HDS Notary - Stage 2

Mafalda Ferreira
81613

Leonardo Epifânio
83496

Pedro Lopes
83540

Group 11 - Alameda

1 Introduction

The goal of this project is to create a notary system that allows users to trade goods in a secure and reliable way, tolerating Byzantine faults from both the notary and user processes. The system was implemented in Java and the communication is made via Java RMI.

2 Protocol

The main task of the notary is to legitimize a trade made between two users. This is achieved by performing a reliable **transaction** which is then logged and certified, using the Portuguese Citizen Card (CC) of the notary. A **transaction** is composed by the transaction ID, the user IDs of both participants (referred to as **buyer** and **seller** from this point on), the exchanged **good** ID, both digital signatures of the participants, and the digital signature of the **notary**, using the CC.

The base protocol (specification) remains the same as in the previous stage, enhanced with features to prevent byzantine faults. One of these features is replication, being the notary component now represented by a set of **N** replicas. **N** must be chosen according to the number of Byzantine faults that the system can tolerate, $N > 3f$. For example, in order to tolerate 1 fault, the **N** must be set to, at minimum, 4.

2.1 Byzantine Servers

The notaries can have undefined behaviour. In order to prevent such scenario, we implemented the **Byzantine One writer N readers Atomic Register** (BONAR for short), which prevents that f faulty processes interfere with the well being of the system. The chosen algorithm was Byzantine Quorum with Listeners and it consists of two components: Read requests and write request. It was implemented as stated in section **4.8** of the course book (*Introduction to Reliable and Secure Distributed Programming, 2nd Edition*). The main idea behind this protocol is that the register appears to execute one common sequence of atomic operations as a (1,N) Atomic Register with tolerance to f Byzantine faults. The protocol associated with the Read Requests is very straightforward (**Algorithm 4.19** of the course book.), as it only requires that a reader obtains the same reply from a Byzantine quorum of processes. The protocol associated with the Write Requests raises more issues (**Algorithm 4.18** of the course book.), because, as the writer writes a new pair *(timestamp, Good)* concurrently to the servers, there's a chance that not all correct servers reply with the same values. In order to address this issue, every server maintains a set of *listeners* associated with each good. These listeners are processes that are known to execute a read operation concurrently. So, when a server receives a write request with a new timestamp associated to a Good, it multicasts the request to all the associated listeners.

2.2 Byzantine Clients

Apart from the server, the clients can also be Byzantine. This means that the clients can ignore the operations, return any value, write different values than the ones specified by the application, send different requests to different values, amongst others. Dealing with *generic* attacks, such as returning random values or ignoring the operations has a high level of complexity, but it's possible to mitigate some *specific* attacks, such as sending different write requests to different servers or only sending a request to some of the servers. In order to protect the system from the aforementioned *specific* attacks, our system relies on an **Authenticated Double-Echo Broadcast**, which is an algorithm that implements a Byzantine Reliable Protocol. In order for a **user** A to send a **message** m to the N servers, the protocol has its basis on three types of messages, represented by Table 1.

[**SEND**, m]	Request sent by user with message m
[**ECHO**, m]	Echo of the request with message m
[**READY**, m]	Willingness to deliver the message m

Table 1: Broadcast Protocol messages

The following steps must be taken:

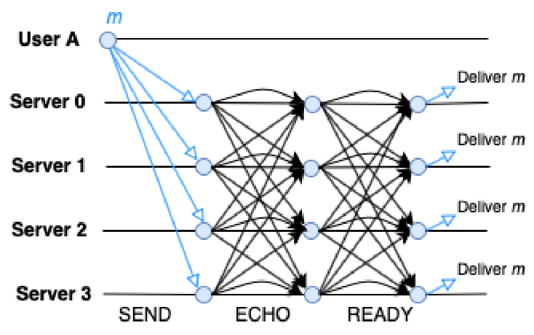

Figure 1: Failure Free execution of Double Echo Broadcast.

