FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Confiabilidade de Sistemas Distribuídos

Professor Nuno Preguiça

Trabalho Prático 2

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

Miguel Anciães nº 43367
Ricardo Amaral nº 43368

INDEX

Introduction

The goal of this project is to create a dependable system that can execute (basic) smart contracts. The system to be developed should extend the project to: (1) maintain additional state and be able to execute basic operations on the state; (2) support recovery of the servers.

The system exports seven operations:

- create(key, initial_value, type) - Add the (key, value) pair of type "type" to the database. The following types should be supported "HOMO_ADD", "HOMO_OPE_INT", and "WALLET".
- get(key) - Returns the value associated with key "key".
- get(key_prf, lower_value, higher_value) - Returns all key with key prefix "key_prf" and values between "lower_value" and "higher_value" (inclusive).
- set(key, value) - Set the value of "key" to "value".
- sum(key, value) - Sum "value" to the value of "key".
- conditional_upd(cond_key, cond_val, cond, list[(op,upd_key, upd_val)]) - Executes the list of updates if the condition holds, where the condition is parameterized by "cond", as follows:
    - 0 -> db[cond_key] = cond_value
    - 1 -> db[cond_key] != cond_value
    - 2 -> db[cond_key] > cond_value
    - 3 -> db[cond_key] >= cond_value
    - 4 -> db[cond_key] < cond_value
    - 5 -> db[cond_key] <= cond_value - The list of updates is parametrized by "op", expressing:
        - 0: set value: db[upd_key] = upd_val
        - 1: add value: db[upd_key] = db[upd_key] + upd_val
- set value: db[upd_key] = upd_val § 1: add value: db[upd_key] = db[upd_key] + upd_val

System design

Servers

The server architecture is the same as the project before, with replicated servers using the bft smart library, bitcoin like authentication system.

In addition to that, two new servers were implemented. A first one, which is only accessible by administrator that can launch a replicas of the server. The second one, is the SGX server, which is ran in a secure environment, emulated by the scone docker images, that is responsible to make operations over sensitive data, for example, when operations cannot be performed homomorphically.

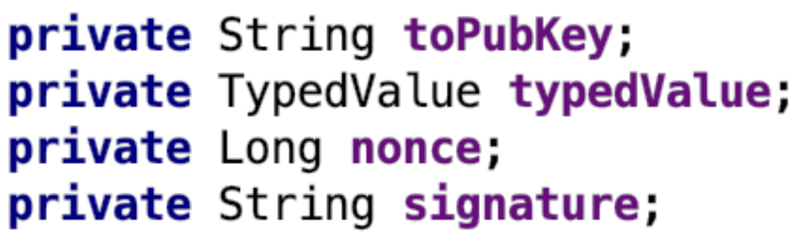
In contrast to what was done in the first project, the value now associated with a userId (user's public key) is now what we call a Typed Value. It contains a value representing the amount, encoded as a String, and a type of value, WALLET, HOMO_ADD or HOMO_OPE_INT. With this, we can differentiate the different values, and since the value itself is encoded as a String, we can check the type and converted to the desired Class, Double for the Wallet type, BigInteger for Homo_Add and for Homo_Ope_Int of even Longs and Integers for the decrypted values of the homo types.

Client

The client maintains the structure of the first project.

In depth

Create Operation (POST)



```
private String toPubKey;
private TypedValue typedValue;
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 typedValue contains the amount that will be transferred to the user along with his datatype, 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 signature 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.

With the wallet and Homo_Add datatypes, if there is already a client in the system with that userId, the amount is added to the account. With the Homo_Ope_Int, that is not possible, so, if an user is already in the system with that userId, a HTTP 400 is returned to the client.

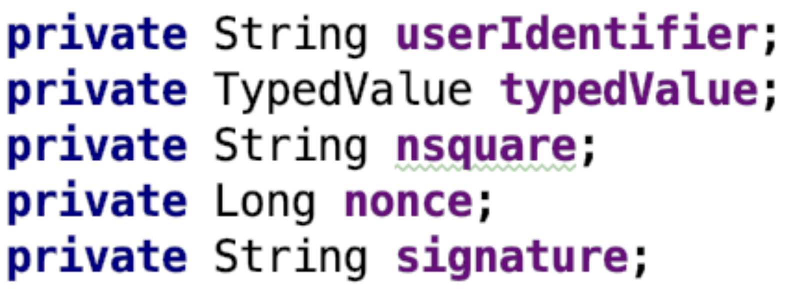
When creating accounts, the client will also send the encryption key to be stored at the replica. To be stored securely, the key is encrypted with the sgx public key. This prevents anyone else to access that encryption key other than the sgx server.

The encrypted keys are stored in the replicas, so the sgx server does not old any state.

Sum (POST)

The sum operation sums the wanted amount to an user account.

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



