ECOLE POLYTECHNIQUE DE BRUXELLES

ELEC-H417

Project: Chat app enabling private communication

Authors :

De La Fuente Perez Javier
Hillergren Sandra
Keci Fjonja
Trouillez Benoît

Professor :

Dricot Jean-Michel

2021-2022

Contents

1 Introduction

The goal of this project is to design and implement a chat app enabling private communication. A server should manage all the communications between several clients. Moreover, security must be implemented ; for example, even if the server is manipulating all the messages, it should not be able to read the content of the messages. Another security principle musts be applied to connect a client.

Several programming languages can be used to implement the latter application. In our case, Java was used to code the program since some of us had already use it in previous projects and, as a lot of other languages, Java provides libraries that can be useful in our project.

As detailed later, two major classes are implemented : Server and Client (which are runnable classes). In addition to this, two other classes, *ServerRun* and *ClientRun*, have been implemented and allow instantiating one single server (instance of *Server*) or one single client (instance of *Client*) and the running of this latter. Note that *ClientRun* should be ran once for each client since it creates and "launches" only one object of the class Client.

2 Classes and architecture of the code

2.1 Classes description

Before talking about the actual communication between users throughout the server or even looking at the global architecture of the code, let's look at the different class in order to understand better the whole code.

Class ServerRun
This class has the same purpose as the previous one : it instantiates a server and assigns to it chosen port and paths in the memory where the server itself will store several information such that the usernames, hashed passwords, the encrypted conversion and so on.

Class ClientRun
This class only contains a main function used to instantiate a client and assign to it chosen host, port, and memory paths for the storage. Since those parameters are sent in the constructors, they become private attribute of the client. After instantiating the client (instance of the class *Client*), it runs it by launching the *run()* method of the client.
Note that to make a client connect with a desired server (created before the client obviously), we must make the port number match with the one of the server [1].

Class Server
The Server class has a socket and a port. It mainly has access to multiple databases to store the usernames, the hashed passwords, the connection status, and the past conversations. This object will be the central element for communication between users : each packet that needs to go from one client to another must pass through the server. As it will be shown below, it starts one thread whenever a client connects to the server (and stop this thread when this client disconnects) and its job is to manage the requests of the client and acts depending on them.

Class Client
This class describes the client of a user (i.e., the intermediate between the server and the user interface). Since a user has a username, the corresponding client is linked to it through a private attribute *username*. Moreover, other private attributes such as its connection status (logged in or not), its current chat contact are present. Each user must have a socket (endpoint for communication) and its port, and will have 2 readers[2]: one to "read" what is the physical user typing on its keyboard, and another to read the messages coming from the server directly derived from the socket mentioned before. It also needs to have a writer[3] which will enable the user to communicate through the server. It will be explained below that a client will run two threads : the first one allows the listening to the server since the second one listens to the user requests (using its keyboard).

[1]In our code, we have chosen *PORT = 1234*.
[2]We decided to use BufferedReader objects for that.
[3]We decided to use a PrintWriter object for that.

It was not mentioned in the description of the class *Server* but since a server also needs to read requests from clients and write response to them, it also has BufferedReader objects and PrintWriter objects. In fact, those are local variables present in each thread : one client manager has its own BufferedReader object to read the corresponding client and one PrintWriter object to send responses to the latter.

2.2 Architecture

The architecture of the code is represented below :

