Project Report

Communication Networks: Protocols ans Architectures

Wannes Nevens, Sam Van de Velde, Louis Van Eeckhoudt, Jonathan Vrijsen

Contents

1 Introduction

The goal of this project is to create a chat application which enables private communication. In order to achieve this, the topology shown in figure 1 is used. Throughout this report, all the features will be explained and all design choices will be justified. For this project, the Python programming language has been used.

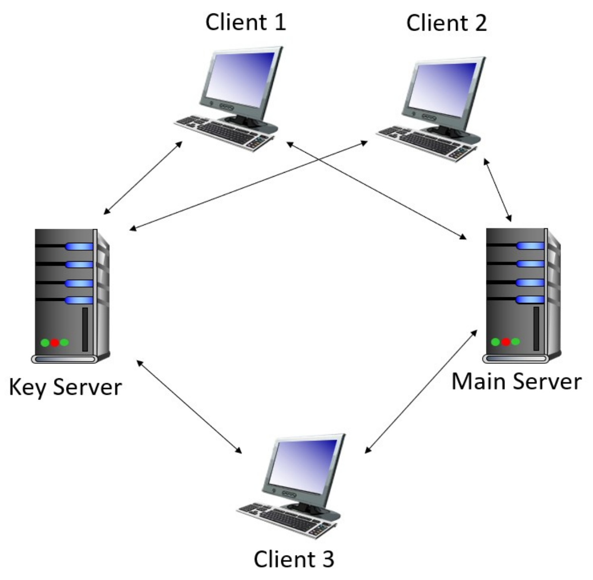


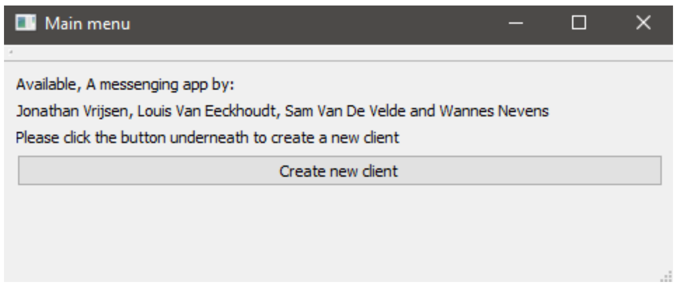
Figure 1: Topology of the chat application. Taken from [1]

2 Features

Before diving into the technical details of the code, it is advisable to first discuss the experience from the viewpoint of the user, e.g. what the gui allows.

2.1 The general overview

Since both servers and all clients are put on the same device within the frame of this project, an extra window is constructed, see figure 2. This window allows to create as many clients as desired. The main server's and key server's overview (see figures 14 and 15) are also immediately shown.



Available, A messenging app by:
Jonathan Vrijsen, Louis Van Eeckhoudt, Sam Van De Velde and Wannes Nevens
Please click the button underneath to create a new client

Figure 2: The welcoming screen allows the creation of as much clients as desired.

2.2 The client window

In the client window, you are first greeted to the registration/login window, as seen in figure 3. Registering with an unused username and three times[1] the same password results in a confirmation message (see figure 4). If a username that's already occupied is requested, or if the passwords aren't identical, the user gets informed, as seen in figures 5 and 6 respectively.

When logging in, if the username isn't in use or the password is incorrect, the user also gets informed with the same message, as seen in figure 7

When logged in succesfully, one is greeted to a new window: the conversation overview window (see figure 8). This new window has many options, including a button to log out (and bring you back to the previous window), a button to refresh the list of ongoing conversations manually, a button to create a new conversation, a textbox showing the current conversations, and an input box to type a new message with a send button on the right of it. Notice as well the username of the logged in client portrayed in the title of the window.

When a new conversation is created, the client requests an overview of all possible accounts to contact, as seen in figure 9. Click on one name to start a conversation with said person, or select multiple people to create a group conversation. When done selecting, click on "Create new conversation" again to confirm (or on "Back" to cancel). Having created a new conversation, you return to the previous screen, now with a new conversation visible. Figures 10 until 13 illustrate a conversation between two users and a group conversation.

[1]Although the odds of making the same typo twice when entering a password are indeed rather low, the odds of making it three times are even lower.

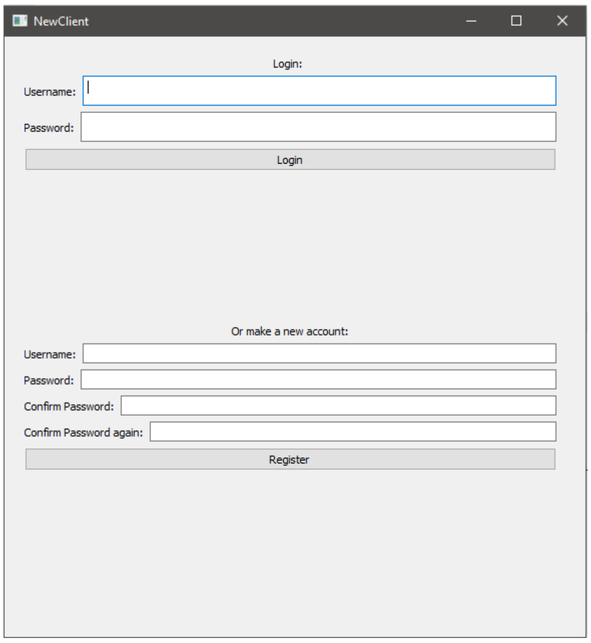
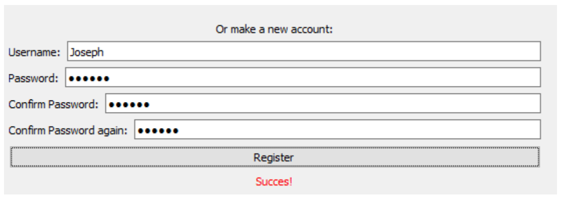


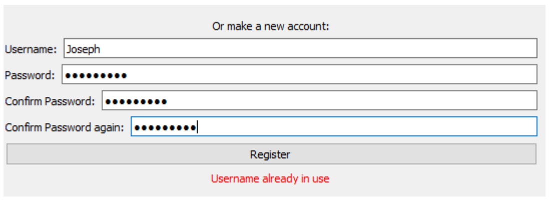
Figure 3: As a client, you're asked to register or log in.



