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 a key exchange and authentication protocol (called handshake in the MessageApp context).

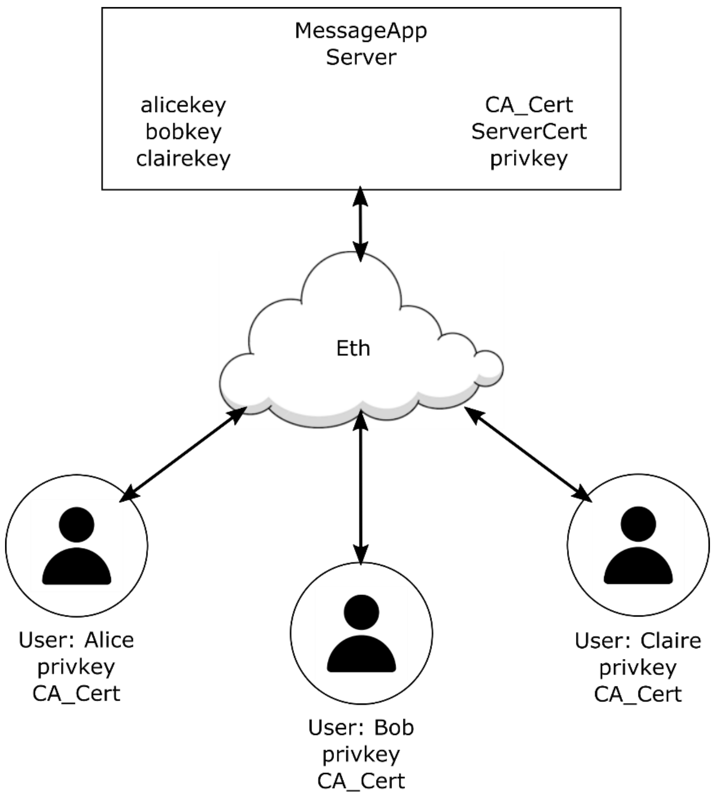


Figure 1 – Overview of the Message system connection

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). 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.

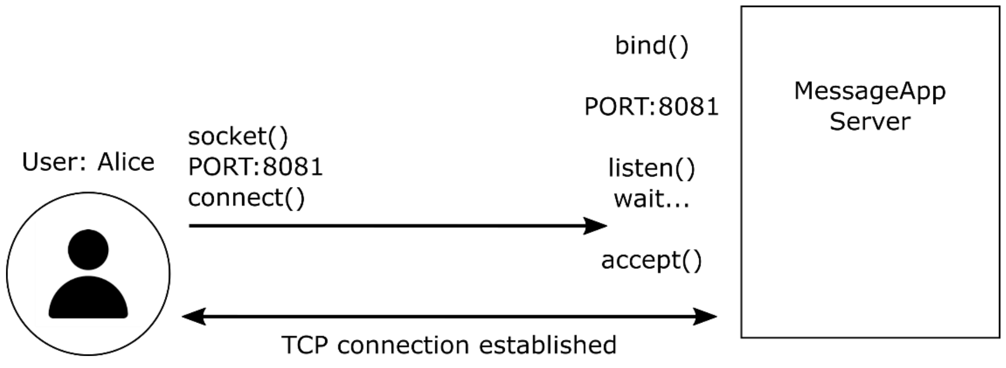


Figure 2 – Client/Server socket connection establishment

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.

After the authentication process provided by the Handshake implemented, the Server will provide the user access to the server application, allowing the now logged in client to call other users in the platform. When a User logoff the Server, the communication Channel is then freed. This implementation allows a limit number of simultaneous users online, defining the capacity of the server as the amount of channel threads that the machine can manage. Is also important to mention that even if all channels are occupied, the server will allow users to connect; however, in this scenario, the user will be sent an error message informing that he or she cannot connect the MessageApp service (no channels available). After the error message sent, the Server then closes the TCP connection.

In addition, another major step performed at the beginning of the Handshake is to check if the provided credentials have already been used by someone else (a previous connection using the same username provided has already been established). This method the avoids registered users connecting using other client's username, and it can even provide a method of detection for a hacked client account (the real user will notice the error when trying to login). For further improvements on the server security of the Server and availability, a basic block-list software program can be implemented, that is, a separate thread that blocks (add to the block list) specific IP addresses that excessively try to access the system and fails to authenticate.

In Figure 3 a schematic of the organization of the client's channel occupancy is shown. Channels are taken as the Server finds a free channel available and designates that user the channel number and the File Descriptor of the socket to be used.

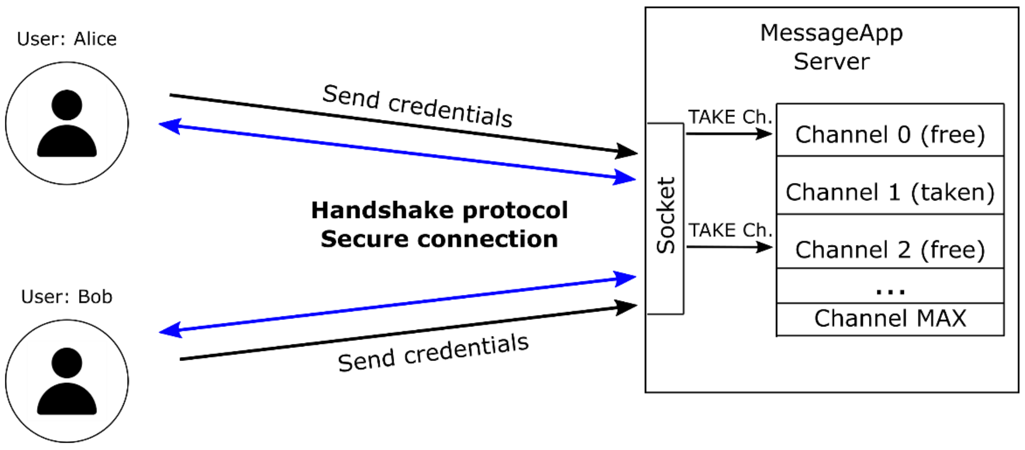


Figure 3 – Server Channel allocation

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 4:

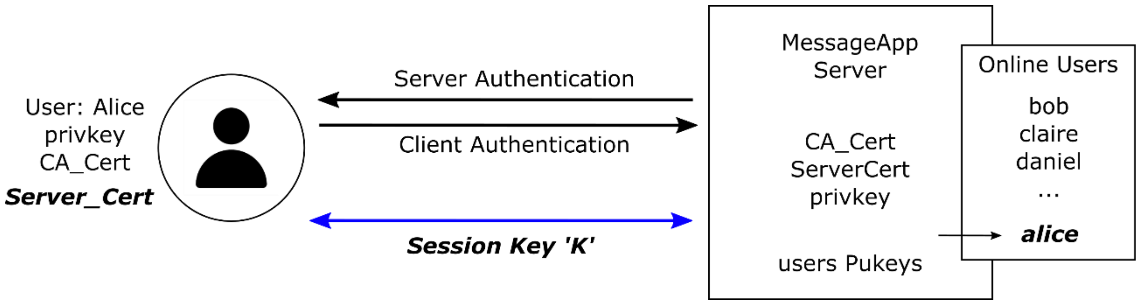


Figure 4 – Client/Server socket connection establishment

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), 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.