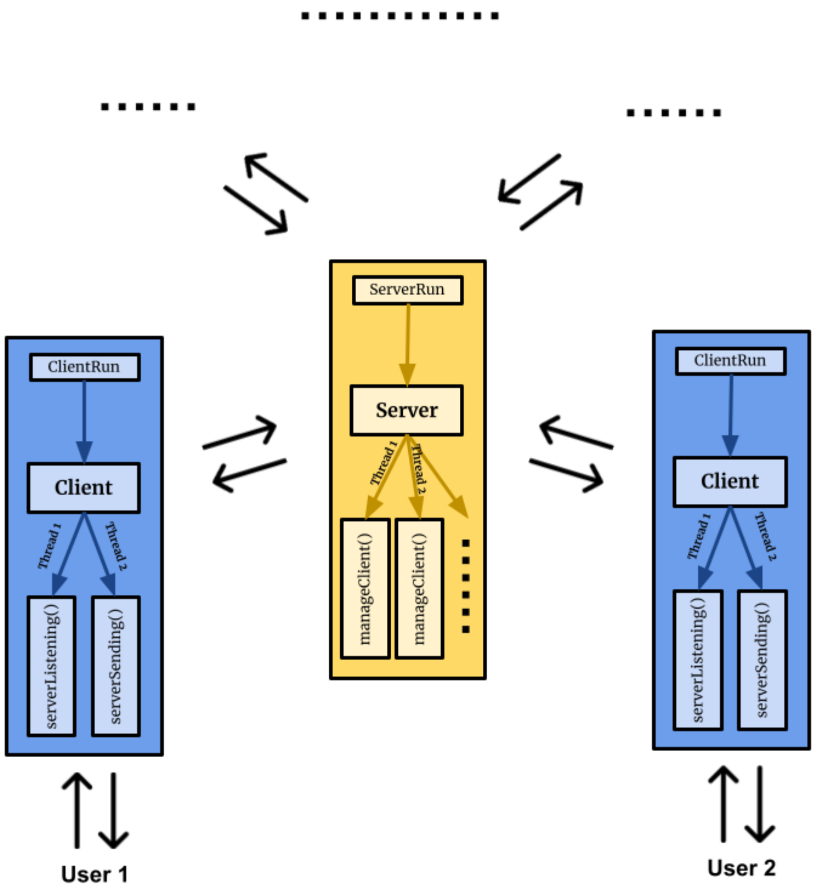

Figure 1: Architecture of the code

As mentioned before, *ServerRun* and *ClientRun* are used to run a server or a client. Since all the clients are managed by the same server, only one server needs to be run. The clients cannot send packets directly between them, they need to communicate with the central server to do this.

On the server side, we can see that for each client connected to it, one thread is running to manage the latter. This means that whenever a client socket is detected by the server socket, the server accepts it and runs a thread especially for this client socket; this thread (which runs the method *manageClient()*) will then only interact with the corresponding client. Therefore, when a connected client decides to disconnect, its socket closes the communication with the server and the latter stops the corresponding thread. The number of threads is thus equals to the number of interactive clients with the server.

On the client side, threads are also launched. Since a client is always having two interactions at the same time, the one with the server and the second with the user interface, it musts contain two different threads. The first one listens to the server socket and acts depending on the request of the server (for example, the server sends an encrypted message from another client and the thread should handle it and display the decrypted message on the interface) since the second one do the opposite which means listening to the user interface and acts depending on the user request (for example, the user wants to send a message to another client and the thread should communicate this to the server socket). Those two threads are called *serverListening()* and *serverSending()* in our code.

2.3 Program operations

Let's explain now the actual communication between users through the server and thus how to code work. To do so, we first consider one client connecting to the unique central server

First, *ServerRun* must be used to instantiate a server (only one because there is one server). As we enter the run function of the server, the server is waiting for a client to connect. After this, we run *ClientRun* to create the first client. We must make sure that the port numbers match as discussed before in section 2.1. Since the server detects the client, it accepts is and run a manager for this latter : the *manageClient* function is called (taking in argument the client socket) and it starts by proposing to the client to sign up or log in, and waits for its answer.

2.3.1 Signing up

In the client object, as the *loggedIn* attribute is initially false, the user first needs to sign up (considering he has never registered before). The username in plain text and the hashed version of the password (this one as a string) is sent to the server. Once this done, the user is correctly logged in (the *loggedIn* attribute becomes true) and the user is ready to potentially start a conversation with someone. On the server side, since the client has signed up and gave its username and hashed version of the password, the server stores this data in its database. This stored information is more than important when the user wants to login to its registered account, as detailed in the next section.

