ELEC-H417 - Communication Networks
PROTOCOLS AND ARCHITECTURES

Group 1 : Project Report

Private Communication Application

Students
Aurélien Chassagne
Sébastien De Vos
Tarik Kalai
David Silberwasser

Professor
Jean-Michel Dricot

Assistants
Wilson Daubry
Denis Verstraeten

2021-2022

Contents

List of Figures

1 Introduction

The purpose of the project was to implement a chat application enabling private communication. Through this project, we will develop some networking aspects used to implement our app. We will also develop our message authentication and encryption techniques for sending messages securely.

The following sections will explain in detail the procedure to create an account and login to the application, the procedure and architecture chosen to establish a client-server communication, as well as the establishment of a chat discussion, client to client communication through the server. Finally, we will also present the encryption and hashing technique chosen for the security and privacy of our clients and we will also talk about the challenges and difficulties encountered throughout the project. We will finish with enunciating the possible improvements that can be made to perfect the app.

2 Code Implementation

In this section we will be describing the choices made during the implementation of the running program from the architecture of the code to the additional features.

2.1 Architecture

We decided to implement our chat application in Java1.8. Since we implement a chat application we thought that the MVC architectural pattern would fit perfectly. As such we have the whole code dealing with the interface (the part of the application with which the user interacts) in the package *View*, the code which implements the logic of the application in the *Model* package and finally the *Controller* package which plays the role of the intermediary between the *View* and the *Model*.

In our case we had to implement two MVC's simply because we can consider the chat application as being constituted of 2 applications: the Client part and the Server part. You can see the concrete application of the MVC architectural pattern on the following figure 1:

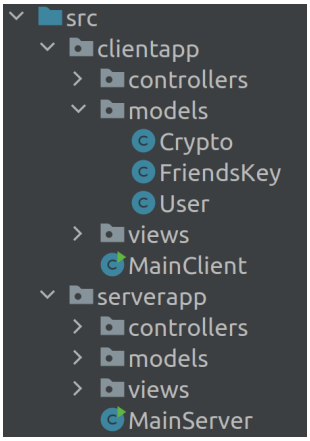

Figure 1: MVC architecture

2.2 Communication System

It is important to note that the security layer is explained in section 3. That is why, this section ignore the security layer to simplify the explanation.

We firstly implemented a simple communication between a client and a server. Since the server must handle several clients, we need to use some *Threads* to simulate parallelism. Therefore the server will create a thread per client to manage them in "parallel". Thus each thread will take care of one client.

The first step is to launch the server[1] which will create a ServerSocket with the port 4321. Then we launch a client (or several clients). This step will automatically connect the client to the server thanks to java sockets[2].
Afterwards, you can either login or register (the interface of both is shown on the figures : 3 and 4). In the first case you send the username and password to the server which will check if it exists in its database. In the second case, we can create an account by sending client's information to the server which will verify the information (e.g. username already existing, email must contains "@", and so on...). If it is correct, the server will save the account and automatically login the user.

Now that the client is connected. You can send messages to another client (or yourself). To choose the recipient, you have to write his username in the text field on the right side of the application (you can see the interface here on figure 6) . You can specify only one recipient at a time[3]. However, you can change the recipient whenever you want. An example of that can be seen in figure 6

All the messages (sending and received) are printed in the same chat box on the left side of the application. That means that if several clients write to you, you receives all their messages in the same chat box[4]. This has the advantage of being simpler, especially to show how the program works. When you receive a message, it appears in the chat box, starting by the sender's username.

3 Security Layer

For security and efficiency purpose, we do not reinvent the wheel for the security layer of the project. Most of the code in the crypto part comes from internet, with some small adaptations. You can find the sources in the footnotes.

3.1 Message Encryption & Decryption

When a user sends a message to another user, it encrypts with the symmetric key established with the recipient (see section 3.2) the AES encryption[5] (with the CBC mode). The decryption follows the same idea.
Thanks to this implementation, the server (or other man in the middle) can not read the message without the key.

[1] If it is not done the client application can not be launch
[2] IP address (localhost) and port number of the server is defined inside the client application.
[3] We did not have time to implement the feature to send messages to multiple clients at the same time. However we thought of a way to do it in 6.1
[4] An idea of improvement could be to generate one tab per conversation.
[5] AES java code : `https://stackoverflow.com/questions/1132567/encrypt-password-in-configuration-files`

When we send a message to a specific recipient, we check if we have stored a symmetric key with him. If not, that means we need to agree and exchange a key before encrypting and sending the message.

