Secure Instant Messaging System

Milestone 2 (Final)

p2g2
Hugo Fragata 73875
Gonçalo Grilo 72608

Contents

1. Introduction

This report describes the second and final milestone of the Secure Instant Messaging System project for the Security class of 2016 of the Degree in Computer and Telematics Engineering at the University of Aveiro.

Secure Instant Messaging System is, as the name states, an instant messaging tool which has certain security properties in communications.

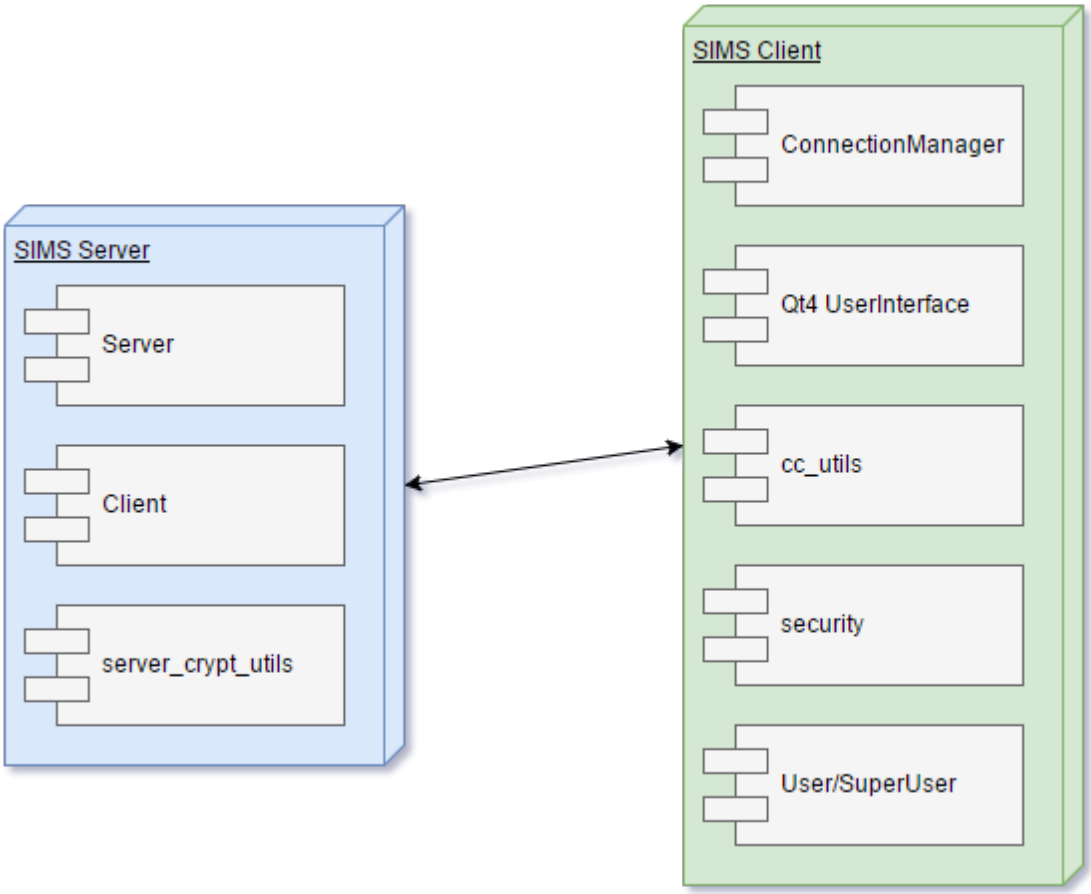
The project goals, besides building an Instant Messenger, are the following:

- Message Confidentiality, Integrity and Authentication
- Destination Validation
- Identity Preservation
- Information Flow Control
- Multiple Cipher Support
- Forward & Backward Secrecy
- Participant & Conversation Consistency

We use Python 2.7 64bit, the cryptography.io, PyKCS11, PyOpenSSL Python libraries and PyQt4.

2. Architecture

The basic system architecture features a simple clients-server system.



Server/Server

Adding/Deleting Clients

Handling Client-Client Information Flow

Receiving and Sending Messages through sockets

Server/Client

Associates buffers, socket, id, cryptographic values and security clearances

Handles releasing its buffers to the server's socket IO

Handles security clearances for Bell Lapadula Information Flow policy

Server/server_crypt_utils

Anything relating to the server's part of dealing with certificates and key exchanges

Client/ConnectionManager

Add/Deletes Client peers
Buffers simultaneous chats
Sends and verifies peer & server acknowledgements
Handles peers & client handshakes
Sends messages to the user interface

Client/UserInterface

Asks for a IP:Port address for the server
Asks for a preferred Cipher Suite
Asks whether the user wants to authenticate with the Portuguese Citizenship Card (CC)
Lists the server's list of connected clients
Says if a peer is sending the user messages
Sends messages to the ConnectionManager module
Displays our sent messages in red at first
Displays our sent messages in green after the associated ack is received

Client/cc_utils

Used to read the Citizen's information in the CC
To get the CC certificate
And to sign data with the CC's private key

Client/User&Superuser

User associates a Name, Cipher Suite, Connection phase state, a message buffer, a waiting ack buffer and cryptographic key.
Superuser extends User with CC certificates, keys and signatures.

