FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Edge Blockchain Provisioning for Mobile Edge Computing Applications

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

BSc Filip Rydzi
Matrikelnummer 01226452

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Dr.techn. Hong-Linh Truong

Wien, 30. April 2019

Filip Rydzi	Hong-Linh Truong

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Edge Blockchain Provisioning for Mobile Edge Computing Applications

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

BSc Filip Rydzi

Registration Number 01226452

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dr.techn. Hong-Linh Truong

Vienna, 30th April, 2019

Filip Rydzi Hong-Linh Truong

Erklärung zur Verfassung der Arbeit

BSc Filip Rydzi
Kĺzavá 31, 83101 Bratislava, Slovakia

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30. April 2019

Filip Rydzi

Acknowledgements

I would first like to thank my thesis supervisor Priv.-Doz. Dr. Hong-Linh Truong. He was a great support ready to answer all my questions. He invested a lot of his time to help me throughout the thesis by providing valuable feedback. He added a lot of effort to form the idea behind this thesis into a scientific topic.

I would also to thank to all the lecturers of the courses I attended during my study at Vienna University of Technology for providing me extensive background knowledge required for this thesis.

Finally, I want to express my very profound gratitude to my parents, my sister and my brother for their support during writing the thesis and all the years of my study. Without that support I would never be able to finish my studies.

Kurzfassung

Die Blockchain in Mobile Edge Computing (MEC) hat in letzter Zeit die Aufmerksamkeit der Forscher gewonnen. Die Entwickler stehen vor neuen Herausforderungen, wenn sie blockchain-basierte Anwendungen in MEC implementieren. Sie müssen Designentscheidungen hinsichtlich der Auswahl einer geeigneten Bereitstellung der Funktionen der Blockchain für die MEC-Komponente treffen. Die ressourcenbeschränkten Geräte des Internets der Dinge (engl. Internet of Things-IoT) gehören zu jenen Komponenten. Durch die Ausführung der Funktionen der Blockchain müssen die MEC-Komponenten komplexe Operationen der Blockchain ausführen. Die komplexen Operationen verursachen einen erheblichen Rechnenaufwand für die IoT-Geräte.
Um den Entwicklern bei der Bewältigung dieser Herausforderungen zu helfen, entwickeln wir in dieser Arbeit ein Blockchain-Benchmarking-Framework. Unser Framework kann verschiedene Muster von Blockchain-Interaktionen zwischen den MEC-Komponenten auswerten. Während einer Auswertung werden verschiedene Bereitstellungen der Funktionen der Blockchain für die MEC-Komponenten evaluiert. Um die Funktionen der Blockchain in den MEC-Komponenten ausführen zu können, ordnen wir den Funktionen ausführbare Blockchain-Artefakte zu. Darüber hinaus kann das Framework bewerten wie gut die Interaktionen unter Verwendung von verschiedenen Konfigurationen der zugrunde liegenden Infrastruktur abschneiden. Unser Framework misst die Leistung und die Zuverlässigkeit einer Auswertung anhand folgender Qualitätskennzahlen: die Transaktionsakzeptanzrate und -zeit, die Skalierbarkeit und die Hardware-Auslastung der Infrastrukturressourcen. Außerdem schlagen wir ein Experiment-Wissens-Service vor. Das Service verwaltet die Daten der Auswertungen. Zweck des Services besteht darin, den Entwicklern das Wissen, das durch die Auswertungen gesammelt wurde, zur Verfügung zu stellen. Die Entwickler können dieses Wissen wiederverwenden, wenn sie ihre eigenen blockchain-basierte Anwendungen in MEC-Umgebungen entwerfen.

Im Rahmen dieser Arbeit haben wir Prototypen des Frameworks und des Service implementiert. Eine Menge von Vehicle-to-Everything (V2X)-Kommunikationsszenarien wurde verwendet um verschiedene Muster von Interaktionen zwischen den MEC-Komponenten zu identifizieren. Um die Flexibilität unseres Frameworks zu demonstrieren haben wir 324 Experimente, basierend auf den identifizierten Interaktionen, erstellt und ausgewertet. Wir haben erklärt was die Ergebnisse der Auswertungen für die Entwickler bieten. Weiters haben wir konkrete Beispiele gezeigt wie die Entwickler von dem Service profitieren können.

Abstract

Blockchain in Mobile Edge Computing (MEC) environments has gained attention of researchers recently. Developers face new challenges when implementing blockchain-based applications in MEC. They have to make design decisions regarding choosing a suitable deployment of blockchain features to MEC components, which include resource-restricted Internet of Things (IoT) devices. By running the blockchain features, MEC components have to carry out complex blockchain operations, which cause a lot of computational overhead for the IoT devices.
To help the developers to address the challenges, we propose a blockchain benchmarking framework in this thesis. The framework is able to evaluate different patterns of blockchain interactions among MEC components. Various deployments of blockchain features to MEC components, involved in the interactions, are also benchmarked. To be able to deploy and execute the blockchain features in the MEC components, we map the features into executable blockchain artefacts. Beside that, the framework can evaluate how well the interactions perform when utilizing diverse configurations of the underlying infrastructure. Our framework measures performance and reliability of a benchmark via following quality metrics: transaction acceptance rate and time, scalability and hardware utilization of infrastructure's resources.
Furthermore, we propose an experiment knowledge service, which manages data related to benchmarks. The purpose of the service is to provide knowledge gathered by the benchmarks to developers. The developers can reuse the knowledge when designing their own blockchain-based applications in MEC environments.

We have implemented prototypes of the framework and the experiment knowledge service within the scope of this thesis. A set of Vehicle-to-Everything (V2X) communication scenarios have been utilized to identify various interaction patterns among MEC components. To demonstrate the flexibility of our framework we have generated and benchmarked 324 experiments based on the identified interactions. We have explained what find outs do the benchmarks provide to the developers. Furthermore, we have shown examples of how the developers can benefit from the experiments knowledge service.

Contents

List of Figures

List of Tables

Listings

Acronyms

IoT	Internet of Things.
IoV	Internet of Vehicles.
MEC	Mobile Edge Computing.
PoW	Proof-of-Work.
RSU	Road-side unit.
TOSCA	Topology and Orchestration Specification for Cloud Applications.
tx	transaction.
V2V	Vehicle-to-Vehicle.
V2X	Vehicle-to-Everything.
V2X PaaS	V2X Platform-as-a-Service.
VANET	Vehicular ad-hoc network.
VM	virtual machine.

CHAPTER 1

Introduction

1.1 Introduction

Brody et al [8] describe the future of Internet of Things (IoT) as *"Internet of decentralized, autonomous things"* [8]. The authors of [8] and Sharma et al [49] identify the requirements of the future IoT environment and state that, to address those requirements, a scalable decentralized trust-less peer to peer messaging solution has to be developed. Those requirements arise because the state-of-the-art (partially) centralized architectures [52], [17] are not able to handle exponential growth of IoT devices [32], can't address security vulnerabilities of IoT devices [35] and aren't able to overcome attempts by corporations, or governments to take control over the data, produced by IoT devices [8].
Recently, many researchers have studied intensively a conjunction of blockchain technology with IoT concept in their works [9], [35], [49], [19], [8], [18], [58], because blockchain has the ability to address the requirements of the future IoT. Kshetri et al [35] summarized a list of benefits brought by utilization of blockchain to IoT. Kshetri et al highlighted that blockchain solution eliminates a need for centralized entity and thus avoids single point of failure. No centralized entity implies that the blockchain-based IoT environment could handle the exponential growth of IoT devices. Furthermore, the authors of [35] explain acquisition of secure message communication among the IoT devices, since the transactions are signed and verified cryptographycally. Therefore, blockchain-based IoT eliminates a possibility of data hijacks. As stated by the authors of [46], blockchain is basically an immutable ledger, thus helps to avoid manipulation of data stored in blockchain.
It would be pretty challenging to manage so many IoT devices by humans in the future, thus autonomous operations carried out by the devices are required [9]. Consider a use case when an IoT device sends a request to chargeable API [22]. The IoT device has to be charged, before using that API. That interaction has to take place automatically, without any user's intervention. Christidis et al [9] state that such challenge is possible

to address by utilizing blockchain's smart contracts and associating each IoT device with a cryptocurrency account.
Nevertheless, Dori et al [19] and Samaniego et al [47] state in their studies that deploying blockchain to IoT will bring a set of new critical challenges. The developers, aiming to implement a blockchain solution in the IoT environment, will have to deal with frequently changing IoT topologies [17], which is challenging in blockchain environments [34]. Furthermore, the developers have to find out where to host the blockchain, since the IoT devices are restricted in computational resources and blockchain involves a mining task, which is a particularly computationally intensive operation. The mining task is time consuming, while in many IoT use cases, a low latency is desirable [52]. Dori et al [19] state that scalability of blockchain is another challenge that is in contrary with IoT topologies, which should be able to scale to large number of nodes. Further, the authors of [19] state that the blockchain protocols create a lot of overhead traffic, which might be unacceptable for some bandwidth-limited IoT devices.
To address the challenges of hosting complex blockchain features to resource limited IoT devices, Xiong et al [58] propose an architecture, which offloads the mining tasks to the edge nodes at base stations of Mobile Edge Computing (MEC) environment. The MEC environment has been intensively studied by researchers in their works [52] [17]. Dorri et al present a blockchain-based IoT in smart home environment in their study [18]. They suggest to offload the blockchain's mining to computational rich resources as well.

To summarize this section, we conclude that blockchain with IoT is a powerful concept. Since blockchain involves computationally intensive operations, which can be too complex for resource restricted IoT devices. There have been architectures proposing to offload the computationally intensive operations to edge nodes of MEC. However there are still open challenges for developers, who implement an application based on blockchain in IoT environments.

