

A dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from this bar, containing the date.

08/01/2017

C# Chat Project

Several thin, curved lines in dark blue and light grey originate from the bottom left corner and sweep upwards and to the right.

Yann KERICHARD - Jérôme LEROI
PROMOTION 2018

Table of contents

Introduction	2
I. Chat features & analysis	2
a. Features	2
b. Project Architecture	3
c. Some thoughts on how to implement concepts.....	4
II. Design: technical choices	6
a. Server side.....	6
b. Client side.....	8
c. Message data structure	9
III. Implementation.....	10
a. Sending-Receiving Message.....	10
b. Communication with GUI thread	11
IV. Demo & screenshots.....	14
a. Client	14
b. Server.....	15
V. Possible enhancements	15

Introduction

During this project, we realized a chat application in C#. This application is supposed to work using a Client side and a Server side, connected with each other by a Socket. Multi-threading was used for this chat, and users have the possibility to create different chat rooms. We are going to tell you the way we analyzed, designed, and realized this project in this report. Have a good reading!

I. Chat features & analysis

To start with, we will be dealing about the project in general: its advancement, the architecture, the features we included and much more.

a. Features

In our project, we realized **all the asked functionalities and way more**. We divided the application in two different parts, as it was suggested in the subject: client side and server side. You can take a look at the different features we implemented:

Firstly, we managed to make an **authentication system**. The user has to log in or to register, if he's new, to be able to use the application. We handle data persistence by using SQLite which is a kind of portable database, which works without SQL server.

Secondly, our application uses **sockets**, which communicates with each other thanks to **TCP-IP protocol**.

Moreover, the software is **multi-thread**. The client side is using 2 different threads that allows to handle the communication with the server, without freezing the interface. The server side is working with multiple threads : there is one thread that handle the incoming connections, and one thread that is dedicated to broadcast messages : from a client to a room of clients, for instance. Furthermore, each client has his own thread, only dedicated to himself on the server.

We also developed an important feature for the rooms. A user can chat in multiple rooms, and create rooms with the name he wants. Our **multi-room system** is great because the other rooms are still updated with the messages, even if the user is on an other room. Users can see the room list and choose which one they want to interact with, if they're not interested in using all rooms.

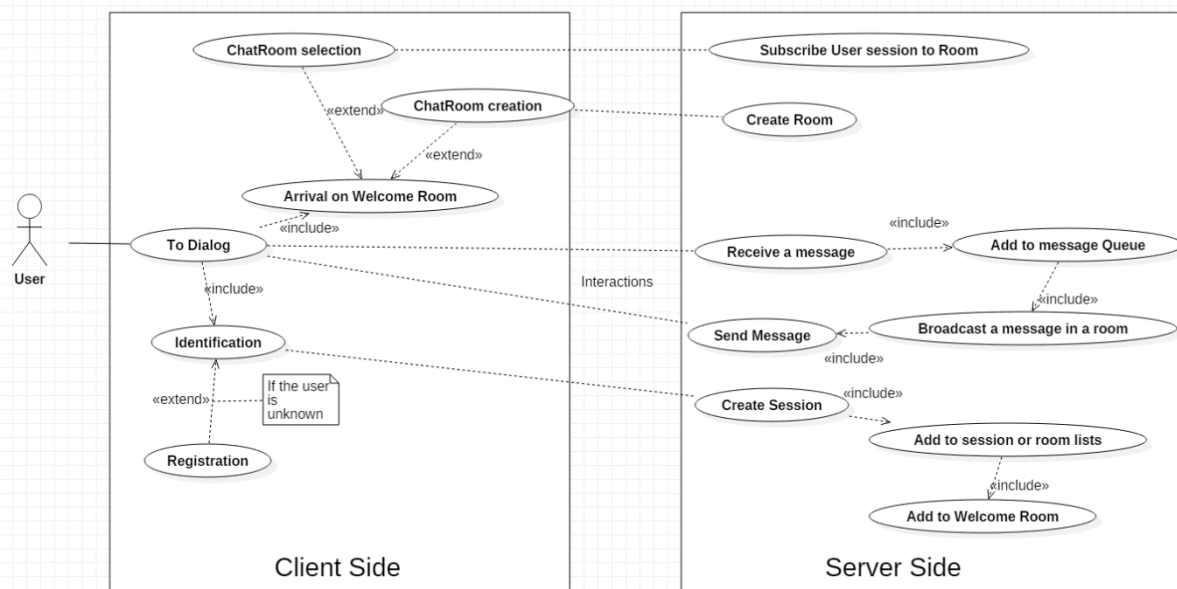
In the multi-room system, we **keep track of the conversation** by saving all the messages in the RAM, and when we have to recover it, we just load it where it is needed. For example, when the user select an other room, the conversation is stored in RAM, and the conversation of the main room is loaded in the interface.

