Implementing a Microservices System with Blockchain Smart Contracts

Conference Paper · February 2019

DOI: 10.1109/IWBOSE.2019.8666520

5 authors, including:

Roberto Tonelli
Università degli studi di Cagliari
127 PUBLICATIONS **1,170** CITATIONS
SEE PROFILE

Maria Ilaria Lunesu
Università degli studi di Cagliari
43 PUBLICATIONS **200** CITATIONS
SEE PROFILE

Andrea Pinna
Università degli studi di Cagliari
20 PUBLICATIONS **227** CITATIONS
SEE PROFILE

Davide Taibi
Tampere University
113 PUBLICATIONS **829** CITATIONS
SEE PROFILE

Implementing a Microservices System with Blockchain Smart Contracts

Roberto Tonelli
Department of Mathematics and Computer Science
University of Cagliari
Cagliari, Italy
roberto.tonelli@dsf.unica.it

Maria Ilaria Lunesu, Andrea Pinna
Department of Electrical and Electronic Engineering (DIEE)
University of Cagliari
Cagliari, Italy
ilaria.lunesu@diee.unica.it, a.pinna@diee.unica.it

Davide Taibi
Tampere Software Engineering Group (TaSE)
Tampere University
Tampere, Finland
davide.taibi@tuni.fi

Michele Marchesi
Department of Mathematics and Computer Science
University of Cagliari
Cagliari, Italy
marchesi@unica.it

***Abstract*—Blockchain technologies and smart contracts are becoming mainstream research fields in computer science and researchers are continuously investigating new frontiers for new applications. Likewise, microservices are getting more and more popular in the latest years thanks to their properties, that allow teams to slice existing information systems into small and independent services that can be developed independently by different teams.**

A symmetric paradigm applies to Smart Contracts as well, which represent well defined, usually isolated, executable programs, typically implementing simple and autonomous tasks with a well defined purpose, which can be assumed as services provided by the Contract. In this work we analyze a concrete case study where the microservices architecture environment is replicated and implemented through an equivalent set of Smart Contracts, showing for the first time the feasibility of implementing a microservices-based system with smart contracts and how the two innovative paradigms match together.

Results show that it is possible to implement a simple microservices-based system with smart contracts maintaining the same set of functionalities and results. The result could be highly beneficial in contexts such as smart voting, where not only the data integrity is fundamental but also the source code executed must be trustable.

***Index Terms*—Microservice, Cloud Native, Blockchain, Smart contract, Serverless**

I. Introduction

Microservice architecture becomes more an more popular in the latest years. Microservices are small and autonomous services deployed independently, with a single and clearly defined purpose [23]. Because of their independent deployment, they have a lot of advantages. They can be developed in different programming languages, they can scale independently from other services, and they can be deployed on the hardware that best suits their needs. Moreover, because of their size, they are easier to maintain and more fault-tolerant since the failure of one service will not break the whole system, which could happen in a monolithic system. Another characteristic of microservices is, being cloud native applications, the support of the IDEAL properties: Isolation of state, Distribution, Elasticity, Automated management and Loose Coupling [23]. Moreover, microservices propose to vertically decompose the applications into a subset of business-driven services. Every service can be developed, deployed and tested independently by different development teams, and by means of different technology stacks. The responsibility of the development of a microservice belong only to one team, who is in charge of the whole development process, including deploying, operating and upgrading the service when needed, reducing the need of communication with other teams [27] and reducing the need of synchronization when identifying the requirements [28].

On the other hand Smart Contracts are computer programs written in a Turing complete programming language, living and running on a blockchain backbone, written with the purpose of enforcing agreements between two parties in a decentralized, untrusted environment with no control of a central authority.

Smart Contracts are self contained programs which are deployed into the blockchain environment where the Ethereum Virtual Machine (EVM) executes them in a decentralized scheme. They can execute any kind of computation and have a storage available where data and state can be saved. As a general view, Smart Contracts can be thought as self contained pieces of code providing services to the external world, and the blockcahin environment can be considered as the gateway by which Smart Contracts provide such services. Even if the code is permanently stored as it is, another Smart Contract enhancing or correcting a previous one can be deployed in the blockchain in order to implement versioning to correct bugs, to refactor or to add new features to the software code. Such possibility renders the plug and play scheme feasible for Smart Contracts living in the blockchain providing an architecture which is strongly overlapping to the microservices one. Furthermore Smart Contracts can easily be implemented so that their state and internal variables in the blockchain

Pre-proceedings Author Version. Please, cite as:
Roberto Tonelli; Maria Ilaria Lunesu; Andrea Pinna; Davide Taibi; Michele Marchesi. "Implementing a Microservices System with Blockchain Smart Contracts"
2nd International Workshop on Emerging Trends in Software Engineering for Blockchain - Colocated with SANER19, 2019

storage are not necessarily used and the persistence can be altered by replacing a Smart contract with a new one. This scheme can be adopted to make Smart Contracts perform as microservices, which are stateless, use only internal data and are not persistent. Finally where microservices architecture is implemented by a API Gateway which allows communications among services and rest calls manage interface communications, Smart Contracts architecture can take advantage of the Web3.js infrastructure for implementing and managing communications between services provided by contracts [13].

The goal of this paper is to prove that it is possible to fully implement a system based on microservices with Smart Contracts. Our objective is to demonstrate that, given the similarity of the two approaches, the architecture of a microservice system can be reproduced by a proper set of Smart Contracts which can provide the same services through the blockchain, with the main advantage that every change and operation are permanently and transparently recorded into the blockchain ledger.

In this paper we deeply dig into such analogy and implement an existing microservices-based system using a set of Smart Contracts written in Solidity and tested on the Remix facility provided by Ethereum. Each Smart Contract provides its own service so that the microservices architecture is completely replicated into the Ethereum blockchain. We adopt Solidity Smart Contracts, where setter public functions and getter public view functions replace the functionality provided by the microservices and rest calls are implemented by means of the calling of specific public functions of smart contracts deployed in the blockchain.