1.2 Motivation Scenario

To motivate the work in this thesis, we illustrate a scenario in the domain of connected vehicles [5]. Connected vehicles present a building block of Internet of Vehicles (IoV) [39]. IoV can be considered an instance of IoT within the MEC environment. Vehicular ad-hoc network (VANET) [5] is a state-of-the-art framework enabling the communication among vehicles, regarding exchange of their driving properties (velocity, acceleration, distance to other vehicles, issued warnings about danger on the road, etc.). As stated by the authors of [2] that kind of communication should enrich safety on the roads, improve the fluency of traffic, and is a key step towards making the vehicles fully autonomous. Nevertheless, VANET is utilizing cellular wireless networks, which are vulnerable to attacks [2].
The utilization of blockchain in this domain gained attention of researchers shown in several published studies [60], [50], [20], [36], [59]. That is mainly because blockchain has a potential to address the security vulnerabilities of VANET. In parallel, Benjamin et al [36] research a possible utilization of smart contracts to enable autonomous interactions among stakeholders involved in the vehicle communication environment.

A scenario, which motivated us towards the research within this thesis, is the following: Consider a developer aims to develop an application, which is utilizing blockchain, for connected vehicles scenarios in MEC environment. The application, we use as an example is a blockchain-based application addressing safe lane change scenario[6]. As depicted in Figure 1.1, instances of the application are running in the components of MEC and are interacting via blockchain by exchanging the driving properties. The application's instances running in vehicles are exchanging the driving properties with other vehicles either directly or over a Road-side unit (RSU) [5] or an edge node. The cloud node is utilized to track a history of the exchanged driving properties. The developer of the application needs to make several design decisions, which involve answering the questions below. We elaborate deeper on the scenario in Chapter 3.

1. Which topologies, representing interaction patterns among MEC components, could arise in the scenario?
2. What blockchain features should be deployed to which nodes across the topology, in order to achieve the desired functionality and certain quality metrics [55], [34] for the topology?
3. Which blockchain implementation is most convenient for the scenario?
4. What are hardware requirements of resources of the underlying topology's infrastructure?

Therefore, one of the goals of this thesis is to develop a blockchain benchmarking framework, which can be used by developers to benchmark the blockchain interactions in the topologies, against a set of quality metrics. The framework should help developers to deal with blockchain, by answering the above-questions 2 to 4 above, during design phase of an application in MEC environment. Please note that the first question has to be answered by the developer.
Additionally we propose and implement a knowledge service, which stores data related to the benchmarks. The main purpose of the service is to reuse knowledge gathered by benchmarks to help developers during design phase of the application. For instance the developers can benefit from the knowledge stored by the service by obtaining recommendations, which help them to address the above-named questions 2 to 4 as well, without repeating benchmarks.

1.3 Research Questions

Based on the scenario and goals discussion above, within the thesis we aim to answer the following research questions:

- **RQ1 Scenarios and Requirements** What interactions among MEC components could arise for an application addressing a scenario in blockchain-based MEC

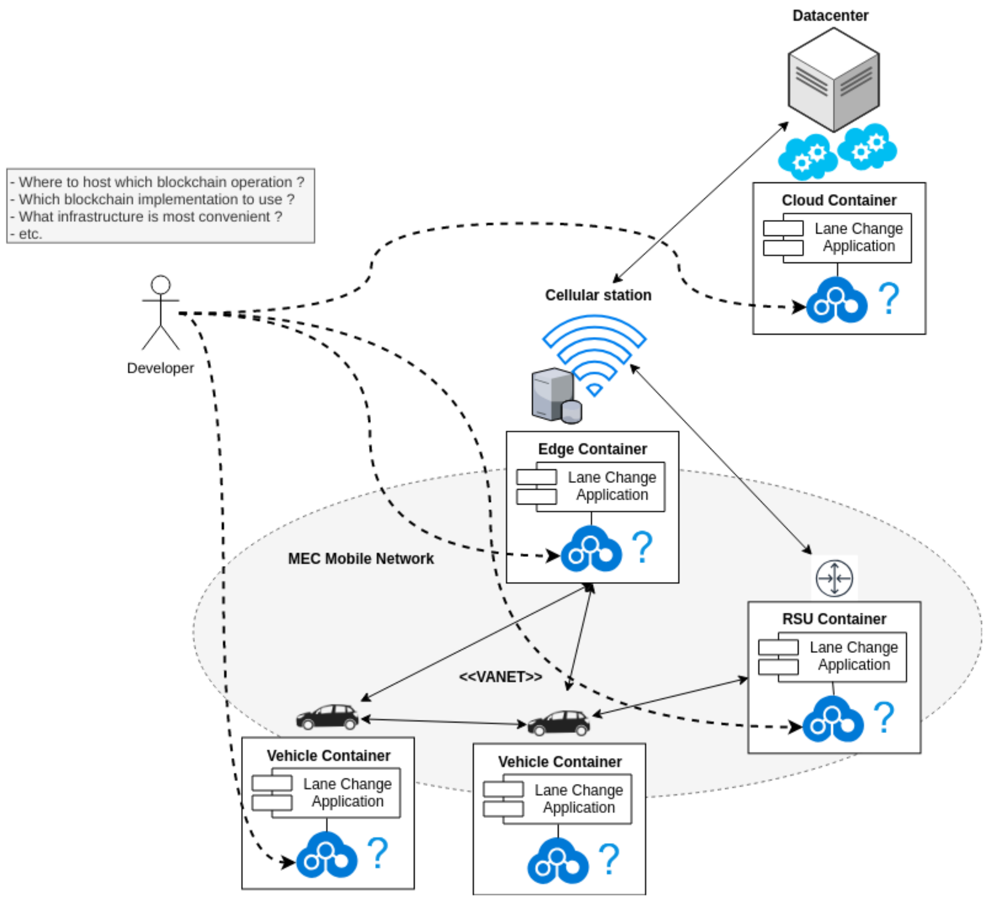


Figure 1.1: Motivation Scenario: Safe lane change [6] in MEC

environment? What metrics of quality do we consider as most important for the developers, who are dealing with the blockchain-based MEC environment? What are the developer's requirements in terms of those quality metrics in the scenarios? How can we map the blockchain features to executable blockchain artefacts, which can be deployed to MEC's components?

- **RQ2 Benchmarks** How do we design a benchmark? How does architecture and workflow of the benchmark framework look like? How can the framework help the developers in addressing the challenges they face when developing blockchain-based application in MEC? Which of the identified interactions among MEC components do we benchmark? What results have been achieved by benchmarking those interactions? What insights do those results bring for the developers?
- **RQ3 Knowledge Service** Which data, related to the benchmarks, do we store in the service? What service operations do we expose by the knowledge service to enable management of the data? How can developers utilize the knowledge stored

by the service? What specification/language do we use to represent a topology of the application?

1.4 Contribution

Towards answering the research questions, the thesis contributes with the following framework and service, supporting the listed key features.

- **Benchmark Framework** - It accepts a specification of an experiment, along with a topology on input. The topology is representing MEC components interacting via blockchain and is going to be benchmarked. In case there is no real MEC infrastructure available for the developer, such that the developer can execute the benchmarks on that infrastructure, the framework interfaces with a provider of resources (VMs, networks, etc.). The resources are used to emulate the real MEC infrastructure for the benchmarks. Furthermore, the framework can simulate various qualities of the network connection between nodes of the topology. Our framework has an ability to automatically deploy blockchain features and software artefacts, responsible for executing the benchmarks, to the topology.
- **Experiment Knowledge Service** - Defines a clear structure of the data, related to the benchmarks, which are stored by the service. The service exposes APIs to enable management of the stored data. It interfaces with *Benchmark Framework* via a *Results Parser* utility, which is used to transform outcome of a benchmark from the framework to the format accepted by *Experiment Knowledge Service*. Furthermore, it can be used by developers to obtain recommendations about deployment of blockchain features and hardware properties of a MEC infrastructure for an application's topology.

We implemented prototypes of the proposed *Benchmark Framework* and *Experiment Knowledge Service* and make them available in the GitHub repository under the following link: `https://github.com/rdsea/blockchainbenmarkservice`. We run extensive benchmarks by utilizing the framework, the obtained results are published in the experiments directory in the repository.

The infrastructure resources (VMs, networks...), we utilized when running our benchmarks, were funded by Google Cloud Platform Research Grant and TU Wien.

1.5 Structure of the work

The remainder of this thesis is structured as follows: In Chapter 2 we give an overview of the State of the Art, required background information and related works. In Chapter 3 we present quality metrics, which we assume to be most important for developers of applications based on blockchain in MEC. In the chapter we present scenarios, we consider in blockchain-based MEC and identify interactions among MEC components, which could occur in those scenarios. We formulate developer's requirements in form of the introduced metrics for the scenarios. In order to address the requirements, we create and benchmark multiple experiments in Chapter 4. We discuss design of a benchmark. Furthermore, we propose and implement a framework to execute the benchmarks. To the end we create a set of experiments resulting from our scenarios, benchmark those by utilizing the framework and evaluate what insights do the results bring for the developers. In Chapter 5, we present a service to store the data related to the benchmarks. We elaborate on the prototype of the service, and explain exposed operations. We describe how the developers could benefit from the service. Finally, the Chapter 6 concludes this thesis and gives future work.

CHAPTER 2

State of the Art

2.1 Overview

In this chapter we begin by explaining the background (Section 2.2) for our research. Within that we discuss blockchain in MEC environments, along with its key operations. We describe two chosen implementations of blockchain (Ethereum and Hyperledger-Fabric) that we selected for the purposes of this thesis and we justify why we selected those. Further we elaborate on the connected vehicles domain. In Section 2.3 we discuss the related work. We focus on topics concerning the cooperation of blockchain with Vehicle-to-Everything (V2X) communication, as well the existing benchmarking frameworks for blockchain systems.

