UNIVERSITÀ DI PISA

Foundations of Cybersecurity
Master Degree in Artificial Intelligence and Data Engineering
Year 2020/21

Secure Chat Messaging

Luca BARSELLOTTI
Marco BONGIOVANNI
Francesco MARABOTTO

1 Introduction

The goal of the project is to implement a secure chat application among two clients. The communication must be confidential, authenticated and protected against replay attacks. Users are already registered on the server through public keys that they use to authenticate. After the log-in, a user can see other available users logged to the server. An user can send a "Request to Talk" (RTT) message to another user. The user who receives the RTT can either accept or refuse. If the request is accepted, the users proceed to chat through the server. Instead, if the request is refused, the users are not going to chat.

2 Application Manual

To compile the project and make it executable, a makefile is provided to create both the client and server executable files. Travel to the main folder of the project with a terminal and type "make" to start the building of the executable files.

To launch the server application:

```
./server_main 127.0.0.1 9090 user_list
```

The arguments are, respectively:

- Server address
- Server port
- File containing the list of registered users

To launch the client application:

```
./client_main alice 127.0.0.1 9090
```

The arguments are, respectively:

- Username
- Server address
- Server port

The password to launch the client is "aaa".

First of all, the server must be running to let the application work. Upon launching it, the server waits for requests from clients.

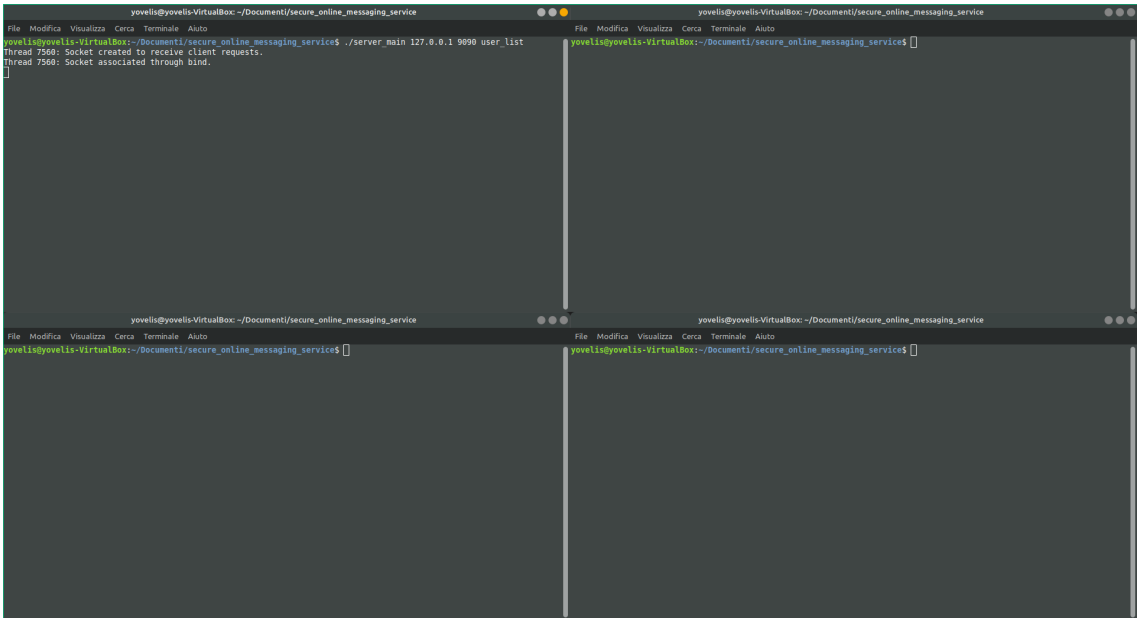


Figure 1: Server start

When the server is active, if a client launches the application, the Authentication Protocol will start and the server will send a message S1 containing a nonce and its certificate.

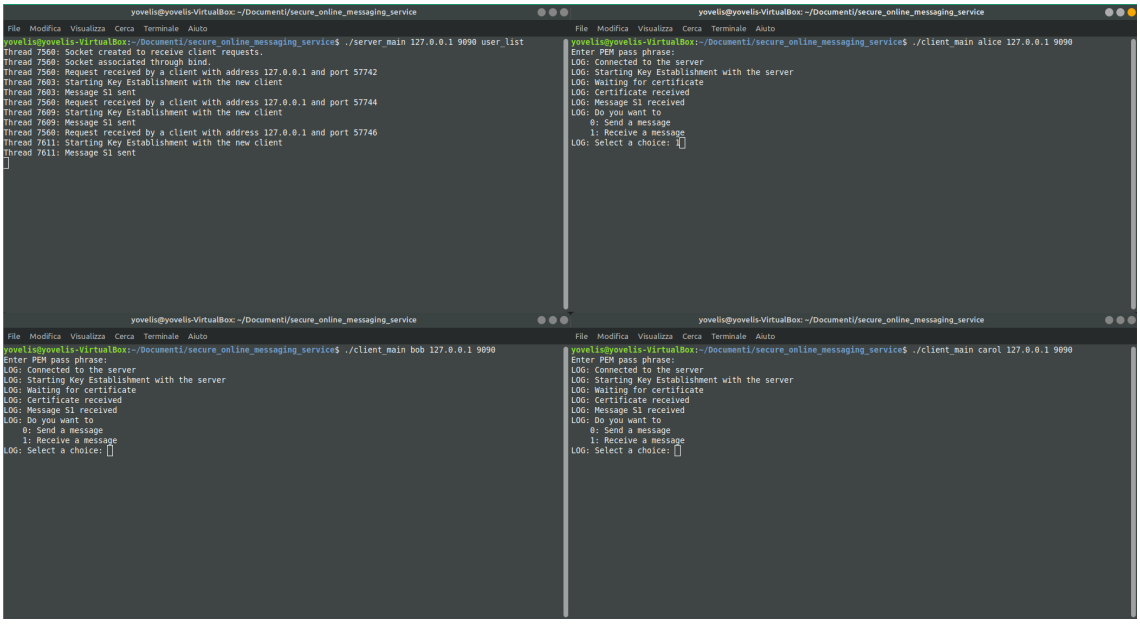


Figure 2: Clients start

Upon receiving S1, the application asks the user which role (s)he wants to interpret. If (s)he chooses to be a Sender, then the server will send him/her the list of available online users, otherwise, if (s)he chooses Receiver, then the application will wait for an RTT from a Sender. Independently from the choice, the key establishment between client and server will be performed to establish a symmetric session key K using ephemeral RSA.

