Model of Dynamic Smart Contract for permissioned Blockchains

Adnan Imeri[1,2], Jonathan Lamont[1] Nazim Agoulmine[2], and Djamel Khadraoui[1]

[1] Luxembourg Institute of Science and Technology (LIST),
adnan.imeri@list.lu,
https://www.list.lu/
[2] Université of Évry Val d'Essonne, Evry, France

Abstract. The blockchain technology and smart contract capabilities encourage scholars and retailers on investigating the new conceptual modeling and technological opportunities for redesigning the current and future business processes. The emerge of blockchain technology indicates the new era in designing and developing the business process mainly by using smart contracts. Along with these opportunities, different challenges are presented in the current state of blockchain technology and smart contracts. Among them, the emphases cases are the alignment of the new changes on the requirements from the business process, to the smart contracts. We present a new way of modeling a smart contract that impacts on the maintainability of the enterprise blockchain-based solution. This paper shows a new approach that allows smart contracts acting dynamically over lift time changes on the business process logic specific to the use case.

Key words: Enterprise modeling, Smart Contract, Immutability, Maintainability, Design Pattern, Hyperledger Fabric

1 Introduction

Since the invention, blockchain technology has been the subject of exploration from academics and industries. The technology that stands behind Bitcoin initially was the focus of cryptocurrency industries [1]. Further, it has been extended in different domain of application, and many governmental, non-governmental, industrial organization are exploring the technological design of the blockchain to improve their activities, business process, etc., [2][3]. The first generation of blockchain, i.e., Bitcoin comes with limited technological capabilities in terms of designing complex business processes [1]. The emerging of the Ethereum (ETH) blockchain framework, with the possibility of deploying smart contract (SC), enabled a new way of execution of application over the blockchain network [4]. The combination of blockchain and SC enabled a new market for a decentralized application that provides a new level of automation of many business processes [7]. Simultaneously, with the opportunities offered by

blockchain and SC, there are various concerns to considers for blockchain-based applications. These issues are on designing a blockchain-based application that should behave as intended by the end-user. Further, the security and privacy issues, performance issues, and programmable issues and maintainability of already running blockchain solutions are amongst the most highlighted concerns when considering designing a blockchain-based application [8]. The research and industrial communities already faced before-mentioned problems, and for overcoming these issues, there are discovered several design patterns as the best practices from the community, that help researchers and developers on designing reliable smart contracts (SC) [9]. Through this research, we intend to provide a new model for improving SC designing and execution. The research problem discussed in this paper is related to the maintainability of immutability feature of SC. Indeed, the code of a SC cannot be modified due to the blockchain immutability property once it is deployed. For any required modification on the existing SC, the deployment of this SC into a blockchain, meaning to have a new reference of this SC on the blockchain. This leads to complicated maintenance tasks according to the number of contracts to update, the eventual static cross-references. Our goal is to enable a way to integrate a dynamic behavior into SCs without deploying them again.

Regarding this research, we implemented a proof of concept, for a particular use case, and it is accessible on a public Git.

The rest of the paper is organized as follows. Section 2 presents an extensive study over the blockchain technology and exploration of Hyperledger Fabric as the selected blockchain framework for this study. Section 3 defines the study problem and a use case. In section 4 we present related works studies. Section 5 shows our conceptual approach and proof-of-concept implementation. The evaluation of the solution is presented in section 6 along with discussion and future works.

2 Background: Blockchain Technology

Blockchain is a decentralized-distributed append-only database that enables storing of the immutable set of transactions, organized in a hash tree (Merkle-Tree) [10] [11]. The transactions present any kind of data such as financial data, textual or numeric data, that are encapsulated on transactions by users [12] [13]. These transactions are gathered together into a candidate block by miners that compete with each other intending to validate this new block [11]. Miners are high-performance computers that is allowed to add a new block on the chain of blocks. The block, besides transactions root it contains other significant components such as timestamp, block header, mathematical difficulty puzzle usually called as “nonce”, the hash of the previous block, thus forming a chain of blocks or ”blockchain” [14].

Blockchain network: The blockchain network is an extensive set of devices that communicate in a peer-to-peer mode (P2P). The nodes are computers-servers that are geographically distributed, and they contain the same copy of

the ledger. The consensus algorithm that allows these notes to agree on the state of the data, removes the need for a trusted third party, thus making blockchain entirely decentralized [11].

Consensus protocol: For agreeing on the state of the data, blockchain uses a consensus mechanism, e.g., Proof-of-Work, Proof of Stake, etc. Once the miner solves a computational mathematical puzzle, it distributes the "nonce" to the other miners. All the miners verify the solution of the puzzle by applying the "nonce", then they approve the adding of the new block in the chain of blocks, and all nodes are updated by adding a new block [14].

