Foundation of Cybersecurity
Project report

Alycia Lisito
Lorenzo Giorgi
Sina Gholami

Academic Year 2020/21

Contents

Chapter 1

Introduction

The purpose of this project is to create an online chat application. In this app, all the exchanges between two clients will go through the server which has to handle several clients at the same time. The following image depicts all the steps required to implement the application.
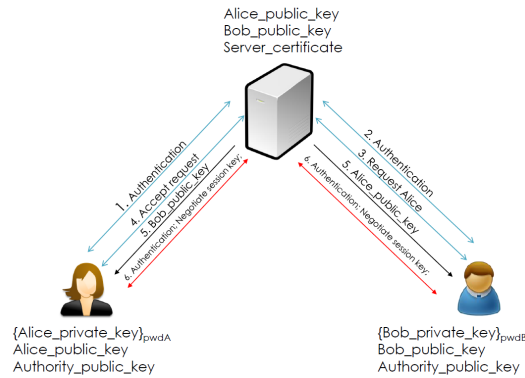

Figure 1.1: Basic Idea of Communications

The server is honest but curious and it will not communicate false public keys on purpose. When the server communicates the public key of the user "Alice", the receiving client trusts that the server has given them Alice's real public key. However, it will try to understand the content of the communications between two clients, thus, a secure communication in which the server cannot intercepts the exchanged messages between two clients must be provided.

Some assumptions were considered about the users of the application:

- Users are already registered on the server through public key.

- Users authenticate themselves through said public key.

- After the log-in, a user can see other available users logged to the server.

- A user can send a "request to talk" to another user.

- The user who receives the "request to talk" can either accept or deny it.

- If the request is accepted, the users proceed to chat through the server using an end-to-end encrypted and authenticated communication.

- When a chat starts, the users cannot start another chat (only up to one chat active at the same time for each user).

- When a user wants to stop chatting, they have to log-off from the server.

When the client application starts, server and client must authenticate themselves. Server authenticates with a public key certified by a certification authority. Client authenticates with a public key (pre-installed on server) and the corresponding private key is protected with a password for each client. After the authentication, a symmetric session key is negotiated. The negotiation provides **Perfect Forward Secrecy**.

All session messages are encrypted and authenticated. Every message in the session is protected against replay attacks. After a «request to talk» is accepted, the server sends to both clients the public key of the other client. Before starting the chat a symmetric session key is negotiated. Again the negotiation provides **Perfect Forward Secrecy**.

1.1   The steps of the app

When the client ("*Alice*") wants to enter in the app, the first step is the log in. Once *Alice* is logged in the app, she can either consult the list of available users in order to send a request or she can log out of the app.

If *Alice* finds a user ("*Bob*") with whom she wants to talk, she can send him a request. *Bob* can then either accept or deny this request. If *Bob* accepts it, then they can start to chat but if *Bob* denies it, then both *Alice* and *Bob* go back to the original menu.

If *Alice* receives a request from *Bob*, she can accept it and then they start to chat, or deny it and again they go back to the original menu.

When *Alice* and/or *Bob* want(s) to stop chatting, one of them has to log out of the conversation which leads them back to the original menu.

Chapter 2

Implementation

To follow these steps and provide aforementioned requirements for the communication, nineteen distinct messages, which all have their use in different cases, were created. Each message contains a **Message Type** field one byte of data. Except Message Type 0 (MT0), the other messages contain at least a 2 bytes **Counter** which provides freshness and prevents replay attacks. In addition, some messages contain **Username** as the identifier of the sender or receiver of the message. In this application, up to 32 bytes were considered for the usernames. Since Galois Counter Mode (except MT0) is used, each message contains additional authenticated data (AAD), encrypted plain text and tag. The tag is 16 bytes and is put at the end of the data frame. Message type, counters and initialization vectors(IV) are placed inside the AAD. The (IV) is 12 bytes.

Generally the data frame which is sent over the network is as below:

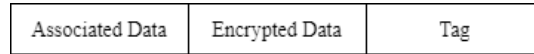| Associated Data	Encrypted Data	Tag

Figure 2.1: Data frame

As mentioned before, counter are used to provide freshness and prevent replay attacks. To ease the implementation, four different counters were created:

- Server to client counter or **Counter$_{Sx}$**: (x is the client who receives the message from the server) this counter counts the number of messages that the server has sent, so far, to this client.

- Client to server counter or **Counter$_{Xs}$**: (X is the client who sends the message to the server) this counter counts the number of messages that the client has sent, so far, to the server.

