UNIVERSITÁ DI PISA
ENGEGNERIA DELL'INFORMAZIONE

Online Messaging System Using End-to-End Encrypted and Authenticated Communication

Project Report

Bruno Augusto Casu Pereira de Sousa

Master's degree Computer Engineering – Computer Systems and Networks
880II Foundations of Cybersecurity

Pisa, July 2021

Contents

Introduction

The project aims the development of a message service using End-to-End Encryption. The communication is to be performed by a TCP connection between users logged to a server. After connected, client and server must implement a protocol to perform a key exchange, and by this ensure Perfect Forward Secrecy and Authentication. In addition, after the user is correctly logged in the server, he or she may request to chat with another user, performing a second key exchange protocol and establishing a unique Chat session key.

On each established protocol, users must be authenticated by its Public Key, already installed in the Server and provided to the clients when a Chat is established; and the Server must be authenticated by means of a certificate, provided by a Certification Authority.

The application developed was designated as *MessageApp*, and its development was based in the C library OpenSSL and its APIs for encryption, key creation and authentication, in addition to extra functions to run the service application. The source codes and a Client/Server environment example are found in the project github repository: <https://github.com/brunocasu/foc_project>

General Architecture and Client/Server TCP connection

The general architecture of the MessageApp system is defined in Figure 1. In this implementation multiple users can connect to a server using a TCP socket. With the set of APIS provided by the C libraries, the server program opens a TCP listening port, and bind to the Clients that request a connection. Each client will have an RSA key pair, and the server will store the users Public Key, as well as its own RSA key pair and Certificate, issued by the Certification Authority. With this initial configuration, the Users and the Server can perform an key exchange and authentication protocol (called handshake in the MessageApp context).

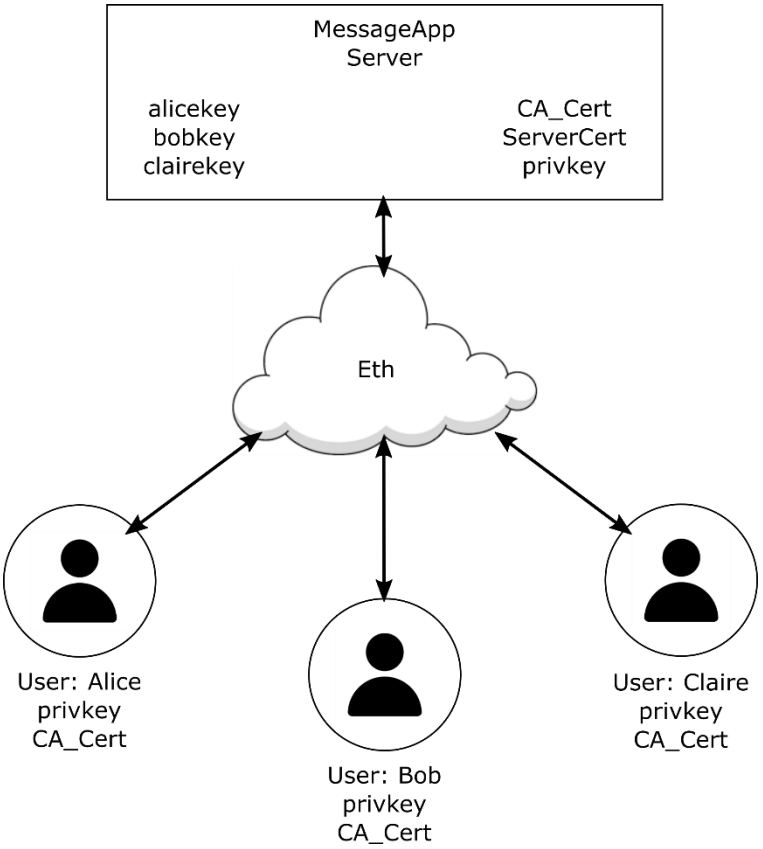

Figure 1 – Overview of the Message system connection

Noticeably, the all the messages exchanged between Client and Server will be transmitted in the Application layer of the TCP connection (Layer 7 of the OSI model). In order to establish the socket connection, the Client will bind to the server listening port, as the server will accept the connection, and keep the socket file descriptor.

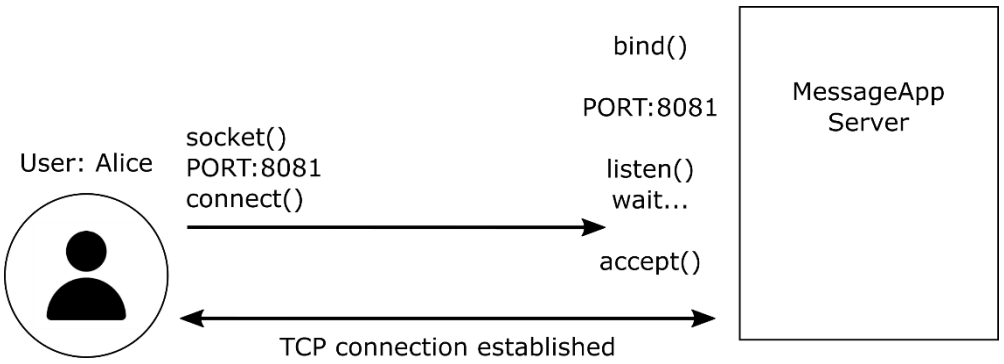

Figure 2 – Client/Server socket connection establishment

As the connection is established the Server will designate the User to a Communication Channel (thread running on the server). At the start of the connection the communication in the Channel is considered NOT secure, as the messages are still being transmitted without encryption in the socket Application Layer. To ensure the security of the Channel, the Client and Server then begin a Handshake protocol, described in the next section.