Immutability: The transactions added on the blockchain are cryptographically signed. Once the transactions appear in blockchain they remain immutable. Any tendency to change them will change the transaction root of Merkle Tree, and the consensus algorithms will deny this change by comparing the current changed block with other blocks from other nodes that contain the same blockchain [13].

Non-repudiation: The properties of immutability and data integrity, enforces the properties of non-repudiation [11].

Availability: The blockchain network maintains the availability, even if some nodes fails to response [1].

On-Chain vs Off-Chain: The design properties of the blockchain allows storing a limited amount of data on chain. These data are the most significant information, transaction and meta-data (hash values) referenced from large files that are stored off-chain [15]

Permission-less and permissioned blockchains: For the permission-less blockchains, there is not required any authorization for accessing the main network of the blockchain, mining transactions and exploring the executed transactions. In contrary, for accessing permissioned blockchain, nodes are required to have permission by the administrator of the blockchain network [13]. The permissioned blockchain, such as Hyperledger Fabric [6] and the SCs are subject of this study.

2.1 The semantic of smart contract

Smart Contract (SC): is a stand-alone program that is executed when certain conditions are fulfilled. It might execute asset transfers, execute another contract, fulfill the conditions from any business process [16] [17]. At the highest level it is expressed by sort of object-oriented programming language, e.g., Solidity [16], JavaScript, Go, and Java. The encoding of SC is sourced from natural language, legal contracts, business agreements, and other domain-specific sources. The enforcement of the agreements that are reached between parties involved, are further translated into SC programming code and added on the blockchain [18]. Nowadays the most prominent SC-enable technology is ETH [19], and another blockchain platform such as Hyperledger Fabric is providing high-level programmable SC. By design, SC has some technological features. SC remains immutable once it is deployed on the blockchain, by contract transaction. Among the main design principles of the SC are [20]:

- SC address (ID), a hash value, that identifies the SC on the blockchain.
- Owner ID, a 256bit hash value, which indicates the owner of the SC.
- *Internal storage*, the SC has its private storage, it holds its execution code with pre-defined parameters, some amount of virtual currency (own balance of the SC).
- *Execution costs*, for the blockchain platforms, what need digital coins to perform a transaction mining, the execution of the SC has its own costs.
- *Enforcement*, the contractual obligation that is expressed in the terms of SC, e.g., transfer assets once the goods are received, are automatically performed as contractual obligations and enforced by SC.
- *Invoke another SC*, by sending a transaction to the SC address.
- *Autonomous*, the predefined parameter on the SC code allow the SC to change the state of on blockchain if executed successfully.
- *Self-Execution*, the SC is a set of autonomous executable agents that are trigger by predefined parameters on the SC code or executed from environments parameters.
- *Event/Method* are set of instruction that are executed in the SC for fulfilling intended task.

SC are invoked by users, by sending the transaction to the contract address, e.g., some amount on it and parameters to execute the targeted SC [17][18][20]. Furthermore, an SC can call synchronously another SC and also as synchronously off-chain service [17]. SC supports an ordered logic that follows the ordering rules *if this than that (IFTTT)*. This semantic is called event ordering logic (or order-execute), and SC events (function invocation) are executed as they are placed. Under this logic, if the current order passes then, continue on the second order, otherwise, throw an exception [19]. This logic is present in almost all SC enabling blockchain platforms, expect in Hyperledger Fabric, which is one of the main focus in this study and it is explained in detail in the following section 2.2.

2.2 Hyperledger Fabric

Hyperledger Fabric (HF) is an open-source blockchain that allows designing and developing private or consortium blockchain-based solutions with a focus on the business-oriented use cases [6][21][22]. HF has a modular and configurable architecture that allows users to adopt blockchain technology for their use case. HF is implemented in the GoLang programming language and supports users with different tools. HF provides Software Development Kits (SDK) for various programming languages [25] such as GoLang, Java, NodeJS, and Python [21][22]. HF allows writing of SC in general-purpose programming, which is beyond domain-specific language i.e., "Solidity for ETH".

2.2.1 HF Overview and functionalities

Entities: Consortium and Organization. HF defines different entities regarding the participants involved in the project. The HF *network* is managed by a group of organizations gathered into a consortium. Each organization should

manage its nodes and should have at least one Certification Authority (CA) node, and one Orderer node [21][22].

Nodes, Peer, Orderer, CA and Client. HF defines four types of nodes. The main type of node is **Peer** nodes which manage the blockchain mechanisms. Peers can join channels, and they can host different SC over each channel[1]. **Orderer** nodes ensure the **consensus** of the HF blockchain network and keep the peer's ledgers consistent [21][22]. Each peer must be connected to an Orderer node. They also execute SC requests in the case where Orderer gives agreement (after checking permissions) [21][22]. **Certification Authority** nodes ensure identity delivery via digital certificates, typically required by each organization to enroll new members. CA is a private root Certification Authority provider which can manage digital identities of participants [21][22]. **Client** nodes can connect to and interact with peers deployed over the network [21][22]. All node types are provided as docker containers [23].

