Dynamic Smart Contract for permissioned Blockchain

Adnan Imeri*
ITIS Department - Trusted Service Systems
Luxembourg Institute of Science and Technology (LIST)
and Université of Évry Val d'Essonne
Esch-sur-Alzette, Luxembourg
adnan.imeri@list.lu

Jonathan Lamont
ITIS Department - Trusted Service Systems
Luxembourg Institute of Science and Technology (LIST)
Esch-sur-Alzette, Luxembourg
jonathan.lamont@list.lu

Nazim Agoulmine
Informatique, Bio-informatique et Systèmes Complexes (IBISC)
Université of Évry Val d'Essonne
Évry, France
nazim.agoulmine@univ-evry.fr

Djamel Khadraoui
ITIS Department - Trusted Service Systems
Luxembourg Institute of Science and Technology (LIST)
Esch-sur-Alzette, Luxembourg
djamel.khadraoui@list.lu

***Abstract*—The blockchain technology and smart contracts capabilities encourage scholars and retailers on investigating the new conceptual and technological opportunities for redesigning the current and future business processes. The emerge of the blockchain technology indicate the new era in designing and developing the business process mainly by using smart contracts. Along with these opportunities, different challenges are present in the current state of blockchain technology and smart contracts. The limited programming abstractions in low level, the novelty of the technology and changes occurs continuously on the blockchain platforms are among the main sources of issues of design reliable 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. 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.**

***Index Terms*—Blockchain, Smart contract, Immutability, Maintainability, Design Pattern, Hyperledger**

I. 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 Ethereum framework enabled a new way of execution of application over the blockchain network [4]. Ethereum is based on an Ethereum Virtual Machine (EVM), a "Turing complete" machine that enabled a new way for designing decentralized application based on smart contract (SC) [4]. The concept of Smart Contracts is quite old [5], [6] but has become popular with the success of Ethereum, and extensively used in other blockchain frameworks such as Hyperledger [7]. The combination of blockchain and smart contract enabled a new market for a decentralized application that provides a new level of automation of many business process [8]. Along with the possibilities offered by blockchain and smart contracts, there are numerous issues to considers while designing a blockchain based application. These issues are on designing a blockchain based application that should behave as intended by the end user. Security and privacy issues, performance issues, and programmability issues are among the most highlighted concerns when designing a blockchain based application [9]. The research community already faced such 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 [10].

Through this research, we intend to provide a new method for improving smart contract designing and execution. The research problem discussed in this paper is related to the immutability feature of smart contracts. Indeed, the code of a smart contract cannot be modified due to the blockchain immutability property once it is deployed. So, any change implies to deploy back again a new smart contract integrating the changes, which leads to maintenance tasks more or less complicated according to the number of contracts to update, the eventual static cross-references. So, our goal is to enable a way to integrate a dynamic behavior into smart contracts without deploying them again. Regarding this research, we implemented a proof of concept, for a particular use case, and published it on a public Git.

The rest of the paper is organized as follows. Section 2 presents an extensive study over the blockchain and detailed concept of Hyperledger as the selected blockchain framework

for this study. Section 3, shows the related work studies. In section 4, we abstract our research problem. The solution regarding this problem is presented in section 5. Finally, our conclusion and future works are presented in section 6.

II. 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) [11] [12]. The transactions present any kind of data such as financial data, textual or numeric data, that are encapsulated on transactions by users [13] [14]. These transactions are gathered together into a candidate block by *miners* that compete with each other intending to validate this new block [12]. *Miners* are high-performance computers that is allowed to add a new block on the chain of blocks. The **block**, besides transactions root and other significant characteristics such as timestamp, block header, mathematical difficulty puzzle usually called as "nonce", etc., it contains the hash of the previous blockchain thus forming a chain of blocks or "blockchain" [15]. These blocks history shapes the blockchain ledger.

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 [12].

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 the new block in the chain of blocks, and all nodes are updated by adding a new block [15].

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 [14].

Non-repudiation: The properties of immutability and data integrity, enforces the properties of non-repudiation [12]

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 [16]

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 [14]. The permissioned blockchain, such as Hyperledger [7] and the smart contracts are subject of this study.

Smart Contract: Is an autonomous computer program running on the blockchain [17]. SC is a stand-alone program that is executed when certain conditions are fulfilled. It might execute transfer assets, execute another contract, fulfill the conditions from any business process [17]. By design SC has technological characteristics, it remains immutable (identified by unique hash value), once it is deployed on the blockchain. The enforcement of the agreements that are reached and added on the blockchain [18]

A. Hyperledger Fabric

Hyperledger [19], [20] is firstly a consortium of different communities which gathered many projects hosted by The Linux Foundation around blockchain topics. Hyperledger is also a bunch of frameworks and tools. So in other words, it contains a batch of tools, documents, code examples, and more; all dedicated to design business oriented blockchain solutions. Hyperledger was born in 2016 to satisfy the lack of available open source frameworks in order to develop blockchain applications.

