Implementation of a Decentralized, Immutable Blockchained Global Wallet Service

Rodrigo Sales 52858, Gonçalo Moura 52360, Group 5

Abstract—This report should include a breakdown of the entire project, with all decisions and implementations explained and justified. With this project, we intended to complete the general idea of a localized blockchain in a central computer, with all the properties and features we considered important, some of them reused from existing systems and some created from scratch. We will approach the chosen architecture as well as reasons for the decision, including an analysis of code and performance for the most important parts, the way it completes the necessary requirements and all implementation difficulties and issues.

1 Introduction

THIS project was developed in two different phases, the first containing a more simple view of a decentralized wallet system, with basic operations and a simple notion of transactions. In this second phase, it was proposed that we reutilized the base of the first phase to develop a complete and accurate blockchain system, with a decentralized ledger, security of operations and consistent states of both the blockchain and the clients' wallets.

As main requirements, it was asked that we implemented a ledger of all operations, immutable and irreversible, that would be able to maintain a trustable record of the progression of the system, as well as an implementation of the notion of a blockchain, with the possibility to create and minerate blocks, containing and using this method to confirm transactions in the system. This system should be supported using BFT-SMaRt, that accomplished a successful replication of the system, maintaining consistency, and that provided the system with protection against byzantine and not-byzantine faults, as well as replicas.

To help accomplish a good implementation of these ideas, a group of operations were suggested, with instructions on the requirements for each of them, and we will discuss them in detail along the report.

To solve this problem and create a system that met all necessary requirements, we decided to create a Java client, that acted as a blockchain client, with an associated list of transactions and his respective wallet. Communicating with this client, we developed a replicated system, utilizing MongoDB for all necessary persistency, that contained a full ledger of operations, with which the system could calculate with safety the state of all existing clients. Reutilizing the base from the previous project phase, we utilized Spring to create the REST communications between clients and system. A decision was made to create a blockchain in the system, replicated and irreversible, that would contain transaction-full blocks, created and minerated by the clients to guarantee confirmation of transactions.

In the next topics, we will approach each of these decisions and implementations in detail, explaining first all necessary concepts for understanding the project and its purpose, followed by a complete analysis of the model and architecture chosen and developed, containing a full explanation of the most important components and features. Finally, a list of the main implementation issues and an analysis as to what happened will be written, with the inclusion of necessary experimental proof and observations of our functioning system, followed by a conclusion.

2 Background

In this topic, we will approach all concepts and features we consider must-know for a full comprehension of the project, that should allow all readers to understand the purpose and decisions made along the way.

2.1 BFT-SMaRt

BFT-SMaRt[1] is a program that handles state-machine replication, whilst withstanding byzantine faults. BFT-SMaRt's design priority is its modularity[2], allowing it to be modified and easily integrated within a system. To test its use, it was integrated in our project to achieve replication and fault-safety. In this report, the system's security and performance will be tested, guaranteeing that all wanted features provided by BFT-SMaRt are functional.

2.2 Blockchain

A blockchain[3] can be considered as a database, but with very different properties and purpose as a regular database. A blockchain is a chained together group of blocks, which contain transactions created by the clients of said blockchain, to guarantee the validity of said transactions and allowing for the transfer of money between users without the need for a verified third party, like a bank, to conform the legitness of a transaction. Each block is minerated by the clients, with the validity of a minerated block accomplished by achieving a difficult and computationally expensive proof of work, that makes sure that all

transactions are valid and that the user attempting to minerate the block is legit. The progress of the blockchain is guaranteed, as transactions are happening at a consistent rate, and so the number of blocks is also increasing constantly. For a user to be able to add his minerated block to the blockchain, it needs to beat all other users in completing the POW (proof of work), and thus making sure that his block reaches the system in first place. Every user that successfully adds a block to the existing blockchain receives a monetary reward, reason as to which blockchains are more and more popular as time goes on.

3 GBWS System Model and Architecture

3.1 Client Architecture

In this project, as mentioned before, we developed a Java client and a server, that is replicated and maintains consistency. To achieve this, the client was developed with the main purpose of being able to communicate with the server in a secure and efficient way, allowing him to be able to execute the necessary operations. Our client was developed with this purpose in mind, and so we decided, instead of using a system like Postman to execute the operations, it was necessary to have a command line that would allow the client to easily communicate with the server and completing multiple operations. This command line can be found in WalletClientCommands class, where simple Spring ShellMethods were included for each necessary operation. Each of these methods executed the implemented operation, present in WaletClientImpl class, and received the response from the server, making sure that the console-response was clear to the user as to what happened in the success or failure of the operation.