Finally, we had enough time to create a **GUI**, which is way better for any user.

b. Project Architecture

We divided the project into 2 solutions for a better usability inside Visual Studio. These solutions are the client side and the server side of the project.

Below, you can find a custom activity diagram that we used to analyze, and to have a global vision of the different general cases that the software had to handle.



We notice the 2 sides of the application : server side and client side, which interact with each other. The user wants to dialog in a chatroom, but for that he must first register, or if he's not new, he has to enter his username and his password into the login form. Once it's done, the server reacts by creating his user session, and by adding the user into the welcome room. Afterward, the user can either select a chatroom to send messages, and the server add his session to the room, or he can create a custom room. Where the user send a message, it's handle on server side by putting it in a message queue, which is going to be inspected by a server thread and sent to all the users of the room.

Our application has been developed trying to respect the **MVC model**. However, there are some differences. Concerning the client side, we decided to gather the controller with the model. Our choice is easily explainable by the simple fact that we didn't really need to store data. The only case where we store data is when we keep track of the conversation, that's not really annoying. So, for the client side, we have the view, and the controller (which include the model part). Now, let's talk about the server side. We have the model part, which include the database (SQLite), and we have the controller that is the brain of the application. We use it to handle communications between clients, and for sessions and rooms management. We didn't find useful to create a View part, because the only case where we could need it, it could be to display the messages on server's console.

c. Some thoughts on how to implement concepts

During the project, we had hesitations on several way to implement some functionalities. I believe that it is important to notify it, because it can explains some of the reasons of our implementation choices.

First of all, we hesitated on the way of implementing the **Message class**. At the beginning of the project, didn't know if it was better to make two Message classes, one for client side, and one other for the server side. Both classes could have been similar, with some specifications depending on which side it was. Finally, we opted for a better solution, which was to share the **same class between both side**. The main advantage of this method was that it was easier to handle only one class for the communication between both side of the application. We could serialize and deserialize the Message class in the same way. Furthermore, we started by using a simple string type to store the content of the message. It worked, but we found an other possible and smarter implementation. We decided to use **a list of string instead of this simple string**. The Content of a simple message from any user was stored at the list's first position, so in this case it was not very interesting. However, it was very useful when we wanted to send a room or user list because we could just put all room names in the list, and we could send only one message from the server, to set the whole room or user list of the client side.

We had an other hesitation concerning the way of implementing the message broadcasting system on the server. We spend a lot of time in order to determine if it was better to **send the messages from the user thread to the room; or if it was smarter to use the user thread to put the message in a queue, and to use an other particular thread** dedicated to send the messages stored in this queue. Finally we decided to use the second solution, because during the time where the user thread would have sent the messages to the other room users, this user thread would not have been able to listen other messages sent by the user : when the thread send, it can't handle message reception at the same time. This is why **we decided to use a specific message broadcasting thread with a queue**. As a

consequence to this way of implement message broadcasting, we asked ourselves an other question. Now, imagine user A wants to send a message in the room R which contains 200 users. **Was it better to create 200 mini-thread, one by user, to send 1 message to this specific user, of was it better to keep only 1 thread with this message queue (that we used), which would send 200 messages ?** We choose to keep this implementation of only one thread with a message queue. But we think that if we wanted to improve the performance of the system, probably the best solution would have been to use a part of these both methods : use 4 threads of 50 sending messages, for instance.