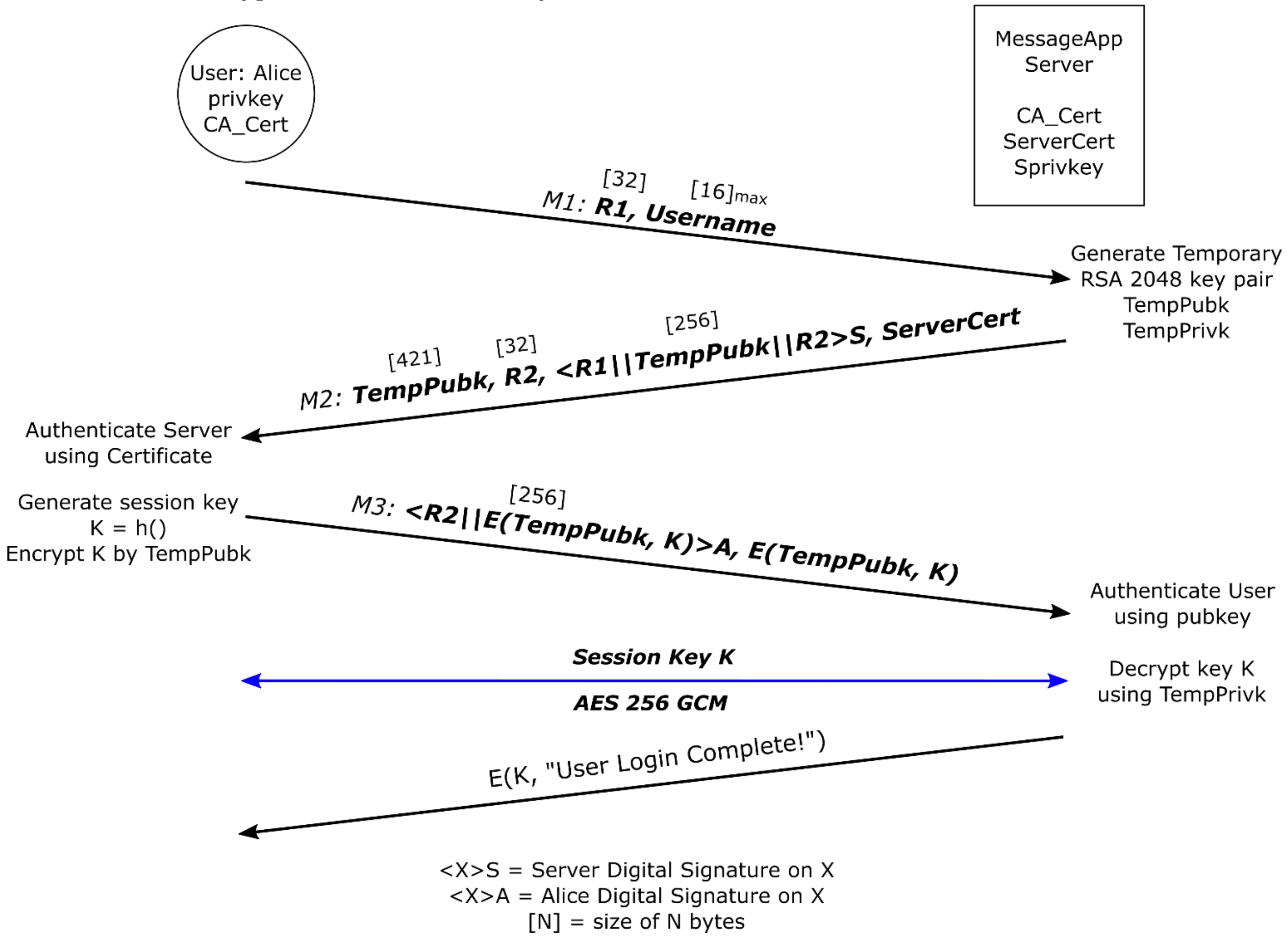

User: Alice
privkey
CA_Cert
MessageApp Server
CA_Cert
ServerCert
Sprivkey
[32]
[16]max
M1: R1, Username
Generate Temporary RSA 2048 key pair
TempPubk
TempPrivk
[421]
[32]
[256]
M2: TempPubk, R2, <R1||TempPubk||R2>S, ServerCert
Authenticate Server using Certificate
Generate session key
K = h()
Encrypt K by TempPubk
[256]
M3: <R2||E(TempPubk, K)>A, E(TempPubk, K)
Authenticate User using pubkey
Session Key K
AES 256 GCM
Decrypt key K using TempPrivk
E(K, "User Login Complete!")
<X>S = Server Digital Signature on X
<X>A = Alice Digital Signature on X
[N] = size of N bytes

Figure 5 – MessageApp Handshake protocol (Ephemeral RSA key exchange)

After the Handshake protocol, Server and Client now share a Symmetric key K of size 32 bytes, and 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.

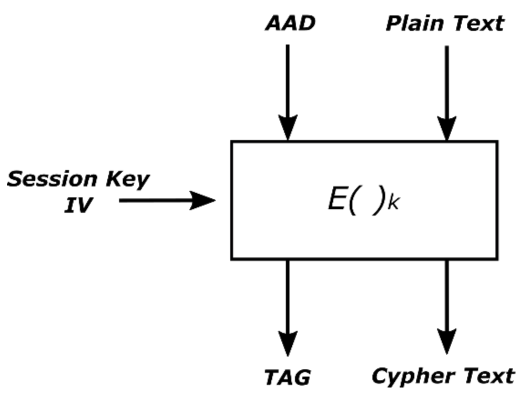


Figure 6 – AES 256 GCM Encryption schematic

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 AAD is also used for the IV, but only the first 12 bytes of the counter). The plain ADD sent is then added with the message and signed, returning the TAG.

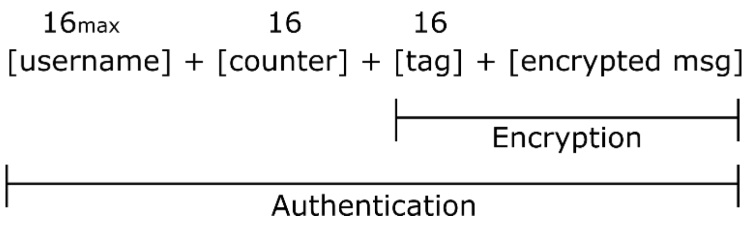


Figure 7 – AES 256 GCM message format for the Session Encryption

As an example, Figure 8 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.

