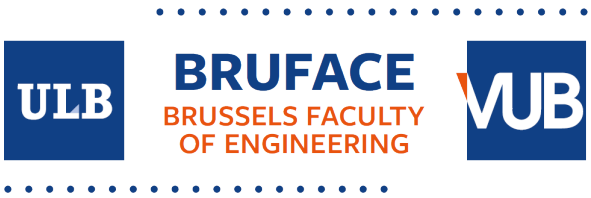

Université Libre de Bruxelles - Vrije Universiteit Brussel

ELEC-H-417 - Communication Networks

Project

Server-based Chat App

Authors	*Section*
Robin Byl	MA1 - IRELE
Nédid Ismaili	MA1 - IRELE
Yang Liu	MA1 - IRIFS
Lucas Stefanidis	MA1 - IRELE

Professor : Jean-Michel Dricot

Date : 22 December 2021

Academic Year 2021 - 2022

Contents

Introduction

This project consists on the creation of a chat application. This application should rely on a centralized architecture, with clients connecting to a server. The server needs to be able to handle client registration and login, get and serve messages, store the conversation between two users and facilitate the key generation for encryption by forwarding protocol messages between two parties. On the client side, users should be able to register or login to the app and start a private conversation with another user after agreeing on a symmetric key.

The language that was used to program the app is Python, as it is a very well known language and allows a fairly easy implementation of a GUI, communication sockets, cryptography etc.

This report explains the architecture and functionalities of the application, and details the challenges that came along the development of said application.

Architecture

This section is dedicated to present the main features that were implemented in the application. Each aspect presented is supported with explanations on the code structure. Everything is happening on an IPv4 protocol, with a connection oriented TCP stream.

Connection

The first thing to do is to boot the server. The interface of the server is shown on Figure 1, the left hand side display the information of every thread passing through the server and the right hand side is an admin command window, where commands can be entered to interact with the database of the server.
Upon the launch of the server, a new pair of private/public key is created and stored in the server database (happens every time when the server is launched). The server will then start to listen for new connections, and upon each new connection will send his public key to the client so the user can either login or register. The client will then send back an encrypted symmetric key to encrypt the next messages in the session with.

When a client ask to connect to the server, they are put in a “Connection Request” dictionary until they enter a valid password. If the client didn’t have already an account, then they must register. The user has to enter a username and a password that will both be sent to the server, the server will then check its database to see if the username chosen

Figure 1: Server graphical interface

is already taken by someone else, if so then the server send a message to tell the user to choose another name. Once the user has an account and entered the right password, they are moved onto the "Connected Client" dictionary, allowing the app to keep in memory every connected client.

On the client side of the app, the user is welcomed with a login interface through which he can connect or register to the app shown on Figure 2. The user has to enter a username and a password that will be sent to the server for verification, as said previously if the user does not already have an account they must register to enter the app.

