FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Confiabilidade de Sistemas Distribuídos

Professor Nuno Preguiça

Trabalho Prático 1

(https://github.com/aanciaes/SecureReplicatedBank)

Miguel Anciães nº 43367
Ricardo Amaral nº 43368

INDEX

Introduction

The goal of this project is to create a system that maintains information about wallets. This system could be used, for example, to maintain the information about some virtual currency or to maintain the information about wallets in some application (e.g. game).

The system exports three (plus one for developer only) operations:

- generateMoney(id, amount) - Adds *amount* to the wallet identified by *id*. This operation only works for admin users.
- transfer(id_from, id_to, amount) - Transfer *amount* from the wallet identified by *from_id* to wallet identified by *to_id*. o This operation fails if balance of the *from_id* wallet is smaller than *amount.*
- get(id) - Returns the amount of money associated to the wallet identified by *id* or HTTP 404 if that wallet does not exist.

System design

Server

The system is implemented using a client/server architecture and exposes a REST API. For this purpose we used the **Glassfish Jersey library**.

The system is able to tolerate Byzantine faults in the servers, using the traditional assumption that for tolerating f faults, the number of replicas should be 3f+1. For tolerating Byzantine faults, the server implements some form of Byzantine Fault-tolerant replication using the already implemented **BFT-SMaRt library**.

Each HTTP server has a class named *ReplicaServer.* This class represents a replica of the server and it’s the class that holds all the data of the system. This class and all the replication system is built with the help of the **BFT-SMaRt library**. In this case, with the help of the *DefaultSingleRecoverable* of this library.

This ReplicaServer class is responsible for all processes of the replication and for maintaining the system correct and concise across all replicas. Requests can be made to any http server, since all the requests go through a consensus protocol in the background in order to ensure a correct response to the client.

For security, the server is using a **Bitcoin** like authentication system. All requests must contain a signature, using the public key of the user performing the operation, to make sure that the user is allowed or not to perform that operation.

We also made some alterations to the **BFT-SMaRt library** to allow all the messages from the replicas to be sign. So, the server that receives the client request, gathers all responses from the replicas and sends them to client along with it’s response, so that the client can verify the response validity.

Client

The client was also implemented to not trust the server by default, verifying all responses and signatures from the server and from all the replicas.

Then, our solution guarantees that:

- The client is connecting to a server of the systems and avoid replaying. TLS is used on all connections between the client and the server.
- The client verifies that the replies from the server are correct.
- The system guarantees that the client has permissions to execute the transfer operations.

In depth

Generate Money Operation (POST)

The group defined that the generate money would be the operation that not only generates some money to a user, but also actually creates a user in the system, if we try to generate money to one that does not exist.

With this amount of responsibility, the group also defined that this operation is only allowed to be performed by an administrator of the system. For that purpose, we created a key pair of public and private key and stored it in the root of this project. This simulates an administrator user that would keep it's private key private and secure.

To perform this request the client must provide to the server the following parameters as a body request (JSON):

```
private String toPubKey;
private Double amount;
private Long nonce;
private String signature;
```

where the toPubKey argument specifies the user that will receive the money (or a new key to create a new user with that amount), the amount that will be transferred to the user, a nonce to randomise the signature, and the signature of the request. The signature is created by adding all of the first three parameters of the request as a unique string, and then signing it with the private key of the client.This signatre is encoded in Base64. For this particular operation, the only signature allowed will be with the administrator private key. If not, the server will reject the request and return a HTTP 403.

With this request, an header must also be passed to the server called *nonce.* This nonce it's generated by the client and just be kept until the server replies. The server will reply with (nonce + 1). This is a security measure to make sure the server is not replying messages.

Transfer Money (POST)

The transfer money request transfers some money from an user to another user. It fails it the user that is making the request does not exists. However, if the user that is receiving the transfer does not exist, it is created. It fails also if the user does not have money left in the account to make the transfer or if the user tries to transfer a negative amount.

This operation needs the following parameters as body request (JSON):

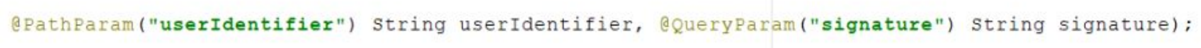
```
private String fromPubKey;
private String toPubKey;
private Double amount;
private Long nonce;
private String signature;
```

The request needs the "address" of the user that is transferring and receiving the money, represented by the *fromPubKey* and *toPubKey*, the amount that is being transferred, a nonce to randomise the signature and the signature itself. The signature it's composed by all previous fields joined into a string, and signed with the private key of the user that is performing the transfer and encoded in base64.