Note that there are some conditions to successfully sign up: the username must not have blank spaces and it must not belong already to the server database (conflict of same username for two users).

2.3.2 Logging in and Authentication

Imagine the user disconnected and comeback to the applications after a certain time, he would like to login to its previously registered account to talk again to its friends and recover its own conversations. He will then choose to log in when the server asks its means of connection.
The server first checks 2 conditions : the username is in its database and the user linked to this username isn't already connected.
If these 2 conditions are validated, then the server proceeds to authenticate the user, by using a challenge-response authentication. First, the server generates a nonce and sends it to the client. Then, the user writes its password and hashes it, adds up the nonce received and hashes all together before sending it to the server. The server, storing the hashed password received when the user signed in, contacts the latter with the generated nonce. The server can then compare the hash response of the client and the one generated by himself ; they must both match to authenticate the client. Otherwise, the server keeps on trying to authenticate the user by sending a different nonce.
The following diagram represents the authentication scheme used in our project :

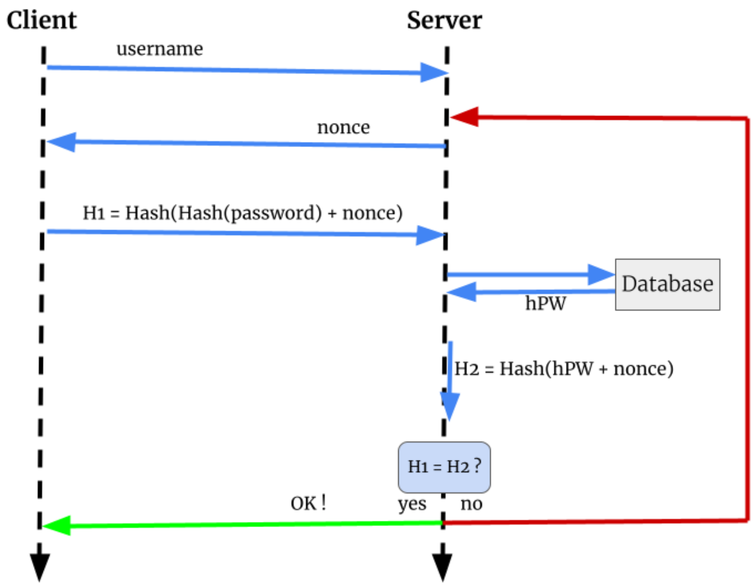

Figure 2: Authentication scheme used in the program - Diagram

This authentication scheme implies one of the principles of network security and it prevents an intruder to replay the user password to steal his identity.

In our case, authentication is not made between two clients communicating, but throughout the server. The first case would be the Authentication Protocol ap4.0 seen in the course, where the nonce, instead of being hashed, is encrypted with the shared key between the 2 clients.

2.3.3 Conversation between two users

In the following paragraphs, let's call A the user who wants to start a conversation with another client and B the latter.

After successfully logging in, the user A can choose to talk to someone in particular (B). To do this, the user A should use a certain command followed by the username of the destination user (B). The server then checks if this username is in its database (i.e., if the destination B exists and is connected). If it is the case, then the server validates to the source (client A) that he is starting a conversation with the destination and tells the destination client (client B) that the client A started a conversation with him.
To ensure secure communication, both clients now need to agree on a symmetric shared key that only them know. The client first checks if a symmetric key does not already exist with the other client ; if it is the case, it means that they already communicated, and they should not generate again a symmetric key[4].

