Université Libre de Bruxelles
2021-2022
IRCI-EIT & IRCI-INFO
ELEC H417 Communication
Networks and Protocols

ECOLE
POLYTECHNIQUE
DE BRUXELLES

BPost Messaging App

Alexandre Flachs, *Elisabeth* Humblet,
Alexis Misselyn, *Jeanne* Szpirer

22 December 2021

Professor : Jean-Michel Dricot

Contents

1 Introduction

As part of our Communication Networks : Protocols and Architectures course, we were assigned the project of implementing a basic chat app.

The goal of our BPost Messaging App is to allow different users to communicate privately. Communications are made between an arbitrary number of clients and a server. They are encrypted by the method we judged most appropriate.

First, this report will give a detailed explanation of the architecture of our Python code, as well as the innovations we made in our project, and finally a list of the technical challenges we had to manage.

2 Architecture

The BPost Messaging App is based on a client-server architecture. We will use the term "client" to talk about a process connecting to the server and the term "user" to talk about a user who has logged in.

2.1 Communication protocol

The first thing we decided was to use a protocol using TCP on the transport layer. The choice of TCP over UDP was easy, considering TCP is more reliable and guarantees the packets will be transferred from source to destination, even though UDP is a bit quicker we can not afford to loose any information in our text messages.

We then had to choose an application layer protocol using TCP which ended up being Websockets. It seemed more appropriate than other possibilities (HTTP for example) because it initiates a connection via a handshake but keeps a bidirectionnal communication channel (full-duplex) which allows the server to send a message to a client without request and vice versa. The exchange of data in a WebSocket is fast thanks to the full-duplex connection.

2.1.1 Messages

Format It was necessary to define a precise format for the messages to be sent and received so that the server and clients could communicate with each other. Each message starts with a number and then contains the information needed to execute the requested request. The numbers mean :

- 0 : Send message (client to server)
- 1 : Log in
- 2 : Create an account
- 3 : Change password
- 4 : Add a contact
- 5 : Send message (server to client)
- 6 : Public key exchange

The messages will therefore always have the following form:

- Client to server
  0<SEP>Username1<SEP>Message<SEP>Username2
  1<SEP>Username<SEP>Password
  2<SEP>Username<SEP>Password
  3<SEP>Username<SEP>CurrPassword<SEP>NewPassword
  4<SEP>Username<SEP>Contact.
  6<SEP>Username<SEP>Public-key

- Server to client
  0<SEP>WillBeSent<SEP>Username2 OR 0<SEP>Error<SEP>Username2
  1<SEP>OK OR 1<SEP>Error
  2<SEP>OK OR 2<SEP>Error
  3<SEP>OK OR 3<SEP>Error
  4<SEP>OK<SEP>Public-key OR 4<SEP>Error
  5<SEP>Message<SEP>Username1

2.2 Server

2.2.1 Client connection

Database In order to keep track of all the users that the server knows, a SQLITE database has been created. It has two tables, one for the clients and one for the sent messages in order to keep track of the exchanges between clients. The structure is illustrated on figure 1. All columns

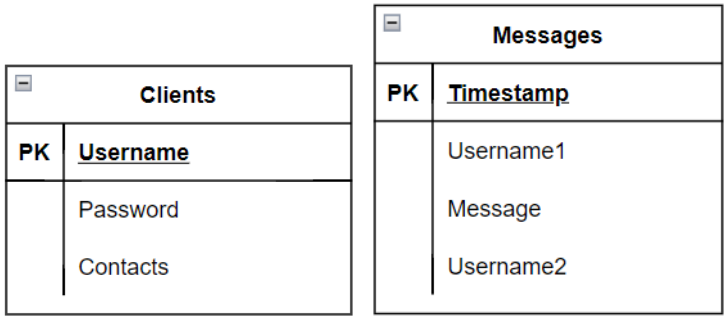

Figure 1: The two tables of the server database.

are filled with elements of type *text* which cannot be null. The timestamp column represents the time at which the server processed the message and concluded that it could be sent. The best thing for these tables would have been to link them by saying that the usernames of the messages table must exist in the clients table but as this check is done by the server in the messages management, this link has not been done. This is one of the possible improvements of this project.
Furthermore, the primary key of messages is the timestamp which means that several messages cannot be recorded if they are sent during the same second. This is not a problem at the moment as the application is not widely used, but if we wanted to improve, we would have to put the timestamp and another column in the primary key.
Finally, access and modification of the database are actions to be placed in a critical section to avoid errors related to the simultaneous modification or reading of the database. This security has been implemented thanks to a boolean variable called *database_lock* which is *False* when the database is accessible and *True* otherwise. Each time the server enters the database, it

must set the boolean to *True* and reset it to *False* when it leaves. This aspect is visible and explained in the example 2.2.2.

