Microservice-based Design of Smart Contracts for Blockchains in IoT Systems

Amir Taherkordi[†‡] and Peter Herrmann[‡]
[†]Department of Informatics, University of Oslo, Norway
amirhost@ifi.uio.no
[‡]Department of Information Security and Communication Technology, NTNU, Norway
herrmann@item.ntnu.no

ABSTRACT

The blockchain technology has gained tremendous attention thanks to its decentralized structure, immutability, and enhanced security and privacy guarantees. Blockchain has the potential to address security and privacy challenges of Internet of Things (IoT). By hosting and executing *smart contracts*, blockchain allows secure and flexible message communication between IoT devices and traceability in IoT applications. The unique characteristics of IoT systems, such as heterogeneity and pervasiveness, pose challenges in designing smart contracts for IoT systems. In this paper, we study those challenges and propose a microservice-based approach to the design of IoT smart contracts. The proposed service model is aimed to encapsulate functionalities such as contract-level communication between IoT devices, access to data-sources within contracts, and supporting interoperability of heterogeneous IoT smart contracts. The flexibility, scalability and modularity of the microservice architecture model make it an efficient approach for developing IoT smart contracts.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; **Distributed systems security**; • **Applied computing → Service-oriented architectures**; **Service-oriented architectures**;

KEYWORDS

Blockchains, Internet of Things, Smart Contracts, Microservices.

ACM Reference format:
Amir Taherkordi[†‡] and Peter Herrmann[‡]. 2018. Microservice-based Design of Smart Contracts for Blockchains in IoT Systems. In *Proceedings of ACM Middleware conference, Rennes, France, December 2018 (SERIAL'2018)*, 5 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Smart devices have facilitated the pervasive presence of a variety of things, interacting and cooperating with each other through unique addressing schemes—Internet of Things (IoT). Smart IoT devices often exchange a huge amount of security, safety-critical and privacy-sensitive data. This makes them appealing targets of various cyber attacks. Equipping such devices with appropriate security and privacy support mechanisms is a challenging issue due to their resource limitations, such as energy and computation. Besides resource scarceness, state-of-the-art security solutions are highly centralized and not well-suited for IoT systems due to the lack of scalability, many-to-one nature of the traffic, and single point of failure [**?**]. Considering privacy, existing privacy preserving methods rely on revealing noisy or summarized data to the data requester, while many IoT applications require users to reveal precise data in order to receive personalized services [**?**]. Therefore, IoT demands a lightweight, scalable, and distributed security and privacy solution. The blockchain technology has the potential to tackle the above challenges thanks to its decentralized, secure, and private nature [**?**].

The blockchain treats message exchanges between IoT devices similar to financial transactions in a bitcoin network. To enable message exchanges, the IoT devices leverage *smart contracts* to model the agreement between the two parties [**?**]. IoT systems can also benefit from smart contracts for other purposes such as tracing consumer-to-machine and machine-to-machine transactions. The unique characteristics of IoT systems, such as heterogeneity and pervasive communication, make the design and development of smart contracts challenging. There exists a number of smart contract programming models such as Solidity [**?**] which are suitable for conventional blockchain systems. Moreover, their design is focused on simple function-based programming, without high-level abstractions for better engineering of smart contract code which is required for pervasive and heterogeneous systems like IoT.

In this paper, we study the challenges in the design of smart contracts for blockchains integrated with IoT systems, such as autonomous operations of smart devices, heterogeneity of contract terms, and intermittent communication between devices involved in a transaction. To address these challenges, we propose a generic microservice-based design approach for developing IoT smart contracts. Flexibility, scalability, and technology diversity of microservices make them an efficient design solution for IoT smart contracts. The proposed service model is aimed to encapsulate functionalities specific to IoT smart contracts, such as contract-level communication between IoT devices, access to external data-sources within contracts, and supporting interoperability of heterogeneous IoT smart contracts. The initial implementation of these functionalities is focused on tacking issues caused by the secured and isolated sandboxed runtime environment of smart contracts.

