Design Document

Secure Private Group Chat

Anthony Arseneau

Computer Networks Final Project

COMP 4911, April 3, 2024

Preface

The following document contains justifications and details of the development process for the Secure Private Group Chat computer networks final project. No external sources were consulted during the development process beyond those included in this document. The server source code is in the project1 folder given and the client source code is in the project2 folder given.

Design Timeline

Step 1

The initial step to creating the Secure Private Group Chat was to make a basic client-server group chat application. Here, there were only 3 Java classes: *ChatServer*, *ChatClientHandler*, and *ChatClient*. *ChatServer* and *ChatClientHandler* were running on the server side of the application. The role of *ChatServer* was to listen on port 12000 for any new client connections and create a *ChatClientHandler* for each of them. Anytime a client wanted to send a message to the chat, the message would be sent to the server to its corresponding *ChatClientHandler* and broadcast to the rest of the connected clients. The chat was used through the terminal. The Java socket programming learning part was done on Medium:
https://medium.com/edureka/socket-programming-in-java-f09b82facd0#:~:text=Socket%20programming%20in%20Java%20is,a%20client%20and%20a%20server.

Step 2

The second step was to create a Java Swing window (*AuthenticatorView*) that would prompt the user for the username and password when starting the application on the client side. No authentication system has been implemented yet, but the given username was used to show the user's name when sending a message. At the same time, a chat view window (*ChatView*) was also introduced using Java Swing to show the chat contents. The window had a large area to show the messages sent and received, a text field at the bottom to input messages, and a send button to the bottom right. When pressing the button, the message in the text field disappears and is sent to the server, where it broadcasts it. At this step, the *ChatClient* class was split in two, now known as *ChatClientListener* and *ChatClientSender*, for a more effortless flow of information.

Step 3

The third step was to implement an authentication system. This was achieved by creating a whitelist.txt file to keep all authorized users' hashed usernames and passwords. The hashing function used was SHA-256 to securely hide users' information in case of a breach in the server. The credentials for each user were set to be on one line each, where the usernames and passwords have a single whitespace separating the two values. Then, before the server granted access to the group chat to the new connection, the first message sent by the user to the server was its credentials. With that information, the server, using a new class named *Whitelist* implementing verifications, could reply with either "Yes" or "No" whether the credentials matched with an entry in the whitelist.txt. If the answer were "No," then the user and server would close connections, and if the answer were "Yes," then the server would grant the user access to the group chat.

Step 4

The fourth step was to implement a cryptographic exchange of information. This step was easier said than done. Two classes were created: *RSACipher* and *AESCipher,* each of which implemented the roles of their respective ciphers. The RSA cipher is excellent for sending and receiving secure messages when no symmetric vital ciphers are established between two parties. Like in this case, the clients and the server are theoretically far apart with no prior symmetric key exchange. Unfortunately, the RSA cipher encryptions and decryptions are slow and have a limited message length. That is why it was used to only send the first message of the client to the server, which is the credentials, and the response from the server to the client, which includes if granted access to the group chat, the secret asymmetric key of the AES cipher used for the rest of communications.

Compared to the RSA cipher, the AES cipher is much faster to encrypt and decrypt and can work with longer messages.

Server Design

ChatServer Class

The *ChatServer* class is the start of the server program. It is positioned in the “middle” of the group chat. All clients send messages to the server so that they can be distributed to the rest of the group chat members.
The *ChatServer* class creates a *ChatClientHandler* object to handle each new connection in separate threads. Every time *ChatServer* is run, it creates and saves a unique new AES cipher.

ChatClientHandler Class

The *ChatClientHandler* class handles all clients connected to the server. It holds a list of connected and validated clients so that it can broadcast received messages to all members at once. This class is responsible for asking the *Whitelist* class to verify the credentials of new clients before adding them to the list of approved connected users. It is also responsible for removing clients from the connected list when they leave.

Whitelist Class

The *Whitelist* class works closely with the whitelist.txt file, keeping a list of authorized users. It is responsible for adding users to the whitelist, printing the whitelist to the .txt file, and validating the given credentials.

whitelist.txt

The whitelist.txt file holds a list of approved users, with each user on one line. Each line has the hashed value of the username separated by a space and the authorized user's password.

server_public.key & server_private.key

The server_public.key and server_private.key files hold the server's public and private RSA keys, respectively. When sharing the client source code with a new user, it is important to share the server_public.key file so that they can also send encrypted credentials to the server.

secret_key.txt & iv.txt

The secret_key.txt and iv.txt files hold the group chat's secret key and initialization vector of the AES cipher. The server gives these files when a client successfully sends valid credentials. With these files, the server can encrypt and decrypt group chat messages.

Client Design

AuthenticatorView Class

The *AuthenticatorView* class serves as the client's starting point. Once the main method is run, a GUI appears, prompting the user to provide a username and a password. Once the fields are filled, two button options are available: "Clear" and "Submit." The "Clear" button will empty the fields if the user wants to restart inputting their credentials. The "Submit" button will store the credentials in variables and start the rest of the program. Figures 1 and 2 show the authentication window:

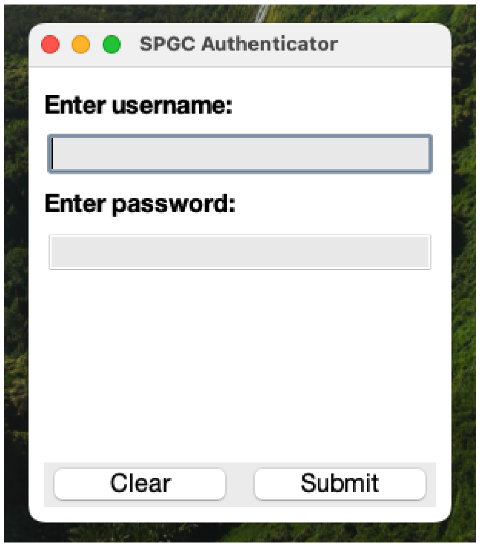


Figure 1 AuthenticatorView GUI with empty fields.