3.2 Server Architecture

In the server side, to complement the functionality of the client, it was required that the server could be able to successfully receive the operations from the clients, executing said operations and returning a response to the client depending on the execution of the operation. Adding to this, it was necessary to implement a way for each replica server to communicate with each other and to make sure that all are always consistent and with the same state. To accomplish the first requirement, an interface named WalletController was implemented, creating the endpoints that will make the connection to the client's endpoints possible. To achieve the implementation of the operations on the server side, the class WalletControllerImpl was developed, implementing all operations requested by the client, with a different response in case of success or failure of execution.

To complete the communication with the different replicas, an interface and a class were also developed. The interface AsynchWalletController was created to maintain endpoints that would make possible the communication between all replicas, with the class WalletControllerReplicationImpl used to send all operations to the ServiceProxy, that will allow an operation to be sent to all replicas, making sure the system is always consistently replicated.

3.3 Communication

To achieve an efficient and safe communication, not only between client and system, but also between the different replicas of the system, we decided to use REST endpoints with Spring. These communications allowed for fast, efficient send and receive of operations between all components, as well as all needed responses. To guarantee security in each communication, a group of HTTPS ports was used, as well as SSL/TLS protocols to ensure a proper communication as accomplished. To do this, different certificates were generated, making sure the handshake protocol could be completed. As an extra layer of security, all operations were signed by the clients and this signature checked for validity by the server. For this, RSA-based keys were generated and utilized to sign and guarantee protection of all communications regarding transactions between client and server.

4 Mechanisms and Service Planes in the GBWS Software Architecture

In this topic, we will go into detail of our implementations for each component, explaining the most important features and how we accomplished each requirement. First off, the client, as already mentioned, was designed with communication in mind, being very few the number of operations it executes. In the communication part of the client's architecture, we decided to approach it by designing simple REST endpoints, that would be able to communicate with the system in a simple but effective way.

4.1 Client Handling Server Responses

In the WalletClientImpl class, we implemented every single operation requested by the client, creating, and obtaining every single piece of data necessary for the system to execute said operation, utilizing another class of our creation, ExtractAnswer, to send the requests and data to the system, receive a response and verify its validity. In the ExtractAnswer class, 4 methods were developed: extractAnswerGet and extractAnswerPost, that differ in the fact that a POST method needs an object of data to be sent to the system for the execution of the operation. These methods execute RestTemplate's getForEntity and postForEntity, that simply send the request to the specified url, with an optional object of data, and receives a response from the server when its execution is complete. This response was used in the third method, extractFromResonse, that divides the response to verify its validity and extract from it the actual necessary information for the client. In this method, another class is used, SignatureVerifier, that guarantees that the number of correct responses is enough to be considered a majority of responses, and therefore valid, and that verifies that, if a response is signed by the server, said signature is valid. These two methods from SignatureValidator are utilized in extractFromResponse, verifying the majority of answers received, and finally executing the fourth method

of this class, convertMostFrequentAnswer, that simply turns the reply from the system into an Object for interpretation from the client.

4.2 Client Signing Operations

As for the signing of operations sent by the client to the server, we decided that security was only necessary at this level for actual transactions, that could possibly contain information that should not be able to be accessed by someone intercepting it. With this in mind, we decided to sign the obtainMoney, transferMoney, transferMoneywithSmartContrat and transferMoneyWIthPrivacy. To achieve this, at the moment of creation of the client, a pair of public and private keys are generated using KeyairGenerator for future use. When each of the mentioned operations was called, we used TOMUtils' signMessage to sign the operation and kept the signature as well as the representation of the signed operation inside the Transaction object, that would then be sent to the system, where it could be verified.

4.2 Client Executing Operations

As previously mentioned, the client is not a component that executes operations, choosing to send the requests to the server for execution there. The only exception to this rule, is the mineBlock operation, that is naturally executed by the client, in an attempt to minerate a block ahead of his competition. Before the client is allowed to minerate a block, and as a way to improve security and verify that a user is legit, as well as creating our own form of proof of finalization, a client needs to send money to a pre-created pseudo-account named FUND, considered an escrow account that will authenticate a client as being able to minerate a block. In this method, we attempt to complete the POW, in this case, by calculating the hash of a block, utilizing its set of transactions, the previous block in the blockchain's hash and a nonce, that should be increased every time the hash does not complete the challenge. In our case, we decided, for testing and simplification reasons, to define the challenge as a single zero as prefix of the hash. When this is achieved, a block is considered valid by the client and can be sent to the server, for further validation and possible entry into the blockchain.