SERIAL'2018, Rennes, France

DOI: 10.475/123_4

The rest of this paper is organized as follows. In Section 2, an overview of integration of IoT and blockchains is provided. The design aspects of smart contracts in IoT systems are discussed in Section 3, along with a microservice-based design model for developing IoT smart contracts. In Section 4, the implementation highlights and challenges are presented. The related work and the conclusions are presented in Sections 5 and 6, respectively.

2 BLOCKCHAINS AND IOT: AN OVERVIEW

Current digital economy mostly relies on a third-party trusted authority for handling financial or operational transactions. For example, it can be an email server or a bank which confirms the delivery of emails or money to a person, respectively. This means that for ensuring the security and privacy of our digital assets we need to rely on a third entity. However, these third parties control and manage all data and information and use typically a centralized costly system for transaction processing. Moreover, they can be hacked, compromised or administered by malicious agents. This is where the *blockchain technology* is introduced to solve these issues by creating a decentralized system without the need for such third parties. A blockchain is basically a distributed data structure, or public ledger of all transactions or digital events executed and shared among participating parties [**?**]. Each transaction in the public ledger is immutable and verified by consensus of a majority of the participants in the system. Blockchain makes trustless, peer-to-peer messaging possible without the need for third-party brokers, like its realization for financial services through cryptocurrencies such as bitcoin [**?**].

IoT systems are often distributed at a large scale with heterogeneous smart devices interacting each other or with other networks and platforms such as clouds. The massive data generated by IoT is characterized by the following special attributes: *i)* sensitivity of data: as it is originated from physical devices of the environment, the data can spread sensitive personal information and reveal behaviors and preferences of device owners; and *ii)* coordination and interrelation of IoT data generated by devices with different nature and context, e.g., geographically spread nodes. To preserve the security and privacy of IoT data and coordinate the flow of IoT data among different devices and systems, several security frameworks have been proposed which are highly centralized. They are therefore not necessarily well-suited for IoT systems due to the difficulty of scale, many-to-one nature of the IoT data traffic, and single point of failure [**? ?**].

To eliminate single point of failure and centralized management of sensitive data by a third party, the decentralized and trustless nature of the blockchain makes it an ideal solution to provide a secure tamper-proof IoT network. Moreover, the blockchain technology can enable the processing of transactions and coordination between millions of smart devices and creating a more resilient and unified ecosystem for devices to interact and transfer data. More importantly, by enabling secure and trustless messaging between devices in an IoT network, the blockchain treats message exchanges between IoT devices similar to financial transactions in a bitcoin network. To enable message exchanges, the IoT devices will leverage *smart contracts* to model the agreement between the two parties [**?**]. Smart contracts are scripts stored on the blockchain with a unique address, and triggered by addressing a transaction to it. Then, they execute automatically on every node of the network with input data provided by the associated transactions.

3 SMART CONTRACTS DESIGN FOR IOT

In this section, we focus on the design concerns of integrating smart contracts with IoT systems. The concerns are essentially originated from the nature of IoT data presented in the previous section, the IoT network architecture, and the unique properties of IoT applications.

The concept of smart contract was first introduced by Nick Szabo in 1993 as "a computerized transaction protocol that executes the terms of a contract". Blockchain is considered an ideal technology for supporting smart contracts. Exchanging parties can directly deal with one another without any interruption and the need to a central system. Additionally, smart contracts are stored in blocks that are electronically linked to one another in a blockchain and all the users have a copy of the stored contracts, preventing all kinds of exploits and contract tampering. IoT systems can benefit from smart contracts for different purposes such as consumer-to-machine and machine-to-machine transactions, developing traceability applications, etc. For example, in cloud-based manufacturing platforms, smart contracts act as agreements between the service consumers and the manufacturing resources to provide on-demand manufacturing services [**?**]. As another example, in supply chain systems, smart contracts can maintain a registry of products and track their position through different points in a supply chain through cryptographically verifiable receipts for product delivery [**?**]. In the following, we discuss the main concerns in designing IoT smart contracts.