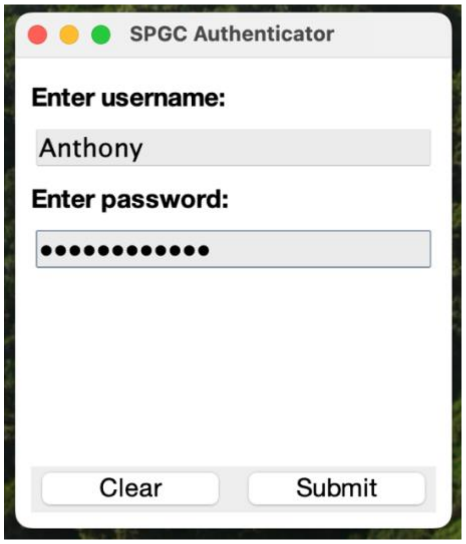


Figure 2 AuthenticatorView GUI with filled fields. Pressing "Submit" will store the values in variables and sent to the server.

InvalidView Class

The *InvalidView* class is a GUI that appears if the given credentials are not valid to connect to the group chat. Two buttons are available: “Leave” and “Retry.” The “Leave” option closes the window and stops the client program, and the “Retry” option opens a new *AuthenticatorView* to let the user try again. Figure 3 shows the invalid view:

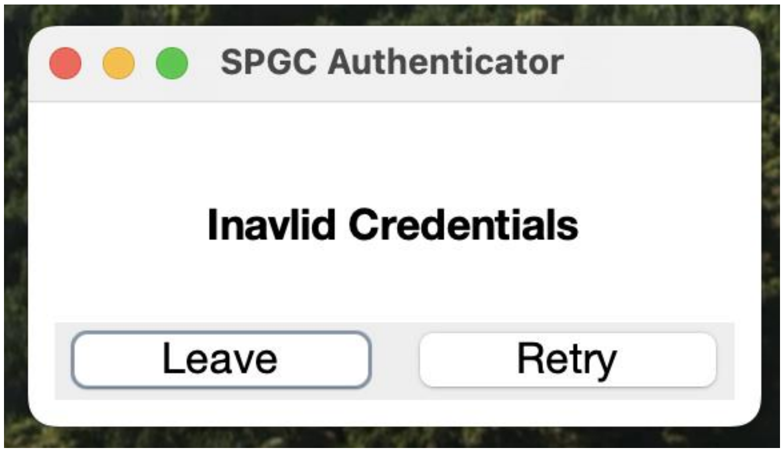


Figure 3 InvalidView GUI to inform the user of the incorrect credentials.

ChatClientListener Class

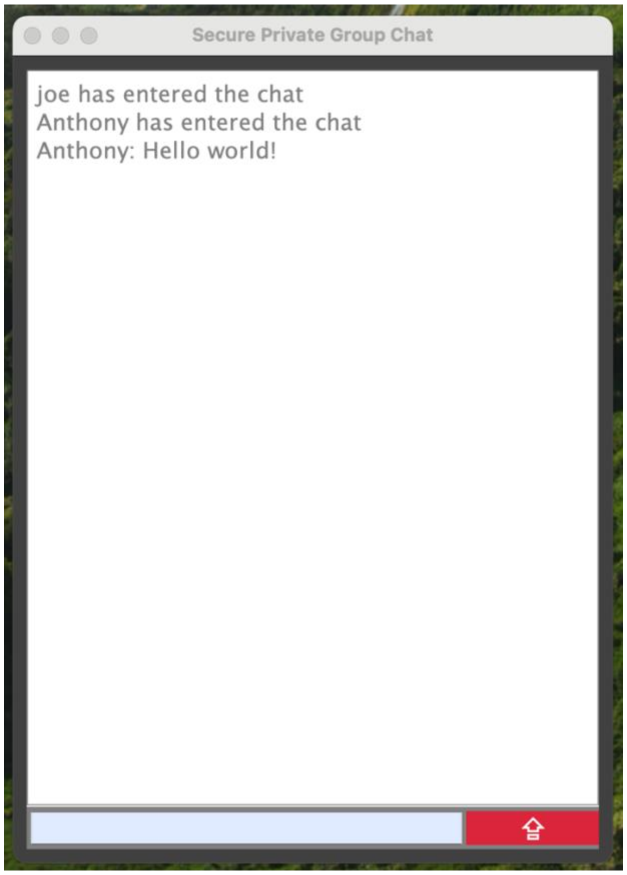
The *ChatClientListener* class initially creates a connection to the server and explicitly listens for incoming information or messages. The first message it receives from the server is the authentication validation and possibly the secret AES symmetric key (if validated). If the credentials are valid, it displays the *ChatView* and listens for any messages from the group chat via the server. Every message it receives is sent to the *ChatView* to be displayed for the user. If the credentials are invalid, then the *InvalidView* window will be shown instead.