Client/Security

Responsible for creating & verifying HMACs
Symmetric AES encryption & decryption
Symmetric key derivation
Verifying CC certificates and extracting its information
RSA signatures & verification
RSA encryption & decryption
ECDH generation of shared secrets and key pairs
Forming NONCEs

3. Messages

In this section we enumerate the different types of json structures.

3.1 CONNECT

Message type used to initialize a new session between client and server.

```
{
"type": "connect",
"phase": <int>,
"name": "client name",
"id": <message id>,
"ciphers": [<cipher combination to use>, ]
"data": <JSON or base64 encoded if binary (optional)>
}
```

3.2 SECURE

Message type used to encapsulate data sent securely between client and server upon their initial connection.

```
{
"type": "secure",
"sa-data": <JSON or base64 encoded if binary>,
"payload": <JSON Sub Message>
}
```

3.3 DISCONNECT

Message type used to disconnect the session between client and server.

```
{
"type": "connect",
"name": "client name",
"id": <message id>
}
```

3.4 Encapsulated Messages

These messages are sent between the client and server, using the secure session already established, encapsulated as a payload of a SECURE message. Therefore, these messages are never transmitted in clear text.

3.4.1 LIST

List clients connected to the server.

```
{
"type": "list"
```

```
"data": [{client-data}, ...]
}
```

3.4.2 CLIENT-CONNECT

Message used for a client to start a secure session with another client.

```
{
"type": "client-connect",
"src": <id_source>,
"dst": <id_destination>,
"phase": <int>,
"ciphers": [<cipher combination to use>, ],
"data": <JSON or base64 encoded>
}
```

3.4.3 CLIENT-DISCONNECT

Sent by a client in order to tear down the end-to-end session with another client.

```
{
"type": "client-disconnect",
"src": <id_source>,
"dst": <id_destination>,
"data": <JSON or base64 encoded>
}
```

3.4.4 CLIENT-COM

Message sent between clients with an actual text message to be presented to users.

```
{
"type": "client-com",
"src": <id_source>,
"dst": <id_destination>,
"data": <JSON or base64 encoded>
}
```

