

UNIVERSITÉ LIBRE DE BRUXELLES



ÉCOLE
POLYTECHNIQUE
DE BRUXELLES



COMMUNICATION NETWORKS : PROTOCOLS AND ARCHITECTURES

ELEC-H417

Project Client-Server based chat app

Students:

CASSIERS Antoine, @Taenariss

FRANCHOMME Florence, @AbflolutlyFlobulous

THONAR Maxime, @Fallen

WANTIEZ Adeline, @AlphaDeltaEcho

(Group 5)

Professor:

DRICOT Jean-Michel, @Luke_SkyRouter

Teaching Assitants:

DAUBRY Wilson, @Obi-LAN_Kenobi

VERSTRAETEN Denis, @LAN_Solo

Academic Year 2021-2022

1 Introduction

The aim of this project is to create a chat app enabling private communication. The app is based on a centralized architecture meaning that the server handles all the message exchanges. This app meets the following requirements:

- Server:
 - allows new users to create an account
 - saves usernames and the corresponding hashed password on a database
 - when a user tries to login, the server authenticates the user using a challenge-response authentication scheme, sending a nonce before verifying the password
 - as long as a conversation is ongoing, the (encrypted) messages are stored in a database
 - forwards (encrypted) messages between clients, the server never sees those messages in plain text
- Client:
 - creates a new symmetric key for each new conversation using Diffie-Hellman exchange, this enables end-to-end encryption of the messages
 - can start a conversation with another existing user by addressing them through their username

2 Architecture

The project is based on a client-server model, meaning that a client never communicates directly with another client. Every sent packet transits via the server. This section is divided in 4 parts: the server, the connection of a new client, the conversation between clients and finally security.

2.1 The Server

The server is an always-on host, hence its IP address is permanent¹ and well-known to all clients. Furthermore, the server has multiple sockets: a main socket (or “welcoming” socket) with a specific port number to process all newly-connected clients, and then a socket per connected client. The idea is the following: the server listens to that main socket and waits for a client to make a connection request. On the client side, as the client knows both the hostname and the port number of the server, the client can request a connection. If the request is accepted, the server allocates a special socket to this user (with its remote endpoint set to the port and the address of this client). By having those multiple sockets, the server can manage both new connection requests and transmit messages between clients.

2.2 Connection of New Client

When the application is launched, the user can either sign up or log in.

- In the former case, the user chooses a username which is then sent to the server to check whether it is available or not. If it is, the user can choose a password, which will then be encrypted using a key obtained via the Diffie-Hellmann Key Exchange process. This will be explained in section 2.4. The obtained encrypted password is sent to the server, decrypted by the server and finally hashed. The login and the hashed password are eventually stored in a text file. The client command window in the case of a sign-up is shown in figure 1.

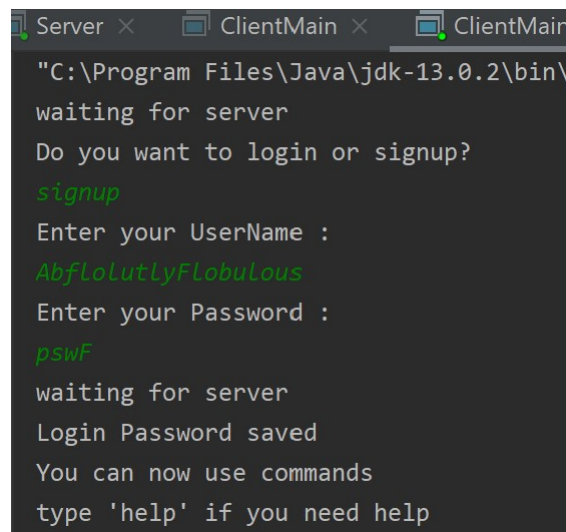
A screenshot of a Java IDE with three tabs: 'Server', 'ClientMain', and 'ClientMain'. The active 'ClientMain' window shows a text-based interface for a client application. The text displayed is: "C:\Program Files\Java\jdk-13.0.2\bin\ waiting for server Do you want to login or signup? signup Enter your UserName : AbfLoLutlyFlobulous Enter your Password : pswF waiting for server Login Password saved You can now use commands type 'help' if you need help". The user input is shown in green text.