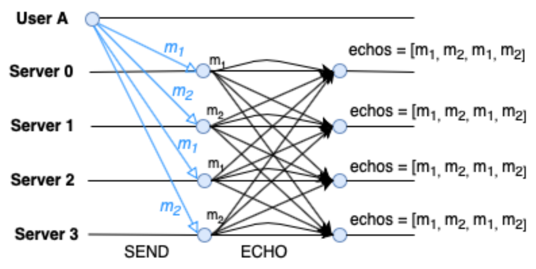

Figure 2: Execution of Double Echo Broadcast with Byzantine User

1. A sends the request [**SEND**, m] to all N servers, as a Basic Broadcast.

2. When a server receives the request [**SEND**, m], if it hasn't send an **ECHO** yet, it **echoes** m - by sending [**ECHO**, m] - to all N servers.

3. Once a server receives a Byzantine quorum of **ECHO**s $\left(\frac{N+F}{2}+1\right)$, if it hasn't send a **READY** yet, it informs all the servers about its willingness to deliver m by sending them a [**READY**, m].

4. Once a server receives a number of equal **READY**s $> 2f$, if he hasn't delivered m yet, it delivers it.

5. If the server only receives $f+1$ **READY**s and hasn't sent its **READY** message, it sends its [**READY**, m] to all the servers. This is crucial for the *totality* property of the algorithm.

If any of the previous conditions referred in the protocol aren't met, then the request is not successful and no changes are made to the servers. These conditions allow the system to tolerate one Byzantine client process without compromising the servers. All the messages traded between the servers are signed and for each protocol message, the receiver server verifies the **signature** of the sender server.

We can see a representation of a correct execution of the protocol in Figure 1 and an execution with a Byzantine client in Figure 2.

In the second execution, the client sends different requests to the servers. This way, the servers won't achieve the necessary quorum to send **READY** messages, making the request unsuccessful.

3 Spam Prevention Mechanism

A spam prevention mechanism was implemented in each *transferGood* request. It applies the Proof-Of-Work concept, where the client must perform a reasonable time consuming task but, in contrast, it's very easy for the server to check that the work was performed. It's a simplified version of *Hashcash*. Having the transaction, which is identified by transaction ID and has additional information like the exchanged good, the participants, with their signatures, are required to append some text (for example, a counter) and compute the SHA-256 digest. The challenge here is to find counter such that the given hash satisfies a challenge, in our case, the digest must start with 8 bits zeroed.

The pre-image resistance of SHA-256 makes it so that brute-forcing is the most efficient way of arriving at this solution. Since checking the correctness of a digest is computationally trivial, the server can quickly check it and discard the request as spam if it is not correct. Furthermore, since the transaction ID is unique, it makes it so that digests can never be reused by clients.

4 Attacks and Assurances

Spoofing - Every message has hashed information encrypted with the sender's private key, creating a verifiable digital signature.
Tampering - Every message contains hashed information encrypted with the sender's private key, alerting the receiver if the data is changed.
Non-Repudiation - Every message contains hashed information encrypted with the sender's private key, constituting a digital signature which authenticates the sender. It is highly unlikely that a user can create a digital signature for another person.
Replay Attack - Each time a user wants to make a request, it first requests a nonce to the server. A randomly generated number, using UUID, is created in the server and assigned to the user. In the next request, the user must use that nonce along with the message and the nonce cannot be used again.
Persistence - After every request, the server, saves its state in two files. The second file is meant as a backup, in order to prevent corruption. Since the save operation is sequential, if the server crashes, at maximum only one of the files will be corrupted.
Atomic Persistence - The system ensures synchronization, there are never two threads saving or changing the state concurrently as there is a global lock preventing such.
Denial of Service - The system reduces the number of requests servers may receive from malicious clients, by imposing a computational cost behind each *transferGood* request.