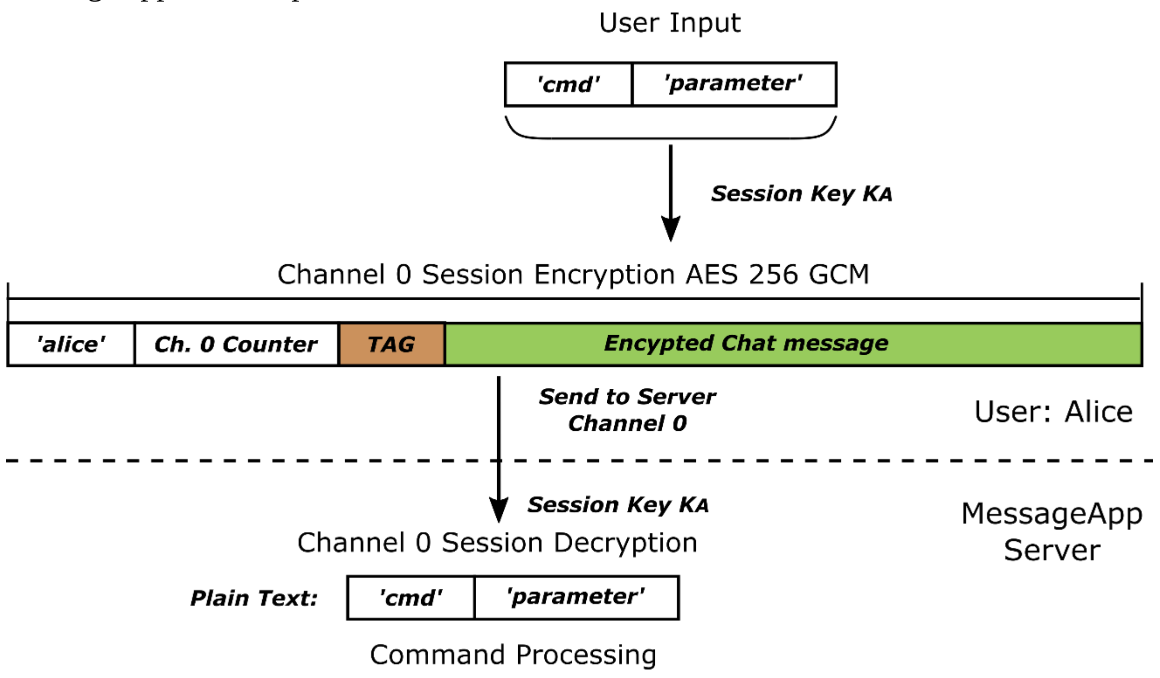


Figure 8 – Message exchange from User to Server using the AES 256 GCM encryption

An important mention to the implemented AES mechanism is the counter handling between Server and Client. For the messages exchanged two counters are used to maintain consistency in the communication and avoid authentic messages being discarded. The proper implementation then uses a Server to Client counter and a Client to Server counter. Each of those is only incremented as either the Server sends a message, or the Client sends a message, avoiding that both peers increment simultaneously its internal counters trying to send a message. Figure 9 illustrate an example of how the two counters are handled is shown, as well as a simultaneous message generation. In the example, the user sends a request to retrieve the online user list (command 'list'), which is async, and the server sends a request to chat from a different user (command 'reqt'), which is also an async message.

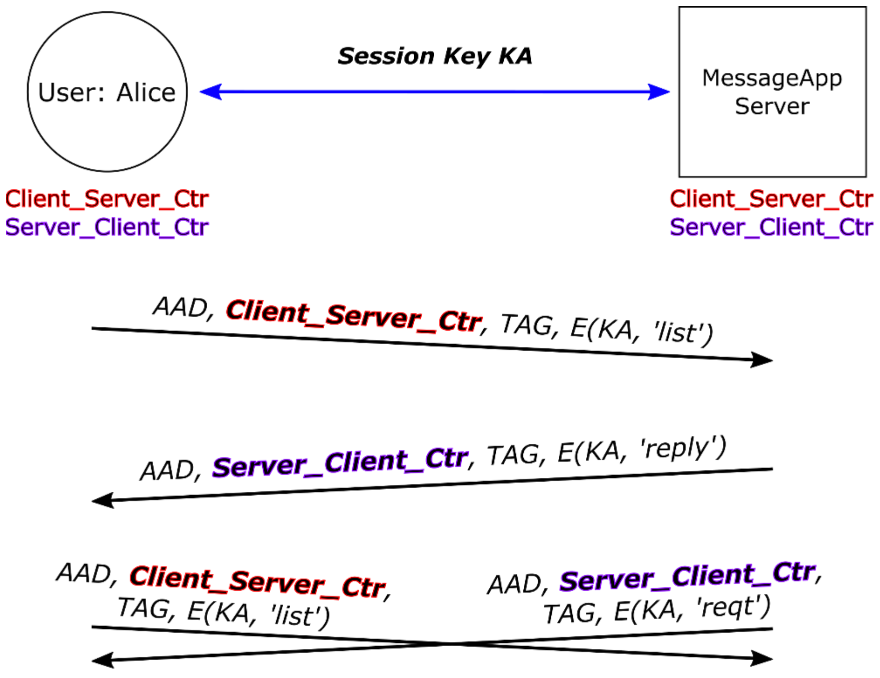

Session Key KA
User: Alice
MessageApp Server
Client_Server_Ctr
Server_Client_Ctr
Client_Server_Ctr
Server_Client_Ctr
AAD, Client_Server_Ctr, TAG, E(KA, 'list')
AAD, Server_Client_Ctr, TAG, E(KA, 'reply')
AAD, Client_Server_Ctr, TAG, E(KA, 'list')
AAD, Server_Client_Ctr, TAG, E(KA, 'reqt')

Figure 9 – Message counter handling in Client/Server communication

From the image, it is possible to observe the behavior of the system when both peers set message at the same time. With this implementation, both Client and Server will have sent different counters in the message, and the other peer will check and increment independent counter, maintaining the consistency of the messages even in this scenario.

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 command interface guidelines are meticulously detailed in the section 'Using MessageApp'. The next session is a comprehensive description on how the messages are managed by the server and does not intent to instruct the user on how to use the application.

In the Message app implementation, the Chat parties, 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’
- ‘refu’
- ‘cach’
- ‘list’
- ‘exit’
- ‘help’

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 try to find the referred username in the logged in users list. If the username sent contains unsafe characters or a username that is not currently online, an error message will be sent to the caller. It is also not possible to request a chat to itself.