Autonomous execution. This feature enables the autonomous operation of smart devices without the need for a centralized authority. The autonomy of smart contracts in typical blockchains is limited to automatic execution of contract terms, while triggering the execution of a smart contract function is basically performed through a user transaction in the blockchain. In the case of IoT, a higher level of autonomy is required as in many applications the functions in a smart contract are triggered and executed based on the contextual situation of devices in the environment, e.g., a 3D printing transaction from one machine to another machine for manufacturing a product.

Heterogeneous contracts. A key player in smart contracts for IoT systems are smart devices themselves. They may directly interact each other to fulfill a requirement or take part in executing a workflow, e.g., in logistics management systems. In a wider scale, in a manufacturing scenario, plenty of smart devices may need to communicate, where each device possesses its own settings in terms of semantics for describing blockchain transactions and programming smart contracts and their dependencies (e.g., database access or network communication). This indicates that we can not rely on a pre-defined smart contract realization model that serves as a general model for designing and developing smart contracts for IoT systems. Therefore, a high-level heterogeneity support is required to enable developing and deploying IoT smart contracts with different semantics for describing blockchain transactions.

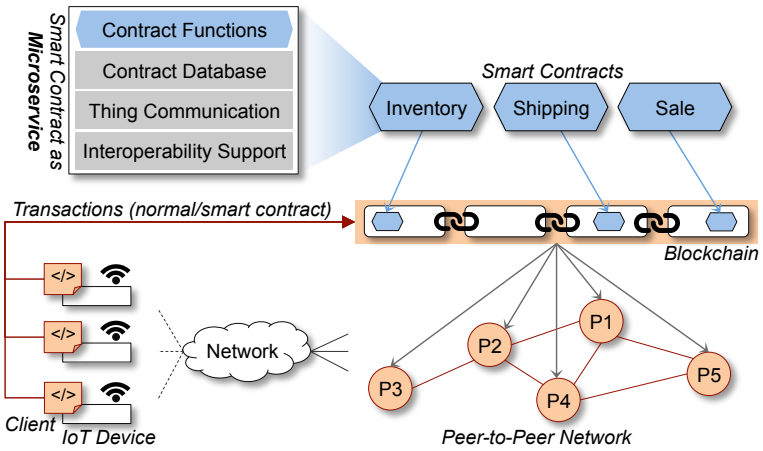

Figure 1: Overall integration model for IoT and blockchain with smart contracts as microservices

Intermittent information flow. IoT devices communicate intermittently. Thus, when a device is granted and verified to perform a transaction, we should not expect that the device is necessarily online. This may lead to a temporary halt in the information flow that involves smart contracts. For example, in the case of supply chain, the lack of network connectivity during the delivery of a shipment may result in the lack of delivery confirmation, and consequently no payment to the supplier will be made. Smart contracts for IoT systems should be designed in such a way to minimize the effect of intermittent IoT communication on their execution and the information flow.

In the next section, we propose a software design model for smart contracts which is aimed to address the above concerns for the design and development of smart contracts in IoT systems.

3.1 IoT Smart Contracts as Microservices

The microservice architecture is one of the recent methods of developing software systems that is focused on building single-function modules with well-defined interfaces and operations. In a monolithic application, built as a single unit of software, a change made to a small section of code might require building and deploying entirely new version of software, leading to low flexibility and scalability in engineering applications. The microservice architecture is defined as a design approach to develop a monolithic application as a suite of small services, each running in its own process and communicating with lightweight mechanisms. These services are small, independently deployable, highly decoupled and aimed to do small tasks [**?**].

Microservices may be developed in different languages and use different data storage techniques, while they promise scalable and flexible development of systems. Microservices has been recently introduced for IoT systems because of the continuous evolvement of IoT applications and growth in the scale of monolithic applications with more complexity in their structure [**?**]. The advantages of microservices make them an efficient design choice for building IoT smart contracts, considering their unique characteristics and challenges discussed earlier in this section.