Our work discusses advantages and disadvantages of the two approaches and demonstrates how the new emerging blockchain technology can be quickly adapted to well perform in substitution of another existing software technology and architecture.

The result of this case study, where we provide the complete Solidity working code (tested on Remix) and design the smart contracts architecture for mapping into the blockchain a concrete example of service provided through a microservices architecture, proves that it is possible to implement a basic microservices-based system with smart contracts. Relative advantages and disadvantages of the two approaches are yet to be fully explored and understood.

The remainder of this papers is organized as follows. In Section II we describe the related works and the ideas behind smart contracts implementation of a microservice architecture, in Section III we briefly compare microservices with smart contracts. In Section IV we describe the microservice-based system we are aiming at re-implement with smart contracts. In Section V we describe the implemented set of smart contracts and how do they work also comparing the source codes of the microservice and the smart contract solutions. In Section VI we discuss the outcomes and finally in Section VII we draw conclusions, providing recommendations on how to implement a smart-contract-based system and how to generalize the results of this study in other contexts.

II. Background and related work

The Ethereum blockchain and the use of smart contracts are quite recent topics in the computer science literature. The Ethereum white and yellow papers [13] [31] issued in 2013/14 describe purpose and architecture of Ethereum blockchain, of the Ethereum Virtual Machine (EVM), of the Abstract Binary Interface (ABI) used to encode programs written in a high level programming language into the bytecode managed by the EVM. Since then, the architecture as well as the Solidity programming language [32] evolved, many researchers started to look for applications and to analyze smart contracts structure and potential.

A. Smart Contracts and Microservices

In this paper we refer to the best known and most used smart contract paradigm, that of the Ethereum blockchain, where the programming language is Solidity, a language derived by C, C++ and Javascript, but our approach can be generalized to different blockchain and programming languages, such as Hyperledger and GO. As already stated, smart contracts are self contained programs which are deployed into the blockchain environment where the Ethereum Virtual Machine (EVM) executes them in a decentralized scheme. They can execute any kind of computation and have a storage available where data and state can be saved. smart contracts interaction can be managed through the Web3.js library, which allows external users to set contracts variables, to call contract functions and to get data from a smart contract. smart contracts can also interact with each other since a smart contract can send messages to other smart contracts in the blockchain. We recall that smart contracts can be thought as self contained pieces of code providing services to the external world, where the "world" can be another smart contract or an external user, and the Web3.js library or the blockcahin environment can be considered as the gateway by which smart contracts provide such services. Even if the code, once deployeed in the blockchain, is permanently stored as it is, due to the blockchain features of immutability of previous blocks, another smart contract enhancing or correcting a previous one can be deployed in the blockchain in order to implement versioning to correct bugs, to refactor or to add new features to the software code. Currently there are projects which adopt this strategy in the blockchain, like 0x.project [30]. As claimed in the introduction, such possibility renders the plug and play scheme feasible for smart contracts living in the blockchain providing an architecture which is strongly overlapping to the microservices one and where the IDEAL properties can be supported as well. Furthermore, even if smart contracts have a state, hold variables and are persistent in the blockchain, they can easily be implemented, for example by the use of "view" and "pure" functions alone, so that their state and internal variables in the blockchain storage are not necessarily used and the persistence can be altered by replacing a smart contract with a new one. This scheme can be adopted to make smart contracts perform as microservices, which are stateless, use only internal data and are not persistent. Finally, pushing the analogy further,

where microservices architecture is implemented by a API Gateway which allows communications among services and rest calls manage interface communications, smart contracts architecture can take advantage of the Web3.js infrastructure for implementing and managing communications between services provided by contracts. It is also possible, but is out of the scope if this paper, to organize interactions among Smart Contracts written in different programming languages, like for example in the Hyperledger blockchain, completing the analogy with microservices where the different services can be implemented using different programming languages.

B. Related works

To the purpose of this paper we focus our attention on the following research works.

One of the first applications of this technology is represented by the electronic voting as argued in [5] by McCorry et al. that discussed about the first implementation of a decentralized and self-tallying internet voting protocol with maximum voter privacy using the blockchain. They choose to implement the boardroom voting protocol as a smart contract on Ethereum. These smart contracts have an expressive programming language and the code is directly stored on the blockchain.

An other valid context, where smart contract have been used, is logistic field as argued by Casado et al. in [8], authors proposed a system that uses smart contracts to remove intermediaries and speed up logistics activities. The new implemented model combines smart contracts and a multi-agent system to improve the current logistics system by increasing organization, security and significantly improving distribution times.

Smart contracts inherit the availability and other security assurances of blockchains, however, they are impeded by blockchains lack of confidentiality and poor performance. In [9] authors present Ekiden, a system that addresses these critical gaps by combining blockchains with Trusted Execution Environments (TEEs). Ekiden leverages a novel architecture that separates consensus from execution, enabling efficient TEE-backed confidentiality-preserving smart contracts and high scalability.

In [10] Zhang et al. propose an IoT E-business model with the aim to redesign many elements in traditional E-business models and realize the transaction of smart property and paid data, on the IoT, with the help of blockchain and smart contract technologies. In addition a description of holonic energy systems and the implicit research required towards sustainability and resilience in the imminent energy landscape was provided.

In [11] Casado et al. propose a new model of supply chain via blockchain, as the used used till now model has some disadvantages, such as the relationships between the members of the supply chain or the lack of information for the consumer about the origin of the products. Using blockchain to build IoT system might facilitate the control and configuration of IoT devices.

In [12] Huh et al. argue that Ethereum has been chosen as blockchain platform because using its smart contract, it is possible to write a Turing-complete code. It also allows to easily manage configuration of IoT devices and build key management system and simply use account as a key management system, which most of blockchain platform supports. the decision to use Ethereum because it allow to can manage the system in a more fine-grained way.

