfer, the client begins the file transfer. Once the transfer is complete the client waits for a ***NewStations*** message from the server, it updates the user and resumes normal operation. If for some reason the transfer is cut by the server, the client exits the program in an orderly fashion.
The *listener* **MUST** be implemented as a thread. If a radio station needs to be changed to another, the UDP socket **MUST NOT** be closed, you need to simply join a different multicast group. When the client changes a station, the transition should be smooth and without too much delay (not more than a one or two seconds). A simple correct implementation of the join/drop multicast membership procedure should be enough for this.

The "*client control"* has to read input from three sources at the same time - stdin, a UDP stream and the server. You **MUST** use select() to handle the **user** and **server** control input(data messages) tasks in a single thread without blocking, you may use a different thread to handle the song upload to the server and as mentioned earlier. Song data is handled in a different thread. You **MUST** use no more than two sockets in your client program (we don't need any more)

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

As we mentioned before, if the client gets an invalid reply from the server (one whose ***replyType*** is not 0, 1, 2 3, or 4) or an ***InvalidCommand***, the client needs to exit the program in an **orderly fashion**.
What do we mean by an orderly fashion?
It MUST close the connection.
It MUST print an appropriate error message for the user.
Release all system resources.
Remember to always close all sockets upon termination.
You don't have to implement an orderly exit when the client is terminated from the outside, e.g. using ctrl+c.

3.2.1 Client Listener *tip*

To play songs we need to use an external program called "play" in our code. It is already installed in your lab PCs. To play a song, aside from listening to a multicast group and receiving data via UDP, you will need to open the program and transfer the data to it with the c function popen(3), this will open a new process and give you something called a pipe to the program "play. You will then be able to use the pipe and write song data to it using a function like fwrite().
a code example is given below.
to read about popen() : http://man7.org/linux/man-pages/man3/popen.3.html

Code example:

```
FILE *fp;
. . .
fp = popen("play -t mp3 -> /dev/null 2>&1", "w");//open a pipe. output is sent to dev/null (hell).
. . .

While(. . .)
        fwrite (msgbuffer , sizeof(char), numbytes, fp);//write a buffer of size numbyts into fp
```

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 Server-Client protocol.

You want the server to stream music, that means, not sending 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 will hold a simple database, this database only needs to contain data about which songs are playing in which stations, and information regarding clients who are currently connected to it. The choice of which data to include is up to you, however it needs to be sufficient for the ‘p’ command (that is discussed in the next few lines) to work. 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 its database, that is, it will print a list of its stations along with the song each station is currently playing, and it will also print a list of clients that are connected to it via the control channel (tcp) along with their IP address. ‘q’ followed by a newline **MUST** cause the server to close all connections, free any resources it’s using, and quit (much like the client).
When a client uploads a song to the server, the server will open a new station and stream the new song at that station it will then send a ***NewStations*** message to all the clients that are currently connect to it.
A ***NewStations*** message will only be sent after a song has been uploaded successfully and a new

station had been opened.
Before sending a ***PermitSong*** message, the server will test and see if a new file with a file name specified in the client's request can be opened, that no other songs with the same name are being played, and that no other clients are currently uploading. If all the terms are met, the server will reply with a ***PermitSong*** message to the requesting client with the ***Permit*** field set to 1. It will then begin the upload procedure with that client. Note the server should act normally during this time. During the upload procedure the server should not send any messages to the client who is uploading, except invalid commands in case of an error (the client stopped uploading).
If a file upload failed the server must free any resources it used.
There is no need to test for corruption of the song data, assume that no errors occur in the song data, and that the data sent is actually a song. However, while writing your programs you should make sure that both files (the original and the sent one) are identical and working.

Additionally:

- The server **MUST** support multiple clients simultaneously.
- The server **MUST** support connection and data requests from multiple clients simultaneously.
- There **MUST** be no hard-coded limit to the number of stations your server can support.
- The number of clients that can be connected simultaneously to the server must be 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 MUST be terminated, however.
- 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.
- If a new station is added, this **MUST** not affect any other station or clients connected to the server.
- ***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. Remember to close any further recourses you used when opening a new station.

3.3 Rate Control

Unfortunately, there are many details to streaming mp3s 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. You must set your streaming rate to 1KiB every 62500 usec.
While TCP allows us to safely send large files at a maximal rate, it does not grantee that the latency over the connection used in the file transfer remains the same, meaning the 100ms response time for the server would probably be hard to achieve during this time in our environment, GNS3 with several virtual machines on a single PC, each increasing. To simplify things, you must set the upload rate at the **client** to 1KiB every 8000 usec.
Use #define to properly include these constants in your code!

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. To test mp3 streaming you may want to slightly change your UDP listener from lab 5 so that it goes into an infinite loop and prints all messages to the screen(stdout). You can pipe the output of your UDP listener into the program "play" to listen to the mp3.

```
./UDP_listener <multicast_group> <port> | play –t mp3 –
```

You should use the binaries we provided to you to test your code and your programs, remember that these programs are not perfect, however they have to work your programs have to work with our programs. And your defense would be based on how your programs function with our programs on the lab PC.

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 VirtualBox machines (pc1, …, pc4) .
- ✓ Your programs must to work well with the executable programs, which we supplied.
- ✓ File names must be as defined in this document.
- ✓ **Due dates:**

  Design papers: during the lab hours 30/12/18 and 31/12/18.

  Entire project due date: ?? ??, 2019

7 Grading

Design paper: (5%)

- Client FSM and short Client design paper. **(50%)**
- Short server design paper. **(50%)**

The following are the main issues we expect your program to handle in order of their importance.

Radio Client: (47.5%)

1. Creating and maintaining a TCP connection, and a proper implementation of the Protocol
2. Treatment for messages outside the protocol scope and the handling of timeouts.
3. Handling song data, a proper implementation of the "*listener"* part of the client
4. Proper implementation of the upload procedure
5. Proper use of "select" and threads.
6. Proper handling of user input.
7. Proper termination of the client.

Radio Server: (47.5%)

1. Maintaining active connections with multiple concurrent clients and a proper implementation of the protocol.
2. Treatment for messages outside the protocol scope and the handling of timeouts.
3. Proper implementation of the upload procedure and a dynamic number of stations.
4. Proper management and implementation of the 'radio stations'(streaming music over multicast).
5. Proper use of "select" and threads.
6. Proper handling of user input, and the implementation of the command keys.
7. Proper termination of the server.

8 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()`[3].
- ✓ To control the rate that the server sends song data at, use the `nanosleep()` and `gettimeofday()` functions.
- ✓ 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.
- ✓ Use `pthread_join()` to wait for an open thread to terminate. To exit a thread use `pthread_exit()` and not `exit()`.
- ✓ Use `select()` to implement timeouts, remember that all file descriptors that select tests will have a single timeout. For the TCP connection, timeouts *may* also be set on the socket using `setsockopt()`.
- ✓ Remember to use `perror()` to handle error messages.

Please let us know if you find any mistakes, inconsistencies, or confusing language!

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