Byzantine Fault Tolerant System for Smart Contracts *

André Rosa

DI - FCT - NOVA University of Lisbon
af.rosa@campus.fct.unl.pt

Abstract. This work assignment consisted on enriching the Digital Wallet System, developed for the first project of this course, with the ability to execute (basic) Smart Contracts. For this, in addition to the already present WALLET data type, the system was improved to support two additional encrypted data types, that leveraged on partially homomorphic encryption, enabling sum and compare operations directly on the encrypted values.
A supplementary requirement for this project was to also add support for remote recovery of replicas.
Therefore, in this report, it's presented an implementation of this system as well as a preliminary experimental evaluation on its performance.

Keywords: BFT · SMR · BFT-SMaRt · Digital Wallet · Smart Contracts · Homomorphic Encryption

* Second assignement of the CSD course

Table of Contents

1 Introduction

The objective of this work assignment was to enrich the Digital Wallet System, developed for the first project of this course, with the capability to execute (basic) Smart Contracts [1]. For this, there is the need for the system to support the creation of variables as well as to provide operations to manipulate or read them. This system already has support for variables of the type WALLET. However, those variables are in clear text and thus do not provide confidentiality on their values. Therefore, this solution leveraged on partially homomorphic encryption, a special type of encryption that enables computations to be made directly on the encrypted values, providing an encrypted result which, when decrypted, is equivalent to the result of the same operations if they had been applied on the unciphered values. In particular, these new data types correspond to homomorphically encrypted integers, HOMO_ADD_INT and HOMO_OPE_INT, that maintain the addition and total order properties, respectively.

In addition to Smart Contracts, the system must also provide a mechanism to support the secure remote recovery of failed replicas.

Therefore, in this report, it's presented an implementation of this system. Details of the implementation are presented in the following sections, being the remaining of this document structured as follows: in Section 1 is presented the overall system architecture and in Section 3 are presented the experimental evaluation results. Finally, to conclude this report, in Section 4 are presented some final observations.

2 System's Architecture

2.1 Data Model

To support Smart Contracts, the system provides the creation and manipulation of variables. Each variable has an identifier (id), its data type, and its current value. As mentioned before, there are 3 different data types: WALLET (ClearText), HOMO_ADD_INT, and HOMO_OPE_INT.

To manipulate those variables, the system exports the following operations:

1. **transfer(id_from, id_to, amount, signature)**: Transfer amount from the wallet identified by *from_id* to wallet identified by *to_id*. This operation fails if the balance of the *from_id* wallet is smaller than *amount* or if the *signature* is invalid, i.e., the transaction was not issued by *from_id* or was tampered. (Only for WALLET Type)

2. **atomicTransfer(List<Transaction> transactions)**: This operation atomically performs multiple transfers, corresponding each transaction to the previous operation. The reason why this operation exists is for the case when a client possesses its money across different accounts and wants to transfer an amount, larger than the balances of each one of its accounts, into a single address, in an atomic operation. Therefore, this allows the client to

issue only one operation instead of issuing multiple sequential transfer operations, and waiting for the completion of each one before issuing the next. Additionally, this operation also allows a client to atomically perform multiple transactions, from different origin accounts, into different destination accounts. (Only for WALLET Type)

3. **balance(id)**: Returns the amount of money associated to the wallet identified by *id* or 0 if that wallet is not registered in the system. (Only for WALLET Type)

4. **ledger()**: Returns the list of all accounts and corresponding money, that are stored in the system. (Only for WALLET Type)

5. **create(***type***,** *id***,** *initial_value***)**: Creates a variable named *id* of type *type* and initial value *initial_value*.

6. **get(***id***)**: Returns the value associated with the variable represented by *id*.

7. **get(***id_prf***,** *lower_value***,** *higher_value***)**: Returns all the keys (of any data type), that match the prefix key_prf, whose associated values are between *lower_value* and *higher_value* (inclusive).

8. **set(***id***,** *value***)**: Sets the value of *id* to *value*.

9. **sum(***id***,** *value***)**: Sums *value* to the value associated *id*, and returns the result of the sum.

10. **compare(***type***,** *id***,** *cond_op***,** *value***)**: Returns the logic value of performing the comparation of tipe *op* between the value associated with *id* and *value*. The alternatives for *cond_op* can be one of the following: $=$, $\neq$, $\leq$, $<$, $\geq$, and $>$.