2.2 Background

2.2.1 Blockchain in MEC Applications and Systems

Blockchain is a tamper-proof, cryptographically signed distributed ledger, with a permanent store of all transactions, which ever took among the participants [46]. Blockchain is known mainly as a backbone technology beyond Bitcoin [41]. Since then, many other blockchain implementations have came into existence with new interesting features. Ethereum introduced smart contracts [46]. As explained in the work by Crosby et al [14], the smart contracts are small computer programs running at blockchain nodes, capable of automatically executing the terms of a contract. When a certain predefined condition is met, the smart contract's code is automatically triggered to perform an operation among involved entities. By utilization of the smart contracts, blockchain can eliminate the need for a centralized authority. Crosby et al [14] also state that blockchain isn't very efficient for storing the data. In [14], the authors state that it's advisable to store only hash of the data to blockchain, while the data itself can be securely stored on an

external storage provider.
Xiong et al [58] have studied intensively a deployment of blockchain to MEC. In their architecture they propose to offload the mining task from IoT devices to the resource richer edge computing nodes. A mobile blockchain application is running in the IoT devices, enabling the interactions with edge computing service providers. The authors explain that these interactions can be modeled as market activities, such that the edge service providers sell data and computing power. The providers are then accordingly being rewarded by IoT devices. This reward process can be considered as transactions in the blockchain.
Dorri et al [18], [19] presented a case study of using IoT with blockchain for smart home from the security point of view. They used an approach having a miner within each smart home, which is responsible for the communication with outside world and preserves the blockchain used for controlling the communication. In their approach, they offload the mining from the smart devices to the miner, which may be a local PC or a smart hub. We can map their architecture to the MEC environment, such that we consider local PC and smart hubs as the edge nodes, while the smart devices are IoT instances.

We want to benchmark blockchain interactions among MEC components. To make the interaction possible, several blockchain operations have to be carried out in the components. Christidis et al [9] identifies those blockchain operations. Below we provide an example of interaction, which involves the blockchain operations.
Assume a following situation: three vehicles (let's call them `vehicle1`, `vehicle2` and `vehicle3`) and an edge node in a certain area are interacting over blockchain in the MEC environment. The vehicles and the edge node are participants of a blockchain network. Each of the participants deploys some blockchain features and a blockchain-based application to address obstacle on the road scenario [27].

Create a transaction
The application in vehicles is subscribed to the data from vehicle's sensors to detect obstacles on the road. Let's assume `vehicle1` detects an obstacle and wants to inform the `vehicle2` and `vehicle3` about that. The application in `vehicle1` creates a blockchain transaction, which should invoke a smart contract with data related to the obstacle (location, size) as parameters. The smart contract will issue a warning about the obstacle and send the warning to other vehicles.

Sign a transaction
The application has access to a private key of the `vehicle1` (or its car owner or driver), retrieves the private key and sign the created transaction.

Submit a transaction
We assume the `vehicle1` runs a blockchain operation, which enables submitting the transactions to the blockchain network. When the transaction is submitted, the application is notified that transaction has been successfully submitted to the blockchain network, but neither the edge node nor the nearby vehicles have received the warning

yet, because the transaction hasn't been included to the ledger yet.

Verify a transaction
Let's suppose the `vehicle2` and `vehicle3` run a blockchin operation, which can verify the submitted transaction. The verification is done by checking the validity of transaction's signature.

Achieve consensus
Since blockchain is decentralized, there emerge different transactions at various times, which may disagree about what is the truth. E.g. a edge node contains transaction that envelops a warning, which states that there is an obstacle at a particular location, but according to `vehicle1`, there is no obstacle at that location. Therefore, consensus has to be achieved to guarantee the correct ordering and validation of the transactions. In our case, it's pretty simple since we assume only a single transaction. However, the authors of [46] explain the importance of consensus. There has to be one or more blockchain nodes, called miners [9], which are responsible for achieving consensus. For our example we assume the miner node is running in the edge node. Detail of consensus can be found at [46] [47] [51] [48].

Accepting a block
When the consensus has been achieved, a new block containing the transaction has been generated by the miner (edge node in our case). The new block is broadcast to other participants of the blockchain network. Those participants (`vehicle1`, `vehicle2` and `vehicle3`) execute the transactions included in the block locally and append the block to their blockchain ledgers. That involves executing the smart contract, which issues warning about the obstacle on the road spotted by `vehicle1`.

As we explained in Section 1.2, one of the challenges faced by the developers, when developing a blockchain-based application in MEC, is to find a deployment of blockchain features to the MEC components. In order to obtain those blockchain features, we group the blockchain operations to the features, such that each feature can be represented by an executable blockchain artefact (more on that in Section 3.5). Then the blockchain artefacts can be deployed to the interacting MEC components.

2.2.2 Implementations of Blockchain

In this section we describe two chosen implementations of blockchain, along with their key operations. We utilize those implementations to test our benchmarking framework. We aimed to concentrate on blockchains, which may be interesting for developers to use in MEC, because of underlying consensus algorithms. There are consensus algorithms [48], which might be promising for the resource-limited IoT devices and low-latency applications. Beside that we preferred blockchains, which are open-source, well-known with a strong community.

2.2.2.1 Ethereum

The first open source implementation of the blockchain technology that we have chosen is Ethereum [23]. We have chosen the Ethereum implementation, because of smart contracts, its rich community and the promising scalability (according to the work by Vukolic et al [57]), which seems to be very feasible for any IoT topology.
The high latency of Ethereum's consensus algorithm is very challenging and may not acceptable for the latency critical use cases. Thus we can consider Ethereum as kind of a baseline in the benchmarks. The key-building blocks and operations, carried out on interactions, of Ethereum are very similar to the general ones described in Section 2.2.1.

1. Create a transaction object.

2. Sign the transaction by creator's private key.

3. Submit and validate the transaction. In this phase an Ethereum node verifies if the transaction was signed by its creator.

4. Broadcast the transaction to the network.

5. Miner node mines a new block containing the transaction.

6. All blockchain nodes receive the new block and perform synchronization with their local copy of the blockchain.

2.2.2.2 Hyperledger-Fabric

The other blockchain that can be used in MEC applications is Hyperledger-Fabric [28] [29] [31]. We have chosen that because it utilizes the Simplified Byzantine Fault Tolerance [48] [28] algorithm to achieve consensus. As stated in the work by Vukolic et al [57] that algorithm can achieve minimal latency, and doesn't add so much overhead to the network as Proof-of-Work (PoW). Thus it might be feasible to use it with IoT devices in MEC environment.
Hyperledger-Fabric has three types of nodes in the blockchain network, namely: *client*, *peer* and *orderer* [28]. The client node is responsible for creating and submitting transactions to the network. All peers contain a local copy of the ledger and chaincode (chaincode is a smart contract by Hyperledger-Fabric). There are three types of peers (anchor, endorser and leader) [28], each with a specific responsibility. The orderers takes care of correct ordering of the transactions before they are added to the ledger. That is the approach how Hyperledger-Fabric achieves consensus.
Further we proceed to explaining the operations carried out on interactions.

1. A client creates and signs a transaction proposal. The proposal is sent to the endorsing peers.

2. Endorsing peers execute the proposal locally and sign it as endorsed. Please note that the endorsing peers only simulate the transaction on their copy of the data, they don't really change any data in this step.

3. As soon as the client receives enough signatures, such that endorsement policy is accepted, it submits the transaction to the ordering service. The endorsement policy might say, e.g. that a proposal is accepted, if and only if at least one peer from each organization endorses it.

4. The ordering service receives the transaction, puts it to the correct order with the other transactions and submits them to the ledger as a new block.

5. The leader peer is notified by the ordering service to distribute the update to all peers.

6. All peers validate and execute the transactions in a new block locally. The data in the ledger are modified in this step.

2.2.3 Connected Vehicles and V2X Communication

As we noted in Section 1.2, we centered our scenario around the domain of connected vehicles in MEC environment. We want to help the developers to implement an application based on blockchain in that domain. Therefore, the developers are interested in which components they can use for deploying the application. Simultaneously, the developers need to know what topologies, represented by the interactions among the components over blockchain, can occur in that domain. The developers deploy blockchain features onto the components and benchmark the interactions by utilizing our benchmark framework. Connected vehicle is a vehicle equipped with networking capabilities, enabling it to communicate over the internet or ad-hoc networks with other endpoints (vehicles, road infrastructures, etc.). As stated by Faezipour et al [24], the concept of connected vehicles has been introduced with its main objectives to improve safety on the roads, and fluency of traffic and to save costs by optimizing the driving properties.
Harding et al [27] identified a set of applications within the domain of connected vehicles. V2X communication systems are the key enablers for the connected vehicles. Baldessari et al [5] identified following main components of the V2X communication system:

- **Vehicle** represents the vehicle itself. It can run various V2X applications and exchange data with other vehicles and entities of the MEC infrastructure (road signals, smart phones, etc.). At a lower level there are two V2X components within the vehicle: on-board unit (OBU) and application unit (AU). The AU is a container for V2X applications and is permanently connected to the OBU, which provides its networking capabilities to the AU. For the purpose of this thesis we consider both OBU and AU as a single part of vehicle component.

- **RSU** is a physical networking device located along the roads. This provides communication capabilities by extending the range of ad-hoc network for the vehicles. It may be connected to the Internet and might run applications as well.

We want to obtain information about the interactions among those components for the developers. That brings us to the domains of V2X system, explained in the manifesto by Baldessari et al [5]:

- In-vehicle domain: comprises the communication within a vehicle. It's composed of OBU and AU. As already mentioned, we consider OBU and AU as one component for the purpose of the thesis. Thus this domain is not interesting for our research.
- Ad-hoc domain (VANET): VANET is ad-hoc network optimized for the mobility of its participants. It utilizes dedicated short range wireless communication technologies. This domain wraps Vehicle-to-Vehicle (V2V), V2I and V2X communications [24]. In V2V vehicles are exchanging driving data (velocity, acceleration, etc.). In V2I we consider passing warnings about obstacles/dangers on the road between RSUs and the vehicles (V2I). V2X includes communication between vehicles and other points of connection as smart phones, pedestrians, etc.
- Infrastructure domain: a vehicle might be connected directly to a cellular station (edge node), which provides Internet connection for the vehicle. In this domain we consider V2I and V2X communications.

According to the study by Xiong et al [58], VANET is a local mobile IoT network within the MEC environment. Baldessari et al [5] support that by showing that edge node is involved in the infrastructure domain of V2X. While the vehicles and RSUs are basically instances of IoT devices connected to the edge node.
Figure 2.1 illustrates a physical layer of V2X in the MEC. It shows three main tiers of MEC: IoT devices, edge tier and cloud tier. The vehicles and RSUs are considered as instances of IoT devices. These communicate among each other and with the edge nodes, which are located at cellular station. The edge node is connected to the cloud and is utilizing capabilities (storage, processing, etc.) of the cloud.