In this section, we will discuss about the **Hyperledger Fabric** framework [19]. There are many other projects into Hyperledger [21]. As framework examples, we can quote *Hyperledger Sawtooth*, *Burrow*, *Iroha*, and *Indy*. On the other hand, there are also tools like *Hyperledger Composer*, *Caliper*, *Explorer*, *Cello* and many other [22]. Each framework and each tool focus on different point for solving particular issues. For instance, Burrow is a permissioned implementation of Ethereum; Composer simplifies the business logic implementation over Fabric; Explorer is a tool to view the blockchain data and dissect the content of blocks; and so on. Thus, the Hyperledger goal is to provide a common base to focus on embedded application, rather than spends times to redefine the wheel and deal with internal blockchain mechanisms [19].

1) Overview & Functionalities: Hyperledger Fabric [19] is implemented in the GoLang programming language. It basically provides some Software Development Kits (SDK) for various programming languages such as GoLang, Java, NodeJS, and Python available today. Fabric is providing the blockchain mechanism, the networking and privacy mechanism, in order to define a **private & permissioned blockchain** network. Basically, Fabric is delivered as Docker [23] container nodes which can be easily deployed to define a network. Once the network is defined, then it is possible to deploy embedded application (i.e. "Smart Contract") on peers. In the Hyperledger environment, these smart contracts are named "Chaincode". In the following, "chaincode" and "smart contract" will have the same meaning: an application that is running inside a blockchain (i.e. hosted by some network peers). Chaincodes are currently developed in Fabric either with GoLang, NodeJS or Java [24].

2) Key concepts: The Figure 1 gives an overview of the Fabric key concepts [25] which will be described in the following [24].

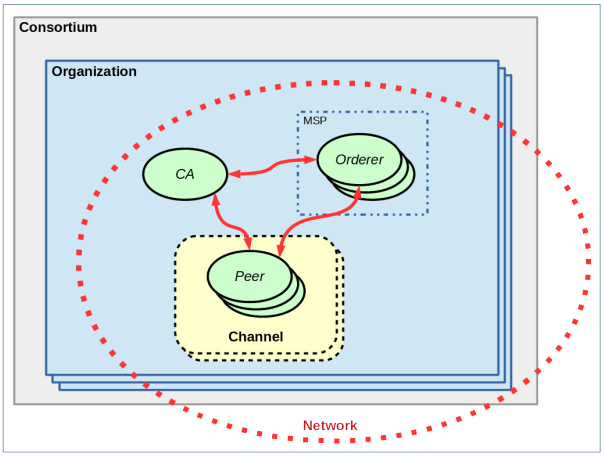

Fig. 1. Hyperledger Fabric key concepts overview.

a) Entities (Consortium & Organization): Fabric defines different entities related to real actors. Basically, the Fabric network is defined and managed by a group of organizations gathered into a consortium. Each organization should manage their nodes and must have at least one Certification Authority (CA) node and one Orderer node [24].

b) Private Sub-Networks (Channels): Channels are providing a private communication link between peers. That is a way to segregate the network into a private sub-network composed of a subset of members/peers. Communications onto each channel is cyphered and controlled by Orderer nodes and CA nodes. Because the network is private and permissioned every action applied by organizations over the network must be done through a specific channel with the right permissions and credentials. Note that a smart contract is mandatorily installed over a channel, which leads to install the contract on each peer belonging to that channel [24] [26], [27].

c) Nodes (Peer, Orderer, CA & Client): Fabric defines four types of node. The main type of node are **Peer** nodes which manage the blockchain mechanisms. Peers can join channels, and so they can host different chaincodes over each channel. Each peer must be connected to an Orderer node. They also execute chaincode requests in the case where Orderer give agreement (after checking permissions) [24].

Orderer nodes ensure the **consensus** of the Fabric blockchain network and keep the peer's ledgers consistent. Several types of consensus algorithms are currently available and can be plugged in Fabric: *solo*, *kafka* and *raft* [26], [27]. These two last protocols are Crash Fault Tolerant (CFT) which means they are able to resist up to N/2 system failures in a network composed of N nodes, while no guarantees are provided against malicious nodes. Put another way, we need 2F+1 nodes to resist to F faults. On the other side, there is still planned for future Fabric releases a Byzantine Fault Tolerant (BFT) consensus protocol to resist up to N/3 failures of any kind including malicious nodes and Byzantine faults (cf. Byzantine General Problem). BFT is more complex, safer and more energy-consuming rather than CFT. However, because Fabric is a permissioned blockchain and nodes have already known digital identities, so they are assumed to be not malicious by default. That is why this type of blockchain are usually using CFT instead of BFT, while BFT is more relevant for public blockchains where there are more chances for nodes to become malicious. Lastly, we can note that the Fabric network scalability related to consensus complexity depends on the number of Orderer nodes over the network [**?**].