And again, along with the request, the nonce goes in the header of the request, and should be saved by the client to match with the nonce returned by the server.

Get Balance (GET)

It returns the amount currently assign to a user. This operation is a GET operation and it receives as a *Path Parameter* the user identifier that wants to check its balance. As a *Query Parameter* the server needs the signature of the request which is composed by the user public key and a nonce to randomise the signature, and signed by the user's private key, encode as a Base64 string.



```
@PathParam("userIdentifier") String userIdentifier, @QueryParam("signature") String signature);
```

This request returns a HTTP 404 Not found if the client does not exist in the system.

Server: Request Authenticity

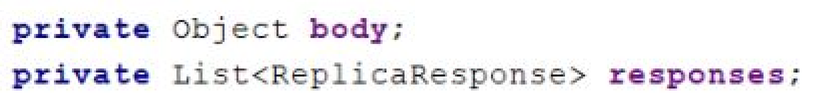
As soon as the server receives a request from the client, it knows the user that wants to perform the operation, by using the publicKey passed (in different ways in different methods) to the server, and tries to verify the signature also sent to the server.

It builds the hash with the other request parameters, decrypts the signature using the user's public key that knows beforehand, and tries to compare the two hashed strings.

If one of these parameters don’t exist, or if the verification goes wrong, the server immediately rejects the request and will not continue the operation. The client receives a HTTP 403 Forbidden. This way we can know for sure (assuming that private keys are secret and not shared) of the authenticity of the request.

Client Response

The structure of the client response it’s the same for all requests:

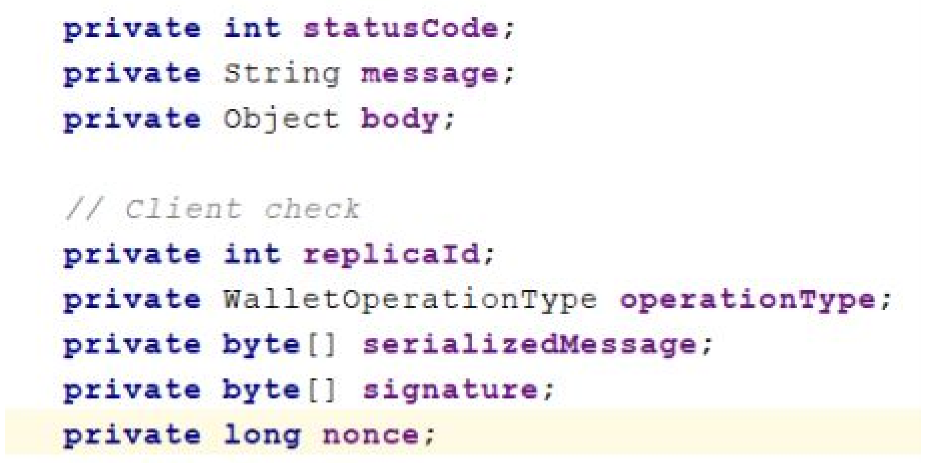


```java
private Object body;
private List<ReplicaResponse> responses;
```

The body contains the actual response to the request, and alongside it has a list of *ReplicaReponses* (see bellow)

Replica Response

Again, all replicas use the same response structure to communicate to the http server:



```java
private int statusCode;
private String message;
private Object body;

// Client check
private int replicaId;
private WalletOperationType operationType;
private byte[] serializedMessage;
private byte[] signature;
private long nonce;
```

The first block is easily self-explanatory, but to the client, the second block is more important because it contains all the information that the client needs to authenticate the server response.

The replica Id shows from each replica the response comes, the operation type represents the operation that was performed, the serialized message and the signature provides authenticity of the replica and the nonce, not only randomises the whole response, but also protects against replaying.

Custom Extractor

To save all messages from the replicas on each request, it was necessary to create our own custom extractor class, implementing the *Extractor* interface of the Bft-Smart Library. The custom class was created equal to the default class with the only difference that it would store all the replica messages, in the form of the previously explained ReplicaResponse class.

To allow multiple concurrent requests, the replica messages are stored in a *Map<Long, ExtractorMessage>* where the key is the nonce of the request since it is supposed to be unique, it clearly identifies the request and the associated replica messages.

Note: When accessing the set of replica messages for a particular request, that entry is deleted, since it is only meant to be used once. This way, we prevent this list to grow to infinity.