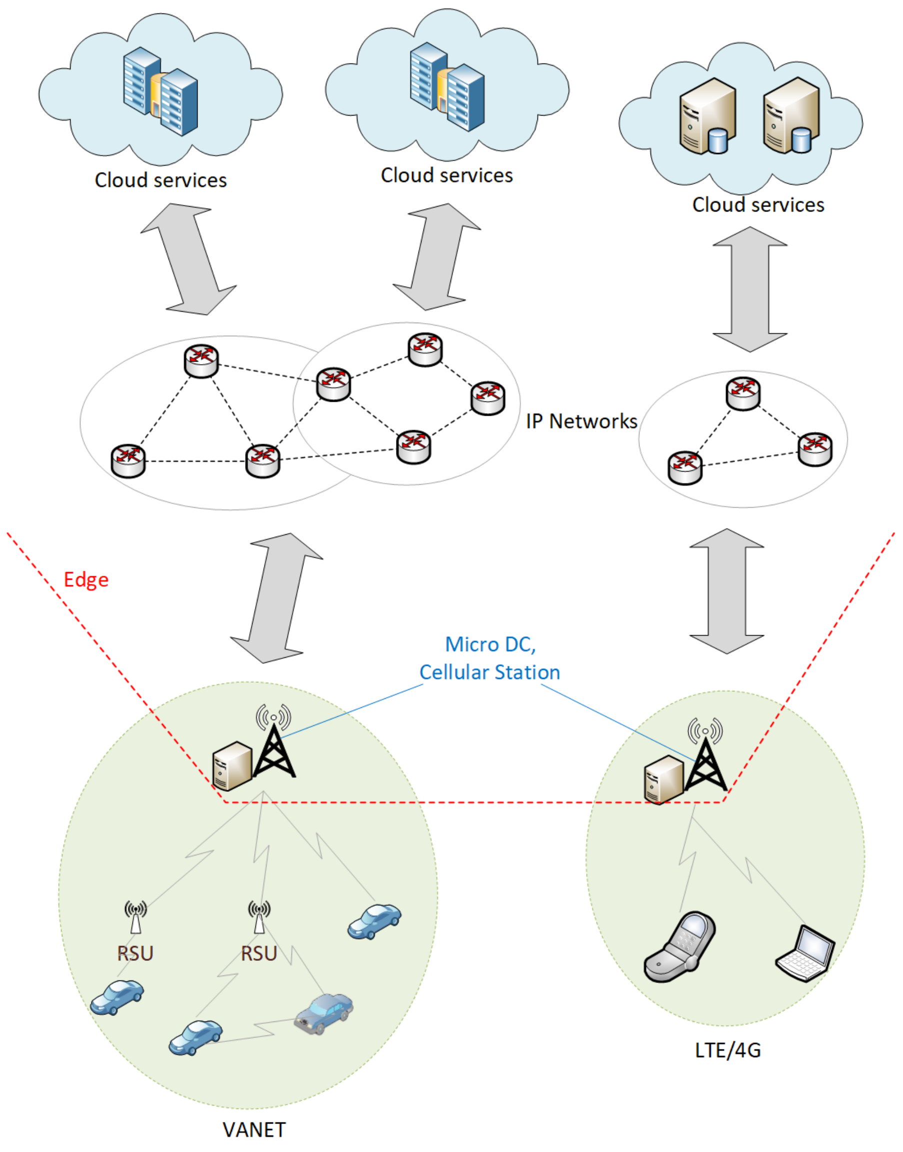


Figure 2.1: Physical view of V2X within the MEC environment, taken from Xiong et al [58]

2.3 Related Work

2.3.1 Blockchain Benchmark Frameworks

Dinh et al [15] proposed a framework, called BLOCKBENCH, to benchmark a private blockchain-based system. It provides APIs to integrate the framework with the private blockchain-based system. The system is further benchmarked against prepared workloads, based on real data and smart contracts. They utilized the following evaluation metrics

to measure the performance of the system based on blockchain: throughput - number of successful transactions per second, latency - response time per transaction, scalability - how does the throughput and latency change when increased number of system's topology nodes, fault tolerance - how does a failed node affect throughput and latency. Currently, their framework supports three major blockchains: Ethereum, Parity and Hyperledger-Fabric.
In our work we reuse the BLOCKBENCH interpretation of the scalability metric. However, BLOCKBENCH doesn't suit needs of the developers, who we want to help by providing our benchmark framework. That is because their focus lays on creating applications based on blockchain in MEC, therefore our framework has to deal with interactions in MEC. Furthermore, the requirement of the developers is to measure hardware utilization of resources in the solution's infrastructure, while BLOCKBENCH doesn't consider that.
A benchmarking framework Caliper [30] has been developed by Hyperledger community. It allows to test a blockchain solution via a set of predefined use cases. A benchmark scenario is configured via a configuration file. In the configuration file a tester can define how to start and stop the solution, set the number of transactions to be created, control the submission rate of transactions and etc. Caliper supports currently only blockchain solutions from the Hyperledger community, including the following: Burrow, Composer, Fabric, Iroha and Sawtooth. However, they promise to support Ethereum in future as well. Caliper considers the following performance indicators for its benchmarks: transaction success rate, throughput, latency, resource consumption.
Caliper doesn't address the requirements of developers, who we consider within this thesis, because they need a framework which focuses on benchmarking blockchain interactions among components in MEC.
A study by Thakkar et al [55] inquires into performance bottlenecks of Hyperledger-Fabric. In the first phase of the work they gained knowledge about the impact of various configuration parameters on the performance. Those parameters included: block size, number of utilized channels, allocation of resources, endorsement policy and utilized state database (GoLevelDB vs CouchDB). They found out that sequential endorsement policy validation and verification, together with the CouchDB were the three major bottlenecks. Based on their find outs, they proposed six guidelines for developers on how to set the considered configuration parameters in order to improve performance of Hyperledger-Fabric. In the second phase of their work, they tried to optimize Hyperledger-Fabric, by addressing their find outs, and managed to improve the overall throughput by 16 times. That study provides valuable insights for our framework, because our framework deploys blockchain features of Hyperledger-Fabric to MEC components. We benefited from the guidelines published in the work by Thakkar et al, when configuring the Hyperledger-Fabric. As one of the guidelines recommends to use lower block size to minimize the latencies. Therefore, we configured the block size to be 10. Although, the authors state that the transaction throughput increases with bigger block size. The other guideline is saying *"To achieve a high performance, define policies with a fewer number of sub-policies and signatures."* [55]. Therefore, the endorsement policy we used is composed only of a single sub-policy.

2.3.2 Blockchain in V2X Communication

The authors of [2] and [38] state that VANET is vulnerable to attacks, which might be very critical in connected vehicles domain. As stated by the authors of [37], [20], [58], [59] blockchain could be used to address those vulnerabilities, because of its decentralized and tamper-proof nature. Furthermore, blockchain's smart contracts could be utilized to automate various processes in the domain.
In the remainder of this section we elaborate on the studies concerning the blockchain in V2X communication. The authors of [37] proposed CreditCoin, a *"privacy-preserving incentive announcement network based on blockchain"*. Their network should be able to address the issues of vehicle networks: difficulty of forwarding a reliable announcement without revealing the identities of users; lack of user's motivation to forward announcements. They have run extensive simulations and benchmarks to analyze performance of the CreditCoin. However, their benchmarks are specific to V2X systems, while our framework is intended for use in any MEC scenario.
Dorri et al [20] presented a blockchain-based architecture to protect data originated from interconnected smart vehicles to avoid location tracking or vehicle hijacking. They evaluate they architecture by illustrating scenarios (wireless remote software updates and dynamic vehicle insurance fees), which could benefit from their solution and elaborate on privacy and security issues addressed by their architecture. In our work we reuse the dynamic vehicle insurance fees scenario. However, their evaluation doesn't include any generic benchmarks, which could be reused by our framework.
Xiong et al [58] presented an architecture in blockchain-based MEC, while they considered VANET domain as well. They propose to offload the mining task from vehicles and RSUs to the resource richer edge computing nodes. The authors explain that interactions between vehicles, RSU and edge nodes can be modeled as market activities, such that the edge nodes sell data and computing power. Nevertheless, the authors considered only blockchains utilizing PoW to achieve consensus, they haven't evaluated any other protocols. While our goal is to support blockchains, using other consensus algorithms as well.
The authors of [59] proposed a decentralized trust management scheme for vehicular networks based on blockchain, to address security drawbacks associated with vehicular networks. By following their approach, the involved parties (vehicles) validate the received messages. The results of the validation is used to generate a trust rating of the vehicle, which sent the message. The trust rating is subsequently uploaded to RSU and persisted to the blockchain, hosted on all RSUs. They combined PoW and Proof-of-Stake consensus mechanism to speed up the process of finding nonce for the hash function (PoW). Their experiments have shown that their proposed approach is efficient in practical vehicular networks. The authors performed a simulation and benchmarked their experiments against a set of performance metrics. These metrics are interesting for our work, we reuse the idea of their *"transmission latency of messages"* metric in our framework.

2.3.3 Knowledge Services for Experiments

To the best of our knowledge there are currently no solutions, which we could reuse for the Experiments Knowledge Service. The authors of [37], [20], [58], [59] didn't provide any information on what services have been utilized to store their experiments.

2.4 Summary

As we found out in this chapter, there are numerous works and studies centered around blockchain in MEC. Many researchers study a potential of blockchain in the V2X environment. Various blockchain benchmarking frameworks have been developed, which could be utilized by developers when creating an application based on blockchain. The related frameworks share various aspects, which include supported blockchain implementations or metrics of quality, with our framework. However, any of the related frameworks doesn't focus on benchmarking blockchain interactions for blockchain-based applications in MEC.

CHAPTER 3

Scenarios and Requirements

3.1 Overview

In this chapter we firstly introduce a set of scenarios in MEC, which we consider in this thesis. We look on those scenarios from a developer's point of view, who wants to create blockchain-based applications for those scenarios.
The developer faces challenges, explained in Section 1.2, when developing the applications. We discuss quality metrics, which we contemplate as most relevant to the developer. Furthermore, we identify MEC components, services and stakeholders, which are relevant to our scenarios. We show that there are various patterns of interaction between the identified components. It's not always the case that all components participate in every interaction. We emphasize the importance of the interactions for the developer. To the end, we explain how we group blockchain operations (identified in Section 2.2.1) into blockchain features, such that a feature is represented by an executable blockchain artefact. The blockchain artefacts can be deployed to the MEC components by the developer.