Certification Authority nodes ensure the identity delivery via digital certificates, typically required by each organization to enrol new members. Fabric CA is a private root Certification Authority provider which is able to manage digital identities of participants. Certificates should fit the X.509 form. Fabric CA nodes can be replaced by any other external CA providers which are compatible with Elliptic Curve Digital Signature Algorithm (ECDSA) [24].

Finally, **Client** nodes are docker containers like each nodes types, which are able to connect to a peer deployed over the network. It should be used for network usage and administration processes.

d) Smart Contract (Chaincode): Chaincode, or Smart Contract, is the blockchain embedded application which is typically a running script inside a peer. Every chaincode that is executed maintains its own 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 chaincode whose values are continuously kept up-to-date by reading asset values changes from the ledger blocks. Put another way, the ledger keeps all value changes in blocks, while the WorldStateDB keeps the last current value for each asset [24].

Besides that there are some chaincode concepts that need to be explained: **Participant**, **Asset**, **Transaction**, and **Event**. Whatever the application that we aim to develop, we must use these concepts in the code [24].

- **Participant** refers to a user which is defined in the chaincode. So if we want to handle some different kind of users in our application, we just have to extend the Participant base type [24];
- **Asset** symbolizes every kind of valuable thing that can be exchanged between persons (i.e. participant) and so changes its ownership [24];
- **Transaction** is a way to exchange the ownership of asset. This process must not be deleted, modified or repudiated by anyone whatsoever. Additionally, it should be time proofed, so must preserve the order of execution [24];
- **Event** is a way to notify data outside the blockchain network. Events are broadcasted on channels and can be caught by an external application which have access rights to listen to the desired channel [24].

So, the following short sentence is summing all this up: *"Over the Fabric blockchain network, transactions allow*

participants to exchange assets, and the outside world can only be notified about that by an event.".

3) Membership Service Provider: The Membership Service Provider (MSP) [24] is a **central trusted authority** which governs valid identities for an organization [28], [29]. It can be seen as a centralized Public Key Infrastructure (PKI) that identifies and uses the different CAs of the network to **deliver and control digital identities** for network members. By default, the MSP uses X.509 certificates as identities over a traditional PKI hierarchy based on this four elements: Digital Certificates, Public and Private Keys, Certificate Authorities, and Certificate Revocation Lists [30].

The MSP access control takes place at different levels: Network, Channels, and Nodes. A *Network MSP* defines who are the network members who are allowed to perform administrative tasks (e.g. creating a channel). A *Channel MSP* is required to manage channel members according to their roles and the channel policy. Finally, a *Local MSP* are deployed into each peer and orderer nodes in order to do the same things as the Channel MSP, except that the scope is restricted to the current peer or orderer node [31] [24].

The security part of Fabric network can be summed up by the two main following things. First, the MSP which uses the CA to deliver and control digital identities for network members. Second, there is Transport Layer Security (TLS) protocol which is used to protect the network against eavesdropping by ciphering the end to end communications between nodes [32]. We can also mention that the access control over Fabric can be defined at both network (nodes) and application (chaincodes) levels. The MSP manages the network accesses, and there are rules (cf. Access Control List) that can be defined into the chaincode logic to manage the application accesses [31].

4) Ways to interact with embedded applications: Here we suppose that the chaincode is actually deployed and up running over some peers of the blockchain network. Typically, in order to use the smart contract, its address over the blockchain must be known, the transaction name to call as well, and the parameters to give. The address of a smart contract is often a hash value. however, over Fabric (a private blockchain), the smart contract address is defined by its name and version, which must be a unique couple. In addition, we must know the IP address of at least one peer hosting the chaincode to connect to. Then, we can connect to the peer to send the transaction request with its parameters to the chaincode [33].

In practice, to do so, we have several possibilities in the Hyperledger environment. The first one, which is also the rawest one, is to build manually the packet to send to the peer embedding all elements previously described. The second method is to define a script using directly the Hyperledger Fabric API to do it with some high-level objects. Alternatively, as a third way, we can use some most high-level tools like Hyperledger Composer, coming on the top of Fabric, to get a most high-level API for scripting, or even an HTTP API (supporting websocket as well) thanks to the Hyperledger Composer REST Server [33] [31].