Client/Server Handshake Protocol and Secure Communication

For the Clinet and Server communication a symmetric key excahnge method must be implemented in order to encrypt all the messages that are sent in the communication Channel. This process must ensure Perfect Forward Secrecy, Authentication and a secure Encryption (safe agains replay attacks). The basic structure of this process is illustraed in Figure 3:

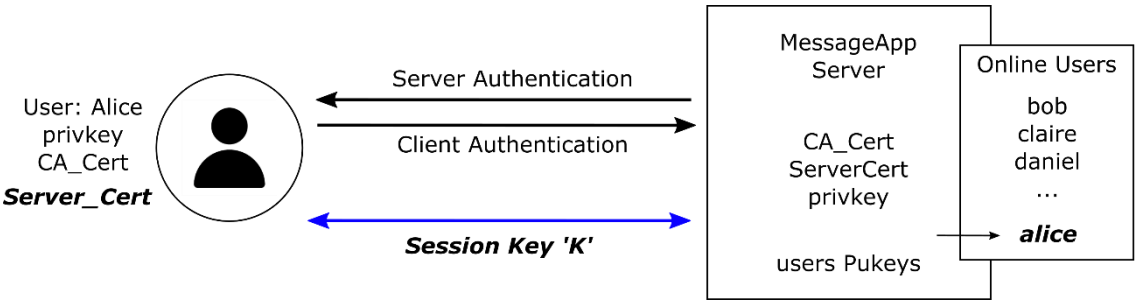

Figure 3 – Client/Server socket connection establishment

Initially, as the socket connection is established, the Client have the CA certificate and its private RSA key (this key is protected by a password), and the server have the public RSA key, as well as its own private key and certificate (issued by the CA). At the end of the transaction, it is expected that both Client and Server are authenticated, and that the User now posesses the Server Certificate, and that it is now Logged in the server.

For the Handshake protocol developed for the system the Ephemeral RSA key exchange method was used. In this mode, the Client will send an M1 message with a random nonce R1 (along with its username) to the server, wich will then create a Temporary RSA key pair, sending it back to the user in M2, as well as a second nonce R2, its signature and the Server certificate.

The sent signature will be done using the Server private key, and it will contain the initial nonce R1, the Temporary Public Key generated and the second nonce R2. By this, the User is then able to Authenticate the server (avoiding replay attacks), as well to ensure that the Temporary Public Key sent is fresh. This is authentication is executed using the Server Certificate sent by it (the certificate contains the Server Public Key).

After the Server authentication in M2, the User now generates a random key K (with 256 bits, or 32 bytes), and send it encrypted with the Temporary Publice Key generated. With the encrypted symmetric key, the User also sends the computed encrypted key with the nonce R2 signed by its private key. Upon receiving M3, the Server then Authenticates the User with its pre-installed Public Key. If Authentication is successful, the Server decrypts the session key K, and finishes the Handshake by sending a conirmation message to the User encrypted with the session key K.