- Send counter: counts the number of messages that a client has sent, so far, to another client.

- Receive counter: counts the number of messages that a client has received, so far, from another client.

For instance, $Counter_{Ab}$ counts the number of messages the client A has sent to the client B until this moment. It is important to note that the receive counter is not present inside the messages, because a party does not send to another party the number of messages that it has received so far. It is just a way to check that the number of received messages from a party x is equal to the number of messages party x has sent so far.

2.1 The Protocol

The **Station to Station (STS) protocol** was used to guarantee perfect forward secrecy, as it is shown, in more details, in the figure bellow:
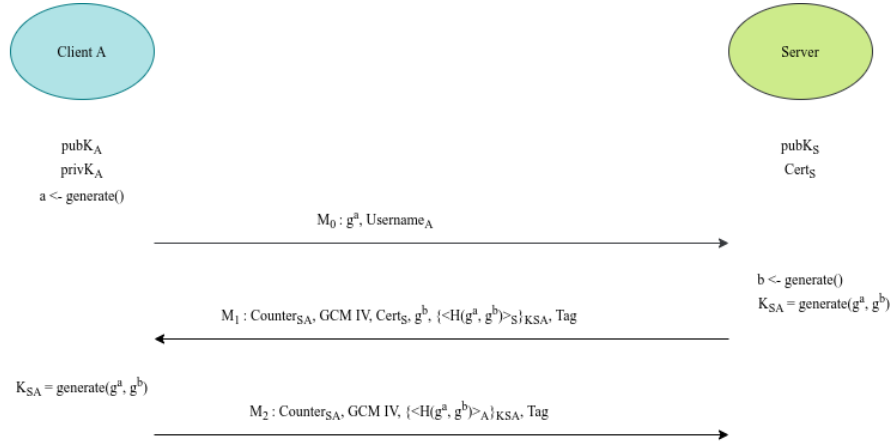
Figure 2.2: Station to Station Protocol

The following algorithms were considered in the protocol:

- AES 256 for GCM encryption
- SHA 256 for hash function
- DH 2048 224 for key establishment
- RSA 3072 for digital signature

The clients and the server use the same algorithms.

2.2 Server-Client Handshake Protocol

In the figure below, the exchanged messages used to establish the key between the client and the server according to the station to station protocol are shown in details:
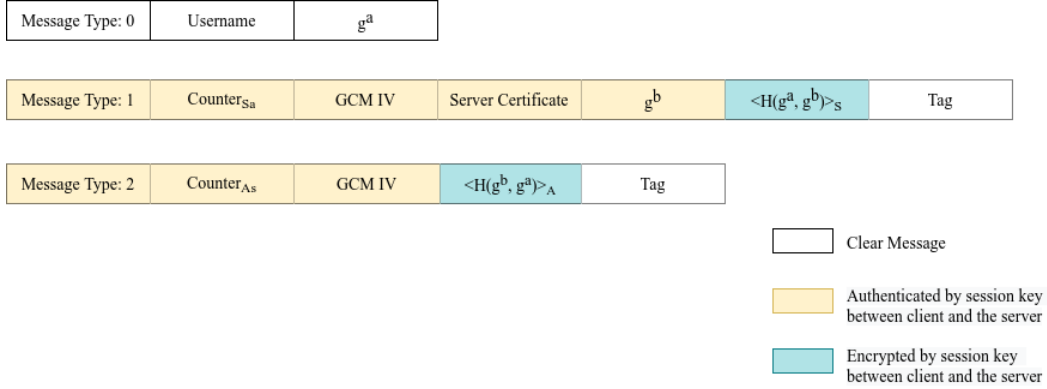
Message Type: 0	Username	g^a

Message Type: 1	$Counter_{Sa}$	GCM IV	Server Certificate	g^b	$<H(g^a, g^b)>_S$	Tag

Message Type: 2	$Counter_{As}$	GCM IV	$<H(g^b, g^a)>_A$	Tag

☐ Clear Message

▭ Authenticated by session key between client and the server

▭ Encrypted by session key between client and the server

Figure 2.3: Server-Cleint Handshake Messages

In the MT0, the client sends three fragments of data: Message Type, Username and DH public key ($g^a[p]$ or just g^a). This message is not encrypted nor authenticated according to the first message in the STS protocol. Upon log into the client application, the client generates a new DH public and private keys and forwards the public key to the server in order to establish a new session key with it.