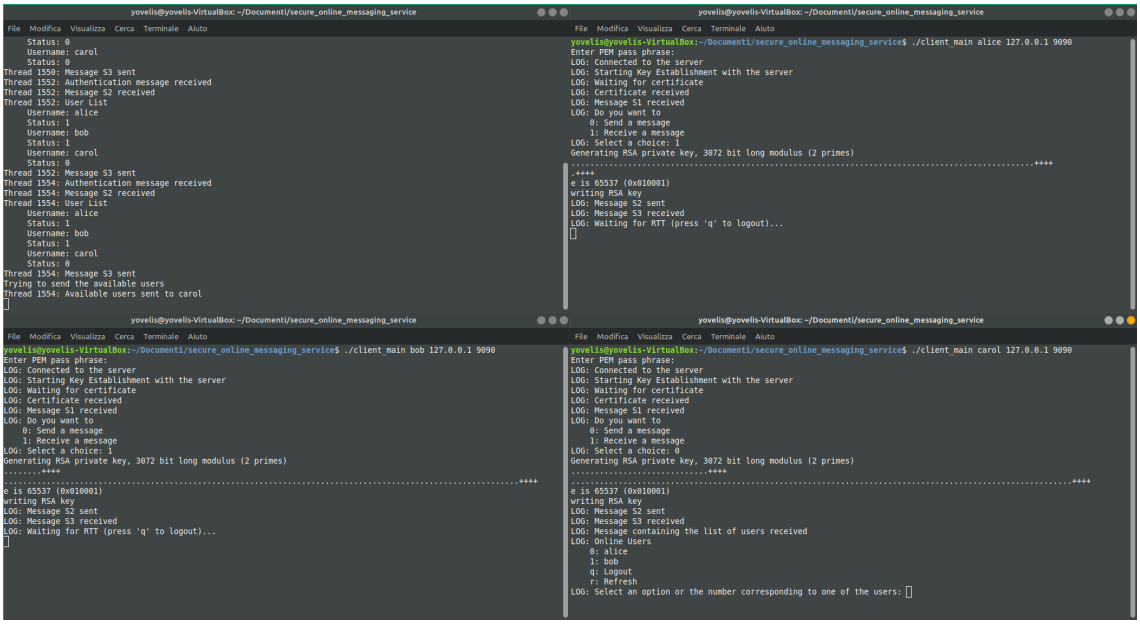


Figure 3: Clients choose their role

Upon receiving the list of available users, the application asks the user to choose one of the available user. Then, an RTT will be sent to that user.

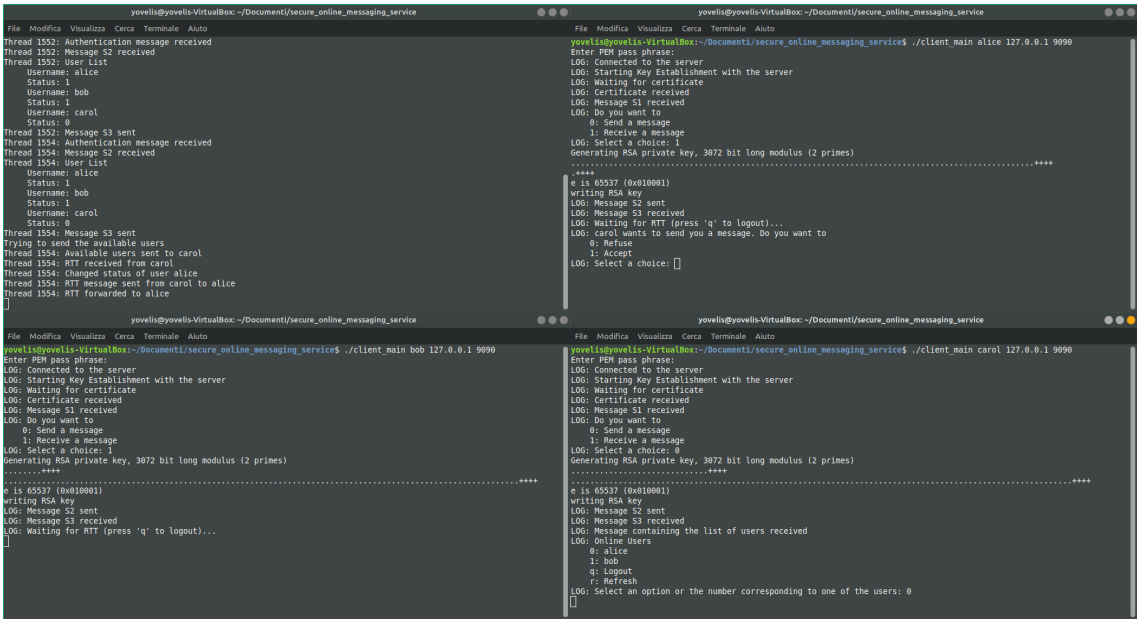


Figure 4: Sender chooses receiver

At this point, the Receiver who receives the RTT from the Sender can choose to accept or refuse it. If (s)he accepts a key establishment procedure starts and a session key is negotiated between the client and the server using ephemeral RSA.

