Highly Dependable Location Tracker

MEIC-T | Group 25

Stage 2 Report

André Augusto 90704
Lucas Vicente 90744
José Oliveira 87678

May 2021

1 Solution

1.1 Primitives

- **Location Proof**: *(proverId, ep, proverLat, proverLng)* signed by a nearby witness.
- **Location Report**: *(provers location + the collection of correct distinct proofs)* signed by the prover.

1.2 Assumptions

- Future report submissions for a certain epoch and proverId matching a previously submitted valid report are deemed invalid.
- Correct users make a request at a time.
- Correct users do not prove fake locations.

1.3 Proximity Service

The proximity service handles all client-to-client interactions, such as requesting and responding to location proof operations. A user which intends to proof their location for a certain epoch, called a prover, must collect f distinct correct proofs, from nearby users or witnesses. Only then, with *f witnesses + 1 prover* users vouching for the same location, is the report considered legitimate.

1.4 Location Service

The location service handles all of the interactions client-to-server, being submitting and obtaining location reports, and obtaining lists of users that were in a specific location at an epoch.

Both services guarantee integrity, authenticity and non-repudiation with the use of digital signatures. On top of that, the location service also guarantees confidentiality with the use of hybrid cryptography AES-256 + RSA-2048.

2 Addressing Design Requirements

2.1 Byzantine Servers

Since provers already sign every report and each proof in the report is signed by the appropriated witness, a byzantine server can not tamper with a report. When obtaining a location report for a certain epoch, the user waits for a byzantine quorum of valid responses and, due to atomic semantics, writes back to a byzantine quorum of servers *(> (n + f) / 2)* whenever the reponses do not match. Both *ObtainUsersAtLocation* and *RequestMyProofs* operations guarantee regular semantics, therefore clients wait for a byzantine quorum of responses *(> (n + f) / 2)* and perform a union of distinct valid reports or proofs received, respectively, in both cases proofs received by themselves or in a report must be signed by a byzantine quorum of servers.

2.2 Byzantine Clients

Byzantine clients can also have a considerable impact on our system thus it is necessary to cope with their malicious behavior. For example, a client might send different valid report submissions to different servers, or only send reports to a certain subset of replicas. To handle these issues we implemented the *Authenticated Double-Echo Broadcast* algorithm, between server replicas. This algorithm consists of two phases, which provide not only detection of byzantine user invalid report submissions, with the *ECHO* phase, but also ensuring that if a process delivers a valid report r, then every correct process will eventually deliver the same report r, with the *READY* phase. However *ADEB* does not guarantee that faulty servers will not deliver a different report, meaning that a concurrent reader might end up receiving two valid reports proving the same location and epoch, but with different proofs, and from the perspective of the reader it is not able to validate which report was actually delivered through *ADEB*. In order to tackle this issue, before sending a *READY* message, each server will sign each proof of the report, and after collecting a byzantine quorum of *READYs* a server is able to deliver a report containing proofs signed by all the servers in that byzantine quorum.

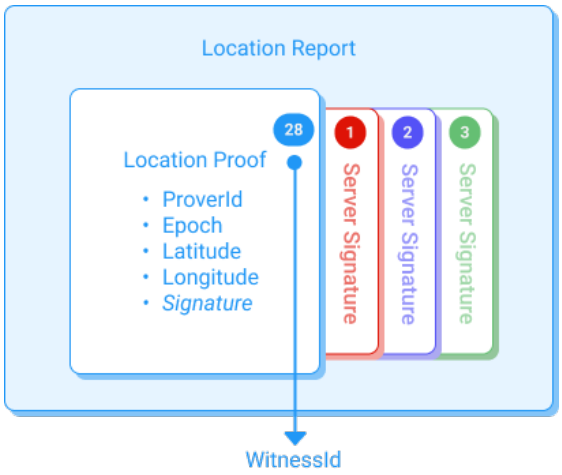

Figure 1: Valid report containing a proof signed by a BQ of servers.

2.3 Spam Combat Mechanism

Report submissions, both when performing a *submitReport* and when writing back after an *obtainLocationReport*, must assure atomic semantics and byzantine client protection, consequently requiring not only thread-protected save of state, but also *ADEB* to detect, correct and validate any byzantine client submission. With this in mind, we consider this operation significantly more expensive than the rest. In order to com-

bat the spam of report submissions, we've implemented proof-of-work, consisting of the generation of a hash prefixed by a configurable number of zeros, adjustable to the required cost, with a nonce suffixed to the contents of the report submission request. Due to the *preimage resistance* of hash functions, this proof-of-work requires a brute-force search of the nonce, on the client-side. With this mechanism we are able to increase the cost of such operation for the client, while keeping the validation of this proof low cost for the server, since the server only needs to calculate the hash with the contents prefixed with the nonce and verify that it is in fact prefixed by the correct number of zeros.

3 Possible Threats and Corresponding Protection Mechanisms

3.1 Drop & Reject Messages

When any request is made, with no response in 1 second, the same message is sent over, until a response is received.

3.2 Manipulate Messages

Since all requests and responses, client-client and client-server, are digitally signed, and verified when received, it is assured data integrity.

3.3 Duplicate Messages and Replays

A prover verifies if any proof received is duplicate and discards them from the report. The server, when receiving a report, will also check the existence of a report for the same user and epoch, if found, it will be discarded and the user is properly notified. We are also using read ids as sequence numbers, in order to guarantee the freshness of query operation requests and responses.

3.4 Eavesdropping

It is guaranteed confidentiality on all client-server and server-server communications, since the messages exchanged are encrypted using an AES-256 key, generated for that specific message. This key, is also encrypted using the public key of the other end, ensuring only the intended receiver is able to decipher the key, and consequently of reading the content itself.

3.5 Server Crashes

Whenever a user submits a new valid report the server saves persistently its current state, which includes user's sequence numbers, reports and system info, as two JSON files, a main one and a backup one. With two files, we assure that in corruption of the main one, the server is able to restore its state through the backup file.

3.6 Private Key Protection

Each entity in the system has its private key stored in a password protected keystore.