If the server receives MT0 successfully, it replies back to the client with MT1. First the server generates new DH private and public keys according to the same DH generation algorithm than the client. Then the server generates a session key with the received client's public key and its own DH keys: $k_{session} = (g^b)^a[p]$. After the generation of the session key, the server concatenates the client's and server's public keys, signs it by using its own RSA private key and finally hashes it. The result of the hash is encrypted with the generated session key. The server also sends its certificate to show the client its validity.

If the client receives MT1 with success, after reception, the client gets the DH public key of the server ($g^b[p]$) and generates the session key using the same approach: $k_{session} = (g^a)^b$. Finally, the client concatenates the DH public key of the server with their own, computes the hash, signs it using their RSA private key and then encrypts it using the generated session key.

When the server receives MT2, it believes that it and the client use the same key.

For instance, if Alice is logged into the client application, the following messages will be exchanged between her and the server in order to establish the session key between them.
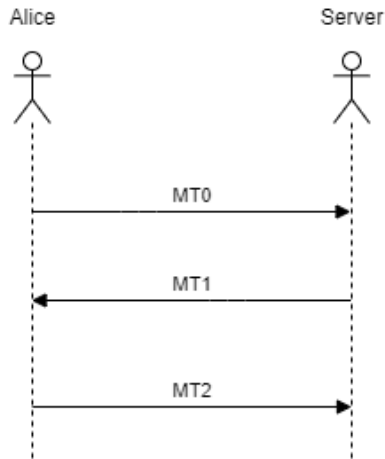
Figure 2.4: Exchanged messages between Alice and the Server

2.3 Get the list of online users

After the establishment of the session key, the client sends automatically a request to get the list of online users. It is also possible for the client to send this request manually (MT3). Accordingly, the server replies to the client MT4 which contains the list of the active users' usernames.
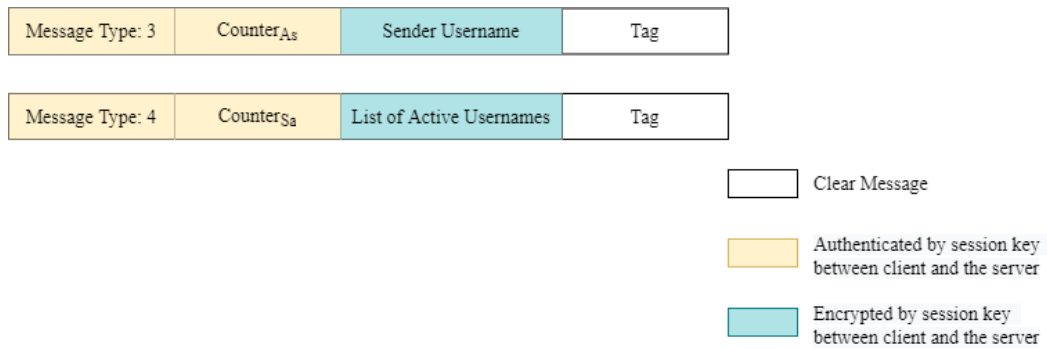

Message Type: 3	Counter$_{As}$	Sender Username	Tag

Message Type: 4	Counter$_{Sa}$	List of Active Usernames	Tag

- Clear Message
- Authenticated by session key between client and the server
- Encrypted by session key between client and the server

Figure 2.5: Alice Requests for Online Users

Now, let's consider that Alice wants to get the list of online users in order to make a request to chat with one of them.
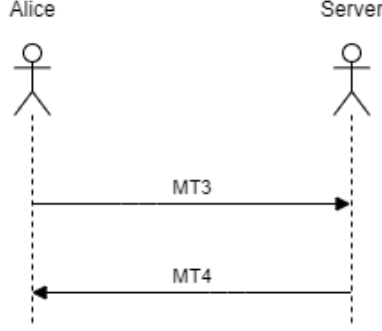
Figure 2.6: Alice requests for the list of online users

2.4 Request to Talk

Let's suppose that there are multiple users online in the application (being online means that the user has successfully established a session key between them and the server and is not chatting with another client). In this scenario, a client can arbitrary choose another client from the list of the available users. If it is the case, a request to talk (MT5) is sent by the requester to the server, and then forwarded (MT6) by the server to the receiver. These two messages are shown in the figure below:

Message Type: 5	Counter$_{As}$	GCM IV	$g^{a'}$	Sender Username	Receiver Username	Tag

Message Type: 6	Counter$_{Sb}$	GCM IV	$g^{a'}$	Sender Username	Receiver username	Tag