ChatClientSender Class

The *ChatClientSender* class creates a connection to the server and explicitly sends information and messages. The first message it sends to the server is the user's credentials and public RSA key. After that, it sends the messages the user gives to the group chat via the server.

ChatView Class

The *ChatView* class serves as a user-friendly GUI that displays the conversation in the group chat. When the user presses the "Send" button, the input is taken from the text field and given to the *ChatClientSender* so that it can be sent to the group chat-connected members. Pressing “X” at the top left corner will send a last message indicating that the user has left and will close the connection with the server. Figure 4 shows the chat view:



joe has entered the chat
Anthony has entered the chat
Anthony: Hello world!

Figure 4 ChatView GUI where it displays the chat and gives the option to write and send messages. This is from the perspective of the user "joe”.

JTextFieldLimit Class

The *JTextFieldLimit* class is a helper class that limits the number of characters in a given text field or password field. This feature is only used in the *AuthenticatorView*.

MyButton Class

The *MyButton* class is a helper class that makes a given button more interactive in terms of colour change when hovering and clicking the button and sets the mouse to a clicking hand when hovering the button.

public.key & private.key

The public.key and private.key files hold the client's public and private RSA keys, respectively. When sharing the source code with other users, it is crucial to erase the private.key copy information before transmitting the file. Otherwise, the new user could decrypt messages only meant to be seen by the previous sender. From here, the user receiving the source code can easily create a new key pair by utilizing the createKeyPair method in the *RSACipher* class.

server_public.key

The server_public.key file holds the server's public RSA key. When sharing the source code with a new user, it is essential to share this file so they can also send encrypted credentials to the server.

secret_key.txt & iv.txt

The secret_key.txt and iv.txt files hold the group chat's secret key and initialization vector of the AES cipher. The server gives these files when the client successfully sends valid credentials. With these files, the client can encrypt and decrypt group chat messages. When sharing the source code with a new user, these file copies should also be erased if the user is not authorized to use the group chat.

Server & Client Shared Tools

AESCipher Class

The server uses the AESCipher class, and the clients utilize the AES symmetric key cipher. It can generate a new key along with an initialization vector and encrypt and decrypt messages that are sent by the group chat. Many other methods are implemented

in the class for valuable purposes. Helpful documentation about the cipher was consulted from Baeldung:
https://www.baeldung.com/java-aes-encryption-decryption.

RSACipher Class

The *RSACipher* class is used by the server and clients to utilize the RSA asymmetric key cipher. It can generate a pair of keys and encrypt and decrypt messages. Many other methods are implemented in the class for useful purposes. Helpful documentation about the cipher was consulted from Baeldung: https://www.baeldung.com/java-rsa.

Converter Class

The *Converter* class implements two useful static methods to convert from one data type to another. The first method converts a byte array into a readable hex string representation of the data. This comes in handy for sending keys through a buffered writer. The `bytesToHex` method was taken from Baeldung:
https://www.baeldung.com/java-byte-arrays-hex-strings.
The second method does the opposite of the first; it converts a hex string representation of the data into a byte array. This is handy for reconverting the original data from a buffered reader. The `hexStringToByteArray` method was taken from StackOverflow:
https://stackoverflow.com/questions/140131/convert-a-string-representation-of-a-hex-dump-to-a-byte-array-using-java.

Server Manual

The server source code is in the project1 folder given.

Initiate Ciphers Keys and Initialization Vector

When making a new server for a group chat, it is necessary to initialize the RSA key pair and the AES cipher secret key and initialization vector. To achieve this, the `createKeyPair` method in the *RSACipher* class creates a public and private key pair for the cipher. For the AES cipher, `generateKey` and `generateIV` methods, respectively, make a secret key and an initialization vector. Once these are created, they can be saved and written to their respective files. The server_public.key file must be shared with all users to receive the first encrypted communications.

<u>For the purpose of testing, the files are already set up and do not need to be modified in order for the program to work.</u>

Initiate Whitelist of Authorized Users

When creating a new server, a unique whitelist is desired. To achieve this, the addToWhitelist method in the *Whitelist* class prompts the server user in the terminal to add names and passwords until the user types “done.” From there, the whitelist is saved and printed in the whitelist.txt file. Only the users with those credentials can access the group chat.

<u>For the purpose of testing, there is a pre-set-up whitelist.txt file with a few credentials already in. Here are some of the authorized users’ credentials to try and connect with:</u>

(USERNAME [SPACE] PASSWORD)

- user1 password123
- user2 password321
- joe iLoveCats2
- Ellie70 yR457i

Note (i): usernames and passwords are case-sensitive.

Note (ii): The given whitelist.txt file actually has six entries, which means that 2 of them would be unknown by an attacker because they are hashed. This is just to prove that it is useful to hash credentials in a database.

Starting the Server

To start the server program in Visual Studio Code, press “run” above the main method in *ChatServer*. From there, users can connect and chat with other people online.

Client Manual

The client source code is in the project2 folder given.

Set Up Server IP Address

When starting the program for a client, the client must specify the SERVER_IP value in *ChatClientListener*. For now, it is set to the local 127.0.0.1 address.

Start the Client

To start a client program in Visual Studio Code, press “run” above the main method in *AuthenticatorView*. Then, fill the text fields with valid credentials, and the chat view will appear.

