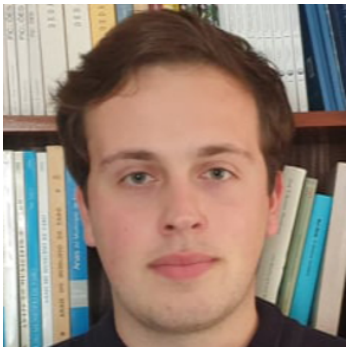
Byzantine Fault Tolerant Banking - Group 26

Highly Dependable Systems 2021/2022



Bernardo Quinteiro
93692

Diogo Lopes
93700

Gonçalo Mateus
93713

Design

Structurally, this project is implemented in Java, contains four main directories and we decided to use remote procedure calls for the communication between the servers and the clients.

Data persistence and corruption prevention is guaranteed in Servers by the use of atomic writes to files, which means that firstly, information is written to a temporary file and, then, if the write is successful, it's written to the supposed destination. With this we are assured that the writes are either completed or not done at all.

The main directories in the project are:

- **server** which contains a folder for each server with its keystore and an atomically written database file. Each server is responsible for answering the clients requests and performing the necessary operations;

- **client** that contains two folders responsible for storing the clients' certificates and keystores, a test folder that contains the JUnit tests and the App which is responsible for the Client API;

- **crypto** that contains cryptographic functions used by the other modules.

The data that is stored atomically in the server's text files is a HashMap object with every account registered in the banking system. Each account is identified by a Public Key and has properties: username, balance, rid, wid and 3 lists. The first 2 lists (pending and history) are lists of the object Transactions. Then, there is a list that stores nonces, in order to prevent replay attacks.

<table>
<tr><th colspan="6">Client</th></tr>
<tr><td rowspan="3">Open Account</td><td colspan="2">Check Account</td><td>Audit</td><td rowspan="3">Send Amount</td><td rowspan="3">Receive Amount</td><td rowspan="3">Rid</td></tr>
<tr><td colspan="3">Proof of Work</td></tr>
<tr><td colspan="3">(1,N) Byzantine Atomic Register</td></tr>
<tr><td colspan="7">(1,N) Byzantine Regular Register</td></tr>
<tr><td colspan="7">Byzantine Quorum</td></tr>
<tr><td colspan="7">Authenticated Perfect Link</td></tr>
<tr><td colspan="7">grpc</td></tr>
</table>

<table>
<tr><th colspan="6">Server</th></tr>
<tr><td rowspan="3">Open Account</td><td colspan="2">Check Account</td><td>Audit</td><td rowspan="2">Send Amount</td><td rowspan="2">Receive Amount</td><td rowspan="3">Rid</td></tr>
<tr><td colspan="3">ADEB with Byzantine Quorum
(in writeback)</td></tr>
<tr><td colspan="3">Proof of Work Validation</td><td colspan="2">ADEB</td></tr>
<tr><td colspan="7">Authenticated Perfect Link</td></tr>
<tr><td colspan="7">grpc</td></tr>
</table>

Integrity

The system provides integrity since every message sent between the client and the server is signed with the corresponding Private Key and then this signature is verified which guarantees the message was not altered, since an attacker cannot replicate a valid signature.