3.2 Focused Scenarios in V2X Domain

In this section we will briefly introduce a set of scenarios within the V2X domain. In Section 3.4 we identify interactions among MEC components in those scenarios. These form a basis for our benchmarks later in the thesis.

- Safe lane change [27] - a vehicle wants to overtake another slow-driving vehicle. Thus it has to determine, whether there isn't any vehicle traveling in opposite direction, which might potentially endanger the overtake maneuver.
- Obstacle on the road warning [27] - a vehicle spots an obstacle on the road and warns other vehicles nearby about the obstacle.

- Toll payments [61] - a vehicle pays toll automatically when exiting the highway.
- Car insurance ([20], [36]) - a vehicle performs an automatic claim of payments in case of an incident.

3.2.1 Components, Services and Stakeholders

The developer needs to know the components of MEC, because instances of the blockchain-based application and blockchain features are going to be deployed to those components. In the following list we present the identified components.

- **vehicle**, **RSU** - discussed in Section 2.2.3.
- **edge node** - communicates with RSUs and vehicles and provides internet connection to them.
- **cloud node** - communicates with the edge nodes and provides storage capabilities.

Resources are required for the components to build a real MEC infrastructure. A vehicle equipped with an embedded powerful computer for the vehicle component, a single board computer with networking capabilities for the RSU and powerful servers for the edge and cloud nodes. We discuss more on the resources what we used for our benchmarks in Section 4.2.2.

We assume the application needs functionalities provided by the following software services. The assumed services are deployed to the identified MEC components as well.

- **V2X Platform-as-a-Service (V2X PaaS)** - This service wraps the functionalities and capabilities required for the V2X communication to work. These include reading data from vehicle's sensors, obtaining driving properties in a vehicle, VANET implementation in the involved components etc. Since we are not interested in the exact functionality of the service, we consider it as a black box, which provides interfaces for other services.
- **Message Broker, Pub/Sub Broker** - Serving as a reliable communication service between software components.
- **Streaming/Processing** - Streaming/Processing of the data between services.

Figure 3.1 sketches a sample deployment of the discussed services, application and some blockchain features to the identified components. In the depiction, topology of the application consists of interacting vehicle, RSU, edge and cloud nodes. An instance of the application and blockchain features are deployed to every involved component.

Additionally these stakeholders are involved in the scenarios:

- **Developer** - is a key stakeholder in our scenarios. We look on the scenarios from the developer's point of view, because our key focus is to help him/her to deal with blockchain in the scenarios.
  The developer aims to implement an application, to address a scenario. The application utilizes blockchain and has to achieve certain quality metrics, depending on the requirements.
- **Vehicle owner** - might own a cryptocurrency account and an issued smart contract with other stakeholders.
- **Insurance company** - collects evidence information about an incident from other stakeholders, issues contracts for various car owners and has an account to repay the incident's costs to their customers.
- **Repair shop** - can provide additional material to the insurance company to claim a repayment.
- **Highway management** - tracks vehicles entering and leaving a highway and issues toll payments for the drivers.
- **Cloud storage** - provides capabilities to store relevant material regarding the payments, or incident evidence.

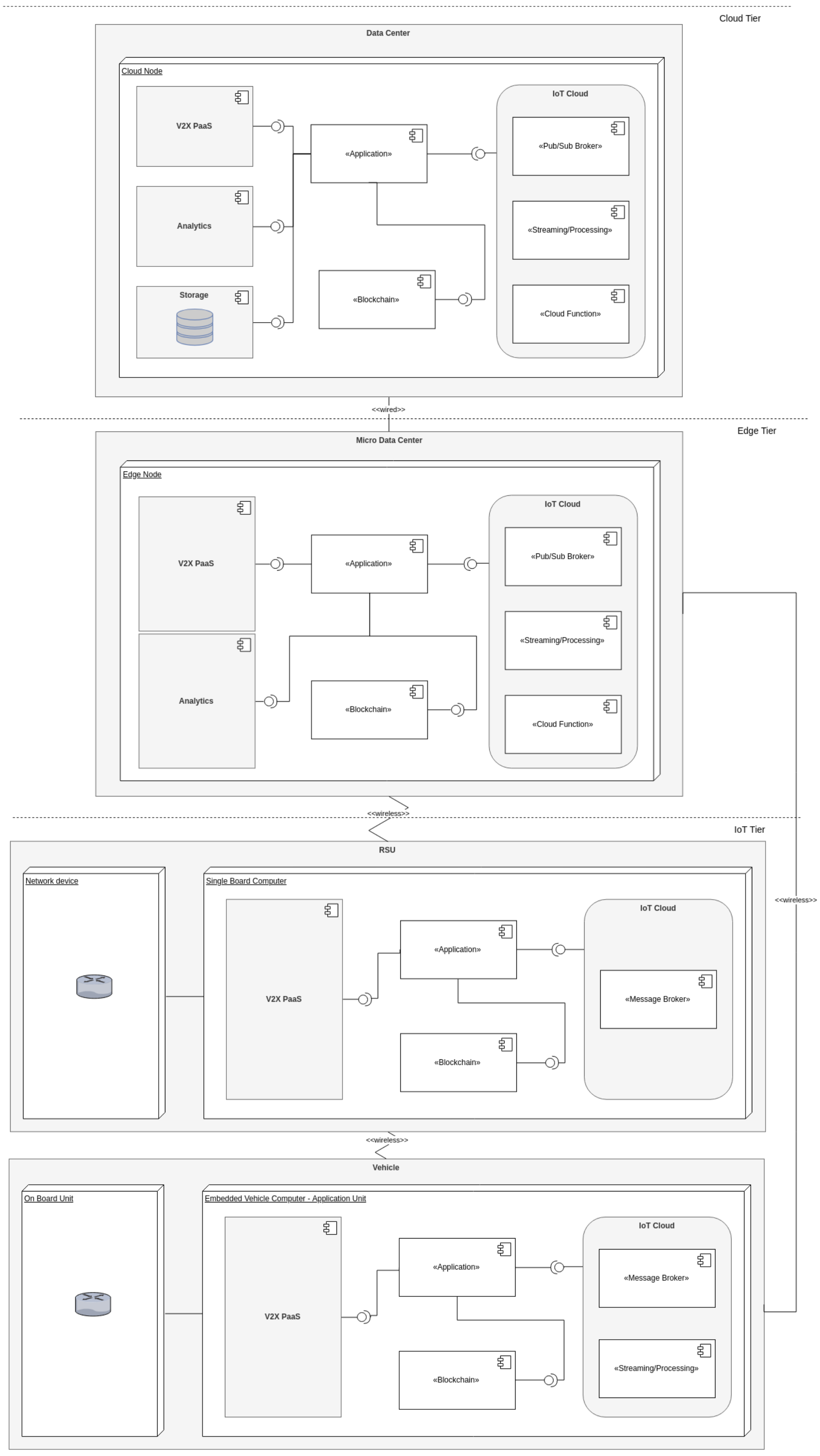


Figure 3.1: A sample deployment of the application, blockchain and services to the components of MEC

3.3 Quality Metrics

There are many quality metrics [34] [10] [44], which have to be considered by developers when developing blockchain-based applications. However, in this section we present metrics of quality, which we consider as most relevant to the developers implementing an application based on blockchain in MEC. The goal of those metrics is to help the developers to address the challenges (identified in Section 1.2) their face when developing the application.

3.3.1 Transaction Acceptance Rate

The first metric we consider is the transaction acceptance rate. It's expressed via the ratio of accepted transactions to the ones which have been submitted to blockchain. We consider this metric, because we help the developers by benchmarking blockchain interactions between MEC components and this metric provides information whether an interaction took place (a transaction was accepted) or not (a transaction was rejected). Furthermore, this metric provides insights for the developers towards the reliability of the application.

In blockchain systems we face throughput and bandwidth issues [34]. These extend even more in MEC environment, since IoT devices are producing a lot of data, which should be exchanged over blockchain in form of transactions.

3.3.2 Synchronization State

As explained in the previous section, some transactions might be non-accepted by blockchain. If there are too many non-accepted transactions in a particular node of blockchain's topology, then the node might be removed from blockchain's topology and consequently lost synchronization state. Therefore, we use this metric, which is measured by the number of blockchain nodes, which have been removed from a blockchain's topology during a period of time. For the purpose of this thesis the period of time is a time-span of a simulation performed by our benchmark framework, more on that in Section 4.3.

This metric provides further insights for the developers towards reliability of the application as well, while the reliability is critical in the domain of V2X [27].

3.3.3 Transaction Acceptance Time

This metric is measured by the time it takes to accept a transaction by blockchain. Since we want to help the developers by benchmarking blockchain interactions among MEC components. We assume this metric is relevant to the developers, because it states how long does an interaction take.

In blockchain enabled systems we witness latency issues [34], since achieving consensus is usually a time-consuming operation. Those issues are relevant to the MEC, because there are computationally restricted IoT devices, on which it might take a long time to run the consensus algorithms. This metrics provides information regarding the performance of the applications for the developers.

3.3.4 Scalability

Scalability is one of issues, we face in blockchain systems [34]. In MEC environment that issue gained even bigger relevance, since the IoT topologies can be large, and change frequently. This metric is defined by Dinh et al [15] and is measured as changes in transaction acceptance rate and time and in synchronization state, when increasing the number of nodes in the application's topology.

3.3.5 Infrastructure Resources Utilization

In this metric we observe hardware utilization of the application's underlying infrastructure's resources. That provides relevant information to the developers about whether an infrastructure suits the requirements of their applications, as well what are the costs of such infrastructure. This metric is measured by hardware utilization of the resources. The *% utilization of a CPU core* and amount of consumed *RAM memory in MBs* are measured. Beside the hardware utilization, the developers would like to consider *quality of network connection*, between the nodes of application's topology, in this metric. Since it may affect the costs of the infrastructure as well.