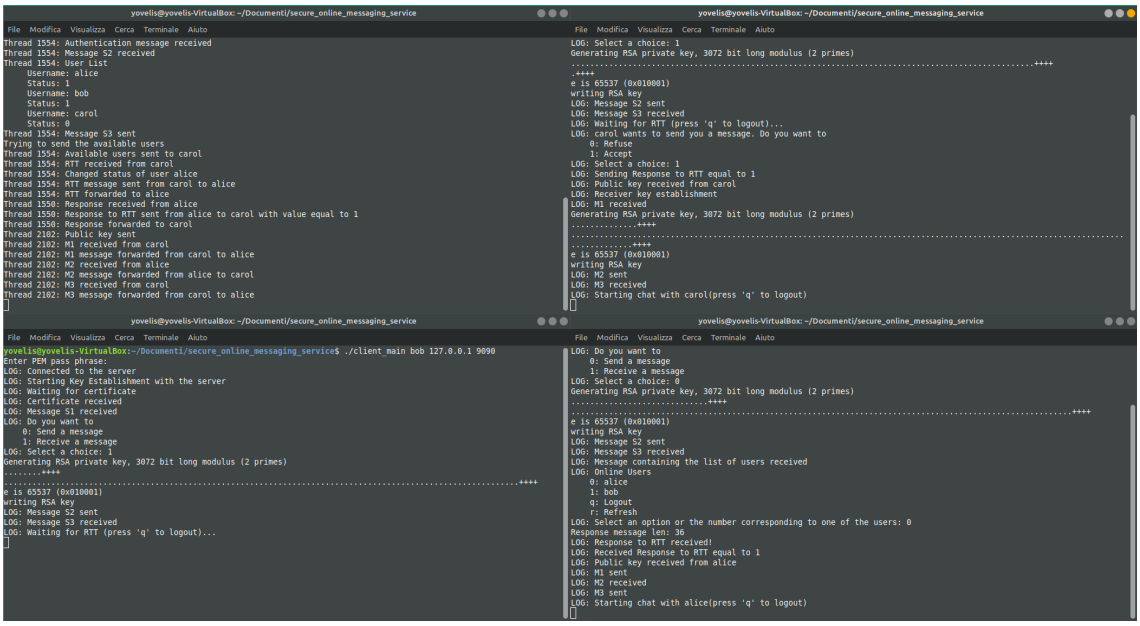


Figure 5: Receiver accepts RTT from sender

When the session key is available for both sender and receiver client, the two peers can start chatting. The chat messages are encrypted two times: one for the communication between client and server and one for the communication between the two peers.

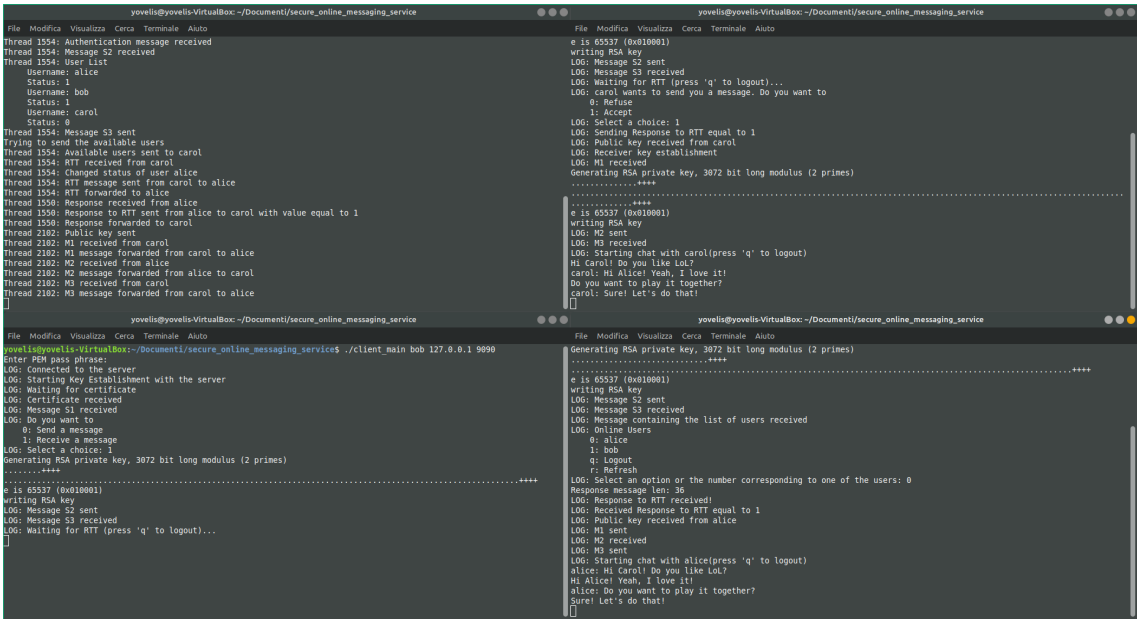


Figure 6: Clients chatting to each other

Finally, if one of the two peers wants to close the chat, (s)he has only to write a message containing the character 'q': a "return to lobby" request is sent to the server, which communicates to the other user to return back to the lobby.

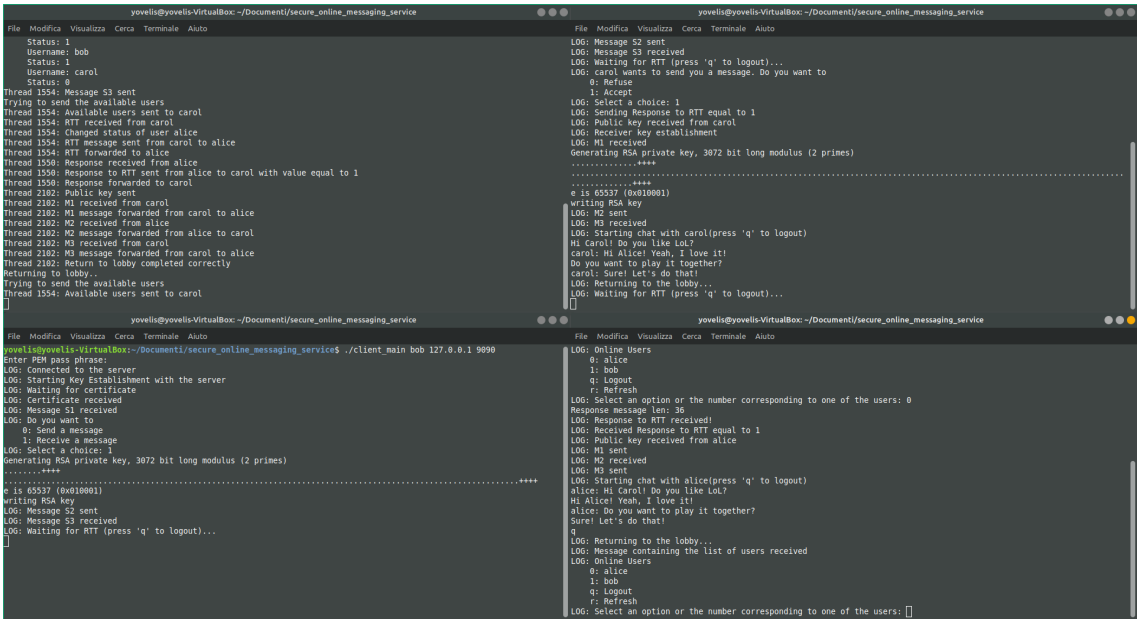


Figure 7: Clients return to lobby

When the receiver user is in the lobby waiting for RTT, (s)he can type 'q' in order to logout from the application, closing the communication with the server.