Cunningham et al. in [14] present an implemented system based on Ethereum platform, a smart contract based, distributed ledger system for facilitate a secure patiens'access to their own records with private data. Through Ethereum were provided the core blockchain capabilities allowing for hosting within a trustless, secure environment, and a smart contract implementation and for the programmatic implementation of the API directly on the platform itself.

Ekblaw et al. in [15] discuss a system that helps patients for a comprehensive, immutable log and easy access to their medical information across providers and treatment sites. Leveraging unique blockchain properties, MedRec manages authentication, confidentiality, accountability and data sharing-crucial considerations when handling sensitive information. Ethereum's smart contracts were used to create intelligent representations of existing medical records that are stored within individual nodes on the network. The contracts were constructed in order to contain metadata about the record ownership, permissions and data integrity. The blockchain transactions in the presented system carry cryptographically signed instructions to manage these properties.

Two recent events raised the attentions of the computer science community on Smartcontracts. The DAO project was initiated in 2016 followed by a great clamor when a improper use and coding of a multisig smart contract allowed a single owner to withdraw Ethers from the wallet and causing the famous Ethereum fork. A year later (Nov. 2017) the Parity wallet, a wallet for exchanging and managing cryptoccurrencies as well as ownership of smart contracts founds was accidentally corrupted by an "unexperienced" user, deleting the wallet library by a misuse of the "suicide" function and causing the frozen of all the Ethers managed by the Parity wallet library (around 560 million dollars at the time). The exceptional case was investigated by Destefanis et. al [20] who illustrated the pitfalls in the specific case and in a general approach to smart contracts programming. They, also, hightlighted the need for a discipline such as Clockchain Software Engineering, addressing the issues posed by smart contract programming and other applications running on blockchains [3]. A case of study about a bug discovered in a smart contract library that caused the freezing of about 500K Ethers (about 150M USD, in November 2017) was deeply analysed together with the source code of Parity and the library in order to recognised best practices for mitigating, in case of need software misconduct. In 2017 Porru et al. [4] recognized the need for a structured approach to the design of smart contract applications running onto a blockchain and coined the term BOSE ("Blockchain Oriented Software Engineering"). In [35] Pinna et al. de-

veloped a tool based on petri nets for analyse cryptovalues transactions within the Blockchain with the aim to evaluate the users behaviour. sono stati sviluppati tool di analisi per le transazioni di criptovalute all'interno della blockchain allo scopo di valutare il comportamento degli utenti.

The lack of inspectability of a deployed contract by analyzing contract state using decompilation techniques driven by the contract structure definition represents a common issue and guides the study conducted by Bragagnolo et al. [21] that presents SmartInspect as an innovative solution that uses a mirror-based architecture to locally represent object responsible for the interpretation of the contract state. SmartInspect facilitates the work of developers in fact it proposes itself as a supports to better visualize and understand the contract stored state without needing to redeploy nor a code customization.

Creating a writing well performing and secure contracts in Ethereum is an issue todays most prominent and find a valid solution is a more difficult task. In [19] using Grounded Theory techniques to extract and identify the patterns For this purpose, Wohrer et al. present a set of common security patterns as a support to specific security issues that can be applied to mitigate typical attack scenarios.

In [22] Fenu et al. analyzed the smart contracts standard used to manage the tokens of 1388 ICOs they examined. These ICOs were published on 2017 on icobench.com website and the analysis was focused on the evaluation of their quality and software development management and also on discover the features that can influence the ICO success. In a similar way Hartmann et al. [34] analyzed a set of ICO evaluation websites to reveal the state of the practice in terms of ICO evaluation. Key information about ICOs collected by these websites are categorised, and key factors that differentiate the evaluation mechanisms employed by these evaluation websites are identified. The findings of this study could help a better understanding of what entails to properly evaluate ICOs. Likewise Ibba et al. [33] performed a deep analysis of thousands ICOs with the aim to understand software engineering activities related to ICOs, to recognize the ICOs developed using Agile methods and to make a comparison between ICOs and Agile ICOs. In addition, the analysis of Agile ICOs concerned particularly project planning, software development, and code features.

In 2018 the similarities between the microservices paradigm and architecture and the symmetric smart contract structure was first noted by [29], and the authors proposed an architecture scheme where the role and the services of each Micorservice were implemented by corresponding smart contracts written in Solidity and running onto the Ethereum blockchain.

Grossmann et al. in [16] presented a generic correctness condition for callbacks, Effective Callback Freedom. They, also, showed that Ethereum might be used to prevent bugs without drastically limiting programming style and can be checked dynamically with low runtime overhead with an particular focus towards to microservices.

In 2017, Kapitonov et al. in [17] show, how to organize a communication system between agents in a peer-to-peer network using the decentralized Ethereum blockchain technology and smart contracts. Was implemented a protocol that allows the connection of a different agents variety to a general network in which each agent can request and offer different services transfer of data from agent sensors, moving to a desired point, cargo transportation and any other work that autonomous agents are able to perform. The proposed system represents a Blockchain-microservices combination considering the strengths of both technologies.

In this work [18] authors describe how content sessions can be dynamically mapped to network service chains through network "softwarization" and the use of microservices. In particular they conducted a study on how blockchain-powered smart contracts and network service chaining can be exploited to support such novel collaboration schemes. The findings of their study showed how existing technologies can be complemented by supporting a wide range of business cases while at the same time significantly reducing costs.

Symmetrically, the paradigm of microservices architecture appeared for the first time in 2012 with the purpose of decompose a monolithic software architecture into separated and independent services with the aim of developing independently deployable services to ease the distributed development and improve the overall system' maintenance easiness. Microservices become mainstream in the last years. Big companies such as Amazon, Netflix, Spotify and many others are using them, and all other companies are adapting and following the trend, to benefit of the advantages of microservices [2], in some cases migrating and re-architecting existing applications to microservices, in other cases starting the development of new application from their Minimum Viable Product [7] with microservices.