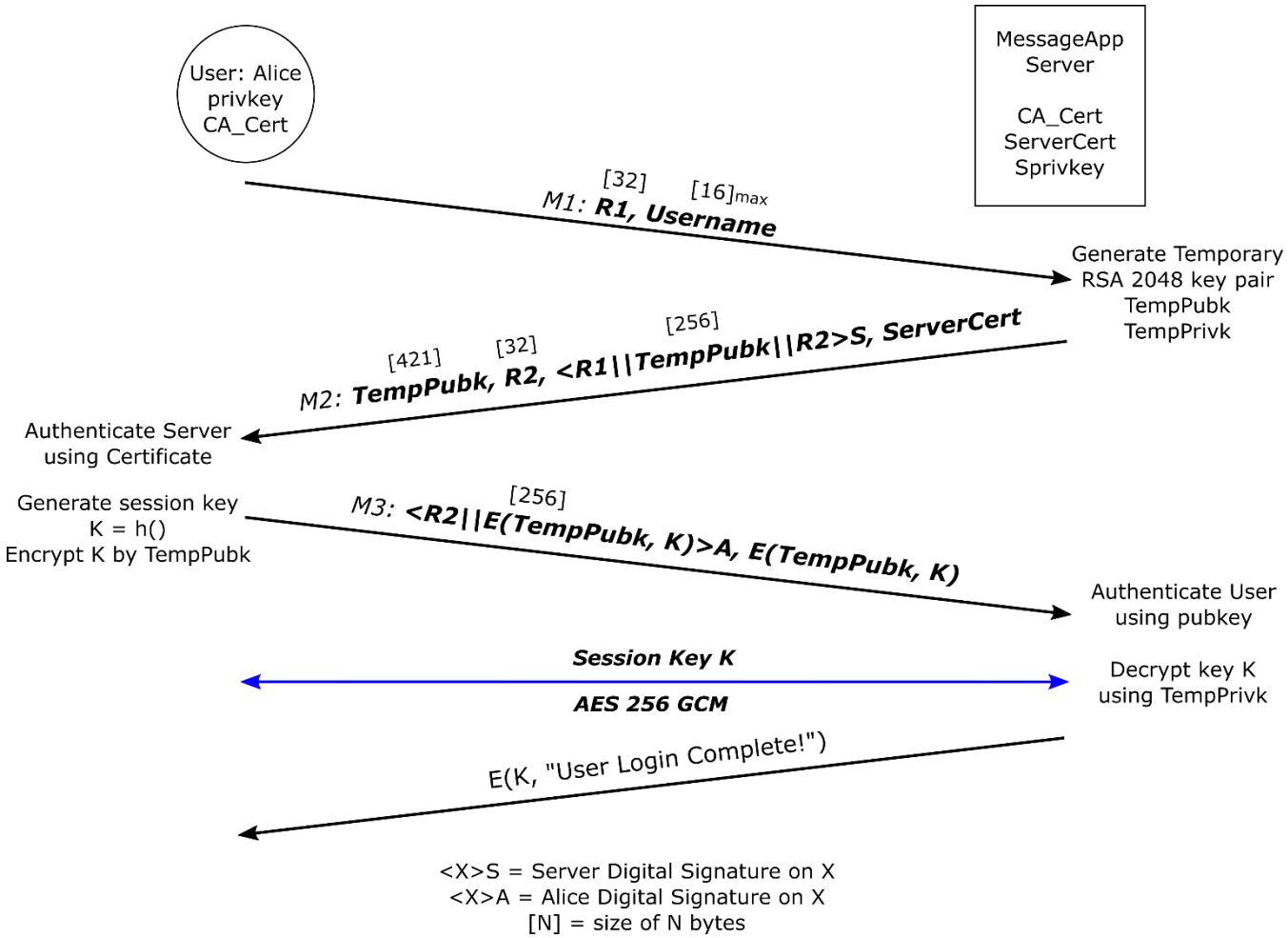

Figure 4 – MessageApp Handshake protocol (Ephemeral RSA key exchange)

As now both Server and Client share a Symmetric key K of size 32 bytes, the messages exchanged between them will be encrypted using AES 256 GCM. This method was chosen as it guaranties authentication in the messages and it is resistant to replay attacks.

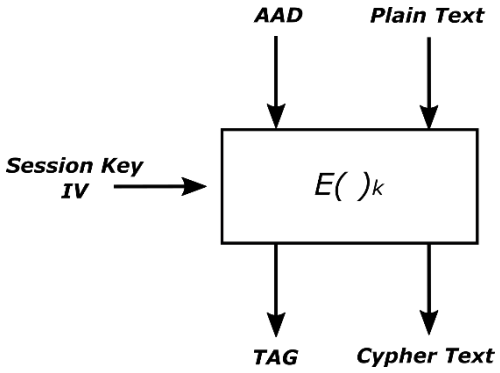

Figure 5 – AES 256 GCM Encryption schematic

The format of the messgaes exchanged are represented in Figure 6. For the AAD value, it was choosen to add the username of the connected client, as well as a 16 bytes counter to prevent the replay attacks (the incremented value of the counter is also used for the IV, but only using the first 12 bytes of the counter). The plain ADD sent is then added with the message and signed, returning the TAG. In this way a high level of security is achieved, as even a 256 bit key was used for the encryption.

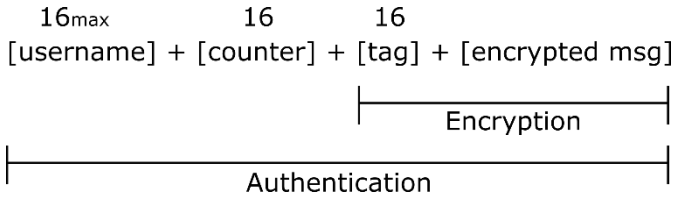

Figure 6 – AES 256 GCM message format for the Session Encryption

As an example, Figure 6 shows the overall formating of a message being sent from Alice to the Server, using the AES encryption and the session key. From this initial communication, the user can send specific commands to the server, in order to use the MessageApp services provided.