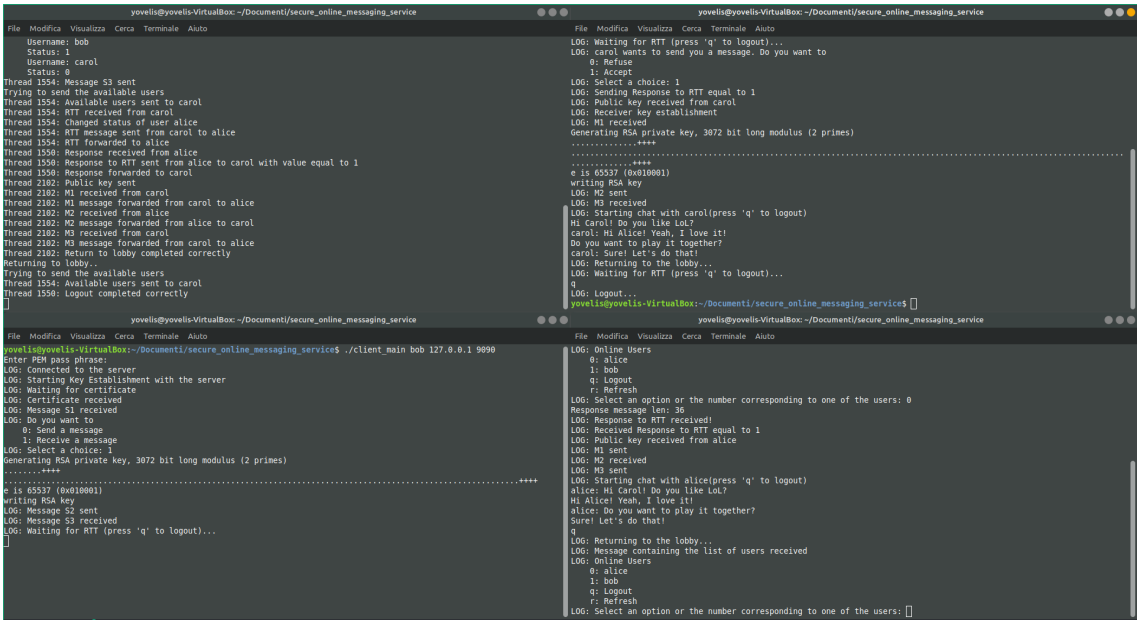


Figure 8: Receiver logouts from the application

While the sender user is in the lobby, (s)he can choose to refresh the list of the available users by typing 'r'. The server will send him/her an updated list of available users.