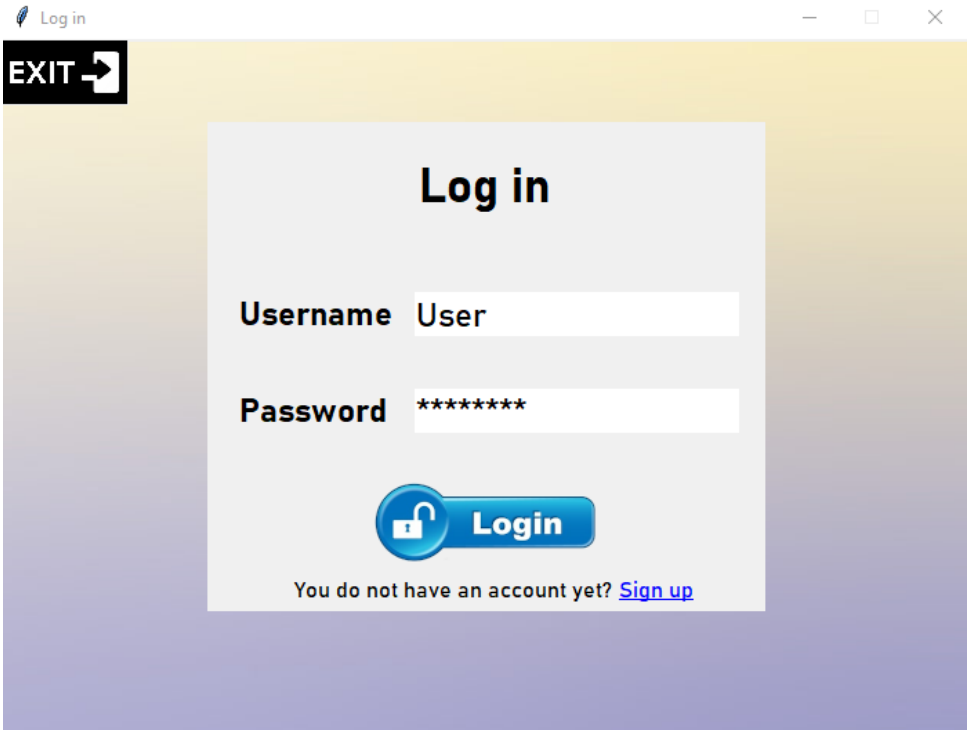

Figure 2: Client login interface

Communication

Once the user is logged in, they arrive on the chat window, shown in Figure 3.

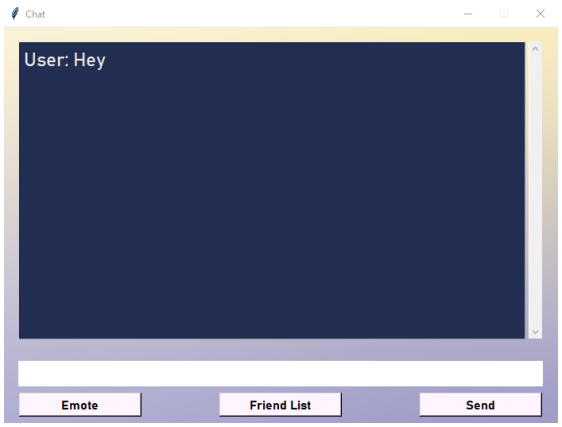

Figure 3: Chat window

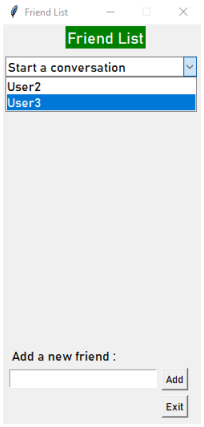

Figure 4: Friend list window

When clicked on, the button "Friend List" pops up a new window (Figure 4) allowing users to add and select friends to start a conversation. The ability to find and add friends is implemented through a database, more explications will be given in the next sections. Users can add friends if two clients enter each other usernames inside the "add a new friend" text box, then both users will be added to each other friend's list. If a user wants to talk to another, all they need to do is select their friend's name in their friend list. The client will then wait, on the server side they will be put in a queue until the other client also joins the queue, then a synchronization between the two will take place.

Once clients are synchronized, the first thing happening is the exchange of keys. The username having the name ranking highest in alphabetical order will be the one deciding on the key. He will fetch the public key of the other user, encrypt the symmetric key and send it to the client. Once both sides have the symmetric key, they will encrypt all the next messages with this key.

Security

Whenever a user registers for the first time, they create a new pair of asymmetric keys, their private key is stored in their own database and their public key is send and stocked into the server database. The same thing apply for the server, each time it boots up, a new pair of keys is created. This pair of keys is generated using the cryptography python package, which uses an RSA algorithm.

Anytime a user want to contact another, they will first fetch the public key of the desired user in order to encrypt a symmetric key. This symmetric key is generated again using the

cryptography package, but now with the Fernet library. The Fernet library guarantees that a message encrypted cannot be manipulated or read without a key, as Fernet is built on top of cryptographic standard such as AES (with a 128 bit key encryption), which is incredibly safe. The use of this type of encryption method (symmetric key) is justified by the fact that not only is it very secure, but also much less computing intensive than an asymmetric method like RSA. This will happen every time a new conversation is started, even if two users already talked before.