```
private String userIdentifier;
private TypedValue typedValue;
private String nsquare;
private Long nonce;
private String signature;
```

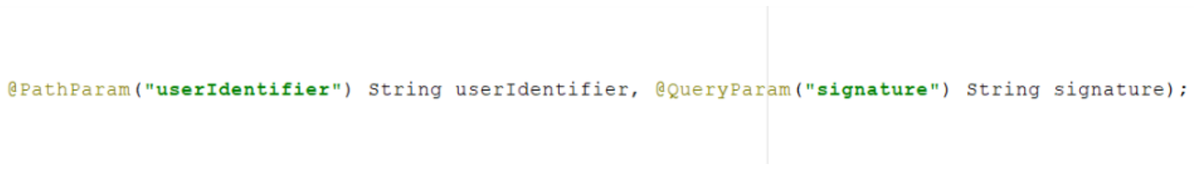
where the userIdentifier argument specifies the user that the money will be added to, the typedValue contains the amount that will be transferred to the user along with his datatype, the nsquare is an argument to the HomoAdd encytpion sum operation, a nonce to randomise the signature, and the signature of the request. The signature is created by adding all of the first four parameters of the request as a unique string, and then signing it with the private key of the client.

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.

This operation is performed normally within the servers on Wallet or Homo_Add data types. With the Homo_Ope_Int data type, it does not support homomorphic addition, so the request is transferred to the SGX server along with the encryption keys, stored securely with the sgx server's public key. Sgx server, decrypts the values and performs the operations in plain text, securely running on a secure hardware. Re encrypts the result and returns it back to the server, that returns it back to the client.

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.



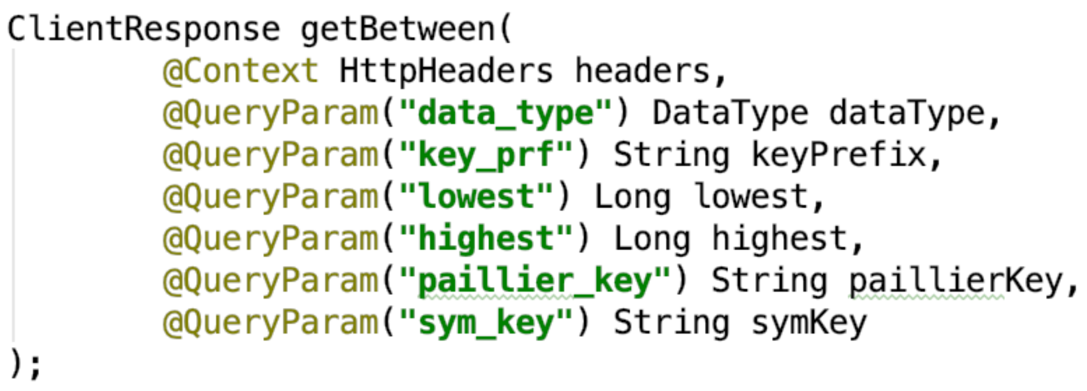
```
@PathParam("userIdentifier") String userIdentifier, @QueryParam("signature") String signature);
```

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

This request does not need to know the data type of the value to be returned, as it’s the clients job to decrypt it accordingly.

Get Balance Between (GET)

This operation returns a list of user ids (public keys) that have the amount between certain values.