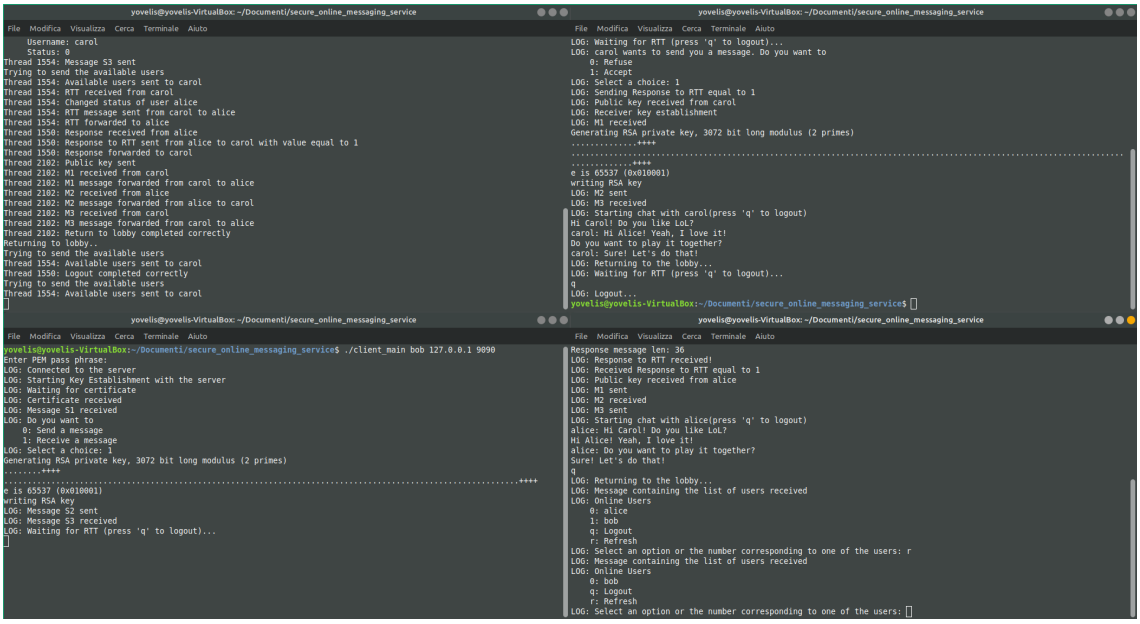
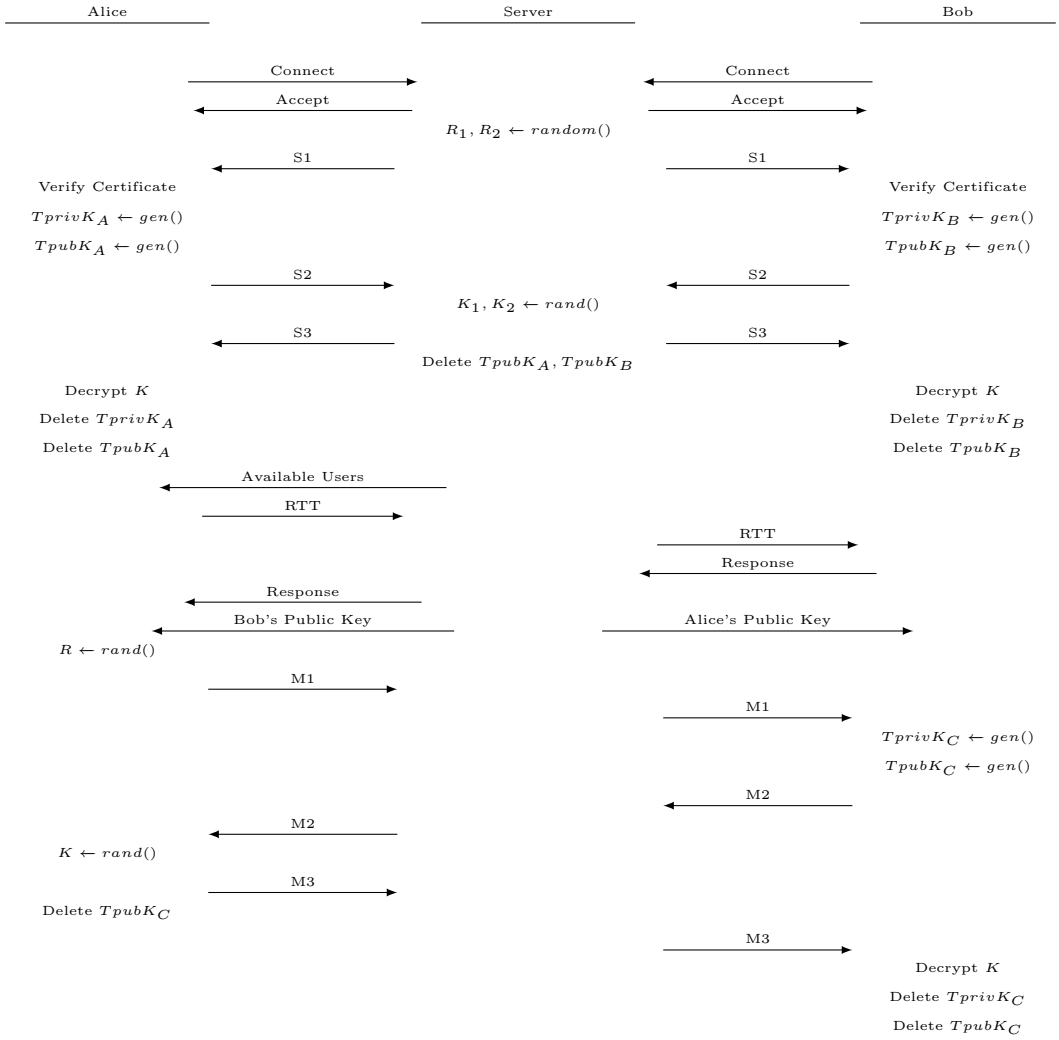


Figure 9: Sender refreshes the list of the available users

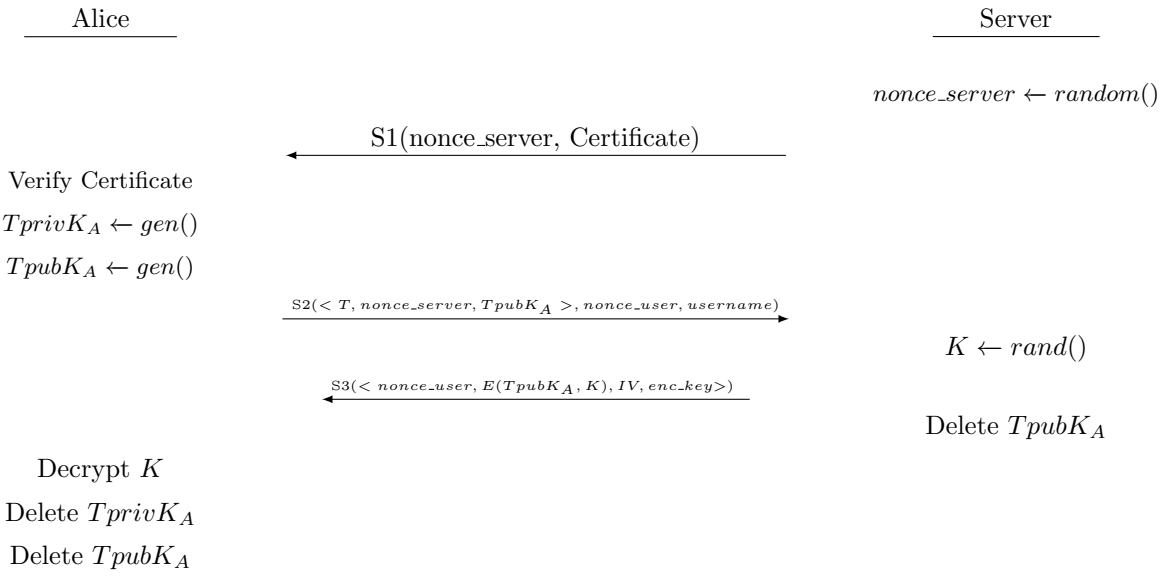
3 Overview of the protocols



4 Design choices

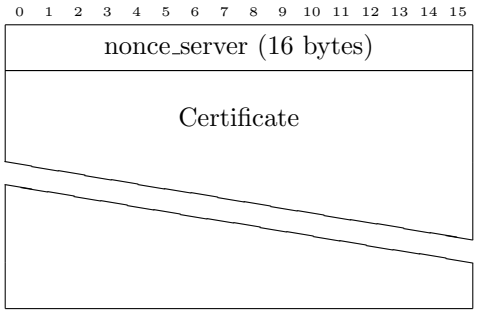
- **TCP**: Considering that there are no requirements regarding the performance of the application in terms of latency, TCP has been considered a good idea since it guarantees that data is not damaged or lost while transmitted.
- **Simple Way** for certificates: the Simple Way is a good choice because it is faster to apply than the Secure Way and it is feasible when the client can trust the Certification Authority. In this project we can assume that the Certification Authority is trustable.
- **AES CBC in combination with RSA**: AES CBC has the advantages of not being deterministic thanks to an Initialization Vector and it is more secure because its encryption is not tolerant of block losses.
- **AES GCM**: The advantage of using AES GCM in respect of AES+HMAC is that it is simpler to implement and less error prone, with the disadvantage of having just one key for both encryption and authentication.

