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.

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.

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

1.2 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**.

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**.

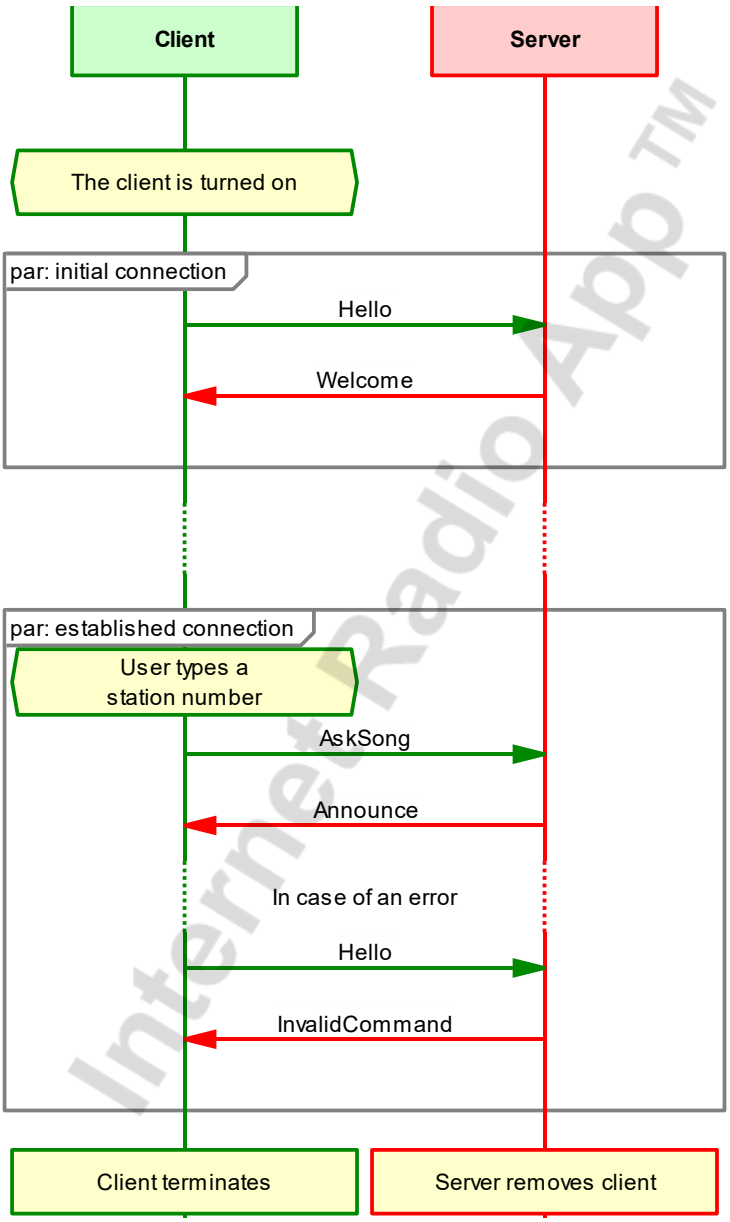
A ***NewStations*** message is sent to all the clients who are currently connected to the server and not a part of a song upload process, and to the client who just finished uploading. The message announces the current number of active stations. It is sent when there is a change in the number of active 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.

1.3 Server-Client Messages, field Specifications

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, Your programs **MUST** send exactly the number of bytes as in the command format.
For each of the string fields, the string must be formatted in ASCII and must **not be null-terminated**.