The last hesitation we had in the project concerned the client side and the way of handle the reception and the sending of messages. We hesitated between using one thread for sending and an other to listen the server, but we chose to use only one thread for both : we found that it was really fast to send a message, and we understood that if the thread was not listening the server because it was sending a message, the message from the server was not lost : it was paused or sent back. So there was no issue.

II. Design: technical choices

In this section, we will review the different components created and used in the project such as the classes or the main methods features.

a. Server side

The chat server has 5 main classes:

- Server class
- Session class
- Message class
- Room class
- Auth class

Server
+ queue: Queue<Message> + counter: int
+ Broadcast(): void + sendToRoom(Message): void + sendToAll(Message): void

Let's see what this is all about, the Server class is the entry point of the application it is where a socket is opened, waiting new incoming connections. When there is an incoming connection on the listening socket, the server creates a session for this user and start a thread with another socket in receive mode, dedicated for this user. This way there are no conflicts when several users send messages at the exact same moment as they all have one socket listening for them on the server. When a message is received by the server, the **Broadcast()** function is called and then depending on the message type, it send it to all the clients connected to the server via **sendToAll(Message m)** or only the one subscribed in a room via **sendToRoom(Message m)**.

Session
+ sessions: List<Session> + rooms: List<Room> + s: Socket + id: int + userName: String
+ send(Message): void + receive(): void + disconnect(): void + linkToRoom(Room): void + removeAllRooms(): void + removeRoom(Room): void + init(): void

The Session class stores several information about the user. There is a *static* List<Session> containing all the users currently connected to the server. Each client has an ID, a

corresponding username, a socket and a List<Room> to which he is subscribed. **Send(Message m)** sends a message thru the session's socket. A thread is also started on **receive()** method during all the client connection time, keeping a socket listening for the client's messages. Other methods are mainly used to subscribe a user to a room **linkToRoom(Room r)** or remove him when he closes his chat program **RemoveRoom(Room r)**.

Message
+ content: List<string> + sender: int + senderName: string
+ Serialize(Message, Socket): void + Deserialize(Socket): Message

The Message class is a shared class between the client and server solutions. It has a function to **serialize()** the message, to be able to transmit it through the socket connection, and the client can **deserialize()** it when received on the socket.

Room
+ rooms: List<Room> + sessions: List<Session> + name: String
+ Remove(Session): void

The Room class contains a *static* List of the rooms available on the server. Each room has a List of sessions which are subscribed to it and the room itself is identified by its name. The functions **Remove()** is called whenever a client disconnect from this room, to unsubscribe him.

Auth
+ co: SQLiteConnection
+ register(string, string): bool + login(string, string): bool + executeQuery(string): int + searchQuery(string): long + userNameTold(string): long + userNameExists(string): bool + generateId(): long

The Auth class is the interface between the server and an SQLite database, allowing some data persistence if ever the server is shutdown. Data saved are the ID, the username and the password. The server uses an integer ID to identify the client but we also introduced the usernames to have a more user-friendly chat. We have some classic functions to **register(string username, string password)** and to **login(string username, string password)**.

Other methods **executeQuery(string sql)** and **searchQuery(string sql)** are for manipulating data and searching within the database.

b. Client side

Concerning the client side, we wanted to have the simplest chat to use for clients, so we created a GUI with Windows Form showing the list of rooms available on the server and a TextBox with the chat messages. The client is prompted to login or register when he starts the program and if the login is successful then the main chat window is displayed.

