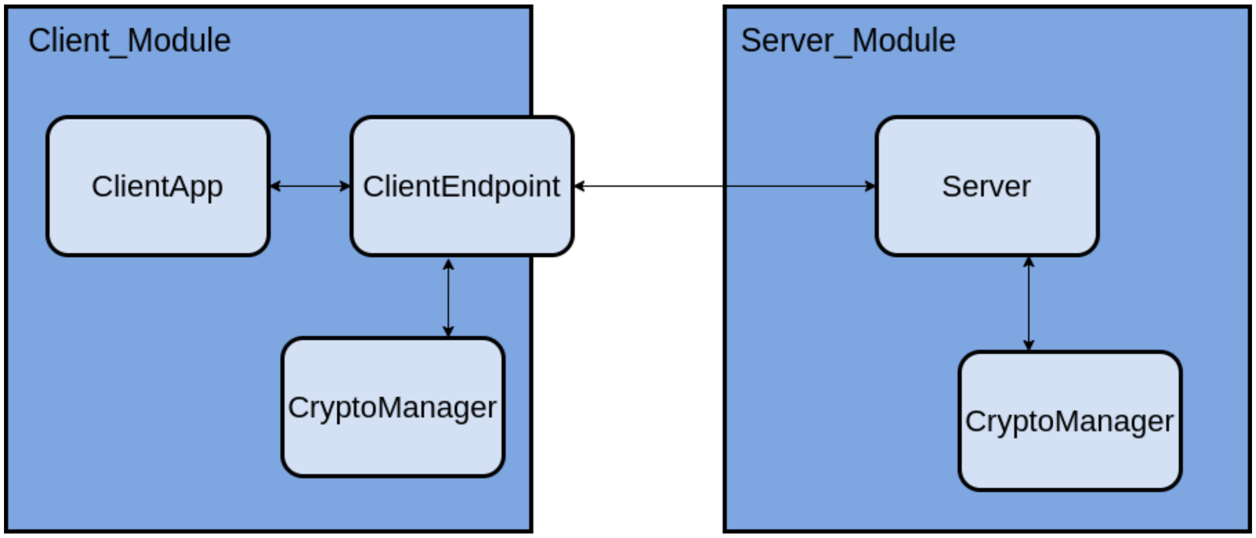
Project Report - Stage 1
Dependable Public Announcement Server

Miguel Grilo - 86489 Simão Nunes - 86512 Miguel Francisco - 88080

1. Architecture Of The System



The **API** requested documents the communication between the **ClientEndpoint** and the **Server**. That communication is implemented with sockets. We modified the API because there were some inconsistencies with the operations, their arguments and the way we wanted to guarantee integrity. For example in reads, we added the public key of the user making the request. The **CryptoManager** contains all cryptographic methods.

2. Message Exchange Protocol

For each operation requested, there is a handshake, to decide both Server and Client Nonces (randomly generated) to guarantee freshness of this operation. From the server side, those nonces are identified by the public key of the user making the request.

Then, the user sends the operation with both nonces appended and the server sends the response with the client nonce appended. Adding up to it, with verifications of nonces in both sides, we are protecting our system from replay attacks, from all possible messages, by throwing specified exceptions when the nonces are different. In this protocol we also implemented timeouts from the client side to protect from drop attacks. If the drop is done after the handshake, the client is advised to verify if his operation was successful, because there is no way to know if it was the request or the server response that it was dropped.

The messages exchange specified above are possible thanks to three objects defined along the process: **Request**, **Response** and **Envelope**.

Whenever a client wants to request a certain operation to the server, it will send a Request object with the proper details specified in its attributes. After this, the server will respond with a Response object, which has a similar structure to the Request.

In both cases, each object is injected inside an Envelope, which will have the original Request/Response and the corresponding ciphered hash to prevent integrity attacks, explained in the next section.

3. Integrity

Our system guarantees integrity in all messages where it is needed by using, as referred before, the object envelope, which contains 2 attributes:

- Request / Response .
- The same Request/Response but **hashed** and **ciphered** in the following way: $\{ H (R) \}_{Pr}$ **,** where **H** is our hash function (SHA2), **R** the request/response which includes the nonces and **Pr** the **private key** of the entity sending the message, which provides **non-repudiation** from both the server and the user.

