**Chat**

**Overview**

This is a client-server application. In this there are clients (users of the chat) and a server. In this chat the clients are in groups and send messages to these, so every member of the group receives that message.

When a client connects (with a username) he/she is added to a global group. Then he/she can send messages to this group or create a new one. Then he/she can invite other users of the chat to this new group.

Two clients can’t have the same name.

Two groups of the same client can’t have the same name.

Client

The clients can:

* Connect (with a user-name).
* Create groups (with a group-name).
* Send invitations to other users to join a group they are in.
* Join a group (through an invitation).
* Send messages in a group.
* Leave a group (if he is the creator of the group, the group is removed and automatically all members leave it).
* Leave the chat.

Server

It is constantly listening to new connections. When a client connects, its connection information (socket, port, etc.) is stored and it is added to a set of currently connected clients. Then a new thread is created. Here we listen for the client commands and process them.

**Implementation**

Commands sent by client and received by server

CLIENT\_CONNECTS

CLIENT\_CREATES\_GROUP

CLIENT\_INVITES\_CLIENT

CLIENT\_JOINS\_GROUP

CLIENT\_LEAVES\_GROUP

CLIENT\_SENDS\_MESSAGE

CLIENT\_LEAVES\_CHAT

Messages sent by server and received by client

SERVER\_COMMAND\_STATUS (it’s the message sent by the server to indicate if the last command was a success or failure)

SERVER\_CLIENT\_INVITED\_CLIENT

SERVER\_CLIENT\_JOINED\_GROUP

SERVER\_CLIENT\_LEFT\_GROUP

SERVER\_CLIENT\_SENT\_MESSAGE

Status Results

When the message sent by the server is SERVER\_COMMAND\_STATUS, a status result is sent too.

The different results are:

STATUS\_OK

STATUS\_CLIENT\_NAME\_EXISTS

STATUS\_CLIENT\_NAME\_DOESNT\_EXIST (used in cases when a client invites another client. This status indicates can indicate that the invited client doesn’t exist)

STATUS\_GROUP\_NAME\_EXISTS

STATUS\_GROUP\_NAME\_DOESNT\_EXIST

STATUS\_RECEIVE\_FAIL

STATUS\_SERVER\_CRITICAL\_ERROR

Group Names

The group names chosen by clients must begin with a letter (a-z, A-Z) or with a number (0-9);

The group name stored in the group class isn’t the same that was chosen by the client, instead, it has the client’s name appended in front of the group name chosen by that client, followed by an underscore (‘\_’). It’s made in that way in order to let different clients choose the same group name.

Note: the global group name is “\_GLOBALGROUP”.

Server-Side

Functions

listener(param): This is the function used in the threads created for each client. This receives as parameter a struct containing a pointer to the server and the client information. Then we listen on the socket in order to receive the client’s name. If we receive a name then the server method *addClient* is called in order to add the client to the set of clients. If *addClient* returns an error we must close the server. If *addClient* returns a warning then we must send a message indicating that the name already exists, and we must listen for another name. In other case, the server method *joinGroup* is called with the new client and the global group name as parameter in order to add the client to the global group. In case of error or warning the server must be closed. If everything went ok, the server method *listenClient* is called where the commands from this client are processed.

If instead of receiving a name we receive a command indicating that the client leaves the group, we must clean the data related to the thread. Note that we mustn’t call the method *leaveChat* because the client wasn’t added to the set of clients.

Server

There are three main classes: Server, Client and Group.

The server has a set of clients (map<username, Client-class>) and a set of groups (map<groupname, Group-class>).

IMPORTANT: These sets must be protected with a mutex when accessed because many threads can access them.

When the client connects a new ‘Client’ is created where its name and connection information are stored. Then it’s added to the set of clients. Besides, a new thread is created. The client instance and the server instance are passed to the function associated to this thread. Here the commands of the client are listened for and processed.

Methods

Result start(): First of all the global group is created. Then, this method listens to new client connections and accepts them. When a new connection is detected, a new thread is created. The function passed to the thread is ‘listener’ and receives as parameter a pointer to the server and the client information.

Result listenClient(client\*): When the client has been successfully added in ‘start’, we must listen to the commands it sends. Here is where that happens. This method consists of a loop where the ‘receive’ function is called. If ‘receive’ returns an error then a critical error happened and we must set the server member ‘criticalError’ to true. Also we must send to the client the command ‘SERVER\_COMMAND\_STATUS’ with the status result ‘STATUS\_SERVER\_CRITICAL\_ERROR’ and then exit the loop. If it return a warning it is probable that the client closed the connection without sending the command ‘CLIENT\_LEAVES\_CHAT’. In this case we must exit the loop. If it return ‘ok’ we must process the command. Depending of the type of command the different methods are call. These are:

|  |  |
| --- | --- |
| Command | Method |
| CLIENT\_CREATES\_GROUP: Client creates a group. | Result createGroup(Client\* client, string groupName) |
| CLIENT\_INVITES\_CLIENT: Client invites another client to a group. |  |
| CLIENT\_JOINS\_GROUP: Client accepted invitation to join group. |  |
| CLIENT\_LEAVES\_GROUP: Client leaves a group.  If the client is the creator of the group, the group must be removed. |  |
| CLIENT\_SENDS\_MESSAGE: Client sends a message to a group. |  |
| CLIENT\_LEAVES\_CHAT: Client leaves the chat. |  |

In all these cases if the result is an error we must set the server member ‘criticalError’ to true. Also we must send to the client the command ‘SERVER\_COMMAND\_STATUS’ with the status result ‘STATUS\_SERVER\_CRITICAL\_ERROR’ and then exit the loop.

If the result is a warning we must sent the corresponding status result to the client.

In case of the result being ‘ok’ we inform the client that the operation was a success through the status result ‘STATUS\_OK’.

One last command can be received. It is ‘CLIENT\_LEAVES\_CHAT’. In this case we must leave the loop.

In any case when we leave the loop, the method *leaveChat(Client\*)* must be called.

<Result, Client\*> addClient(name, clientSocket): Here a new Client is created with ‘name’ name and ‘clientSocket’ socket, and added to the map of clients. It returns the result of the operation and the new client if there wasn’t a client with ‘name’ name. In other case the client returned is NULL. The result can be an ‘error’ if there was a critical error (error with mutex wait or signal), a ‘warning’ if there was a client with that name or ‘ok’ if everything was ok.

Note: Clients map must be protected when the insert method is called.

TODO

Cuando alguien se agrega hay que avisarle a este cliente la existencia de todos los demás usuarios. Además hay que avisarle a los demás que el cliente se agregó (como se agrega al grupo global con joinGroup solo hay que avisarle a los integrantes de ese grupo que se agregó a ese grupo)

En general, cuando se une a un grupo, o agrega un grupo o con otras operaciones, hay que avisarle al que hace la operación de la existencia de los demás clientes y especialmente a los demás clientes de la acción que realizó el primero.

COSAS A TENER EN CUENTA

MEJORAS

Cuando se accede al map de grupos o de clientes en el servidor se protegen con un mutex (solo se protege el que se usa). Esto se hace tanto cuando se accede para modificar o para leer los maps. Esto hace que si dos clientes tienen que acceder pero no modificar los maps, por ejemplo para enviar un mensaje o porque el servidor está informando de algún evento, se tenga que esperar a que se termine de recorrer los elementos. Una posible mejora sería que se puedan acceder y recorrer de forma simultanea cuando no se están modificando. De esta manera se podría, por ejemplo, enviar mensajes de forma simultanea. Sin embargo de esta forma los mensajes podrían llegarles en distinto orden a los distintos clientes.