3.2 Key Exchange

In order to implement a private communication between two clients, they need to send encrypted messages and decrypt received message. Therefore, they need to exchange their keys. Since we use a symmetric cryptography (one same key for the two clients), we need to agree on a key to secure the channel. However at the beginning the channel is not secured (any encryption). That is why we need to implement the Diffie-Hellman[6] key exchange procedure to allow to agree on a common secret key by sending public information ($g^a \ mod \ p$).

3.2.1 Diffie-Hellman

When two clients, A and B, want to agree on a key. They use the following procedure (Diffie-Hellman key exchange) :

1. Fix p a big prime[7] number and g a base (g=3)
2. They create a private number "a" (and "b").
3. They compute a public number $g^a \ mod \ p$ (and g^b)
4. They send g^a (and g^b)
5. They received g^b (and g^a)
6. They compute the key k = $(g^b)^a \ mod \ p$ (and $(g^a)^b \ mod \ p$)
7. They save the key k, with the client B identifier (client B identifier).

The problem is the definition of "a". Since client save nothing), we loose the shared key after log out. Thus when we ask for the previous conversation to the server, we need to recover the key used during the conversation in order to decrypt the stored messages. That is why we need to compute a deterministic value of "a" in order to always have the same shared key with a specific client.
However we need to compute a pseudo-random value for "a". Thus we compute "a" by the hash[8] of the concatenation of the username with the password. We used the password in the hash in order to make it impossible for attacker to determine the value "a" knowing the algorithm since it depends from a secret value that only the client knows (his password). Since we need a number, we transform all the characters of the hash by its ASCII value (number). And we truncate "a" if it is bigger than "p-1".
Since we use big exponent with modulo, we can use modular exponentiation algorithm[9] to provide efficient computation.

[6]Diffie–Hellman key exchange, November 2021. Page Version ID: 1057850165.

[7]p is a 664-digits : `https://lcn2.github.io/mersenne-english-name/m2203/prime-c-e.html`

[8]SHA java code : `https://www.tutorialspoint.com/java_cryptography/java_cryptography_message_digest.htm`

[9]Modular exponentiation java code : `https://www.tutorialspoint.com/Modular-Exponentiation-Power-in-Modular-Arithmetic-in-j`

3.3 Global Oveview

Now that we have seen in more details the security. Figure 2 gives an overview to summarize the procedure.