5) Transaction validation protocol:

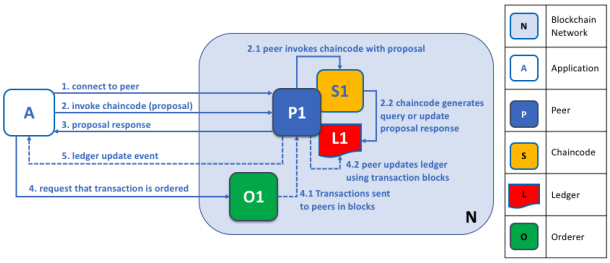

Fig. 2. Hyperledger Fabric transaction validating protocol.

Hyperledger Fabric goes through three steps in order to execute a transaction: the **proposal** phase, the **endorsement** phase, and the **validation** phase citehttpsbui6:online.

As shown by the Figure 2 [34], we assume an external application which wants to execute a transaction into the chaincode (S1) deployed over the peer (P1). Once the application is connected to the peer, it should be able to invoke chaincode functionalities. So it submits a transaction proposal to the peer which generates and evaluates the proposal into its own ledger, and reply to the application by a proposal response. Then, the application is able to request that transaction is ordered to be finally executed. Once this request is sent to the orderer node, orderer will put the transaction in a block to guarantee a single order for each transaction. This step done by orderer is called an endorsement. Peers will then process transactions onto their ledgers. During this last phase, the peer executes the transaction once again, and in the case where it gets same results as in the proposal step, the transaction is applied to its ledger. The transaction is now validated, and an event might be emitted to notify the application that the transaction is recorded [**?**].

III. Related Works Studies

Smart contracts are subject many studies from academia, research organizations and the business market perspective. Designing smart contracts 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 smart contract. Mainly these design patters are on "security of smart contracts" [35], "structural patterns" [36], "privacy issues" [9], "performance issues" [37] [38] [35]. The research from [39], propose an upgradable smart contract by using proxy pattern. Research from [36] summarize smart contract design patterns based on the existing smart contracts and further apply some of the design patterns in a real work blockchain-base application for traceability. There are presented different classifications of patterns for designing smart contracts such as "action and control", "authorization", "lifecycle", "maintenance", "security" are presented in [40]. Within certain classes of the design patterns for smart contracts, our research is essentially linked to the maintenance pattern, and intend to improve the current way of maintaining smart contracts. The

main related design patterns proposal **satellite** and **contract relay** [40], are the essentially related to our research works. The satellite pattern is using two smart contracts: *satellite* and *base* [40]. It enables to update a variable from *base* by calculating the value from the *satellite*. This allows 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 contrary to this solution, we do not use the second smart contract, but rather the asset notion from Hyperledger (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, with 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 in the correct way, and further maintain it. In our case, we are interested in permissioned blockchain, particularly Hyperledger Fabric as one of the main blockchain frameworks used by industry [41]. To the best of our knowledge, none of these studies does not consider the dynamicity of the smart contract based on the parameters of its functions, for the Hyperledger Framework. We intend to present a new way of managing smart contracts in a dynamic way by providing a prominent solution for permissioned blockchains.

IV. Smart Contract issues

Either permission-less or permissioned blockchain, a smart contract remains immutable once it is deployed, which means all the terms and logic remains unchanged, this because of immutability properties of the blockchain. Thus, in case we need to make some changes in the logic of the smart contract, we should redeploy it, and a new hash will generate as an ID for that smart contract. The new address of the contract should be distributed to all stakeholders that are calling this smart contact. The issues here is that all the other smart contracts that have called this smart contract (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, there are thousands of contracts that call the same smart contract. So, 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 to solve it in a simple way. Just as using code libraries, using several inter-connected smart contracts becomes visible for high-level business processes. Thus, we can assume a complex system using several smart contracts 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.

A. Smart contract problem definition

a) Immutability: In the context of the maintainability of the blockchain-based systems, the immutability of the smart contract also imposes some challenges. This because, any time we update (add new functionality) overs smart contract, that leads to a new address (a new smart contract). So, from the user point of view who wants to use the smart contract, it obliges to know the newest address before to be able to call the new one. Otherwise, he will call the latest one. Figure 3 is illustrating these issues by showing changes to the smart contract address.