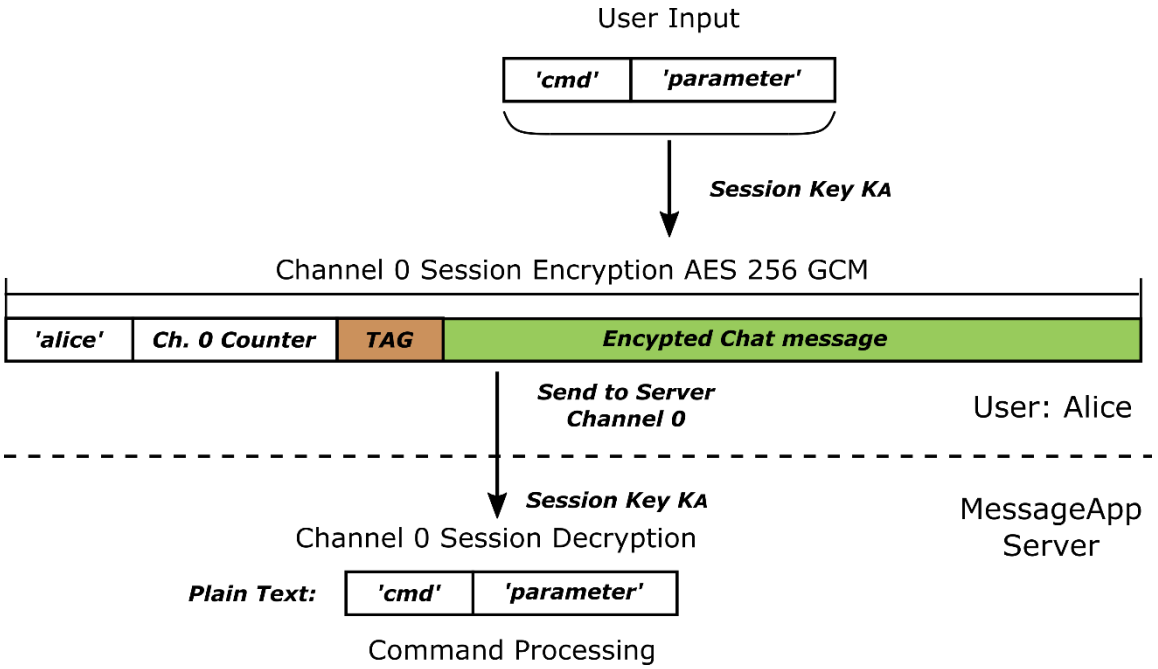

Figure 7 – Message exchange from User to Server using the AES GCM encryption

MessageApp Commands

After the establishment of the Session key between client and server, the User now can send commands to interact with the server. All commands are defined by a four-letter string, and for commands that do not require a parameter entry, the letter x can be added to the message.

The Chat parties, in the application context, are defined as the Caller and the Receiver, and depending on which sate of the communication, the server will allow or not the issuing of specific commands to it. That is done in the server by associating a state for each user logged in a communication Channel. The available commands to send (User to Server) are:

- ‘chat’’receiver_username’
- ‘acpt’’caller_username’
- ‘refu’’caller_username’
- ‘listx’
- ‘exitx’

The first command, defined with the string ‘chat’ is used to send to an online user a chat request. The string must be followed by the username of the person that is desired to begin the Chat. As the Caller sends the ‘chat’ command to the server, the Server will send the request to the Receiver Channel. If the parameter does not contain a valid username, or the requested user it is not logged, the Server will reply with an error message.

In the Receiver state (when a request is received by a user), the user then may accept the chat request by sending the command ‘acpt’ to the server, followed by the caller username, or refuse the chat by sending the ‘refu’ command, also followed by the caller username. As the Receiver accepts the Chat, the Server then sends both users the other party Public Key, completing the Chat establishment.

The server then binds the users Channels. This means that now on every message that is sent from User A to User B (or the other way around) with the command ‘frwd’ is redirected to the linked Channel. However, a few considerations must be made. The messages that are sent in this step are decrypted by the server, meaning that an additional step of encryption must be performed before the users can send its Chat messages. Also, the Server only process the ‘frwd’ command after the Chat is accepted, otherwise the commands are ignored.

Other features are also important for operating that application and they are the possibility for the client to retrieve a list of online users and to log off the Server. To perform these actions, respectively, the user then must send the command ‘listx’ or send the command ‘exitx’. The proper execution of the commands is better described in the last section of the repot.

In Figure 8 a representation of the messages exchanged when Alice invites Bob to Chat using the MessageApp:

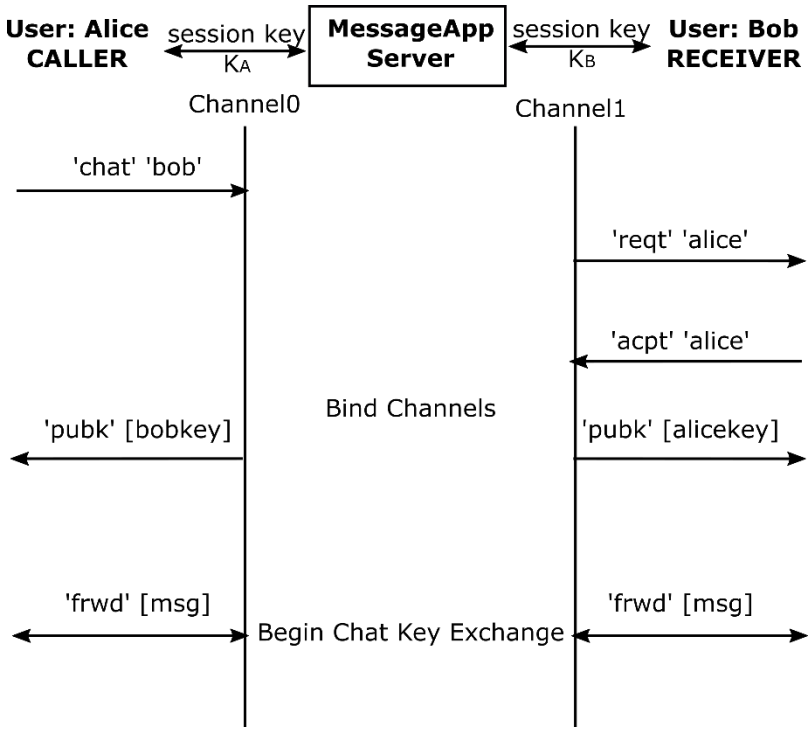

Figure 8 – MessageApp Chat request and accept command processing

In addition to the previously listed commands and operations, the application will handle the messages using other identifiers, they are:

- ‘reqt’’caller_username’
- ‘pubk’’public key txt’
- ‘frwd’’msg’

The forward command, as previously mentioned, will then allow the Server to identify the messages to be forward during the Chat. The ‘pubk’ command is the identifier to the clients that the Chat was accepted, and the parameters contains the Public Key of the users in text format. The request, however, is a command that only the users identify, and it I used to indicate that a request was sent to this username.

