WhatsUP: Messaging app

Sebastian Fernandez Lopez, 0235314, Data Intelligence and Cybersecurity Engineering

***Abstract*— This report goes into detail of developing a messaging application using Java. In this project a server and a client are coded to achieve this. The main structure of the project follows a single server-clients paradigm and utilizes a non-relational database to store users, chats, digital certificates and encrypted private keys.**

Keywords*— *Messaging application - Sockets - Servers - Java - Digital Certificates - Encryption

I. Introduction

A. Single server-Client systems

Single server-clients systems are the most basic distributed systems and serve a great purpose; to connect multiple clients with a server that acts as a channel of communication between them and handles the logic of this channel. Whenever a system such as this is implemented there are two main points to consider: how will the different components communicate and what roles will each of them play. Every component must adapt to the way it receives information and be careful on how it sends it to make sure the recipient gets it the right way.

B. Objective of the project

This project seeks to replicate a basic version of the popular messaging app, Whatsapp. The implementation will be able to accept as many clients as wanted, manage the users with a phone number which will be used to send messages and validate the login; the messages will be stored in a database to make it persistent and have a contacts page to decide which chat to interact in. In addition, the application will offer an encryption layer with different options that will be explained later on. And along with this layer, a digital certificates infrastructure will be implemented plus a way to store encrypted private keys so the user does not have to store them themselves.

C. Purpose of the report

By writing this report I want to explain in as much detail as possible how I designed and implemented this idea in Java; plus the complications that came along with it. Later in this report I will state some of the problems that came up and how I solved them plus some of the logic behind the working idea of the messaging application.

II. The Project

A. Design

***Client*—** Before starting the implementation of the client I took into account certain factors. First I needed to plan how many pages i would need to what i had in mind which ended up being an initial page to capture the login details of the user, a contacts page where the logged in user would see all their active chats and an option to add a user by number to chat with them; and finally the main page where a user could chat with a specific user and receive messages in real time. Once I had in mind how many pages I needed the next task was to determine how they would communicate internally, along with their independent logic which I will explain now.

When the user first opens the application, they are welcomed by a login page where they will write their name, number and a passphrase (which will be used to decrypt the user's private key stored in the database); in case it is a valid user they will be directed to the contacts page. For this implementation the passphrase will have no role in the login validation but if it is wrong the private key will be wrongly decrypted and serve no purpose (this will be used to demonstrate how a sniffer would not be able to see the messages if they somehow hacked the application and entered as another user). If the number is registered and the name doesn't match an error message will be prompted, if the number isn't registered yet a new user will be registered in the database with the given name and a random profile picture.

Once in the contacts page, the user is able to see all the active chats just like in the messaging app whatsapp; that is with a profile picture, name and last message of every contact (in case there is one). There is a logout

button and there will also be a plus button that will show a pop up window allowing the user to write a number, if the number doesn't exist an error message is prompted and if it exists the user is directed to the the main page where they can chat with the user registered to that number. If they just clicked on an already existing chat they are directed to the same page but with their respective chat identifier.

When a user is already in the main page, if there are messages in the database with that id, they are loaded and displayed. A user can create a message by writing in the text field at the bottom and send it. To the left of the text field there will be a dropdown menu to select the type of message the user desires to send: plain text, digital envelope, signed, asymmetrically or symmetrically encrypted. The messages they send will be colored gray and the ones they receive will depend on the type of message; blue for signed messages, green for plain text, yellow for digital envelopes, brown for symmetrically encrypted ones, pink for asymmetrically and in case the message has been signed and tempered with, red . There is also a back button for the users to go to their contacts page.

After that whole logic was done, the only thing pending was to determine how it would communicate with the server. Since the client would be performing a lot of operations that require a response from the database, I opted to use JSON messages with a keyword to identify the operation. The client could send one of the following types of messages: newMsg, validateLogin, chatRequest, checkUser, checkChat, requestMessages, addUser, requestPubKey and requestPrivKey. And to not extend me more I will not specify what each type of message contains but I created structures for each type.

Server— For the server design it was considerably easier than the client because I did not need to handle any interaction logic. The way I designed my server was to first just open the port so the clients connect and plan a listener to handle all types of messages. The types of messages the client can send are the ones the server can receive and the server can send any of the following ones: loginResponse, messageIncoming, ContactsRecieved, numberCheck, chat_idAttached, ChatAttached, refreshContactos, incomingMsg, pubKey_attached and privKey_attached.