Joseph

Succes!

Figure 4: If the registration has been completed succesfully, this message will appear.



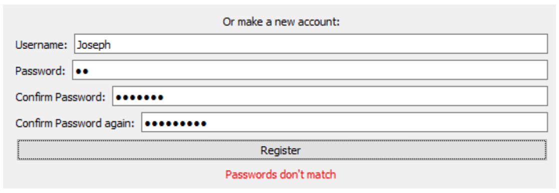
Joseph

Username already in use

Figure 5: If the username is already occupied by another user, this message will appear.

2.3 The main server window

In this window, one can see the saved conversations (in the left column) and the symmetric keys that have been constructed for each connected client (right column). The public and private key are also (partially, due to their length) shown above.


Or make a new account:

Username: Joseph

Password: ●●

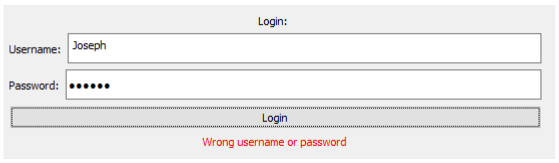
Confirm Password: ●●●●●●●

Confirm Password again: ●●●●●●●●●

Register

Passwords don't match

Figure 6: If the passwords don't match, this message will appear.


Login:

Username: Joseph

Password: ●●●●●●

Login

Wrong username or password

Figure 7: If the password is incorrect or the username isn't in use, this message will appear.

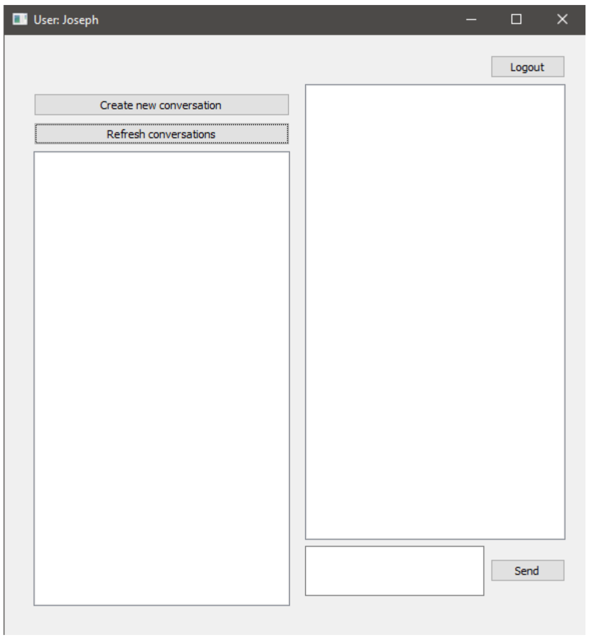


Figure 8: When logging in succesfully, you are greeted to this new window.

2.4 The key server window

In this window, one can see the registered users, with username and encrypted password, (in the left column) and the symmetric keys that have been constructed for each connected client (right column). The public and private key are also (partially, due to their length) shown above.