Sets and dictionaries Each time a new client connects to the server, its websocket is added to a list containing all the websockets of connected clients. This does not mean that the client is in the database. To do this, the server has a dictionary containing the username of the connected clients registered in the database as keys and the associated websocket as values. This makes it possible to send a message to a client whose username the server only knows, since this is what the user who wishes to communicate sends. When a client successfully logs in or creates a new account, their username and websocket are registered and associated. This information could have been in the database, but as the websockets change with each run, it's not worth making them persistent. Another dictionary is also created to store messages to be sent to users who are not logged in when someone tries to contact them. This will also be explain in the example 2.2.2.

2.2.2 Message management

When the server receives a message from a client, it must deal with it. For this purpose, a *manage_messages* function has been created. It parses the first character of the message to be processed to know what to do with it and redirects the server to the function that can parse the message in more detail. Each function then looks at the customer's request and checks that all the conditions are met to fulfil it. If some conditions are not met, the server sends an error message to the client. To illustrate this explanation, a function will be detailed in this report.

```
async def try_to_send_message(self, split_message):
    """Need to analyze the split_message to check if the second
    user is in the DB."""
    username1 = split_message[1]
    message = split_message[2]
    username2 = split_message[3]
    while self.database_lock:
        await asyncio.sleep(0.3)
    self.database_lock = True
    if self.database.client_in_database(username2) and
    self.database.client_in_database(username1):
        self.database_lock = False
        """We need to say to user1 that his/her message
        will be well sent."""
        await self.send_message("0" + self.sep + "OK" + self.sep
        + username2, username1)
        """We need to send the message to user2 and save it into
        the database."""
        print("Sending message to", username2)
        await self.send_message("5" + self.sep + message
        + self.sep + username1, username2)
        while self.database_lock:
            await asyncio.sleep(0.3)
        self.database_lock = True
        self.database.insert_new_message(username1,
        message, username2)
```

```
            self.database_lock = False
        else:
            self.database_lock = False
            """One the clients is not in the database so failure."""
            print("Error in sending")
            await self.send_message("0" + self.sep + "Error"
            + self.sep + username2, username1)
```

In this example, user1 is trying to send a message to user2. The function receives the message as a list with all the elements of the request separated. This is made possible by the systematic use of the <SEP> separator character. The first step is to store each element in the corresponding string, then we can use the strings to do the tests. To send a message, only one condition must be met: both clients must be in the database of clients that the server owns. If this is not the case, the server sends a correctly formatted error message to the client that wanted to send the message. On the contrary, if both clients are registered, then the server can send the message to user2 and send a confirmation to user1. There will therefore be two messages sent to two different clients and an addition to the message database. If one of the clients is not currently logged in (but is registered in the database), their message is added to a dictionary which lists the messages to be sent to a client when they connect. Each username is a key which has as value a list of strings which are the messages to send. When a client logs in, the list of messages to be sent for that client is scanned so that he or she is up-to-date.
This example also highlights the use of the database. Before setting the *database_lock* variable to *True*, a while loop has been coded to wait until the database is free to change the value. Once the database is free, *database_lock* is set to *False* again to allow other methods to access the database.

2.3 Client

When a client opens the app, it connects to the server by creating a websocket. Once the connection is established, the client can do a certain number of actions.

Each action requested by a client (log in, send a message, etc) requires confirmation from the server that the data the user entered corresponds to what is stored in the database. This is what is called a "Challenge-response protocol". For instance, the username of the person the user wants to send a message to must exist in the contacts list in order for the message to be created and then sent. Otherwise, the server sends an error message and the user must try again. Another example of this type of authentication is when a client logs in. They enter their username and password and the server has to confirm whether the password is valid or not for the given username.

Therefore, each action adds a certain amount of information formatted in a specific way (cf. 2.1.1) to a list of elements to send to the server. The server then treats that information, confirms or invalidates the action and sends it back to the client, as stated in 2.1.1.

```
def format_send_message(to_server):
    sender = to_server[0]
    message = to_server[1]
    receiver = to_server[2]
    return str(0)+sep+sender+sep+message+sep+receiver
```

A database also exists on the client side, which includes all the common keys that users have with each other. Each time a client adds another to their contacts list, the server sends it the latter's public key so that the common key can be generated and stored. This information should not change between runs so it should be stored persistently.

2.4 Security

Their are different layers of security we need to setup.

Password storage : The passwords entered by the clients are hashed before being transmitted to the server, that gives a low security level but it provides confidentiality.

Client-server exchanges : Consider an exchange of information between the server and a client, we don't want to let the possibility for an attacker to access the data of that exchange, nor could we be victims of man-in-the-middle attacks. To avoid these problems we used the TLS (wss:// for websockets) which is described lower.

Client to client exchanges : Now that we know that we are protected from exterior attacks we need to consider the exchange between two clients. Due to the client-server architecture every message will pass through the server, we don't want the server to be able to understand the messages between the clients, but we don't want to use P2P connections between client for multiple reasons, we thus need to encrypt the messages. The procedure is also described below.

TLS, or its predecessor SSL (Secure Sockets Layer) allows a connection between a client and a server to satisfy the authenticity of the server, confidentiality of the exchanges and the integrity of the exchanged data, using it we can secure our client-server connections.

