

Highly Dependable Systems

Project - Stage 1

Highly Dependable Location Tracker

Daniel Matos
89429

João Soares
89475

Tiago Fonseca
89542

April 2021

1 Design

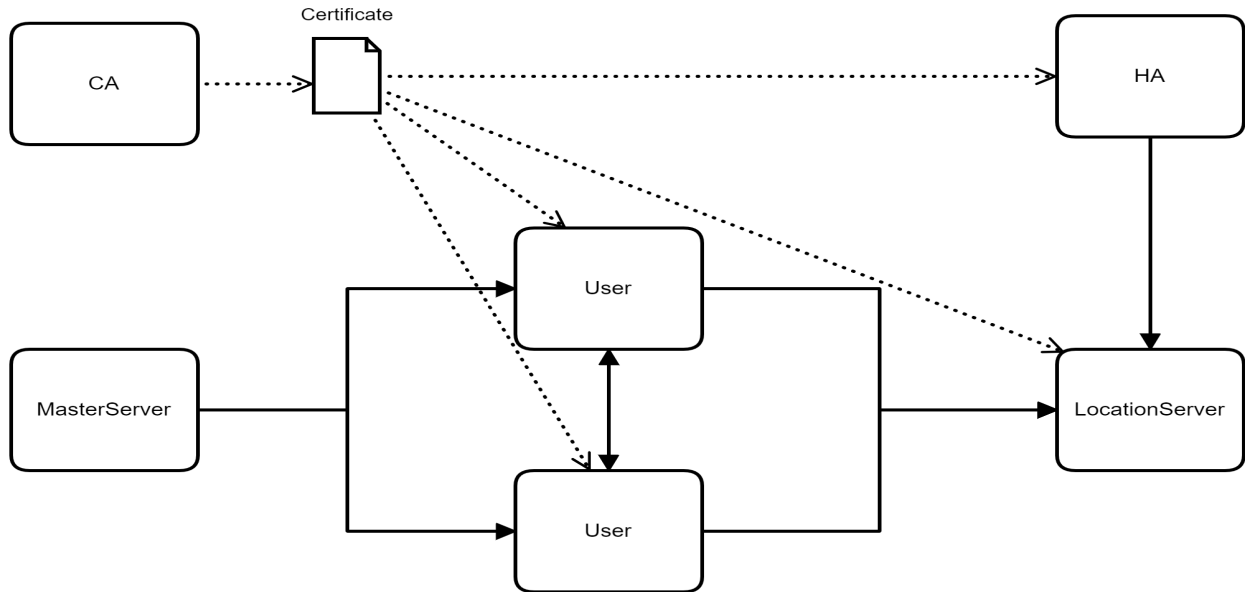


Figure 1: Overall System Design.

Our system relies on a **MasterServer**, that is controlling the grid and the users' movements. It periodically sends this information to all users which will communicate with each other in order to gather proofs. If close enough, they will get proofs and can generate a *LocationReport* that will be sent to the **LocationServer** (or just **Server**) in order to consult it later on. **HA** is the one that can consult any *LocationProof*.

Users can have 6 different levels, being level -1 a "normal" user and 5 the "hardest" byzantine. Users from higher levels can also do what the ones from lower levels do. The levels are as follows:

0. Forge reports with *self-signed* proofs
1. Skip epoch communication
2. Tamper fields in requests
3. Reject other user's requests

4. Redirect requests to other users
5. No information verification

2 Properties Assured

- **Confidentiality**

Between **User/HA** and **Server**, a secret key is generated to symmetrically cipher the message and it is also encrypted with the **Server**'s public key (obtained from a certificate), so only it and the sender know it

- **Integrity**

All communications use *RSA-SHA256* in digital signatures¹ and also *ChaCha20-Poly1305* for symmetric encryption, that uses a *MAC*. To ensure integrity of reports on **Server** crash, we used **SQLite** in order for the data to be consistent in case that crash occurs in the middle of a write

- **Authentication**

All users have their certificates provided by the CA, that all entities trust and all messages are digitally signed¹ with the sender's private key

- **Authorization**

Digital signatures are used in all messages¹

- **Non-Repudiation**

All users have their certificates and the sole possession of their private key, ensured by saving them in a keystore protected with a password that only the owner knows (the password is derived from the user id by using the algorithm *PBKDF-2* with *HMAC-SHA1*, in order to simplify the process), and all messages are signed

- **Impersonation Protection**

Same as above

- **Freshness**

In the requests where the **User/HA** queries the server, a nonce is used (that the server stores), and for the other requests, the entity saves the pairs (*user_id*, *epoch*) and checks for freshness against them

3 Communication

3.1 User - User

User communication ensures all properties stated except confidentiality and freshness. Users broadcast their message and expect the near users to answer. They don't exchange their coordinates in order to ensure privacy protection. The requester must send his id and epoch, signed, so that the witnesses can check it with sender's public key. The prover, if near, will answer with his id, the requester id and the epoch (also signed). With this information the requester can check the signature in order to validate the contents of the response (and also check if prover is really near it).

3.2 User/HA - Server

This communication ensures all properties stated¹ since it has information about a **User**'s location. Both parties secure their communication with an hybrid cipher. The generated key will be used to cipher the contents of the message (user's id and location, epoch and a list of all the proofs). A nonce is used to generate the *ChaCha20* keystream and also used to prevent replay attacks on the server side. Both the generated report and proofs contain a signature of their fields, so they can be checked by the **Server**

Users and **HA** can request a *LocationReport* from the server. Users can only see their reports, while **HA** can see all available. For this communication, the security mechanisms are the same, with the difference being that the **Server** is the one that sends the reports.

4 Byzantine User Detection

4.1 By the Users

When getting a message from other users, the signature is checked, so level 2 users can be detected. Also, when a **User** gets a proof from a witness, it consults the grid to check if the prover is near or not, allowing us to detect level 5 users.

4.2 By the Server

Detecting level 0 users is trivial as we only need to check the proofs (duplicates and *self-signed* are discarded). In a synchronous system it's easy to detect level 1 users, because there are proofs signed by them, but they won't have a report for that epoch. In an asynchronous system we can only speculate that. Level 3 can be detected by analyzing the reports of one epoch and checking all the users' coordinates, and inferring why one would not have signed a proof for a nearby user. Level 4 users can't be perfectly caught, but the **Server** can suspect of them by doing the same checks as for level 3.

5 Other Dependability Guarantees

- To fully trust a report, it needs to have $F' + 1$ proofs, otherwise it is flagged as potentially correct
- F' (maximum number of byzantine users near another user) $< F$ (number of byzantine users in the system)
- Communication is asynchronous
- The system needs to have at least $F + 2$ users

¹The only non-signed message is the response from the **Server** to the **User** when a report is submitted, but it isn't a problem, since it just decides what the **User** prints to *stdout*