3.4 Components Interaction Scenarios and Requirements

In this section we look on the scenarios, introduced in Section 3.2, from a developer's point of view. When the developer is developing a blockchain-based application for the scenarios then he/she finds out that different possible patterns of interaction among the MEC components could arise. The instances of the application are running in the MEC components, which participate in an interaction, and are exchanging data over blockchain among each other. It's very important for the developer to identify those interactions, because each interaction results in a different topology of the application. The developer needs to know how does a blockchain interaction between the instances of the application perform in the view of the quality metrics. To determine that the developer can utilize the benchmarking framework, developed in this thesis. As already noted in Section 1.2, the developer has to deal with the challenges listed below when developing the application.
We deploy services, identified in Section 3.2.1, instances of application and blockchain features to the interacting MEC components. Further we identify participating stakeholders and emphasize a list of requirements in a form of introduced quality metrics for every scenario.

- Which blockchain implementation to be used for a scenario?

- What interaction does perform the best, in the view of selected quality metrics, for a particular scenario ?
- Which blockchain features should be deployed to which nodes of a topology, representing an interaction, in order to achieve the desired functionality and certain quality metrics for the topology?
- What are the requirements in terms of hardware for the application's underlying infrastructure?

3.4.1 Vehicle to Vehicle Interaction

Assume a developer wants to implement a blockchain-based application to address the lane change scenario [27]. The developer finds out that the scenario involves vehicles interacting between each other. No further MEC components are involved in the scenario. Utilizing blockchain in the scenario enriches safety by forcing integrity of exchanged messages, because of tamper-proof capability of blockchain. This means that the vehicles may continue using VANET to exchange messages between vehicles, but the message's payload (driving data) is hashed and the hash is stored in blockchain.
The diagram on Figure 3.2 sketches this interaction. The topology representing this interaction consists of two vehicles (`vehicle1` and `vehicle2`). Both deploy some blockchain features, run an application to address the lane change scenario and utilizes the services described in Section 3.2.1.
The Streaming/Processing service, running in `vehicle1`, pulls driving data from V2X PaaS and filters the events, which are related to an overtake operation. These are further streamed to the application. The application connects to `vehicle2`, to determine if it's safe to perform overtaking maneuver. To find out that the application in `vehicle2` asks it's V2X PaaS to spot the surroundings by its sensors (e.g. return the distance to and velocity of a vehicle traveling in opposite direction). The returned sensor's data are being passed back to `vehicle1` and the hash of the data is stored to blockchain asynchronously. Then the application in `vehicle1` uses blockchcain to verify the integrity of the returned sensor's data by comparing its hash with the one stored in blockchain. As soon as the data's integrity has been successfully verified, the application can answer to V2X PaaS whether it is/isn't safe to overtake.
In this case, the developer has to decide what blockchain features should be deployed within the blockchain nodes in the vehicles. Since this is a safety application, the developer is focused to find a deployment, which maximizes the transaction acceptance rate and synchronization state, while minimizing the transaction acceptance times of blockchain. Please note that when verifying the data's integrity, increased waiting times are occurring in the application of `vehicle1`. Because it has to be waited until the transaction (created in `vehicle2`) is accepted and visible for `vehicle1`. The goal of the developer is to find such deployment, which minimizes those waiting times.

Interaction id	1
Pattern	vehicle - vehicle
Stakeholders	Developer
Requirements	high tx acceptance rate and synchronization state, short tx acceptance time
Description	A V2V communication application, two or more vehicles exchanging driving data.
Scenario Example	lane change scenario

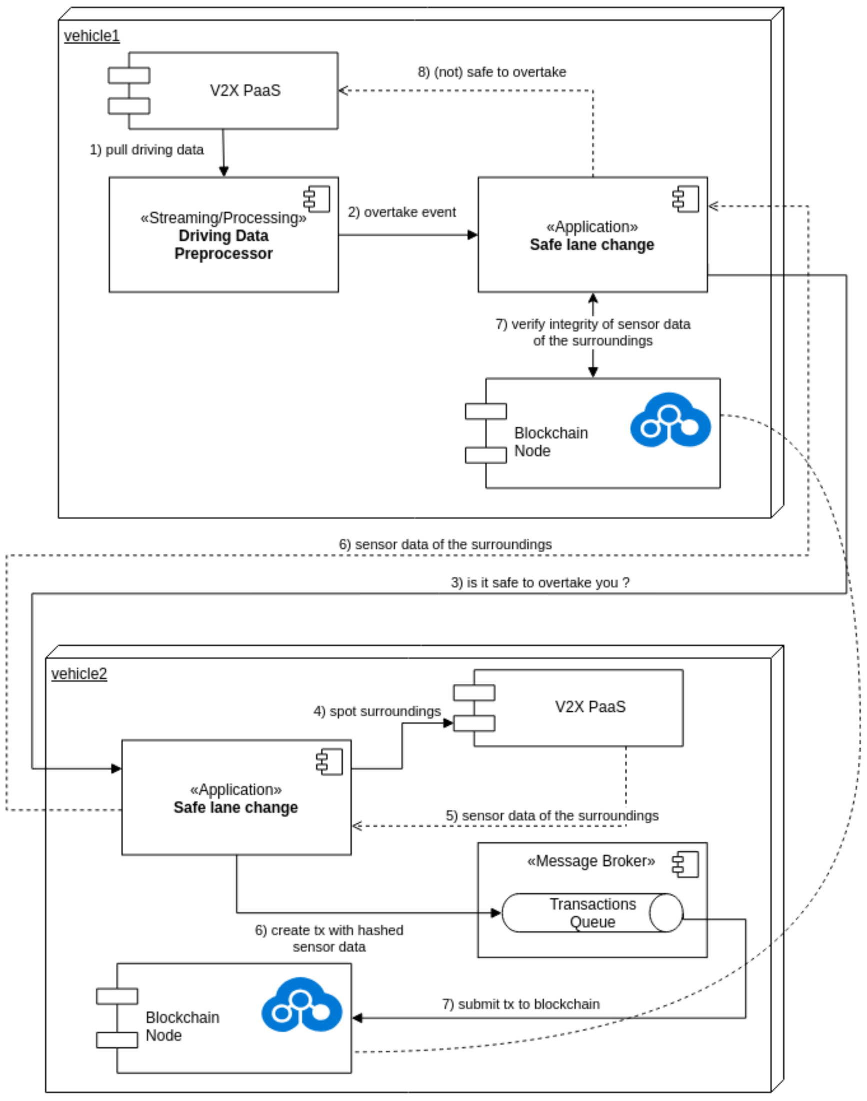

vehicle1
V2X PaaS
8) (not) safe to overtake
1) pull driving data
«Streaming/Processing» Driving Data Preprocessor
2) overtake event
«Application» Safe lane change
7) verify integrity of sensor data of the surroundings
Blockchain Node
6) sensor data of the surroundings
3) is it safe to overtake you ?
vehicle2
4) spot surroundings
V2X PaaS
«Application» Safe lane change
5) sensor data of the surroundings
«Message Broker»
Transactions Queue
6) create tx with hashed sensor data
Blockchain Node
7) submit tx to blockchain

Figure 3.2: Lane change scenario in blockchain-enabled V2V

3.4.2 Vehicle to RSU Interaction

The first interaction we consider for the obstacle on the road scenario [27] includes vehicles and RSUs. The application for the scenario should warn the drivers about an obstacle (an accident or a fallen tree branch) on the road. In this interaction a RSU is involved, beside vehicles. Because we assume a vehicle wants to pass a warning to another, but cannot do it directly, because of the distance between the vehicles. Thus a RSU is utilized for a multihop communication between the vehicle.
In this scenario blockchain is used to verify integrity of messages being passed between vehicles and RSUs. That is done by storing hashes of the messages in blockchain. It enriches VANET communication by providing more security.
Figure 3.3 illustrates the scenario's interaction on service level. The depiction shows two vehicles (`vehicle1` and `vehicle2`) and a RSU `rsu1`. The Streaming/Processing filters data from sensors of `vehicle1`, which may observe an obstacle on the road. Based on that data, a warning is created in the application. It's is published to a queue of RSU's message broker. Simultaneously, the transaction wrapping hashed warning is created and submitted to blockchain asynchronously. When another vehicle (`vehicle2`) enters the communication range of RSU, and subscribes to its queue, it receives the warning. The integrity of the warning is verified via its hash stored in blockchain. If the integrity's correct, then the vehicle takes an appropriate action (slow down or change the line).
In this interaction the developer has to find out, what blockchain features to deploy to the blockchain nodes in vehicles and RSU. As it was the case in previous interaction, waiting times (when verifying integrity of the warning) might be experienced in the application in `vehicle2`. While the developer's intention is to develop system, which guarantees the message's integrity in shortest possible time. Thus the requirements are to maximize the transaction acceptance rate and synchronization state, while minimizing the acceptance time.