In the case of a valid online user, the server then proceeds to generate a request message, and send it t the Receiver, as well as a confirmation message to the Caller that the chat request was submitted. In addition to the chat command, the caller can also choose to cancel the call before the receiver accepting it; this is done by sending the command ‘cach’.

In the Receiver state (when a user receives a request), the user then may accept the chat request by sending the command ‘acpt’ or refuse the chat by sending the ‘refu’ command. As the Receiver accepts the Chat, the Server then sends both users the other party Public Key, completing the Chat establishment. Noticeably, the Receiver can try to issue other commands to the Server, however, the application will try to prevent other commands to be received until the request has been resolved.

After the pubkey distribution, the server then binds the users Channels, which will begin its own key exchange protocol. As this step is completed, the server then expects to receive the messages from the users with the ‘frwd’ identifier, allowing the messages to be forwarded to the Chat users.

Besides the chat commands, the application also has additional features that are important for the correct operation of the platform, they are: the possibility for a client to retrieve a list of online users, to log off from the Server and to request the list of commands. To perform these actions, respectively, the user then must send the commands: ‘list’, ‘exitx’ or ‘help’. In Figure 10 a representation of the messages exchanged when Alice invites Bob to Chat using the MessageApp:

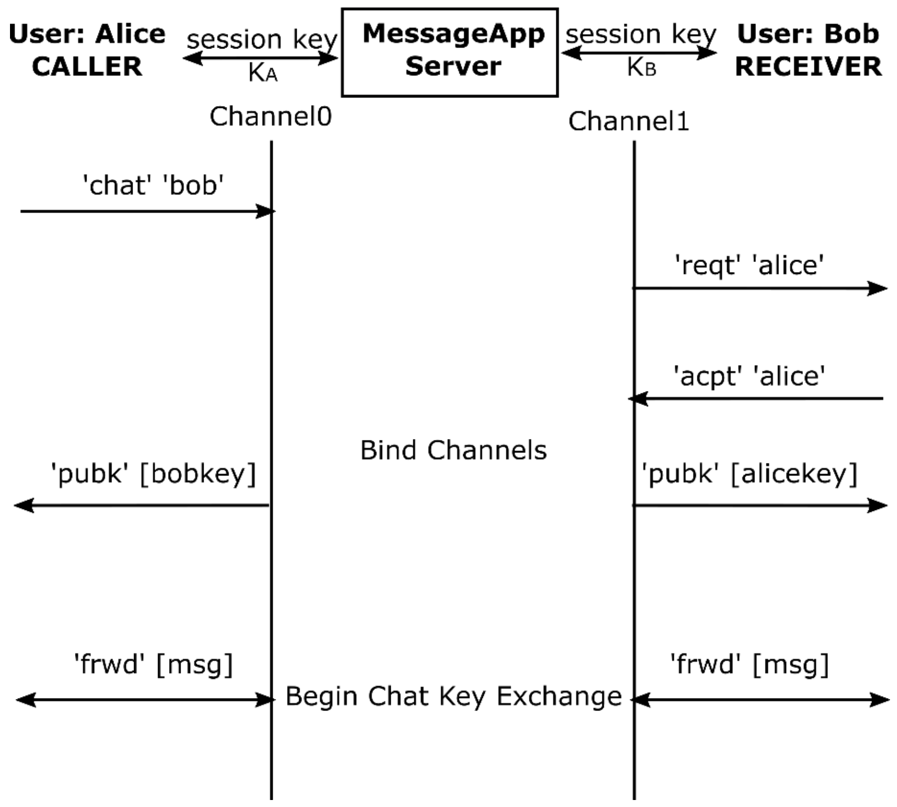


Figure 10 – MessageApp Chat request and accept command processing

Aside the Chat establishment, the server must also provide the handling of the disconnection process. This must then guaranty that no user will be left in an incompletion state, which is if the caller or the receiver abruptly disconnects during a request, not providing a response to the server. In the available sate (not in chat neither pending to respond a request or receive a confirmation) the user can execute the ‘exit’ command, closing the application.

However, the Server must also detect when an abrupt disconnection happens and inform the other user that the Chat has been disabled (if in the process of establishing a chat or even when the Chat is already up). In this case the Server must release the other channel from any pendency and send an error message to the user, as well as free the disconnected channel.

During chat, a similar process is done, as any user can execute a graceful disconnection by typing a specific string in the chat interface. In this scenario, the Server informs the disconnection to the other peer and remove all pendency returning both users to the available state.

To provide an example of how the Disconnections are handled, in Figure 11 the messages are demonstrated in the case of any disconnection during the request process.

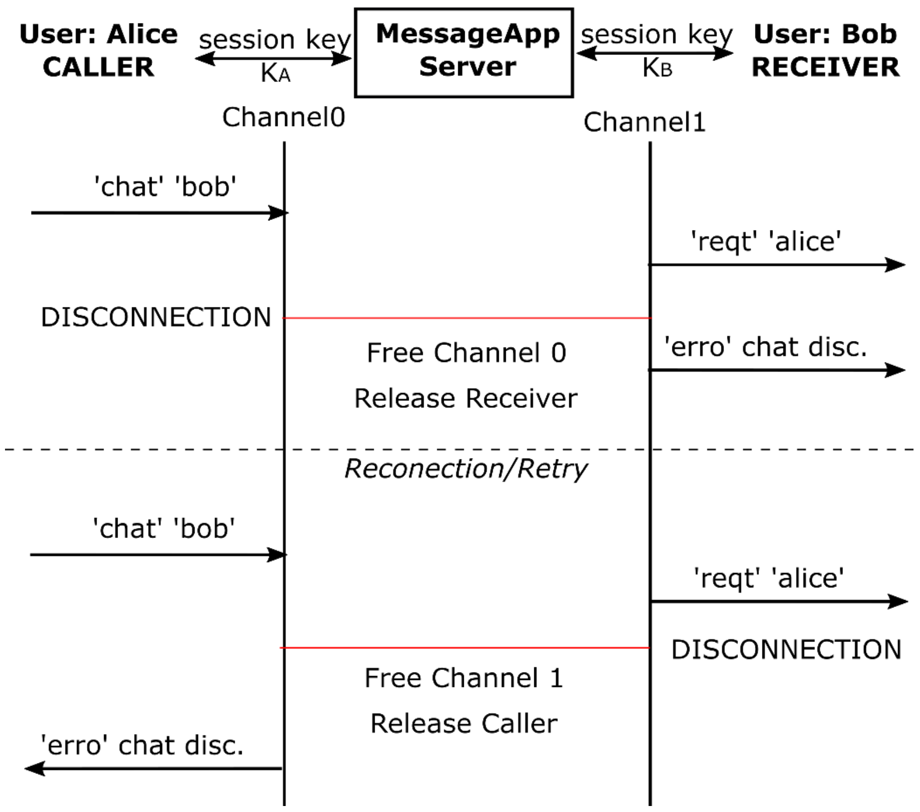


Figure 11 – MessageApp Chat disconnection message handling

The other message codes that are used by the server and the client are implied commands and do not require user direct input, they are:

- ‘reqt’
- ‘exch’
- ‘pubk’’public key txt’
- ‘frwd’’msg’
- ‘erro’

The forward command, as previously mentioned, will then allow the Server to identify the messages to be forward during the Chat. The ‘exch’ is the graceful disconnection during 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 ‘reqt’, however, is a command that only the users identify, and it I used to indicate that a request was sent to this username, and to identify the confirmation from the server that the request has been sent to the Receiver. Finally, any error messages are sent to the users or server with the ‘erro’ string identifier.