III. The Proposed architecture

Before describing the details of the case study we implemented we compare the microservice and the smart-contract architecture.

In order to map a general microservice architecture into an architecture where services are provided by smart contracts we consider as paradigmatic example that of an information system where doctors can keep track of their patients and of the diagnosis they made.

Microservices-based systems are usually built with three layers. The graphical user interface, the API-Gateway and the microservices (see Figure 1). Any device can be connected to the system through the API-Gateway. The role of the API-Gateway is to provide a custom interface for each type of device, usually exposing a set of REST APIs. As example, the API-Gateway can implement a set of interfaces for a web application, other interfaces for a mobile application and other interfaces to expose data publicly. The API-Gateway then calls the different microservices and returns the result to the caller. The direct communication between the graphical user interface and the single microservice is considered an anti-pattern, even if technically possible [25] [26]. The API-Gatweay can also

be responsible of the load balancing. Microservices usually adopt lightweight message queues to communicate, adopting the publisher-subscriber pattern. A microservice commonly publishes the message into a channel of the message bus. Other microservices can subscribe to the same channel and read the messages received.

Similarly to the common microservice architecture, the corresponding smart contracts architecture is built on two layers as depicted in figure 2. The first layer is the interface between external applications and the blockchain. It provides the Application Binary Interface (ABI) to interact with the smart contracts and the User Interface in which ABI are embedded. Specifically, an external App can use ABI exposed by each smart contract to compose and perform service requests. The ABIs allow to specify the functions to call and manage the correct format of data exchanged with the smart contracts.

The second layer is composed by the set of smart contracts deployed into the blockchain. In this architecture our solutions is the simplest: each Micoservice is implemented by a single "atomic" smart contract, which allows to simply convert the software code usually embedded in a Java class providing the microservice into the corresponding Solidity code for each smart contract. Other solutions can be devised, but we consider this the easiest for mapping one architecture to the other for our purposes. Layer communications is governed by remote procedure calls (RPC) by means of the Web3.js library provided by Ethereum 4, which is designed for implementing Javascript code for executing blockchain transactions, calls to contracts, queries and for generally manage the interaction among external Apps and blockchain. In this layer the blockchain features, specific for each blockchain and each Contract programming language, can then be used for implementing specific services. In our model each user is identified by its Ethereum address. An account service is provided (for example through a mapping construct) by a smart contract which registers and manages the accounts. For example, one can deine services for an online shop (which is a typical example for microservice architectures) and Customers profiles can be defined so that different permits or privileges can correspond to different services provided by the layer. Data are automatically and permanently stored on the blockchain, so that local resources are not needed for this purpose since each blockchain node (which can implemented in a private blockchain) holds its own copy. Login and registration procedures are implemented each by an atomic microservice in separated smart contracts. For the online shop example an inventory service can record information on products on the blockchain and can expose the information on the Storefront web page or directly to the Apps. An ad-hoc smart contract can manage the delivery service recording all the needed data.

IV. The Microservices-Based System

In this section we describe the microservices-based system that we re-implemented with smart contracts. To validate the feasibility of our approach, and for reason of simplicity, the system of this case study is composed by only three microservices. The same implementation approach could be extended to more complex systems by adding more microservices.

In this case study, we adopted a simple microservice-based application composed by three microservices. The system is an extension of the tutorial presented by Edureka [24].

The system is composed by three microservices written in Java: Doctor, Patient and Diagnosis. The goal of the system is to allow doctors to keep track of diagnosis for the diseases of their patients. The Edureka tutorial shows how to implement the system by microservices and how to call them to produce an output by composing the formatted outputs (in JSON format) obtained by calling other microservices.

The microservices are accessed from the web user interfaces through an API-Gateway that routes the requests and forward the messages. The system is depicted in Figure 1 while the original source code is available online [24].

Beside its simplicity, the system implemented includes several characteristics of real and bigger systems. It exposes APIs to connect to the graphical user interface, it enables microservices to communicate between each other, and stores the data in independent non-sql databases.

V. Blockchain implementation of the Case Study

In this section we describe how we implemented the microservices-based system described in Section IV with smart contracts based on Ethereum blockchain. We started from the premise that Ethereum allows the creation of decentralized applications operating in devices connected to the peer-to-peer network. That application works in the Ethereum Virtual Machine and are named smart contracts. To create a running smart contract, developers have to write its source code in solidity language, to compile it with the most updated version of the solidity compiler, and to deploy the smart contract in the blockchain. Once deployed, the smart contract becomes a unique and independent resource, characterized by its address, its logic and its ABI. The smart contract address represents the resource URL from the blockchain's point of view. For that reasons, and to replicate the behaviour of the case study, we implemented a total of three atomic smart contracts, one for each microservice we wanted to implement. Our goal is to create a remotely callable smart contract Doctor which generates requested data by calling other two remote contracts knowing only their address and their interface.

To easily manage the dependencies between elements declarations we wrote our three smart contracts in a single source code compliant with the version 0.5.0 of solidity. We decided to use one of the experimental additional features of the solidity compiler. In particular our system uses the "AbiEncoderV2" to enable functions to returns composite data type like "structs".