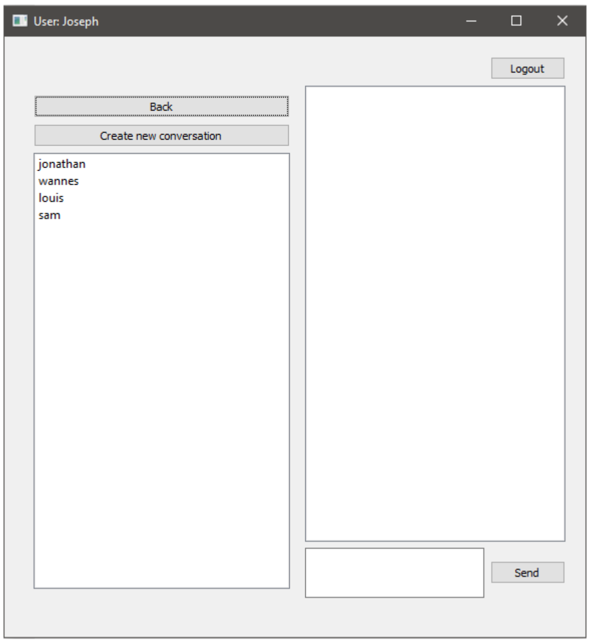
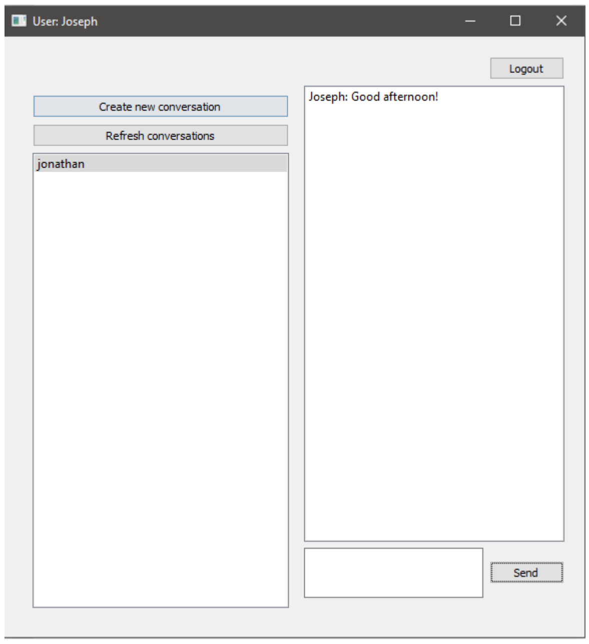
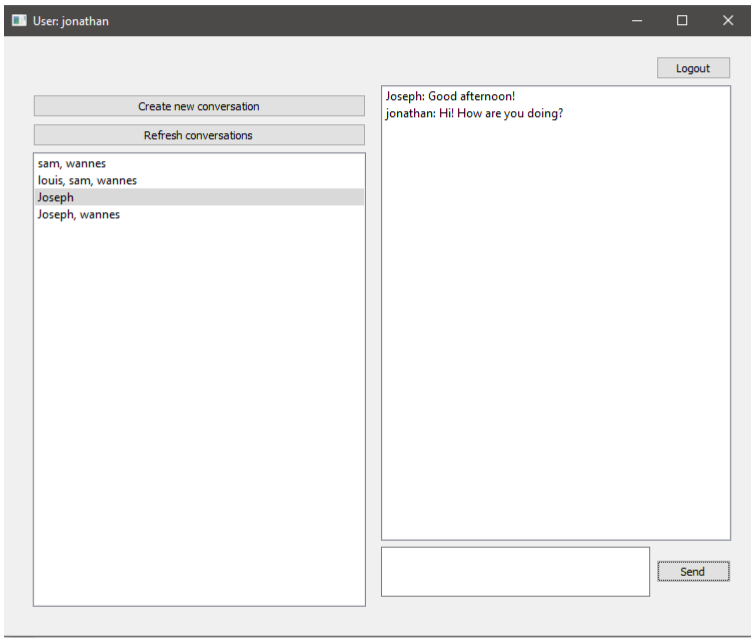


Figure 9: When clicking on "Create new conversation", all possible contacts appear. Notice that you can select multiple accounts at once to create a group chat.



Joseph: Good afternoon!

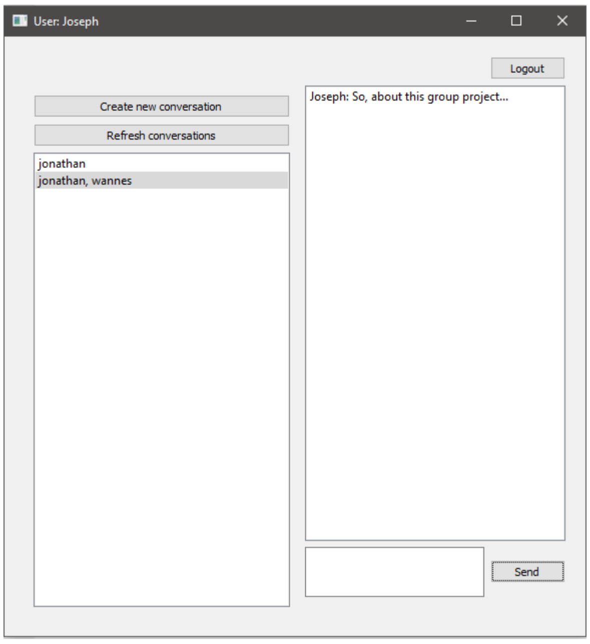
Figure 10: The user Joseph has created a conversation with user Jonathan, and has sent a message in this conversation.



sam, wannes
louis, sam, wannes
Joseph
Joseph, wannes

Joseph: Good afternoon!
jonathan: Hi! How are you doing?

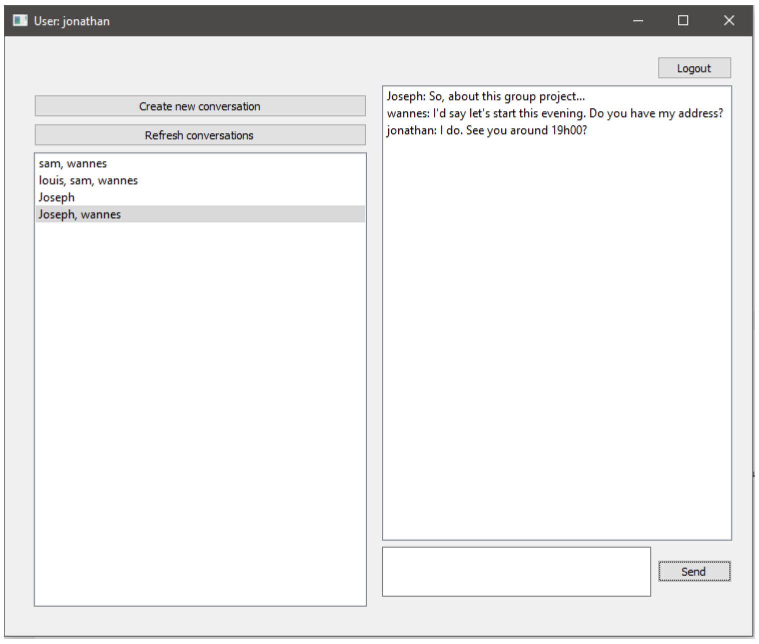
Figure 11: Jonathan receives the message and is able to respond.



jonathan
jonathan, wannes

Joseph: So, about this group project...