Interaction id	2
Pattern	vehicle - RSU - vehicle
Stakeholders	Developer
Requirements	high tx acceptance rate and synchronization state, short tx acceptance time
Description	A multi-hop communication between two or more vehicles over a RSU.
Scenario Example	An obstacle on the road warning scenario

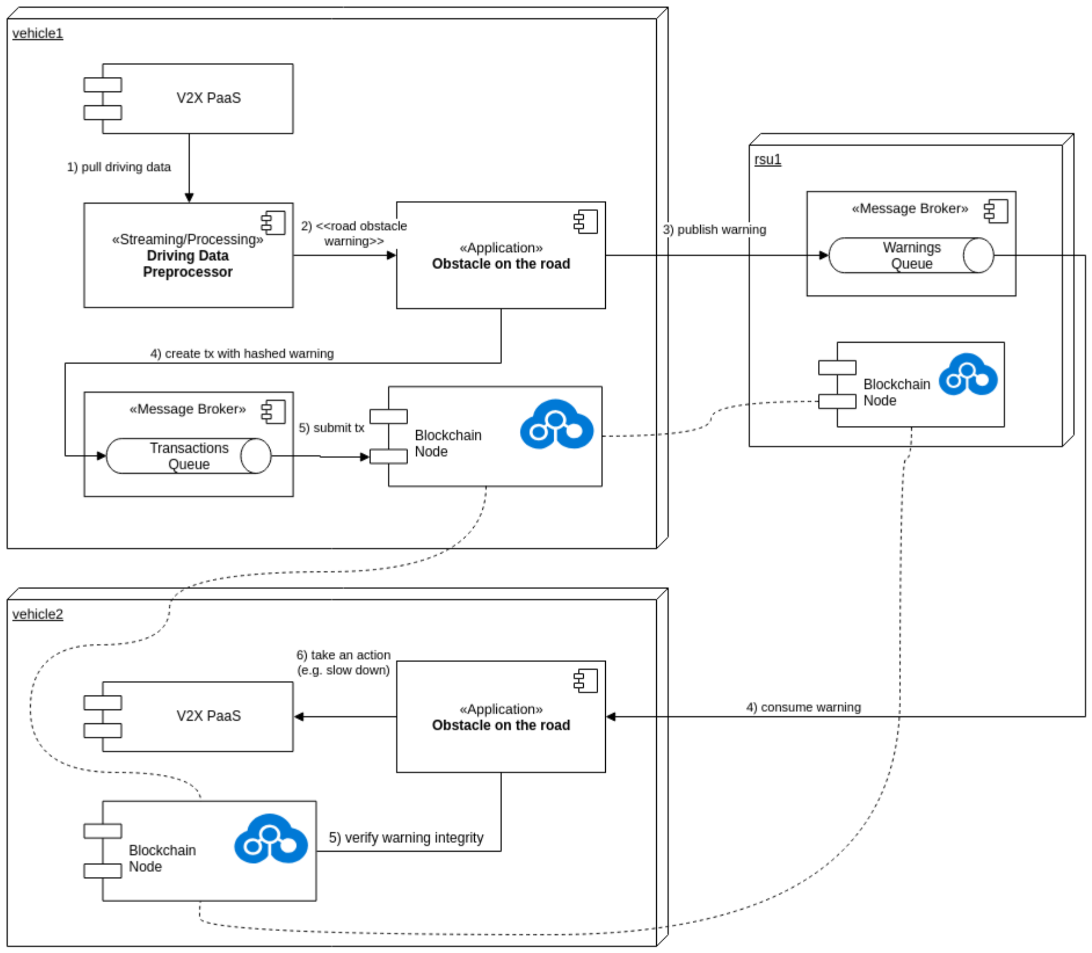

vehicle1
V2X PaaS
1) pull driving data
«Streaming/Processing» Driving Data Preprocessor
2) <<road obstacle warning>>
«Application» Obstacle on the road
4) create tx with hashed warning
«Message Broker»
Transactions Queue
5) submit tx
Blockchain Node
3) publish warning
rsu1
«Message Broker»
Warnings Queue
Blockchain Node
vehicle2
6) take an action (e.g. slow down)
V2X PaaS
«Application» Obstacle on the road
4) consume warning
Blockchain Node
5) verify warning integrity

Figure 3.3: An obstacle warning scenario in blockchain-enabled V2X over RSU

3.4.3 Vehicle to Edge Interaction

The second interaction, we assume for the obstacle on the road scenario [27], is using edge node instead of RSU. That might occur, if there isn't a working RSU near the vehicle. Therefore, it has to connect and communicate over the edge node, which is computational more powerful than a RSU. The involved services, message exchange, role of blockchain and requirements in this case are the same as in the previous interaction (Section 3.4.2).

Interaction id	3
Pattern	vehicle - edge node - vehicle
Stakeholders	Developer
Requirements	high tx acceptance rate and synchronization state, short tx acceptance time
Description	Communication between vehicles, over an edge node as middleware, since there is no RSU nearby.
Scenario Example	An obstacle on the road scenario

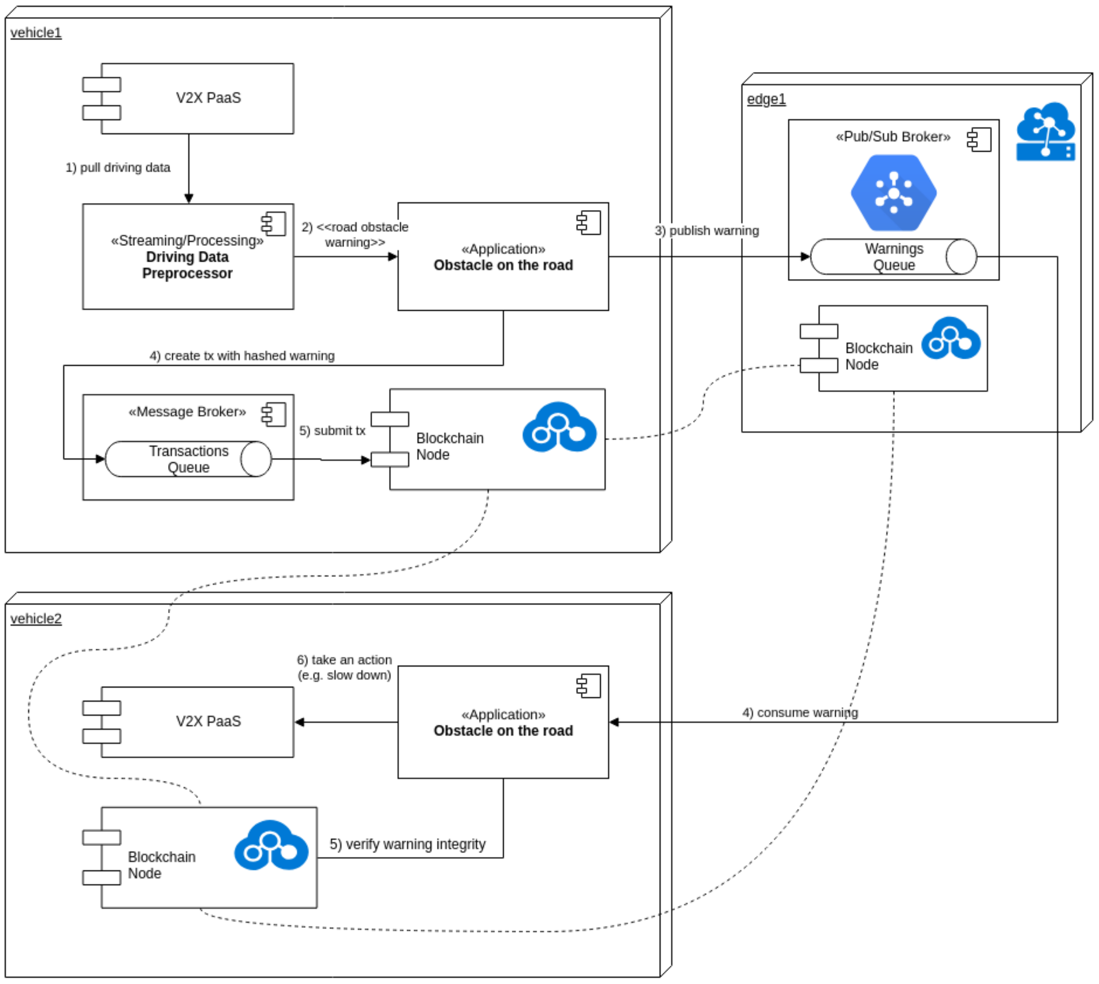

vehicle1
V2X PaaS
1) pull driving data
«Streaming/Processing» Driving Data Preprocessor
2) <<road obstacle warning>>
«Application» Obstacle on the road
3) publish warning
4) create tx with hashed warning
«Message Broker»
Transactions Queue
5) submit tx
Blockchain Node
edge1
«Pub/Sub Broker»
Warnings Queue
Blockchain Node
vehicle2
6) take an action (e.g. slow down)
V2X PaaS
«Application» Obstacle on the road
4) consume warning
Blockchain Node
5) verify warning integrity

Figure 3.4: An obstacle warning scenario in blockchain-enabled V2X over edge node

3.4.4 Vehicle to RSU and Edge Interaction

This is the third interaction, which is considered for the application, dealing with obstacle on the road scenario [27]. In this interaction both RSU and edge node are involved beside the vehicles. While the messages flow from one vehicle to another, through a RSU, and an edge node. The edge node enriches the topology by providing more computational capabilities. This may bring benefits for blockchain's requirements. Thus it's interesting, from the developer's point of view, to benchmark this interaction against the one utilizing RSU (Section 3.4.2), or edge node (Section 3.4.3).

Interaction id	4
Pattern	vehicle - RSU - edge node - vehicle
Stakeholders	Developer
Requirements	high tx acceptance rate and synchronization state, short tx acceptance time
Description	Communication between vehicles, over a RSU, while utilizing edge node's capabilities.
Scenario Example	An obstacle on the road scenario