```
ClientResponse getBetween(
        @Context HttpHeaders headers,
        @QueryParam("data_type") DataType dataType,
        @QueryParam("key_prf") String keyPrefix,
        @QueryParam("lowest") Long lowest,
        @QueryParam("highest") Long highest,
        @QueryParam("paillier_key") String paillierKey,
        @QueryParam("sym_key") String symKey
);
```

where the data type is the type of accounts where the operation will compute. keyPrefix is a prefix that the keys must must along the the lowest and high condition. If not provided, no key prefix will be matched. Paillier key and syn keys are actually a typo on the version delivered, as they are not used.

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.

The group decided that this operation would work on a single data type at a time. Lowest and highest are encrypted accordingly to the data type by the client, and will only be compared with the same data type. This could be done for all data types together, but with the presence of the condition_upd operation, we decided not to do so.

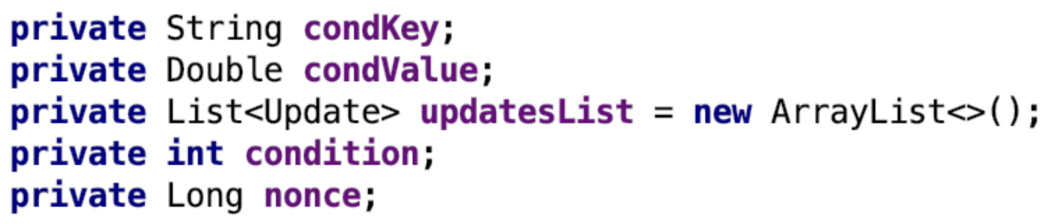
Again, with Wallet and Homo_Ope_Int data types, this comparison operation can be performed intrelly on the server, but with the Homo_Add data type there is no support for comparisons on encrypted data, so it must be performed securely on the sgx server. The data is sent along with the encryption key, decrypted on the sgx secure module, and compared in plain text. A response true of false is sent back to the replica server which constructs a list of keys that match the criteria.

Set (POST)

This operation sets the value of an account regardless of what it was before. It does not need to know the type, as it just replaces the value, and there is no need for any operations.

This operation receives the same arguments as the create client which are essentially a client user identifier, a typed value containing an amount a a type, a nonce and a signature for verification.

Condition Update (POST)



```java
private String condKey;
private Double condValue;
private List<Update> updatesList = new ArrayList<>();
private int condition;
private Long nonce;
```

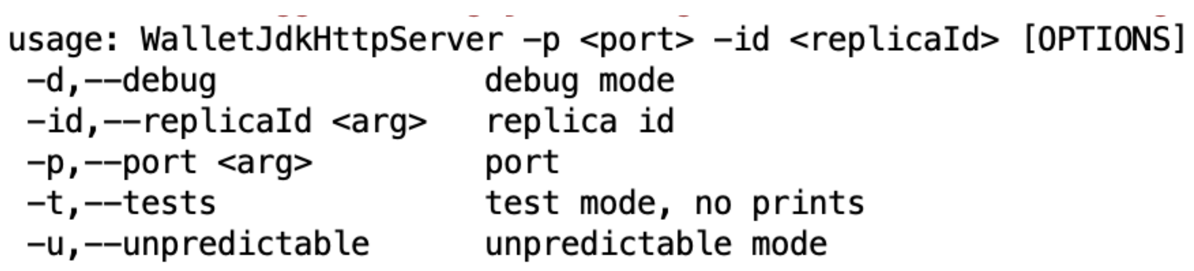
This operation perform a list of operations “updateList” if a condition “condition” holds between “condValue” and a specific value associated with the user on the DB linked with “condKey”.

In this “updateList” each Update, has the operation to be executed, a client key and a value, this value is encrypted with the private key of the user associated with that client key.

Running the program

(Enable each mode with apache commons-cli)

Server:

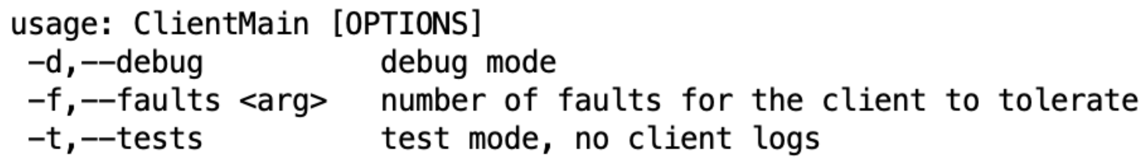


```
usage: WalletJdkHttpServer -p <port> -id <replicaId> [OPTIONS]
 -d,--debug                debug mode
 -id,--replicaId <arg>     replica id
 -p,--port <arg>           port
 -t,--tests                test mode, no prints
 -u,--unpredictable        unpredictable mode
```

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
```

Sgx Server:

```
docker run -it --privileged -it -p 6699:6699 -v $(pwd):/home -w /home
sconecuratedimages/apps:8-jdk-alpine java -cp /home/projectJar/project-v2.2.jar
rest.sgx.server.SGXServer
```