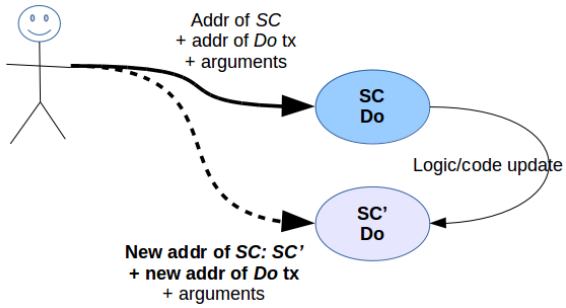

Fig. 3. The basic problem of immutability of smart contracts

b) Cross-references: From the industry perspective, involving several smart contracts and cross communication to perform high-level tasks, still, the problem remains the same. Figure 4, shows the issues of logic flows for cross-reference smart contracts. In case of a logic update, either from the caller side, either from the executor side, there will be another address to know. For instance, considering two smart contracts whose SC1 calls SC2, and if we only want to update SC2 (to SC2′), then we must change SC2 address reference into SC1 to point to SC2′. That implies to also change SC1 to SC1′. This workaround is not accurate for a system in production due to heavy infrastructure maintenance. However, in this case, its even worse if 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 reciprocity, just because we cannot guess the address of a future smart contract to hardcode it in advance in the logic. Thereby, here there no workaround to do, except avoiding bidirectional cross-references for maintainability.

B. Use-case definitions

Assuming that we have a smart contract which aims to check the temperature whose are collected by one IoT sensor storing its data directly into the blockchain. This smart contract

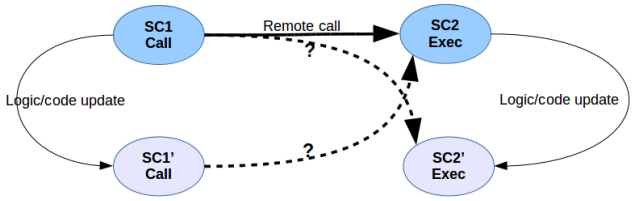

Fig. 4. Illustrating the problem for extended to cross-references for smart contract.

is checking on demand the current temperature (from the IoT device), and if this temperature exceeds a specific threshold, some 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 [42].

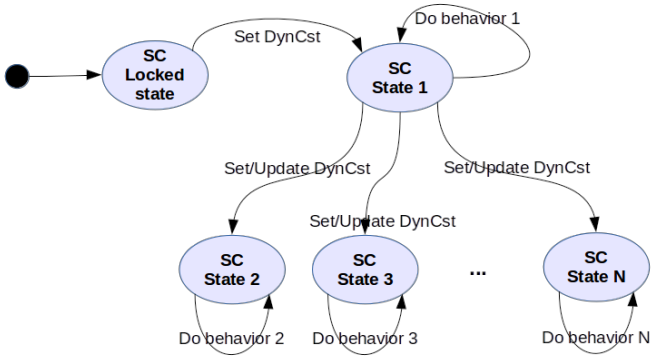

Fig. 5. State machine representation of the solution.

Detailing this smart contract will be; it has three main functionalities. The first one is to **collect** the temperatures provided from IoT sensors. This IoT device is authenticated and sending its data directly in the blockchain for collection the transaction. The second functionality of this smart contract is to **set** and change the temperature threshold of detection. This changes must be restricted to some of the stakeholders who are the responsible and authenticated user in the permissioned blockchain, e.g., Hyperledger. The third transaction is to **notify** all involved stakeholders when the temperature threshold is exceeded, and further actions need to avoid any possible consequences.

The issue is regarding that threshold. The common approach is to define the threshold as a static constant into the smart contract code (i.e. as a hardcoded and immutable variable). Certainly, we will need to update the contract each time when the constant has to change. The basic idea is then to make this constant as a variable (i.e. an updatable asset) in order to have a dynamic behavior of the contract, and by the way avoiding to change the code of the contract.

V. Solution using dynamic parameterization

A. Dynamic constants

The core solution behind this research is to define a smart contract, specific to a use case that will have a static code, but it will run dynamically. Mainly the dynamic part of these smart contracts remains in their arguments of their transactions. We propose the usage of the blockchain technological abilities to store data, which further enables the possibility to store "dynamic constant" (DynCst) into it. That is considered a variable or an asset following the Hyperledger Fabric properties. This variable leads to base the smart contract code on that internal data (i.e. constant) order to have a dynamic behavior with a static code in the case when the *DynCst* is updated. Emphasizing that providing this *DynCst* as arguments of the smart contract transactions is an external input (e.g., from the external API call, a.k.a. "Oracle" in the Ethereum community).

B. State machine representation

In Figure 5 we show how *DynCst* is working for the two functionalities **Set** and **Do** that the Smart Contract (SC) has. *Set* corresponds to the ability to set (if doesnt exist) or update (if exist) the dynamic constant that will be stored in the blockchain. And the *Do* transaction is using this *DynCst* to adapt the behavior of the code according to its value.

