SEC-Report

Tomás Guerra 102178 / Diogo Branco 92453 / Beatriz Alves 86391

April 1, 2022

1 Introduction

The goal of this project is to develop a highly dependable banking system, with Byzantine Fault Tolerant (BFT) guarantees. The system was implemented in Java and for the client-server communication it was used gRPC (Google Remote Procedure Call).

2 Protocol

The main goal of the Byzantine Fault Tolerant Banking is to maintain a set of **BankAccounts**, each of them identified by a public key pair generated upon sending the first request to the server, and their **Transactions**, identified by their ID. Each client that opens an account in this system starts with an initial predefined balance of 500. In order to perform the transaction from **client** C to **client** G of **amount** A using the implemented system B, we must take a few steps:

1. C sends ***openAccount*** request to B
2. G sends ***openAccount*** request to B
3. G sends ***sendAmount*** request to B with its parameters being the **sourceClient** G, the **destinationClient** C and the **amount** A to be sent
4. C sends ***checkAccount*** request to B
5. C sends ***receiveAmount*** request to B with the **transactionID** as a parameter to confirm the right **transaction**
6. G can send ***audit*** request to B to check C's **transactionHistory**
7. In every request that modifies an account's status, it is also sent the **signature** of the message, to guarantee integrity and authenticate the sender. This signature is verified with the **public key** included in the same message

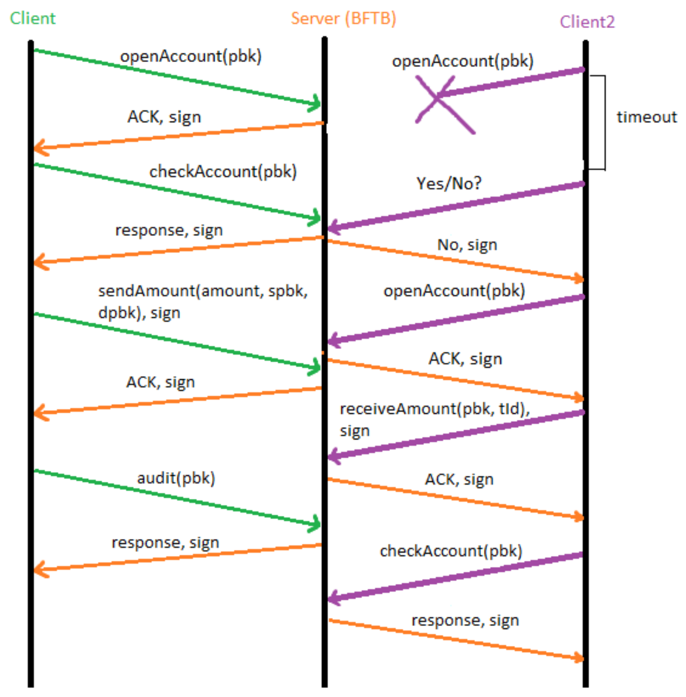

Figure 1: Developed Protocol

3 Design

3.1 System architecture

For this stage 1 of the project we have several clients that can communicate with only one existing server. For recovery purposes if the server crashes it can get back to work normally as we are using a Postgresql database to store the data necessary for that recovery.

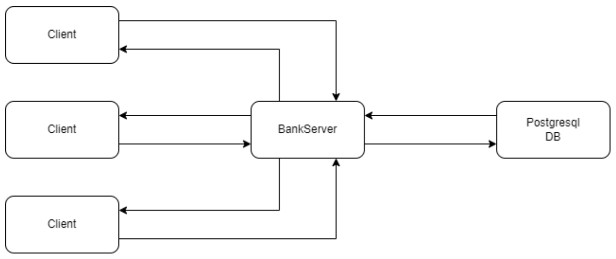

Figure 2: System architecture

3.2 Communication and messages

For the communication between the clients and the server we are using gRPC. All gRPC services and messages between clients and the server are defined in the bank.proto file. For the message design firstly, we decided that none of the messages passed between client and server needed to be encrypted because knowing that anyone will be able to see the transaction history of others and consult their balances, we assume that there is no need for confidentiality in the exchange of messages. Secondly, every response message that the server sends to the clients are signed by him so that the clients know that it was the server who sent the message. Thirdly, only to the **sendAmount** and **receiveAmount** requests that the clients send to server are signed by them as they are the only ones that can interact with the clients balance.

4 Security guarantees

4.1 Confidentiality

Like said in the 2.2 subsection, we assume that there is no need for confidentiality in the messages between the clients and the server since all clients are able to check transaction history and balance of each other and none of the requests will contain information that others can‘t know. So attacks like **eavesdropping** or **Sniffing** don‘t really affect our system.

4.2 Integrity

Our system provides integrity and authentication in every message because we add a signature in all of them. Any attack of **Unauthorized data modification** will be detected by our system.

4.3 Replay Attacks

To prevent replay attacks, we could use **nonces**. When the client wanted to make a request, they would first ask the server for a nonce.

Then, the client would add that nonce to the original request. The message's signature would also be based on the nonce to guarantee it wasn't modified. When the server received the request, it would check if the nonce matched the one it had sent.

If an attacker attempted to replay the client's messages, the server would detect that the nonce had already been used and reject the request.

We weren't able to implement this feature due to time constraints.

4.4 Availability

Since our system only has one server at this stage, it is quite weakened regarding to attacks like **DoS**.

5 Other types of dependability guarantees

At this stage we are told to assume that only our server is prone to crash. To solve this we add a timeout from the moment we send the request to the server and if this timeout passes, we will send another message to the server asking if it performed the intended operation or not.

If the server crashes in the middle of a state-changing operation, the database performs a rollback. This ensures there isn't any data corruption and the server is always in a valid state.

We use synchronized functions to access the database to guarantee there aren't any errors resulting from concurrent updates.