The chat client is composed of 4 main classes which can be related to an MVC architecture :

For the controller

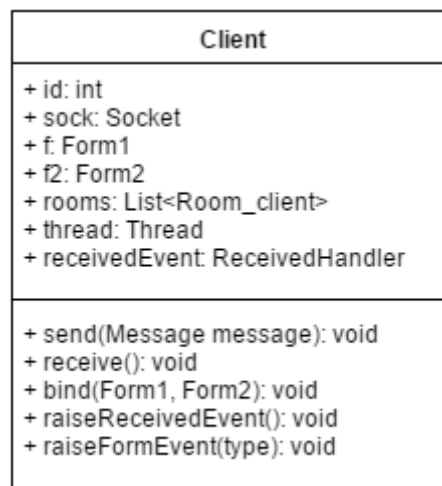
- Client class

For the model

- RoomClient class

For the view

- Form1 class
- Form2 class



The client class acts as the controller, the method **receive()** is started in a new thread to have socket listening permanently for some incoming messages from the server. When sending a message, the method **send()** serialize the message and then transmits them via a *NetworkStream*. We also created some methods to raise an event **raiseFormEvent()** and notify the GUI thread to update the interface.

Room_client
+ name: string + chatbox: string + subscribe: bool
+ save(string): void + getIndex(string): int

The Room_client class represents each room available, with its name and the messages sent on it, saved with **save()**. The user can be subscribed to that room or not. If not, he won't receive messages automatically.

The Form1 class represents the main window containing the chat and the room list. On the other hand, Form2 is the login form that shows up when the program is launched. When the user launch the chat he can see all the public rooms available on the server, by default he is only subscribed to the *Welcome_room*. But as soon he selects a room in the left menu, he is subscribed to that room and starts receiving message. He can switch between multiples rooms without losing any messages and he can even create its own room.

In order to make the communication possible between the thread of the controller and of the view we used some events and invoke methods we will explain in more detail in the implementation part.

c. Message data structure

Even though there is only one class representing the messages, there are different types of messages running through the application.

For the messages received by the server: (receive() in Session.cs)

- Login message, when the users sign up or sign in, he sends his identifiers
`message.Room == null && message.Content[0] == "signin"`
- Subscribing message, when the user wants to join a new room
`message.Room == null && message.Content[0] == "signup"`
- Room adding message, when the user wants to create a new room (he automatically subscribes it)
`message.Room == null`
- Classic message, sent by a user to a room of users.

When all the previous conditions are false, we know this is a classic message sent by the client.

For the messages received by the client: (receive() in Client.cs)

- Server message, when the sender equals 0
`serverMessage.Sender == 0`
- Room adding, when the content of the message is add
`serverMessage.Sender == 0 && serverMessage.Content[1].Equals("add")`
- Room removal, when the content of the message is remove
`serverMessage.Sender == 0 && serverMessage.Content[1].Equals("remove")`
- Classic message, when the message doesn't enter in the previous conditions

III. Implementation

We will now explain more in details, some specifics algorithms we made and that we thought useful to be included in the present report.

a. Sending-Receiving Message

The first thing to do to communicate and be able to send/receive messages is to connect server and client side. In this aim, we use TCP-IP protocol, and a socket, as you can see in the next picture.

```
sock = new Socket(
    AddressFamily.InterNetwork,
    SocketType.Stream,
    ProtocolType.Tcp
);

IPEndPoint iep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 1212);
sock.Connect(iep);
```

Once the socket is connected, we can start the thread that will handle the communication.

```
thread = new Thread(receive);
```

Once it's done, we create a stream that allows us to send/receive messages. Then, we use the method `serialize` to send a message, or `deserialize` to receive it.