In Figure 12, 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.

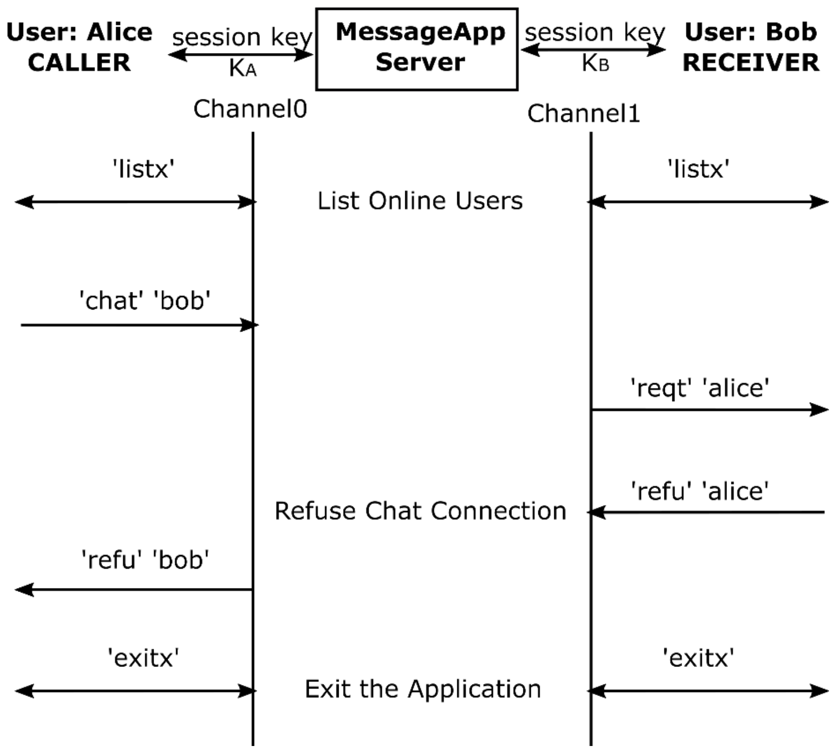


Figure 12 – MessageApp Chat request and refuse response command processing

Chat Key Exchange and Secure Communication

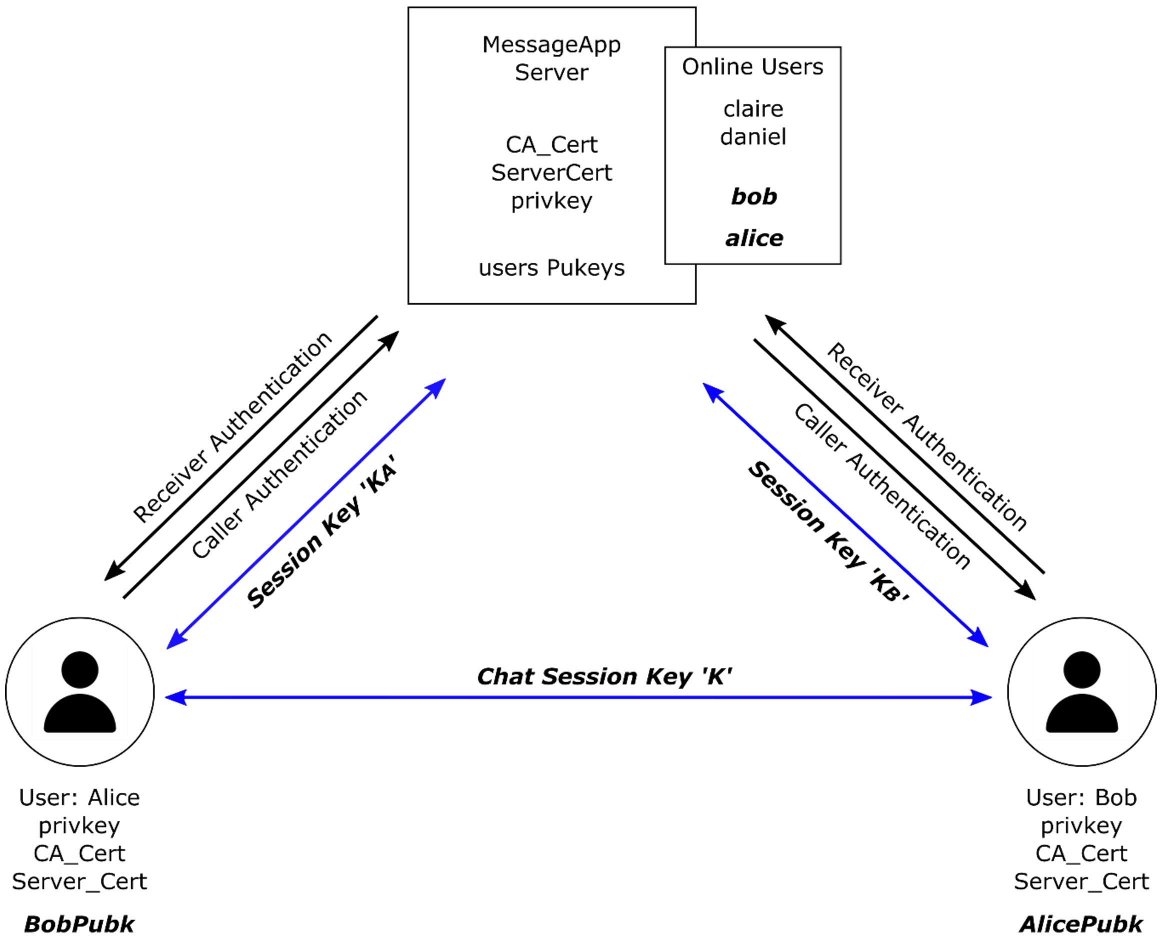


Figure 13 – Chat establishment schematic (scenario after Server distribute the Pubkeys)

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 possess 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.