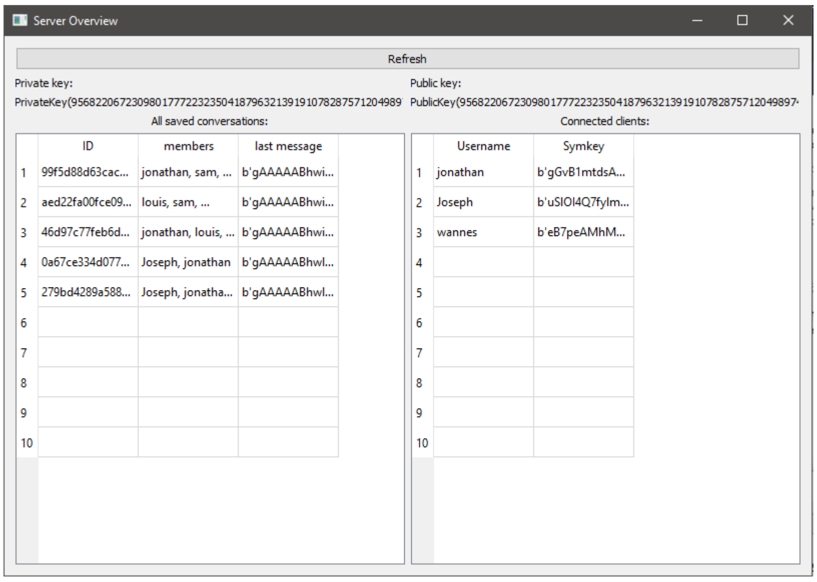
Figure 12: The user Joseph now also has created a group chat with users Jonathan and Wannes. He sent a message in this group chat.



sam, wannes
louis, sam, wannes
Joseph
Joseph, wannes

Joseph: So, about this group project...
wannes: I'd say let's start this evening. Do you have my address?
jonathan: I do. See you around 19h00?

Figure 13: Jonathan also sees the group chat and is able to participate.



Private key:
PrivateKey(956822067230980177722323504187963213919107828757120498

Public key:
PublicKey(9568220672309801777223235041879632139191078287571204989

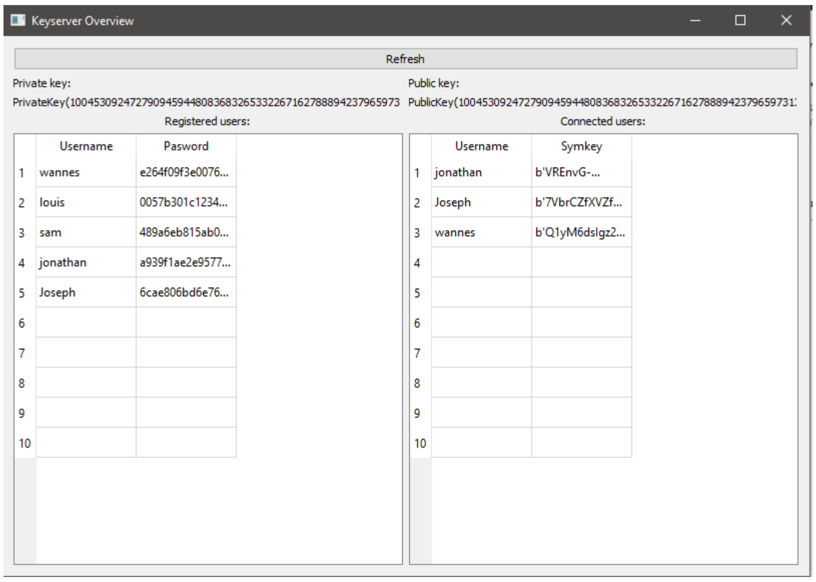
All saved conversations:

	ID	members	last message
1	99f5d88d63cac...	jonathan, sam, ...	b'gAAAAABhwi...
2	aed22fa00fce09...	louis, sam, ...	b'gAAAAABhwi...
3	46d97c77feb6d...	jonathan, louis, ...	b'gAAAAABhwi...
4	0a67ce334d077...	Joseph, jonathan	b'gAAAAABhwl...
5	279bd4289a588...	Joseph, jonatha...	b'gAAAAABhwl...
6			
7			
8			
9			
10			

Connected clients:

	Username	Symkey
1	jonathan	b'gGvB1mtdsA...
2	Joseph	b'uSIOI4Q7fylm...
3	wannes	b'eB7peAMhM...
4		
5		
6		
7		
8		
9		
10		

Figure 14: An overview of the main server, showing its data. Notice the encrypted symmetrical keys and messages.


Keyserver Overview

Refresh

Private key:

PrivateKey(1004530924727909459448083683265332267162788894237965973

Public key:

PublicKey(10045309247279094594480836832653322671627888942379659731

Registered users:

	Username	Pasword
1	wannes	e264f09f3e0076...
2	louis	0057b301c1234...
3	sam	489a6eb815ab0...
4	jonathan	a939f1ae2e9577...
5	Joseph	6cae806bd6e76...
6		
7		
8		
9		
10		

Connected users:

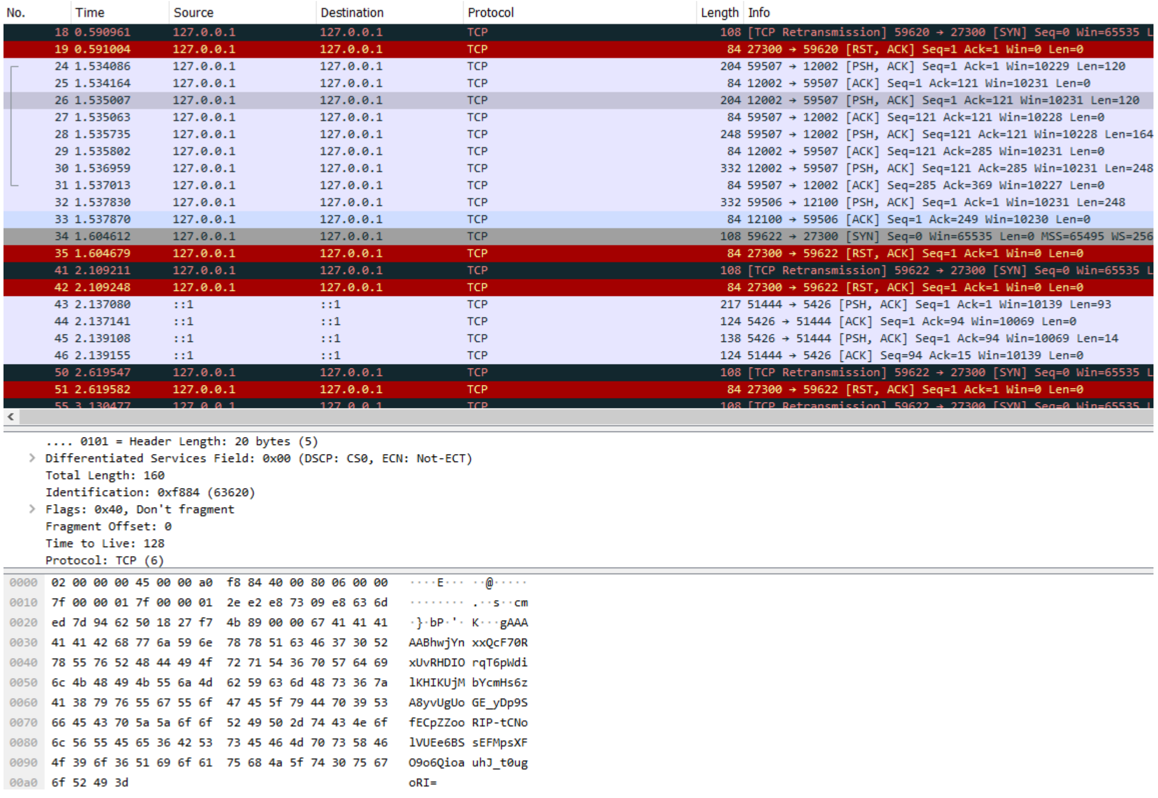
	Username	Symkey
1	jonathan	b'VREnvG-...
2	Joseph	b'7VbrCZfXVZf...
3	wannes	b'Q1yM6dslgz2...
4		
5		
6		
7		
8		
9		
10		

Figure 15: An overview of the key server, showing its data. Notice the encrypted passwords and keys.