Sample Input/Output

Sample 1

The Secure Private Group Chat effectively works using two computers. One computer runs the server program and a client program, while the other only runs a client program. Instead of using 127.0.0.1 in *ChatClientListener*, the subnet IP of the computer running the server program is used instead. This illustrates that the networking part of the project works. The following 5-11 figures show this part of the project working:

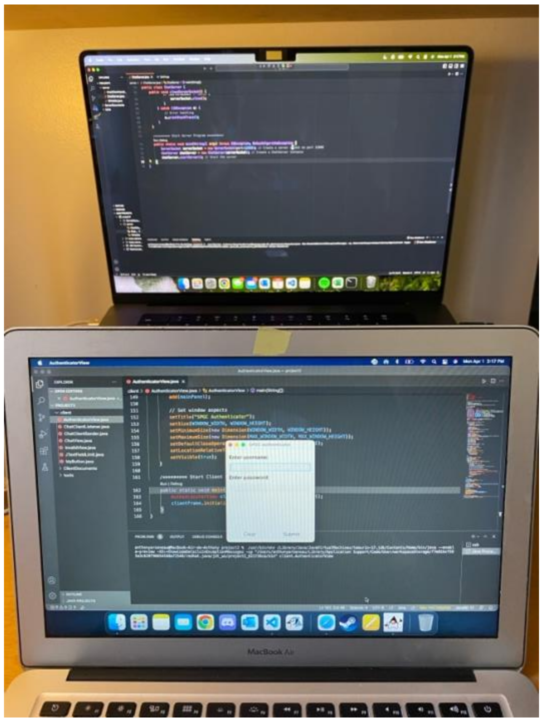


Figure 5 Top computer running the server program while the bottom computer opens the AuthenticatorView.

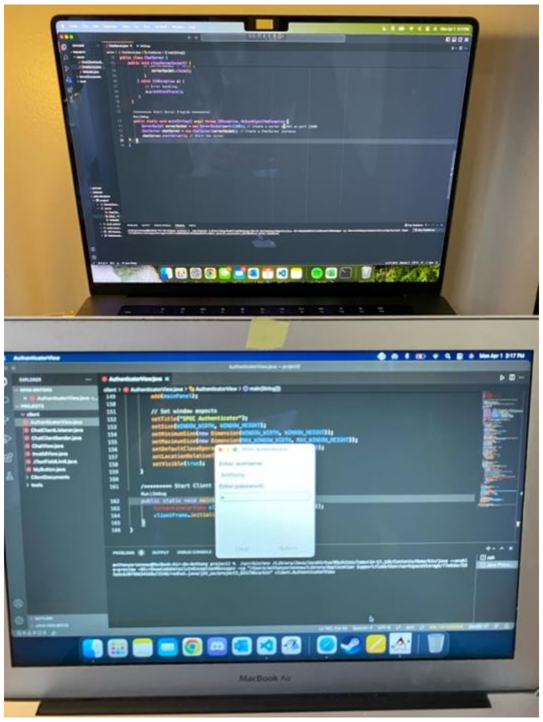


Figure 6 Bottom computer fills in the credentials with a username of "Anthony."

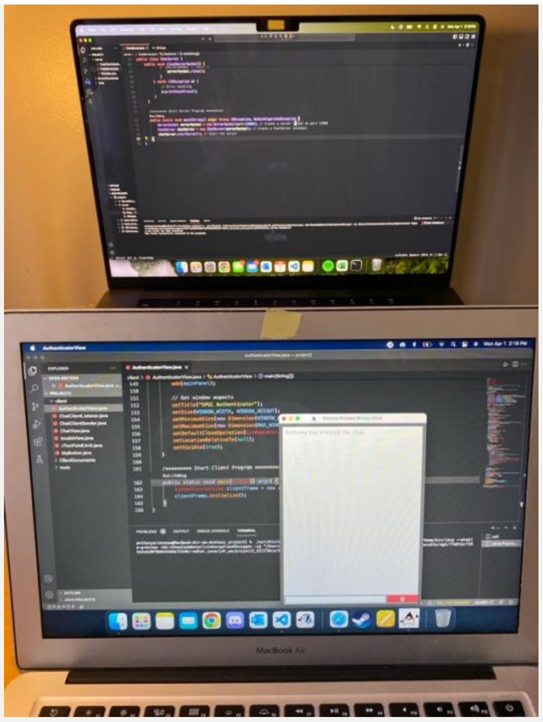


Figure 7 Bottom computer had a successful authentication.

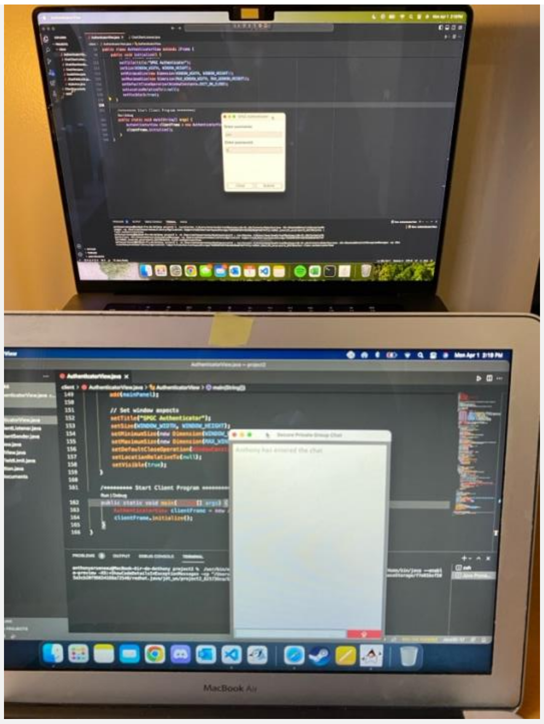


Figure 8 Top computer is now also running a client program with a username of "joe."