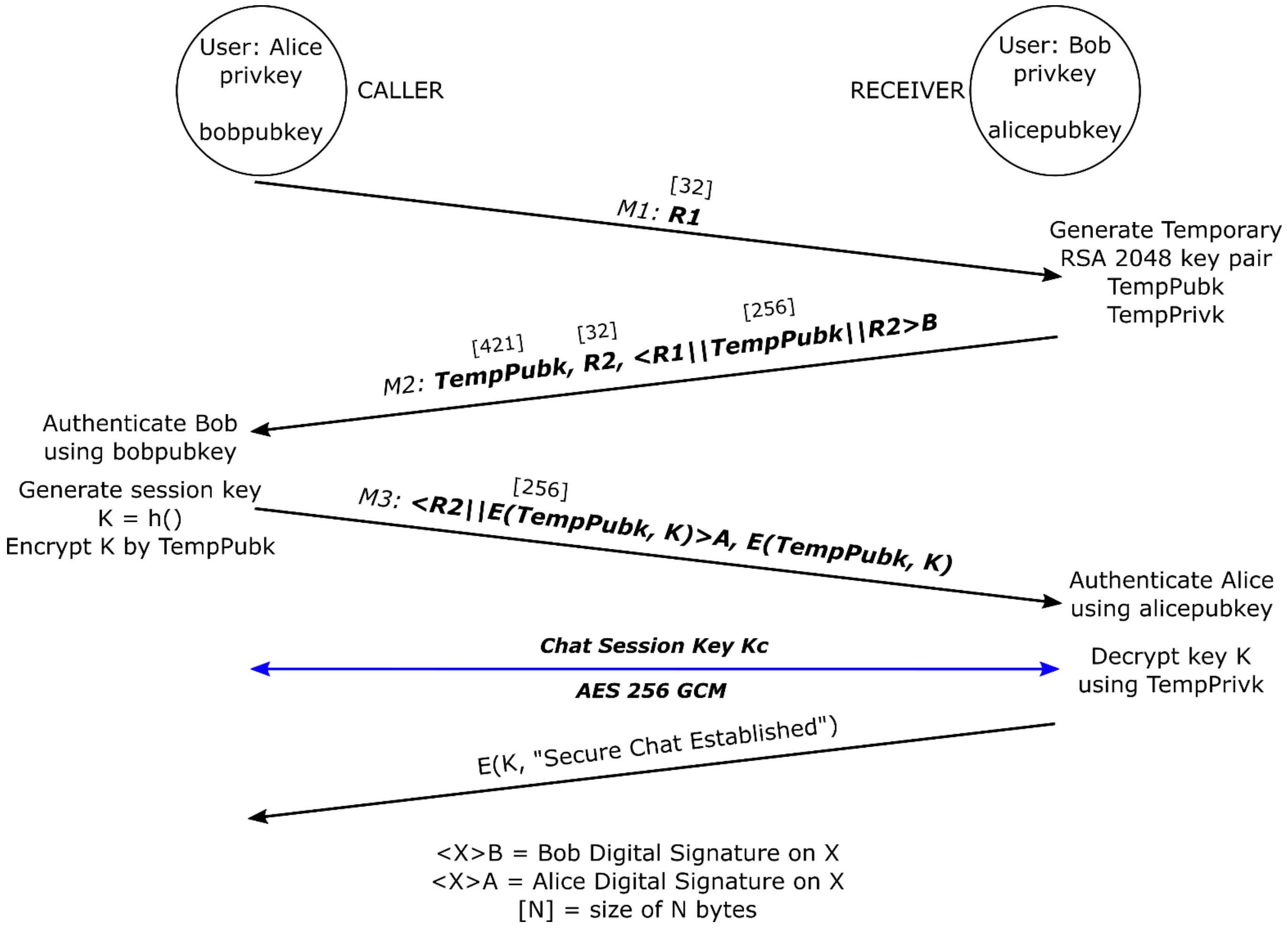

User: Alice
privkey
bobpubkey
CALLER
RECEIVER
User: Bob
privkey
alicepubkey
[32]
M1: R1
Generate Temporary RSA 2048 key pair
TempPubk
TempPrivk
[421] [32] [256]
M2: TempPubk, R2, <R1||TempPubk||R2>B
Authenticate Bob using bobpubkey
Generate session key
K = h()
Encrypt K by TempPubk
[256]
M3: <R2||E(TempPubk, K)>A, E(TempPubk, K)
Authenticate Alice using alicepubkey
Chat Session Key Kc
AES 256 GCM
Decrypt key K using TempPrivk
E(K, "Secure Chat Established")
<X>B = Bob Digital Signature on X
<X>A = Alice Digital Signature on X
[N] = size of N bytes

Figure 14 – 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.

The messages that are exchange in this last step, and further Chat text messages, 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 15 illustrates the messages seen by an outsider (in Figure 14, the messages are shown as if someone is looking into the server decrypted messages).