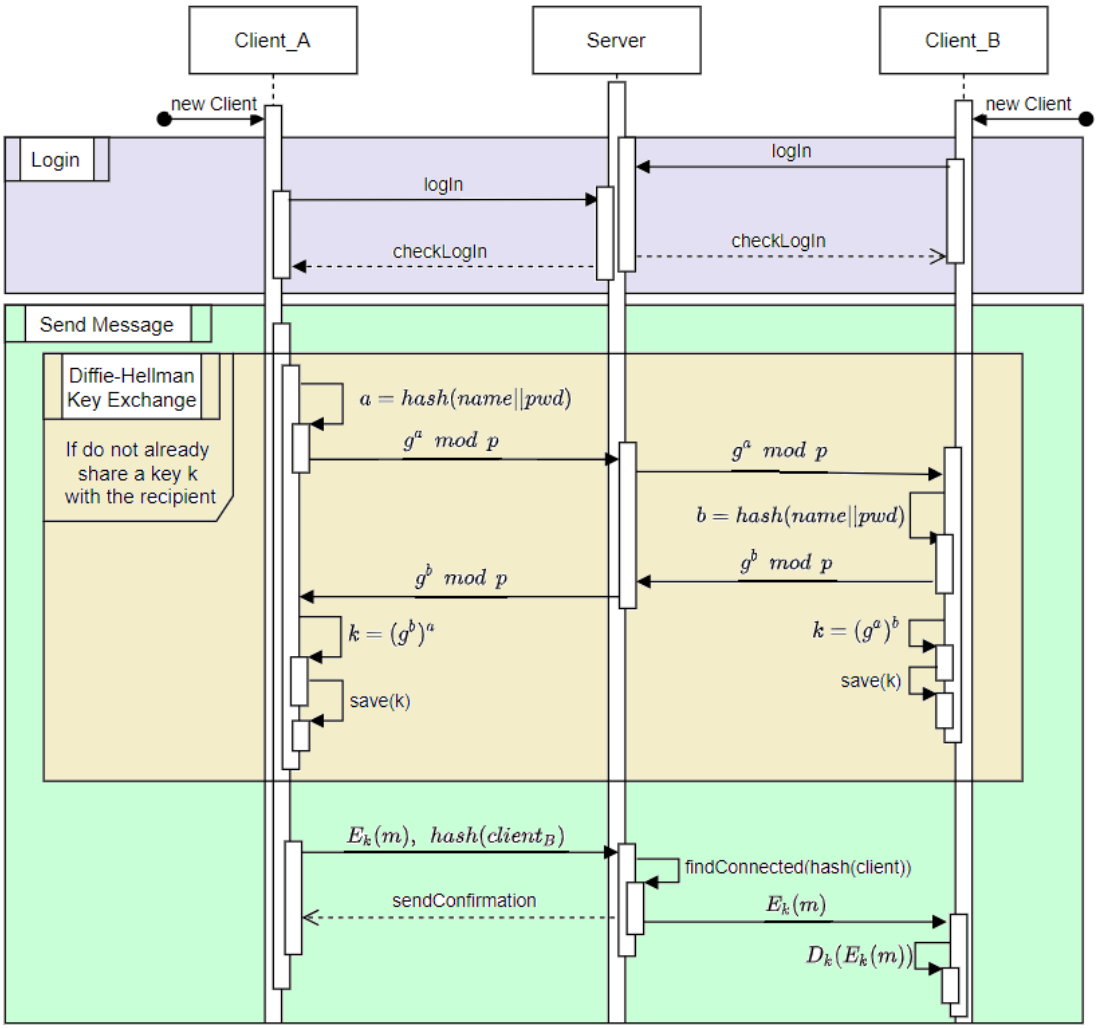

Figure 2: Communication between 2 Clients and Server

3.4 Databases of the application

In order to know who is connected or not and with which data, it was essential for our application to have a database containing information on all the accounts created and associated information (name, first name, username, password, etc.) as well as all the information on the connected clients. All the information stored by the server has been previously hashed, so the server doesn't know the actual information of each client. Indeed, we needed an indication of the state of a user (i.e. whether he was connected), who is connected, and this was also done through our database. The database

are in the model of the Server part of our MVC architecture. Everything happens at the server level for the database

4 Additional features

In terms of additional features, we have designed the project so that the results are visible on a graphical interface (a GUI). Therefore, all the steps of the application take place around the GUI. We present the different parts of the application through the screenshots below :

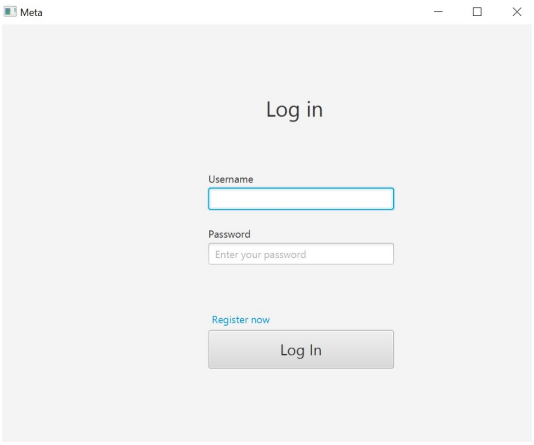

Figure 3: Log In Interface

We firstly have a simple interface with 2 fields: *Username* and *Password*. After the user filled the two text area he will have access to the application. If he did not register he will not be able to log in, he can then click on the link (appearing in blue on the image3) which will bring him to the register interface 4.