2 références | JeKoCorp, il y a 39 jours | 1 auteur, 1 modification

```
public static void Serialize(Message m, Socket s)
{
    NetworkStream ns = new NetworkStream(s);
    BinaryFormatter formatter = new BinaryFormatter();
    try
    {
        formatter.Serialize(ns, m);
    }
    catch (SerializationException e)
    {
        Console.WriteLine("Failed to serialize the message. Reason: " + e.Message);
        throw;
    }
    finally
    {
        ns.Close();
    }
}
```

2 références | JeKoCorp, il y a 27 jours | 1 auteur, 2 modifications

```
public static Message Deserialize(Socket s)
{
    NetworkStream ns = new NetworkStream(s);
    BinaryFormatter formatter = new BinaryFormatter();
    try
    {
        return (Message)formatter.Deserialize(ns);
    }
    finally
    {
        ns.Close();
    }
}
```

b. Communication with GUI thread

In this very interesting part, we are going to see how a simple thread can communicate with the GUI, in order to stay thread-safe.

At the beginning of the project, we faced a thread-safe issue : the thread which were receiving message was trying to access some components, like a textbox, inside the winform, which is running by GUI thread. This issue was creating a crash of the application when a textbox was accessed by message receiving thread, when the GUI thread was still creating the winform. How did we handle it ?

The solution that we implemented, was to use an event in the message receiving thread, which could detect the receipt of the message, and could call the GUI thread to modify the GUI. We used call back methods (invoke).

```
public delegate void ReceivedHandler(Client c, Message m);
public event ReceivedHandler receivedEvent;
```

Here we create the event and the delegate associated to this event in the message receiving thread.

```
receivedEvent += new ReceivedHandler(f.Afficher);
```

Now, we subscribe, or link, a method to the event.

```
2 références | JeKoCorp, il y a 39 jours | 1 auteur, 1 modification
protected virtual void raiseReceivedEvent()
{
    if (receivedEvent != null) //Check for subscribers
    {
        receivedEvent(this, serverMessage);
    }
}
```

Above is the virtual method that checks the methods which subscribed to the event, thanks to the delegate. This raiseReceivedEvent() allows to raise the event. We use it when we detect the arrival of a message from the server on the client side.

```
delegate void SetTextCallback(object sender, Message m);
```

Now, we have an other delegate that is created on the GUI thread, which is used when we want to use call back methods such as invoke instructions.

```
public void Afficher(object sender, Message m)
{
    if (this.discussion.InvokeRequired)
    {
        SetTextCallback d = new SetTextCallback(Afficher); //Thread safe : ajout texte à textbox
        this.Invoke(d, new object[] { this, m });
    }
    else
    {
        //Affichage (code assez long...)
    }
}
```

Finally, we can see here the using of call back method. We understood that the message receiving thread is asking the GUI thread to update the GUI, and it's waiting till the GUI thread finished the operation. We also could use BeginInvoke() method, which works a little bit differently. BeginInvoke() method puts the instructions that's GUI has to execute in a queue. This way, the message receiving thread hasn't to wait the end of the GUI updating operation, which is executed by the GUI thread, and can go to the next instructions.

In the case where we just wanted to secure a resource between two threads, we used a mutex to lock the resource. It allowed a resource to be accessed one thread at a time.

When the user wants to switch room, we store the conversation in a class. The textbox in the winforms are limited in the number of characters that it could contain. So we implemented an algorithm that could store something like 145 messages, which is the number maximum of full messages (a message can contains like 200/300 characters).

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    if (previousIndex != listBox1.SelectedIndex)
    {
        string room = listBox1.GetItemText(listBox1.SelectedItem); //Recupere le nom de la room selectionnée
        int index = Room_client.getIndex(room); //Trouve son index

        if (Client.rooms[index].subscribe == false) //envoi du message subscribe pour la room nouvellement selectionnée
        {
            c.send(new Message(new List<string>() { room }, null, 0));

            Client.rooms[index].subscribe = true; //On active la room chez le client
        }