5 Authentication



After the TCP connection between the user and the server, the server sends its certificate to the user in clear with a nonce to verify the authenticity of the user (message S1). Upon receiving the certificate, the user verifies it using the certificate of the Certification Authority and the Certificate Revocation List that (s)he has in local. In this way, the certificate cannot be modified, otherwise it will not pass the verification. If another valid certificate replaces entirely the server certificate, it will not pass the verification of the subject name in the certificate. After the verification of the certificate, the user generates two ephemeral RSA keys, $TpubK_B, TprivK_B$, and sends the message S2 that implements the same mechanism to verify server authenticity with a nonce. Upon receiving the S2 message, the server verifies the authenticity of the user checking if the nonce that it has sent in S1 is equal to the one received and signed by the user in S2. Then, it generates K, encrypts it using RSA in combination with AES CBC and sends S3 signing the nonce received from the user.

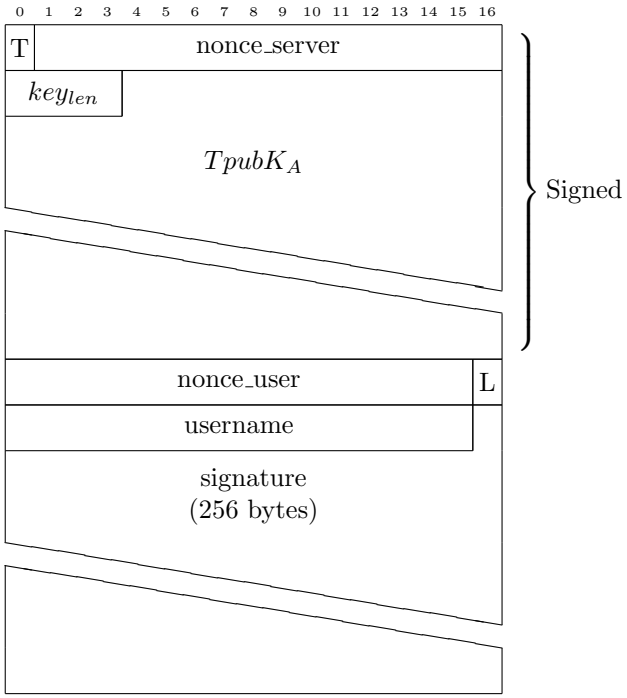
S1:



Fields:

- **nonce_server**: This nonce is generated by the server and is used to verify the authenticity of the user.
- **Certificate**: This is the certificate of the server.

S2:



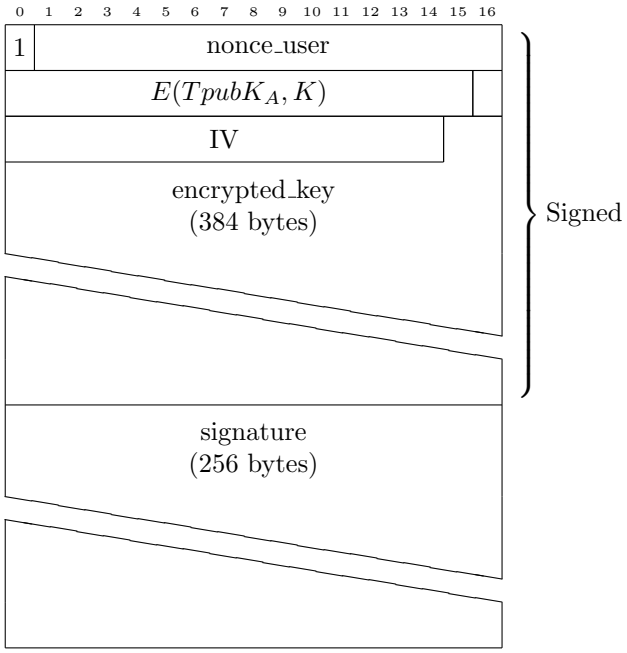
The first byte indicates the type of user:

- 0: The user is a Receiver.
- 1: The user is a Sender.

This value represents also the type of message that the user is sending. A message starting with "0" or "1" is recognized as a "Authentication Message". The other fields:

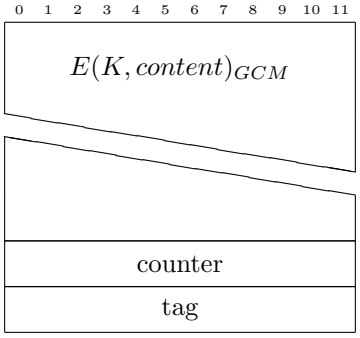
- **nonce_server**: This is the nonce generated by the server.
- key_{len}: It indicates the length of the ephemeral public key generated by the user.
- $TpubK_A$: This is the ephemeral public key generated by the user.
- **nonce_user**: This nonce is generated by the user and is used to verify the authenticity of the server.
- **L**: It indicates the length of the username.
- **username**: This is the username of the user that is authenticating.
- **signature**: The signature is computed on nonce_server, key_{len} and $TpubK_A$ using the user's private key. The server has the user's public key to verify if the signature is correct.

S3:



This value represents also the type of message that the user is sending. A message starting with "0" or "1" is recognized as a "Authentication Message". The other fields:

- **1**: This is the type that indicates that the message is a "S3 message".
- **nonce_user**: This is the nonce generated by the user.
- $E(TpubK_A, K)$: K is the session key that will be used for the communication between the client and the server.
- **IV**: This is the random IV generated for encrypting K with the asymmetric encryption using $TpubK_A$.
- **encrypted_key**: This is the encrypted key used for encrypting K with the asymmetric encryption using $TpubK_A$.
- **signature**: The signature is computed on the nonce_user, the encrypted symmetric session key, the IV and the encrypted key using the server's private key. The user has the server's public key to verify if the signature is correct, because it is included inside the certificate.