Clear Message

Authenticated by session key between client and the server

Encrypted by session key between client and the server
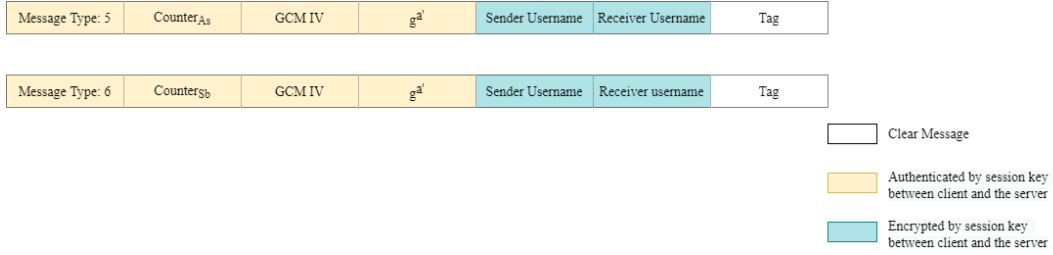
Figure 2.7: Request to talk messages

MT6 encryption key is different from the one used in MT5 because, in MT5 the key used for the encryption is the session key between the requester and the server while in MT6 the key is the session key between the server and the receiver of the message. Just like for MT0, the requester generates new DH public and private keys and place the public key ($g^{a'}[p]$ or simply $g^{a'}$) inside the MT5.

2.5 Client-Client Handshake protocol

Let's assume that the receiver of the request to talk wants to accept it. In this case, several messages will be exchanged to establish the ephemeral key between the two clients. The exchanged

messages can be found below:
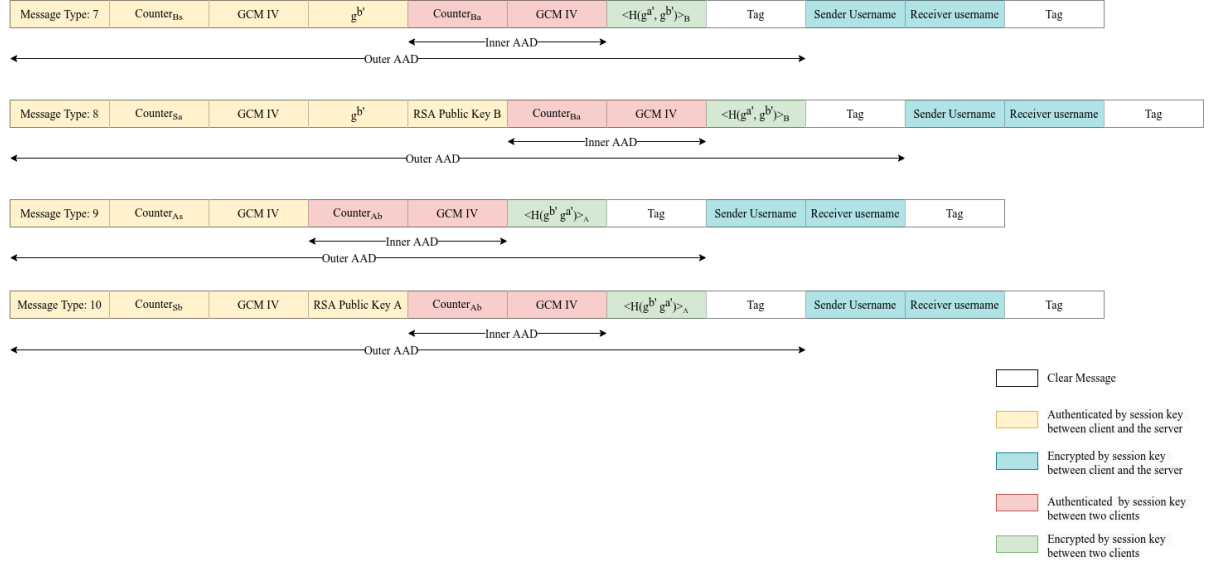
Figure 2.8: Client-client handshake messages

In the MT7, the receiver of the request to talk also generates new DH public and private keys. Then by using the other DH public key $(g^{a'})$ and their own, they generates the new session key: $k'_{session} = (g^{b'})^{a'}$. They concatenates the two DH public keys $(g^{a'}, g^{b'})$ and computes the hash of it. Then, they signs it with their own RSA private key and encrypts the signature using the new generated session key $k'_{session}$. For this encryption part, a new counter is also considered between the two clients as well as a new different initialisation vector. Both of these (counter and IV) are put inside the message data frame.

It is important to note that this message has two encryption parts. Indeed, the green part is encrypted using the session key $k'_{session}$ between the two clients, while the blue part is encrypted using the session key $k_{session}$ between the client and the server. The AAD of the outer GCM is composed by the yellow, red and green parts and the tag of the inner GCM.