Figure 1: Sign-up

- In the latter case, the user enters his username, which is then encrypted using the key obtained via a Diffie-Hellmann Key Exchange, and sent to the server. If this username exists in the database, the server responds with a nonce that the client then encrypts and sends back to the server. This encrypted nonce is decrypted and compared to the nonce that was sent in the first place. If the two match, the user is then asked to enter his/her password, which is encrypted and sent to the server. The server decrypts it then hashes it and checks whether it matches with the one from the database. If it does, the client is finally connected. All these steps are illustrated in the scheme on figure 2.

¹However, if the server is turned on then off again, the IP address can change.

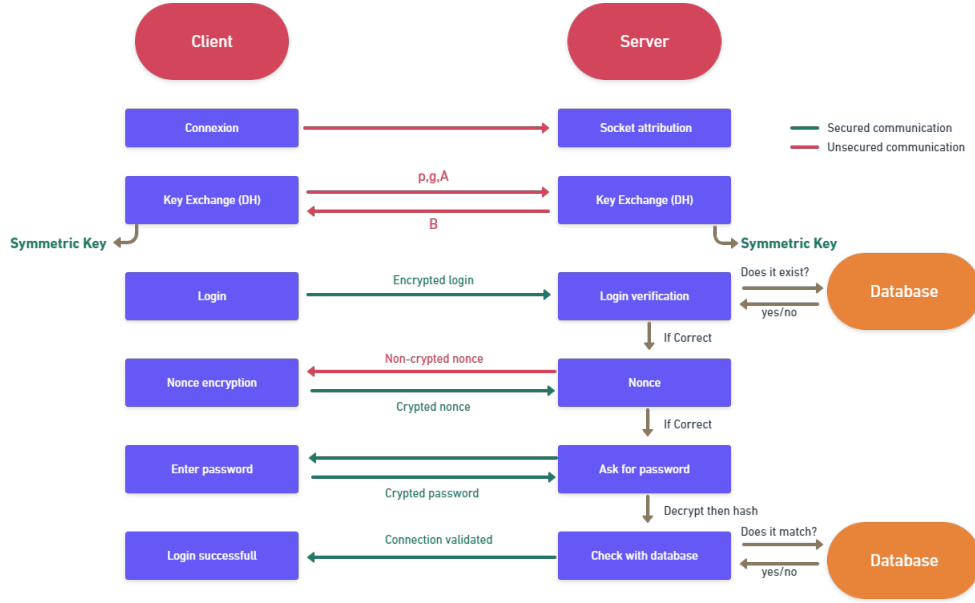


Figure 2: Client-Server communication during login

2.3 Conversation Between Clients

Once a user is successfully logged in, his socket is stored along with his username. This user can then specify, using the command `to`, to whom he wants to send a message. Once this recipient has been given, the socket of this recipient is stored next to the username of the sender. The server thus stores for each connected client the information shown in table 1.

sender's username	sender's socket	recipient's socket
-------------------	-----------------	--------------------

Table 1: Stored information of each connected client

By doing so, whenever a client sends a message (using the command `send`), the server knows to which socket it must forward this message. The client can of course change recipient whenever this suits him: simply by invoking the `to` command, the recipient's socket attached to his username will be changed. Note that this way of working implies two things: first, users can only chat to connected users, and second, as the server only stores the messages for ongoing conversations, once the user changes his recipient, his other conversation will be lost, similarly as it would be on a Snapchat conversation or when using the vanish mode on Messenger. Figure 3 illustrates a communication between two clients, with the server acting as intermediary. The results can be seen on the figure 4 with a quick chat between @Taenariss and @Fallen .

