Highly Dependable Systems

Project - Stage 2

Highly Dependable Location Tracker

Daniel Matos	João Soares	Tiago Fonseca
89429	89475	89542

May 2021

1 Design

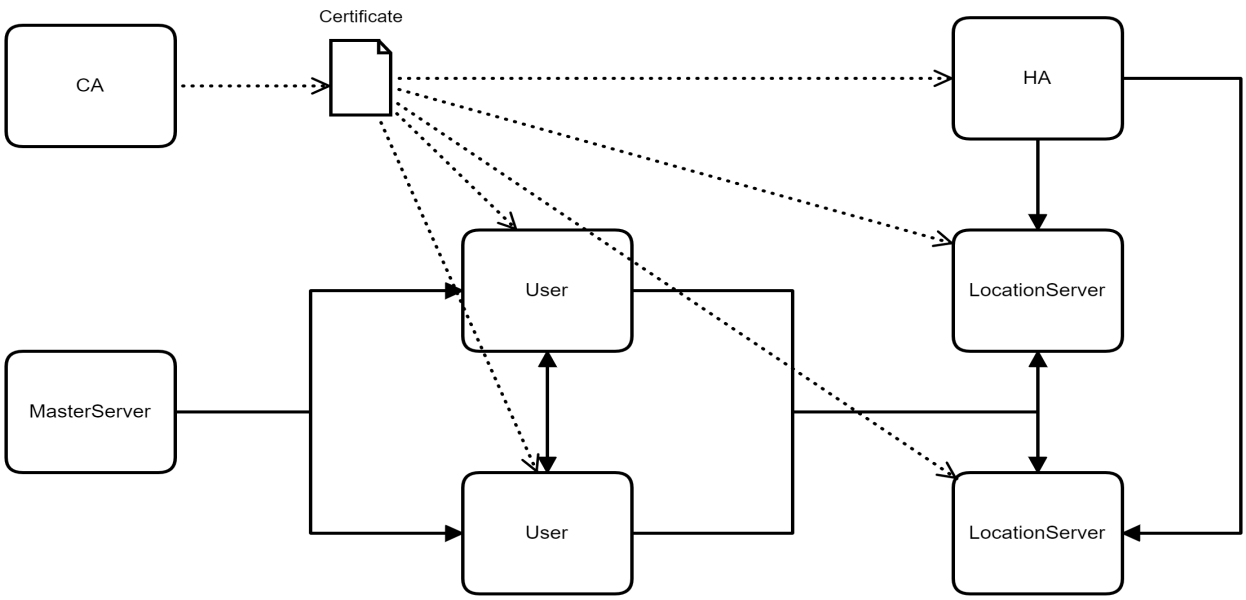

Figure 1: Overall System Design.

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**s (or just **Server**s) in order to consult it later on. **HA** is the one that can consult any *LocationProof*.

Users can have 7 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

Servers can have 2 levels, being level -1 a "normal" server and 1 the "hardest" byzantine. The level is the following:

1. Skip epoch communication

2 Improvements on Previous Version

- **Message not Ciphered**

  When submitting a report, the server sent a response with a boolean stating if the report was valid and accepted by the server or not and it was not being ciphered. Although it only influenced a single print in the user-side, it could lead the user to think it is generating false reports, if a replay attack occurred. Now this response is ciphered as well, so replay attacks cannot trick our users into thinking they are producing invalid reports when they aren't.