3 Architecture

3.1 A mandatory clarification

Before delving into the details of the network, it's propably best to clarify that the network described below is imitated on a single device: a single piece of hardware plays the role of the main server, the key server, and all the clients part of the application network. That means that, instead of packets travelling through the internet, they simply leave and enter a port immediately. These signals can still be observed in wireshark as shown in figure 16. The code is written in a way that this isn't final and can easily be transformed into a 'real' networking application, but this was the only way to inspect all participating nodes from one device. From here on, the architecture will be talked about as if communication between different nodes of the internet occur, but do keep this detail in mind.



```
No.    Time        Source        Destination   Protocol  Length Info
    18 0.590961    127.0.0.1     127.0.0.1     TCP          108 [TCP Retransmission] 59620 → 27300 [SYN] Seq=0 Win=65535 L
    19 0.591004    127.0.0.1     127.0.0.1     TCP           84 27300 → 59620 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
    24 1.534086    127.0.0.1     127.0.0.1     TCP          204 59507 → 12002 [PSH, ACK] Seq=1 Ack=1 Win=10229 Len=120
    25 1.534164    127.0.0.1     127.0.0.1     TCP           84 12002 → 59507 [ACK] Seq=1 Ack=121 Win=10231 Len=0
    26 1.535007    127.0.0.1     127.0.0.1     TCP          204 12002 → 59507 [PSH, ACK] Seq=1 Ack=121 Win=10231 Len=120
    27 1.535063    127.0.0.1     127.0.0.1     TCP           84 59507 → 12002 [ACK] Seq=121 Ack=121 Win=10228 Len=0
    28 1.535735    127.0.0.1     127.0.0.1     TCP          248 59507 → 12002 [PSH, ACK] Seq=121 Ack=121 Win=10228 Len=164
    29 1.535802    127.0.0.1     127.0.0.1     TCP           84 12002 → 59507 [ACK] Seq=121 Ack=285 Win=10231 Len=0
    30 1.536959    127.0.0.1     127.0.0.1     TCP          332 12002 → 59507 [PSH, ACK] Seq=121 Ack=285 Win=10231 Len=248
    31 1.537013    127.0.0.1     127.0.0.1     TCP           84 59507 → 12002 [ACK] Seq=285 Ack=369 Win=10227 Len=0
    32 1.537830    127.0.0.1     127.0.0.1     TCP          332 59506 → 12100 [PSH, ACK] Seq=1 Ack=1 Win=10231 Len=248
    33 1.537870    127.0.0.1     127.0.0.1     TCP           84 12100 → 59506 [ACK] Seq=1 Ack=249 Win=10230 Len=0
    34 1.604612    127.0.0.1     127.0.0.1     TCP          108 59622 → 27300 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256
    35 1.604679    127.0.0.1     127.0.0.1     TCP           84 27300 → 59622 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
    41 2.109211    127.0.0.1     127.0.0.1     TCP          108 [TCP Retransmission] 59622 → 27300 [SYN] Seq=0 Win=65535 L
    42 2.109248    127.0.0.1     127.0.0.1     TCP           84 27300 → 59622 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
    43 2.137080    ::1           ::1           TCP          217 51444 → 5426 [PSH, ACK] Seq=1 Ack=1 Win=10139 Len=93
    44 2.137141    ::1           ::1           TCP          124 5426 → 51444 [ACK] Seq=1 Ack=94 Win=10069 Len=0
    45 2.139108    ::1           ::1           TCP          138 5426 → 51444 [PSH, ACK] Seq=1 Ack=94 Win=10069 Len=14
    46 2.139155    ::1           ::1           TCP          124 51444 → 5426 [ACK] Seq=94 Ack=15 Win=10139 Len=0
    50 2.619547    127.0.0.1     127.0.0.1     TCP          108 [TCP Retransmission] 59622 → 27300 [SYN] Seq=0 Win=65535 L
    51 2.619582    127.0.0.1     127.0.0.1     TCP           84 27300 → 59622 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
    55 3.130477    127.0.0.1     127.0.0.1     TCP          108 [TCP Retransmission] 59622 → 27300 [SYN] Seq=0 Win=65535 L
```

```
    .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 160
    Identification: 0xf884 (63620)
  > Flags: 0x40, Don't fragment
    Fragment Offset: 0
    Time to Live: 128
    Protocol: TCP (6)
```

```
0000  02 00 00 00 45 00 00 a0  f8 84 40 00 80 06 00 00   ····E··· ··@·····
0010  7f 00 00 01 7f 00 00 01  2e e2 e8 73 09 e8 63 6d   ········ .··s··cm
0020  ed 7d 94 62 50 18 27 f7  4b 89 00 00 67 41 41 41   ·}·bP·'· K···gAAA
0030  41 41 42 68 77 6a 59 6e  78 78 51 63 46 37 30 52   AABhwjYn xxQcF70R
0040  78 55 76 52 48 44 49 4f  72 71 54 36 70 57 64 69   xUvRHDIO rqT6pWdi
0050  6c 4b 48 49 4b 55 6a 4d  62 59 63 6d 48 73 36 7a   lKHIKUjM bYcmHs6z
0060  41 38 79 76 55 67 55 6f  47 45 5f 79 44 70 39 53   A8yvUgUo GE_yDp9S
0070  66 45 43 70 5a 5a 6f 6f  52 49 50 2d 74 43 4e 6f   fECpZZoo RIP-tCNo
0080  6c 56 55 45 65 36 42 53  73 45 46 4d 70 73 58 46   lVUEe6BS sEFMpsXF
0090  4f 39 6f 36 51 69 6f 61  75 68 4a 5f 74 30 75 67   O9o6Qioa uhJ_t0ug
00a0  6f 52 49 3d                                        oRI=
```

Figure 16: In wireshark the messages sent can be observed, here packages number 24 through 31 is the login conversation between a host and the keyserver. (note that the message is encrypted and unreadable)

3.2 General Overview

The application software is divided in three parts. Each node runs a different part of the application to account for the different functions it plays, as seen in figure 1.

(i) Software unique to the main server

(ii) Software unique to the key server

(iii) The client software running on all the other nodes (since there exists only one main server and key server)

These Different nodes will be discussed below.

3.3 The main server

The first part of the application network, is the main server. This node stores all the conversations, each being encrypted by a key solely owned by the participants of the conversation. It is thus apparent that the main server is not able to read these messages, an advantage further discussed in 4.5. The communication between the client and the main server is performed using a secure channel, which is based on a symmetric key only known by a client and the main server. The question that arises is how to securely agree on a symmetric key, in a way that no intruder can obtain this shared symmetric key as well.

This is done using the following handshaking procedure. The client sends a "keyrequest", as well as its own public key, to the main server. The main server receives this message and concludes that a client wants to install a secure channel between them. Hence, the server generates a symmetric key, encrypts this with the client's public key and sends at first its own public key and secondly the encrypted symmetric key to the client. These messages allow the client to decrypt the the symmetric key. After this procedure, a secure channel based on an agreed symmetric key is completed.

One might wonder why the public key of the main server is needed as well in order to decrypt and decode the received symmetric key. This is because the integrity of the encrypted symmetric key is also ensured. More specifically, the main server creates a signature based on the message it wants to send (the symmetric key), its own private key, and a hashing algorithm (SHA-1 has often been used as a hashing algorithm throughout this project). The client can verify this signature using the decrypted symmetric key, the server's public key and the same hashing algorithm by recalculating the signature and comparing it to the received signature.

Three things should still be noted:

- The used algorithm for the asymmetric keying is RSA. Its library can be installed and implemented in Python.
- Since encryption using asymmetric keys requires a lot of computations with respect to symmetric keys, it is only used to exchange a symmetric key and create a secure channel.
- RSA limits the length of the encrypted message to the length of the key, making it not suitable to secure any message between the client and the server anyway.