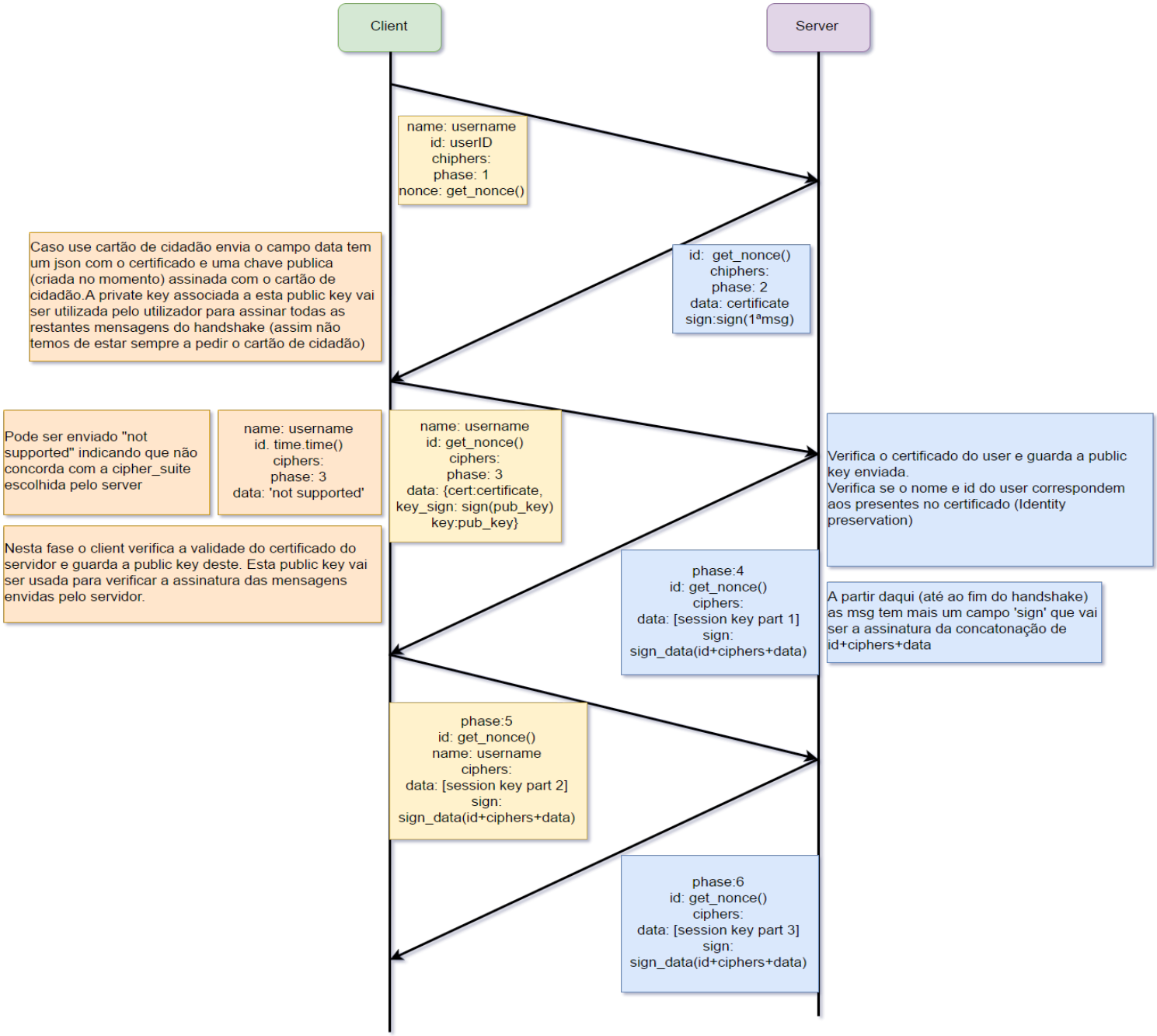
3.4.5 ACK

Acknowledgment message sent after every other message is received to the peer.

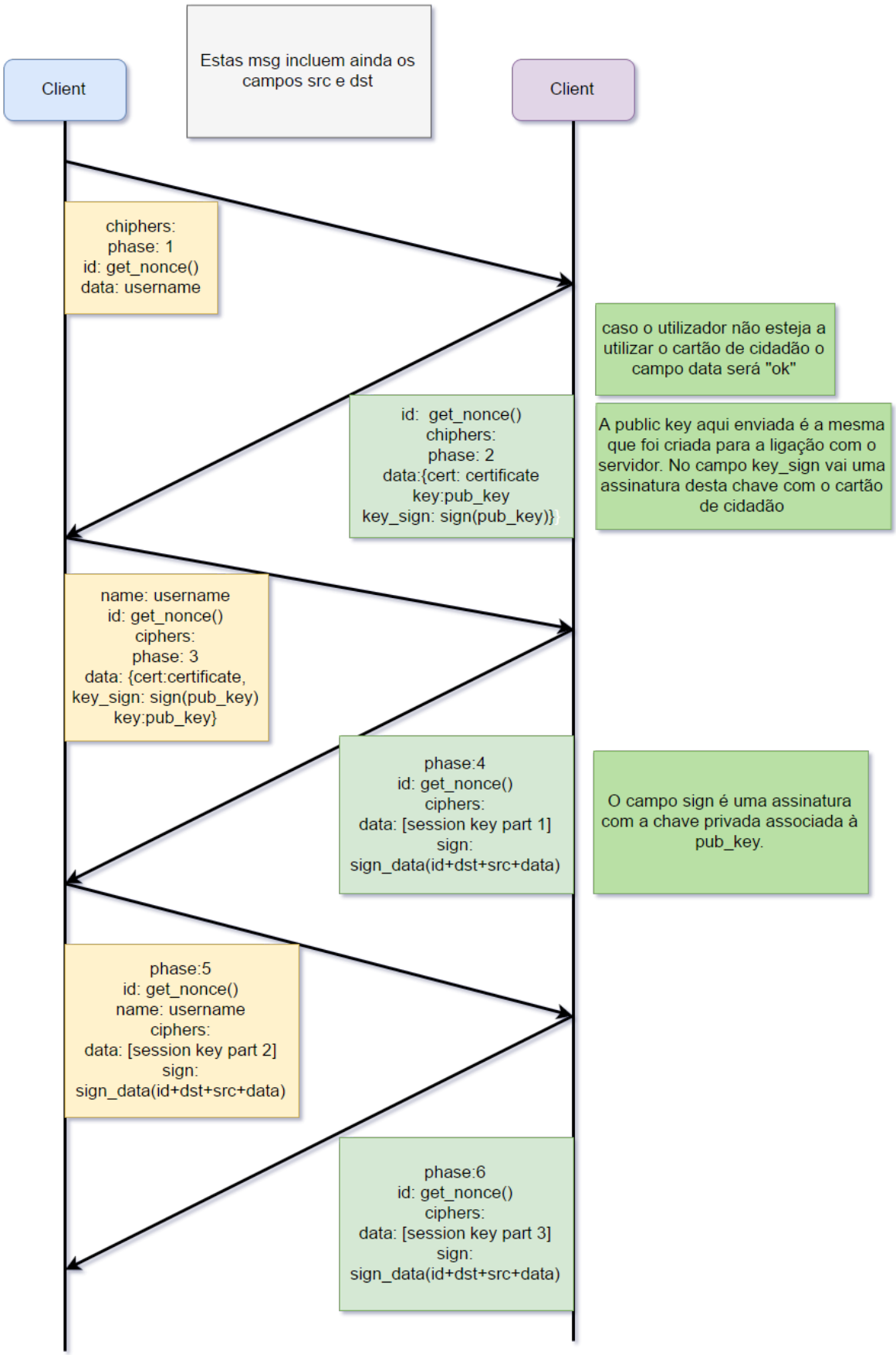
```
{
"type": "ack",
"src": <id_source>,
"dst": <id_destination>,
"msg_id": <msg_id>
}
```