2.2.2 Smart Contract (SC) or Chaincode in HF

In the HF jargon, a SC is referred to as "chaincode". Chaincode, or SC, is the blockchain embedded application which is typically a running script inside a peer. Every SC that is executed maintains its database, to store data or the state of the code execution. This database is called WorldStateDB. This is a database local to each channel and SC whose values are continuously kept up-to-date by reading assets, values changes from the ledger blocks. The ledger keeps all value changes in blocks, while the WorldStateDB keeps the last current value for each asset [21][22].

Amongst the main SC main concepts that are necessary for any kind of application on HF:

- **Participant** refers to a user that is defined in the SC. For handling different users in our application, we extend the "Participant" type [21][22];
- **Asset** expresses every kind of valuable thing that can be exchanged between persons (i.e. participant), and therefore changes its ownership [21][22];
- **Transaction** is a way to exchange the ownership of the asset.
- **Event** is the only way to communicate data outside the blockchain network. Events are broadcasted on channels and can be caught by an external application that has access rights to listen to the desired channel [21][22].

3 Problem definition: The issues of maintaining the immutability of the SC

Immutability: For permissionless or permissioned blockchain, an SC remains immutable once it is deployed on the blockchain. This means, all the terms and the logic implemented behind the SC, remains unchanged over time. Thus, in

[1] A channel can be considered as a independent sub-blockchain.

case we need to make some changes in the logic of the SC (add or remove events), we should redeploy it, and a new hash will generate as an ID for that SC. From the user, that needs to use this SC, they are obliged to know the newest address of SC, before being able to invoke it. Otherwise, the user will call the old one, since its hash value is already mapped on the current SC. Figure 1 is illustrating these issues by showing changes to the SC address.

The new address of the contract should be distributed to all stakeholders that are invoking this smart contact. The concern is that all the other smart contracts that have invoked this SC (now with new hash address) should be changed. That means that we have to reconfigure the entire system (i.e., all objects need to have the new address). For instance, if there are thousands of contracts that call the same SC, then, this would be extremely difficult to reconfigure it (cf. maintenance), and the performance will become a concern for the blockchain-based applications. Furthermore, this implies the automation capability of the process decreases in this sense.

This problem is a concern for enterprises that are intending to move some of their business logic over the blockchain. Indeed, to keep the maintainability, we need modularity, which means that a complex problem should be divided into several minor problems further, to solve them much more easily. Just as using code libraries, using several inter-connected SCs becomes useful for high-level business processes. Thus, we can assume a complex system using several SC with cross-references and automation. However, using static addresses for the contracts that are hardcoded in the contract logic is leading to uncomfortable maintenance, as explained above.

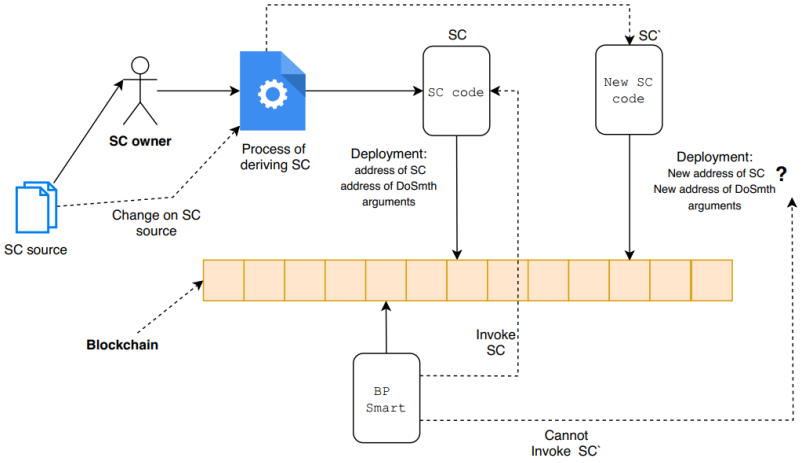

Fig. 1. The basic problem of immutability of smart contracts.

