Highly Dependable Location Tracker – Stage 2

Highly Dependable Systems

Félix Saraiva *98752*

Bruno Freitas *98678*

Alexandru Pena *98742*

Group 27

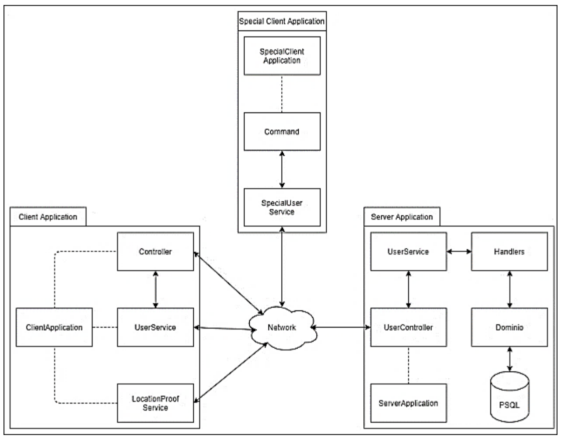
1. Introduction

The core objective of this project is to develop a secure and reliable location tracking and contact tracing system, similar to others introduced by the current world pandemic situation. Furthermore, this system shall tolerate Byzantine faults from both Servers and Clients. Our implementation is established in Java with an asynchronous communication model using REST-based technology – SPRING Boot.

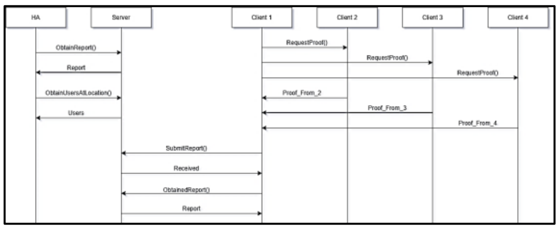
2. Protocol

Our system is divided into three main modules: The Special Client, the Client, and the Server.

The Special Client Application provides a command line interaction to execute the respective Test Cases.



The main Functionality of the HDLT is to receive periodic locations of the users and validate these locations with the collection of proofs provided by nearby users.



By establishing a majority of $f + 1$ proofs (verified by the server) a user can submit a valid report, containing the collection of proofs gathered. Each proof contains the **epoch**, the **witness** ID, the prover's proof **request** which holds the **coordinates**, **epoch**, and the ID of the **prover**. The collection of such proofs is embedded into a **Report** which is then sent to the server via a SecureDTO which contains: the encrypted data, a nonce and the Digital Signature, respecting the assurances in chapter 4.

This 2nd stage brings new functionalities. For example, the Clients can now request a list of all the proofs that a user submitted as a witness on the specified epoch(s). Another feature is replication, now a set of N replicas represent the Servers. N establishes the number of Byzantine faults that the system can tolerate, $N > 3f$. For example, to tolerate one fault, N must be set to a minimum of 4.

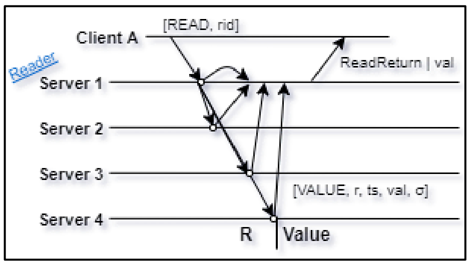
2.1. Servers Logic

The system may also be subject to Byzantine faults that affect Server processes. To prevent such a scenario, we implemented both the **(1-N) Byzantine Atomic Register** and the **(1-N) Byzantine Regular Register** in the respective operations described in the "*Design Requirements*".

The algorithm applied to the **atomic** processes is the Byzantine Quorum with Listeners (Section **4.8** of the course book[1]), and it consists of two components: Read requests and Write request. In the **Read** Requests, the reader must obtain the same reply value from a Byzantine Quorum of processes, more than $\frac{N+F}{2}$. On the other hand, the **Write** Request is more complex since the algorithm works only if the writer does not crash and assuming that $N > 3f$. Also, the writer may concurrently write a new timestamp/value pair to the processes, originating on not all correct processes may send back the same values. To solve this, servers maintain a set of *listeners*, so when a server receives a write request, it multicasts the request to all registered listeners.

For the **regular** semantics operations, the **Authenticated-Data Byzantine Quorum** algorithm is applied (Section **4.7** of the coursebook). In this algorithm, we only apply the **Reader** component.

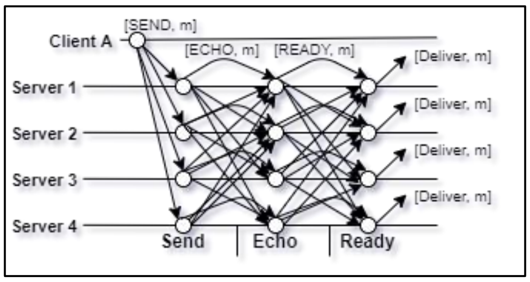
The reader verifies the signature on each pair and ignores invalid ones returning the last written value (with the highest timestamp).