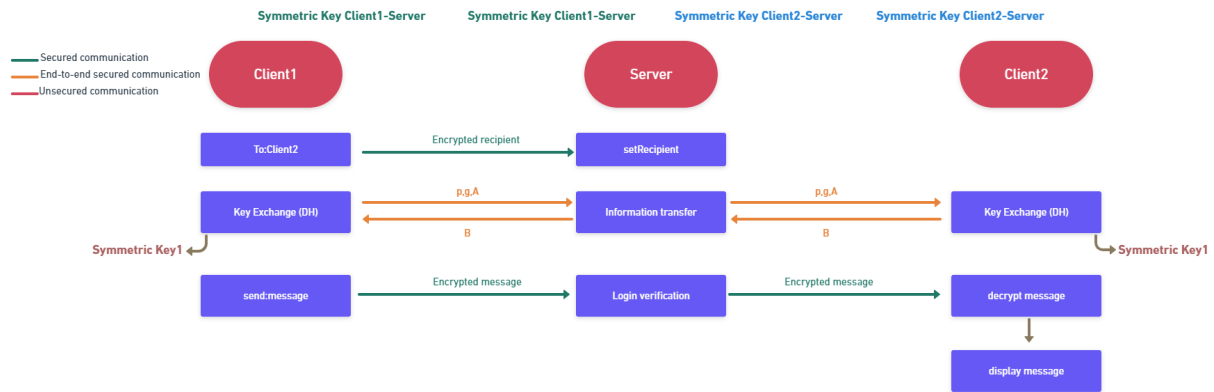


Figure 3: Client-Client communication

```

Server x ClientMain x ClientMain x ClientMain x ClientMain
Enter your Password :
pswT
waiting for server
login successful
You can now use commands
type 'help' if you need help
to:Fallen
waiting for server
the server's ready, send your message using the command 'send:'
send:Do you want to play LOL of Legends?
from Fallen : Sure, but my team sucks...Do you want to team up?
to:Fallen
waiting for server
the server's ready, send your message using the command 'send:'
send:let's goooo

Server x ClientMain x ClientMain x ClientMain x ClientMain
Fallen
waiting for server
waiting for server
Enter your Password :
pswF
waiting for server
login successful
You can now use commands
type 'help' if you need help
from Taenariss : Do you want to play LOL of Legends?
to:Taenariss
waiting for server
the server's ready, send your message using the command 'send:'
send:Sure, but my team sucks...Do you want to team up?
from Taenariss : Let's gooooo

```

Figure 4: The beginning of a beautiful friendship between @Taenariss and @Fallen

On the client side, as there is a thread to listen to the server and a thread to send messages, a user can send and receive messages at the same time. This is especially useful when you're all excited because France just got eliminated from the Euro and both @Obi-LAN_Kenobi and @LAN_Solo send messages at the same time to explain how beautiful life is.

The server serves (excuse the pun...) as intermediary between each pair of clients as the clients never directly communicate to each other. This means the data sent from a client to the server can be of various types: it can be something the server has to forward to another client or it can be another action such as changing the recipient. Therefore, each set of data sent to the server contains an "action header". This header informs the server which action it must take.

2.4 Security

Different mechanisms are in place to ensure a secure communication between the clients. In our case, confidentiality and authentication have been implemented.

To ensure authentication, the challenge-response method is used. To avoid replays, a nonce is also used. The idea is that if @Fallen wants to log in, he will first send a crypted message containing his username to the server². The server will decrypt the message and check if the username exists in the database. If so, the server will generate a nonce in the form of a random number between 0 and 1,000,000, which will then be sent to the client. The client will encrypt it with the key and send it back. Then, the server will decrypt the nonce with the same private key and test whether it matches the non-encrypted one. If the two numbers are the same, the server will ask the user to enter his/her password, which will be encrypted and sent to the server. Finally, the server decrypts the message, hashes the password and

²using the symmetric Client-Server key

checks if it matches that from the database. If it does, the client is be successfully logged in.

Regarding confidentiality, it must be ensured both between two clients and between a client and the server. The former for privacy reasons and the latter for security reasons³. In both cases, it is guaranteed using symmetric encryption. To agree on a symmetric key, the Diffie Hellman key exchange is used. The idea of this key exchange is the following:

As a starting point, @Taenariss and @Abf10lutlyFlobulous agree^a on 2 numbers, a large prime p and a base g . Then,

- @Taenariss chooses secret number $a < p - 1$, computes $A = g^a \bmod p$ and sends A to @Abf10lutlyFlobulous .
- @Abf10lutlyFlobulous chooses secret number $b < p - 1$, computes $B = g^b \bmod p$ and sends it to @Taenariss .

Finally,

- @Taenariss computes $K_A = B^a \bmod p = (g^b \bmod p)^a \bmod p = K_{AB}$
- @Abf10lutlyFlobulous computes $K_B = A^b \bmod p = (g^a \bmod p)^b \bmod p = K_{AB}$

^ain practice, all these choices are done by the client (i.e. the application and not the actual user) and the server in case of client-to-server encryption.