For the ones that need it, the server will also handle the Database connection and requests; it will process and structure the corresponding response and use it as needed. It will also be responsible for generating the digital certificates and private keys of each user and handle how they are stored in the database.

Database— To design the database I began by choosing the one I considered best for this scenario which was a non-relational database and I chose MongoDB. Given the simplicity of the design I didn't really need a big schema and by not using a relational database I didn't have to worry about multiple relations between tables.

For my design I have a Users collection which stores a user_id, name, phone_num and profile_pic. This collection is the one used for the login validation, checking if users exist and to name the corresponding phone numbers.

Additionally I have a Chats collection that stores a user_id, destination_id, last_message and a messages array with all the messages between user_id and destination_id. Which serves the purpose of storing all the messages and handling how the messages display in the chat.

I also have two collections to store digital certificates, AR1 and AR2. Each certificate is constituted by a unique identifier, user_id, name, public key and two dates, valid from and valid to. And the last collection I'll use in this project is to store the user's private keys and in this one the only thing stored is the user_ and their encrypted private key.

Encryption infrastructure— To handle all the encryption requirements of this project I basically separated the problem into four parts: the encryption methods and the server, client and database role. For this project I opted for simple substitution as the encryption method, the keys are integer numbers and the alphabet used has letters, numbers and punctuation marks. The server is in charge of generating the digital certificates and private keys. The digital certificates contain a unique id, public key, name, creation and expiration date. And it is always verified that there

aren't two equal public keys when creating a digital certificate. The private keys are created based on the randomly generated public key generated for each user and is encrypted with a passphrase only the user knows. Then on the client side is where the messages can be encrypted and/or signed depending on its type and also decrypt the private key retrieved from the database with the user's phrase. And then the database will store the digital certificates in the AR1 or AR2 so anyone can request a user's public key and then also a collection for the encrypted private keys so the user can get it and decrypt it.

B. Implementation

Client— For the implementation of the client I began with an FXML project to handle all the GUI. I coded the pages as described in the design section and assigned each component an id and an action to the ones that required one.

For the communication protocols I opted to use a PrintWriter (sends String objects) object as output and BufferedReader as input. The connection handler is running on a separate class and thread to avoid interference with the main GUI thread and handles the communication. The messages are sent as JSON objects through the socket to the server. I will not go into detail into every type of message sent from the client to the server to not extend this report more than it's needed but every message follows the same logic: an action and the necessary data.

Regarding the communication, each client has a listener actively registering new messages and depending on the type of message a handling method to process that information, usually it calls a method from the corresponding page controller and does whatever it is needed.

To handle all the encryption requirements I implemented a type to the Message struct that is sent to the server whenever the user wants to send a message. This type can be between 1 and 5 meaning the following: 1 (plain text), 2 (signed message), 3 (digital envelope), 4 (symmetrically encrypted), 5 (asymmetrically). It will encode the message according to the type chosen and decode when it receives one as well.

The hardest problem I came across was since I was running the listener on a different class and thread to communicate with the FXML controller I had to use static methods. Due to that the controller is initialized by the project and I cant access its instance from another class. So to solve this I used a public static method to get into the controller and then use platform.runlater to access the main thread and had already declared an instance as "this" and I accessed that and then used a simple private method to do whatever I needed. The problem came that the FXML injections worked only in the instance initialized by the project, they could not be static.

Leaving this behind, the rest was relatively easy. Other important points about the implementations are: to know which contact the user wants to chat with, when they are retrieved from the database, the controller assigns each contact element an id like "chat_ssfsdfsd" where the gibberish is the chat_id. The new messages are handled in two parts, first they are sent to the recipient and then added to the databases to avoid latency.

Server— The implementation of the server was very straight forward, I only needed a listener receiving the messages from the client with the same logic as the client listener: a switch deciding on how to handle the message.

Regarding the digital certificates and private keys, I implemented a java class in charge of handling all this, it can generate random keys and private keys given a public key. This class is used whenever a request to create a user is received by the server to generate its digital certificate and its private key to store it in the database.

Besides this, the "hard" part was to connect to the mongo database to handle all the methods that required it. The server could do either of the following operations: check if a phone number exists and name matches, add user to users collection, check if a chat exists between user A and B, get all the messages, get the last message, create a message and add messages to the message array from a specific chat.