First of all we reproduced the microservice Patient and Diagnosis services written in the Java source code that aim to collect data and to provide simply getter and setter functionalities. To reproduce the concept of data encapsulation, we implemented these two classes as solidity libraries. In facts,

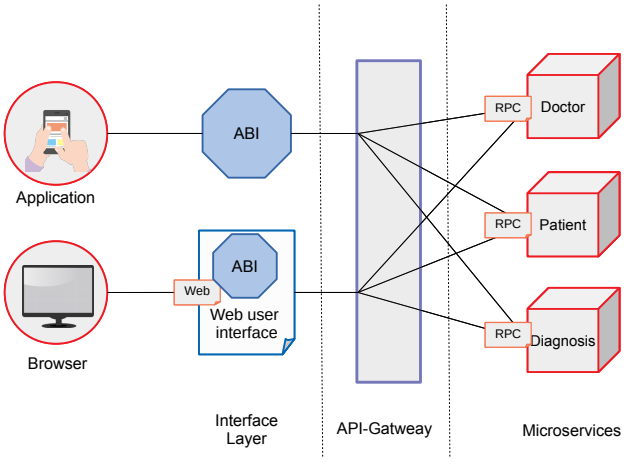

Fig. 1. Microservices-Based Architecture of the system

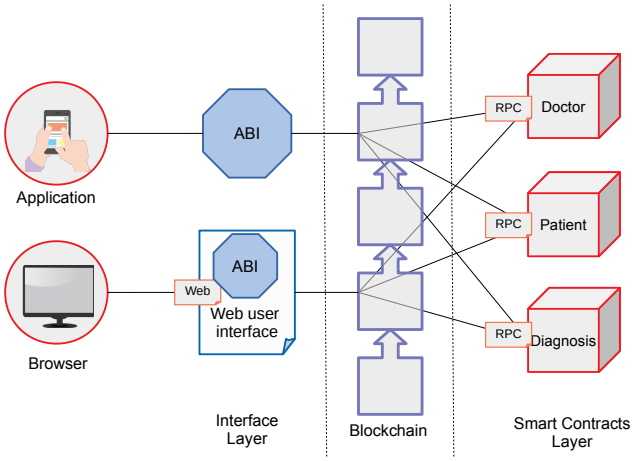

Fig. 2. Blockchain-based Architecture of the system

if library functions are called, their code is executed in the context of the calling contract. In addition, libraries can refer to its own elements using the keyword *self*. This behavior is similar to that of a class calling a class method in object oriented languages.

Subsequently we reproduced the class Doctor. Variables of this class includes diagnosis and patient that are respectively instances of the classes Diagnosis and Patient. In our smart contracts, class instances are implemented as "struct" variables managed by the libraries functions. The library Patient includes variables ID, patient_name, patient_email and related getters and setters. The library Diagnosis includes variables ID, disease_name, disease_ description and related getters and setters.

After created the libraries, we reproduced the behavior of classes PatientRest and DiagnosisRest by solidity contracts. In original, the two Java classes manage an array of instances of Patient or Diagnosis. Instead of arrays, we used the mapping data type. This is a key-value data structure that is more efficient than arrays in the Ethereum blockchain in terms of resources and gas consumption. Rest calls have been replaced by specific "public functions". For this reason we calls our contract PatientPseudoRest and DiagnosisPseudoRest. In solidity the keyword *public* is a visibility modifier that allows the function to be callable both by other contracts and by blockchain users. The public function of the contract PatientPseudoRest representing the microservice is *getPatient*. It takes as input an ID of a patient and returns a Patient data. The public function of the contract DiagnosisPseudoRest representing the microservice is *getDiagnosis*, which takes as input an ID of a diagnosis and returns the complete set of data matching with the ID.

Finally we reproduced the behavior of the class DoctorRest. We ported the behavior of the Java class. It knows the URI of the microservices Diagnosis and Patient, and provides a callable REST function that, given the ID of a patient and the id of a disease, calls and acquires information generated by the two microservices. Finally it returns a composition of the data. We implemented a contract DoctorPseudoRest that calls at run time the contracts. For this reason DoctorPseudoRest must know the addresses of PatientPseudoRest and of DiagnosisPseudoRest. and so the contract constructor requires these two addresses. The DoctorPseudoRest provides the function *submitOrder* replicating the behavior of the equivalent REST call in the original case study. The calling of remote contracts is performed by means of the definition of a contract pointer that allows the contract to execute its public functions. The function *submitOrder* produces a formatted output that includes data related to the specific patient and to the specific disease.

A. Smart Contracts code

Below we report the smart contracts source code.
The first two rows indicate instructions to the compiler.

```
pragma experimental ABIEncoderV2;
pragma solidity >=0.4.25 <0.6.0;
```

The code continues with the definition of the library Diagnosis. This library contains a "struct" in which the diagnosis data are defined. The functions defined below are the getter and the setter. These functions can refer to the data stored inside the library by means of the keyword self.

```
library Diagnosis{

    struct aDiagnosis{
        uint id;
        string disease_name;
        string disease_description;
    }

    function getId(aDiagnosis storage self) view
    public returns(uint id){
        return self.id;
    }
```

```
    function getName(aDiagnosis storage self) view
    public returns(string memory disease_name){
        return self.disease_name;
    }

    function getDescription(aDiagnosis storage self)
     view public returns(string memory
    disease_description){
        return self.disease_description;
    }

    function setId(aDiagnosis storage self, uint
    anID) public {
        self.id=anID;
    }

    function setName(aDiagnosis storage self, string
     memory disease_name) public {
        self.disease_name=disease_name;
    }

    function setDescription(aDiagnosis storage self,
     string memory disease_description) public{
        self.disease_description=disease_description
    ;
    }

}
```