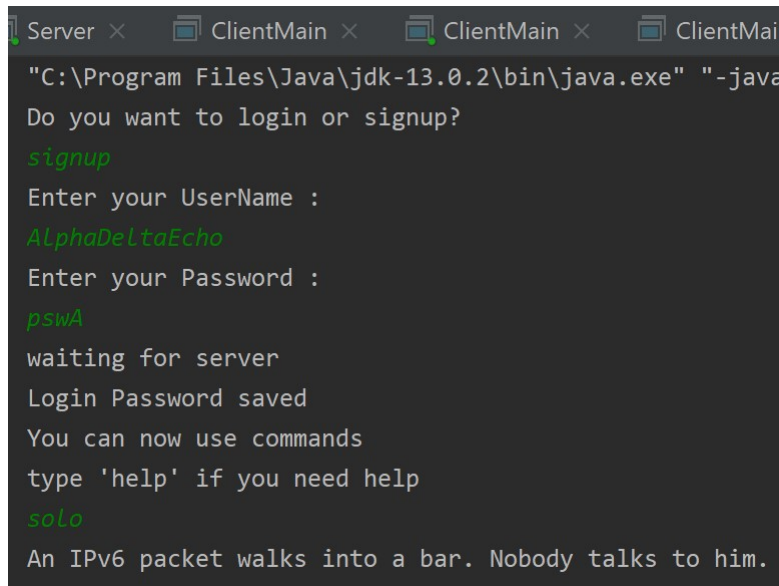
This key-exchange is performed between a client and the server when a new client instance is run (to exchange information such as password, login, ...), but also between each pair of client (and such client-to-client key is unknown to the server) when a conversation is started.

Another privacy aspect that hasn't been mentioned is the storage of passwords. To avoid storing passwords in plain text, the server hashes them beforehand, using the "SHA-256" hash function, then stores them in the database. When a user tries to log in, the (encrypted) password is decrypted by the server, then hashed, then compared to the hashed stored password.

³when the client logs in or signs up

3 Innovation and creativity

In case the user feels a bit lonely or has no friend on that server, he/she can use the solo mode, and exchange (bad networking) jokes with the server. In practice, the server has a database of jokes and whenever the client sends a message with the header corresponding to the “solo mode”, the server randomly chooses a joke and sends it back to the client. An example is given in figure 5.



```
Server x ClientMain x ClientMain x ClientMai
"C:\Program Files\Java\jdk-13.0.2\bin\java.exe" "-java
Do you want to login or signup?
signup
Enter your UserName :
AlphaDeltaEcho
Enter your Password :
pswA
waiting for server
Login Password saved
You can now use commands
type 'help' if you need help
solo
An IPv6 packet walks into a bar. Nobody talks to him.
```

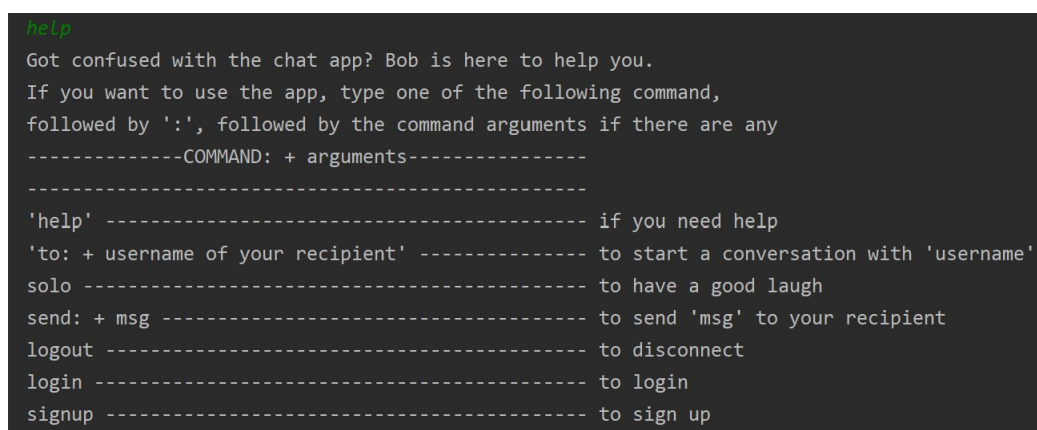
Figure 5: As @AlphaDeltaEcho has no friend, the solo mode is actually the only mode she uses

To give you just a little taste of this mode, here’s another reply from the funniest server out there:

```
1 I heard a great one about IPSEC, but you wouldnt get it - its an inside joke.
```

Hopefully, by now you want to test this mode yourself. And please do not hesitate to do so or the server itself will have to use this mode out of boredom.

