Byzantine Fault Tolerant Wallet Implementation*

André Rosa

DI - FCT - NOVA University of Lisbon
af.rosa@campus.fct.unl.pt

Abstract. The task of this work assignment was to develop and implement a system to maintain information about a Digital Wallet, that can be used to create some virtual currency or wallets in some application, like an app store or a game.
The requirements of this project were to provide anonymity and require authentification from the clients, as well as to be secure and tolerate Byzantine Faults.
Therefore, in this report, I present my solution which leverages on a Byzantine Fault Tolerant State Machine Replication protocol library called BFT-SMaRt, to replicate the state of the Wallet across the different replicas. This system's architecture is client/replicated server that exposes a REST API. This solution also implements several mechanisms to provide anonymity and security for the clients.
Finally, I conducted a preliminary experimental evaluation on the performance of my solution on different system configurations: tolerating 0 byzantine faults, tolerating 1 byzantine fault but not happening and tolerating 1 byzantine fault and having a byzantine replica.

Keywords: BFT · SMR · BFT-SMaRt · Digital Wallet

* First assignement of the CSD course

Table of Contents

1 Introduction

The objective of this work-assignment was to develop and implement a system to maintain information about Digital Wallets. Each wallet has an address and the corresponding money deposited in the system. In turn, this system could be used to create some virtual currency or wallets in some application, like an app store or a game.

One of the requirements of this project was to provide anonymity to the clients so that the accounts in the system could not be easily traced to the owner. Another important aspect of the system is to guarantee that only the owners of the accounts can transfer money from it, i.e., the system has to require authentification from the clients to perform transferences. Additionally, the system has to be secure in order to protect the users from attacks, like replay of old messages that could lead to repeating transactions, for instance. Finally, the system has to be able to tolerate the compromise of some replicas, i.e., the system has to able to tolerate Byzantine Failures of some replicas.

Therefore, in this report, I present my solution to such a system. This solution was implemented in Java 8 [1], and leveraged in a library called BFT-SMaRt [3] to perform Byzantine Fault Tolerant State Machine Replication.

Details of the implemented solution are presented in the following sections, being the remaining of this document structured as follows: in Section 1 is presented the overall system architecture and in Section 3 are presented the experimental evaluation results. Finally, to conclude this report, in Section 4 are presented some final observations.

2 System's Architecture

The architecture of this system (Figure 1) is a client/replicated-server architecture, that exposes a REST API with the following operations:

1. **transfer(id_from, id_to, amount, signature)**: Transfer amount from the wallet identified by *from_id* to wallet identified by *to_id*. This operation fails if the balance of the *from_id* wallet is smaller than *amount* or if the *signature* is invalid, i.e., the transaction was not issued by *from_id* or was tampered.

2. **atomicTransfer(List<Transaction> transactions)**: This operation atomically performs multiple transfers, corresponding each transaction to the previous operation. The reason why this operation exists is for the case when a client possesses its money across different accounts and wants to transfer an amount, larger than the balances of each one of its accounts, into a single address, in an atomic operation. Therefore, this allows the client to issue only one operation instead of issuing multiple sequential transfer operations, and waiting for the completion of each one before issuing the next. Additionally, this operation also allows a client to atomically perform multiple transactions, from different origin accounts, into different destination accounts.

3. **balance(id)**: Returns the amount of money associated to the wallet identified by *id* or 0 if that wallet is not registered in the system.

4. **ledger()**: Returns the list of all accounts and corresponding money, that are stored in the system.

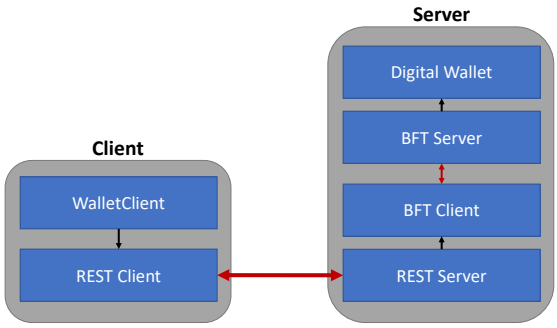


Fig. 1. Overall system's architecture.

2.1 Server

In order to obtain a solution that tolerates Byzantine Faults, my implementation leveraged on a Byzantine Fault Tolerant State Machine Replication protocol library called BFT-SMaRt [3], to replicate the state of the Digital Wallet across the different replicas, using the traditional assumption that for tolerating f faults, the number of replicas should be 3f+1.

Whenever a replica receives a request from a client, it relys on the BFT Client (BFT-SMaRt) component to arrive at consensuses in which order the operation is executed and what is its result. Then, upon being decided, the BFT Server executes the operations, on the Digital Wallet, in order. Afterwards, when replying to the client, the contacted replica aggregates all the replies received by the BFT Client (BFT-SMaRt client) from all the replicas (BFT Server) from who it got a message, and sends the to the client. These replies are signed by the replica that generated it, and so they allow the client to verify a result's authenticity by verifying if it has valid signatures of $f + 1$ replicas.