Notes

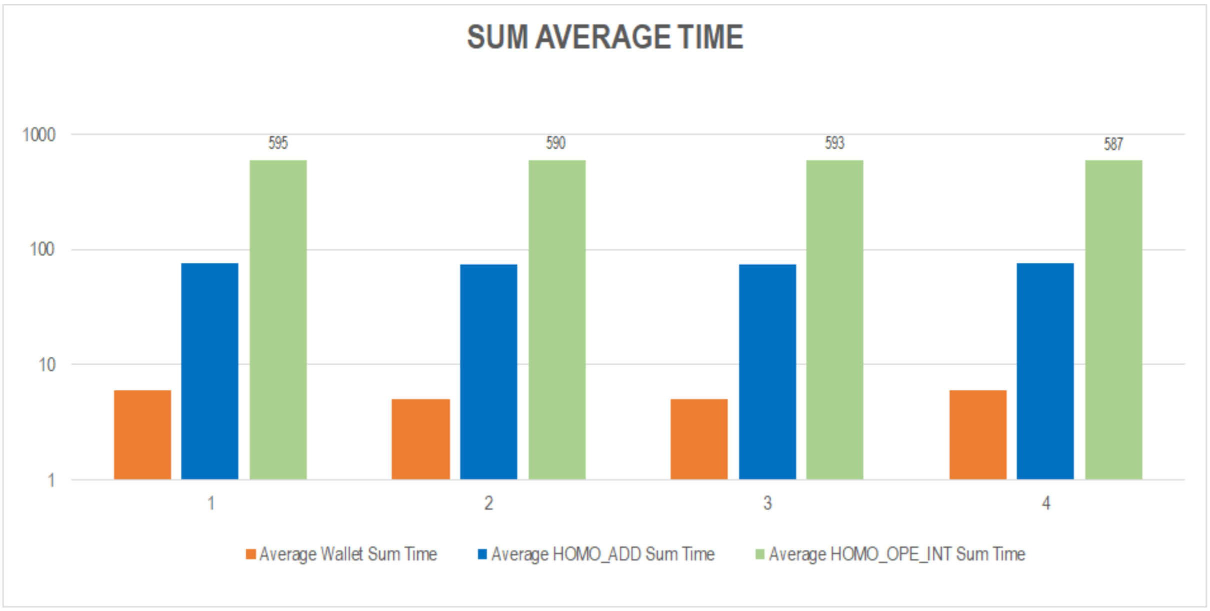
Palliet key to string:

When trying to use HelpSerial.encodeToString on the Palliet key it results in a huge string, big enough to overflow the encrypt limit of RSA. As a workaround we decided to generate an AES symmetric key, encrypt the result of HelpSerial.encodeToString with AES and then finally encrypt that result with RSA.