11. **conditional_upd(***cond_type***,** *cond_id***,** *cond_op***,** *cond_val***,** *list*[**(***op***,** *upd_type* *upd_id***,** *upd_val***)**]**)**: Executes the list of updates if the condition holds, being *cond_op* one of the following: $=$, $\neq$, $\leq$, $<$, $\geq$, and $>$. The support update operations *op* can be either sums or sets.

2.2 Architecture

Overall, this system is divided into 4 diferent logical components, that are presented on Figure 1. Next, are explained each one of these components individually.

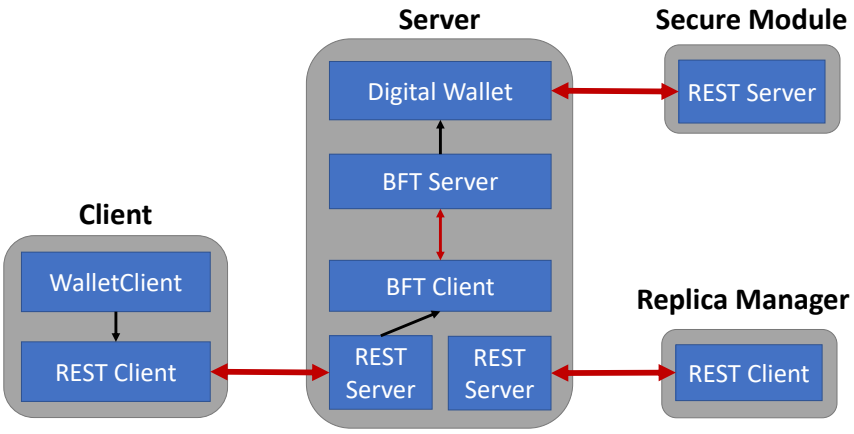

Fig. 1. Overall system's architecture.

2.3 Client

Each client maintains a list of the identifiers of its variables and their corresponding type. Upon the creation of a variable of an encrypted data type, the client generates a new key for that variable, and encrypts its value with it. These keys are exclusivly kept (stored) by the client and is then used in all the operations over that variable, either to encrypt the arguments (to sum, set, and compare values) or to decrypt the result.

All the operations are invoked by the user with plain text integers. Then, if the variable being manipulated is of an encrypted data type, the client application retrieves its associated key and encrypts the respective values before sending a request to the server. Upon receiving the result, if it's encrypted, the application resorts again to the previously obtained key to decrypts the value and present it to the user.

In the case of the get(id_prf, $lower\_value$, $higher\_value$) operation, the prefix match is done locally, at the client, obtaining a list of all the variables' identifiers (of any data type) that matched the prefix. Afterwards, the $lower\_value$ and $higher\_value$ are encrypted with each one of those variables key's, creating a list of tuples $< id, lower\_value\_ciphered, higher\_value\_ciphered >$, that is sent to the server.

Like in the previous work assignment, the communications between client and servers are done through REST.

2.4 Servers

To increase its reliability and availability, like in the previous work assignement, the system resorted to BFT-SMaRt [3] to perform Byzantine Fault Tolerant State

Machine Replication of its state across different replicas, using the traditional assumption that for tolerating f faults, the number of replicas should be $3f + 1$.

Each replica is responsible to execute all the client's operations over the storaged variables. In the case of WALLET variables, the replicas are able to execute all the operations. However, in the case of HOMO_ADD_INT and HOMO_OPE_INT, only sum and set or compare and set operations, respectively, are possible to be made directly over the encrypted values, being sum operations over HOMO_OPE_INT and compare operations over HOMO_ADD_INT impossible to be performed. Therefore, the replicas need to delegate these operations to a secure container (Secure Module), that decrypts the variables, performs the operations, and then returns an encrypted result. For this, the Secure Module needs to have access to the keys of those variables. However, this module does not contact directly with the client. Thus, for each operation that requires the server to resort to the Secure Module, the clients send an additional argument, the key associated with that variable, encrypted with a secret key, shared only between the client and the Secure Module.

