HDS Notary - Project 2

Cíntia Almeida (MEIC - 85139) Kevin Corrales (MEIC - 94131)
Mário Silva (MMA - 93799)

May 2019

1 Introduction

The goal of this project is to extend the previous system of HDS Notary to tolerate faults on both the user and server sides. To achieve this, we implement a system of $(1, N)$ - Byzantine Atomic Registers, which gives us some strong reliability guarantees, namely:[1]

1. *Termination:* If a correct process invokes an operation, then the operation eventually completes.
2. *Validity:* A read that is not concurrent with a write returns the last value written; a read that is concurrent with a write returns the last value written or the value concurrently written.
3. *Ordering:* If a read returns a value v and a subsequent read returns a value w, then the write of w does not precede the write of v.

In addition to this, we also implement Byzantine Reliable Broadcast to cope with malicious users, and a mechanism for combating spam from the client's side.

2 The system in more detail

2.1 (1,N)-Byzantine Atomic Registers

Each tuple <GoodID, UserID> is associated with a $(1, N)$- Byzantine Regular Register, wherein the process *writer* is the UserID and the write and read values are the variables associated with the GoodID (for example, in our case, *ONSALE/NOTONSALE*, GoodCounter, and Timestamp).

If we allow f Byzantine Notaries, we create $N = 3f + 1$ Notary replicas with which the User communicates, to meet the requirement that $N > 3f$, which is important in the Termination and Validity guarantees.[1]

2.2 Byzantine Users

In this second stage, we allow for the existence of malicious users, which have the ability to alter the client library to attack the system.

In our case, a clear example of this would be a User sending two distinct but valid transferGood() requests to different Notaries, which poses a challenge to our guarantee that all correct processes give the same response.

To counter this, we implemented an Authenticated Double Echo Broadcast which ensures that if a correct process delivers a certain message, then all correct processes deliver that same message, independently of whether the sender is faulty.

2.3 Byzantine Reliable Broadcast

The possibility of a client being able to send different write values to different notaries, would be a big vulnerability in the system. For example a malicious user could send an intentionToSell() request with different buyers to each Notary. This would enable the user to sell the same good twice. To prevent the above situation we implement a Byzantine Reliable Broadcast System, between the Notaries,using the Authenticated Double-Echo Broadcast algorithm[1]. This algorithm takes into account the number of server nodes in the system , N, and the number of tolerated faults ,f.

Every time a notary receives a write request from a client, he broadcasts the received request to all the other notaries in the system - performs and ECHO. He then proceeds to wait for other ECHO broadcasts from the available notaries.

When the notary receives an echo, he stores the message received in a dictionary and proceeds to compare the messages previously received. For each message that matches the content of the newly received message a counter is incremented; If the counter is greater than $\frac{N+f}{2}$ the notary starts a second round of broadcasting , notifying that he is ready to commit the changes requested from the client and waits for second stage broadcast from other available servers.

As in the first broadcasting phase , each time he receives a new broadcast he stores it in a map and compares it to previous received messages and compares it to these,and increments a counter each time there is a match between messages.

The notaries are ready to commit changes when the counter is greater than twice the number of faults tolerated. At this point they can clean the maps and the counters. To guarantee the authenticity of the messages and the senders, each notary broadcasts the message and signature of the client as well, signed with his private key. The receiver than validates the messages with their corresponding private key;

2.4 Spam combat

A client may attempt a *Denial of Service Attack*, where they spam the Notary replicas with a large amount of ”dummy” request. To prevent this, we require that the User produce a *nonce* i, an integer that when appended the message he wants to send, such that the hash starts with a minimum number of zeros.

If the hash used is good enough, as is the case of SHA256, there is no better strategy for finding the nonce than brute-force search. This assures that on average it will take a client a certain time interval to produce each message – exponential on the amount of zeros at the start. On the notary side, it is very simple to check this proof of work so there is no significant computational cost on its protocol.[2]

2.5 Data persistence

With the objective to implement data persistence we implemented JSON files to save all the current goods of the users, this way all the notaries and connected users have the current state of the system.

3 Remaining faults and attacks

In this section we address the possible remaining attacks.

1. *Replay attacks:* These are form of attacks which valid message/information is fraudulently repeated. To prevent this attack we use encryption to prove that the message comes from the correct sender.

2. *Sybil attacks:* In comparison with Replay attacks, instead of replaying the valid information, the message sent is forged. The objective of this attacks is to gain a disproportionately large influence.

References

[1] *Introduction to Reliable and Secure Distributed Programming*, Cachin & Gerraoui & Rodrigues, 2nd Edition, Springer (2006)

[2] *Hashcash – A Denial of Service Counter-Measure*, Back (2002)