Dependability

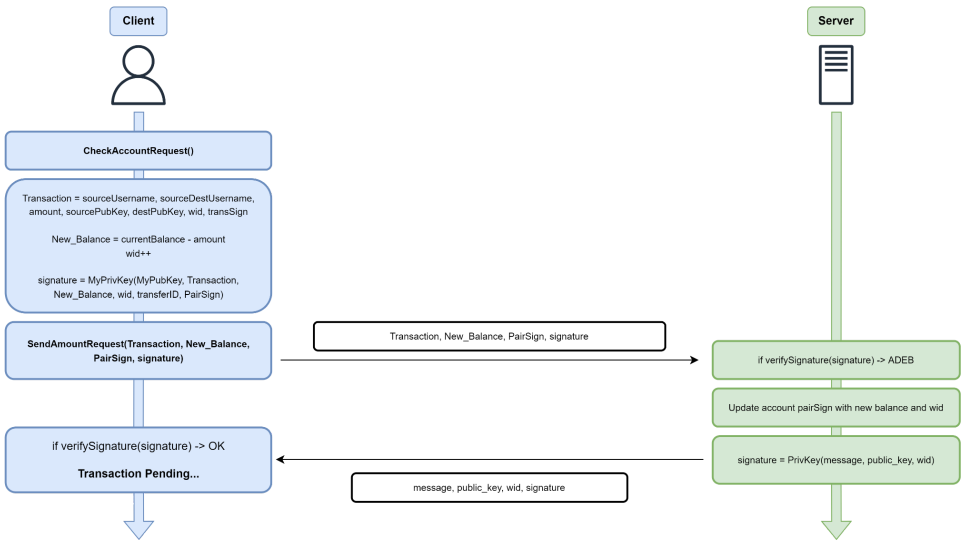
In order to guarantee a client from the bank is performing the requests, the client has to be logged in using a username and password.

The servers are stateful, so every time they're rebooted they'll possess the accounts information, which is guaranteed by the use of atomic writes.

Client API Methods

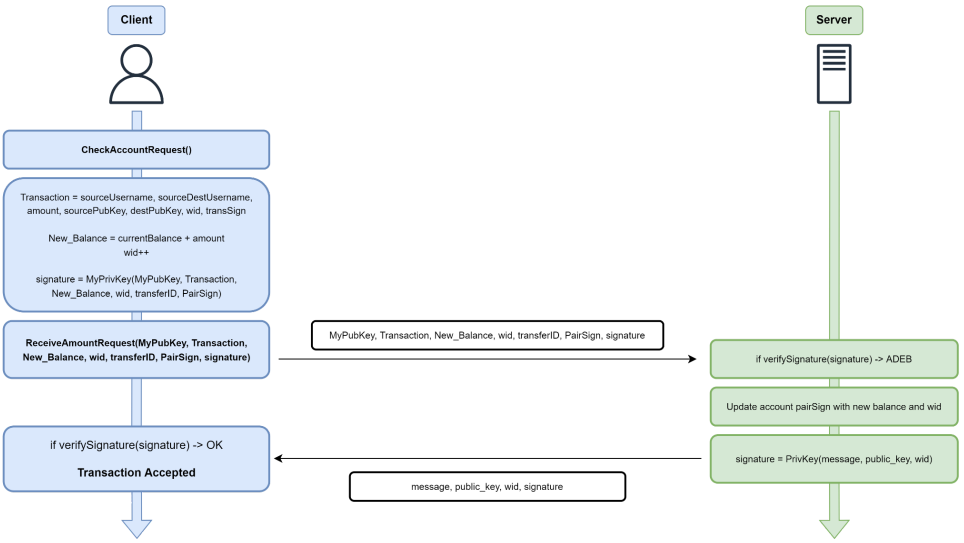
Every account has a rid and wid which are incremented values during the reads and writes respectively. After a login a ridRequest will be sent so the client obtains the most recent rid.

SendAmount()