4. Processes & Clearances

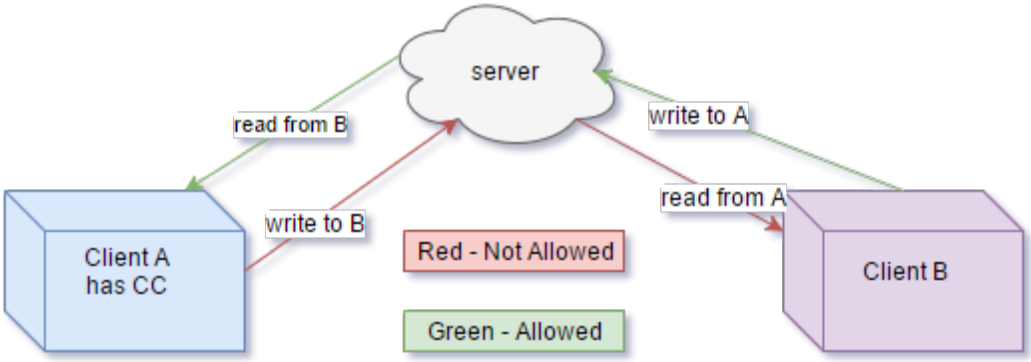
4.1 Client Server Handshake



4.2 Client Client Handshake



4.3 Server Information Flow Policy



Our server, relating to IFP has two hierarchical levels: users authenticated with the CC and users without. The server takes note of which one has a valid CC in the server.py/Client class. That class has two methods which recieve another server.py/Client instance named *permission_to_write* and *permission_to_read*, where both return booleans.

5. Cryptographic Security

5.1 Supported Cipher Suites

Our SIMS implementation supports two secure cipher suites: "RSA_WITH_AES_128_CBC_SHA256" and "ECDHE_WITH_AES_128_CBC_SHA256". They differ in the kind of key exchange protocol is used. Bulk symmetric encryption and HMACs are exactly the same. It supports different cipher suites between Client-Server and Client-Client. The suite is negotiated during handshakes.

5.2 ECDH Key Exchange

Our security module allows for ECDH key exchange. We have a Key Pair generating function and a Get Shared Secret function. We use a NIST P-384 elliptic curve to generate the key pairs. To generate the session key we apply a concatenation key derivation function to bring the key to the desired size.
The key exchange is similar to 5.3, but each peer generates an ephemeral key pair and sends the public component to the other peer. After receiving the public key from the peer each client will generate the shared secret using his private key and the received key. The key exchange is explained in 4.1 and/or 4.2. [7]

5.3 RSA Key Exchange

We have functions for Generating and Loading RSA(Private/Public)Key objects. And most importantly Sign/Verify and Encrypt/Decrypt. The key exchange is explained in 4.1 and/or 4.3. For padding we use both OAEP with the masking generation function MGF1 both of these use the hashing algorithm SHA256.

After cipher suite agreement, the server/responder peer generates an ephemeral RSA key pair with 4096 bits and sends the public key to the initiator. The initiator generates a session key, and encrypts it with the received public key, using OAEP with SHA256 for padding, the resulting encrypted text is sent to the server/responder. After the reception of this packet the peer can decrypt its contents with the private key generated earlier. After this step both parties have a shared secret that will be used to encrypt (see 5.4) the rest of the communications in this session. [4]

5.4 AES Encryption & Decryption

We use Fernet from cryptography.io for all symmetric cryptography. Fernet does encryption and authentication. The reason we choose Fernet is to mitigate cryptographic implementation mistakes. It is built with AES128 in CBC mode, PKCS7 for padding and SHA256 for HMAC. The Key Format is (Signing-key ‖ Encryption-key). The Token Format or Cipher Format is (Version ‖ Timestamp ‖ IV ‖ Ciphertext ‖ HMAC). For the HMAC Format is used SHA256(Version ‖ Timestamp ‖ IV ‖ Ciphertext). [3]

Bit length of fields:

Signing-key, 128 bits
Encryption-key, 128 bits
Version, 8 bits
Timestamp, 64 bits
IV, 128 bits
Ciphertext, variable length, multiple of 128 bits
HMAC, 256 bits

5.5 CC Certificate Verification

We verify if the certificate is a valid CC in the trusted chain and minding the CRLs. We downloaded the chain and some CRLs from the CC Website [6]. Using the PyOpenSSL X509Store [1] we easily verify if a certain Certificate is valid under a trusted chain.

5.6 Self-signed server cert

It is created with the server's private key.
We check the validity in a X509Store. [5]

5.7 RSA Signature & Verification

All our signatures use PSS for padding and SHA256 hashing algorithm except the ones made with the CC in which it is used PKCS1 and SH1. We use cryptography.io's classes RSAPrivateKey with the method *sign* to sign and the RSAPublicKey *verifier* to verify a signature. [4]

5.8 RSA Encryption & Decryption

We once again use the cryptography.io library, this time we use the methods *encrypt* and *decrypt* of the RSAPublicKey and RSAPrivateKey classes, respectively. [4]

5.9 Symmetric Key Derivation

Using the cryptography.io library we call the PBKDF2HMAC class using for the *'salt'* argument a alphanumeric NONCE denoted *s_k*. Then we call the method *derive(Current_Key)* of said class to obtain the derived key associated with such *s_k*. [2]