Cross-references: From the enterprise modeling perspective, involving several SC and cross communication to perform high-level tasks, still, exposes the same

problems. Figure 2, shows the issues of logic flows for cross-reference SC. In case of a SC logic update, either from the caller side, either from the executor side, there will be another SC address to know. For instance, considering two SC, where SC1 calls SC2, and if we only update SC2 (to SC2′), then we must change SC2 address reference into SC1 to point to SC2′. That implies also to change SC1 to SC1′. These sidesteps are not ideal for a system in production due to heavy infrastructure maintenance. However, this case, it's even worse when we have cross-references, i.e., if SC1 calls SC2 and SC2 calls SC1, then we fall into a deadlock situation because SC1 and SC2 have a hardcoded address of the other by exchange. That comes from the fact that we cannot guess the address of a future SC to hardcode it in advance in the logic. Thereby, here there no workaround to do, except avoiding bidirectional cross-references for maintainability.

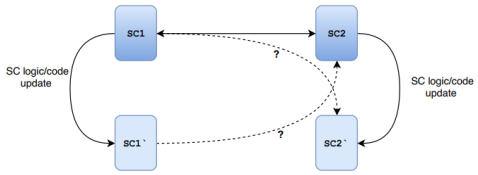

Fig. 2. The problem of cross-references for smart contract.

3.1 Use case of temperature checking. The issues of SC in a dynamic environment

Temperature checking: Considering that we have a SC which aims to check the temperature that are collected by one IoT sensor, storing its data directly into the blockchain. This SC is checking on demand the current temperature (from the IoT device), and if this temperature exceeds a specific threshold, a set of users should be notified based on the pre-defined condition in a particular use case, e.g., transporting dangerous goods, that is sensitive to a temperature degree [24].

Detailing the features of this SC, it has three core functionalities. The first one is to **collect** the temperatures provided from an IoT sensors. This IoT device is authenticated and transmitting its data directly in the blockchain. The second functionality of this SC is to **set** and change the temperature threshold of detection. The third functionality is to **notify** all involved stakeholders when the temperature threshold is exceeded, for taking the necessary actions need to avoid any possible consequences.

The issue is regarding the *threshold* and the *set* transactions. The common approach is to define the threshold as a static constant into the SC code (i.e. as an hardcoded and immutable variable). Certainly, we will need to update the contract each time when the constant need to be changes, and this exposes enormous problems as we mentioned above.

This use-case doesn't deal explicitly with cross-references because there is only one SC, but the mechanism of the solution remains the same. Indeed, for both two SCs, by storing the target SC address as a variable (in place of the *threshold*) that allows to dynamically change that address and then unlock the deadlock situation. For overcoming these issues we propose a new way of managing the SC, by storing the threshold constant as a variable into a SC asset. The SC asset is an editable variable that allows to update it dynamically (as required by the use case), and avoiding the change of the code of SC. In the following section 4, we present some related works towards designing maintainable SC, and further in section 5, we present the model of dynamic SC for permissioned blockchain.

4 SC designing, modeling and development: Related Works Studies

SC is the subject of many studies from academia, research organizations and also from industry. Designing SC is one of the main challenges highlighted recently by scholars and industry. The literature review shows that there are presented several design patterns for supporting the best practices for designing a SC. Mainly these design patterns are on "security of SC" [25], "structural patterns" [26], "privacy issues" [27], "performance issues" [25, 28, 29]. The research from [35], proposes an upgradable SC by using a proxy pattern. Research from [26] summarizes SC design patterns based on the existing SC and further apply some of the design patterns in a real work blockchain-based application for traceability. There are presented different classifications of patterns for designing SC such as "action and control", "authorization", "lifecycle", "maintenance", "security" are presented in [30]. Within certain classes of the design patterns for SC, our research is essentially linked to the maintenance pattern, and intend to improve the current way of maintaining SC. The main related design patterns proposal **satellite** and **contract relay** [30], are the essentially related to our research works. The satellite pattern is using two SC: *satellite* and *base* [30]. It enables to update a variable from *base* by calculating the value from the *satellite* thanks to the address reference of the *satellite* held into *base*. This allows us to dynamically change the value of the variable by just upgrading the *satellite* contract with the newest calculus and updating the *satellite* address in the *base* contract. In the contrary to this solution, we do not use the second SC, but rather the asset notion from HF which can be compared to a internal variable into ETH (requiring the only update by transaction call). Further, the **contract relay** pattern is using two contracts: *base* and *relay*. The *relay* contract serves as an entry point in order to provide the latest version/address of the *base* contract, and then forward any call to it. This is a proxy enabling to upgrade the *base* contract without upgrading the user entry point (*relay*). Nevertheless, the drawback that the newer data storage needs to be consistent to avoid data corruption. These two design patterns propose solutions including good practices for maintenance issues that well fit public blockchains. It requires sophisticated

programming skills in order to implement it correctly, and further maintain it. In our case, we are focused on permissioned blockchain, particularly HF as one of the main blockchain framework used by enterprises [31]. To the best of our knowledge, none of the studies mentioned above does consider the dynamicity of the SC based on the parameters of its functions (class methods), nor the applicability of these SC design patterns on permissioned blockchain, e.g., HF. Furthermore, our study measures in section 6 that there are issues of scalability with satellite patterns in case there is a large number of transactions.