In Figure 9, and example of messages exchanged represents the situation when Alice calls Bob and he refuses the connection. After the command processing and exchange the users log off the server.

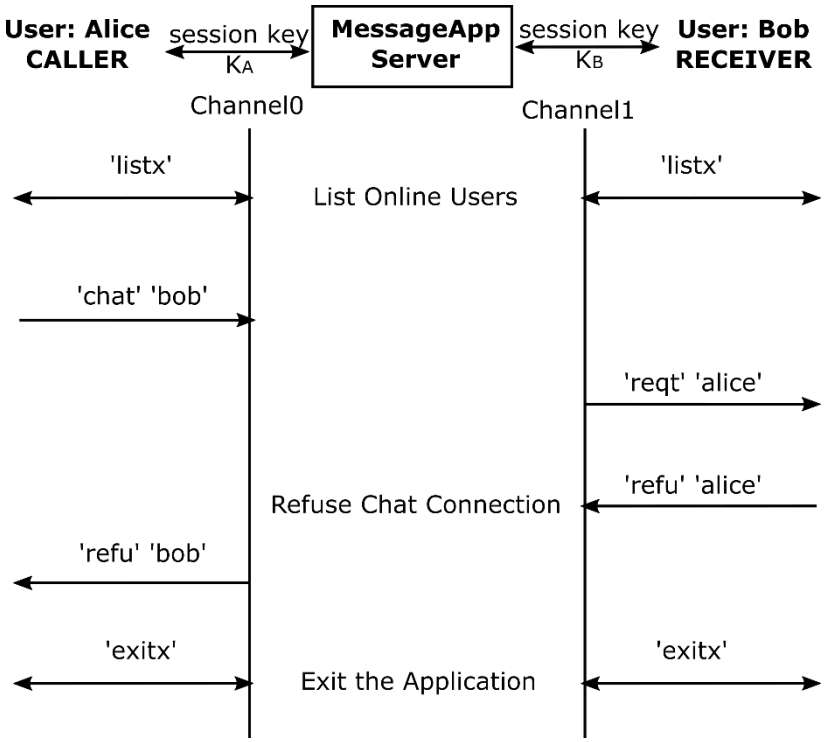

Figure 9 – MessageApp Chat request and refuse response command processing

Chat Key Exchange and Secure Communication

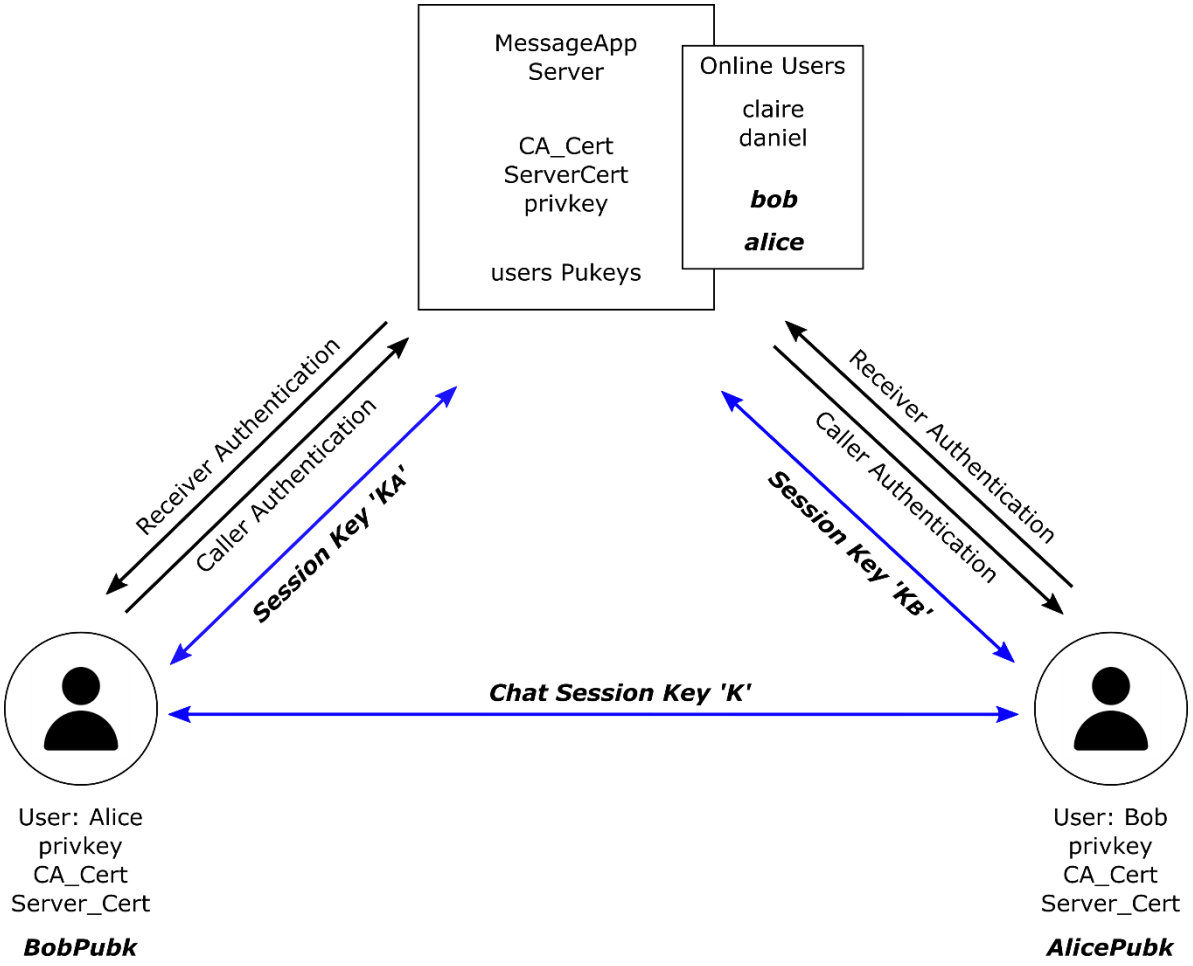

Figure 10 – Chat establishment schematic (scenario after Server distribute the Pubkeys)