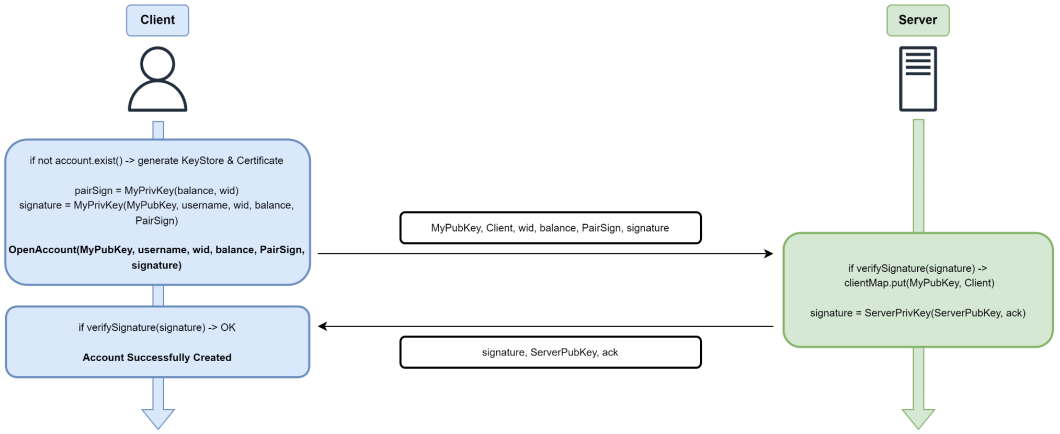


Before sendAmount(), a checkAccount() is made, in order to obtain from the servers the last values written (balance and wid). This balance will then be decremented with the amount wanted to send and the wid will be incremented. These values will then be signed to a pairSign that will be updated in the server and is used to prevent byzantine servers.

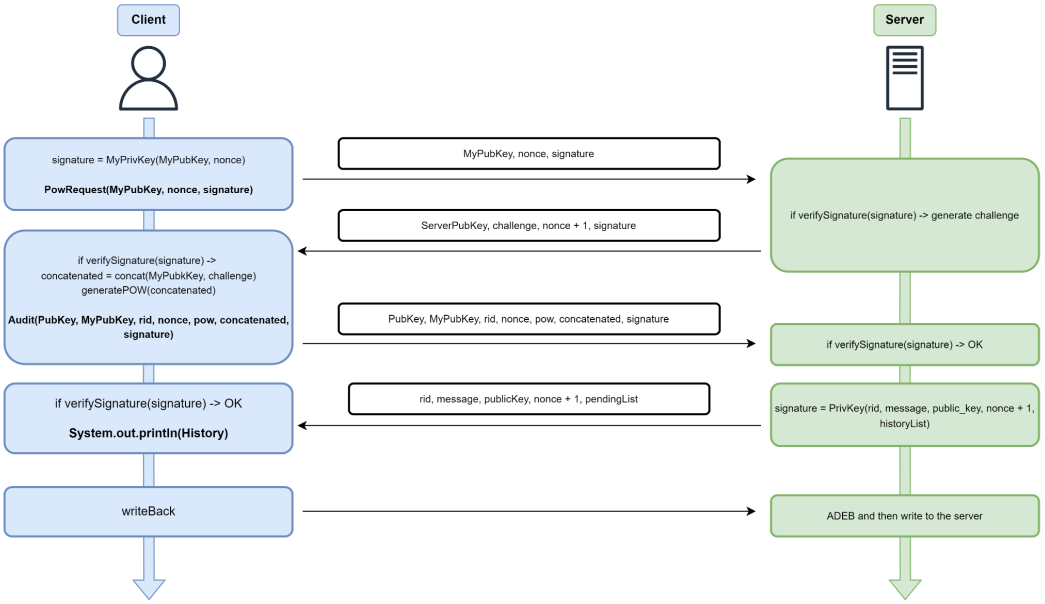
ReceiveAmount()



OpenAccount()