We propose a new way of managing SC in a dynamic way by providing a prominent solution for permissioned blockchains.

5 Dynamic SC for permissioned blockchain based on dynamic parameterization

5.1 Dynamic parameterization

We present an approach that allows defining SC, specific to a use case, that will have a static code deployed on the blockchain, but it will run dynamically. Mainly the dynamic part of these SC remains the parameters of their transactions. We propose the usage of the blockchain technological features to store data, which further enables the possibility to store "dynamic parameter" (DynParam[2]) into it. That is considered a variable or an asset following the HF terminology. This variable leads on relying the SC code on that internal data (i.e. constant) in order to have a dynamic behavior for the cases when in the case when the *DynParam* is updated, in an SC that has immutable (static code). Emphasizing that providing this *DynParam* as arguments of the SC transactions is an external input (e.g., from the external API call, a.k.a. "Oracle" in the ETH community).

5.2 State machine representation

In Figure 3 we show how *DynParam* is working for the two functionalities **Set** and **DoSmth** that the SC has. *Set* corresponds to the ability to set (if doesn't exist) or update (if exist) the dynamic parameter that will be stored in the blockchain. That variable can be of any type, even though it is usually string or integer. And the *DoSmth* transaction can be of any purpose while it is using this *DynParam* to adapt the behavior of the code according to its value. Further, if the *DynParam* is not set/defined, the SC cannot run (cf. Locked state). If *DynParam* is set, we can run the *DoSmth* functionality, which will be one behavior (or let's say state 1). If we update by setting another value in *DynParam*, then the *DoSmth* transaction will change in consequence (i.e.

[2] The dynamic parameter term presents a constant (which is static for the time being) and it will change when a specific SC is called for updating its value, then globally it turns to be dynamic for long-time point of view.

behavior/state 2, 3, ... N), still based on the same static code/logic. This solution assumes that the *DynParam* is given by one authorized user from the outside of the blockchain and checked by the transaction itself to accept or revoke it.

Moreover, for automation purposes, we can easily extend that solution by substituting the user by an automated call of the SC to an external database to getting back the new value for the *DynParam* while the user identity is still known and allowed by the contract.

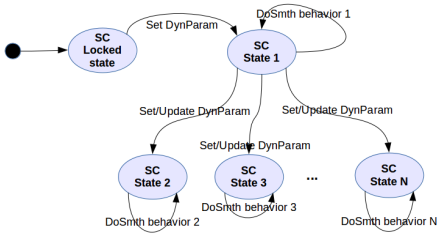

Fig. 3. State machine representation of the solution

5.3 Use case implementation

The enterprise based model is showed on Figure 4, and for the implementation is completed by using the Hyperledger Composer[3] v0.20.4 over HF v1.4.1 This solution is accessible on GitHub [32]. On the left side of the model, there are presented "On Chain" components of the model, while in the right part of the model, there are presented components that supports blockchain solution. For developing this model, three SC for performing the necessary functionalities expressed in transactions, e.g., *Collect* (TX1), *Set* (TX2), and *Notify* (TX3). The *DynParam* and *Temperature* are defined as assets and also *Temperature-Exceeded* as an event to notify stakeholders when the temperature threshold is exceeded. The *Collect* transaction serves to collect the last temperature value from the sensor. *Set* is used to define and change the needed dynamic constants. In our case, it is unique and named "Threshold_Tabc", which is hardcoded in the *Notify* transaction code to avoid users the need to know the name of this parameter. The *Notify* transaction serves to check if the temperature is exceeded the threshold (DynParam) defined by the *Set* transaction. In the case where that happens, *Notify* will generate and send the *TemperatureExceeded* event to alert the stakeholders allowed to read that event. Thus, *Notify* is illustrating the usage of the dynamic constant through access to "Threshold_Tabc". So, an update of the threshold does not require an update of the *Notify* thanks to the *Set* transaction and the use of assets.

[3] Composer is now deprecated but still usable, next tools getting the succession and doing quite the same are named Convector & Hurley.