After the Client/Server Authentication process, and after the proper Chat commands are performed between the users (Chat is established), the communication between Caller and Receiver must then be encrypted with a Chat session key, which the server does not have access to. From the previous server messages, both users now posses the Public Key of each other, and communicate with the server using its respective sessions keys. Figure 10 illustrate the scenario after the Chat is established between the parties.

In this scenario, a second Key exchange protocol was used to provide the Chat a particular symmetric session key, and to ensure perfect forward secrecy and authentication. For this process a second Ephemeral RSA key exchange protocol was implemented, in order to establish a 32-byte key for the AES 256 GCM encryption.

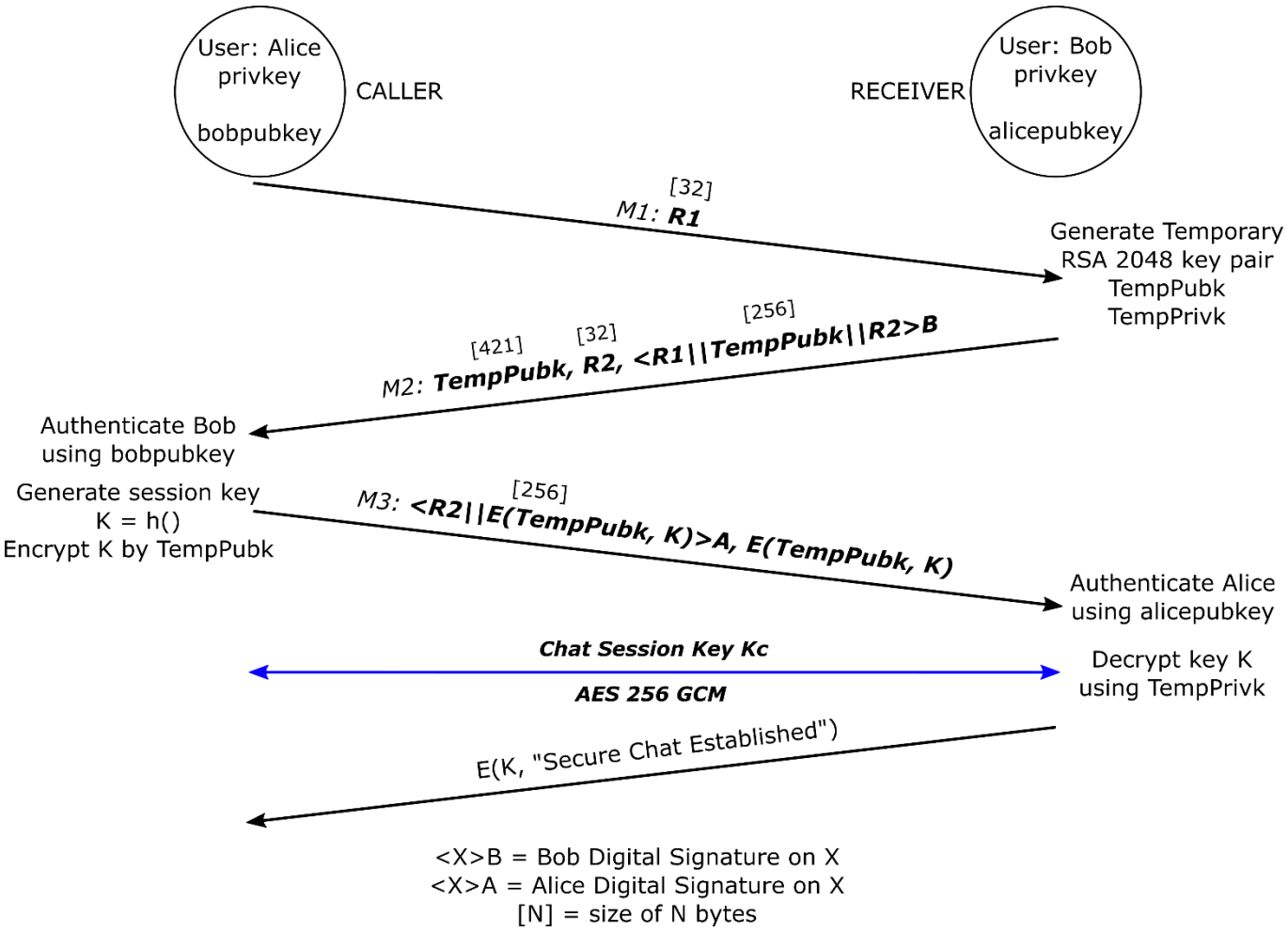

Figure 11 – Ephemeral RSA key exchange for Chat session key establishment

The protocol for the key exchange between the users in the Chat is like the one used for the Client/Server key exchange. However, in this case, the users are authenticated by its public key, and not by a certificate. Finally, after the Chat session key is established, the users will then encrypt their Chat messages using another layer of AES 256 GCM encryption, ensuring End-to-End encryption.

