Grupo 9: Afonso Ribeiro - 89400, Duarte Santos - 89438, Sofia Carvalho - 89539

Highly Dependable Location Tracker

Highly Dependable Systems

The goal of this project was to develop a Highly Dependable Location Tracker that operates on a Client-Server architecture with multiple clients running simultaneously.

We chose to implement our application with RESTful services, using Spring Boot.

To implement a notion of time inside our project, we split time into epochs and designed, inside every client, a "step" command which increases the current epoch. This command can be called by any client and is propagated to every client, so that the current epoch is equal amongst all clients. Each user automatically requests the proofs regarding its current epoch before increasing it. As this is a simplification of the real world, we assume that the propagation of this command is always successful.

- **Threats and Protection Mechanisms:**

To assure secure communication, a key pair for each user, server and health authority is generated before running the project. These are saved in separate key stores, each belonging to a single entity. This also guarantees a certain degree of persistence, as keys remained stored even if a failure occurs.

- **Confidentiality, Integrity and Non-Repudiation:**

Concerning the channels between two clients, we assumed that the messages did not need to be confidential.

Regarding the channels between a client (including HA) and a server, we assumed that an attacker could read, drop, reject, manipulate, and duplicate messages and, therefore, needed to assure confidentiality, integrity and non-repudiation while using this channel.

To achieve this, every message sent is encrypted with a symmetric key. The keys between client-server are refreshed every few epochs, and between server-server before each broadcast. The message is also signed using the sender's private key. The generated secret is then encrypted with the receiver's public key and sent along with the encrypted message and signature.

For increased security when receiving messages from the server, we send information about an original request along with the server's response, so that it is possible for the client to verify that the response corresponds to what was originally requested.

- **Forged Location Proofs and Reports:**

Furthermore, there is the problem of having an attacker (or a byzantine user) submitting reports with false proofs.

To mitigate this problem, a witness signs every proof that they send so that it cannot later be altered or repudiated – the proof contains the witness's information about the 'epoch', 'location', 'witness' and 'prover'. Furthermore, they check if they are effectively near the prover. Every time a report is submitted, the client (sender) discards any proofs that have a different 'epoch', 'location' or 'prover' than the ones alleged in the report. It also discards repeated proofs, proofs with a wrong signature, and proofs witnessed by the original prover.

When the request is delivered on the server, the server then checks if it contains any invalid proofs and, if it does, the server immediately discards the request, as that means that the client that sent that same request is byzantine. Finally, the server only accepts the report if it has a minimum of F+1 valid proofs (F is the number of tolerated byzantine users).

- **Freshness:**

To overcome the challenge of assuring the messages' freshness to avoid replay attacks, a unique nonce, is present in each message. If a message is replayed, this nonce would be outdated (compared with the last received nonce, for that sender), and consequently that same message would be deemed as not fresh.

- **Server Persistence:**

Finally, to assure the integrity of the server's data, we are using a MySQL database, which guarantees its persistence even in the case of server failure. Along with this measure, there is also the previously mentioned key stores, that guarantee the persistence of relevant keys.

- **Anti-Spam Mechanism:**

To prevent DoS attacks, we implemented a mechanism based in Proofs of Work. The clients must compute a nonce that, when added to the message, creates a hash with N leading zeros. This computation is easy to verify by the server but must be computed with brute-force by the client, therefore creating a valid proof of work by the client. The computation increases in complexity with the number of leading zeros. We chose only 2 leading zeros (8 in binary) to make testing easier (this parameter can be easily changed).

This mechanism is used in all the client's requests, since they are expensive - they trigger broadcasts and accesses to databases.

- **Server Replication and (1-N) Registers:**

When replicating the server into multiple replicas, there appeared the need to synchronize data between all the replicas, which we achieved using 1-N Byzantine Registers (Both Regular and Atomic). By using these models, we also protect against Fs possible byzantine or crashing servers, as the conclusion of an operation relies on a quorum that depends on Fs. The implementation is as defined in the course slides.

- **Byzantine Reliable Broadcast:**

There is also the possibility that the byzantine clients interfere with our replica synchronization by sending different reports to the servers. To tolerate this, we implemented the Byzantine Reliable Broadcast (Authenticated Double-Echo) which assures that if a message is delivered to a correct server, then every correct server also receives it, even if the sender is faulty.

Finally, there is the possibility that a byzantine server (S) allies with a client (C) to perform a joint attack. By doing so, they could try to store an invalid report. However, when S broadcasts a write to other servers, they always check if a report is valid before storing it. Thus, they can only go as far as storing it in S's database - which will not have consequences as every read requires the said quorum of equal server responses.

There is one attack that we are not currently protected against - C is able to access the reports of other clients from S. To protect against this, we could have the servers store the reports still ciphered. This way, C would not be able to read them, nor would S. This would be a possible future improvement to our project.