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

2.5 Server-Client Protocol basics



Internet Radio App ™

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

2.6 Upload Procedure

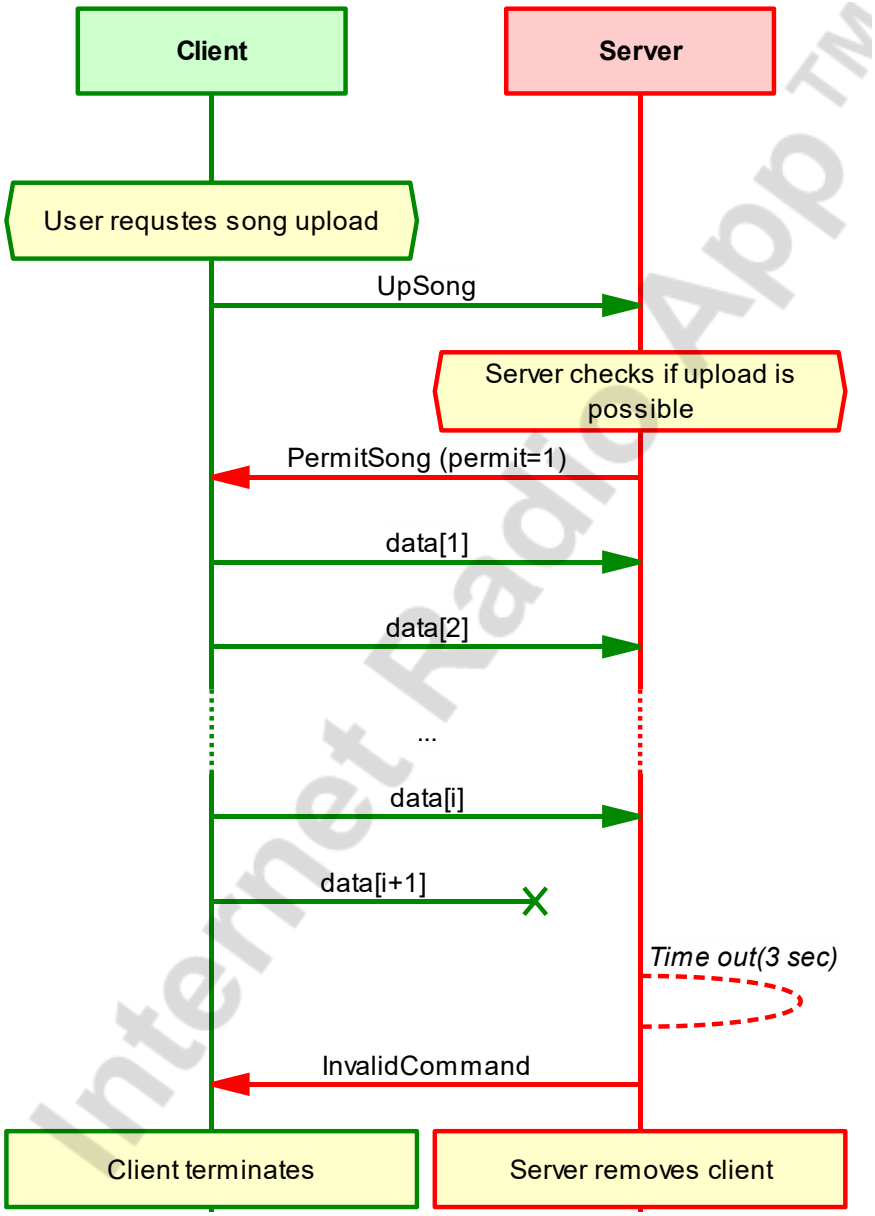
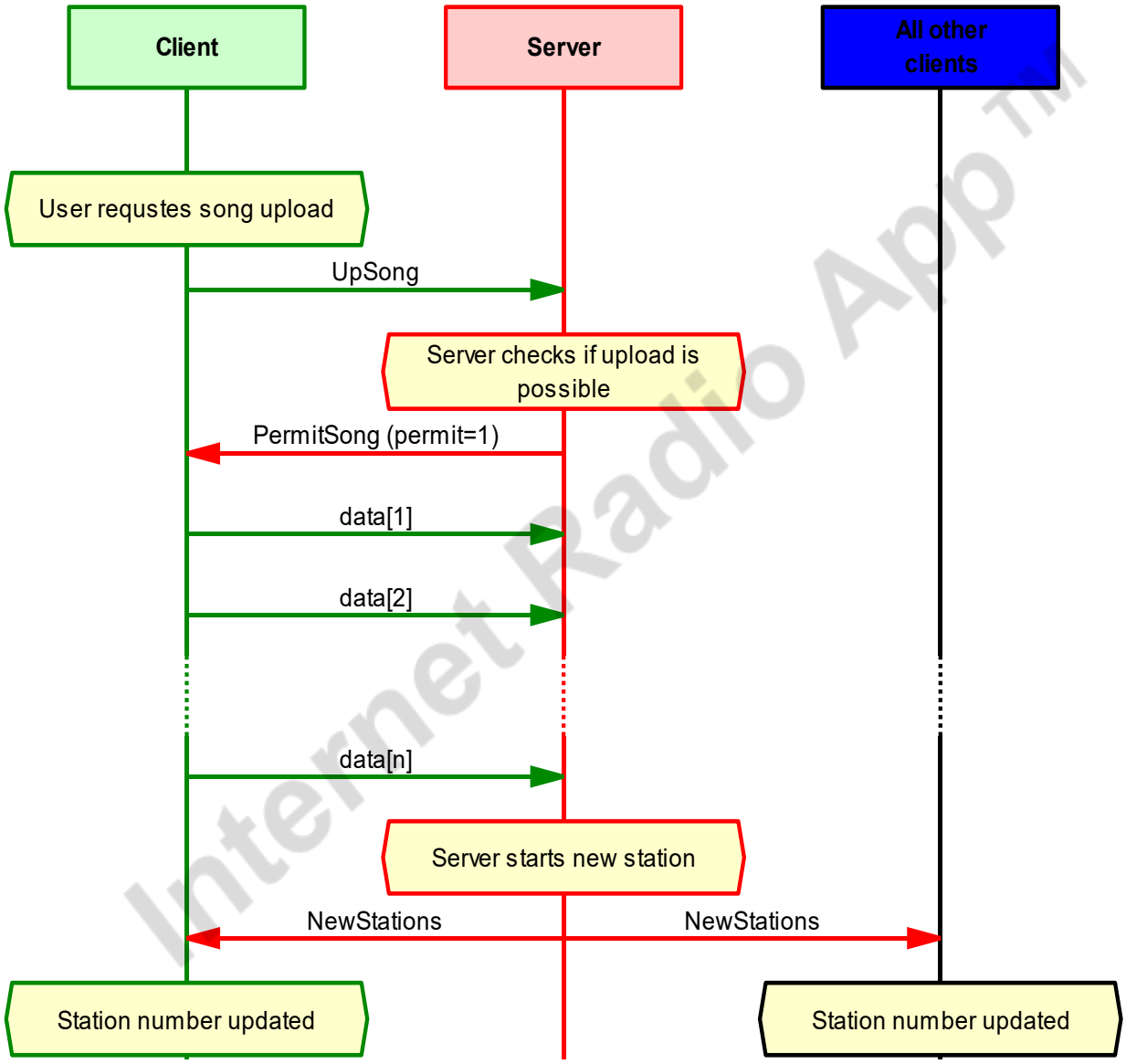


Fig2.6.1: A failed *upload procedure* .



Internet Radio App ™

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.

2.7.2 Client

On the client side, invalid uses of the protocol **MUST** be handled simply by disconnecting and printing the appropriate reason.

2.8 Timeouts

Sometimes, a host you're connected to may misbehave in such a way that it simply doesn't send any data.

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. 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

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.
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.
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. Song data is handled in a different thread.
You **MUST** use no more than two sockets in your client program.

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 play and transfer the data to it with the c function popen(3),: http://man7.org/linux/man-pages/man3/popen.3.html

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.
Each station will contain just one song. 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 Server-Client protocol.

- 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, as well any other socket.

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.

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()`.
- ✓ 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.
- ✓ 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.