Now once two client have agreed on a symmetric key, it will exclusively be used to encrypt the messages. Confidentiality is then assured, but there is also a need to ensure message integrity and an end-point authentication.

Both message integrity and end-point authentication are done in the same step, by using the `sign()` method. The `sign()` method take in argument the message, and create a hash based on this message using the SHA-1 algorithm before encrypting everything with the private key of the user. That way, some computing resources are saved, as SHA-1 output a 160 bit hash and the asymmetric encryption is done on a much smaller sequence, all the while message integrity and authentication is guaranteed. Indeed, a couple `(m,H(m))` is sent to the server where `m` is the encrypted message and `H(m)` is the signed hash. The use of SHA-1 here is due to the size of the project, SHA-1 is not considered secure these days and should be replaced by SHA-256 or SHA-512 on a bigger project.

Database

It is necessary to build a database for such a real-time communication software. To do so, SQLite ,which is an embedded database that can be use in Python, was used.

Two databases are built altogether: one for the server, one for the client. Different tables are created in these databases so that different data about the users can be stored inside (e.g., `user_id`, `account`, `password`, `private_key`, `public_key`, etc.)

Function `DB_Interaction` is created to interact with the database, such as querying whether a user name is already taken or not.

The table **accounts** is unique in the client-side database, each time when a new user is registered, a private key and a public key will be assigned to the corresponding user, and these data will be stored in the account table.

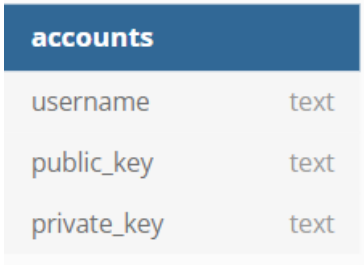

Figure 5: accounts

Figure 6: users

Then the table **users** is created both in the client-side database and the server-side database. It aims to check whether the user is already created.

Furthermore, two tables **friend_list** and **friend_requests** are also created both in the client-side database and the server-side database. Indeed, when the current user tries add a friend, they will first put the friend in the friend_requests and only when the friend does the same operation to the current user, they can become friends and put each others in the friend_list.

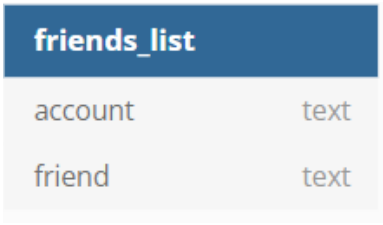

Figure 7: friends_list

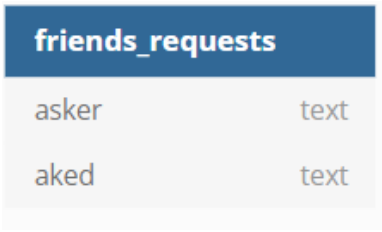

Figure 8: friends_requests

Moreover, a table **history** is built in order to store the messages between the users. The users selects the person who he wants to chat with, and the user pulls up the history of conversations with that person from the database.

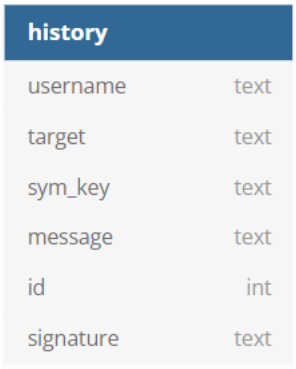

Figure 9: history

Innovation and creativity

With such project, it is hard to find outstanding innovations without complicating too much the workload on the code. Therefore, with the limited time we had, we decided to mainly enhance the user experience with improvements on the user interface. Both server and user interfaces are designed to be convenient for any new user.

As mentioned previously, a friends list has been added to the app. It appears on the right side of the chat as shown on Figure 10. Thanks to this list, users can save other profiles and the conversation they had with them, as well as add new friend to their list via their username.

Regarding the chat itself, an addition to the chat is the access to an emote panel which let the user choose among a selection of emotes that can be added directly to the messages. This panel is located on the side of the chat window, as shown on Figure 10, and allow users to have more expressive conversations.