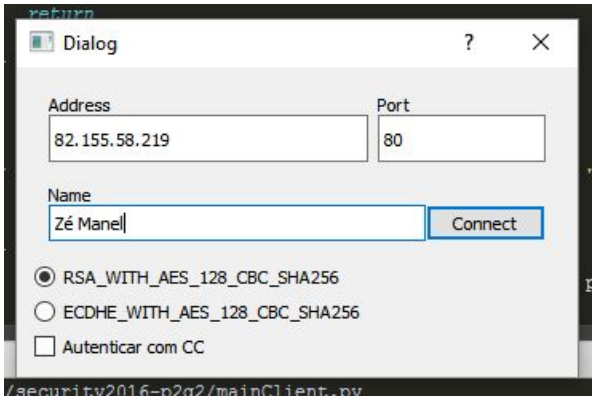
5.10 Generating NONCEs

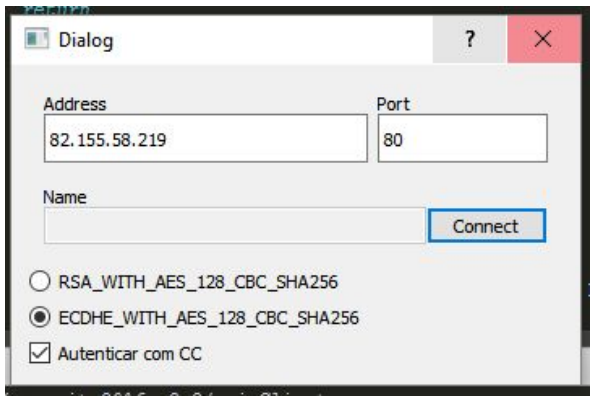
NONCEs are generated using the randint() function of the random.SystemRandom() class. We pick a number between zero and nine 16 times. Our NONCEs are a string of length 16 made up of only numeric characters.

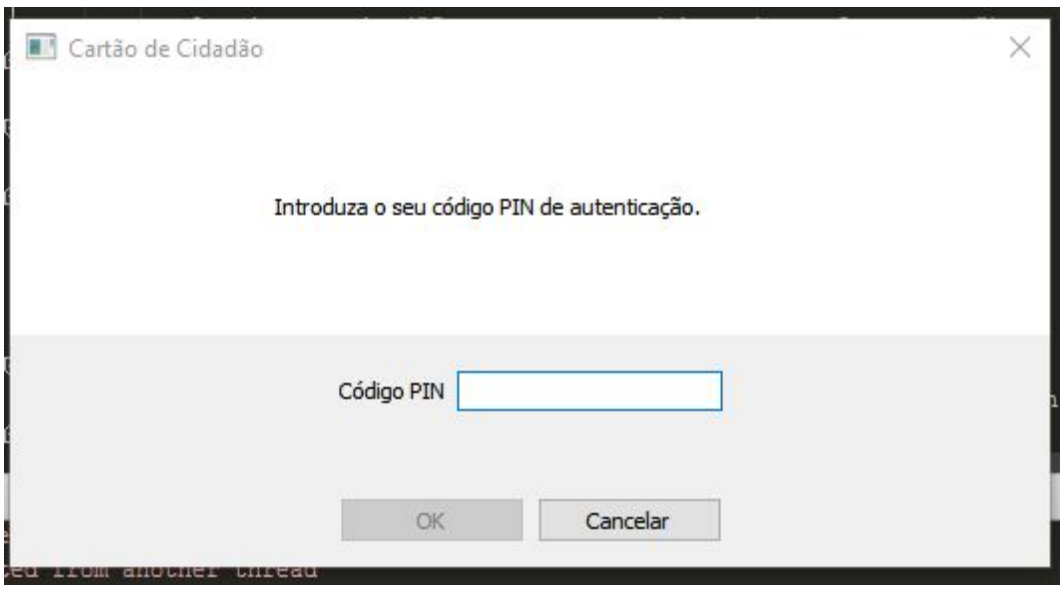
6. User Interface

6.1 Dialog Box

The SIMS Dialog Box is the first thing the user sees. It asks for an *IP* and a *Port* to connect to and a *preferred cipher suite*.
If the user has the CC Software installed and their CC connected to the computer it allows for the *"Autenticar com CC"* checkbox to be enabled. If the user wishes to authenticate with their CC they then check that box, otherwise they don't. If they do, the *"name"* text box becomes disabled, because the *"name"* will be the name present in the CC. Also, in authenticating with the CC, another Dialog Box will pop up asking for the Auth PIN of that CC.