From now on, each message sent from server to client or viceversa will be encrypted and authenticated with AES_GCM. The packet will contain:

- $E(K, content)_{GCM}$: The message content will be encrypted using the session key K.
- **counter**: The counter will be used both as IV for encryption and also to guarantee the freshness and to avoid replay attacks from malicious users.

- **tag**: The tag is the AES_GCM signature, computed basing on the IV, passed to AES_GCM as AAD (Additional Authenticated Data).

Two counters, initialized with the pseudo random IV value generated during S3 and reduced to the 12 least significant byte (LSB), are stored and managed by both server and client in local. They increment the counter associated to the one that is sending the message before sending/receiving it, so in this way they both know the expected value.

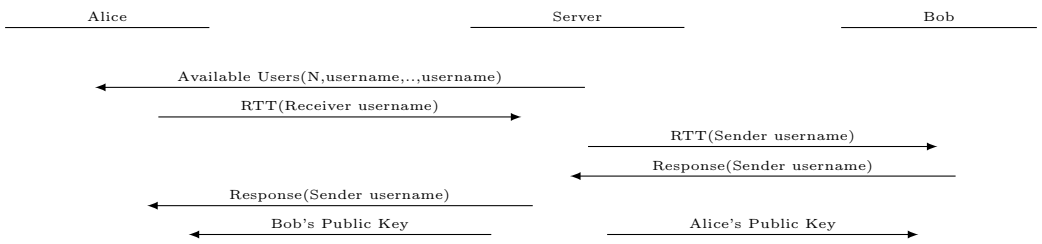
Two counter are used because in this way it is possible to avoid race conditions among the two involved actors of the communication.

6 Logout

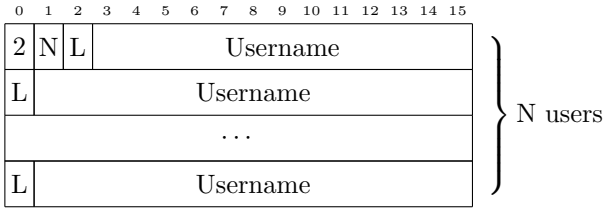
The byte represents the type of message, namely "Logout Message".

Before starting the chat between two clients, the logout message, when received and checked by the server is used to close the thread and the socket that manage the user which has requested to logout.

7 Request to Talk



7.1 Available Users



Fields:

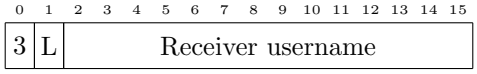
- **2**: This is the type of the message, corresponding to "Available Users Message".
- **N**: This is the number of users contained in the message. The upper bound on the number of users that can be inserted into a single message is given by the dimension of this field. It has been implemented in one byte, corresponding to a maximum of 255 users, but it can be incremented just using more bytes.
- **L**: This indicates the length of the subsequent username.

7.2 Refresh

The byte represents the type of message, namely "Refresh Message".

This message is used by the Sender Client to ask the server to send him/her again the list of the available users.

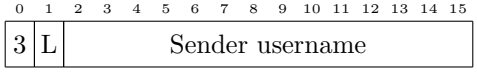
7.3 RTT sent from the Sender User to the Server



Fields:

- **3**: This is the type of the message, corresponding to "RTT Message".
- **L**: This indicates the length of the username.

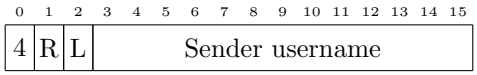
7.4 RTT sent from the Server to the Receiver User



Fields:

- **3**: This is the type of the message, corresponding to "RTT Message".
- **L**: This indicates the length of the username.

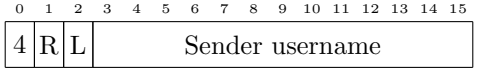
7.5 Response sent from the Receiver User to the Server



Fields:

- **4**: This is the type of the message, corresponding to "Response Message".
- **R**: This indicates the response to the RTT chosen by the receiver (1 for "accept", 0 for "refuse").
- **L**: This indicates the length of the username.

7.6 Response sent from the Server to the Sender User



Fields:

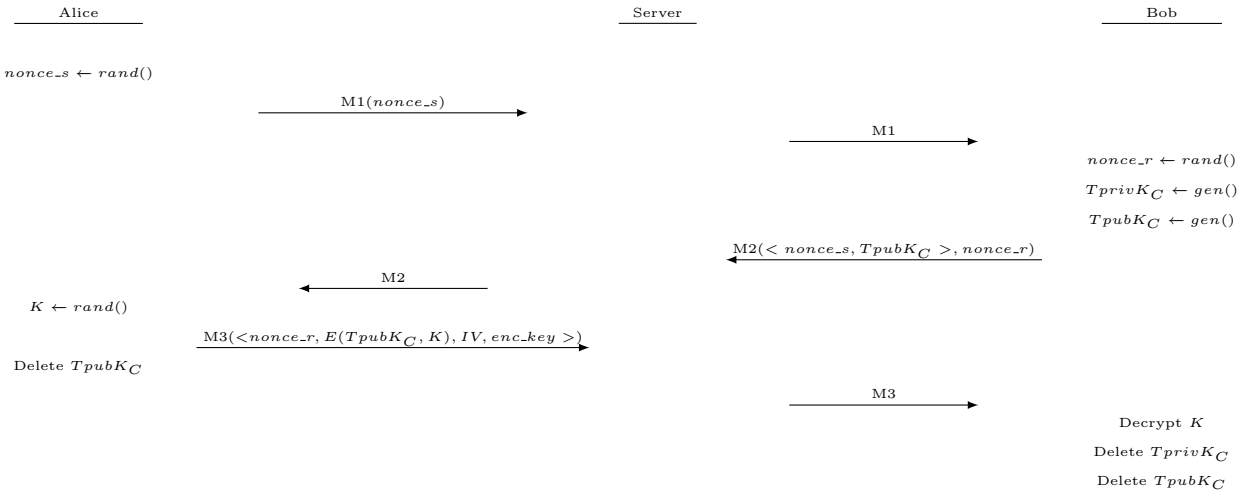
- **4**: This is the type of the message, corresponding to "Response Message".
- **R**: This indicates the response to the RTT chosen by the receiver (1 for "accept", 0 for "refuse").
- **L**: This indicates the length of the username.