The server, then, forwards the MT7 (with the MT8) with an additional information: the RSA public key of the receiver of the request to talk. The issuer of the request to talk receives the MT8 and creates the MT9 following the same procedure as the receiver of the request to talk. Again the server forwards the MT9 (with the MT10) to the receiver of the request to talk with the RSA public key of the requester. In this way, the receiver makes sure that the issuer is also using the same ephemeral key. The following figure shows the MT4 to MT10 sequence between two clients and the server.

9
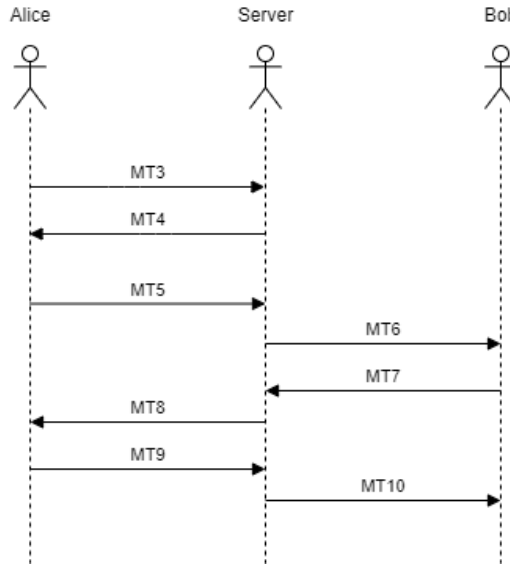
Figure 2.9: Key's establishment between Alice and Bob

2.6 Deny request to talk

In the case in which the receiver of the request to talk wants to deny the request, two messages were implemented. One is for the rejection of the request and the other is to forward the message from the server to the requester.
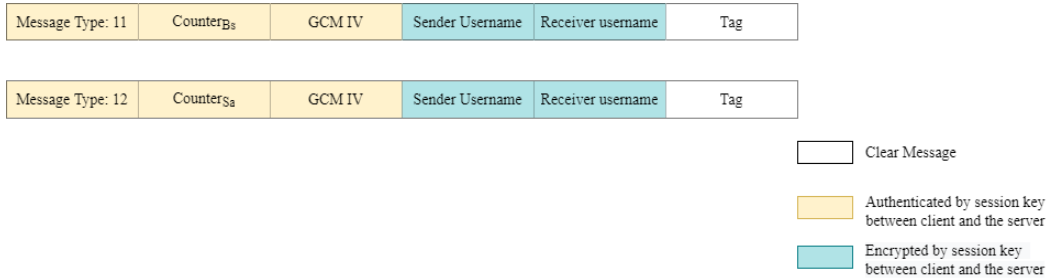
Figure 2.10: Reject Request to Talk Messages

For instance, Alice first asks the server to obtain the list of online users (MT3). Bob is online so the server replies back to Alice that Bob is among the online users (MT4). Alice sends the request to talk to Bob and the server forwards it to Bob (MT5 and MT6). Bob denies the request to talk and the server forwards the rejection to Alice (MT11 and MT12).
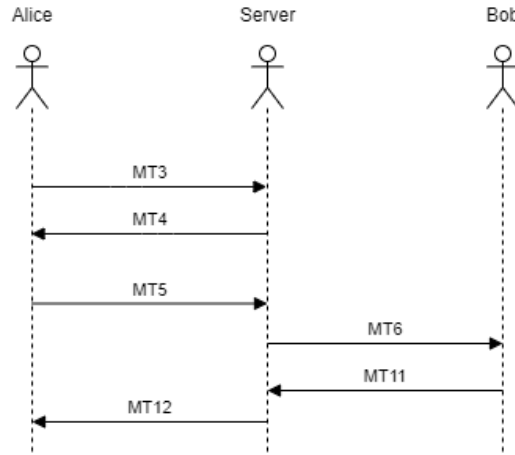
Figure 2.11: Bob Rejects Request to Talk from Alice

2.7 Chatting

Now if the session key is established between two clients, they can start to chat with each other. The two following messages were implemented so the two clients can exchange messages to chat. In the MT13, the sender sends a message while in the MT15 the message is forwarded from the server to the receiver. Each message have maximum $10k$ characters, as a consequence, if the user inserts more characters, the message is divided into portions of $10k$ characters which are sent toward the server.