Thus, these read operations need to receive more than $\frac{N+F}{2}$ acknowledgements to complete their executions, assuming that $N > 3f$.

2.2. Clients Logic

Like the previous stage, clients can also be Byzantine, and although we have prevented Byzantine behaviour between clients, the new implementations present new issues. For example, clients can try to send different values to different Servers with valid signatures, send different requests to different values, and ignore some operations. Dealing with generic attacks, such as returning random values or ignoring the operations, has a high level of complexity, but it is possible to mitigate more specific attacks, such as sending different write requests to different servers or only sending a request to some servers. Our proposed solution to this challenge is implementing an **Authenticated Double-Echo Broadcast** (section **3.18** of the course book). This algorithm represents an improvement to the Auth Echo Broadcast Algorithm by adding the **retransmission** and **amplification** steps, which increases performance and safety by making all processes aware of all messages.



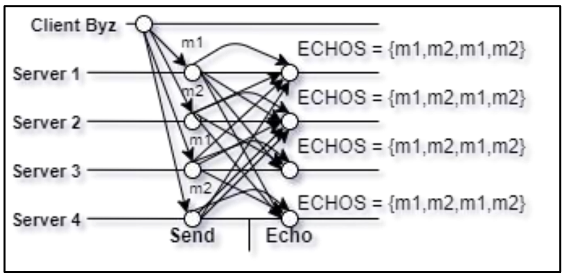
In the example above Client A, sends a message *m* to all *N* servers, a server receives the message and if it has not sent an ECHO yet, it sends [**ECHO**, m] to all *N* Servers. Once a server receives a byzantine quorum of ECHOs, $\frac{N+F}{2}+1$, and has not sent a READY yet, it sends a [**READY**, m] to all *N* servers showing that it is ready to deliver *m (Retransmission step)*. Then if a server receives more than *f READYs* and has not sent its Ready messages, it sends a [**READY**, m] to all *N* (*Amplification Step*: crucial for the *totality* property of the algorithm). Finally, once a server receives a number of equal READYs $> 2f$, it delivers *m* if it has not delivered it yet.

If any of the steps described above fail, the message is not accepted, and no changes occur in the servers.

To show that these steps tolerate Byzantine clients without compromising the servers, the example below demonstrates a **Byzantine**

[1] *Introduction to Reliable and Secure Distributed Programming, 2nd Edition*

Client trying to send different requests to the servers.



As demonstrated, the servers do not obtain the necessary byzantine quorum to send READY messages, so the request fails. Also, it is important to mention that the Register protocols in the client-side work in a synchronous way to avoid client-side errors, this does not compromise the system security in any way.

3. Spam Prevention Mechanism

Our system implements a Spam prevention mechanism to reduce the number of requests servers may receive from Byzantine Clients. By applying a **Proof-Of-Work** concept, similar to Hashcash. Each client must perform a reasonable computational expensive task when sending a request. We have applied this technique to every request sent by a client since every request must be digitally signed and encrypted. Therefore, we consider that every request can be "expensive" if spammed.

Our PoW consists of resolving a "puzzle", calculating an SHA-256 digest of the data to be sent, plus a counter. The challenge is to find a counter value such that the given hash satisfies the puzzle solution. In our case, the digest must start with 1 bit zeroed.

4. Attacks & Protections

Confidentiality – The data of each SecureDTO is encrypted using AES/CBC with a unique session key per packet, generated from a randombytes field. These randombytes are protected using the Server's Public Key that encrypts the randombytes using RSA.

Non-Repudiation – Every packet is signed using SHA256RSA assuring non-repudiation and Integrity.

Spoofing – Digital Signatures assure that no one can impersonate a different entity.

Replay Attacks – By using a unique nonce per packet we assure that each packet cannot be replayed by an attacker.

Tampering – Digital Signatures assure that each message cannot be tampered.

Message Stealing – Prevented because the Sender's ID is signed and sent with each message.

Denial of Service – Reasonable Computational cost "puzzle", in every request, embedded in a PoW.

Surreptitious Forwarding – Protected because the receiver will drop the forward packet, since that same user did not start a communication with the attacker.

Out of Range Proofs – Each Client verifies if the respective request is in its range.

Fake Proofs/Reports – The Server verifies if in the report submitted are duplicated proofs, if the proof belongs to the submitted report, if the proofs and the request proofs are from the same epoch (each proof contains the request proof inside of it, also verified) and if the Digital Signature is valid. These verifications eliminate non-valid proofs and check if it has enough valid proofs to tolerate up to *f* byzantine clients, if the rule is verified then the report is valid.