After instantiating the secure channel, the server dedicates a thread to this client which is now considered to be a "connected client". A socket dedicated to this connected client is created, and a thread is launched on the server to receive and handle any message coming in from this socket. The client can request different things:

(i) Since the official registration of the user happens at the smaller key server (discussed later on in the report), the client will need to confirm its registration to the main server. Remember that, by design, no direct communication between the key server and the main server exists. To get a confirmation of the successful registration of the user at the key server side, a certificate[2] created by the key server is sent by the client to the main server. This certificate is unique for each registered user. The main server validates this certificate and adds the username of the registered user proposed by the client to its list of "known users". Notice that a client cannot fake a registration to the main server without first obtaining a certificate from the key server, hence without first completing an actual registration.

(ii) A request for logging in can be done as well. Logging in actually takes place at the key server side (as discussed later on). Using the certificate the client obtained after successfully logging in at the key server, it can proof to the main server that it is indeed logged in correctly. Consequently, a logged in user is linked to the connected client at the main server side.

(iii) The client can request all the conversations its user is already part of. The main server then returns all conversation id's accompanying linked to these conversations.

(iv) Using the obtained id's, the client can request a conversation belonging to an id, and the server, first checking if the client is indeed part of the conversation it requests, returns the conversation belonging to this id. Note that the last check is needed in order to prevent that an entire conversation can be obtained by only handing the conversation id. This limits in a way the power of the key server, which only doesn't know any member of the conversation, as will become clear later on during this report. Remember that the messages in the conversation are stored encrypted in the main server. So the client will also need to somehow obtain the key related to the conversation via the key server.

(v) The client can request a list of all known users, in order to see with whom he would desire to start a new conversation. The main server then replies with all successfully registered users.

(vi) The client can request to create a new conversation with one or multiple other accounts. The server updates its data accordingly.

(vii) The client can request to add a message to an ongoing conversation, by sending the id of the said conversation and the new message. The server updates its data accordingly, and sends this message to all other members of the conversation that are connected, thus causing that the connected members to be updated automatically instead of having to check for updates themselves. If a user is logged in at multiple clients at the same time, all the other clients receive the new message as well.

(viii) The client can request all members of a conversation, sending a conversation id. The Server checks if the client is part of the said conversation, and returns the requested usernames.

(ix) The client can request to log out. The server updates its data accordingly, and the user related to the connected client is deleted. Note that, as long as the client itself is still active, the actual connecting socket nor the secure channel is destroyed.

a final remark on how the state of the server is stored while shutting down, and how it is retrieved while launching. Both the known users as all the conversations are stored in the case of this project in text files, mainly by using the JSON library in Python. Be sure to have a look at the *conversations.txt* and *known_users_main_server.txt* files to get an idea of the state of

[2] A sufficient certificate is in this case an encrypted version of the username, using a symmetric key only known by the key server and main server. In reality, this key could be supposed to be pre-installed while creating the servers. Within this project, it is stored in a simple .txt file and read by both servers while launching.

the main server before launching and after closing[3]. Note that the messages in a conversation are stored in an encrypted way.

A second remark on how the address of the server is distributed. The server will regularly broadcast its address, together with an identifying tag, using UDP. The keyserver will do the same. On client startup, it will first listen to the broadcast until both server addresses have been acquired. Afterwards the client can start the handshaking procedure to create a secure channel.

3.4 The key server

The second part of the application network, is the key server. This server isn't necessary to have a working application network, but introduced in order to have a secure one.

As its name suggests, the key server stores the keys used for decrypting the conversations stored in the main server, resulting in a lot of security advantages, see 4.5. The key server not only stores but also creates new keys. Other than that, actions like registering and logging in occur first at the key server side rather then at the main server. Hence, the username and password (to be exact: the SHA-1 hash of the password) are stored in the key server as well.

The key server communicates with clients in a similar way as the main server. The same method is also used to create a secure channel between the client and the key server.

Once the client is connected to the key server, a couple of requests can be done:

(i) A client could try to register for the first time, thus sending a username and password. The key server first checks if the username already is in use and, if not, returns the corresponding certificate, used as a proof of registration by the client in order to register at the main server as well.

(ii) A client could try to log in, thus sending a username. The key server checks if the user exists, and if so, asks for the password. This is the start of the challenge-response authentication used to log in.

(iii) A client could then, in a next message, send the SHA-1 hash of the password. If this hash corresponds to the hash received while registering the user, the password is correct and the key server returns the certificate of this user. The client can use this certificate to log in at the main server as well. During the rest of the connection, the client is now concluded to be logged in correctly and marked as such by the key server, and thus this client is allowed to request conversation keys, just until the user is logged out again.

(iv) Now, two variations of the same thing can occur. The client could try to request a new conversation key for a new conversation, or it could request for the key belonging to a specific conversation. Both cases only require the conversation id as an input[4] . In the first case, the key server generates a new key (the probability of an identical key being generated is statistically negligible) and returns this, as always via the secure channel. In

[3]Consequently, it should only be seen as an easy way to not having to constantly register each user after launching the project. Of course, it introduces new security issues, but they are considered to be outside the scope of this project since this is obviously not the way in which data is stored in a real life scenario.

[4]It is interesting to discuss the mechanism behind these conversation keys. A conversation id is a hash of all the members of the conversation, sorted alphabetically. Hence, if one knows all the members, it knows the id of the conversation. This is the case for the clients (which can obtain the members of a conversation via the main server) as well as the main server itself. Since the main server does not know any password, only a logged in client can calculate a conversation id an request the linked conversation key. In the other hand, the key server does only know the id of a conversation and thus not all the members, making it harder (yet obviously not impossible) for the key server to retrieve conversations from the main server and decrypt these (remember that the main server checks if the client that requests the conversation is part of the conversation).

the other case, the key is returned. For both of these steps, it is of course checked that the client is properly logged in before returning the keys.