In order to secure the communications between the clients (REST Client) and the system's replicas (REST Server), protecting them from eavesdropping, tampering and replaying, this system resorts to TLS connections, where each replica is authenticated. Additionally, the system also presents the option for the replicas to also authenticate the client of the TLS connection, if intended.

In order to create money in the system, there are special "admin" addresses that possess infinite money. Therefore, to deposit money into an account, an admin account needs to do a transaction to that account. In this system, there could be multiple admin accounts being their addresses known to all the replicas.

2.2 Client

In order to guarantee that a client has permissions to execute a transfer as well as to provide anonymity for the clients, this system leverages on the solution presented by Bitcoin [2], where each client account is represented by a public-key cryptography key pair, where the public key is used to identify the address to and from whom the transactions are, while the private key is used to authenticate each transaction, by signing it. Therefore, in this system, there is not the need to keep a list of the registered users and the corresponding passwords, since only who has the associated private key can produce valid transactions from the public key. Since it's not possible to directly infer the owner of some address, this solution provides a level of anonymity to its users. Additionally, this mechanism also allows a client to possess multiple key pairs, difficulting the analysis of transfer patterns, which enhances the anonymity of the system even more. As a result, the client application (Wallet Client) is responsible for handling the generation of the key-pairs as well as to sign the transactions before contacting one of the replicas.

Upon receiving a reply from a replica, the client application verifies that the included replies from the other replicas are valid, i.e., if they have valid signatures on $f + 1$ replies. If it's not possible to contact a replica or the reply it sent was not valid, the client application tries to contact another replica. In this case, the operation might have been previously executed and issuing another request would make it possible be executed again. Therefore, to tackle this problem, each client request is enriched with a nonce, a number only used once, that makes possible for the replicas to distinguish between new requests and replay requests. If an operation was already executed by a replica, then it just sends the previously computed result, which requires the replicas to store the result of the previous operation of every client. Additionally, the contacted replica might be Byzantine and propose to BFT Client (BFT-Smart client) different arguments than the ones the client requested. In this case, to identify this, each reply is enriched with a hash computed from the operation and the nonce. Since all the replies are signed by the replica that generated them, this hash can't be tampered and thus allows proving that the contacted replica proposed to BFT-Smart client the correct arguments for the operation. Afterwards, upon receiving a reply, the client compares the hash of the reply with the hash of the operation it requested, and if they match and all the signatures are valid, then the reply is considered valid.

3 Experimental Evaluation

This Section presents the setting and results of the performed experimental evaluation of the implemented solution.

3.1 Experimental Setting

To evaluate the performace of the system, I measured its *Throughput* and *Latency*. The *Throughput* correpsonds to the number of operations performed per unit of time and the *Latency* corresponds to the average time required to perform an operation in the system and receive its result.

The system's performance was evaluated on three different configurations: for $f = 0$ (tolerating no faults), for $f = 1$ (tolerating one fault) with all correct replicas, and for $f = 1$ with one byzantine replica.

For each configuration, there were run multiple tests, with different ratios of write (transaction) and read (balance) operations, during 3 minutes each. Each test consisted in measuring the Latency and Throughput for different amounts of threads (from 1 to 8 threads) issuing parallel requests to the servers, being each test run 3 times to averege the results, in order to amortize the effects of external and uncontrollable factors.

Due to hardware constraints, both the client application and the system's replicas were all executing on the same Virtual Machine. This machine was running Ubuntu 18.04.2 LTS with 4GB of RAM and a Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz, with 4 cores.

3.2 Experimental Results

The experimental results are presented in the Figure 2, matching, for each measured Throughput, the correspondent registered Latency. First of all, all the results are very irregular, especially the case where $f = 1$ (Figure 2.a). I suspect that this behaviour is due to the fact that both the client, issuing requests through multiple threads, and all the replicas were running on the same Virtual Machine. What was expected was that, as the Throughput increased, the Latency also increased, since larger Throughputs imply larger amounts of simultaneous requests to the servers and thus, due to queueing of requests, the response time for each request slightly increases. Nevertheless, in general, the higher the ratio of transactions, the higher the average Latency of the operations, as expected.

For the configuration of $f = 0$ (Figure 2.a), with only 1 correct replica, overall, as the Throughput increased, the Latency decreased, which is counter-intuitive. However, after a Throughput at around 120 operations per second, generally speaking, the Latency started increasing, as expected. For this configuration, the maximum measured Throughput was 171.24 operations per second, corresponding to a Latency of 46.72 milliseconds, for a transaction ratio (T) of 10%, and the maximum measured Latency was 53.63 milliseconds, corresponding to a Throughput of 18.64 operations per second, for a transaction ratio (T) of 90%.