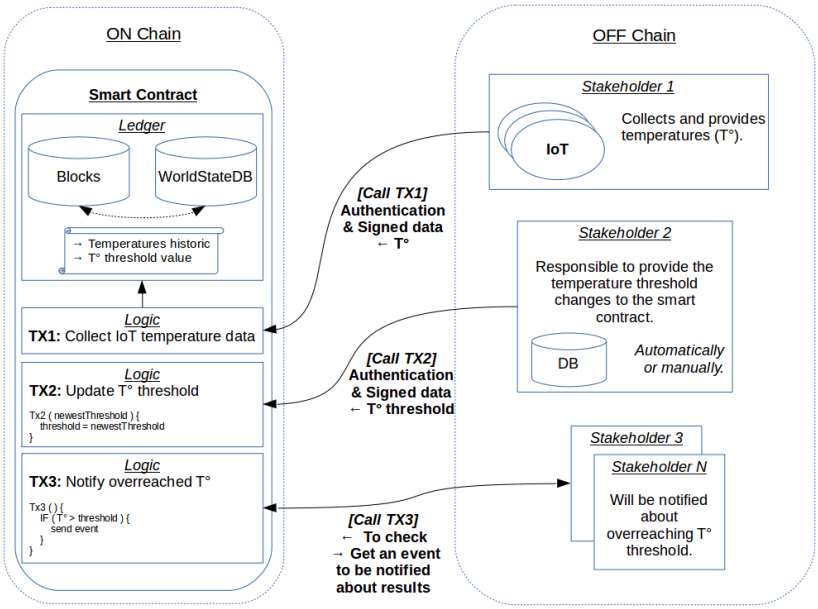

Fig. 4. The conceptual solution for the illustrated use case.

6 Evaluation and discussion

We evaluated our proposed solution by comparing it with the Satellite design pattern proposed on [30] and previously presented in section 4. The satellite pattern has three main transactions: the first one is used to update the address referring to the satellite deployed, the second transaction serves to process and calculate the value of the variable relaying to intermediate call of a satellite, and the third one serves to use the calculated result stored in a variable in order to adapt the contract behaviour by doing "something" indent by user, based on that variable. So, each satellite is dedicated to do one single thing, which implies that we need to deploy another satellite when the calculation has to change, but we can still reused a old one if that calculation has already been defined and deployed. Compare with satellite pattern, our approach uses two transactions: *the first one serves to update the internal variable whose the value is provided as argument of the transaction*, and *the second one uses that value as intended by user*. In our approach we assume the calculation for any variable is performed off-chain, as presented in 4.

The methodology is the following: *1)* We implemented the satellite pattern over ETH framework by using Truffle and Ganache tools; *2)* We implemented our solution using HF by using Convector and Hurley tools; *3)* We executed both scripts for testing both test-cases [32], in order to collect empirical data, and for being able to compare both metrics. It worth noting that our scripts are running sequentially, meaning that, it start new transaction only if the previous has been

completed. Also, to avoid distorting the results, we reinitialize the network after each calculated point (i.e. a group of transactions). Moreover, time spent by the setup and deployment of the network and contracts is not taken into account, as global running time on metrics results, as we intend to evaluate the required time for updating a constant/parameter.

The observed metrics are the number of the transaction resulting over the blockchain. For each update of the variable, the time spent to process all transactions, the total weight of all transactions gathered into blocks, and finally, the number of blocks created. Our scripts process the following set of input transaction[4]: 2, 4, 8, 10, 20, 30, 40, 50, 100, 200, 500, 1000. Mainly the technical differences between both test-cases are: 1) *observed networks* (a local ETH versus a local HF); 2) Blockchain settings regarding the *blocks creation*: ETH builds a block each 15 seconds where its size is fixed to 1KB, whereas HF builds a block each 2 seconds or if there are more than 10 transactions per block or if the block size exceeds 99MB); 3) Benchmark *scripts* for test-cases (NodeJS for ETH through Truffle against Bash for HF). The tests are performed by using the same set of updating transactions, and we use the same processing power environment: a 64bits GNU/Linux 4.15 Virtual Machine running Ubuntu 18.04.3 LTS, with 8GB of RAM and 3x 3.40GHz.