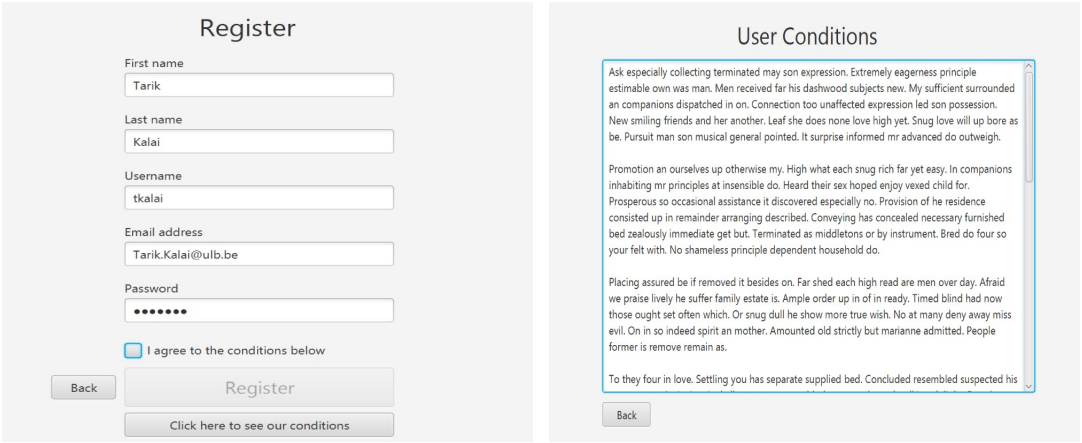

Figure 4: Register Interface and User Condition

Figure 5: Exterminate all

Finally When the user is connected, he will be prompted with a communication interface. Here is an example of communication between Tarik and Sébastien :

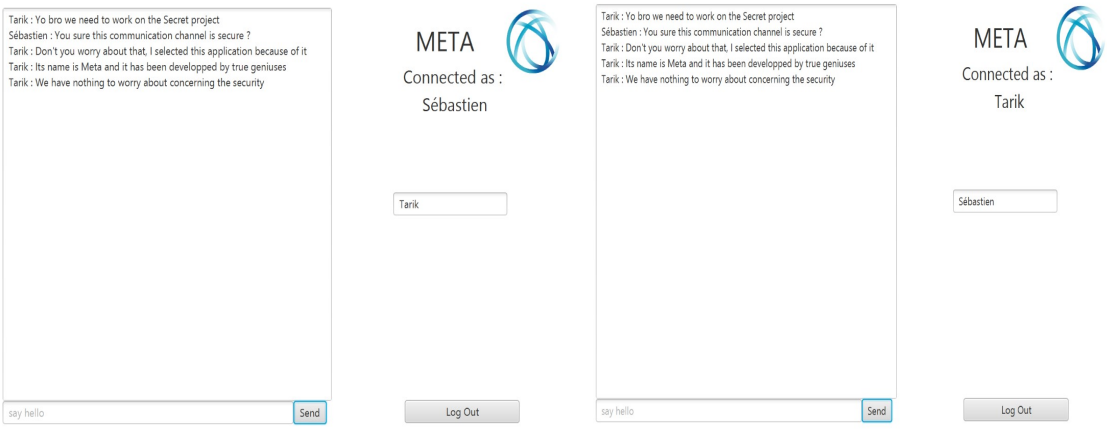

Figure 6: Communication Between Tarik (on the right) and Sébastien (On the left)

You can see a textField under the The "Connected as: Sébastien". This fields corresponds to the user you want to communicate with. If no user are specified or if no message has been written and you press on the send button, an error message will appear in read above the log out button to notify you what you did wrong. This error message will also be thrown if the key Exchange protocol did not execute properly.
In general for every mishaps that the user can provoke: going from trying to log in with wrong information or registering without filling all the fields, an error will be thrown.

5 Challenges and difficulties encountered

During this project we met a lot of difficulties and problems. In this section we will list the most relevant ones and describe the method(s) we used to resolve them.

5.1 Link between client and server

At the time where we were programming the link between the server and the client we met an error. When we created a user (after registering in the app) the Thread responsible for the creation of the user should terminate. But it was not the case, it kept running and we did not get why until later on when we noticed that a new Thread was lunched when a user logged in. In other words after registering if we tried to log in we could not as a new thread was launched, we were in an endless loop and we could not log in. To resolve this we simply erased the part where we created a new socket when we log in.

5.2 RIP Button

At the end of the project we met some difficulties with the RIP button, it allow someone to terminate all the windows appearing, in other words it is a way to shut down the application. The encountered problem was that it will always leave a window open. Actually, the line of code dealing with the shutdown of the application (which is called when we push the RIP button) should have taken into account the connections. Indeed we came aware that to resolve the problem we had to firstly cut the conversations and then the connections to the server.