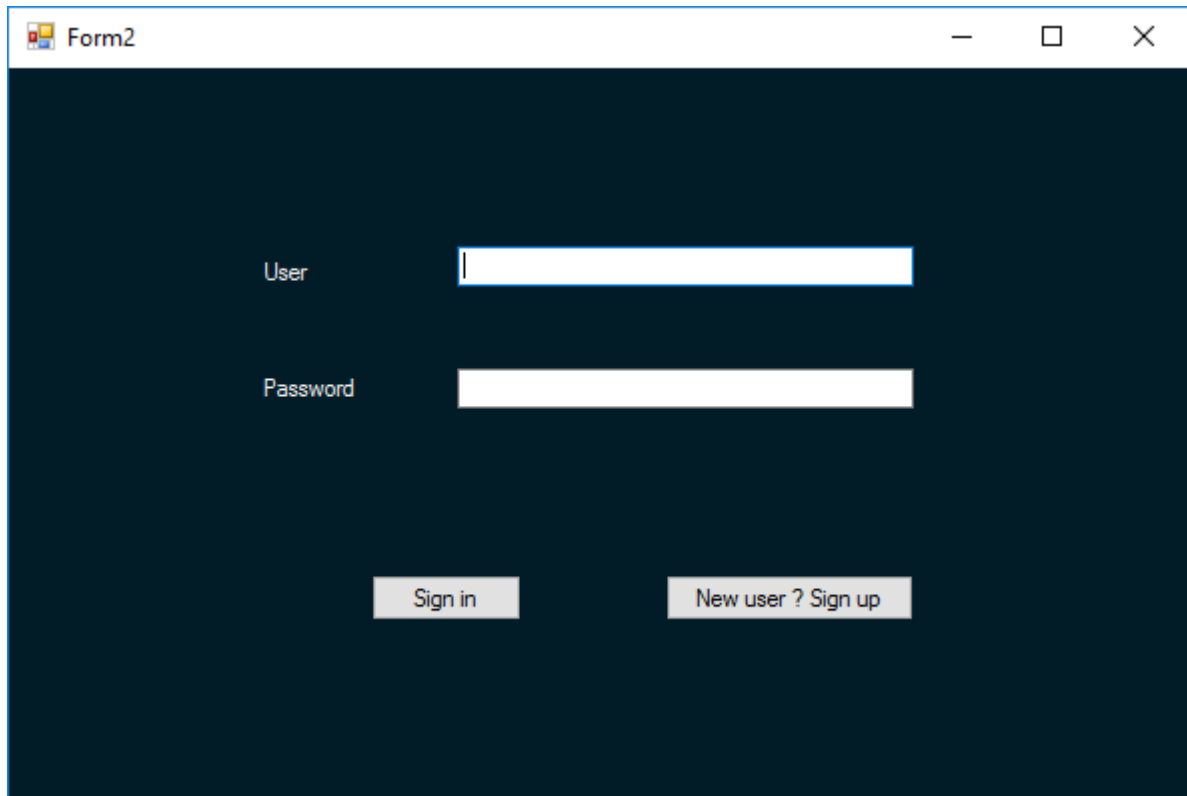
        texte.Focus();

        lock (discussion)
        {
            Client.rooms[previousIndex].save(discussion.Text); //Sauvegarde la discussion dans la mémoire
            discussion.Clear();
            discussion.AppendText(Client.rooms[index].chatbox); //Affiche son texte
            previousIndex = listBox1.SelectedIndex;
        }
    }
}
```