4.3 System Communications

In this topic, we want to analyze how the system not only communicates with the client but also with its replicas, guaranteeing consistency in their state. To communicate with the client, the server needs to specify its endpoints for each operation, as a way to create a channel between both components, so that requests and responses can be traded. In the WalletController interface, these endpoints are defined and the communication with the client is initialized. As the channels are created, the server now needs to receive the requests from the clients, analyze its data and execute the operations. This is achieved in a combination of classes, Server class and WalletControllerImpl class. In the Server class, we create the invokeCommand method to make the connection between which operation is being requested by the client and the extraction of its data from the request. To do this, a Path Enum was created, and the path is read from the request sent by the client. In each operation, the data from the request is read by the server, and the WalletControllerImpl methods that execute the specified operation is called, sending it the arguments read from the request. The method sendFullReply is called in the execute ordered and unordered, receiving the response from the WalletControllerImpl execution, and sending the response to the client for interpretation. In the WalletControllerImpl, the operations are, as mentioned, completely implemented, with all changes reflected across all replicas and sending a response back to be delivered by the sendFullReply method.

To verify the signed operations sent by the client, a KeyPairGenerator is also used at the moment of creation, giving the server keys that will make it possible to verify the signature sent by the client.

As for the communication with the different replicas, two classes and an interface were used: the interface AsynchWalletController is used to simply create the endpoints for communication between all the replicas, the class ClientAsynchReplicator is used to create the methods that will utilize the AsynchServiceProxy to communicate with the different replicas, and send the operations across, waiting for the responses. The final class, WalletControlleReplicationImpl will implement the previous interface, and will call the methods from the ClientAsynchReplicator, returning the response from the replicas.

4.4 Server Persistency

To guarantee persistence in the blockchain and all transactions, we decided to use MongoDB and integrate it with Spring. We created repositories for Blocks, Users (or Clients), and Transactions. These repositories are created at the creation of each replica, with a database for each one, and all the previous replication features guarantee their consistency and state across all of them.

4.5 Blockchain Implementation

Our blockchain was created, for easy access and insertion, as an ArrayList in the server side and replicated through multiple instances. To guarantee that blocks could be created and added, a genesis block had to be created, so that this hash could be used in the calculation of the next block's hash. This genesis block was created and added to the blockchain at the moment of creation for each replica. A ledger of all operations was kept untouchable as a repository, so that no unplanned modifications happened. To help in the progress of the blockchain, a list of not mined yet transactions was kept, so that when a client requested transactions to minerate a block, no already mined transactions would be considered and guaranteeing that no transaction would be left behind without being confirmed. To make sure that the client with the best block would be chosen to the blockchain and guaranteeing that the progress of the system was protected, a queue of all blocks ready to be added to the blockchain was also kept, as a way for the system to decide which block would be better to select to enter the blockchain with the

rest being discarded.

5 Implementation Issues

With these projects of big dimension, implementation issues are always present, and our project is no different. To start off, our first phase of the project had an impossible to correct error that had us stuck until almost the very end. Due to this, we could not deliver a project that we were proud and wanted to correct that in this phase. In this case, it ended up being good for us because we could create the blockchain almost from scratch, guaranteeing that no previous implementations caused problems.

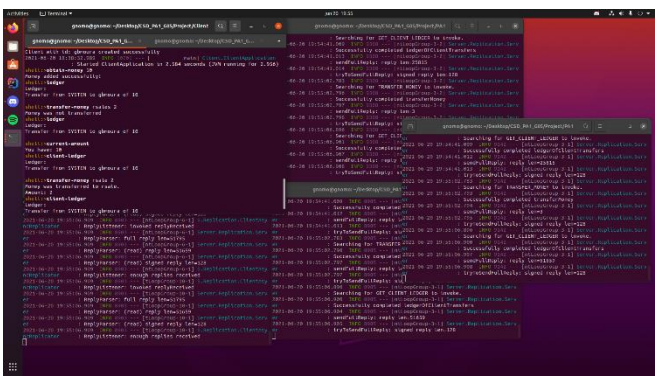
Another problem we had, in this phase of the project, was that we had difficulty sending multiple objects in the requests from the client to the server due to lack of serialization in our objects. Due to this, we had to compromise and send a lot of extra information inside the objects sent, like inside the Block or Transaction objects.

Finally, we couldn't' complete the gold operations, not due to any problem in our code, but because of time constraints because of other courses' tests, projects and discussions.