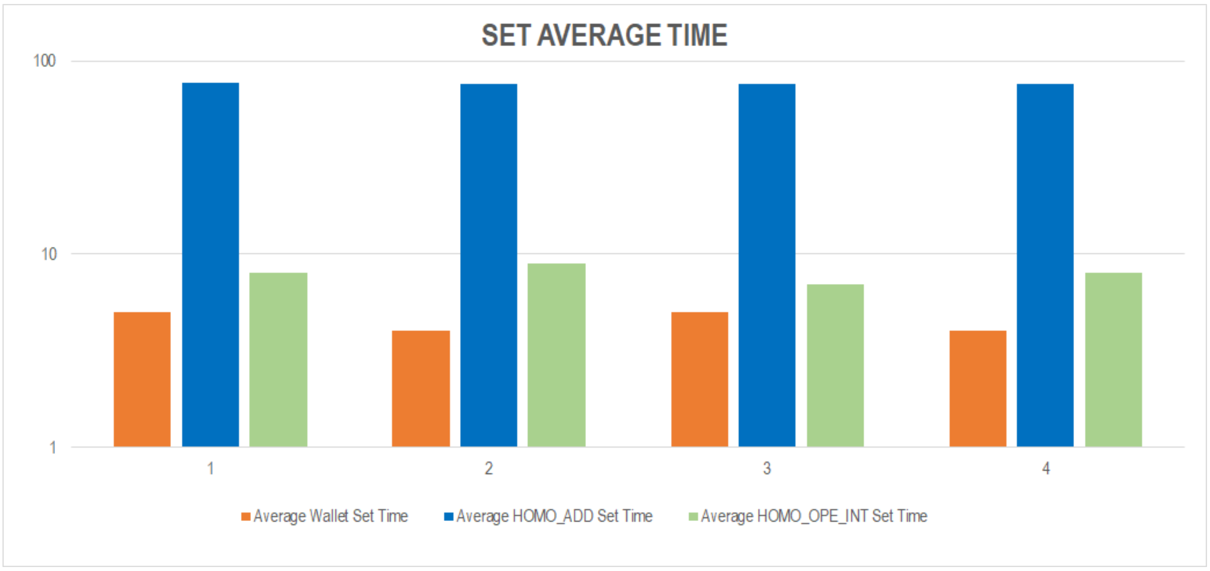
Evaluation

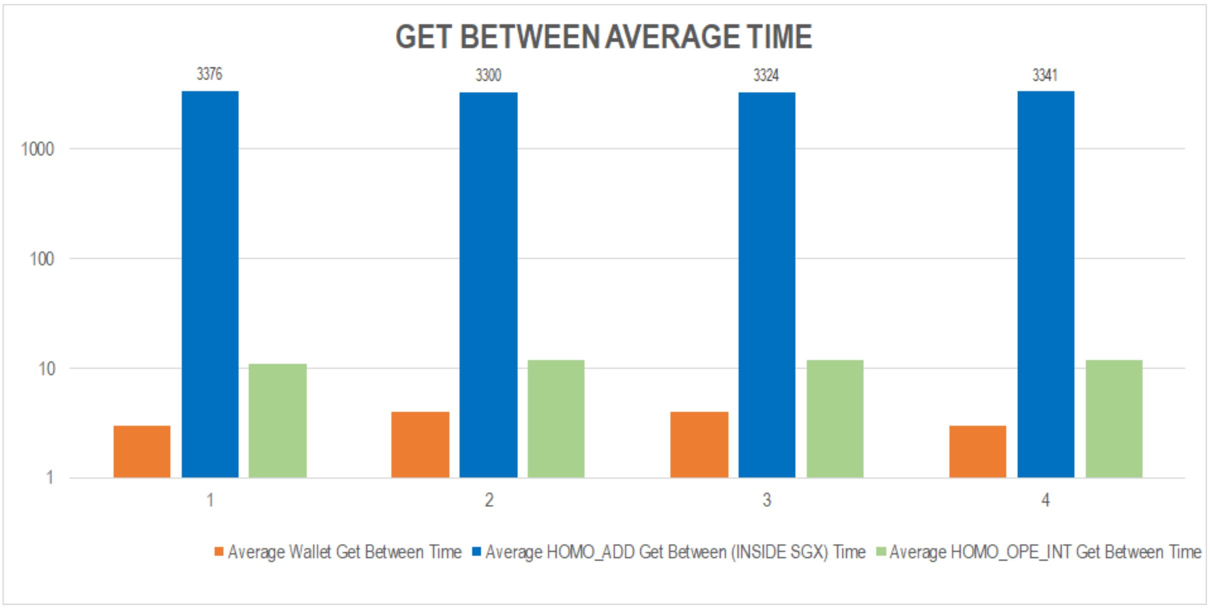
For each one of the next graphics we evaluated the average time of each operation after creating 10 users and running 3 threads at the same time during 3 minutes. The times are tested 4 times for each type of client “Wallet”, “HomoAdd” and “HomoOpeInt”.

The Y axis represent our average times in ms (milliseconds).

