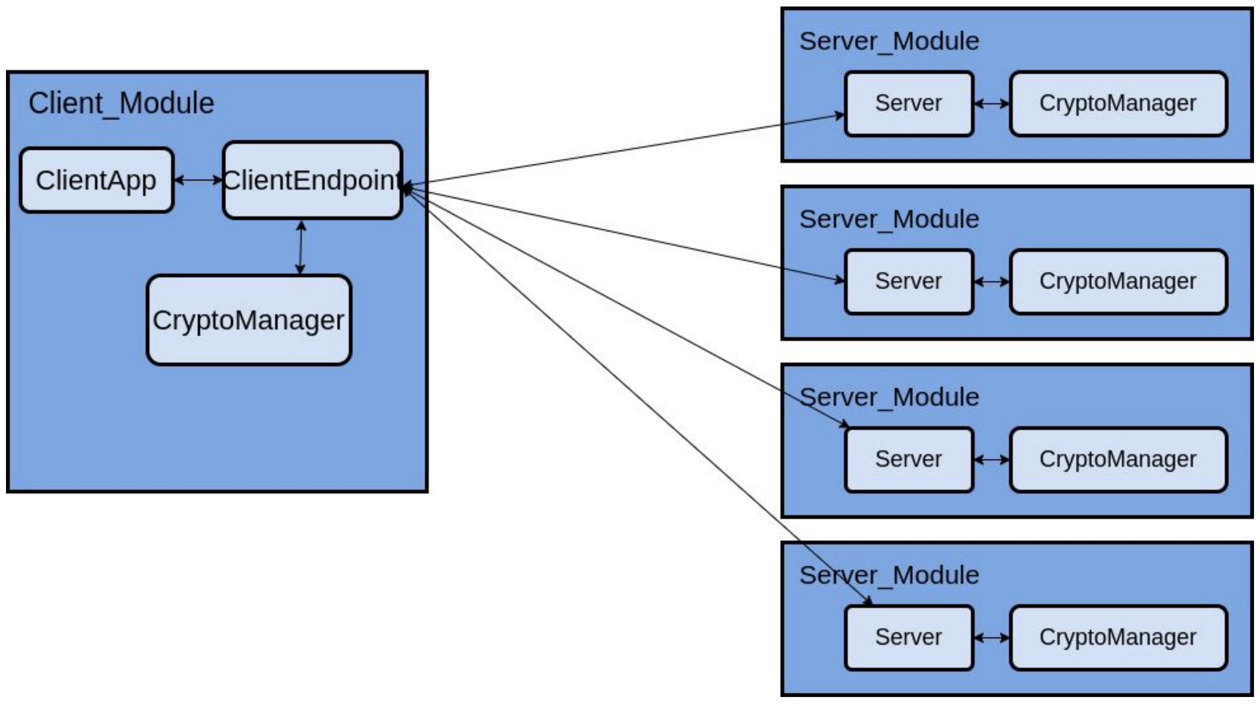
Highly Dependable Systems - 2019/2020

Project Report - Stage 2
Dependable Public Announcement Server

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

1. Architecture Of The System



The **API** 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, there is always a **handshake**. If the operation includes a request and response, both **Server** and **Client Nonces** are generated. Otherwise, only with one request/response, only one nonce is generated (128 randomly generated bits using Java Secure Random) to guarantee that this message was not seen before.

These nonces are identified in both modules by the PublicKey of the process (client or server).

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 specific exceptions when the nonces are different.

In this protocol we also implemented timeouts from the client side to **protect from drop attacks**.

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 inserted inside an Envelope, which will have the original Request/Response and the corresponding signature to **prevent integrity attacks**, explained in the next section.