```
private void clearDiscussion()
{
    int lines = discussion.Lines.Length;
    int linesMax = 145;
    if (lines > linesMax)
    {
        int difference = lines - linesMax;
        discussion.Lines = discussion.Lines.Skip(difference).ToArray();
    }
}
```

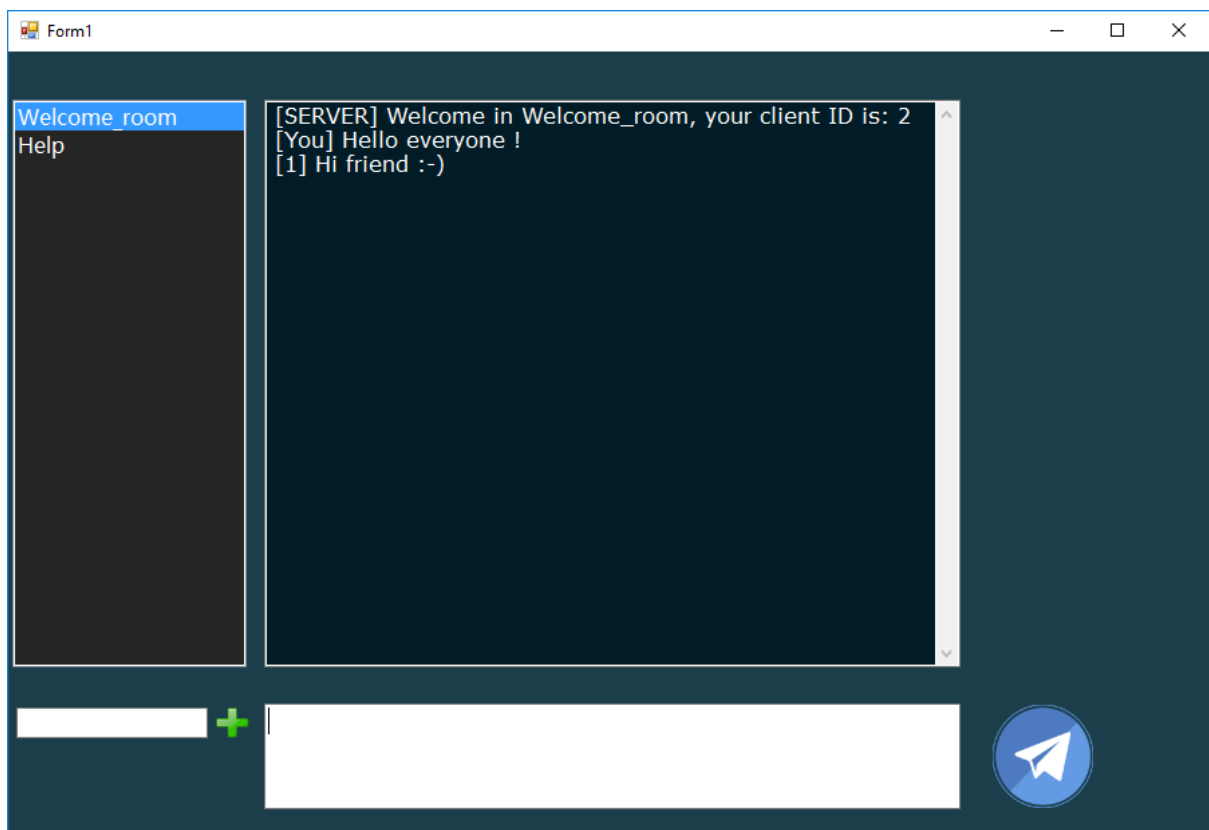
IV. Demo & screenshots

a. Client



The screenshot shows a window titled "Form2" with a dark blue background. It contains two white input fields: "User" and "Password". Below the "Password" field, there are two buttons: "Sign in" and "New user ? Sign up".

Figure 1: Login form



The screenshot shows a window titled "Form1" with a dark blue background. On the left, there is a sidebar with a "Welcome room" header and a "Help" button. The main area is a chat history displaying messages from the server and other users. At the bottom, there is a message input area with a green plus icon and a blue circular button with a white paper plane icon.

Form1

Welcome room
Help

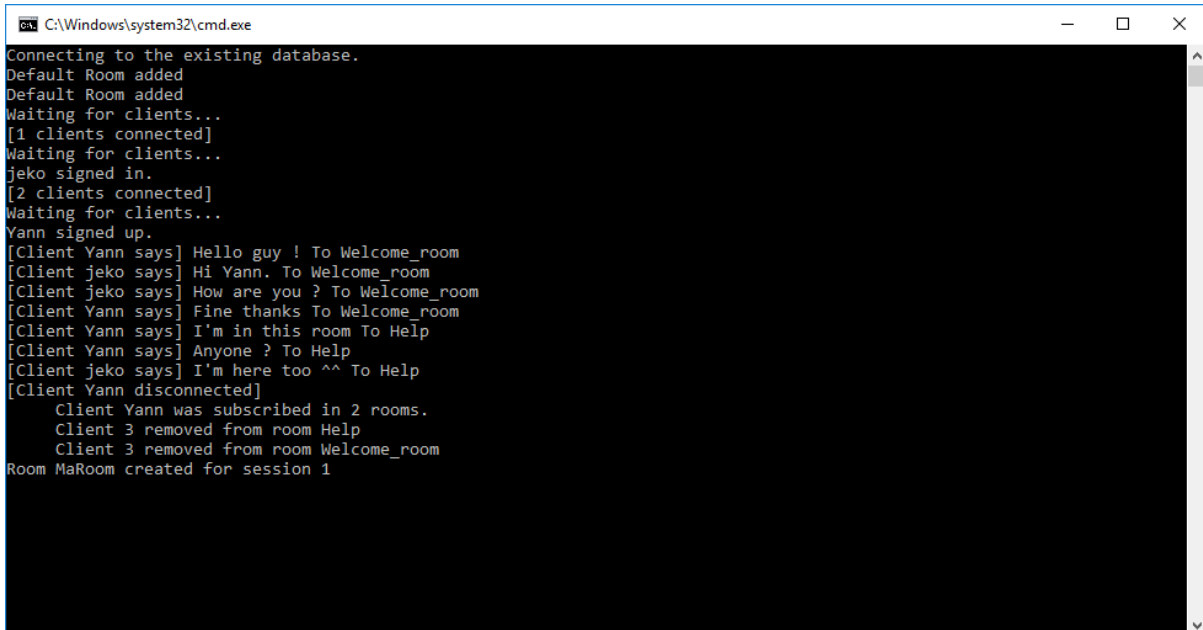
[SERVER] Welcome in Welcome_room, your client ID is: 2
[You] Hello everyone !
[1] Hi friend :-)

+ [input field]

[send button]

Figure 2: Chat room main window

b. Server