So, if the *DynCst* is not set, the SC cannot run (cf. Locked state). If *DynCst* is set, we can run the *Do* functionality, which will be one behavior (or lets say state 1). If we update by setting another value in *DynCst*, then the *Do* transaction will change in consequence (i.e. behavior/state 2, 3, ... N), still based on the same static code/logic. This solution assumes that the *DynCst* is given by one authorized user from the outside of the blockchain.

Moreover, for automation purpose, we can easily extend that solution by substituting the user by an automated call of the smart contract to an external database to getting back the new value for the *DynCst*.

C. Discussions: Security concerns & limitations

The exposed solution well fits the permissioned blockchains because the need for access control and authentication to distinguish users that have the right to update the *DynCst* from the users that are allowed only to use the smart contract. The main threat is regarding storage problems due to blockchain storage constraints. Indeed, a data which is updated too much frequently (e.g. each minute) will point out a massive problem of storing place because the blockchain will keep the full history of all modifications of that constant. Regarding the public blockchains, the remaining issue is: "what is restricting users to spam or flood the blockchain data with new or fake values?". That is why access control is required, and also that this solution makes no sense in public blockchains where users can trigger each transaction without restriction.

Furthermore, it could be necessary to authenticate the IoT devices as well to certify that the right and authorized devices provide the data, and ultimately to avoid spoofing or flooding by fake values (i.e., temperatures in our use-case).

Furthermore, it could be a good practice to strengthen the design of the code from the beginning by taking care of the implementation of the chaincode logic to avoid application crashes in production. Indeed, accepting misspelling values

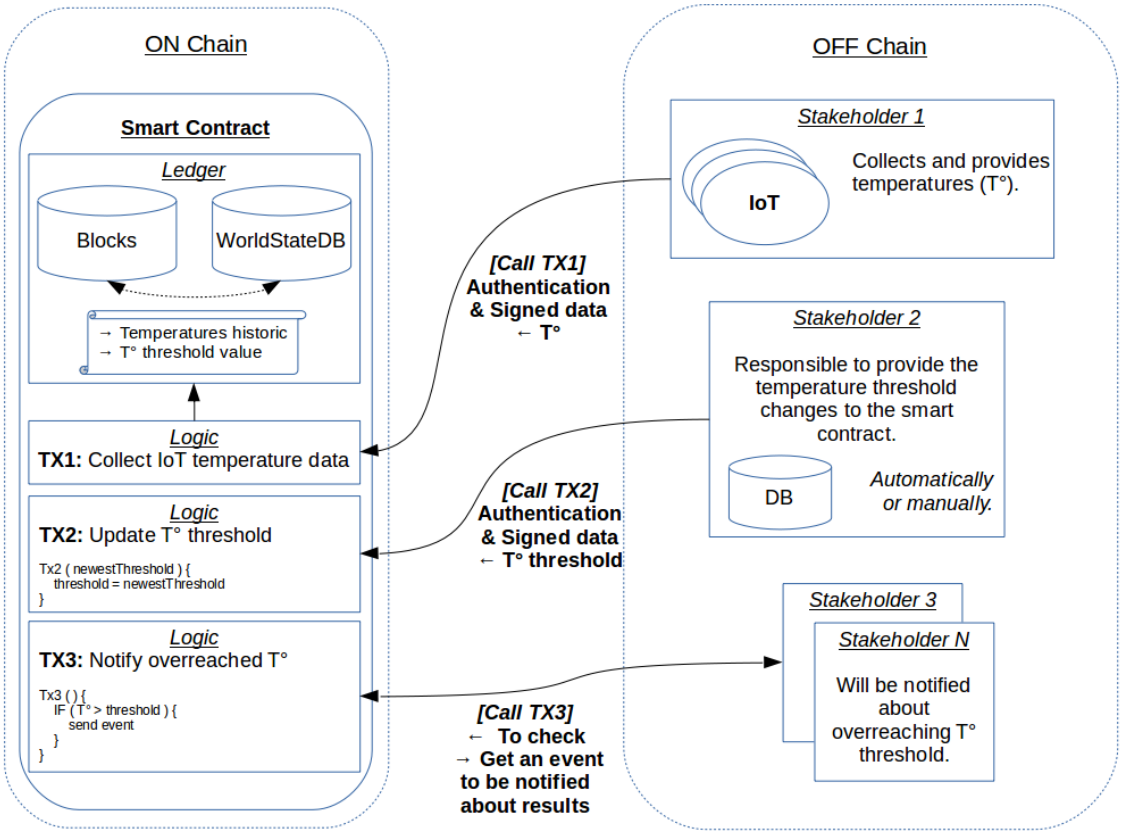

Fig. 6. Illustrating the use-case solution.

for the *DynCst* might lead to a breakdown if the code does not correctly handle it. Similarly if there is no value defined for the *DynCst*. A value must be set to make the contract runnable.