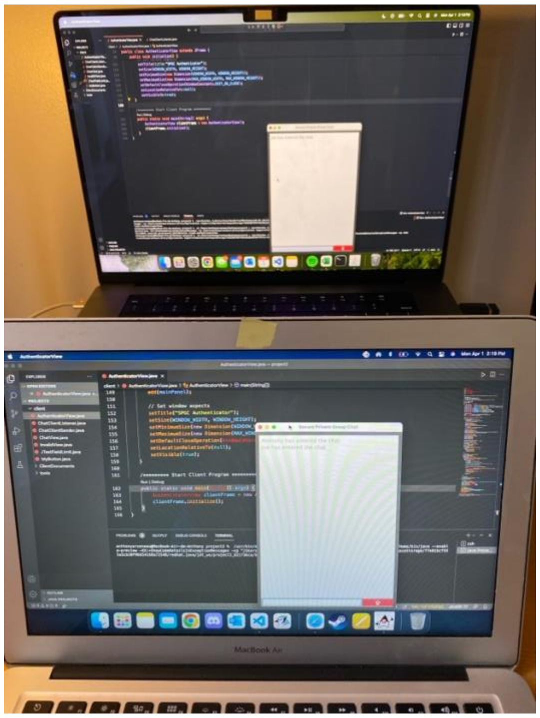
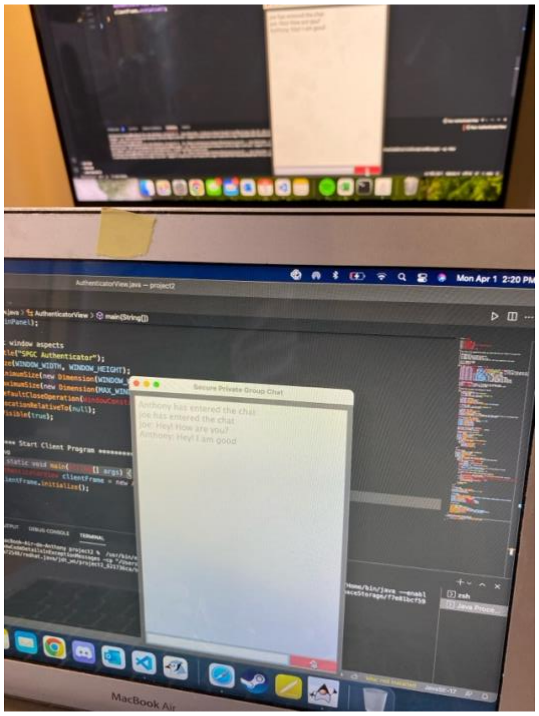
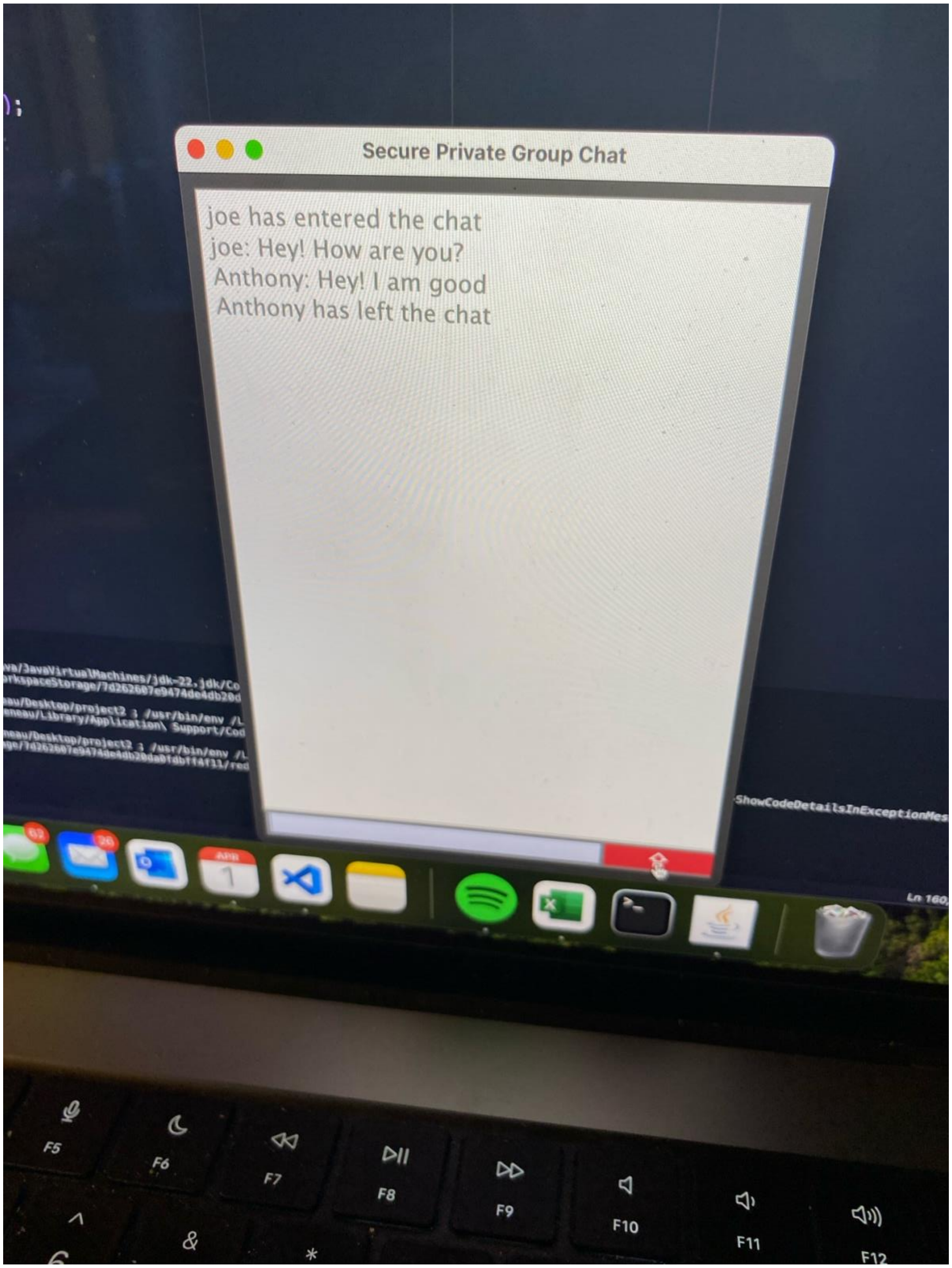