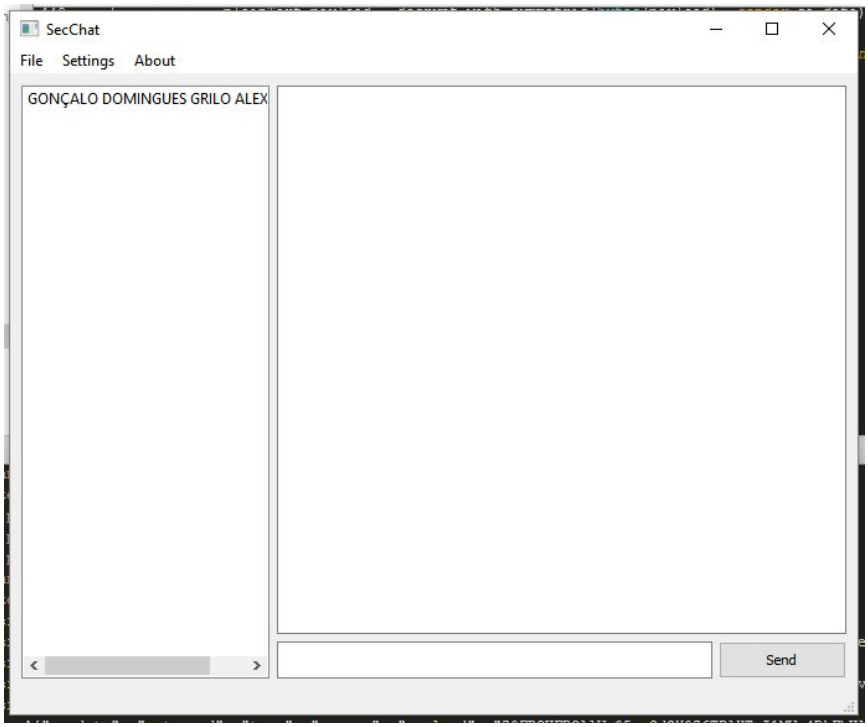


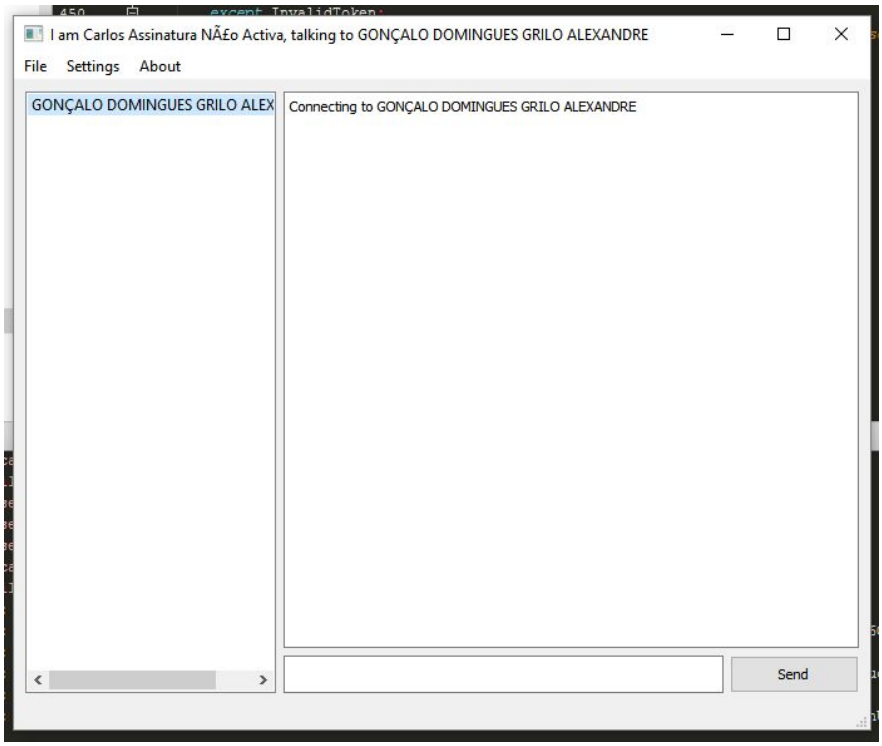




6.2 Connecting to a peer

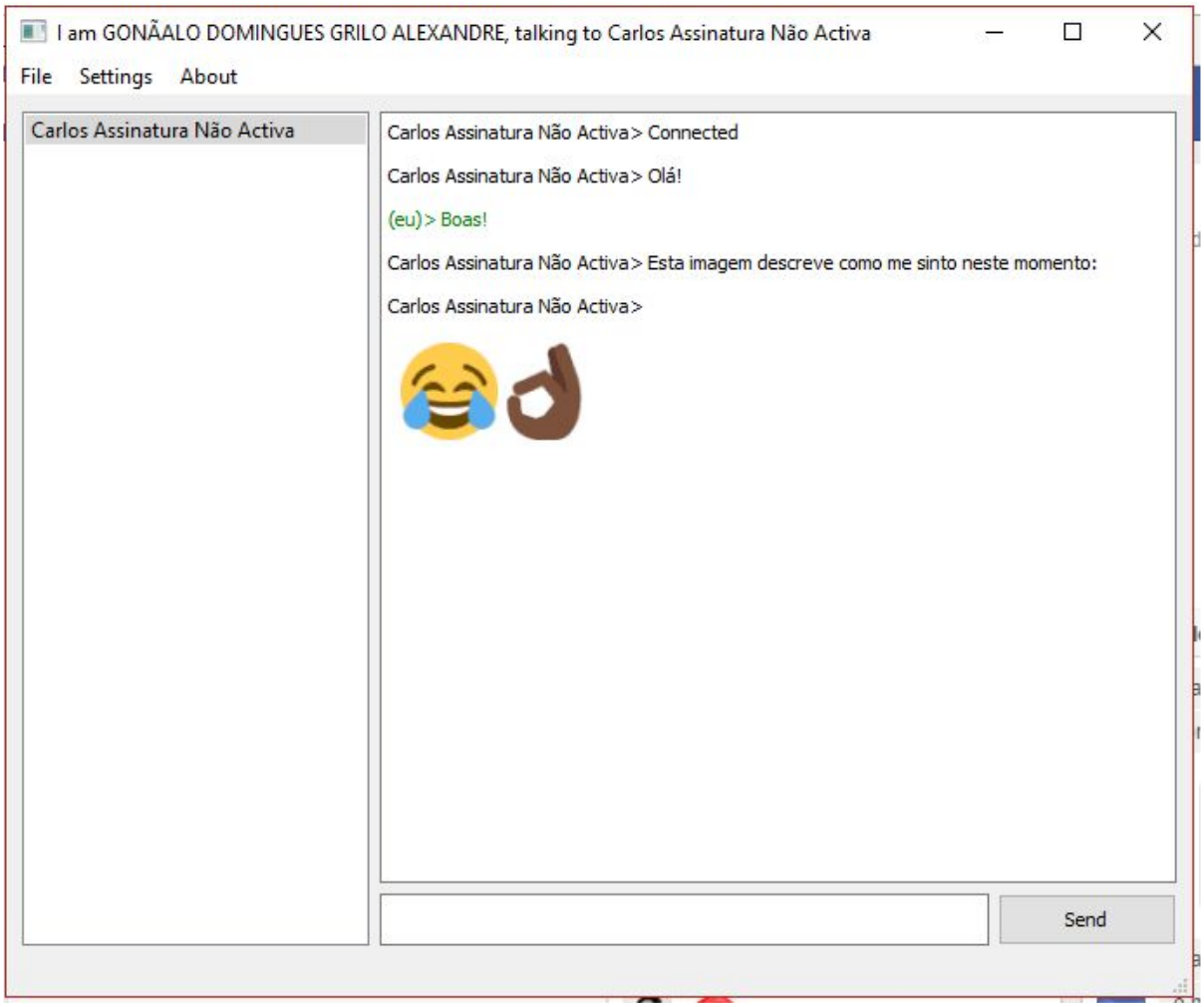
After a successful connection to the server the user want to talk with someone. A list of connected users appears on the left. A user double clicks in a peer name to connect.