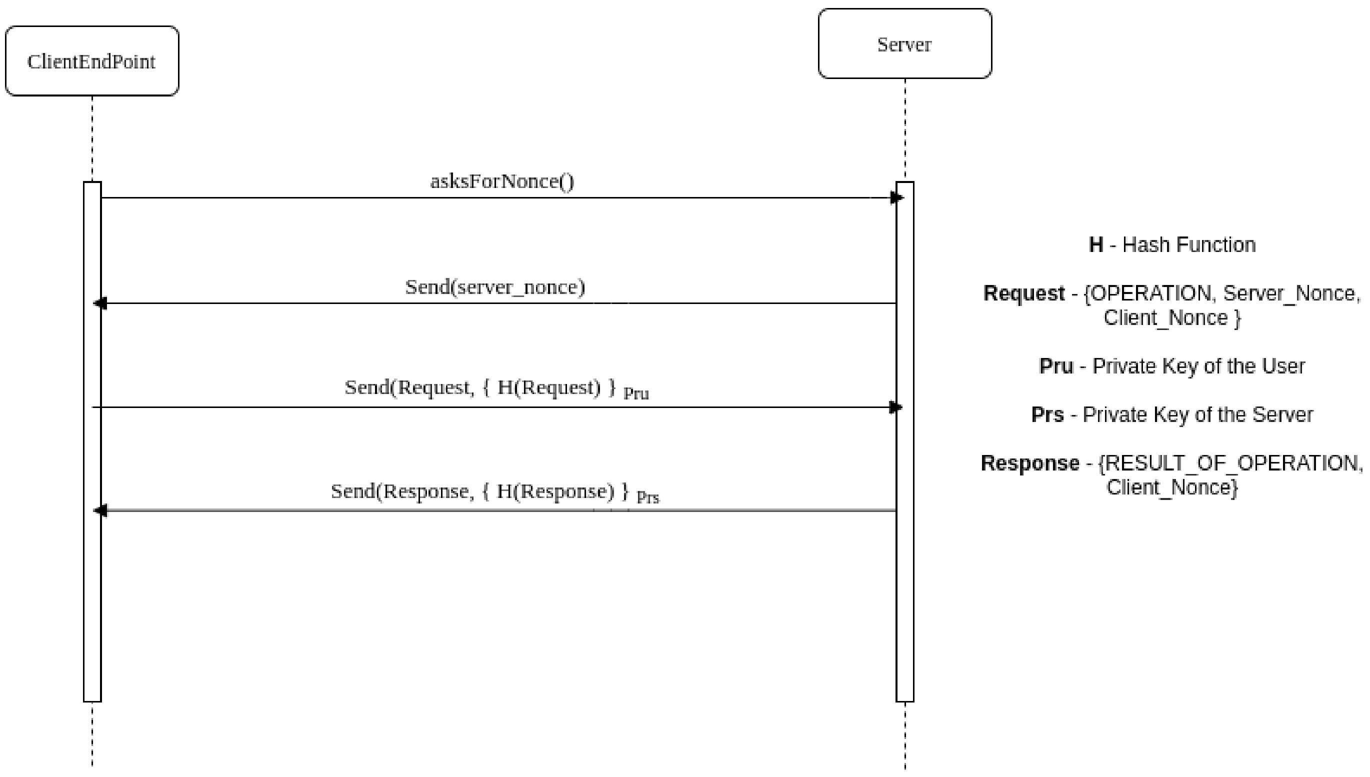
Both entities when receiving a message, they verify the envelope by deciphering with the public key of who sent it, hashing the open Request/Response and verifying if the hashes are the same:

$\{ H (R) \}^{-1}_{Pu} == H (R)$ **,** where **Pu** is the public key of the sender.

There is only one operation that doesn't check the hash which is the **readGeneral** operation. Everyone can register in the system and everyone can read from the general board everything they want so there is no point in integrity attacks to this operation.

We also guarantee **integrity** of the posts and the registered users by having a **persistent local file system** on the server. Even if the server shuts down for no reason at any moment of the execution, the user has a timeout which will expire and return the appropriate exception message to him. After rebooting it will still have all the persistent information.

All of our protocol can be viewed in this sequence diagram:


ClientEndPoint
Server
asksForNonce()
Send(server_nonce)
Send(Request, { H(Request) } Pru
Send(Response, { H(Response) } Prs
H - Hash Function
Request - {OPERATION, Server_Nonce, Client_Nonce }
Pru - Private Key of the User
Prs - Private Key of the Server
Response - {RESULT_OF_OPERATION, Client_Nonce}

Our application design guarantees that won't happen any catastrophic events to the user registered or to his posts while connected to the system. He will never be unregistered or have his posts deleted. Therefore we guarantee **Safety**.

We cannot guarantee **availability** because in a certain instant of time the server can be down and the connection is refused, making the user unable to request any operation. Although, we guarantee **Reliability** because our application assures that over time, that user will be able to execute that operation once refused, even through drops and replay attacks. Our design delivers suggestions to the user to confirm certain operations.