Figure 1 shows the main elements of the model for integrating IoT and the blockchain, which encompasses our proposal for designing smart contracts as microservices. On the right side, there is a peer-to-peer network for hosting and maintaining the blockchain, where each node has a copy of the blockchain. Each block in the blockchain can contain normal transactions and/or the bytecode of smart contracts. Having the address of the smart contract code available, other transactions can execute a smart contract function and create new transactions. On the other hand, IoT devices should be equipped with client code to interact with the blockchain. The interaction can be either a *normal transaction* or a *smart contract transaction*. The former refers to creating typical transactions which should be stored in a block, e.g., transferring digital assets like bitcoin from an account to another account when a delivery is performed in the supply chain application. The latter can be either creating and posting a new smart contract to the blockchain, or invoking a function of a given smart contract deployed on the blockchain.

As mentioned before, smart contracts are computer code stored in blocks, containing a set of functions implementing the terms of a contract—Contract Functions. The top right part of Figure 1 includes three sample smart contracts for the logistic and supply chain applications. In their traditional design, each one includes only Contract Functions, e.g., functions implemented using Solidity language in the Ethereum blockchain platform [**?**]. The top left part shows our proposal for implementing smart contracts as microservices. A microservice, in this model, contains not only the Contract Functions, but also functionalities that are essentially specific to a smart contract, addressing the aforementioned IoT smart contract design concerns. All of them are encapsulated in a microservice along with the Contract Functions. In the rest of this section, we discuss these functionalities.

Thing Communication is a key functional requirement, arising from the fact that smart contracts cannot access and fetch directly the data they require, e.g., traffic-related information for estimating cost in the workflow of goods delivery. For that, the smart contract requires to communicate with a third-party system or another

IoT device to complete the execution. For example, it can establish a RESTful communication to an IP-enabled device to fetch the required information. This functionality also handles failures in communication with other devices to overcome the issue of intermittent information flow. Additionally, Thing Communication encompasses logic for autonomous execution of contract functions based on different contextual situations of the IoT application, e.g., in the asset tracking use case, the `sendMoney` function will be executed if a container and a retailer share the same location [**?**]. To this end, every stakeholder carries a BLE, GSM or LTE radio and then the IoT application triggers the blockchain Client on devices that are co-located.

Contract Database allows a contract to communicate with a trusted data provider. For example, the data source can be a secure application running on an hardware-enforced Trusted Execution Environment (TEE) such as Industrial IoT TEE for Edge Devices (IIoTEED) [**?**]. Contract Database serves as a data access point for an individual smart contract. A relevant example of data source is IPFS (Interplanetary File System). IPFS files are content-addressed, identified by their hashes. In order to fetch a data file, the entire network is searched for a file corresponding to a particular hash. Thus, it is an ideal file storage and sharing technology for developing decentralized IoT access control models [**?**].

It should be noted that, according to the general specification of smart contracts, their communication with the off-chain world is either limited (i.e., to other smart contracts) or not allowed. For example, Ethereum Virtual Machine (EVM) [**?**] is only sandboxed but completely isolated, meaning that the code running inside the EVM has no access to the network, file system or other processes. To communicate with other parties like a data source, we need to make sure that the data fetched from the original data-source is genuine and untampered. One solution, developed by Oraclize [**?**], is to accompany the returned data together with a document called authenticity proof, which can be built using technologies such as auditable virtual machines and TEE. In the next section, we discuss this issue in detail.

Interoperability Support is proposed to support heterogeneity among devices interacting with a smart contract. The most common type of heterogeneity appears in the semantics for describing the transactions added to the blockchain by executing different smart contracts. For example, high diversity of IoT devices in logistic applications can lead to workflow transactions that are semantically heterogeneous. Interoperability Support encompasses mechanisms for interpreting transactions produced by the corresponding smart contract to a general form interpretable and traceable by the blockchain.