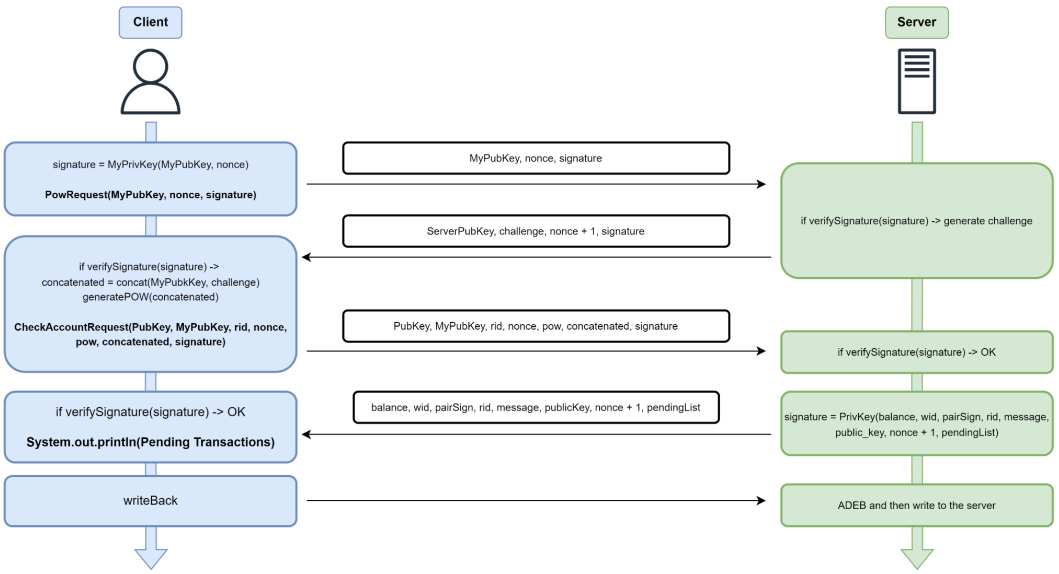


Audit()



Firstly, a PowRequest will be made to which the server will reply with a randomly generated byte array. Then, this array will be concatenated with the user public key and a pow will be generated and sent with the rest of the parameters in the request. When at the server, there will be a proof of work validation and, if correct, the rest of the method can be processed. After the last write is returned to the client, a writeBack request to all the servers will be made.

CheckAccount()



Base considerations

In order to tolerate *f* byzantine servers, the number of servers that need to be created is N=3*f*+1.

All the exceptions thrown by the servers are signed with the server's Private Key and with a rid/wid/nonce, in order to prevent man in the middle.

The Private Keys are kept confidential in Keystores generated by keytool. Each server and client has its own keystore which contains a Public Key Certificate and the corresponding Private Key.

(1,N) Byzantine Regular Register

A correct client sends a request to all servers and waits for a Byzantine Quorum (*2f+1*) of responses, this way, if a transaction has been registered, the client will receive the last written value from at least one server.

(1,N) Byzantine Atomic Register

A correct client sends a read request (checkAccount and Audit) to all servers and after verifying that he has received a valid response by a Byzantine Quorum of servers, he will send to all servers a writeBack request with the parameters received from the previous response.

ADEB

When a client needs to perform a write (sendAmount(), receiveAmount() and writeBacks), the servers will perform ADEB. This will guarantee that at least a quorum of servers has the same request from the client, avoiding inconsistencies between them.

Spam Combat Mechanism

In order to mitigate an attack where a client could flood the system with computational expensive requests, a Proof of Work mechanism has been implemented. A client requests a byte[] nonce to the server, concatenates this response with its Public Key and then computes the work depending on the previously set difficulty, receiving a *long* value that is sent to the server, which verifies the *proofOfWork* and if successful can proceed.

This "work" is implemented in the read methods *checkAccount()* and *audit()* since these don't only involve returning a string to the client, but also performing a writeback to the server and a consequent ADEB exchanging a quadratic number of messages between the servers.

Guarantees

- **Drop message prevention:** In case the request times out, the requester will retry to send it until receiving a response;

- **Impersonation:** A client won't be able to make transactions for another client since requests are signed with the client's private key, so the signatures won't match.

- **Man-in-the-middle:** If an attacker tries to modify a request for example to receive money in his account, it will not be possible since the signature will not be verified by the server.

- **Replay:** If an attacker replays a request it will be automatically rejected by the server since the nonce will be matched in the accounts event list or the rid will be inferior to the one on the server which means an old request is being resent;

- **Authenticity:** Signatures are generated and appended on every message with the corresponding request information;

- **Byzantine Server Safeness:** Write requests such as sendAmount, receiveAmount and writeBacks are only concluded after executing the ADEB;

- **Byzantine Server changes transactions:** Each transaction contains a wid of when it was performed and is sequential, so when performing an *audit* it's possible to verify if a transaction has been added, removed or duplicated by a byzantine server;

- **Byzantine Server sends wrong balance:** Each client has a pair signature of it's balance and wid, which makes it possible to detect when making a *checkAccount()* if the returned balance has been tampered or not;

- **Manipulating:** If an attacker intercepts a message and tries to manipulate its content it will be detected by the server and the request will be rejected.

- **Data Persistency and Non-Corruption:** The Server keeps its data in a txt file that is atomically written, which survives to system crashes, keeping it consistent even if it happens in the middle of an operation.