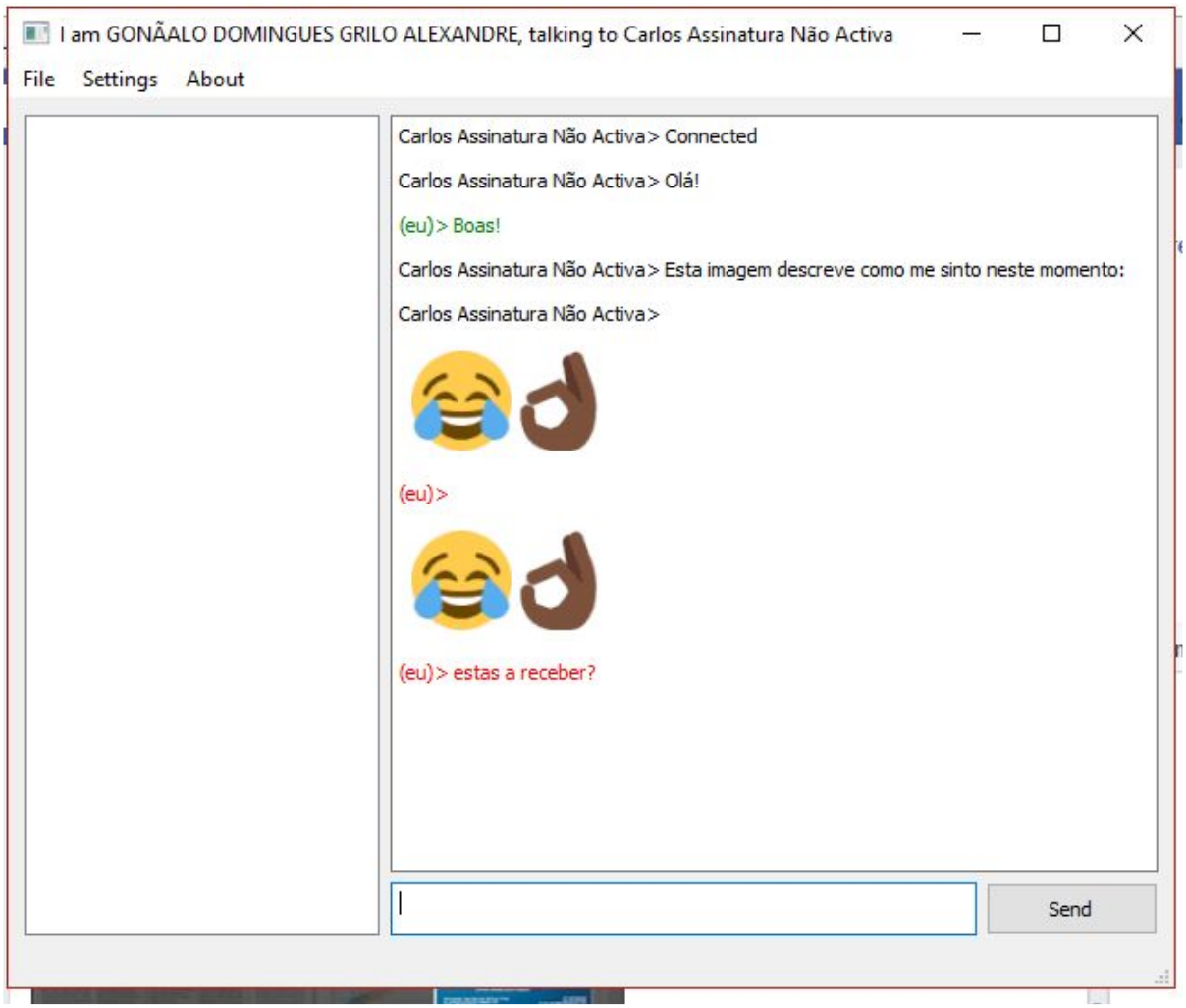




6.3 Messaging

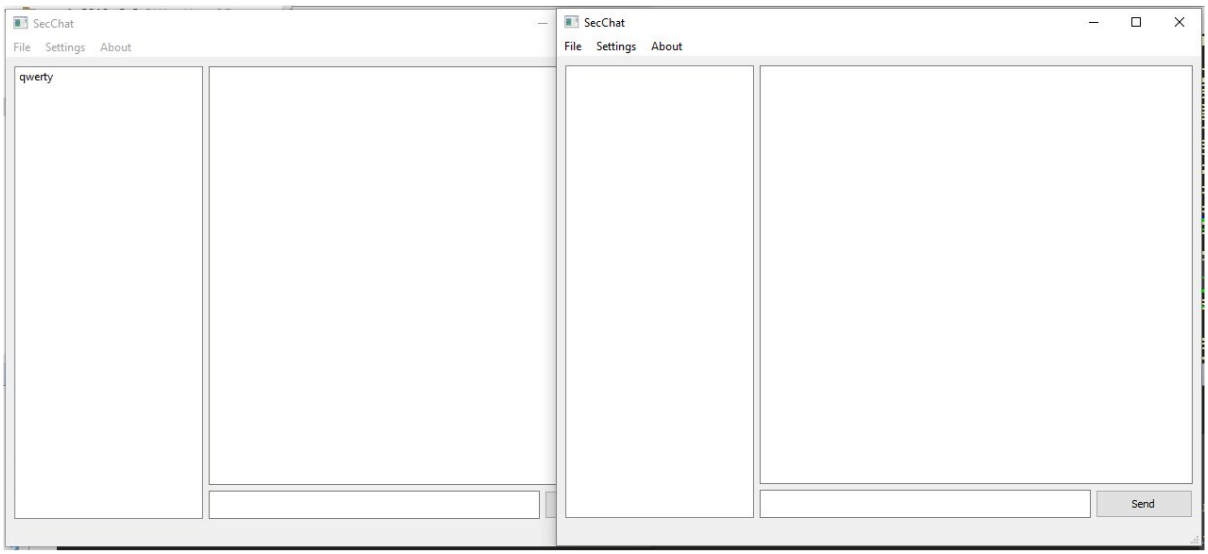
Each line in the message box represents a message. Before the actual typed message appears the name of the peer or *(eu)* if it's a message sent from us. The peer's messages appear in black print. Our sent messages appear in red until the respective ACK is received, at which point they turn green in print. The system also supports an image displayer, by typing a specific message(*/top*), the user interface loads and presents a local client image. This could be easily extended to fully support secure emojis.





6.4 User Hierarchy & IFP

The qwerty user (on the right) didn't authenticate with the CC. The user on the left did. That follows the policy described in 4.3.



7. Needs & Features

7.1 Message Confidentiality, Integrity and Authentication

This is a M1 requirement. Accomplished and described in M1 delivery.

7.2 Destination Validation

This is a M2 requirement. But accomplished and described in M1 delivery.

7.3 Identity Preservation

Every time a client connects we check the validity of his CC by checking his certificate. To make sure he is in possession of the card we receive a public key created by him, this public key is signed with the CC. All following handshake messages will be signed with the private key associated with the previously sent public key. this way we can be sure that we are making the handshake with the owner of the CC. In the handshake the users trade a session key that is used for secure communications therefore completely preserving identity as well as secrecy and integrity.

This is a M2 requirement. How we achieve this is described in sections 4.1, 4.2, 5.5, 5.6

7.4 Information Flow Control

This is a M2 requirement. How we achieve this is described in sections 4.3 and 6.4

7.5 Multiple Cipher Support

This is a M1 requirement. Accomplished and described in M1 delivery.

7.6 Forward & Backward Secrecy

This is a M1 requirement. Partially accomplished and described in M1 delivery.

7.7 Participant Consistency

This is a M2 requirement. Not implemented.

7.8 Conversation Consistency

This is a M1 requirement. Described in M1 report.

8. References

[1] https://media.readthedocs.org/pdf/pyopenssl/latest/pyopenssl.pdf
[2] https://cryptography.io/en/latest/hazmat/primitives/key-derivation-functions/
[3] https://cryptography.io/en/latest/fernet/
[4] https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa
[5] https://cryptography.io/en/latest/x509/
[6] https://pki.cartaodecidadao.pt/
[7] https://cryptography.io/en/latest/hazmat/primitives/asymmetric/dh