For the configuration of $f = 1$ (Figure 2.b), with 4 correct replicas, the results were highly irregular, since the lines, representing the different transaction ratios, weren't smooth and crossed over multiple times with each other. However, like in the previous configuration, after a Throughput at around 120 operations per second, overall, the Latency started increasing, as expected. For this configuration, the maximum measured Throughput was 160.10 operations per second, corresponding to a Latency of 49.9 milliseconds, for a transaction ratio (T) of 10%, and the maximum measured Latency was 56.42 milliseconds, corresponding to a Throughput of 141.79 operations per second, for a transaction ratio (T) of 50%. As expected, due to needing a larger number of replicas, this configuration was slower than the previous one, which requires only 1 replica, displaying a difference of 2.79 milliseconds between the maximum recorded Latencies of each one.

Finally, the configuration where $f = 1$ (Figure 2.c), with 3 correct replicas and a byzantine one, had the most stable results, since the lines, representing the different transaction ratios, were almost smooth and, overall, as the Throughput increased, the Latency also increased, being that from a throughput of 90 operations per second this becomes more prominent. For this configuration, the maximum measured Throughput was 150.74 operations per second, corresponding to a Latency of approximatly 53.1 milliseconds, for a transaction ratio (T) of 1%, and the maximum measured Latency was 70.45 milliseconds, corresponding to a Throughput of 120.33 operations per second, for a transaction ratio (T) of 90%. This configuration was slower than the previous one, displaying a difference of 14.03 milliseconds between the maximum recorded Latencies of each, showing that in the presence of a byzantine replica the system becomes much slower.

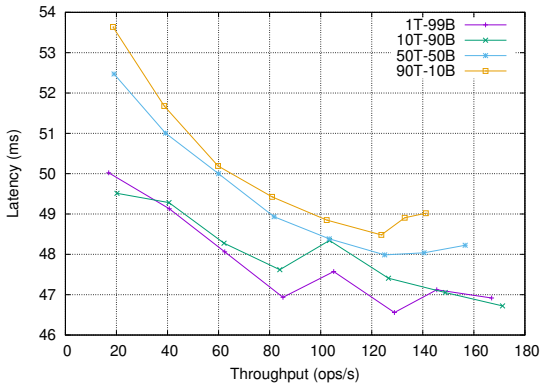
4 Final Remarks

To conclude, in this report I presented my implementation of a Digital Wallet, that leveraged on the library BFT-SMaRt [3] to handle Byzantine Fault Tolerant State Machine Replication of its state (amount of money of each account). The presented solution also provided high levels of anonimity for the clients as well as security between the communications between them and the system's replicas.

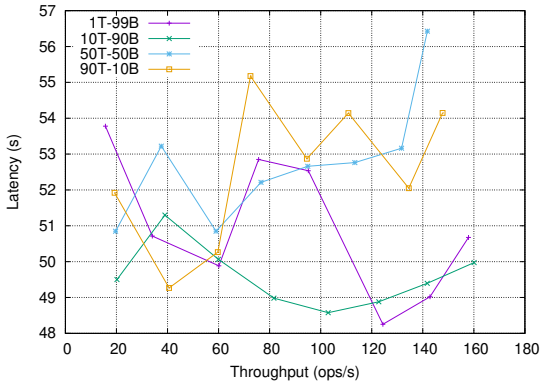
Although in the experimental evaluation, the results were highly irregular, overall, as the Throughput or the ratio of transactions (T) increased, the Latency also increased, as expected. As said before, I believe that this irregular behaviour was due to both the client application and all the replicas running on the same Virtual Machine, and thus, more tests have to be performed, preferably on separate machines, in order to fully evaluate the performance of this system.

References

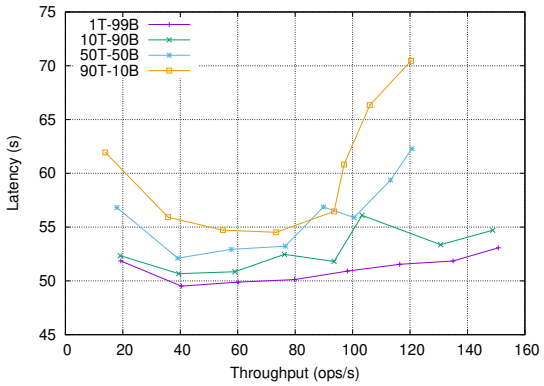
1. Java 8, `https://docs.oracle.com/javase/8/docs/`
2. Nakamoto, S., et al.: Bitcoin: A peer-to-peer electronic cash system (2008)
3. Sousa, J., Alchieri, E., Bessani, A.: State machine replication for the masses with bft-smart (2013)



a) 1 replica with $f = 0$.



b) 4 replicas with $f = 1$.



c) 4 replicas, 1 of them being byzantine, with $f = 1$.

Fig. 2. Variation of the Latency in relation to Throughput of the different system's configurations. T corresponds to the % of Transaction operations and B to the percentage of Balence operations.