Database— As mentioned before, the database implementation was just to create a mongo cluster with

four collections, Chats, Users, PrivateKeys, AR1, AR2. AR1 and AR2 are used to store the digital certificates. And then finally just generate a mongo client in the Java application with the URI, username and password.

JSON messages— As stated before, the method to send information between servers and clients was done by json messages. I used the gson library which transforms java objects into json objects and vice versa, which means i created a class for every type of message to have control over everything in a controlled manner.

C. Technical explanation

In this section I will go into extreme detail about how the flow of the application works for the sake of explaining the technical side of the project. First I'll jump into the client flow and then the server. I will go into detail of how the requests are processed in the server section and in the client one just what information is sent and received.

Client— For the application to work the server needs to be already up and running in port 5109. The program begins when a client is initialized. A client TCP socket is created and tries to connect to the server socket and a thread is also started to handle the communication with the server. Once the application has connected successfully the user can start interacting with it and the first main action happens. The user is first presented with the login/register page where they need to input their name, number and passphrase. A request is then sent to the server via a PrintWriter containing the user's name, number and passphrase. And a response from the server is received: -1 if the number is registered but the name does not match or the user_id if the name and number matched or if the number wasn't registered and the user was created and added to the database. Once the login validation has been successful the user_id is set as a global variable to know which user is using the application and also their safety phrase for once their private key needs to be decrypted. Finally they are moved to the contacts page.

When the contacts page is loaded a request is sent to the server with the user's id to get all their chats that are in the database. Now the user has two possible courses of action, to click on an existing chat or create a new chat with a number. Either way when they go into a chat the second main action is triggered. First the user_id, destination user_id and destination name are set as global variables for the chat controller so that it knows what conversation it is and with whom. I'll call this page the Main page since it is where the user can chat with their contacts. When the main page is loaded 3 main things happen: the application checks if the chat between those users exists or creates one if it doesn't, it also sends a request to get the destination user's public key and another to request their encrypted private key from the database. The first request is completed when the client receives from the server a list of all the messages and prints them in the page. I'll go back to how the messages are printed later on. Then the keys requests are completed in the background where the destination user's public key is set and the user's private key is decrypted with their passphrase and set. The mechanism for decrypting is easy, since the private key was originally encrypted with the passphrase, the encoded key is symmetrically decrypted with the same passphrase (the key is the hash code of the passphrase). If the passphrase is wrong the value after decrypting it will probably not be an integer number so it returns a 0 and the user won't be able to decrypt the messages sent to them.

Now that the messages have been loaded and the application has the user's decrypted private key and the destination user's public key, it can start sending messages. The main page controller has a variable called "msgType" which is initially set to 1, this is what tells the server what type of message is being sent, for example, 1 corresponds to the type plain text message. The user can select from a dropdown menu which type of message they want to send and the msgType variable changes accordingly. Then when the user hits send, the message can be processed in 5 different ways depending on the msgType. In case it is a plain text message, nothing else is needed; in case it is a signed message, its hash is generated with a custom hash function which adds the ascii value of all the characters and then this hash is encrypted with the sender's private key; case it is a digital envelope, a random symmetric key is created, the message's hash is generated, both the hash and message are encrypted with the random key and finally the random key is encrypted with the destination user's public key; in

case it is a symmetrically encryption the message is encrypted with the key given by the user and lastly if its asymmetrically encrypted the message is encrypted with the destination user's public key. Once the message has been processed it's sent to the server in a newMsg struct which contains the processed message, the signature (empty if message wasn't signed), the encrypted random key (empty if it wasn't a digital envelope), sender id, recipient id and the message type. And the message is also printed in the sender's chat.

The only big functionality left is to receive messages. Whenever a message is sent and processed in the server the receiving end can see the message instantly if they are connected and have the chat opened. When a client receives a message two things happen: the message and signature are decoded according to the message type and then printed onto the chat with the right color. If the message has been signed and the hash doesn't match the decrypted signature the message will be colored red to show it has been tampered with.

Server— Since the actual flow of how the application is supposed to work has been covered in the previous section, here I'll just go into detail on how the most important requests are processed: handling connected clients, creation of users and their keys, processing incoming messages and sending messages to specific users.

Before any request can be made, the server has to evidently be up and running in port 5109, so for this when the server is run, a server socket is created in this specific port and stays open to accept any client socket. In addition the server also connects to the Mongo database with the provided URI.