Figure 2.12: Chatting messages

At some point during the chat, one of the client may decide to stop talking with the other client and start talking with another one. In this case, the client sends the log out of the conversation

message (MT15) and the server forwards it (MT16) to the other client (just to inform them that the other client has exited the chat). The two messages for this purpose are as follow:

Message Type: 15	Counter$_{As}$	GCM IV	Sender Username	Receiver username	Tag

Message Type: 16	Counter$_{Sb}$	GCM IV	Sender Username	Receiver username	Tag

☐ Clear Message

☐ Authenticated by session key between client and the server

☐ Encrypted by session key between client and the server
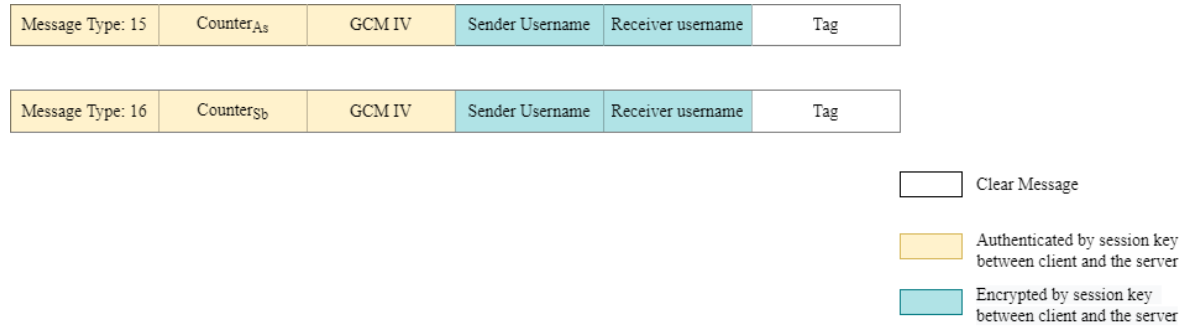
Figure 2.13: Exit Chatting Messages

2.8   Logout & error messages

There are only two messages left: one to log out a client from the app and the other one for the server's errors.

If a client wants to log out, the server must be informed, the MT17 was created for this reason. This way, the server knows that the client logged out of the app so it must delete all the session information related to this client (such as the session key and counters). In addition, if the client suddenly closes their application while chatting, the other client and the server must be notified. In this case, the server also sends the MT16 to the other client to notify them that the conversation is over.

After the establishment of the key between the server and the client, the server might face an issue in parsing the received message. In this case, the server replies back to the corresponding client that it cannot handle their request.

Message Type: 17	Counter$_{As}$	GCM IV	Sender Username	Tag

Message Type: e	Counter$_{Sa}$	GCM IV	Error Message	Tag

☐ Clear Message

☐ Authenticated by session key between client and the server

☐ Encrypted by session key between client and the server
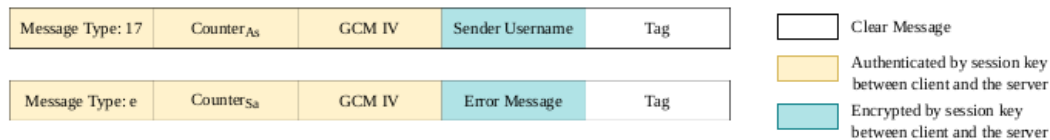
Figure 2.14: MT17 & error messages

In the following sequence diagram, the whole protocol from MT0 to MT16 is shown. MT11 and