```
C:\Windows\system32\cmd.exe
Connecting to the existing database.
Default Room added
Default Room added
Waiting for clients...
[1 clients connected]
Waiting for clients...
jeko signed in.
[2 clients connected]
Waiting for clients...
Yann signed up.
[Client Yann says] Hello guy ! To Welcome_room
[Client jeko says] Hi Yann. To Welcome_room
[Client jeko says] How are you ? To Welcome_room
[Client Yann says] Fine thanks To Welcome_room
[Client Yann says] I'm in this room To Help
[Client Yann says] Anyone ? To Help
[Client jeko says] I'm here too ^^ To Help
[Client Yann disconnected]
Client Yann was subscribed in 2 rooms.
Client 3 removed from room Help
Client 3 removed from room Welcome_room
Room MaRoom created for session 1
```

Figure 3: Server window

V. Possible enhancements

Some possible enhancements that we can add later to the program would be:

- The creation of a private messaging system, which would be quite easy as it would be nothing more than a “special” room with max 2 users subscribed.
- We also thought about displaying a list of all the users connected in a room on the GUI
- For the security during messages transmission over the network, it would be an interesting feature to some encryption to the message.
- A last idea, maybe a bit more crazy, would be to implements some bots. This would allow to have a better interactivity with the server, which would be able to do more than just sending or receiving messages.

To conclude, we started this project with a bit of apprehension about how we would manage to create a network based chat. We discovered the sockets and we had the opportunity to use the threads and the events, which we didn’t really used before this project. We also discovered new important technologies such as GitHub or C# language.