Figure 9 Top computer had a successful authentication. Bottom computer receives a message saying, "joe has entered the chat."



```
Anthony has entered the chat
joe has entered the chat
joe: Hey! How are you?
Anthony: Hey! I am good
```

Figure 10 View of the group chat from the perspective of Anthony.

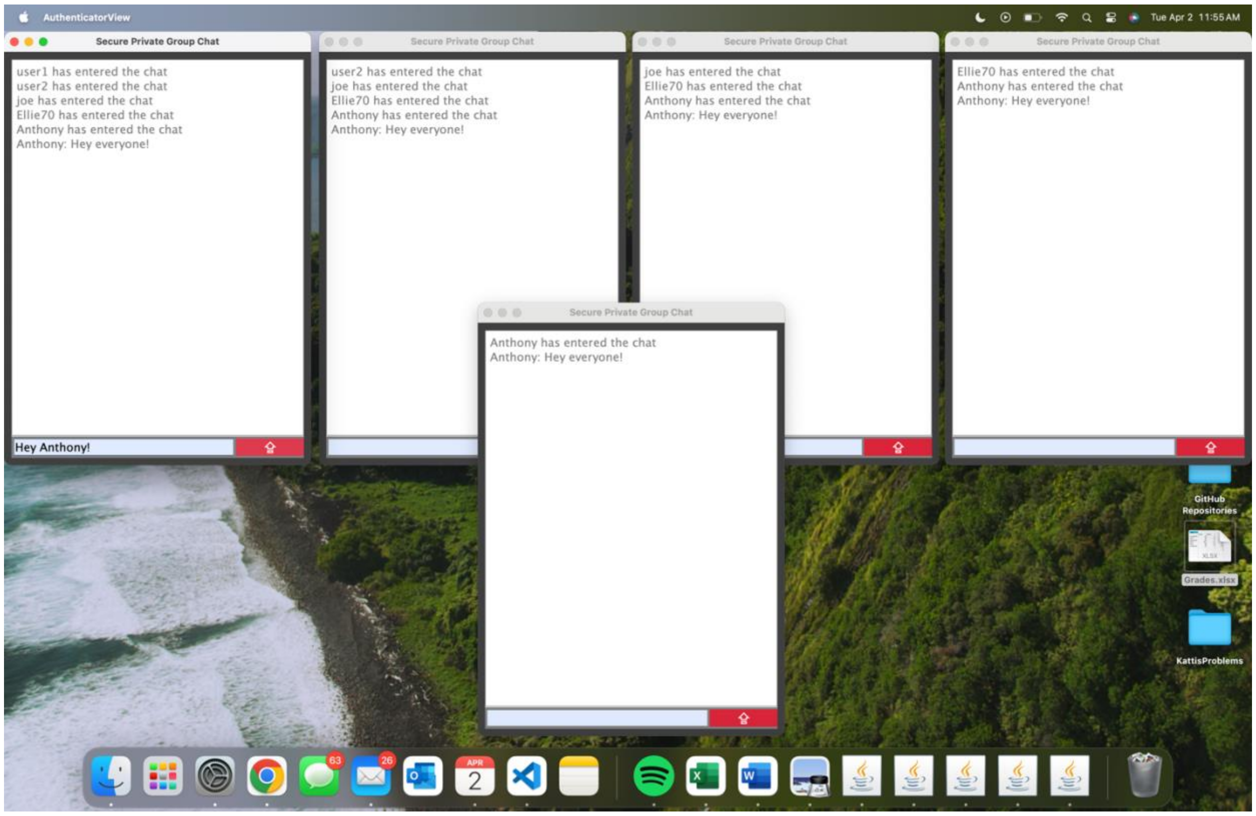


joe has entered the chat
joe: Hey! How are you?
Anthony: Hey! I am good
Anthony has left the chat

Figure 11 View of the group chat from the perspective of joe. The user, Anthony (bottom computer), has left the group chat at the end of the conversation.