MT12 are not represented because here the request to talk was accepted.
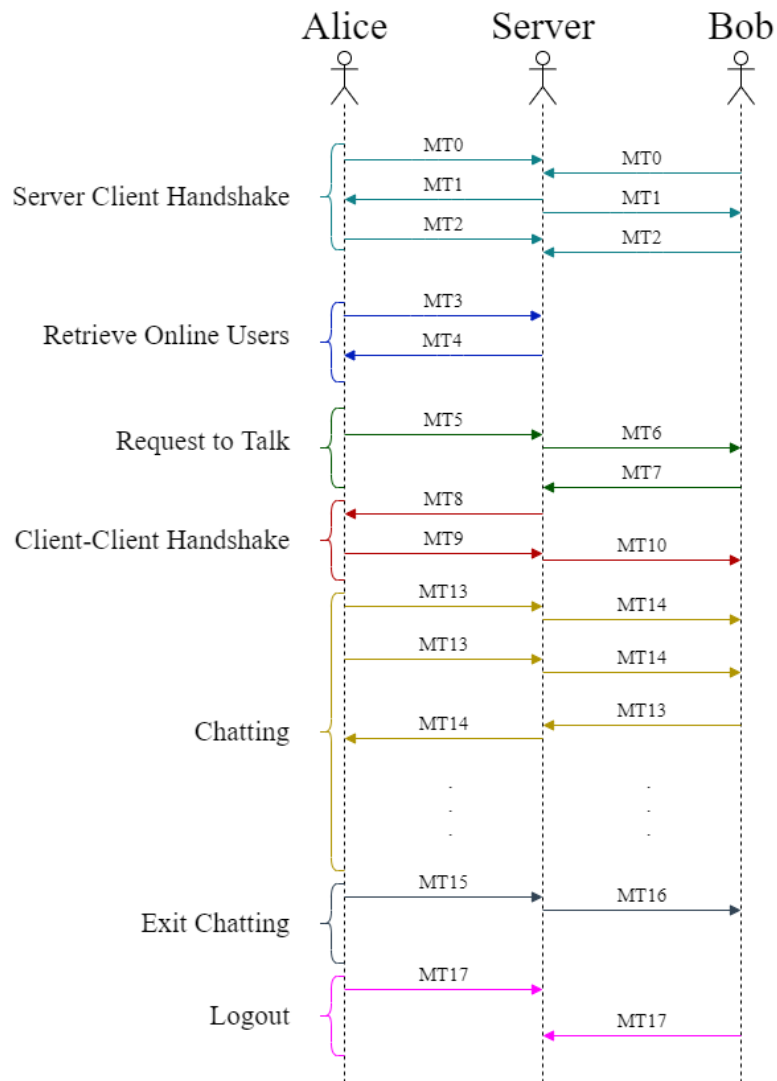
Figure 2.15: Sequence of Messages

Chapter 3

User Manual

Before starting the server or the client, the source files need to be compiled. To simplify this task, a makefile has been created:

- `make` or `make all`: compile both the client and the server.

- `make server`: compiles only the server.

- `make client`: compiles only the client.

- `make clean`: deletes the client and server executable.

Note: C++ compiler is required to be installed on the system (`g++` was used).

3.1 Starting the server

Once the code is compiled, the server can be run with the command: `./server port`
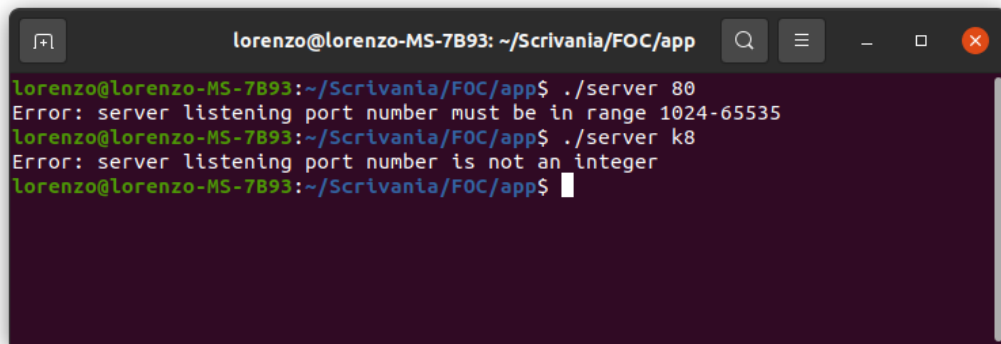
Figure 3.1: Server start

port is the port the server uses to accept incoming request from clients. The **port** parameter is optional: if no parameter is provided, the server runs on port 8888.

port must be an integer number in the range 1024-65535. It is not possible to use a port below 1024 because they are standard ports and the only possible way to use them is with root privileges.

If an invalid value of **port** is provided, the user sees in the terminal an error with a meaningful message regarding the error they made.
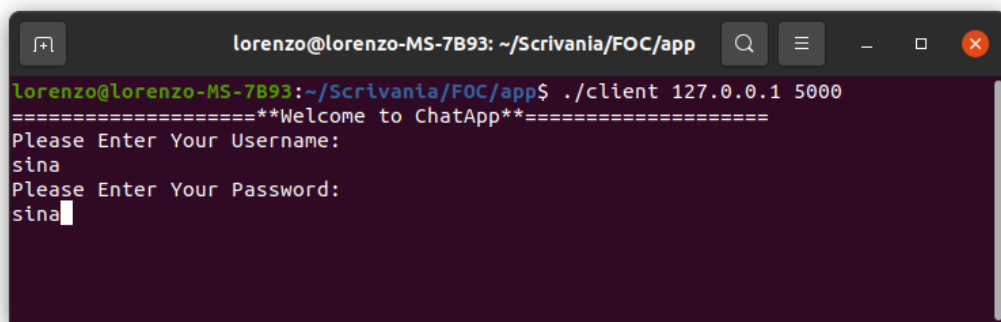
Figure 3.2: Server start errors due to wrong *port* parameter provided

3.2   Login with a client

The user starts the client using: `./client IP port`

Figure 3.3: Client start

With:

- `IP`: is the IP address of the server. If not provided by the user 127.0.0.1 is used as default.