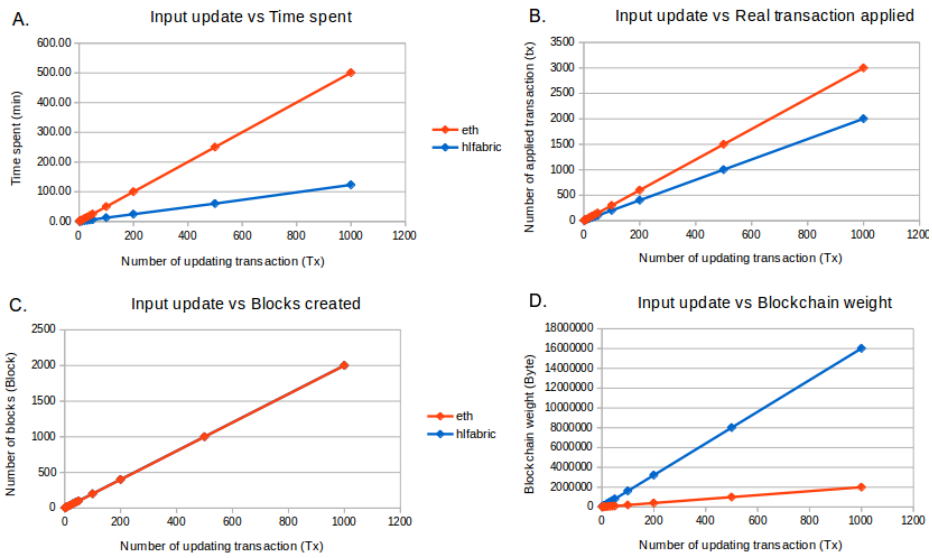

Fig. 5. Results comparing our solution against satellite pattern

[4] Inputs are grouped by a set of N updates resulting to obtain one point for each set. Here, the input set named "update" refers to the number of updates we want for the internal variable, independently of the test-case implementation. In other terms, for one update of the variable the effects are measured in the metrics, e.g., how many real transactions are applied for one requested logical update.

The figure 5 is composed of four different graphs and shows the results of this study. The x-axis of the graphs shows the number of updating transaction batches. Reminding that for the satellite pattern for one input transaction leads to three applied transactions (showed in orange colors in all graphs), whereas for our solution that results in exactly two transactions (showed in blue color in all graphs). The graph on B shows these evidences and in our quantify our solution as more efficient. The graph in A shows the total time needed to execute both solutions. The results from A, proof that ETH is around five-time slower than HF. Next ascertainment, the graph on C shows the number of created blocks. The two curves are superposed. There is a two factor between input and output (i.e. one update leads to two created blocks). However there is a difference that cannot be seen on the graph, HF is still using two blocks in addition to the double inputs. This is due to the need for HF to create the variable before being able to update it (where ETH needs to create the Satellite before using it). Moreover, the number of the created block may vary in a real network where transactions might be provided by peers in the meantime following a distributed way (against the sequential way used here for test-cases). Finally, the graph on D exposed the total blockchain weight resulting in comparison to the input sets. We can see that ETH is eight times lighter than HF. This is due to the default block size and the sequential execution of transactions. In fact, HF has in average blocks of 8KB and 1KB for ETH. And because of the sequentially processing, we do not exploit the storage of several transactions in each block. One update for ETH gives 2 blocks (with 1+2 transactions), on the other side, one update for HF gives 2 blocks (with 1+1 transaction).

6.1 Discussions

This approach impacts directly the quality of the blockchain-based solution. In this paper, we propose a solution that facilitates the maintenance tasks with a focus on SC. Our approach is enabling the definition of dynamic parameters stored as an asset (cf. HF) instead of being hard-coded in the SC logic, as any classical constants/parameter. The use of assets allows updating the value of this variables through a transaction without the need to upgrade the SC itself. For the proposed approach, a proof of concept (PoC) is developed for supporting our conceptual solution and providing access to this solution regarding our defined use case.

The modeling part of this approach allows extending on other use cases, e.g., legal reasoning use case. Since the laws are subject to change, they might be seen as obstructions for developing a sustainable blockchain-based solution. By employing our approach, the “law articles” might be saved on the assets and they might be change-over-time without disturbing the entire system.

Regarding the security, this exposed *set* transaction causing changes must be restricted to some of the stakeholders who are responsible for maintaining the system. For the case of permission blockchain, e.g., HF, the stakeholder must be an authenticated and authorized user. In the case of a public blockchain, e.g.,

ETH, this transaction must be restricted to the owner of the SC or a specific list of the authorized users hard-coded in the contract.

For future works, we intend to extend the current approach and investigate the possibility of implementing our solution in different blockchain frameworks i.e., Quorum [33] and Corda [34].