The following code defines the microservice Diagnosis. In place of the REST call, it has a public function *getDiagnosis*. Likewise the original Case study, our SC microservice records a collection of Diagnosis data. These data are written in the contract constructor and are stored in the blockchain at the moment of contract deployment.

```
contract DiagnosisPesudoRest {

    mapping (uint => Diagnosis.aDiagnosis) private
    diagnosis;
    uint numberOfDiseases;

    //In real cases, constructor data should be load
     from a database.
    constructor() public {

        Diagnosis.setId(diagnosis[1],1);
        Diagnosis.setName(diagnosis[1],"Viral Fever
    ");
    Diagnosis.setDescription(diagnosis[1],"Caused by
     viruses are among the most frequent causes of
    fever in adults. Common symptoms can include a
    runny nose, sore throat, cough, hoarseness, and
    muscle aches.");


        Diagnosis.setId(diagnosis[2],2);
        Diagnosis.setName(diagnosis[2],"Pneumonia");
    Diagnosis.setDescription(diagnosis[2],"Symptoms
    include a cough with phlegm or pus, fever,
    chills and difficulty breathing.");

        Diagnosis.setId(diagnosis[3],3);
        Diagnosis.setName(diagnosis[3],"Malaria");
    Diagnosis.setDescription(diagnosis[3],"Symptoms
    are chills, fever and sweating, usually
    occurring a few weeks after being bitten.");

        Diagnosis.setId(diagnosis[4],4);
        Diagnosis.setName(diagnosis[4],"Typhoid");
    Diagnosis.setDescription(diagnosis[4],"Symptoms
    include high fever, headache, stomach pain,
    weakness, vomiting and loose stools.");

        numberOfDiseases=4; //can be updated
    dynamically
    }


  function getDiagnosis(uint id) view public returns
     (uint ID, string memory name, string memory
    description) {
        return(Diagnosis.getId(diagnosis[id]),
    Diagnosis.getName(diagnosis[id]), Diagnosis.
    getDescription(diagnosis[id]));
  }


}
```

In this section we don't show the implementation of the library Patient and of the contract PatientPseudoRest which are very similar to the previous ones.

In the following we show a portion of the library Doctor. It defines the variables and the methods related to the object of Doctor type in the original class. Data include a Patient instance and a Diagnosis instance. The instruction "using" defines to which library the compiler has to refer for each data type. In this piece of code we show the getter and the setter functions for the "diagnosis". Thanks to the ABIencoderV2, the getter can return a "struct" variable.

```
library Doctor {

    using Patient for Patient.aPatient;
    using Diagnosis for Diagnosis.aDiagnosis;

    struct aDoctor{
        uint id;
        uint amount;
        uint appointementDate;
        Patient.aPatient patient;
        Diagnosis.aDiagnosis disease;
    }

    [...]
    function getDiagnosis(aDoctor storage self) view
     public returns(Diagnosis.aDiagnosis memory
    disease){
        return (self.disease);
    }

    function setDiagnosis(aDoctor storage self, uint
     ID, string memory disease_name, string memory
    disease_description) public {
        self.disease.setId(ID);
        self.disease.setName(disease_name);
        self.disease.setName(disease_description);
    }
    [...]
}
```

The following listing defines the Doctor microservice. The constructor takes in input the addresses of the two con-

tracts DiagnosisPseudoRest and PatientPseudoRest. The original REST function is now implemented with the function *submitOrder*. This function, given idPatient, idDiagnosis and the amount, calls the public functions getDiagnosis and getPatient of the two smart contracts. Then the function combines acquired data with produced data (the ID and the date) and produces a Doctor type data. The data is finally sent as output. The behavior of this function is described in the UML diagram in Figure 3.

```
contract DoctorPseudoRest {

    using Doctor for Doctor.aDoctor;
    using Patient for Patient.aPatient;
    using Diagnosis for Diagnosis.aDiagnosis;

    address private DiseaseRestContract;
    address private PatientsRestContract;

    constructor(address DRC, address PRC) public {
        DiseaseRestContract = DRC;
        PatientsRestContract = PRC;
    }

    function submitOrder(uint idPatient, uint
    idDiagnosis, uint amount)
        public view returns(
            uint ID,
            uint Amount,
            uint AppointementDate,
            Patient.aPatient memory thePatient,
            Diagnosis.aDiagnosis memory theDisease){

        Patient.aPatient memory patient;
        Diagnosis.aDiagnosis memory disease;
        Doctor.aDoctor memory doctor;

        DiagnosisPesudoRest DR = DiagnosisPesudoRest
    (DiseaseRestContract);
        (disease.id,disease.disease_name,disease.
    disease_description) = DR.getDiagnosis(
    idDiagnosis);

        PatientPesudoRest PR = PatientPesudoRest(
    PatientsRestContract);
        (patient.id,patient.patient_name,patient.
    patient_emailid)=PR.getPatient(idPatient);

        doctor.id=uint(keccak256(abi.encodePacked(
    now))); //an automatic generated ID
        doctor.amount=amount;
        doctor.appointementDate=now;
        doctor.patient=patient;
        doctor.disease=disease;

        return (doctor.id,
                doctor.amount,
                doctor.appointementDate,
                doctor.patient,
                doctor.disease);
    }
}
```

The gas cost of this function is zero. This is because all operations in the function *submitOrder* are "read only" and involve only "memory" variables that do not change the state of the blockchain.

For example, to create an "order" we can call the function *submitOrder* with the following input:

```
{
  "uint256 idPatient": "2",
  "uint256 idDiagnosis": "3",
  "uint256 amount": "550"
}
```

The smart contract produces the following output:

```
{
  "0": "uint256: ID 66348293397218483239...
  "1": "uint256: Amount 550",
  "2": "uint256: AppointementDate 1546442202",
  "3": "tuple(uint256,string,string): thePatient 2,
    Rachel,patient2@edureka.co",
  "4": "tuple(uint256,string,string): theDisease 3,
    Malaria,Symptoms are chills, fever and sweating,
     usually occurring a few weeks after being
    bitten."
}
```

B. UML Sequence Diagram

In this section we show the sequence UML Diagram related to "order creation" according to the principles of BOSE. The diagram in Fig. 3 represents the following steps:

1. A "client" asks the contract DoctorPseudoRest to create an appointment by means the function createOrder, given the disease_ID, patient_ID and the amount.
2. The contract DoctorPseudoRest, at this point, asks the PatientPseudoRest and DiagnosisPseudoRest contracts for all the data related to those specific IDs.
3. The two contracts respond by providing the required data (a Patient "Object" and a Diagnosis "Object" respectively)
4. Finally the contract DoctorPseudoRest builds the requested data and returns it to the client.