A command `help` has also been implemented in order to help the user. Bob will explain all the possible commands to have the best experience with this beautiful app (after all, we did hire the best UX/UI⁴ designer for the occasion, as you can clearly see on figure 6).



```
help
Got confused with the chat app? Bob is here to help you.
If you want to use the app, type one of the following command,
followed by ':', followed by the command arguments if there are any
-----COMMAND: + arguments-----
-----
'help' ----- if you need help
'to: + username of your recipient' ----- to start a conversation with 'username'
solo ----- to have a good laugh
send: + msg ----- to send 'msg' to your recipient
logout ----- to disconnect
login ----- to login
signup ----- to sign up
```

Figure 6: Command `help`: Bob is here to help you

⁴User eXperience, User Interface

4 Challenges

First of all, in case you're getting bored with this report, we've quickly activated the solo mode for you:

```
1 The problem with UDP jokes? I don t get half of them.
```

As with all projects, it often feels like more time is spent debugging than actually implementing what needs to be implemented. Here are some of our best challenges.

One of the first issues we faced was the impossibility for a client to do both the action of sending and receiving messages at the same time. To deal with this difficulty, we implemented a special thread whose only purpose was to continually listen to what was sent on the client socket. This solution brought other challenges. Indeed, the sending and the reception were then separated, making the update of some attributes trickier. A function, `waiting`, was therefore added to observe the state of some attributes, notifying the client whenever there has been an update.

The communication client-client via the server was hard to implement, especially the order of the steps to connect clients and transmit messages. Indeed, as both the server and the client run in parallel, it was hard to make sure that the timeline was followed on both sides. The solution we found was to add a header to every packet sent between a client and the server. By doing so, the server and the client always know what action must be done with every piece of information received and when they have to wait for a response. The code is not following a timeline anymore but rather takes dynamic decision according to the type of packet received, which is indicated by the header.

Although we tried to use Java Libraries as much as possible for cryptography, we couldn't find anything practical for some functionalities, and it was much simpler to implement it ourselves. Typically, we did that for the Diffie-Hellman key exchange as well as for the nonce generation.

The application has been developed over LAN, as a starting point. However, once we tried to make it work over the Internet, it simply wouldn't work anymore. According to most forums (shoutout to stackoverflow.com without whom our LAN version wouldn't work either), chances are it is due to firewalls or port forwarding. As allowing the server port through the computer's firewall wouldn't work, alternatives have been looked up. A trick proposed⁵ is to tunnel the messages over standard web ports (80 for `http`, 443 for `https`), as it will then go through most firewalls. This can for instance be done using WebRTC⁶. This turns the chat app into a web client and thus avoids the firewall issues encountered. Unfortunately, we didn't succeed in the implementation of such protocol. We did however think of a way to share the IP address of the server (that is in fact implemented but isn't of any use as the app only works over the LAN): the idea is that when the server is turned on, it will display its address IP, and then when the client is launched, this server IP is given as an argument. Regarding the socket of the server, the port number is fixed and thus the client application knows it.

⁵Quora Forum

⁶Real-time communication for the web

5 Conclusion

The aim of this project was to design a chat app enabling private communication. The result meets the requirements presented in the introduction. Indeed, a new user is able to create an account or to login if he already owns one. The server saves the username and the corresponding password on a database using a hash function. When a user tries to login, the server performs an authentication using a challenge-response authentication scheme then verifies the password. As long as a conversation is ongoing, the encrypted messages are stored in a text file. The server cannot read those messages in clear. From the client point of view, for each new conversation, a new symmetric key is created using Diffie-Hellman exchange. A client can, whenever this suits him, starts to talk to another connected user. As a bonus, some extra functionalities were added to the app. Indeed, if a client feels a bit lonely, by using the command `solo`, the client will receive a networking joke from the server. A command `help` has also been implemented to display all the possible commands to correctly use this magnificent app.

Hopefully this report was not too boring to read, and just in case here's another server's reply to `solo`:

```
1 The punchline often arrives before the set-up. Do you know the problem with UDP jokes?
```