4 IMPLEMENTATION HIGHLIGHTS

There are a number of well-known blockchain platforms featuring smart contract functionality, such as Ethereum [**?**], Hyperledger Fabric [**?**], and NEO [**?**]. In this paper, we adopt Ethereum as the smart contract development framework. The main advantage of Ethereum's smart contract platform is the high degree of standardization and support it offers. Additionally, Ethereum comes with a set of well-defined rules for developers for how to develop smart contracts and make the development less risky. It is because, in Ethereum, extensive effort has been made to improve the development and operation of smart contracts. Moreover, Ethereum features its own smart contract programming language, Solidity, which facilitate development and setting up smart contracts [**?**]. Solidity is a high-level language for implementing smart contracts. It is influenced by C++, Python and JavaScript and is designed to target EVM. Solidity is statically typed, supports inheritance, libraries and complex user-defined types.

The main implementation goal is to provide a programming framework which enable the developer to encapsulate the contract functions and functionalities in a microservice. This requires to understand the runtime environment of smart contracts in Ethereum. In particular, we need to investigate the features and limitations of the *secured* runtime environment of smart contracts. As mentioned before, Ethereum smart contracts run on the Ethereum Virtual Machine (EVM). The EVM is a sandboxed, completely isolated runtime for smart contracts. This means that every smart contract hosted by the EVM has no access to the network, file system, data sources, or other processes running on the computer hosting the EVM. This makes the implementation of the proposed microservice model challenging, in particular with respect to the functionalities that need interaction with one of more of the aforementioned sources. To tackle this issue, we first need to look into the architectural design of the EVM.

Figure 2 depicts the main architectural element of the EVM. The EVM code (i.e., the byte code of the smart contract) is hosted in an immutable virtual ROM within the EVM. The EVM manages different kinds of data depending on their context. There are three main types of data: memory, stack and storage. Memory and stack are volatile spaces used to store data during execution and small local variables, respectively, e.g., passing arguments to internal functions. Storage is a persistent read-write word-addressable space in which the contract stores its persistent information. The amount of gas required to save data into the storage is considerably high, as compared to other operations of the EVM.

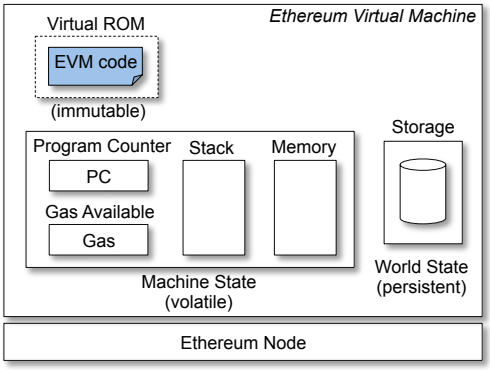

Figure 2: The main architectural elements of the EVM

The inside architecture of EVM shows that smart contracts have the capability to communicate with the storage for storing and retrieving data, even though this space is limited to some extent because it is structured as a key-value mapping of 2^{256} slots of 32 bytes each. Moreover, there is no possibility to communicate with

entities out of the EVM. Considering the proposed microservice design, the storage of the EVM can serve as Contract Database. In this way, for smart contract C we can create a new *database contract* C_d which is used only for storing and retrieving data of C. By separating the contract database from the contract itself, the deployment and management of new versions of the contract in the blockchain will become less costly. In order to get access to C_d from C, cross-contract message calls can be leveraged. Contracts can call other contracts through *message calls*. To call a function of another contract in Solidity, *external* functionals should be defined. As an example, the following code snippet shows the structure of database contract C_d:

```
pragma solidity ^0.4.0;
contract MyContractDB {
    uint public dataA = valueA;
    uint public dataB = valueB;
    ....
    function setDataA(uint value) external returns (uint) {
        dataA = value;
        return dataA;
    }
   function  getDataA(uint value) external returns (uint) {...}
}
```

Using contract Application Binary Interface (ABI), other contracts in Ethereum can interact with C_d.