Thereby, the threats to mitigate are: access control for the *Set* transaction, avoiding many updates on *DynCst*, for performance purposes, and authenticating the source of data (i.e., stakeholders, IoTs or chaincodes).

D. Use-case implementation

The implementation of the explained use-case is completed by using the Hyperledger Composer v0.20.4 over Hyperledger Fabric v1.4.1. This solution is accessible on FramaGit [43]. The conceptual view of the solution is shown in Figure 6. As stated in section IV-B and shown by Figure 6, we implemented the three transactions *Collect* (TX1), *Set* (TX2), and *Notify* (TX3). The *DynSC* and *Temperature* are defined as assets and also *TemperatureExceeded* 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 constant. The *Notify* transaction serves to check if the temperature is exceeded the threshold (DynSC) 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.

VI. Conclusion and Future Works

In this paper, we propose a solution that facilitates the maintenance tasks for smart contract changes. A smart contract being immutable is leading to be upgraded each time to change its behavior. Our solution is enabling to define dynamic constants stored as an asset (cf. Hyperledger Fabric) instead of being hardcoded in the smart contract logic as any classical constants/variables. The use of assets allows updating the value of this asset/constant through a transaction without the need to upgrade the smart contract itself. For the proposed solution, proof of concept is developed for supporting our conceptual solution and provided access to this solution regarding our defined use-case.

For the future works, we intend to extend the current solution by using different blockchain frameworks such as Quorum [44] and Corda [45], to see the possibility of implementing our solution in different blockchains.

REFERENCES

[1] Satoshi Nakamoto. Bitcoin - A Peer-to-Peer Electronic Cash System, 2008. URL: https://bitcoin.org/bitcoin.pdf.

[2] Di_2019-global-blockchain-survey.pdf. https://www2.deloitte.com/content/dam/insights/us/articles/2019-global-blockchain-survey/DI_2019-global-blockchain-survey.pdf. (Accessed on 06/29/2019).

[3] Julija Golosova and Andrejs Romanovs. The advantages and disadvantages of the blockchain technology. In *2018 IEEE 6th Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, pages 1–6. IEEE, 2018.

[4] Vitalik Buterin. A next generation smart contract and decentralized application platform, 2016. URL: http://blockchainlab.com/pdf/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf.

[5] Nick Szabo. Smart Contract, 1994. URL: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html.

[6] Nick Szabo. Smart Contracts - Building Blocks for Digital Markets, 1996. URL: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

[7] Hyperledger - Website. URL: https://www.hyperledger.org/.

[8] Jan Mendling, Ingo Weber, Wil Van Der Aalst, Jan Vom Brocke, Cristina Cabanillas, Florian Daniel, Søren Debois, Claudio Di Ciccio, Marlon Dumas, Schahram Dustdar, et al. Blockchains for business process management-challenges and opportunities. *ACM Transactions on Management Information Systems (TMIS)*, 9(1):4, 2018.

[9] Maher Alharby and Aad van Moorsel. Blockchain-based smart contracts: A systematic mapping study. *arXiv preprint arXiv:1710.06372*, 2017.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable object-oriented software addison-wesley. *Reading, MA*, page 1995, 1995.

[11] What is a merkle tree? beginner's guide to this blockchain component. https://blockonomi.com/merkle-tree/. (Accessed on 06/25/2019).

[12] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, and Huaimin Wang. Blockchain challenges and opportunities: A survey. *Work Pap.–2016*, 2016.

[13] Paolo Tasca and Claudio J Tessone. Taxonomy of blockchain technologies. principles of identification and classification. *arXiv preprint arXiv:1708.04872*, 2017.

[14] Xiwei Xu, Ingo Weber, Mark Staples, Liming Zhu, Jan Bosch, Len Bass, Cesare Pautasso, and Paul Rimba. A taxonomy of blockchain-based systems for architecture design. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 243–252. IEEE, 2017.

[15] Andreas M Antonopoulos. *Mastering Bitcoin: Programming the open blockchain*. " O'Reilly Media, Inc.", 2017.

[16] Xiwei Xu, Cesare Pautasso, Liming Zhu, Vincent Gramoli, Alexander Ponomarev, An Binh Tran, and Shiping Chen. The blockchain as a software connector. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 182–191. IEEE, 2016.

[17] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. Exploiting the laws of order in smart contracts. *arXiv preprint arXiv:1810.11605*, 2018.

[18] Eliza Mik. Smart contracts: terminology, technical limitations and real world complexity. *Law, Innovation and Technology*, 9(2):269–300, 2017.

[19] Hyperledger Fabric - Documentation. URL: https://hyperledger-fabric.readthedocs.io/en/release-1.4/.