In this sequence, the contract DoctorPseudoRest is at the same time the server for the actor "client" and the client of the services PatientPseudoRest and DiagnosisPseudoRest.

VI. Discussion

In this paper we discussed the concrete possibility of replicating a microservice architecture by mean of a set of Smart Contracts. The topic is tackled with two parallel approaches.

The first one is more general and theoretical and presents the problem, the similarities and the possible distances among the two architectures. However we report how in general a microservice architecture can be mapped into an architecture where Smart Contracts and blockchain are the backbone of the software services provided. The concept of supporting the IDEAL properties is discussed for both cases and the differences among an approach where the software providing services resides in a proprietary server, always accessible and always upgradable and the approach where software is deployed into a blockchain structure where no proprietary servers can be considered for running the software and software can be updated less easily are described.

In the second one we focused the attention on a specific case study where a concrete and paradigmatic example of software

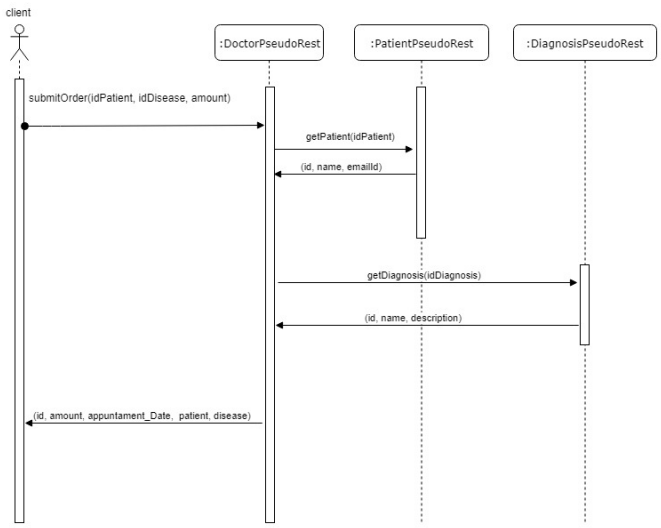

Fig. 3. Sequence diagram of the case "order creation"

service provided by a microservice architecture with three Java classes, even if very simple, has been analyzed and fully reproduced by using three corresponding Smart Contracts. In this specific case study we used a Software engineering approach oriented to the blockchain software, presenting also a sequence diagram which is a guide for both architectures and helps understanding how the set of Smart Contract can solve the problem of providing services as well as the microservice approach. The case study presents a solution showing for the first time how the microservice paradigm can concretely be mapped into a Smart Contract corresponding architecture.

The importance of this result with respect to the research in the field of blockchain-oriented software engineering resides in the fact that the microservice paradigm is presently widely used to provide online services or general services where a monolithic large software system is decomposed into business oriented independent smaller services. Our case study then shows that the Blockchain Oriented Software Engineering can find applications in fields that are presently out of reach of Smart Contract software, simply applying our proposed parallelism between the two architectures and opens new perspective to the practical uses of blockchain.

While the microservice architecture allows to easily update the software and a larger control of the overall software system, all the computations must be performed by the servers where the software resides with all the associated consequences. On the other hand the Smart Contracts approach presents the advantage of broadcasting the computational part to the blockchain nodes and allows intrinsically to keep trace of all changes, update, login, and general transactions performed on the blockchain structure which otherwise require an ad-hoc structure. Costs and benefits of the two approaches are not easy to understand and manage because they may strongly depend upon the specific software service under consideration. Nevertheless our paper traces a path for mapping an existing and largely used software paradigm into the blockchain field, and more specifically into the field of BOSE.

VII. Conclusions

In this paper we afforded the problem of mapping a general microservices software architecture into a software architecture performing the same tasks and providing the same services organized by mean of a set of corresponding Smart Contracts running onto a blockchain.

We concretely solved the problem by implementing a paradigmatic example of microservice pattern with three Smart Contracts written in Solidity where opportune variables and functions realize the same tasks of the corresponding set of Java classes. Our study demonstrates the feasibility of the approach and shows that it is possible to implement a simple microservices-based system with smart contracts maintaining the same set of functionalities and results. We corroborate the study with a BOSE approach where the general software architectures and sequence diagrams are described. This opens up new perspectives for extending the cases where the use of blockchain and Smart Contracts can find applications providing a concrete example where an existing and already diffuse software paradigm can be easily mapped using a blockchain approach. Future work include the re-development of complex systems with smart contracts and the investigation of the similarities between microservices, function as a services and smart contracts.

References

[1] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[2] Taibi, D., Lenarduzzi, V., & Pahl, C. (2017). Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. IEEE Cloud Computing, 4(5), 22-32.

[3] M. Marchesi, L. Marchesi, R. Tonelli, An Agile Software Engineering Method to Design Blockchain Applications, ACM Proceedings of the 14th Central and Eastern European Software Engineering Conference (2018).

[4] Porru, S., Pinna, A., Marchesi, M., & Tonelli, R. (2017, May). Blockchain-oriented software engineering: challenges and new directions. In Proceedings of the 39th International Conference on Software Engineering Companion (pp. 169-171). IEEE Press.

[5] McCorry, P., Shahandashti, S. F., & Hao, F. (2017, April). A smart contract for boardroom voting with maximum voter privacy. In International Conference on Financial Cryptography and Data Security (pp. 357-375). Springer, Cham.

[6] Lenarduzzi, V., Lunesu, I., Marchesi, M., & Tonelli, R. (2018). Blockchain applications for Agile methodologies. In 19th International Conference on Agile Processes in Software Engineering and Extreme Programming. XP (Vol. 2018).

[7] Lenarduzzi, V., & Taibi, D. (2016). MVP explained: A systematic mapping study on the definitions of minimal viable product. Paper presented at the Proceedings - 42nd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2016, 112-119. doi:10.1109/SEAA.2016.56

[8] Casado-Vara, R., Gonzlez-Briones, A., Prieto, J., & Corchado, J. M. (2018, June). Smart Contract for Monitoring and Control of Logistics Activities: Pharmaceutical Utilities Case Study. In The 13th International Conference on Soft Computing Models in Industrial and Environmental Applications (pp. 509-517). Springer, Cham.