6 Experimental Observations and Analysis

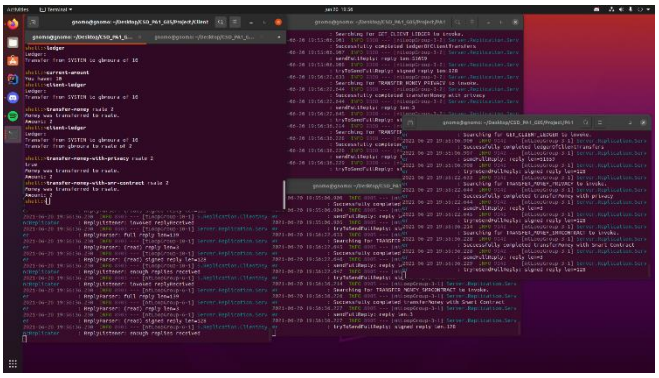
In this topic, we will utilize images and proof of the execution of our system as a way of validating our operations and proving its functionality and efficiency.

First, we want to prove that the operations obtainMoney, transferMoney, currentAmount and both ledger operations are functional and run on all replicas.

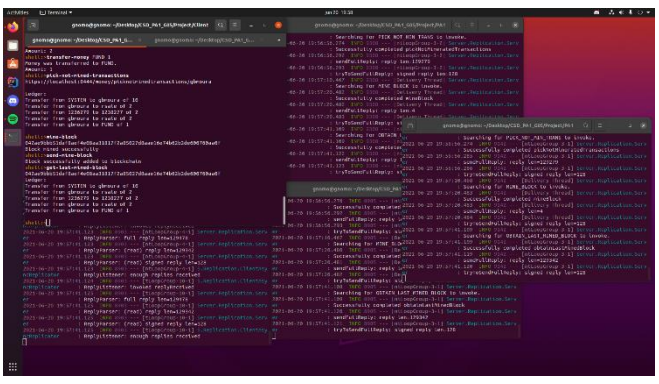


(Please zoom in for better viewing)

In this image, a client with id "gbmoura" is created, and an obtainMoney is performed. By executing ledger next, we can see that the operation has been registered as completed, and the currentAmount proves that the operation has been considered as successful. Next, we try to transfer money to a user that does not exist yet, and from the response we can see that the system does not allow it. By creating the client, we can observe that the transfer is successful.



In this image, we prove that transferring money with privacy and with a smart contract also work correctly, receiving a positive message from the server, meaning that it was completed with success.



In this image, we start by transferring money to the mentioned FUND, that allows the client to minerate blocks. Next, the client asks for not minerated transactions, and the system returns all the transactions not yet confirmed. Whit the operation mineBlock, we receive a successful message and the hash, and as we can see it starts with a zero, just like the challenge requests. Following the mining by sending the block to the system, our block is successfully added to the blockchain, since it is the only one attempting it at the moment. When getting the last mined block, the system responds with a representation of the hash and the list of transactions it contains.

7 Coverage of the Requirements

In this project, we completed every necessary requirement, presenting a blockchain with all properties asked, guaranteeing that all operations are correctly implemented and that return all important information. The only actual limitation we possess at the moment is that this project was designed with a local execution in mind, so the execution of multiple replicas in different systems was not tested and therefore we cannot prove its functionality.

As mentioned before, the gold methods were not attempted due to lack of time, as well as the variations of the ledger methods, but all other operations are functional and confirmed.

8. Conclusion

With this project, we learned how to implement a complex communication between client and server, since our last implementation of such a system was in SD course and here, we were allowed to explore at greater length its possibilities. We learned the basic functions of a blockchain, how it works, its main components and how we could create a representation of our own interpretation. We expanded our knowledge of TLS, HTTPS, Spring/REST and applied it to an interesting and original project in Java context. As possible future improvements, we could attempt to finalize all operations, as well as expanding the project to work with multiple replicas in different domains.

References

[1] [1] A. Bessani, J. Sousa, E. Alchieri. State Machine Repli-cation for the masses with BFT-SMaRtW.-K. Chen, *Linear Networks and Systems*. Belmont, Calif.: Wadsworth, pp. 123-135, 1993. (Book style)

[2] J. Sousa and A. Bessani, "From Byzantine consensusto BFT state machine replication: A latency-optimaltransformation," em Proc. of EDCC'12, 2012

[3] Investopedia. 2021. Blockchain Explained. [ONLINE] Available at: https://www.investopedia.com/terms/b/blockchain.asp#:~:text=In%20a%20blockchain%2C%20each%20node,reference%20point%20to%20correct%20itself