Client: Response Authenticity

From the client side there’s the need to verify if the server responses were not compromised.

We created a variable “`faults`” with a default value of 1 but that can be passed as an argument to the Client. This variable represents the number of faults that the client wants to tolerate.

To achieve that we implemented, **for each replica response**, the follow verifications:

Nonce - Check if returned nonce is equal to some arithmetic operation defined between client/server. Example (Nonce = Nonce + 1) .

OpType - Check if returned operation (Get Balance, Transfer Money or Generate Money Operation) was the same operation sended from the client.

Amount - Check if returned amount was the same amount sended to the client.

Status - Guarantee that every replica response has a status code of 200 (OK).

Signature - This is the last verification and ensures that the message sended by the replica server was in fact created by each replica.

In the eventuality of any of this confirmations fail it will be added 1 to a general variable controlling the number of conflicts.

If this number is greater than the variable "`faults`" previously explained the Client will reject the operation and show the following message **`"Conflicts found, operation is not accepted by the client".`**

Extra Functionalities

Unpredictable Mode

The server can run in an unpredictable mode, meaning that the server can randomly produce wrong messages.

There are two levels of errors produced:

- Replica Level
- Http Server Level

At the replica level, there is a 20% probability that the replica in unpredictable mode can write a wrong value when transfering money, generating money or just checking the balance. This type of error is meant to be handle alone by the Bft-Smart library.

At the Http server level, there are five types or errors that can be induced by the server:

1. Amount mismatch - Server returns a value different of all the replicas responses.
2. Wrong Operation - Server changes the type of the operation performed on a replica response.
3. Wrong signature - Messes up a replica response signature
4. Nonce mismatch - Changes the nonce of one replica response
5. Wrong status code - Changes a status code on a replica response

Error number One, forces an error on the server response, and does not change any of the replica responses. Error 2 through 5 are error forced on an already signed replica response, therefore, the client should analise and discover the error when verifying all the replicas responses.

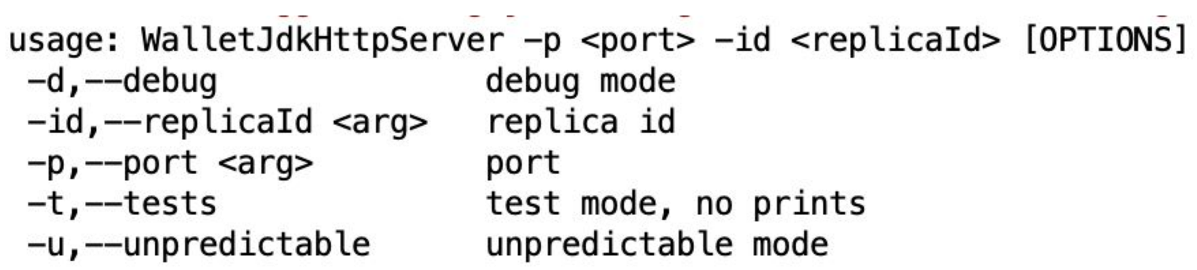
Log Level

Different log levels are implemented and can be choosen in the command line arguments when running the server and the client. Logs are present on every class of the project and uses the **Apache Log4j** library. There are three modes of logging. The default level is ***INFO***, and prints info logs, warns and errors. The second is a ***DEBUG*** level, and prints the same as the level above plus another set of debug logs, used for tracing and developers. The third log level is a test level, and does not print any logs to the console, so it won't affect the timings of the tests.

Running the program

(Enable each mode with apache commons-cli)

Server:

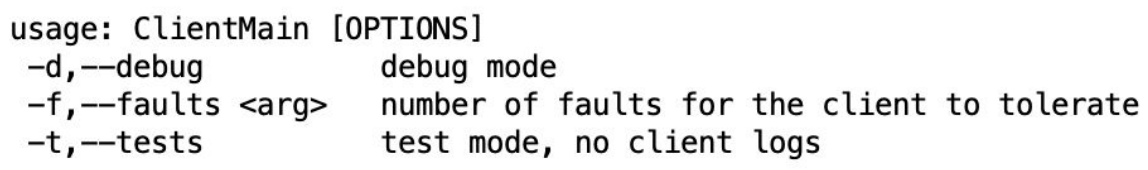


```
usage: WalletJdkHttpServer -p <port> -id <replicaId> [OPTIONS]
 -d,--debug                debug mode
 -id,--replicaId <arg>     replica id
 -p,--port <arg>           port
 -t,--tests                test mode, no prints
 -u,--unpredictable        unpredictable mode
```

The id and port arguments are mandatory to run the program, and the id represents the id of the replica. All servers must have different ids and matching the hosts.config file of the BftSmart library.