8 Bad Response

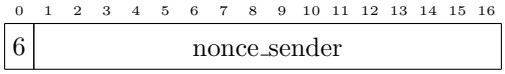
The byte represents the type of message, namely "Bad Response".

The "Bad Response" message is used by the server to communicate to a Sender Client that the Receiver to which (s)he wants to talk with is not available anymore. In response to the "Bad Response", the Sender Client sends an "Acknowledgment Message" to the server (where the byte represents the type of message, namely "Acknowledgment Message"):

9 Key Establishment



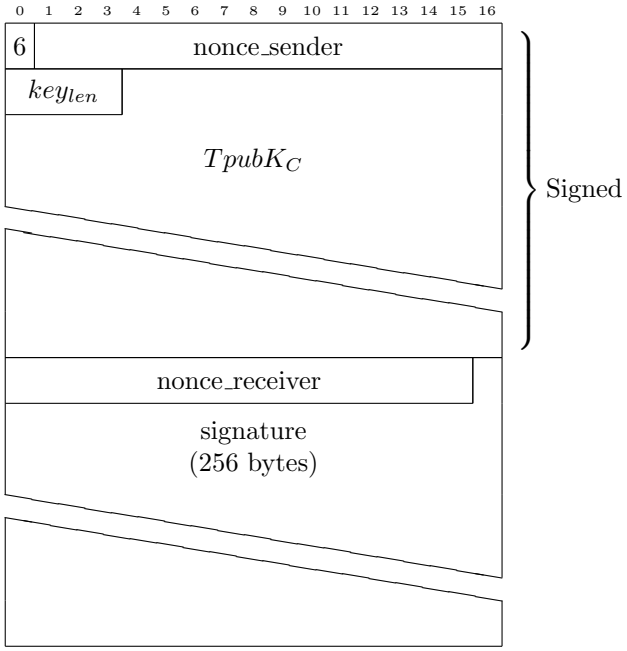
9.1 M1



Fields:

- **6**: This is the type of the message, corresponding to "Key Establishment Message".
- **nonce_sender**: This is the nonce generated randomly.

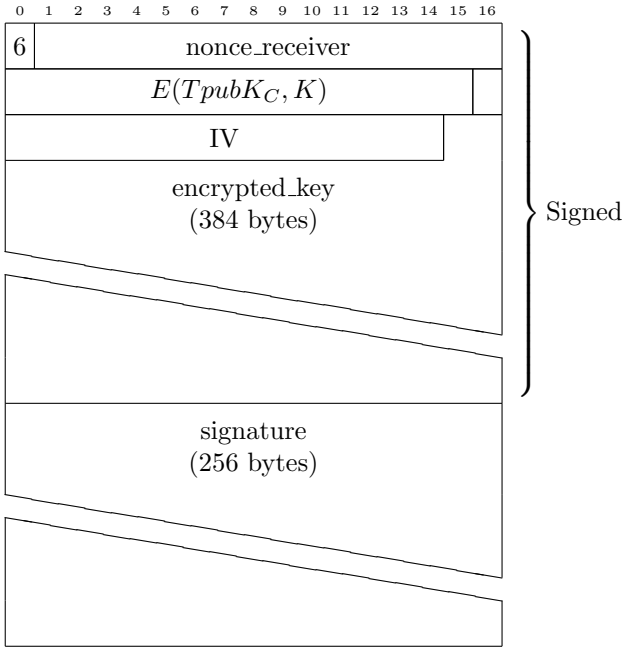
9.2 M2



Fields:

- 6: This is the type of the message, corresponding to "Key Establishment Message".
- **nonce_sender**: This is the nonce generated by the Sender User.
- key_{len}: It indicates the length of the ephemeral public key generated by the Receiver User.
- $TpubK_A$: This is the ephemeral public key generated by the Receiver User.
- **nonce_receiver**: This nonce is generated by the Receiver User and is used to verify the authenticity of the server.
- **signature**: The signature is computed on nonce_sender, key_{len} and $TpubK_A$ using the Receiver's private key. The Sender has the Receiver's public key to verify if the signature is correct.

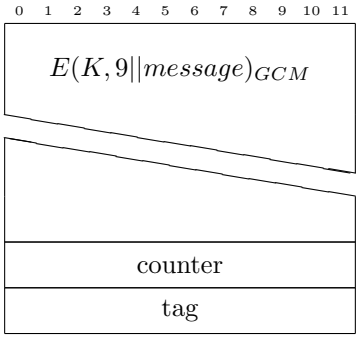
9.3 M3



Fields:

- **6**: This is the type of the message, corresponding to "Key Establishment Message".
- **nonce_receiver**: This is the nonce generated by the Receiver.
- $E(TpubK_C, K)$: K is the session key that will be used for the communication between the two clients.
- **IV**: This is the random IV generated for encrypting K with the asymmetric encryption using $TpubK_C$.
- **encrypted_key**: This is the encrypted key used for encrypting K with the asymmetric encryption using $TpubK_C$.
- **signature**: The signature is computed on the nonce_receiver, the encrypted symmetric session key, the IV and the encrypted key using the Sender's private key. The Receiver has the Sender's public key to verify if the signature is correct.

10 Chat



The two counters, one associated to the Sender User and one associated to the Receiver User and stored in both the local machines, are implemented and managed in the same way of the session among client and server.

Fields:

- **9**: This is the type of the message, corresponding to "Chat Message".
- **Message**: This is the message given as input by the user.
- **counter**: The counter will be used both as IV for encryption and also to guarantee the freshness and to avoid replay attacks from malicious users.
- **tag**: The tag is the AES_GCM signature, computed basing on the IV, passed to AES_GCM as AAD (Additional Authenticated Data).

11 Return to Lobby

The byte represents the type of message, namely "Return to Lobby".

After starting the chat between two clients, the "Return to Lobby" message is sent by one of the client to the server to stop the chat. When received, the server returns to the main thread for that client and communicates, using another "Return to Lobby" message, to return to the lobby to the other client.