[9] Cheng, R., Zhang, F., Kos, J., He, W., Hynes, N., Johnson, N., ... & Song, D. (1804). Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts.

[10] Zhang, Y., & Wen, J. (2017). The IoT electric business model: Using blockchain technology for the internet of things. Peer-to-Peer Networking and Applications, 10(4), 983-994.

[11] Casado-Vara, R., Prieto, J., De la Prieta, F., & Corchado, J. M. (2018). How blockchain improves the supply chain: Case study alimentary supply chain. Procedia computer science, 134, 393-398.

[12] Huh, S., Cho, S., & Kim, S. (2017, February). Managing IoT devices using blockchain platform. In Advanced Communication Technology (ICACT), 2017 19th International Conference on (pp. 464-467). IEEE.

[13] Buterin, V. (2014). A next-generation smart contract and decentralized application platform. white paper.

[14] Cunningham , J., & Ainsworth, J. (2018, January). Enabling patient control of personal electronic health records through distributed ledger technology. In MEDINFO 2017: Precision Healthcare Through Informatics: Proceedings of the 16th World Congress on Medical and Health Informatics (Vol. 245, p. 45). IOS Press.

[15] Ekblaw, A., Azaria, A., Halamka, J. D., & Lippman, A. (2016, August). A Case Study for Blockchain in Healthcare:"MedRec" prototype for electronic health records and medical research data. In Proceedings of IEEE open & big data conference (Vol. 13, p. 13).

[16] Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., & Zohar, Y. (2017). Online detection of effectively callback free objects with applications to smart contracts. Proceedings of the ACM on Programming Languages, 2(POPL), 48.

[17] Kapitonov, A., Lonshakov, S., Krupenkin, A., & Berman, I. (2017, October). Blockchain-based protocol of autonomous business aon blockchains. A case of study about a bug discovered inctivity for multi-agent systems consisting of UAVs. In Research, Education and Development of Unmanned Aerial Systems (RED-UAS), 2017 Workshop on (pp. 84-89). IEEE.

[18] Herbaut, N., & Negru, N. (2017). A model for collaborative blockchain-based video delivery relying on advanced network services chains. IEEE Communications Magazine, 55(9), 70-76.

[19] Wohrer, M., & Zdun, U. (2018, March). Smart contracts: security patterns in the ethereum ecosystem and solidity. In Blockchain Oriented Software Engineering (IWBOSE), 2018 International Workshop on (pp. 2-8). IEEE.

[20] Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A., & Hierons, R. (2018, March). Smart contracts vulnerabilities: a call for blockchain software engineering?. In Blockchain Oriented Software

[30] Will Warren, Amir Bandeali. 0x: An open protocol for decentralized exchange on the Ethereum blockchain, https://0x.org/pdfs/0x_white_paper.pdf

Engineering (IWBOSE), 2018 International Workshop on (pp. 19-25). IEEE.

[21] Bragagnolo, S., Rocha, H., Denker, M., & Ducasse, S. (2018, March). SmartInspect: solidity smart contract inspector. In 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE) (pp. 9-18). IEEE.

[22] Fenu, G., Marchesi, L., Marchesi, M., & Tonelli, R. (2018, March). The ICO phenomenon and its relationships with ethereum smart contract environment. In Blockchain Oriented Software Engineering (IWBOSE), 2018 International Workshop on (pp. 26-32). IEEE.

[23] Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media. ISBN: 978-1491950357

[24] S. Kappagantula (2018). "Microservices Tutorial Learn all about Microservices with Example" Edureka. Online: https://www.edureka.co/blog/microservices-tutorial-with-example

[25] D. Taibi, V. Lenarduzzi, and C. Pahl (2018) Architectural Patterns for Microservices: A Systematic Mapping Study, in 8th International Conference on Cloud Computing and Services Science, CLOSER

[26] D. Taibi, V. Lenarduzzi (2018) On the definition of microservice bad smells. IEEE Software, 35(3), 56-62. doi:10.1109/MS.2018.2141031

[27] Taibi, D., Lenarduzzi, V., Ahmad, M. O., Liukkunen, K. (2017). Comparing communication effort within the scrum, scrum with kanban, XP, and banana development processes. Paper presented at the ACM International Conference Proceeding Series, , Part F128635 258-263. doi:10.1145/3084226.3084270

[28] Taibi, D., Lenarduzzi, V., Janes, A., Liukkunen, K., Ahmad, M. O. (2017). Comparing requirements decomposition within the scrum, scrum with kanban, XP, and banana development processes doi:10.1007/978-3-319-57633-6_5

[29] Tonelli, R., Pinna, A., Baralla, G., & Ibba, S. Ethereum Smart Contracts as Blockchain-oriented Microservices.

[31] https://ethereum.github.io/yellowpaper/paper.pdf

[32] https://media.readthedocs.org/pdf/solidity/develop/solidity.pdf

[33] S. Ibba, A. Pinna, M. Lunesu, M. Marchesi, R. Tonelli, Initial Coin Offerings and Agile Practices, Future Internet, Vol. 10, N. 11, Pag. 103

[34] Hartmann, F., Wang, X., & Lunesu, M. I. (2018, March). Evaluation of initial cryptoasset offerings: the state of the practice. In Blockchain Oriented Software Engineering (IWBOSE), 2018 International Workshop on (pp. 33-39). IEEE.

[35] Pinna, A., Tonelli, R., Orru, M., Marchesi, M. (2018). A Petri Nets Model for Blockchain Analysis, The Computer Journal, Volume 61, Issue 9, 1 September 2018, Pages 13741388, doi:10.1093/comjnl/bxy001