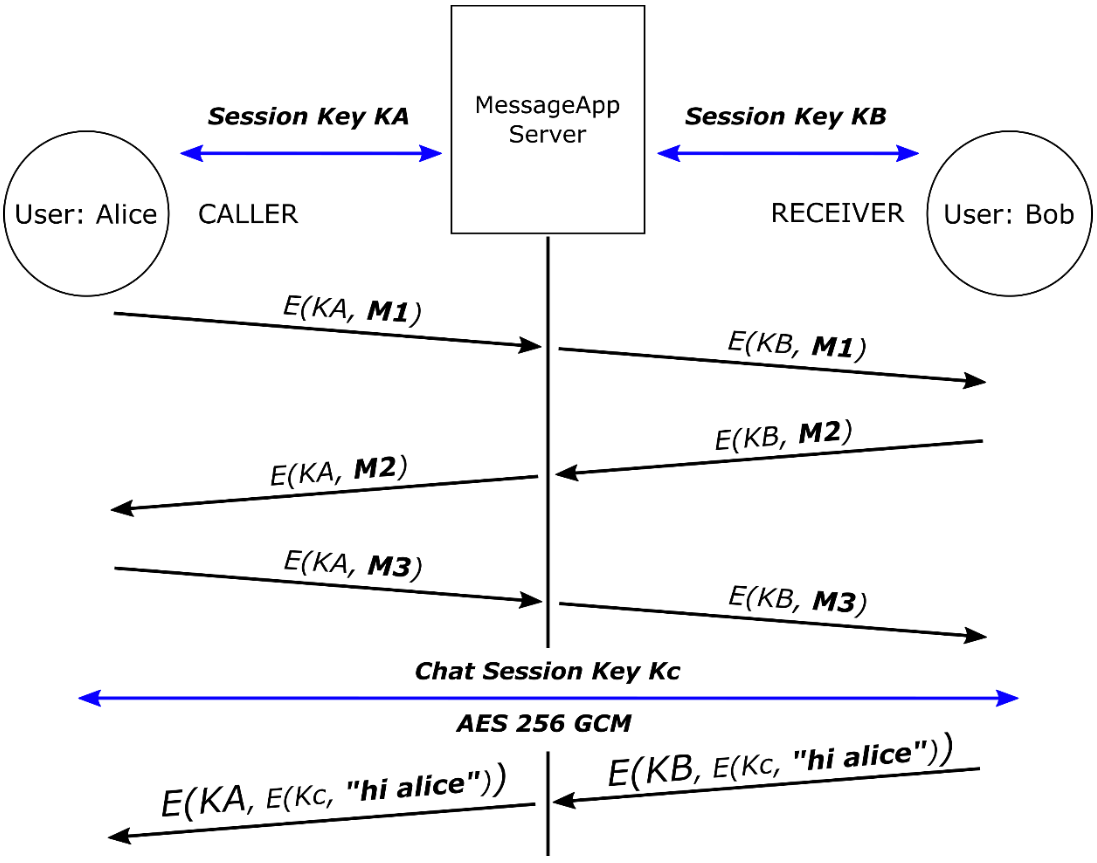


Figure 15 – Chat key negotiation with Server encryption

As the symmetric 32-byte key is shared between the user in Chat, the format of the encrypted messages is described in Figure 16. Again, it was used AES 256 GCM for authenticated encryption.

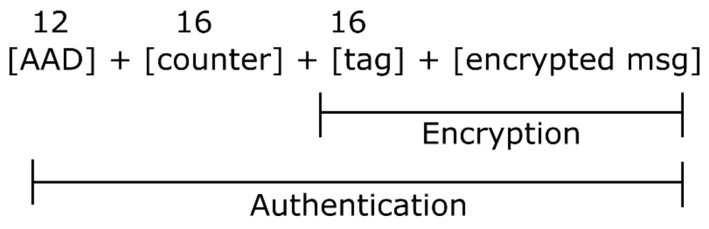


Figure 16 – Chat AES 256 GCM message format

As a final example of the message contests sent in the application context, a text transmission from Alice to Bob is represented in Figure 17. All the encryption layers are shown, 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.

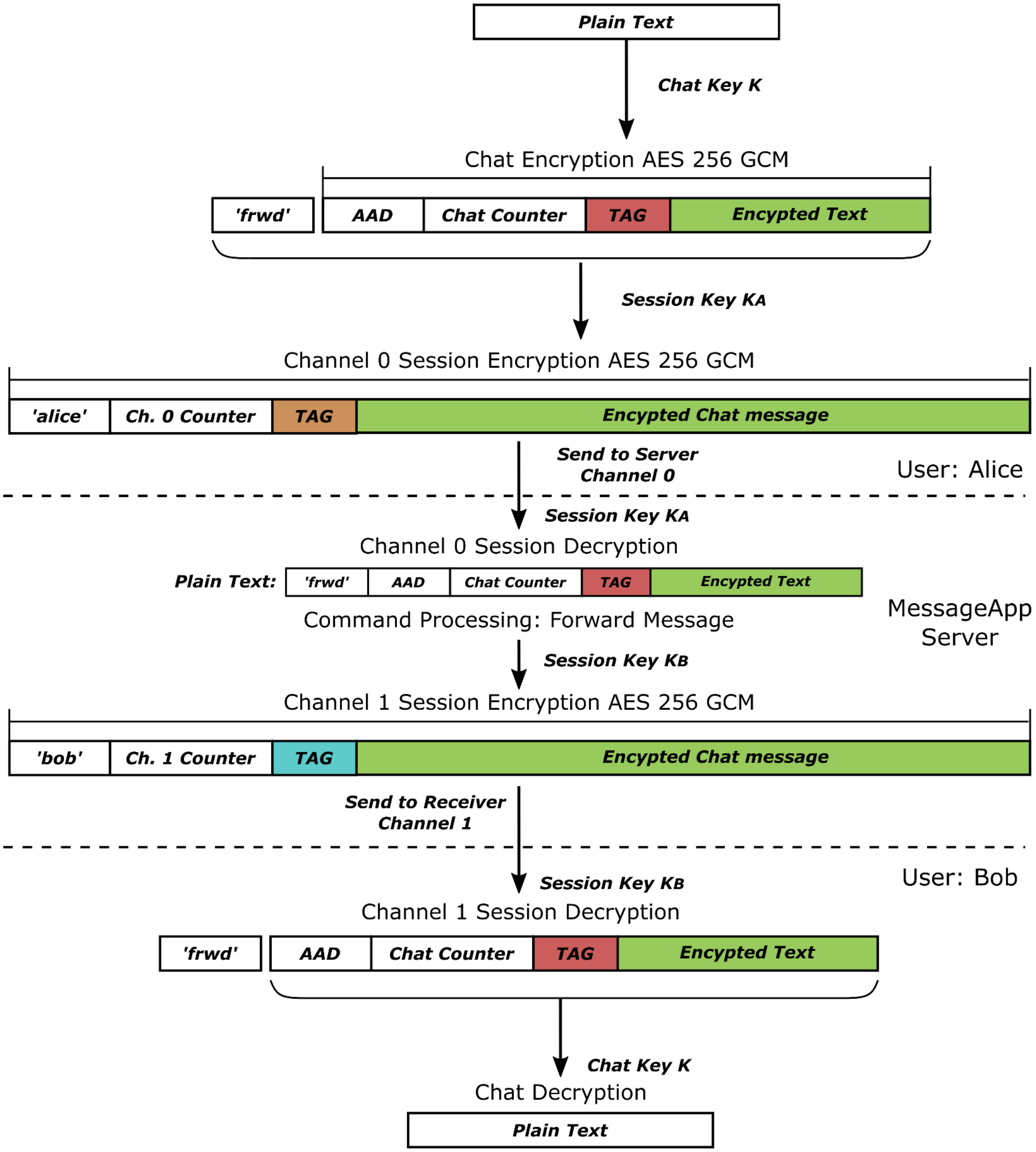
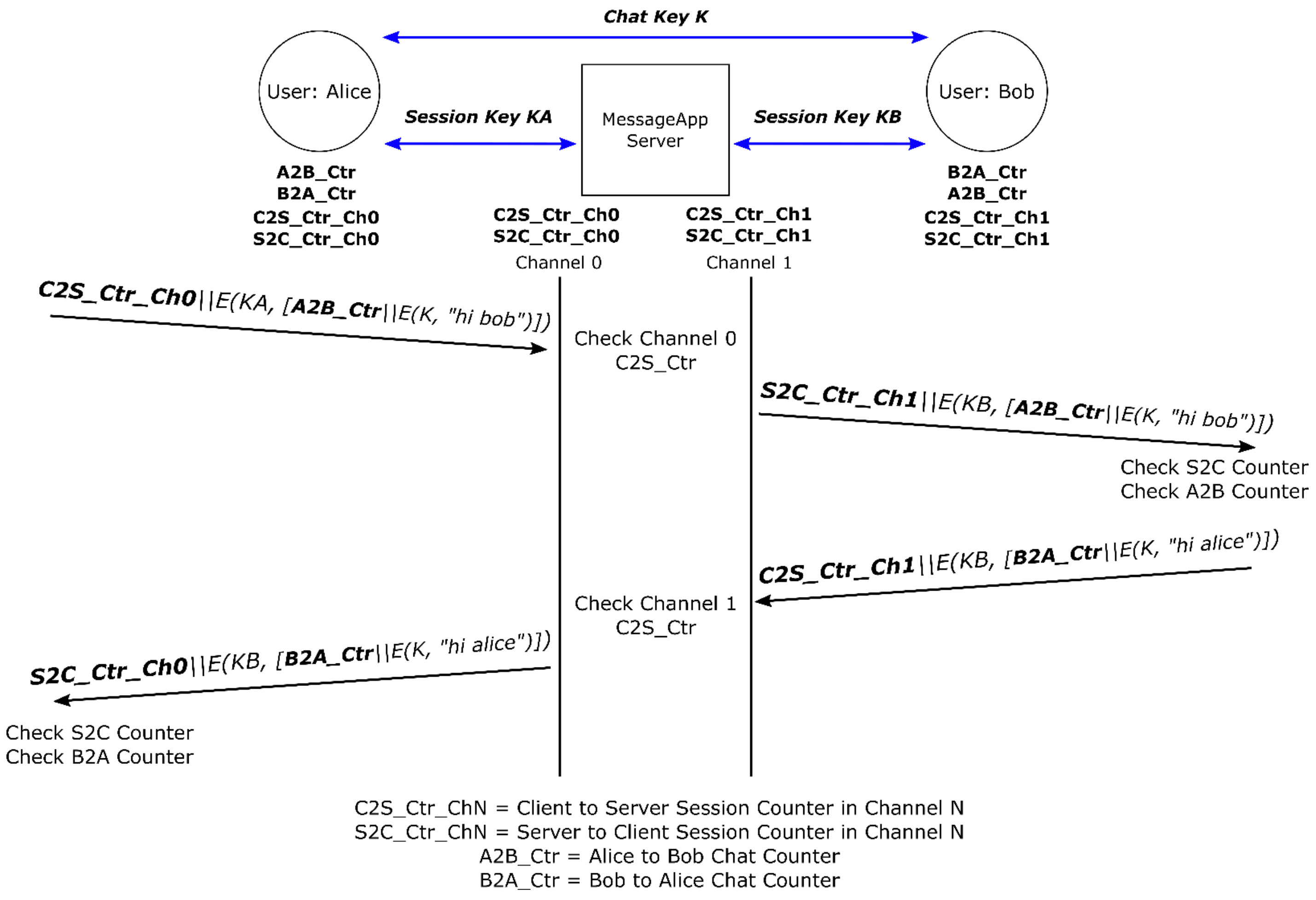