[20] Androulaki Elli, Cachin Christian, and al. Hyperledger Fabric: A distributed operating system for Permissioned Blockchains, 2018. (Accessed on 07/01/2019). URL: https://dl.acm.org/citation.cfm?id=3190538.

[21] Hyperledger Fabric - Wiki. (Accessed on 07/01/2019). URL: https://wiki.hyperledger.org/.

[22] Whats the difference between the 5 hyperledger blockchain projec. https://www.sdxcentral.com/articles/news/whats-the-difference-between-the-5-hyperledger-blockchain-projects/2017/09/. (Accessed on 07/01/2019).

[23] Docker - Website. URL: https://www.docker.com/.

[24] https://buildmedia.readthedocs.org/media/pdf/hyperledger-fabric/latest/hyperledger-fabric.pdf. https://buildmedia.readthedocs.org/media/pdf/hyperledger-fabric/latest/hyperledger-fabric.pdf. (Accessed on 07/01/2019).

[25] Hyperledger Fabric - Documentation - Key concepts. (Accessed on 07/01/2019). URL: https://hyperledger-fabric.readthedocs.io/en/release-1.4/key_concepts.html.

[26] Hyperledger Fabric - Documentation - Ordering service. (Accessed on 07/01/2019). URL: https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer/ordering_service.html.

[27] John Ousterhout Diego Ongaro. RAFT, In search of an Understandable Consensus Algorithm, 2014. URL: https://raft.github.io/raft.pdf.

[28] Hyperledger Fabric - Documentation - MSP. (Accessed on 07/01/2019). URL: https://hyperledger-fabric.readthedocs.io/en/release-1.4/msp.html.

[29] Hyperledger Fabric - Documentation - Membership. (Accessed on 07/01/2019). URL: https://hyperledger-fabric.readthedocs.io/en/release-1.4/membership/membership.html.

[30] Page not found - opu coin. https://www.opucoin.io/wp-content/uploads/2018/10/OPU-Whitepaper-v3.2.pdf. (Accessed on 07/01/2019).

[31] Membership hyperledger-fabricdocs master documentation. https://hyperledger-fabric.readthedocs.io/en/release-1.4/membership/membership.html. (Accessed on 07/01/2019).

[32] Hyperledger Fabric - Documentation - TLS. (Accessed on 07/01/2019). URL: https://hyperledger-fabric.readthedocs.io/en/release-1.4/enable_tls.html.

[33] Welcome to hyperledger fabric ca (certificate authority) hyperledger-fabric-cadocs master documentation. https://hyperledger-fabric-ca.readthedocs.io/en/latest/. (Accessed on 07/01/2019).

[34] Hyperledger Fabric - Documentation - Application and peers (picture). URL: https://hyperledger-fabric.readthedocs.io/en/release-1.4/_images/peers.diagram.6.png.

[35] Maximilian Wohrer and Uwe Zdun. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8. IEEE, 2018.

[36] Yue Liu, Qinghua Lu, Xiwei Xu, Liming Zhu, and Haonan Yao. Applying design patterns in smart contracts. In *International Conference on Blockchain*, pages 92–106. Springer, 2018.

[37] Christopher K Frantz and Mariusz Nowostawski. 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)*, pages 210–215. IEEE, 2016.

[38] Smart contract safety: Best practices & design patterns sitepoint. https://www.sitepoint.com/smart-contract-safety-best-practices-design-patterns/, Michiel Mulders 2018. (Accessed on 06/29/2019).

[39] Github - clearmatics/smart-contract-upgrade: Updating ethereum smart contracts without requiring a hard fork for every new update. https://github.com/clearmatics/smart-contract-upgrade. (Accessed on 06/29/2019).

[40] Maximilian Wohrer and Uwe Zdun. Design patterns for smart contracts in the ethereum ecosystem. *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)*, pages 1513–1520, 2018.

[41] Blockchain platforms 2019 — best blockchain platforms. https://www.leewayhertz.com/blockchain-platforms-for-top-blockchain-companies/. (Accessed on 07/01/2019).

[42] Adnan Imeri, Abdelaziz Khadraoui, and Djamel Khadraoui. A conceptual and technical approach for transportation of dangerous goods in compliance with regulatory framework. *JSW*, 12(9):708–721, 2017.

[43] Jonathan Lamont. PoC - hyperledger dynamic smart-contract, 2019. URL: https://framagit.org/Grapha/hyperledger-dynamic-smart-contract.

[44] Github - jpmorganchase/quorum: A permissioned implementation of ethereum supporting data privacy. https://github.com/jpmorganchase/quorum. (Accessed on 07/01/2019).

[45] An open source blockchain platform for businesses — corda. https://www.corda.net/. (Accessed on 07/01/2019).