Additionally, all the replicas have two additional operations: to lauch or stop them, in order to provide the option to remotly restart failed replicas. The lauch operation requieres a URL of the program to be executed, an hash to verify its integrity, and a password to only allow authorized entities to perform this operation. When a lauch operation is requested, it is only executed if the replica is not already runnig. In either cases, it returns the id associated with the current instance running. On the other hand, the stop operation requires a password and the id of the instance to stop, to prevent concorruent stop operations on the same replica to stop newly restarted instances. The operation is ignored if the id provided as argument does not match with the current instance id. Therefore, only the wanted instances are stoped.

2.5 Replica Manager

This component consists on a simple REST Client, that enables a system's administrator to manualy execute lauch and stop operations on replicas.

2.6 Secure Module

Like stated before, the replicas need to leverage on a Secure Module to execute operations that are not possible to compute over encrypted variables. In these cases, the replicas leverevage on a secure container (SCONE container [2]) to execute the operations in a private context, over the decrypted variables. On this container, only the application itself can access the unencrypted data and code, being that neither of them are accessible in clear text (not even for root users).

However, in a preleminary test phase, the performance of the getBetwenn operation for HOMO_ADD_INT was 100x slower than the other operations. Therefore, in order to improve its performance, in the case of HOMO_ADD_INT variables, instead of the client encrypting the extremes with the key associated

with each id that matched the prefix, they are encrypted only once with the secret key shared between the client and the Secure Module. This change greatly boosted the performance of the operation to about only the double of the remaining operations. Additionaly, a cache was added to this module, to store recently decrypted values and thus require less decryption operations, which slightly incresed the overall performance of the operations delegated to the Secure Module.

3 Experimental Evaluation

This Section presents the setting and results of the performed experimental evaluation of the implemented solution.

3.1 Experimental Setting

To evaluate the performace of the system, it was measured its *Throughput* and *Latency* of each operation, for each data type. The *Throughput* corresponds to the number of operations performed per unit of time, and the *Latency* corresponds to the average time required to perform an operation in the system and receive its result.

Each operation was issued sequently 100 times, being each test repetly 3 times to averege the results, in order to amortize the effects of external and uncontrollable factors.

The system's performance was evaluated on a configuration with $f = 1$ (tolerating one fault), but being all correct replicas.

Due to hardware constraints, the client application, the system's replicas (4), the Replica Manager, and the Secure Module (not running in Scone) were all executing on the same Virtual Machine, that was running Ubuntu 18.04.2 LTS with 4GB of RAM and a Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz, with 4 cores.

3.2 Experimental Results

The experimental results for lantency and throughput are presented in the Figure 2.

Overall, the operations over WALLET variables present the lowest latency and highest throughput, as expeted, since they don't require any type of cryptographic processing. All the operations of this type present similar values of latency and throughput.

Similarly, the operations over the type HOMO_OPE_INT presented almost identical results, except on the sum operation, that is delegated to the secure module.

In contrast, the HOMO_ADD_INT type was the one that presented the highest latency (and lowest throughput) overall. For this reason, one can infer that the cryptography operations (encrypt and decrypt) on this type of homomorphic encryption is heavier that the ones for the HOMO_OPE_INT type. Its operation

with the highest latency (and lowest throughput) was the getBetween operation being much higher than all the other operations (of any data type). These result migh be due to the fact that, on the one hand, the cryptography operations for this type of is much heavier thatn the others, and, on the other hand, each one of the encrypted values, associated with *id*s that matched the prefix, need to be decrypted, agravating the before mentioned problem.

In sum, as expected, the results delegated to the Secure Module presented significantly more latency than the operations performed over encrypted or in clear text variables, due to the fact that the Secure Module needs to decrypt both the variables and their arguments as well as encrypt the final result, before sending the reply to the replica that invoked the operation.

4 Final Remarks

To conclude, in this report it was presented a implementation of a Digital Wallet with support for Smart Contracts, that leveraged on the library BFT-SMaRt [3] to handle Byzantine Fault Tolerant State Machine Replication of its state. This state corresponds to a set of variables (and their associated values) than can be of three different types: WALLET (clear text), HOMO_ADD_INT, and HOMO_OPE_INT (both cipher text).

Regarding the experimental evaluation, overall the operations that were delegated to the Secure Module presented the worst results on performance, due to the fact that they require to perform encryption and decryption of values. To concluede, since all the system's components were running on the same Virtual Machine, more tests should be performed on separate machines, in order to fully evaluate the performance of this system.