Noticeably, the messages that are exchange in this last step are not visible to the Server, as the decryption performed only retrieves the MessageApp command, and cannot decrypt the plaintext sent by the users. Figure 12 illustrates the messages seen by an outsider (in Figure 11, the messages are shown as if someone is looking into the server and can't decrypt any of the exchanged messages).

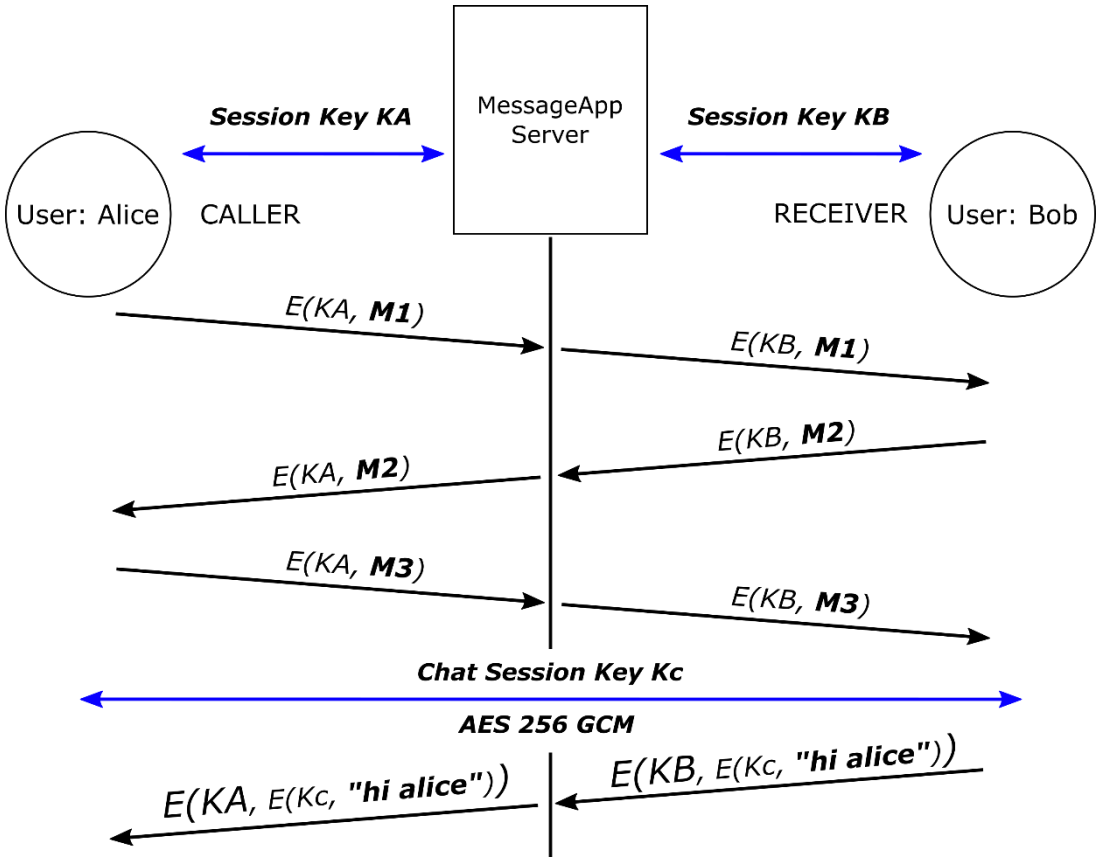

Figure 12 – Chat key negotiation with Server encryption

As the symmetric 32-byte key is shared between the user in Chat, the format of the encrypted messages is described in Figure 13. Again, it was used AES 256 GCM for authenticated encryption.

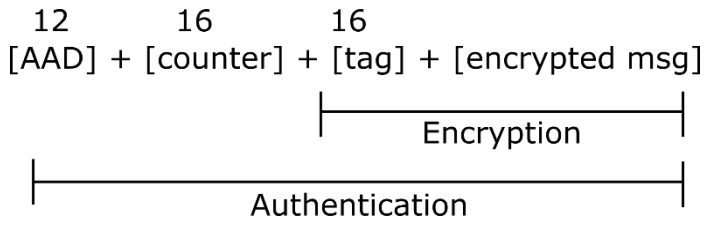

Figure 13 – Chat AES 256 GCM message format

As a final example of the message contests sent in the application context, a text transmission from Alice to Bob is represented in Figure 14. All the encryption layers are represented, and it is noticeable that despite the Server receiving and decrypting the messages between the clients, the second layer of encryption ensures the security of the text sent by the users.

