Introduction

The chatroom as the name suggests is a server-client application to simulate a chatroom.

Implementation

Server

The server-side is composed of *main.cpp*, utils.cpp and utils.h. The server side can be launched by opening a terminal in the server-side directory and using make; ./server <port>.

main.cpp

main.cpp re-uses the code from https://www.geeksforgeeks.org/socket-programming-in-cc-handling-multiple-clients-on-server-without-multi-threading/ It provides the code to basically set up a master socket that acts as a hub to other sockets, thus mimicking a server.

The code is similar to the code to the one showcased in TP9. The only notable difference being that it is error proofed.

We could have re-written the error-proofing ourselves, but it would be a waste of time if we beat around the bush and rewrite it. We decided to just cite it and leave the code as it is and focus on the modification of it. The code was modified to allow the transmission of messages between clients and servers, and to change the port assigned.

Messages received by the server are processed with readMessage. When the server detects data being received, it checks whether it is a message. The first message received by a client will always be the username. The server stores this message into an array of strings called usernames. The client with id x will have its username stored in usernames[x]. If a client already has his username assigned and the server recieves a message. Then the server will prepend the nickname to the message received. Finally, the server will send the new message to all other users with sendMessage

utils.cpp

utils.cpp contains two functions, readMessage and sendMessage.

readMessage requires two parameters, the socket ID and an instance of the message struct, which is the message received by the socket. The function will read the messages received by a socket, byte by byte. It will return true if it succeeded and false otherwise.

SendMessage requires two parameters, the socket ID and the instance of the message struct to send. The function transmits the attributes of the message struct by calling the inbuilt socket function send() for each attribute.

utils.h

It declares the functions in utils.cpp as well as defines the struct message as requested in the announcement of the project.

Client

The client-side is composed of a *main.cpp*, as well as the same utils.cpp and utils.h as the server-side. The client side can be launched by opening a terminal in the client-side directory and using make;./client <nickname> <serveripaddress> <port>. The client and the server share the same utils files because they both send and receive messages encoded in the same manner.

main.cpp

Some of the code reused is from https://www.geeksforgeeks.org/socket-programming-cc/ , which is similar to the code shown in TP8

main.cpp is responsible for creation of the socket and its communication with the server side. The client-side runs on two threads, the main code itself for receiving and a created one for sending.

It reads the arguments parsed upon execution of the client-side. It needs a minimum of 3 arguments, the nickname, the IP address of the server and its port. The address is converted into binary form, the failure of which only happens if the address provided is not an IPv4 address. If it is converted successfully, the socket establishes a connection to the address.

An instance msg of the message struct is initialized. The client's username is assigned as the msg.message and the instance is sent to the server via sendMessage. Then the msg.message is destroyed.

A thread is created after a successful connection and runs the function userThreadFun which runs readMessage under a while loop conditional on the readMessage. It couts the messages received

The messages are sent by running a while loop conditional on getline(cin,line). The user can input his message. The time of the input and the input will be assigned to the respective attributes of the msg instance. The instance is then sent using sendMessage from utils.cpp. The client can get out of the loop by clicking CTRL+D, which ends the program.

Limitations and Possible Solutions (if applicable)

The program has only been tested by one machine running the latest version of ubuntu. It was tested by launching 3 instances of terminal, one for the server, one for client A and one for client B. The clients were able to connect to the localhost IP address (127.0.0.1) as well as the machine's personal local IP address (192.168.0.123). It should theoretically allow LAN connections to be received since the server socket binds to all available IP interfaces, however, it was not confirmed by using two different machines as clients.

Users are unable to see the past messages sent by other users before their connection. - It can be solved by keeping a history and sending the history to new connections. However, this will take up more resources.

Users can have the same name. The server does not take into consideration that another user exists with the same name. - This can be solved by checking if the username is taken in the usernames array and sending a message back to the client that the username is taken.

The user cannot see the timestamps nor the nickname on their messages or their message draft. Having a cout of their username did not resolve it since a receiving message will scramble the draft.

Sometimes, a segmentation error occurs if many messages are tried to be sent at the same moment on a client side. We were not able to properly diagnose the problem

If a client is typing, and they receive a message, the received message is printed beside the client's current draft. Since the received message starts on a new line, the client's draft is above the received message. However, he can continue typing on the draft, but any new characters are below the received message. This may cause a bit of confusion.

Obstacles

Aside from trying to overcome a few of the fixable limitations, the main obstacle encountered was designing how the server will process the messages and correspond it with the sender's username. We initially wanted to add two attributes to the message struct: the username and the size of the username. However, this would eliminate compatibility with other groups. Thus, we implemented this feature as described above. However, we never tried to develop around the idea of compatibility. The server-client is compatible with the server-clients of other groups. However, due to the nature of how the server gets the username (i.e., the client side must send the username as a message), if a client-side of another group connects to our server-side, their first message will be their username. Similarly, if our client-side connects to another group's server-side, then their server will receive a message of our username and send it to everyone (unless it is implemented in the same manner as ours.)