- `port`: is the port on which the server application listen. If not provided by the user 8888 is used as default.
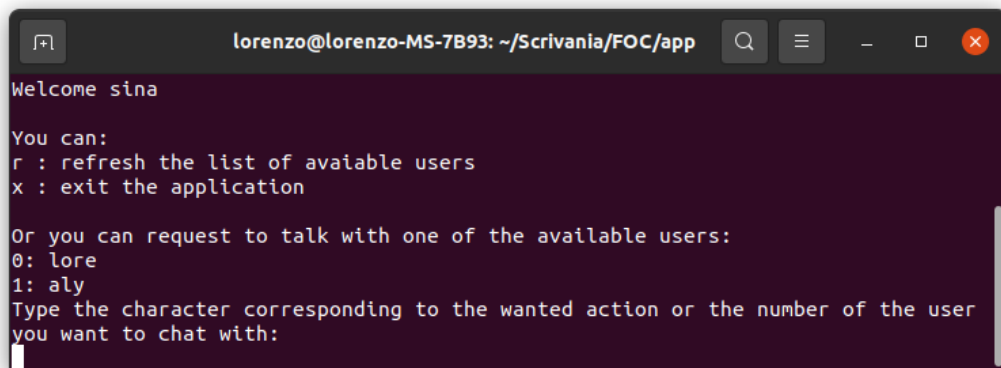
Note: it is not possible to specify a custom port without specifying the IP address. For example:

- `./client 5000` does not work.

- `./client 172.16.3.15` works using default port 8888.

- `./client` works using all the default parameters (127.0.0.1 and 8888).

The user is asked to enter their username and password to access the service. The password to be entered is the one that protects the access to the user's private key file. If the user enters an incorrect username or password, the application asks the user to enter them again.

When the user enters the correct data, the client tries to establish a secure communication with the server using the protocol described in paragraph 2.1. The server will not accept multiple connections from the same user, in this case the second connection attempt will fail and the user will receive an error message.

Once the session key is established between the client and the server, the main menu is shown to the user with the list of online users.

Figure 3.4: Client main menu

From this interface it is possible to update the list of online users, log out from the app or send a request to talk to one of the available users. If a request to talk arrives, it is displayed to the user, who can decide whether to accept or deny it. While chatting it is not possible to receive
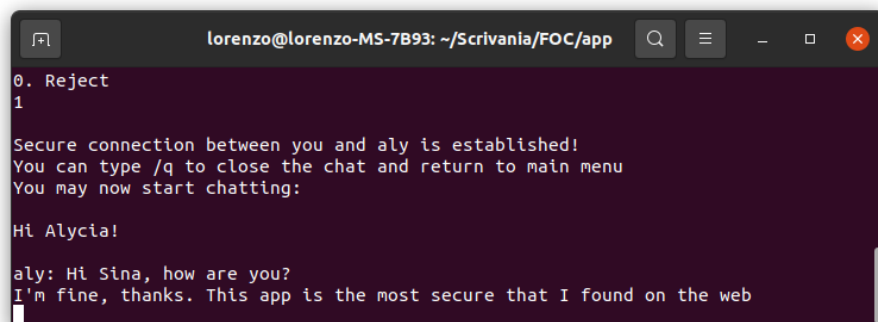
Figure 3.5: Request to talk received by a user. The user can accept or reject it.

request: the server keeps track of the user's status (so if the user is already chatting, it automatically denies the request). A user who is chatting is also removed from the list of the available users.

Once the user has accepted the request to talk, the two clients are put in communication through a secure logical communication channel. The two clients never send messages to each other directly, but always through the server, as explained earlier.

Figure 3.6: Chat interface for the client receiving the RTT

With the chat interface, it is possible to send messages to the other client. A message is sent when the enter key is pressed. Messages from the other client are printed on the terminal as soon as they are received. It is possible to interrupt a conversation by simply typing /q and pressing the enter key. The other client will be informed by the server according to the protocols previously defined. Both clients will return in the main menu, ready to start a new conversation.

Figure 3.7: Chat interface for the client that sent the RTT