When a client instance is created and connects to the server's port, two things happen: the server socket accepts it and assigns it a UUID to identify it and then adds a key-value pair to a hashmap where all the sockets are stored using the UUID as key. This will serve the purpose of having access to specific sockets inputs and outputs plus having a way to identify the clients when they disconnect or any issues occur.

When a login validation request is made to the server the following series of events are triggered. Firstly, the server sends a request to the database to search for the number given and check if it exists. If the result is not null, meaning the user exists, it then checks that the name given mathcess the one in the database, if so the user_id is sent to the client socket, if not a -1 to tell the client that the name and number doesn't match. But in case the result was null, meaning the number isn't registered yet, a user is created and registered to the database. After this when the user_id is available the server registers in the hashmap this user_id with their corresponding UUID to be able to access the other hashmap with only the user_id.

This Registering of a user in two parts: the creation of the user and the creation of their keys. At the beginning of the process is to create a user model with the name, number, user_id and random profile picture. And now the second part is to create their digital certificate and private key. First a collection to store the certificate is randomly chosen, AR1 or AR2, then a random key is generated (random integer number from 0 to Integer.MAX_VALUE constant in java). Before going any further the ARC collection (the Central Registration Agency which stores all public keys held in any AR) is checked to see if the generated key already exists for a digital certificate. This key will be the user's public key so the next step is to create a UUID for this certificate, assign the user_id and user's name and the dates it will be valid from and to (2 years).

Then it's the turn of the private key, Since the private and public key need to complement each other adding up to the length of the alphabet being used, it is necessary to have the public key to create the private one. Since the key can be a large integer and the alphabet is less than a 100 characters long I developed a formula to calculate the private key based on the principle that a key with value n is equivalent to a key with the value n+len(abc) since it just goes around the alphabet another time using modulus. Then the way the private key is calculates is as follows with pub being the public key and len the alphabet's length: *privKey = (int)(pub/len)*len+(len-(pub%len))* . With this formula the private key is guaranteed to work as expected and also be of similar length as the public key. But it still needs to be encrypted in order to be

safely stored in the database and this is where the passphrase comes in since it is gonna be used to encrypt the private key. Taking advantage of the already existing *String.hashCode* function we get a decently unique integer for the given passphrase and that is the key used to encrypt the key. Once it is done, the encrypted private key is stored in the *privateKeys* mongo DB collection along the user's id.

The third most important part of the application is receiving and sending messages and this is how the server processes those requests beginning with incoming messages. Whenever a client sends a message with an action tag of *"newMsg"* the server does two things: redirect the message to the receiving end and store it to the database; the database process happens after sending the message to avoid any latency. So when the message is received it is deconstructed to get the user_id of the destination user plus all the parts of the message (text, signature, encoded random key, message type and timestamp).

Then a method is called to send the message to the destination user. This method works by first verifying the hashmap that stores user_id-socket pairs to see if the user is currently connected to the server, in case it is not it doesn't need to send the message instantly. If they are connected the server can access the output of the socket corresponding to that user. Then it simply sends the message with an action tag of *"incomingMsg"* so the receiving user can handle it properly. Then the second part is sending this message to the database which also updates the chat element *"last_message"* which is shown in the contacts page.

These are the most important processes the server can handle and are the core of the application, it also has other secondary handling methods to check if a number exists, if a user has a chat with another user or getting a user's keys. This one is worth jumping in, when requesting a user's public key the server first looks into the AR1 collection to see if that user is registered in there and if not it checks the AR2 collection. Once it is found in either of them it can be sent to the client.

III. Conclusion

In conclusion, the hardest part of this project was learning how the basic structure of a client-server system works and solving the threading issues. Throughout this project I learned a lot about how servers communicate with other components in a system and how they can handle different connections simultaneously. And now with the encryption and digital certificates layer, the application feels more and more like something that could be really useful in real life. Moreover, despite the complications i think java was a good choice for a project like this given the portability and non-dependency on specific hardware. Since it runs in a JVM, it can simulate threads, run on any environment with jdk and at the same time have access to the ports to create sockets.

IV. References

[1] C. Kidd, "Distributed Systems Explained," Splunk, https://www.splunk.com/en_us/blog/learn/distributed-systems.html#:~:text=A%20distributed%20system%20is%20simply,a%20single%20device%20ran%20it. (accessed Sep. 12, 2023).

[2] J. Meza "Sockets en Java" Programarya, https://www.programarya.com/Cursos-Avanzados/Java/Sockets (accesed Sep. 12, 2023)