First conversation between user A and user B

Let's take the case where it is their first time starting a conversation between them, then a symmetric key must be shared securely.
The Diffie-Hellman Key Exchange has been chosen to provide this secure exchange and an explanation about this exchange is given in the course. In order not to go into details, the steps of the Diffie-Hellman algorithm are given below and are followed by a diagram :

1. Both clients generate a pair of public and private key.
2. Both clients send the public key to the other client through the server.
3. Both clients receive the public key from the other client and generates the symmetric key using his own generated private key and the public key of the other client.
4. Thanks to mathematical properties, both client A and client B generated key are the same hence the name "symmetric" key.

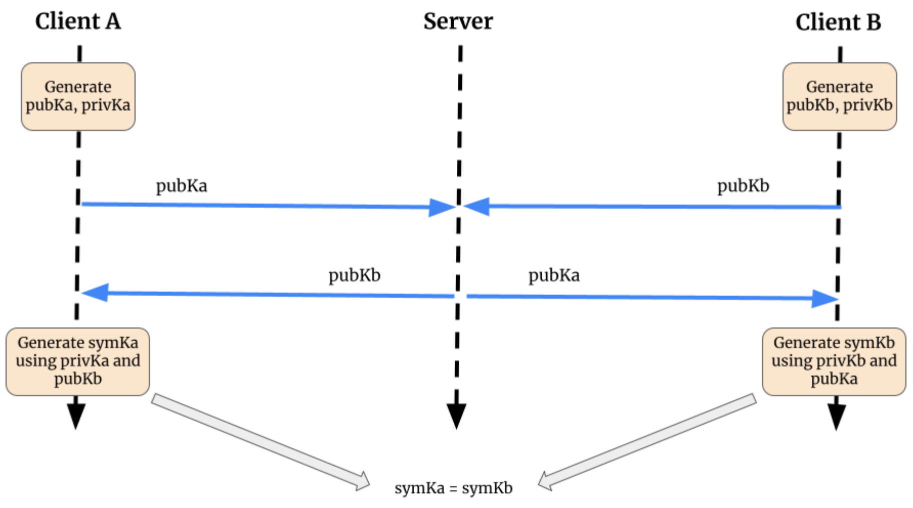

Figure 3: Diagram representing the Diffie-Hellman algorithm implementation in our project

[4]They must keep the same one to recover the messages stored before in the server database since those are encrypted with their previous symmetric key !

The shared symmetric key is finally stored in each client database so that when the clients want to talk back together latter on, the key does not have to be regenerated, and the previous messages can still be decrypted as detailed in the next point.

Now that the shared symmetric key has been generated on both sides of the communication (client A and client B), they can communicate securely since every message to be sent from one to the other is first encrypted at the source, flowing through the server who only see the cypher, and finally decrypted at the receiver where the initial message (i.e. the plaintext) can be read properly.

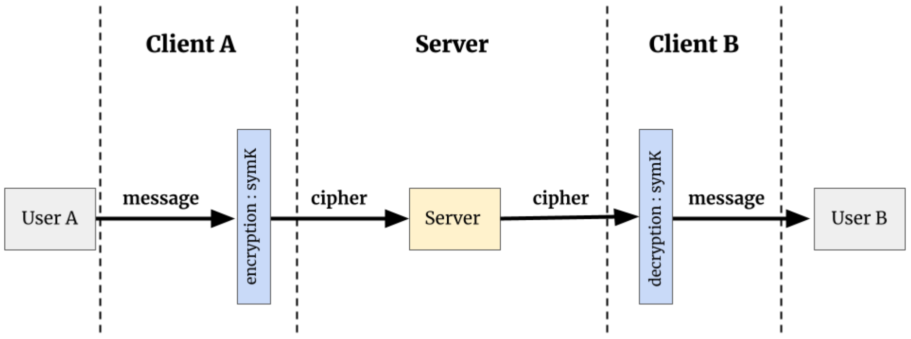

Figure 4: Encryption and decryption - Message from user A to user B

Resume a conversation between user A and user B

If the clients have already conversed in the past, it means that the key exchange has also already been done. The Diffie-Hellman is then not used since both clients only need to recover the shared key in their private database.
When the conversation resumes, the server provides all the previous encrypted messages to both clients so that they can decrypt them and thus get the history of the conversation. The acquisition of the history of the conversation can also be done using a certain command by a user as explained in the Innovations and creativity section