Sample 2

Here, the clients only use the local 127.0.0.1 address for testing. Five clients successfully connected to the group chat. The following 12-14 figures show a conversation with everyone:



```
user1 has entered the chat
user2 has entered the chat
joe has entered the chat
Ellie70 has entered the chat
Anthony has entered the chat
Anthony: Hey everyone!
```

```
Hey Anthony!
```

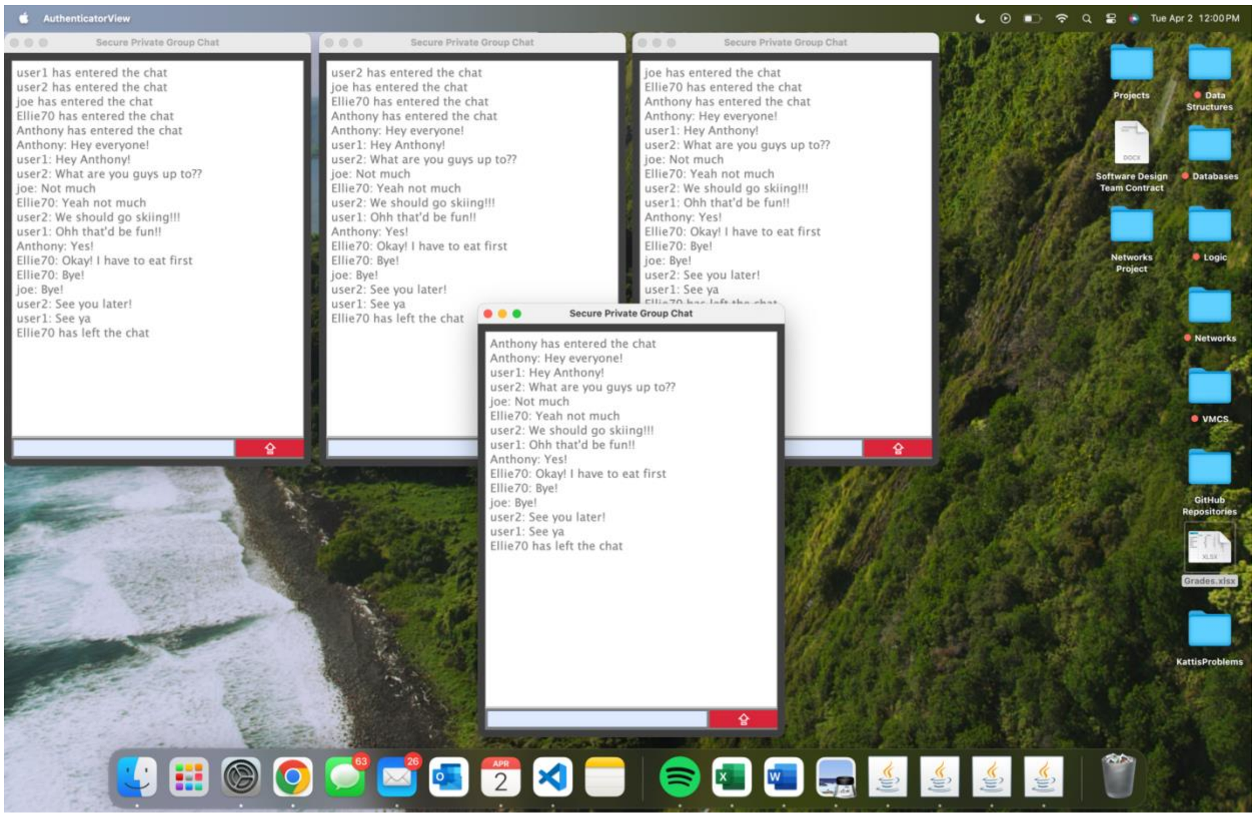
```
user2 has entered the chat
joe has entered the chat
Ellie70 has entered the chat
Anthony has entered the chat
Anthony: Hey everyone!
```

```
joe has entered the chat
Ellie70 has entered the chat
Anthony has entered the chat
Anthony: Hey everyone!
```

```
Ellie70 has entered the chat
Anthony has entered the chat
Anthony: Hey everyone!
```

```
Anthony has entered the chat
Anthony: Hey everyone!
```

Figure 12 Five windows opened for five different users. In order of connections: user1, user2, joe, Ellie70, Anthony.