Figure 17 – Format of the messages when sending a plain text to the friend in Chat

Another important remark from the message exchange is the handling of the session counters and the Chat counter. In this way, a User when connected to the server must keep track of two different counters, which is one to send the messages (client to server counter), and one to receive the messages from the server (server to client counter), as previously mentioned at the end of the Handshake protocol. The multiple counters used are a mandatory feature to prevent replay attacks, and the server then maintains the handling of two counter per channel.

Also, as the Chat is established, both users must also keep track of two Chat related counters, one used to send messages from the caller to the sender and one used to track the messages sent from the receiver to the caller. An example on how the counters is organized in the messages is illustrated in Figure 18:



Chat Key K
User: Alice
Session Key KA
MessageApp Server
Session Key KB
User: Bob
A2B_Ctr
B2A_Ctr
C2S_Ctr_Ch0
S2C_Ctr_Ch0
C2S_Ctr_Ch0
S2C_Ctr_Ch0
Channel 0
C2S_Ctr_Ch1
S2C_Ctr_Ch1
Channel 1
B2A_Ctr
A2B_Ctr
C2S_Ctr_Ch1
S2C_Ctr_Ch1
C2S_Ctr_Ch0||E(KA, [A2B_Ctr||E(K, "hi bob")])
Check Channel 0 C2S_Ctr
S2C_Ctr_Ch1||E(KB, [A2B_Ctr||E(K, "hi bob")])
Check S2C Counter
Check A2B Counter
C2S_Ctr_Ch1||E(KB, [B2A_Ctr||E(K, "hi alice")])
Check Channel 1 C2S_Ctr
S2C_Ctr_Ch0||E(KB, [B2A_Ctr||E(K, "hi alice")])
Check S2C Counter
Check B2A Counter
C2S_Ctr_ChN = Client to Server Session Counter in Channel N
S2C_Ctr_ChN = Server to Client Session Counter in Channel N
A2B_Ctr = Alice to Bob Chat Counter
B2A_Ctr = Bob to Alice Chat Counter

Figure 18 – Counter organization in MessageApp message exchange

Using MessageApp

To use the developed application in a demonstration environment, the created user must have its public key copied 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 interface command is 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 cannot contain any special characters and must not exceed 15 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 first six bytes if 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. To check all the available commands if any updates are available, the user can type the help commands. The message from the server will also contain extra fields, like current version, user information in the server, encryption capabilities and other parameters regarding the session:

```
[SERVER-COMMAND]->helpx
```

To start chatting with others, use the following command to request the list of online users:

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

After verifying the available users (the ones already in a chat session will be indicated in the list), to chat with a friend in the server type the following command (in the example the friend username is ‘bob’):

```
[SERVER-COMMAND]->chat bob
```

From the Receiver side, you may accept or not the connection, as a message from the server will be displayed in the interface containing the Caller username. From the Client application interface type in the following command to accept the Chat request:

```
[SERVER-COMMAND]->acptx
```

If it is not desired to chat with the caller, type the following command:

```
[SERVER-COMMAND]->refux
```

Considering that the Receiver might stall to reply to the request, or other situation that prevents the Receiver to respond, the Caller can execute the following command to Cancel the Chat Request (both users will have its pendency in the Server removed, and will be available for other Chats):

```
[SERVER-COMMAND]->cachx
```

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 is desired to exit the program inside the Server command (not in the Chat) type in the command:

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