Extra comments

Note that when a user A wants to start a conversation with the user B, it does not mean that the conversation with the user A will automatically open on the user B. Although the key exchange will occur in any case, the destination user (B here) only receives a notification telling him that a conversation has started with user B. The user A can send messages, it will be stored, and the user B will receive notifications (and not the messages directly). The client B must tell the server that he also wants to talk to client A to see the conversation. The notifications feature is among others explained in the following section dealing with additional implementations and innovations.
For sure, the user can leave a conversation, go back to it later, start a conversation with another user, and so on.

3 Innovations and creativity

3.1 Additional implementations

Beyond the basic requirements, additional ones have been implemented :

1. **Get all the connected users**
   A first feature is that any user can ask the server to know all the other connected clients. By doing that, the user can know which other users are reachable at the moment.

2. **Get the friends list**
   The user can also access its friends list which contains all the users (connected or not) with whom he has already talked with, and which are not blocked (see feature 5).

3. **Get the history of a conversation with another client**
   Imagine user A has talked with user B a certain day and a few days later, user A would like to access the history of the conversation (even if user B is not connected). It is possible to do it using a single command, asking the server to provide him the history stored in the server database[5].

4. **Change the password**
   If a user is not happy with its old password, he can change it by using a command. For sure, the server will first authenticate him again before asking what its new desired password is.

5. **Block and unblock users**
   If another user is spamming user A or is unfriendly to him, user A has the right to block him which means that he will not receive any messages or notifications (see next feature) from this user ; thus, any messages or notifications coming from this user are ignored by the client of user A. The blocked user is stored in a blacklist of user A. It is possible for user A to unblock him and get all messages that the other user tried to send while being in the blacklist.

6. **Notifications**
   If user A is not talking with user B (i.e., talking with another user or just no conversation opened), if the user B tries to reach user A, the latter will receive a notification alerting him that user B started a conversation or sent a message.

3.2 Ideas for possible improvements and advanced features

Although basic requirements are met and some additional features have been implemented, there are other features that could be interesting to implement. We had a few ideas, but due to a lack of time, it was not possible for us to implement them in time. These are listed below, with the explanations about how we could implement them if we had more time :

[5]The stored messages are in fact encrypted, as already mentioned before.

1. **Make group conversation**
   A good improvement could be to allow users to communicate in group chats. First, a group name must be chosen by the creator of the group, this name will be the one used in the *currentConversation* private variable of the clients when they are talking in the group. When a member of the group wants to broadcast a message to the other members, it just has to send the message with the destination being the group name : the server can then handle the encrypted message and send it to all the members since it knows who belong to this group thanks to database storing.
   Let's first consider a 3 users conversation group : user A, user B and user C. A possible issue could be the following one : How to be sure that all the members are sharing the same symmetric key since the Diffie-Hellman algorithm only concerns two clients ? To ensure that, after a symmetric key has been generated using DH by two of the members[6] (client A and client B), one of the two, let's say client A, should encrypt the symmetric key of the group using the symmetric key he shares with the third client. The encrypted key is thus flowing through the server but is encrypted. When client C received is, it must decrypt it using the symmetric key he is sharing with client A and finally obtains the same symmetric key as the two other members of the group : all the members know the same private key, in a secure way.
   A simple diagram representing the steps for a 3 users conversation key agreement is displayed below :