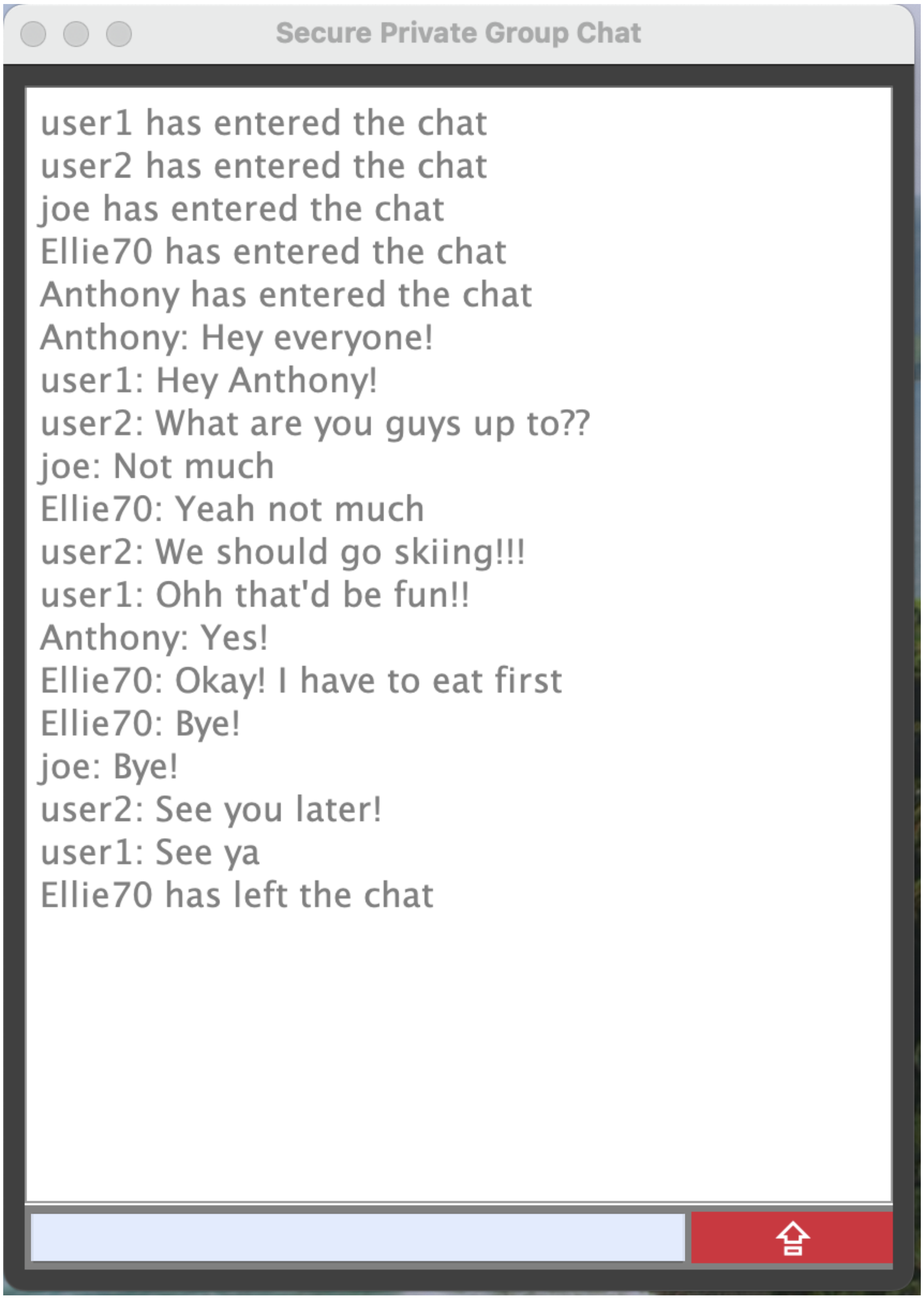
```
user1 has entered the chat
user2 has entered the chat
joe has entered the chat
Ellie70 has entered the chat
Anthony has entered the chat
Anthony: Hey everyone!
user1: Hey Anthony!
user2: What are you guys up to??
joe: Not much
Ellie70: Yeah not much
user2: We should go skiing!!!
user1: Ohh that'd be fun!!
Anthony: Yes!
Ellie70: Okay! I have to eat first
Ellie70: Bye!
joe: Bye!
user2: See you later!
user1: See ya
Ellie70 has left the chat
```

```
user2 has entered the chat
joe has entered the chat
Ellie70 has entered the chat
Anthony has entered the chat
Anthony: Hey everyone!
user1: Hey Anthony!
user2: What are you guys up to??
joe: Not much
Ellie70: Yeah not much
user2: We should go skiing!!!
user1: Ohh that'd be fun!!
Anthony: Yes!
Ellie70: Okay! I have to eat first
Ellie70: Bye!
joe: Bye!
user2: See you later!
user1: See ya
Ellie70 has left the chat
```

```
joe has entered the chat
Ellie70 has entered the chat
Anthony has entered the chat
Anthony: Hey everyone!
user1: Hey Anthony!
user2: What are you guys up to??
joe: Not much
Ellie70: Yeah not much
user2: We should go skiing!!!
user1: Ohh that'd be fun!!
Anthony: Yes!
Ellie70: Okay! I have to eat first
Ellie70: Bye!
joe: Bye!
user2: See you later!
user1: See ya
Ellie70 has left the chat
```

```
Anthony has entered the chat
Anthony: Hey everyone!
user1: Hey Anthony!
user2: What are you guys up to??
joe: Not much
Ellie70: Yeah not much
user2: We should go skiing!!!
user1: Ohh that'd be fun!!
Anthony: Yes!
Ellie70: Okay! I have to eat first
Ellie70: Bye!
joe: Bye!
user2: See you later!
user1: See ya
Ellie70 has left the chat
```

Figure 13 Full conversation with the five connected users. Ellie70 left the chat at the end of the conversation.



```
user1 has entered the chat
user2 has entered the chat
joe has entered the chat
Ellie70 has entered the chat
Anthony has entered the chat
Anthony: Hey everyone!
user1: Hey Anthony!
user2: What are you guys up to??
joe: Not much
Ellie70: Yeah not much
user2: We should go skiing!!!
user1: Ohh that'd be fun!!
Anthony: Yes!
Ellie70: Okay! I have to eat first
Ellie70: Bye!
joe: Bye!
user2: See you later!
user1: See ya
Ellie70 has left the chat
```

Figure 14 Full conversation of the group chat from the perspective of user1, the first connected user in the conversation.