3. Integrity in the Communication

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

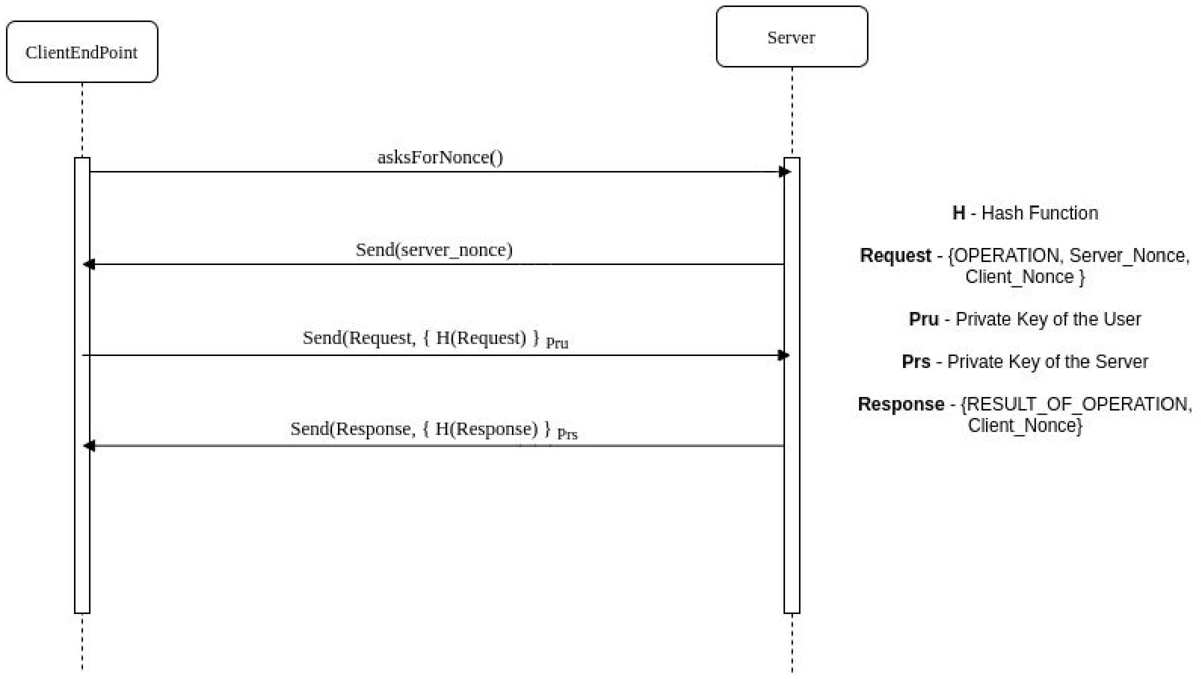
- Request / Response .
- The same Request/Response but **signed** in the following way: $\{ H (R) \}_{Pr}$ **,** where **H** is our hash function (SHA2), **R** the request/response which includes the nonces and **Pr** the **private key** of the entity sending the message, which provides **non-repudiation**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

Both entities when receiving a message, they verify the signature with the public key the sender:

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

We also guarantee **durability** 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.

An example 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 guarantee **availability** for f faulty server (in our example 1 faulty server) by implementing the algorithms that are explained in the next section. We also 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 by suggesting the user to confirm certain operations when a timeout expires.

4. Post / Read operations

For each user's announcement board we wanted to guarantee atomicity of the post operation. This means, when you are reading from an Announcement Board that is currently being written by the respective user, you can either read the new updated

Board or the one right before the write operation. To achieve that specification we implemented the suggested **(1,N) Byzantine Atomic register** with quorum listeners.

5. PostGeneral / ReadGeneral operations

For the general board we initially implemented the **(1,N) Byzantine Regular register** which allows multiples readers but only one writer. We than transformed it to the **(N, N) extension** that allows multiple writers:

- Implemented a read operation of the latest write timestamp before every write operation.
- Allowed writes with the same timestamp, in the case that they are concurrent.
- Implemented a tie break function to decide which write is the most recent, in spite of both having the same timestamp.

Our tie break function uses the username as rank but the system is ready to receive any other function.
This function is applied at the writing moment at the server. The posts with the same timestamp are reordered under the tie break and saved so that the reading operation stays the same.

Our implementation allows to maintain the same causal order of posts in the general boards of the different replicas. Since our tie-break function is deterministic, if all replicas receive all postGeneral operations then all replicas will have the same order.

6. Byzantine Clients

We assume clients may be malicous and change the **Endpoint** library to their benefit. If so, our system will neglect messages that are nonsense with the proper message's fields checks.
To cope with the fact that a byzantine client can send an operation to **only a partition of servers**, compromising the internal state of the different boards in the different replicas, we implemented the **Authenticated Double Echo Broadcast** that guarantees **consistency** among correct processes. Even the messages sent in this protocol (ECHO / READY) are preceded by an handshake and a sign/verify to guarantee resilience to integrity and replay attacks.