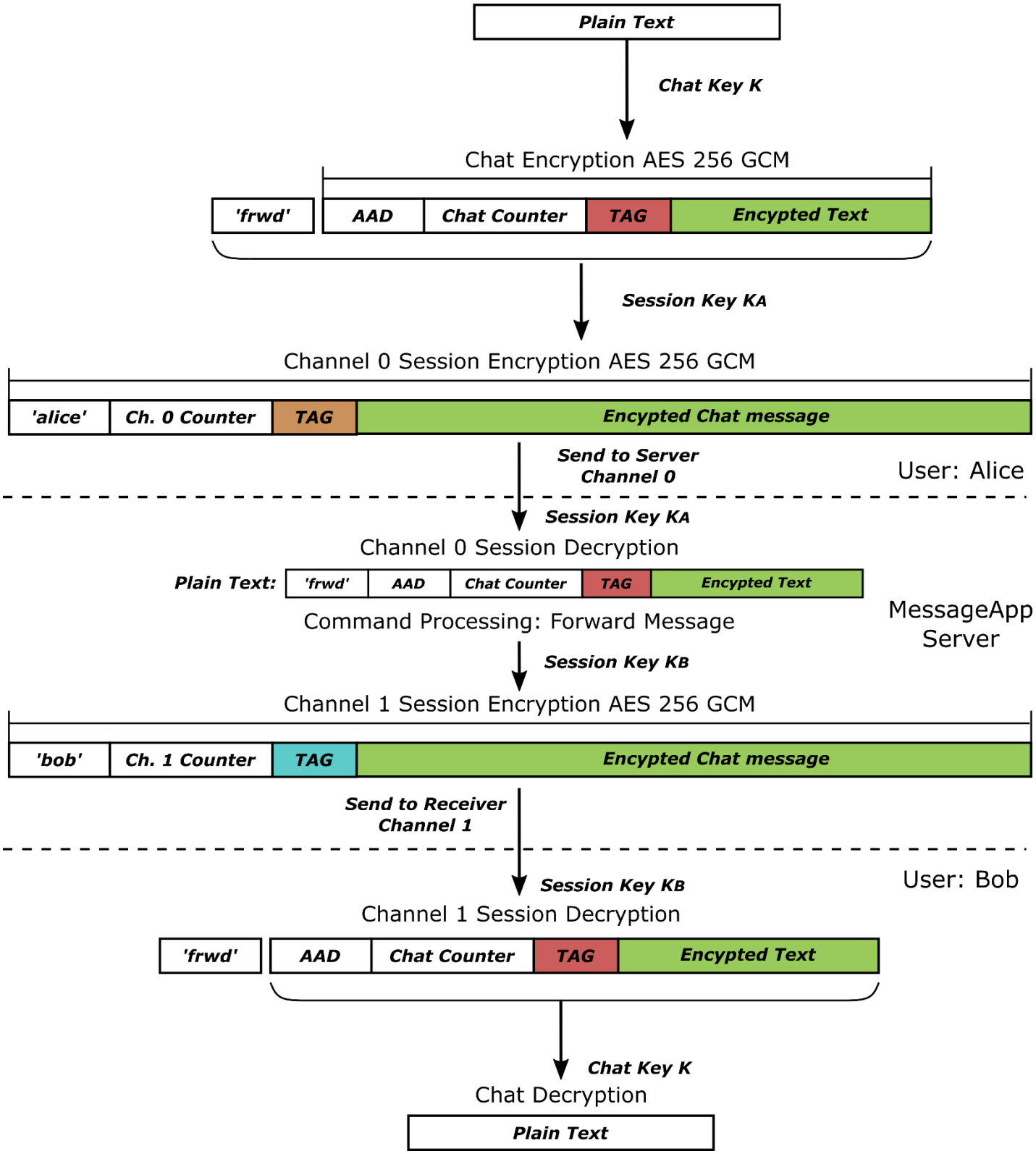

Figure 14 – Format of the messages when sending a plain text to the friend in Chat

Using the MessageApp

To use the developed application, the user must have its public key installed in the server folder. To execute the MessageApp Server run the command at the terminal (bind in PORT 8081):

```
./server 8081
```

After that, multiple clients can connect to the Server listening Port. The following command is then used to launch the client program and bind to the server TCP listening PORT.

```
./client 127.0.0.1 8081
```

After the execution of the client program, the user then must submit its username (this parameter is also configured in the Server database, and it is linked to the user Public Key in the server). The username can't contain any special characters and must not exceed 16 characters in length. After the username, the program will request the PEM password used to protect the private key. With these parameters inserted the program will then execute the handshake with the server, receiving the peer certificate. The information of the Certificate will be displayed as well as the temporary public key used in the transaction, and finally for didactic purposes the Session key will also be displayed.

After Client/Server authentication, the user is now logged in the server database, and can interact with the server. As previously mentioned, some commands can be executed. Use the following command to request the list of online users:

```
[SERVER-COMMAND]->listx
```

To chat with a friend in the server it is necessary to type the following command (assuming that you would like to chat with the user 'bob'):

```
[SERVER-COMMAND]->chatbob
```

For general purpose, replace the character 'bob' with the desired username that you wish to chat with. From the Receiver side, in the example where *alice* is calling *bob*, you may accept or not the connection. The server will send the request with the identification of the Caller, and if its is desired to start the chat type the following command:

```
[SERVER-COMMAND]->acptalice
```

Again, in the example the Caller is *alice,* and the receiver is *bob*. For general purposes, replace the character 'alice' for the caller username (if the caller username is misspelled, the chat cannot be completed). If it is not desired to chat with the caller, type the following command in this case:

```
[SERVER-COMMAND]->refualice
```

As the chat session is started, the system will automatically perform the Chat key exchange, and will return now a different interface in the screen, indicating that the messages typed now are to be sent to the connect user. When is desired to end the communication type in the chat interface:

```
[CHAT]->/exit
```

If it's desired to exit the program inside the Server command (not in the Chat) type in the command:

```
[SERVER-COMMAND]->exitx
```