Introduction

This project is a chat environment that allows clients to communicate with each other. The chat also supports audio messages that are implemented with the UDP protocol. The project is constituted by two main modules, the *client* and the *server*. The repository, which can be cloned from: *https://github.com/tratteo/MultimediaChat.git*, presents two folders containing the two modules and a *common* folder that contains APIs used by both server and client.

How to run

Dependencies:

- Alsasoundlib: the library used to handle the registration and the playing of audio messages. It can be installed with *sudo apt-get install libasound2-dev*

Each module contains a make file that can be used to compile it.

NOTE: relative paths between each module and the *common* folder are important as the makefile relies on them for compilation.

In order to run the server, after it has been compiled, just run the command *./mc_server.out*

In order to run the client, after it has been compiled, just run the command *./mc_client.out <server_ip_address>*

There are some cases in which the server and the client must be run with the root user, read the Known Issues paragraph.

Server

The server accepts the requests of the clients and handles them asynchronously in a separate thread. It also keeps track of all the connected clients, providing feedback when a certain addressee, requester from another client, is either offline or does not exist at all. The server has also a primitive database module. During start up, this module deserializes the data from disk and fills in its structures containing information about registered users, such as their credentials and their chats with other clients. When the server shuts down, all the updated data are serialized back to disk, this way the server is able to store all the information about new users. The server handles the dispatching of both text messages and audio messages. Each client communicates only with the server, eventually specifying the addressee of the message. If the addressee is online, the server forwards the message to it.

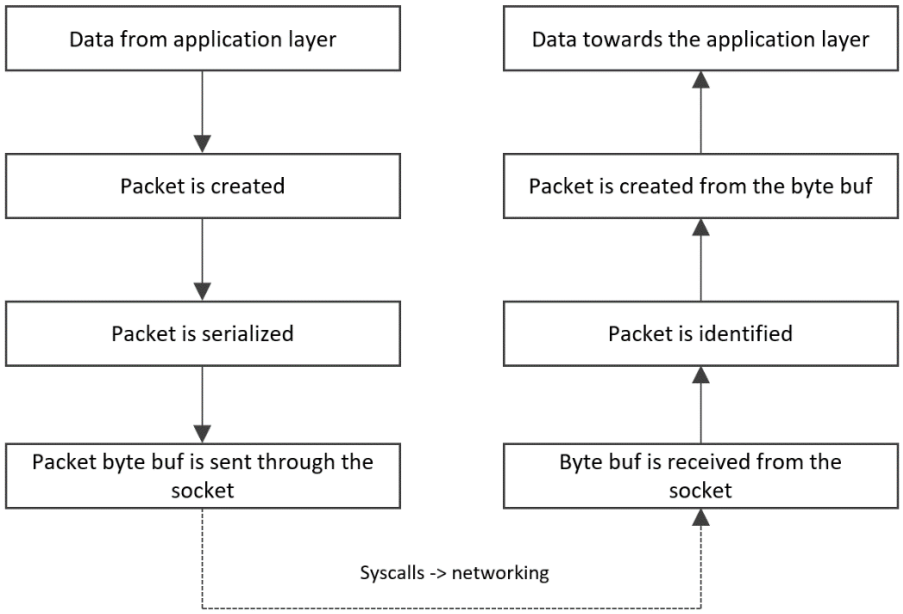
Client

The client provides a simple interface that allows the user to select the desired addressee and to send text as well as audio messages. However, in order to use these functionalities, login is required. Once the client is launched, if it is able to connect to the server, it asks the username and the password. If a username that does not exist is entered, the server stores the newly registered user and informs the client that it has been registered and that can use the service. If the username was already present and the password is correct, the client simply logs in and finally, if the username

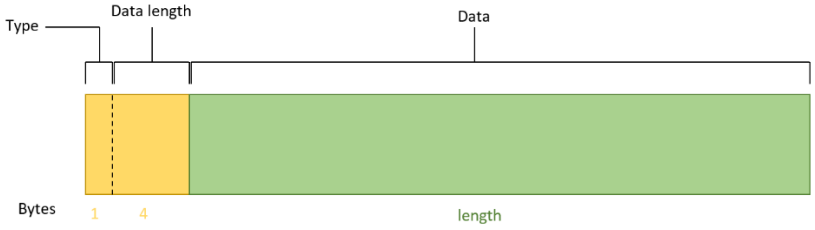
exists but the password is incorrect, the server informs the client that the credentials are invalid.

Design patterns

In order to make messages passing between server and client much easier and intuitive, a small protocol that includes packets has been developed. The functioning of this protocol is showed below.



Both the server and the client have a background thread that is in charge of polling the socket file descriptor, using the *poll()* function, reading data only when it is available. Since data received are raw bytes, a small protocol that defines what kind of packet has been received is mandatory. Thus, the first 5 byte of the buffer (notice that, in this case, we are on TCP, therefore data arrival order is assured), are bytes required for the protocol, in particular the first byte specifies the *type* of the packet, allowing up to $2^8 = 256$ types (the byte is treated as unsigned), the 4 consequent bytes are composed together with bitwise operations in order to obtain and unsigned integer that represents the length of the data fragment of the packet. The packet structure is represented in the image below.



Data inside the packet need to be interpreted in different ways, according to the type of the packet. For this reason, a class *Payload* and inherited classes, have been implemented. By looking at the type of the packet in the first byte, it is possible to

know which payload was sent. Therefore, it is sufficient to pass the data segment to the correct payload class and it will handle the correct deserialization and interpretation of the data.

Audio messages

Audio messages are transferred using the UDP protocol. When a sender wants to send an audio message to another client, entering the */r* command will start the registration. Once the registration has been completed, the data are stored temporarily on the disk. Afterwards, the client sends to the server all the audio metadata, such as its length in bytes, the number of UDP packets (segments) required to send all the data, the sender username and the addressee. At this point the client reads in chunks the audio form the file and sends the packets. In order to assure that the packets are interpreted in the correct order, the client reserves the first 4 bytes of the packet to specify the index of the current packet (UDP protocol does not assure the correct order of data). At this point the server has been informed of the incoming audio and prepares to receive it. The server creates a matrix of dimensions [n.segments][PACKET_SIZE + sizeof(int)], in this way, whenever a packet is received, the first 4 bytes are read into an int *index*, and each packet is stored in the correct index of the matrix. Finally, the data are forwarded to the selected addressee of the audio. At this point the protocol repeats with the only difference that now the sender is the server and the receiver is another client.

Known issues

Valgrind alsasound

This issue presents himself whenever the client is run with the *valgrind* tool. When using the APIs from alsasound library, either for registering an audio or for playing it, the valgrind tool report thousands of errors (mainly specifying that a conditional jump, an if, depends on the value of a not initialized value).

UDP packets dropping

Due to the UDP protocol, it may occur that some packets are lost when sending an audio message. For this reason, the receiver of an audio cannot wait indefinitely for each of the segment to arrive, in fact, in case some packets are lost, the receiver would wait forever. The receiver has some sort of a timeout. Whenever a packet is received the timeout is reset. Whenever the *poll()* returns that no data is available the timeout is incremented. When the timeout reaches a maximum defined values that depends on the total number of segments of the audio, the poll routine is interrupted and the audio is saved as it is, even if some packets have been dropped.

UDP random port bind permission denied

There are cases in which, both the client and/or the server may stop with a permission denied message. This is because whenever a client connects, both the server and the client create a UDP socket on a random port and communicate these ports with each other. Since in Linux ports in range 0-1024 are considered privileged, if a port is created inside this range, if the application has not been launched with root user, it stops. In order to fix this, just run the server and the client as root user.