2.4.1 Encryption method

When a user logs on a machine for the first time the program generates a secret and a private key and stores them in .txt files (those should in theory be secured and only accessible by using the user's password) then the user sends its public key to the server. When a user adds another user to its contacts list, they receive their new contact's public key. The Diffie-Hellman method is used to generate a shared-common key by using the public foreign key with the local private key. In the end, a user has one shared-key for every other user in its contact list. Those are stored in a local database (which should be secured too). The users can now encrypt or decrypt their messages using a symmetric algorithm with their common key, this is implemented using python's cryptography.fernet module. All of this method is illustrated on figure 2.

Possible improvements on the security level would be to guarantee the integrity of messages

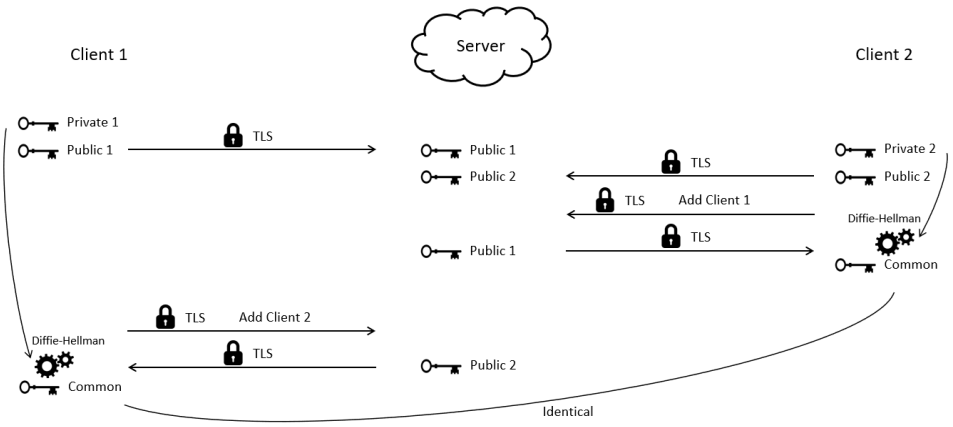

Figure 2: Use and exchange of different keys by the server and the clients.

between two clients (check that the server didn't change the original message), for example by

sending the messages with their hash, as well as the authenticity (proving that the sender wrote the message). The latter could be implemented by using the private key of the sender at send time and it's public key at the reception of the message.

3 Innovation and Creativity

Multiple extra features were implemented in the app, beyond the requirements of the project.

3.1 Password confirmation

When logging into their account, the user has to type in their password twice and resubmit the form if there was a typo. The client can thus make sure they didn't make a mistake typing their password the first time.

3.2 User friendly menus

There are two different menus that can appear when using the app. The first one is before you are logged into an account, and lets you register or log in. The second one allows you to do everything else, send a message, add contacts or change your password.

3.3 List of contacts

A user must add other users to its contacts list. This list is then printed when the user wants to send a message, to allow him to remember the usernames of its contacts. Those usernames are the only one the client can send messages to. Basically, it prevents people to chat with random strangers.

3.4 Change password

A logged in user can decide to modify their password at any time by selecting the corresponding option in the menu and putting in their current password, then their new password. this action can be done as many times as the user wants.

4 Challenges

4.1 Git

Git was just as much a helpful tool as it was a challenge during this project. It helped us merge our different parts of the code but it also slowed down some of the development. Using Github was a especially a learning opportunity for the ELEC students in the group, who never used it before.

4.2 Websockets

The python websockets package documentation is not very complete, it gave us some challenge at the beginning. Moreover websockets really benefit from the async-await coroutine implementation method, which we were not familiar with.

4.3 Databases

To create and use databases, SQL must be understood and used correctly. SQLite is not the most complicated for this since a complete package exists to use it with Python but it was still necessary to dive back into it. It is also important to manage the connections to the database, at each operation, the connection must be opened and then closed.

5 Conclusion

The goal of this project, designing and implementing a basic chat app enabling private communication, is achieved. A user can create an account, log in, and above all, send messages to other users. The architecture behind the app is based on a client-server design, which uses TCP in the form of Websockets as a way to communicate between the different parts of the design. A few databases were implemented on the server to store the information of the users and the exchanged messages, as well as on the client side to store the user's contacts list common keys. Extra features added a little creativity to the original assignment, such as password confirmation, user friendly menus, a list of contacts and the ability to change passwords.

To ensure the privacy of the users and prevent a third-party user to "listen" to a conversation or the server to read the messages that go through it, three levels of security were implemented. Also, the passwords are hashed before being stored in the server database, the connections are secured using TLS and the messages are encrypted using symmetric keys.

As stated earlier in this report, some improvements could have been made to the code and the project.

This project allowed us to work as a group and overcome technical and logistical challenges. We had a lot of freedom with this assignment and we used external tools such as git and python libraries to help us reach our goal. We learned from each other and gained experience through the process.