As mentioned above, the size of storage is somewhat limited and, beyond this, the gas cost of interacting with the storage is quite high. A complementary solution to the above data access model is using Oraclize [**?**]. it supports various types of data sources, such as URL (to get access to any webpage or HTTP API endpoint) and IPFS (to get access to any content stored on an IPFS file). To read data from these data sources, queries should be created. A query is an array of parameters which needs to be evaluated in order to complete a specific data source type request. For example, in the case of sample smart contracts in Figure 1, the Shipping contract requires to communicate with a device-specific locally-deployed service that calculate the shipment cost of a specific IoT device category. Using Oraclize, the query is described as:

```
oraclize_query("URL", "http://127.0.0.1/ShipmentCost
                                    ?device=VerticalPump")
```

The result of executing the query will be broadcast as a transaction carrying the result. In the default configuration, the transaction will execute the `_callback` function which is implemented by the developer in the smart contract .

To conclude, the Contract Database can be either realized as a new contract co-located with the main contract in the EVM or deployed as a separate data-source service external to the contract deployed in the EVM. In the latter case, the service will be accessible through a REST API or IPFS, using Oraclize libraries. Likewise, for functionalities related to Thing Communication, Oraclize can be leveraged to communicate with IoT services that are external to Ethereum nodes. For implementing Interoperability Support, similar to C_d, we propose developing new smart contracts with a set of well-defined external functions that merely perform semantics analysis and mapping. In this way, such functionality will serve as a reference for semantic interoperability between heterogeneous IoT smart contracts. Shared by all above three functionalities, the implementation approach, proposed in this section, meet the essential programming requirements for developing IoT smart contracts as microservices.

5 RELATED WORK

Although smart contracts have recently received considerable attention by the research community and industry, Most existing work on IoT smart contracts has so far focused on the issues in integrating blockchains and IoT, such as designing lightweight blockchains for IoT.

In [**?**], a smart contract-based framework is proposed to implement distributed and trustworthy access control for IoT systems. The authors use the Ethereum smart contract platform to provide an access control method for static and dynamic access rights validation. In [**?**], a blockchain-based solution is proposed to address scalability in managing access in large-scale IoT systems. From a different view to scalability, A. Dorri et al. in [**?**] propose a lightweight scalable blockchain model to overcome the concerns of limited scalability, significant bandwidth overheads and delays for blockchains integrated with IoT. EdgeChain [**?**] uses a credit-based resource management system to control the IoT devices' resources obtainable from the edge server. The authors propose using smart contracts to regulate IoT devices' behavior and enforce policies. The above approaches are mainly focused on addressing the scalability issue in IoT smart contracts.

K. Christidis et al. in [**?**] discuss how smart contracts allow automated complex multi-step IoT processes. The authors indicate that smart contracts enable cryptographic verifiability of IoT workflow and significant cost and time savings in IoT workflow execution. As an example, in [**?**], a decentralized, peer-to-peer blockchain platform for industrial IoT is proposed to enable cloud-based manufacturing and on-demand access to manufacturing resources. Both above works are mainly about the usefulness of smart contracts in executing IoT workflows. In [**?**], a microservice architecture is proposed to develop scalable and secure smart surveillance systems. For data protection and synchronization, the proposed framework uses the blockchain and smart contracts. However, in this work, microservices are proposed for the components of the surveillance systems, not for design of smart contracts.

6 CONCLUSIONS AND FUTURE WORK

The blockchain technology and smart contracts have great potential in automating, securing and scaling message communication in IoT systems. In this paper, we studied the design concerns in using smart contracts for IoT systems and proposed a microservice-based approach for developing IoT smart contracts. The adoption of microservice design model tackles challenges such as heterogeneity and pervasiveness in designing IoT smart contracts. However, implementing IoT smart contracts as microservices with the proposed functionalities comes with some programming challenges that we explored in this paper. The future work includes further investigation on contract interoperability and requirements for *container* platforms that can host smart contracts.