References

1. Foundation, E.: Ethereum's white paper (2014), `https://github.com/ethereum/wiki/wiki/White-Paper`
2. Scontain: Scone, `https://sconedocs.github.io/Java/`
3. Sousa, J., Alchieri, E., Bessani, A.: State machine replication for the masses with bft-smart (2013)

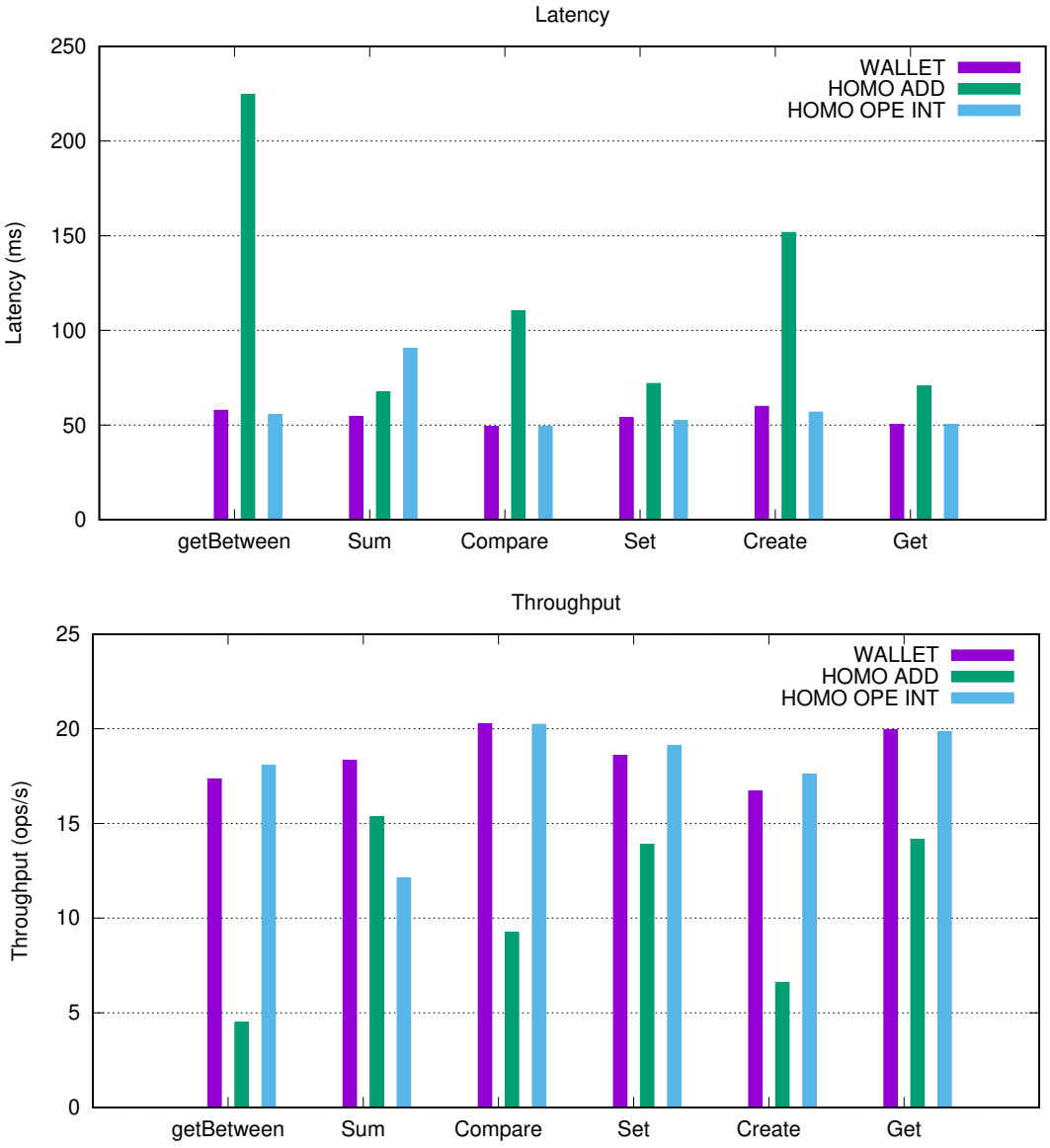

Fig. 2. Variation of the Latency and Throughput of the different system's operations.