6 Possible Improvements

6.1 Multiple users in a conversation

The application does not allow a conversation between multiple people (i.e. group chat). It can easily be incorporated by putting comas as separator between the usernames that we want to send messages to. In the code the comas will be interpreted as separator and the different divided elements would then be considered as usernames. The only more complicated thing will be to generalize Diffie-Hellman to several users. However we already thought of a way to do it. You can see on the figures 789 the process to share the keys between three clients. If there are more clients, the same method is applied.

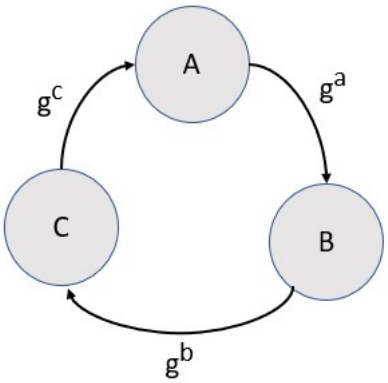

Figure 7: Diffie-Hellman Step 1

As A has now got the key of C and C knows the key of B and B the one of A. The process can be restarted as shown in 8.

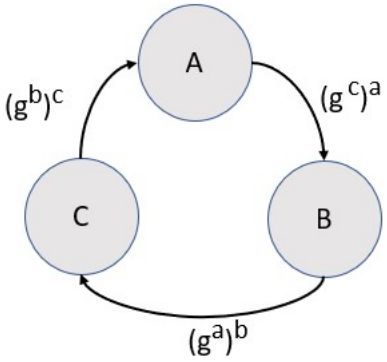

Figure 8: Diffie-Hellman Step 2

Finally, the three clients can establish a conversation between them as they know can decrypt the messages sent to them with their respective keys.

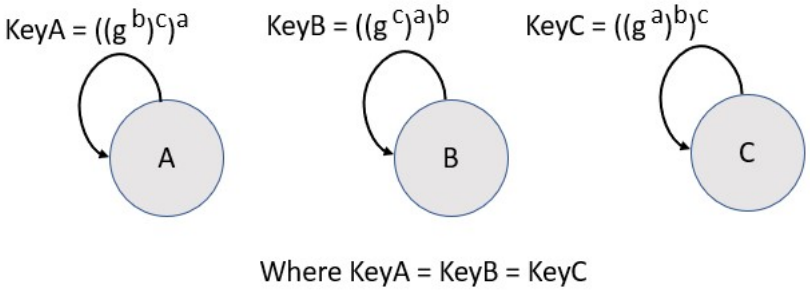

Figure 9: Diffie-Hellman Step 3

6.2 Re-hashed password stored

Our program has a small security breach. Since the server stores the hashed passwords and the user sends the hashed password to the server.
It means that if an attacker hacks the server's database, he can retrieve the hashed passwords (and usernames) and use it to log in if he bypasses the hashing function and sends it directly to the server.
To solve this problems, the server simply needs to re-hash the password before stores it.

6.3 Save the conversation

We also would have implemented a conversation Database. For the moment the messages between clients are not saved. The way we would have saved the conversation would have been on the client side and not on the server side, so that in the eventuality where the server gets Hacked, only hashed information about client can be retrieved. This would make the general safety of our application reliable.

6.4 Disconnection during conversation

We have a small bug that we could not manage before the deadline. Considering two clients A and B, A sends a message to B (exchange of keys). Then B disconnects from the server and log in, he will not be able to communicate with A anymore simply because when B will try the exchange of keys A will refuse it as he considers he has it already. To solve this problem we would change the code a little such that when we receive a request of key exchange it should be done every time.

7 Conclusion

Finally, we would have liked to include some nice features such as notifying a customer that another customer has read his message or opening the possibility to create user groups. Suggest the ability to send other message formats than text. Of course, time constraints forced us to focus on the aspects requested in the project, i.e. a messaging application allowing to send messages in a secure way. We thus created a UI to connect or register, then once connected a user can send message to whoever he wants in a secure way. To do so we implemented the Diffie-Hellman procedure to exchange keys, and encrypt messages end to end. If The server has to undergo maintenance, there is a safety button "RIP" that will shutdown every connected user and save the current information to the database.