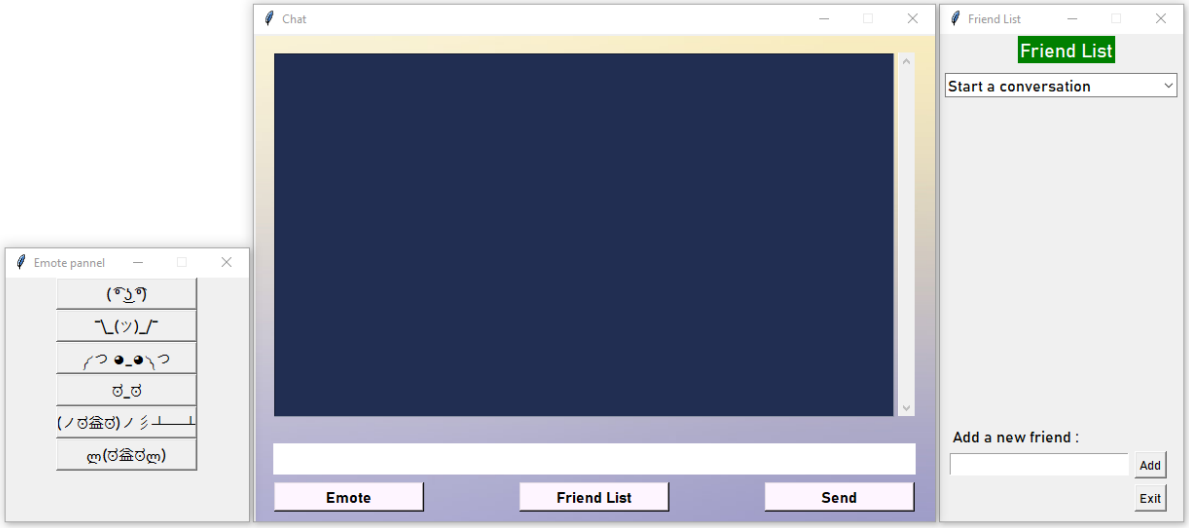

Figure 10: Chat configuration with emote panel (left) and friend list (right)

We also did some improvements the server side. For instance, we added a two-windowed terminal which allows to run, debug or access to the server internal data really easily. Indeed, the implementation of buttons and the addition of custom commands has helped us a lot in the coding process. It also allows any user to run the server in one click.

Moreover, concerning security, signatures were used in order to ensure that the user who is sending a message is actually the one they pretend to be.

Challenges

The very first challenge that came was to choose the language that is appropriate for this project. The online references are mostly JavaScript but we chose Python because we were exposed to Python in college, and we thought Python was easier to get started with.

Unfortunately the library available in python to send data does not allow for internet communication, and seems very difficult without dedicated servers, or at least some kind of paying subscription. When server and client run on the same private network everything is fine but, it seems impossible to connect from one network to another with that library from what we found in different source material.

Secondly, we have encountered many difficulties in the use of various libraries. We were not used to use those, we had to find all kinds of information to make up for the lack of knowledge in this respect. That issue morphed into a lack of time to implement functionalities and debug problems in time span given for the project.

Finally, even though we were using Github to update the code throughout the whole process of coding, it was still complicated to match the different chunks of code when we were working on different parts that were linked. In fact, we frequently had to adjust the code with new variables of functions created because of the link that connects every scripts together.

Conclusion

This project allowed to really see what happens on the application layer side of things. Although Python sockets does not allow internet communication, the application allows for two users to have a private and secure conversation, after their profile have been saved on a server, and have this conversation saved. Users can also have a friends list to retrieve other users more easily.

A few things can still be upgraded, notably on the security side of things. For instance the passwords on the server side could be hashed before being stored in the server database in order to assure the full encryption of data, a fairly easy implementation of this would be done by including the hashlib Python library, hashing the passwords before saving them in the database and each time someone want to login, their input is hashed and compared to the hashed password in the server database.

Another thing that can be upgraded is the loading of the message historic, when a client

fetch the data on the server database, when deciphering the key there might be a type conversion problem, resulting in a loss of data and raising an error provoking a crash.