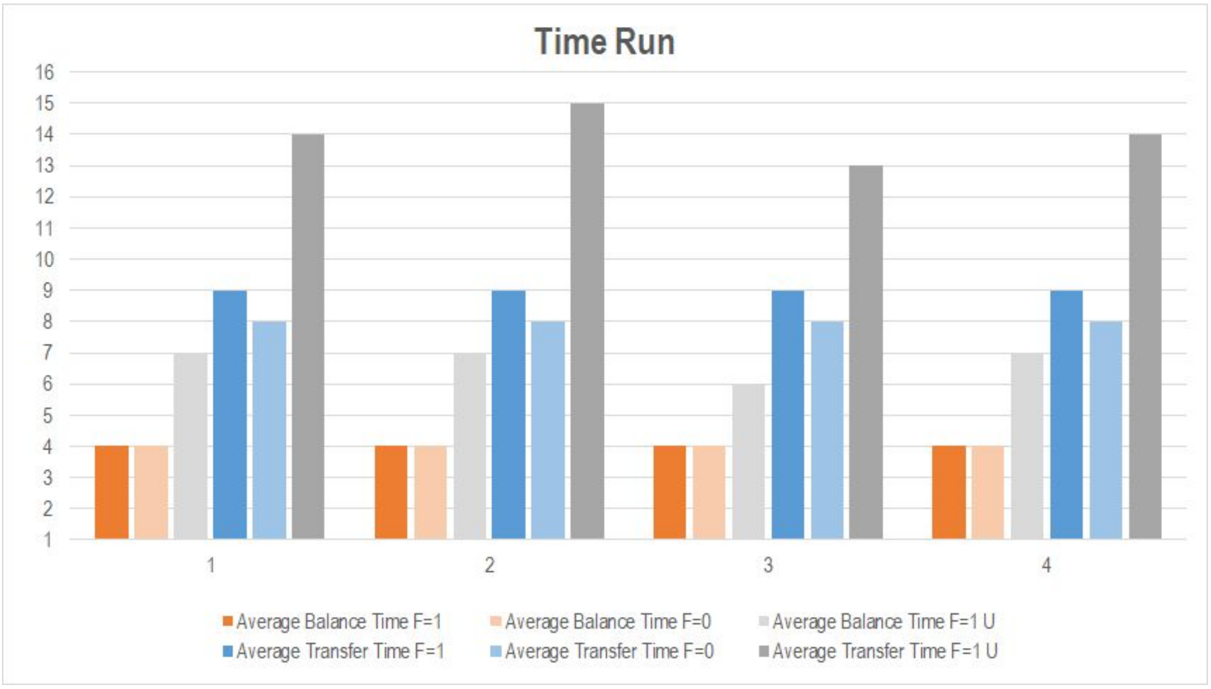
The port argument represents on which port is the Http server (not the replica) going to run.

Client:



```
usage: ClientMain [OPTIONS]
 -d,--debug          debug mode
 -f,--faults <arg>   number of faults for the client to tolerate
 -t,--tests          test mode, no client logs
```

Evaluation



The graphic above show our average times for operations (Get Balance and Transfer Money) for 4 tests with 3 different configurations.

The Y axis represent our average Get Balance and Transfer Money times in ms (milliseconds).

For the Get Balance operation we have the darker orange bar test with F ("`#Maximum number of faulty replicas" in bftSmart configuration file`) = 1 and on the lighter orange the F = 0. We can assume that without provoqued fails the Get Balance times maintain constant and it's approximately 4ms. On the other hand, Get Balance in a unpredictable environment gets worst and rises to approximately 7ms.

For the Transfer Money operation we have the darker blue bar test with F ("`#Maximum number of faulty replicas" in bftSmart configuration file`) = 1 and on the lighter blue the F = 0. We can see a very small advantage for the F = 0 with 8ms and having F=1 with a average of 9ms. Has we confirmed for the Get Balance with an unpredictable environment times get worst rising to approximately 14ms.

Conclusions

With all the work done in this project, we can then conclude that there is a trade-off between security, consistency, availability and speed. To guarantee a correct byzantine system we need to give up some speed to allow the system to analyze and try to overcome some faults.

However, in the current days, there is a bigger need for dependable and secure systems, and we are happy to trade-off a few milliseconds to ensure that we have secure and available system at all times.