3 Algorithm

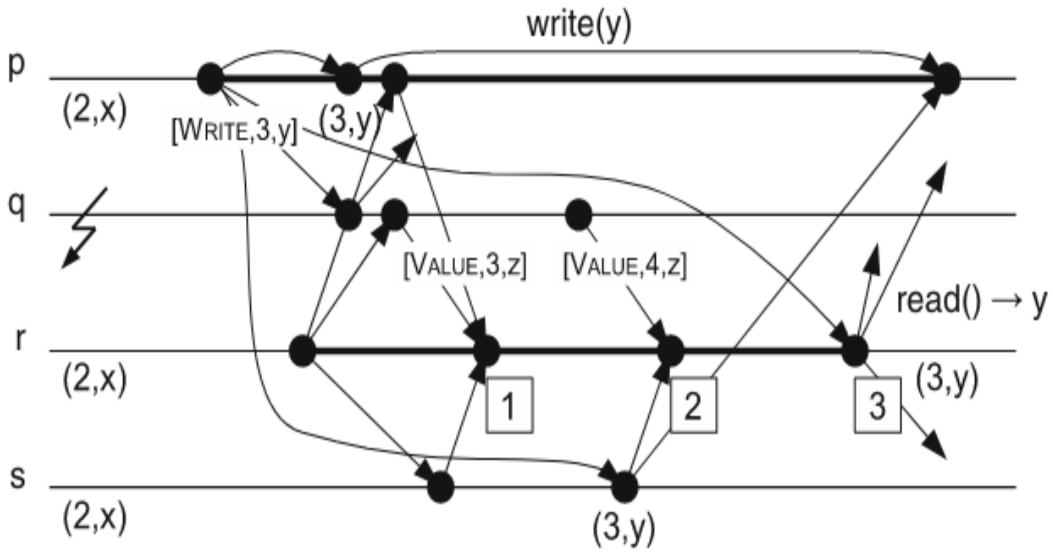

Figure 2: Overall Algorithm.

3.1 Byzantine Quorum with Regular Semantics

A Byzantine Regular Register for multiple readers and one writer was implemented for the *obtainUsersAtLocation* and *requestMyProofs* using an authenticated-data byzantine quorum: we trust the values returned by the servers, and only need to check if the reports/proofs are valid, because the servers cannot forge them as they don't have access to the users private keys. These operations have regular semantics because a server can have some proofs/reports others don't have, so this was the most appropriate way to deal with them.

3.2 Byzantine Atomic Quorum with Atomic Semantic

We implemented a (1, N) Byzantine Atomic Register, for the operations *obtainLocationReport* and *submitLocationReport* using a byzantine quorum with listeners: when a read request is issued, it is sent to all servers and we wait for a byzantine quorum of **equal** responses. When the request reaches one of the servers, it checks if there is a report. If there is, it returns it, if not, the client is added to a list of listeners, so it can be notified when a report becomes available.

When a write request is issued, it is sent to all servers, then it is broadcasted among them. If this operation succeeds, the server then stores the report in its database, sends it to the listeners waiting for the report for the given *(epoch, user id)* pair and acknowledges back to the issuer. If it doesn't succeed, a negative *ack* is sent to the writer.

3.3 Dealing with Byzantine Clients

Since the previous algorithm assumes that clients are not Byzantine, we were forced to find a way of preventing this problem. We implemented a broadcasting system on our servers that whenever a new location proof is submitted by a user, all servers contact each other and check if they have a byzantine quorum of responses so that they can proceed with their operation. By doing this, correct servers prevent themselves from being fooled by a Byzantine client that sends different requests to different servers, by simply discarding these kinds of requests.

All the servers have a timeout, that when triggered, we assume that either the other servers are not responding (because they may be byzantine), just crashed or messages were lost in the network. After that time or after all responses are received, all servers will have the same final state (except for the ones that are Byzantine and behave as they want).

While broadcasting, servers also verify the received nonces as well as the signatures of all the received reports and broadcast messages to see if they have not been tampered/forged by Byzantine Servers. A check was also added to ensure that a given server only sends one response to the others. If more are sent, they will be discarded.

4 Spam Combat Mechanism

In order to prevent an overuse of costly operations on the server, we require our users to perform some work before making a request. To achieve this, we implemented a Proof Of Work mechanism, where users have to find an hash with a given pattern. For testing purposes we did not ask for a very complicated pattern (otherwise we would have to wait a bit for the client). In a real life scenario this problem would be harder.

To prevent clients from reusing the Proof Of Work, they must hash the content of the message with the nonce they are going to send to the server. This way we ensure it is a new valid PoW every time, and not some client just trying to shutdown our servers.

The server will verify this work by comparing it with its own hash of the data using the nonce sent. If they are the same, the request is valid, otherwise the server simply drops the request.

This mechanism is only performed on the *requestMyProofs* operation as required, because it is the costliest.

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
- Byzantine Servers must be smaller than (Number Of Servers - 1) / 3

- Every entity has the sole possession of its private key
- Messages can be ommitted by the communication layer

6 Conclusion

We really enjoyed this project. It was a challenge to implement such an algorithm to prevent Byzantine behaviour. Our clients have have different levels of Byzantine behaviour, while our Servers can only have one level. We were expecting to implement more levels (tampering, for example) to easily test our system, but due to time issues we could not do it. This project also helped us understanding more the implications of byzantine users in real world applications, and how we can mitigate their actions.