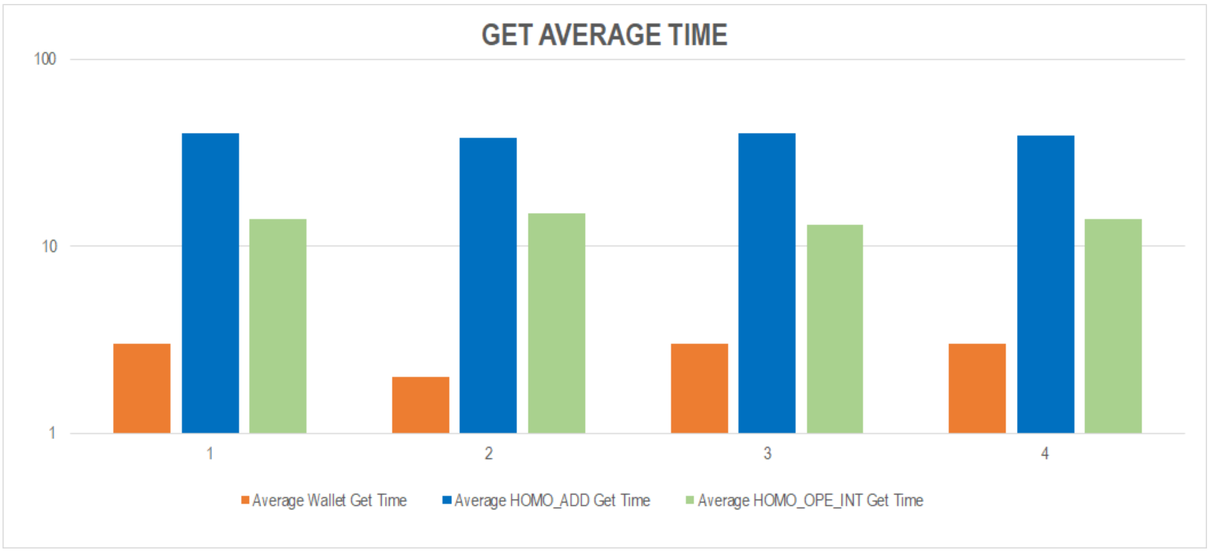


As expected as sum operations with HOMO_OPE_INT run inside SGX this is the worst performer for the sum operation. With Wallet client being the best also as expected.

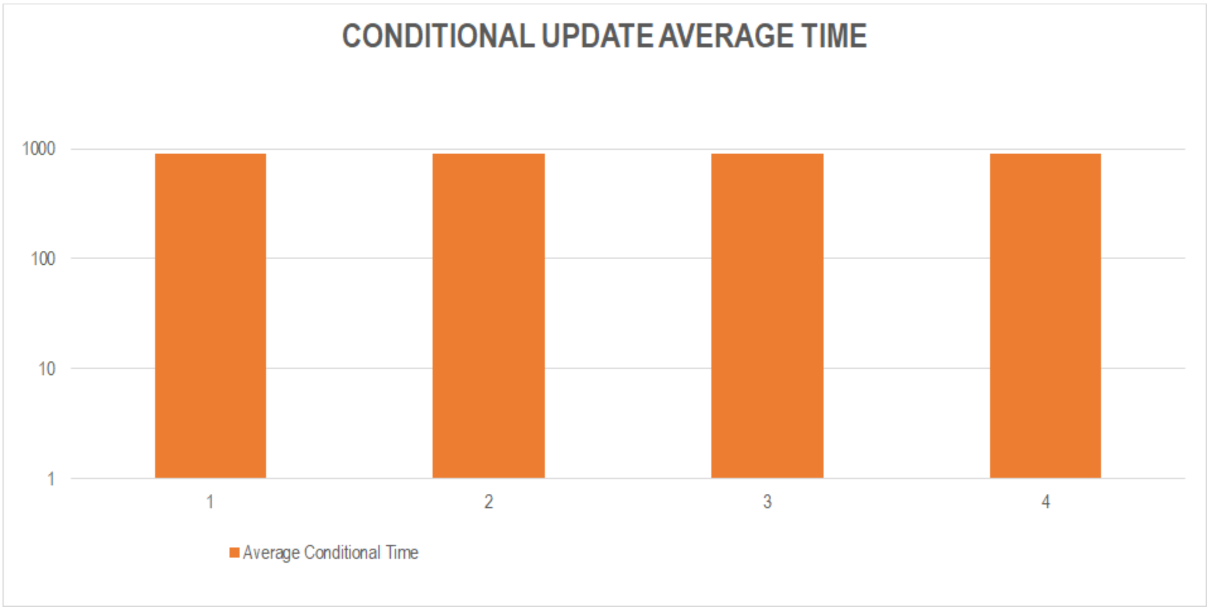




As expected,compare operations with HOMO_ADD running inside SGX, this is the worst performer for the getBalanceBetween operation.



For the Conditional Update operation we created 3 clients, one of each type and the list of updates contains the 2 operations allowed (SET and SUM) for each of the 3 clients created.



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 encrypted operations and database we need to give up speed especially when running operations inside safe environments like SGX.

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