   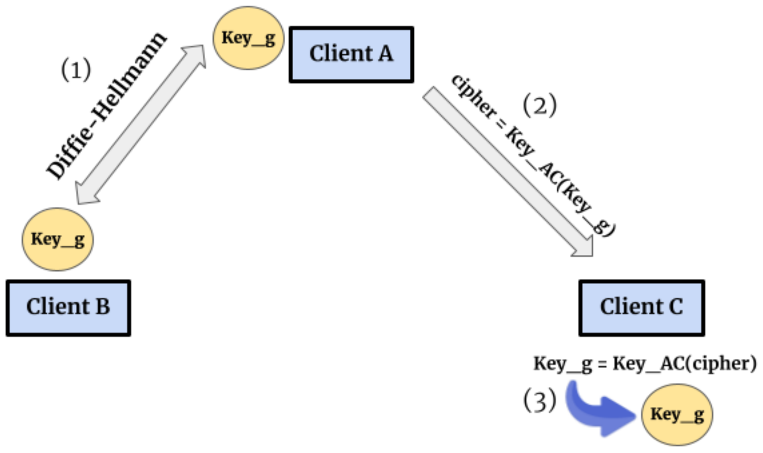

   Figure 5: Steps of a key agreement for a group of 3 users

   Note that in the diagram, the link between users still passes through the server. Therefore, the key is encrypted before being sent to the client C.
   For more members, it is easy to understand that the same principle can be used to share the group symmetric key to the joining members of the group. The latter can be added using a command : the server should just update its database containing the groups and their members.

2. **A more user-friendly interface**
   In our final application here, the interface for the user is just the terminal provided

[6]This key MUST BE different than the one generated for the private conversation between the two users !

by IntelliJ IDEA. Even if it works, the attractiveness of this latter is not the best one, a user-friendly interface could make the use of the application more enjoyable : for example, instead of using a command to start a conversation ("*TALK username*"), the user could just click on the name of the user on the interface using its pointer. Another example is that instead of writing and receiving messages in a terminal, an interface such as any messaging applications could be done containing messages on the left or right (if the message is received or sent) and a label available to write the content of the message to be sent.
Again, due to a lack of time, we decided to focus on the application features instead of the interface implementation, but we know it is possible using for example FXML files which provides composition of a GUI. Those kinds of files have already been used by us in a previous project.

3. **Digital signature**
   Another possible implementation to add would be digital signatures (with computational efficiency): to do this, every time user A communicates with user B, user A will send a pair of values : the message itself and the message signed with user A's private key. Therefore, when user B receives this pair, he can use user A's public key to unsign (or decrypt) the signed message and compare it with the message itself. If they are equal, then user B verified that he is really receiving messages from user A. We must not forget to encrypt with the symmetric key the pair of values sent to have confidentiality.

4. **Computational efficiency**
   To achieve computational efficiency, instead of signing the message (which could be quite long in terms of size), we sign the hashed message, which is fixed in terms of length, and send the message and this signed, hashed message to user B. User B can unsign the signed hash with user A's public key and compare it with its computed hash of the message.

4 Challenges

The biggest difficulty to challenge was the lack of time to finish the additional implementations. The chapter of security in networks was also studied in class a bit late and thus we had a small amount of time to understand the theory and put it in practice in the project.

Another difficulty we have met was the symmetric key agreement between two users when they decided to start a conversation. The Diffie-Hellman algorithm was well understood, but we first decided to try implementing the algorithm from scratch. After several trials and issues in the implementation, we found libraries in Java providing the possibility to generate the needed key instead of implementing them by ourselves.

5 Conclusion

This project was useful to put in practice some of the fundamental elements of network security, such as confidentiality, by way of encrypting and decrypting messages. We therefore saw both symmetric key and public key cryptography, by the way of the Diffie-Hellman key exchange, which allows two users that have no prior knowledge of each other to jointly establish a shared secret key over an insecure channel, by using its public and private keys.

Cryptographic hash functions were also seen, which are useful for achieving end-point authentication between user and server. However, we could have gone deeper and assure message integrity (also called message authentication) by using HMACs, which was simple and didn't require an encryption algorithm. We also could have implemented digital signatures, which requires notions of both public key cryptography and hash functions.