Final remark is that data is stored the similarly as for the main server. Have a look at the *conversation_keys.txt* and *registered_users.txt* files.

3.5 The clients

The clients interact with both the main and the key server: hence, each client needs two ports. Note that the clients don't know about each other's location in the network, and thus aren't able to communicate directly with each other like in a peer-to-peer network. This unawareness brings the obvious security advantage of the other users not being able to locate you without your permission, as further elaborated upon in 4.5. The client is able to reach out to both servers and thus initiate the creation of secure channels between themselves and each server.

With the main server, the following interactions could occur securely after this creation:

(i) The main server could send a message with the id according to the right conversation. Consequently, the client updates the conversation in its local memory. This can also be done anytime by refreshing and thus sending a request to the main server.

(ii) The main server could also send all available conversation id's, in order to know which conversations to request. The client can respond by fetching the conversations using the given id's afterwards.

(iii) The main server could be sending the requested list of all existing users, as a response to a contact request. This data is stored in the client's memory as well, and allows the user to see which users are registered in the app.

With the key server, the next interactions could occur securely after the creation of a secure channel using symmetric keying:

(i) Within the frame of the challenge-response authentication scheme, the key server could be asking for the right password to log in. Sending the SHA-1 hash of the right password via the instantiated secure channel is the desired response.

(ii) The key server could transmit the keys (as always, via the secure channel), which need to be stored in the memory of the client in order to be used to decrypt messages of conversations.

A final remark is that the client never stores data after a session. Each time the application is ran again, the client needs to access the keys from the key server, and the conversations and such from the main server. This brings safety advantages, as further discussed in 4.5.

4 Security

Although the security measurements can be derived from the section about architecture, an extra chapter to this important subject is added. This way, it'll be easier to summarise which steps have been undertaken in order to assure safe communications.

4.1 The server-central model

The two servers are central in this network, as seen in figure 1, with each client connecting only to these two devices. This way, it's only possible to send messages from one user to the other indirectly (with the main server acting as a median) without the other being aware of one's location on the network. This way, a user with bad intentions isn't able to figure out his victim's location on the internet.

4.2 The two-server model

The messages of conversations are stored encrypted on the main server, to ensure the main server (or, for example, some bad people hacking the server) isn't able to read the conversations it contains. These keys are physically stored on another device: the key server. This way, the key server has all the keys but no conversations, and vice versa. Although communication between the two servers doesn't happen by design, the two still share a (one-time manually configured) identical key, in order to validate if information that the client sends is validated by the other server. In other words, this shared key allows indirect communication between the servers, to ensure each other that the client isn't lying. In a way, the same way that the servers are medians between two clients willing to communicate, the clients also act as medians between the two servers when carrying information (e.g. information concerning registration or login) that one server is forbidden to obtain directly from the other.

4.3 Asymmetric keying

Each node of the network generates upon initialisation secure channels with specified other nodes, based on his own public and private key and the public key of the node at the other side of the secure channel. In order to create these, asymmetric keying allows to safely agree on a symmetric key between both ends of the secure channel.

Because of earlier mentioned reasons, asymmetric keying is only used during the initialisation of a secure channel. Hence, the only communication that happens in an unencrypted way during the entire life time of the program is the exchange of public keys.

4.4 Symmetric keying

After the initialisation, which contains the creation and sharing of symmetric keys using asymmetric keying, all communication on the channel between node and server, from the transfer of other keys and conversations to the registering and logging in of users, occurs encrypted using these keys.

Other uses of symmetric keys in the project are for encrypting conversations stored in the main server and creating certificates of registration and login from the key server.

Throughout this project, the Fernet library has been used to generate symmetric keys. This is an easy-to-use library in Python, which contains all needed functions. As a result, all symmetric keys are the same length.

4.5 Hashing

Sending or storing a password to or at the key server are risky operations, even if it encrypted.

To resolve this, and to resolve other possible security breaches in this application, hashing has also been applied. The idea behind this is that applying the hashing on a certain variable will always result in the same, unrecoverable, outcome. Thus, when comparing two hashings of the same variable, the two hashed variables should also be identical (unless of course different hashing algorithms have been used for each variable).

Hence, the client only sends a SHA-1 hash of the password, so that the actual password is only ever stored in the mind of the person registering or logging in a user.

Due to the property that the outcome of a hash occurring twice for different inputs is negligible within the small scale of this project, it is also used to create and obtain the unique conversation ids based on all the members of a conversation.

5 Conclusion

As a conclusion, one can state that the final result of the project is a working and secured chat application. The used topology is based on a main server, as well as a key server used for security measures, which causes that clients do not directly communicate with each other. All data sent from one node to another happens in an encrypted way via secure channels, except for the sharing of public keys. The conversations are stored in an encrypted way, and can only be read by logged in users, and passwords are never sent nor stored in their original form.

Apart from the original project assignment, a couple of extra features have been added:

- All communication between all nodes takes place via secured channels, created via asymmetric keying.
- The state of the servers (conversations, registered users...) is stored while shutting down, and loaded while launching the application.
- Multiple extra security measures have been implemented by means of the used topology, the conversation ids...
- The creation of conversations with more than two members is possible as well.

5.1 Known issues

One final remark in this report is a summary of the known problems that may occur while running the app:

- Refreshing doesn't always work from the first time, and causes a non-fatal exception from time to time.
- Closing down the application takes a while, and doesn't always succeed.

The main reason for these bugs can be linked to threading, and can be solved by properly ensuring thread safety. This is thus the main point of improvement for this project.

References

[1] Kurose, J., Ross, K. (2016). Computer Networking: A Top-Down Approach, Global Edition. Pearson Education.