References

1. Satoshi Nakamoto. Bitcoin - A Peer-to-Peer Electronic Cash System,2008. URL: https://bitcoin.org/bitcoin.pdf
2. Blockchain Survey: Blockchain gets down to business, https://www2.deloitte.com/content/dam/insights/us/articles/2019-global-blockchain-survey/DI_2019-global-blockchain-survey.pdf, Retrieved 18 October 2019
3. Golosova, J., Romanovs, A.: The Advantages and Disadvantages of the Blockchain Technology. In the 6th Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE), IEEE, pp.1-6, (2018)
4. Vitalik Buterin. A next generation smart contract and decentralized application platform, http://blockchainlab.com/pdf/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf. 2016. Retrieved 19 October 2019
5. Nick Szabo. Smart Contract, http://www.fon.hum.uva.nl/rob/Courses/ InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html. (1996). Retrieved 19 October 2019
6. Hyperledger - URL: https://www.hyperledger.org/. Retrieved 19 October 2019
7. Mendling, J., Weber, I., Aalst, W.V.D., Brocke, J.V., Cabanillas, C., Daniel, F., Debois, S., Ciccio, C.D., Dumas, M., Dustdar, S. and Gal, A.: Blockchains for business process management-challenges and opportunities. ACM Transactions on Management Information Systems (TMIS), (2018)
8. Alharby, M., van Moorsel, A.: Blockchain-based smart contracts: A systematic mapping study. arXiv preprint arXiv:1710.06372. (2017)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software Addison-Wesley. Reading, MA, (1995)
10. What is a merkle tree? beginner's guide to this blockchain component, https://blockonomi.com/merkle-tree/. Retrieved 19 October 2019
11. Zheng, Z., Xie, S., Dai, H. N., Chen, X., Wang, H.: Blockchain challenges and opportunities: A survey. International Journal of Web and Grid Services, 14(4), pp.352-375. (2018)
12. Tasca, P., Tessone, C. J.: Taxonomy of blockchain technologies. Principles of identification and classification. arXiv preprint arXiv:1708.04872.(2017)
13. Xu, X., Weber, I., Staples, M., Zhu, L., Bosch, J., Bass, L., Pautasso, C. and Rimba, P.: A taxonomy of blockchain-based systems for architecture design. In IEEE International Conference on Software Architecture (ICSA). pp. 243-252. (2017)
14. Antonopoulos, A. M.: Mastering Bitcoin: Programming the open blockchain. " O'Reilly Media, Inc." (2017)
15. Xu, X., Pautasso, C., Zhu, L., Gramoli, V., Ponomarev, A., Tran, A. B., Chen, S.: The blockchain as a software connector. In 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA). pp. 182-191. (2016)

16. Solidity — Solidity 0.5.12 documentation, https://solidity.readthedocs.io/en/v0.5.12/. Retrieved 21 October 2019
17. Kolluri, A., Nikolic, I., Sergey, I., Hobor, A., Saxena, P.: Exploiting the laws of order in smart contracts. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 363-373. ACM. (2019)
18. Mik, E.: Smart contracts: terminology, technical limitations and real world complexity. Law, Innovation and Technology, 9(2), pp.269-300. (2017)
19. Home — Ethereum, https://ethereum.org/. Retrieved 21 October 2019
20. Luu, L., Chu, D. H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security (pp. 254-269).(2016)
21. (2019). Buildmedia.readthedocs.org, https://buildmedia.readthedocs.org/media/pdf/hyperledger-fabric/latest/hyperledger-fabric.pdf. Retrieved 21 October 2019
22. Introduction to Hyperledger Fabric, https://hyperledger-fabric.readthedocs.io /en/latest/blockchain.html.Retrieved 21 October 2019
23. Docker - Website. URL: https://www.docker.com/.Retrieved 19 October 2019
24. Imeri, A., Khadraoui, A., Khadraoui, D.: A Conceptual and Technical Approach for Transportation of Dangerous Goods in Compliance with Regulatory Framework. 12(9), (pp.708-721). Journal of Software (JWS). (2017)
25. Wohrer, M., Zdun, U.: Smart contracts: security patterns in the ethereum ecosystem and solidity. In 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE) (pp. 2-8).(2018)
26. Liu, Y., Lu, Q., Xu, X., Zhu, L., Yao, H.: Applying design patterns in smart contracts. In International Conference on Blockchain (pp. 92-106).Springer, Cham. (2018)
27. Alharby, M., van Moorsel, A.: Blockchain-based smart contracts: A systematic mapping study. arXiv preprint arXiv:1710.06372.(2017)
28. Frantz, C. K., Nowostawski, M.: From institutions to code: Towards automated generation of smart contracts. In 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W) (pp. 210-215). (2016)
29. Michiel Mulders: Smart contract safety: Best practices and design patterns, https://www.sitepoint.com/smart-contract-safety-best-practicesdesign-patterns/, Retrieved 19 October 2019
30. Wöhrer, M., Zdun, U.: Design patterns for smart contracts in the ecosystem. In 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) (pp. 1513-1520). (2018)
31. Top Blockchain Platforms of 2019 for Blockchain Application: https://www.leewayhertz.com/blockchain-platforms-for-top-blockchain-companies/. Retrieved 22 October 2019
32. Gr4pha/hyperledger-dynamic-smart-contract.. GitHub.: from https://github.com/Gr4pha/hyperledger-dynamic-smart-contract. Retrieved 22 October 2019
33. Quorum - jpmorganchase/quorum: A permissioned implementation of supporting data privacy. https://github.com/jpmorganchase/ quorum. Retrieved 22 October 2019
34. Corda: An open source blockchain platform for businesses https: //www.corda.net/. Retrieved 22 October 2019
35. Clearmatics/smart-contract-upgrade, https://github.com/clearmatics/smart-contract-upgrade. Retrieved 25 October