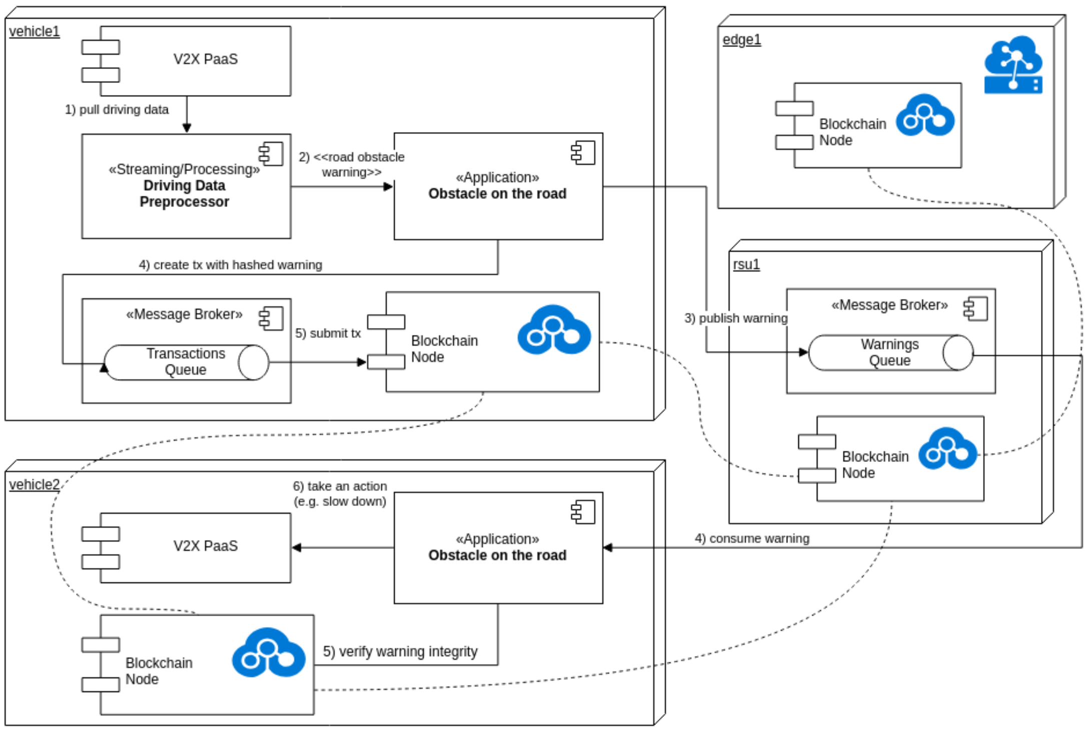

vehicle1
V2X PaaS
1) pull driving data
«Streaming/Processing» Driving Data Preprocessor
2) <<road obstacle warning>>
«Application» Obstacle on the road
4) create tx with hashed warning
«Message Broker»
Transactions Queue
5) submit tx
Blockchain Node
edge1
Blockchain Node
rsu1
3) publish warning
«Message Broker»
Warnings Queue
Blockchain Node
vehicle2
6) take an action (e.g. slow down)
V2X PaaS
«Application» Obstacle on the road
4) consume warning
Blockchain Node
5) verify warning integrity

Figure 3.5: An obstacle warning scenario in blockchain-enabled V2X over RSU and edge node

3.4.5 Vehicle to Edge and Cloud Interaction

Assume a developer develops a blockchain-based application for automatic costs reclaiming in case of an incident [20] [61]. He/she considers an interaction among vehicles, edge nodes and cloud nodes components. Although, in this case the message exchange in the interaction doesn't happen synchronously and in a sequence, as it was the case in previous interactions. But rather we have a set of stakeholders collaborating via the components.
Figure 3.6 illustrates the scenario with the involved services and stakeholders. The figure shows one instance of vehicle (`vehicle1`), edge node (`edge1`) and cloud node(`cloud`). There is a smart contract running in blockchain, which should trigger an action to claim costs in case of an incident. The responsibility of the application in the vehicle is to obtain relevant data from the car's systems about the incident. These data are submitted to blockchain directly. In some cases these data aren't enough to trigger the blockchain's smart contract. Therefore, other stakeholders like repair shop in our scenario have to provide additional information and submit it to the blockchain. As soon as there is enough data as a proof of the incident then the smart contract is executed and credits are transferred from crypto-currency account of the insurance company to the driver's crypto-currency account. Issuing new smart contracts and management of the existing ones is under the responsibility of the insurance company.
For the topology representing this interaction, the developer has to decide what blockchain features to deploy to the blockchain node running in vehicles, edge node and cloud node. The goal of the developer is to propose a blockchain-based architecture, which doesn't require expensive infrastructure while ensures a sufficient rate of accepted transaction and state of synchronization.

Interaction id	5
Pattern	vehicle - edge node - cloud node
Stakeholders	Developer, Driver, Insurance company, Repair shop
Requirements	low infrastructure costs, sufficient transaction acceptance rate and synchronization state
Description	A vehicle needs to communicate with services running in edge nodes and cloud nodes. These cannot be deployed in a RSU, since may require more computational power, than provided by RSU.
Scenario Example	Automatic reclaim of costs in case of an incident

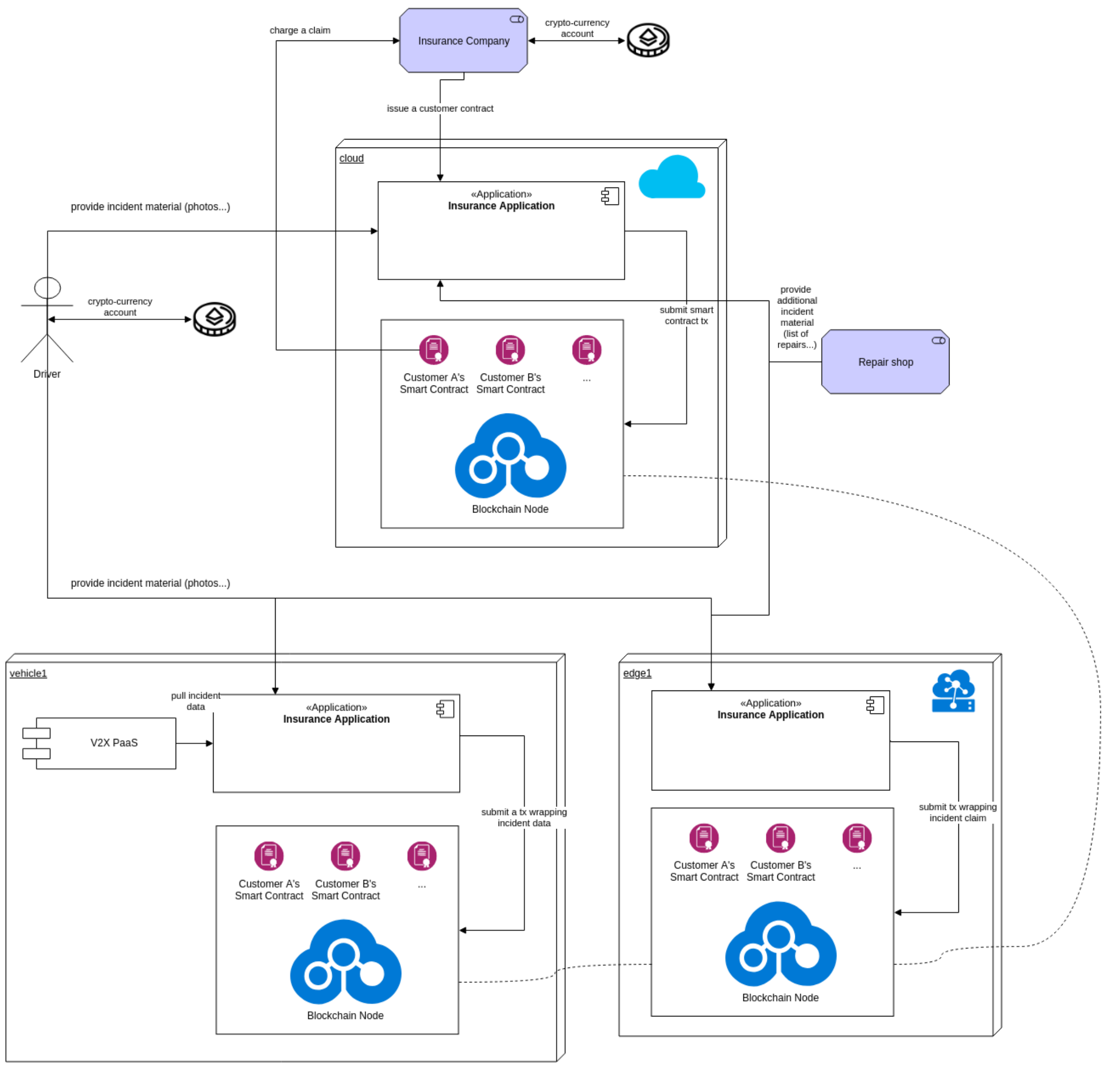

charge a claim
Insurance Company
crypto-currency account
issue a customer contract
cloud
«Application»
Insurance Application
provide incident material (photos...)
crypto-currency account
Driver
submit smart contract tx
provide additional incident material (list of repairs...)
Repair shop
Customer A's Smart Contract
Customer B's Smart Contract
...
Blockchain Node
provide incident material (photos...)
vehicle1
pull incident data
V2X PaaS
«Application»
Insurance Application
submit a tx wrapping incident data
Customer A's Smart Contract
Customer B's Smart Contract
...
Blockchain Node
edge1
«Application»
Insurance Application
submit tx wrapping incident claim
Customer A's Smart Contract
Customer B's Smart Contract
...
Blockchain Node

Figure 3.6: Automatic reclaim of costs in case of an incident in blockchain-enables V2X environment

3.4.6 Vehicle to RSU, Edge and Cloud Interaction

In this case, a developer implements a blockchain-based application to address autonomous toll payments scenario [61]. A message exchange among vehicle, RSU and edge node, while utilizing storage capabilities of the cloud node is considered in the scenario. By utilizing blockchain and smart contract, we could automate the process of toll payments without using the central authority, which is utilized by the state-of-the-art automatic payment systems [3] [54].
A smart contract is implemented, which listens for an event, when a vehicle leaves the highway. The smart contract will trigger an action to make a payment by the driver to

the highway management. Furthermore, some relevant information (e.g. photo, recipe, location and time of the exit) are stored to the cloud storage.
Figure 3.7 sketches the interaction in a blockchain enabled autonomous toll payment scenario. In the first step, there is a RSU, placed on exit of the highway, ordering a vehicle to pay toll. The vehicle connects to the RSU and sends toll data (this include all necessary information required to pay the toll) to RSU's message broker, which forwards the data to the pub/sub broker in edge node. Simultaneously, in the vehicle the hash of the data is calculated and submitted to vehicle's blockchain node. The RSU, edge node and cloud node are other peers of our blockchain network. As the toll data are consumed by the toll payments application in the edge node, blockchain is utilized to verify whether the message hasn't been tampered along the interaction. If the integrity is valid, then the relevant data are stored to the cloud storage, while smart contract executes an action to make payment to the highway management by the user.
In this interaction we assume that the developer's task is to minimize the costs of underlying infrastructure, while a sufficient rate of accepted transactions and state of synchronization should be achieved.

Interaction id	6
Pattern	vehicle - RSU - edge node - cloud node
Stakeholders	Developer, Driver, Highway Management, Cloud Storage
Requirements	low infrastructure costs, sufficient transaction acceptance rate and synchronization state
Description	A vehicle requires services running in the edge node, or utilizes the capabilities of the cloud node, while the communication with those services isn't direct, it flows over a RSU station.
Scenario Example	Toll payments

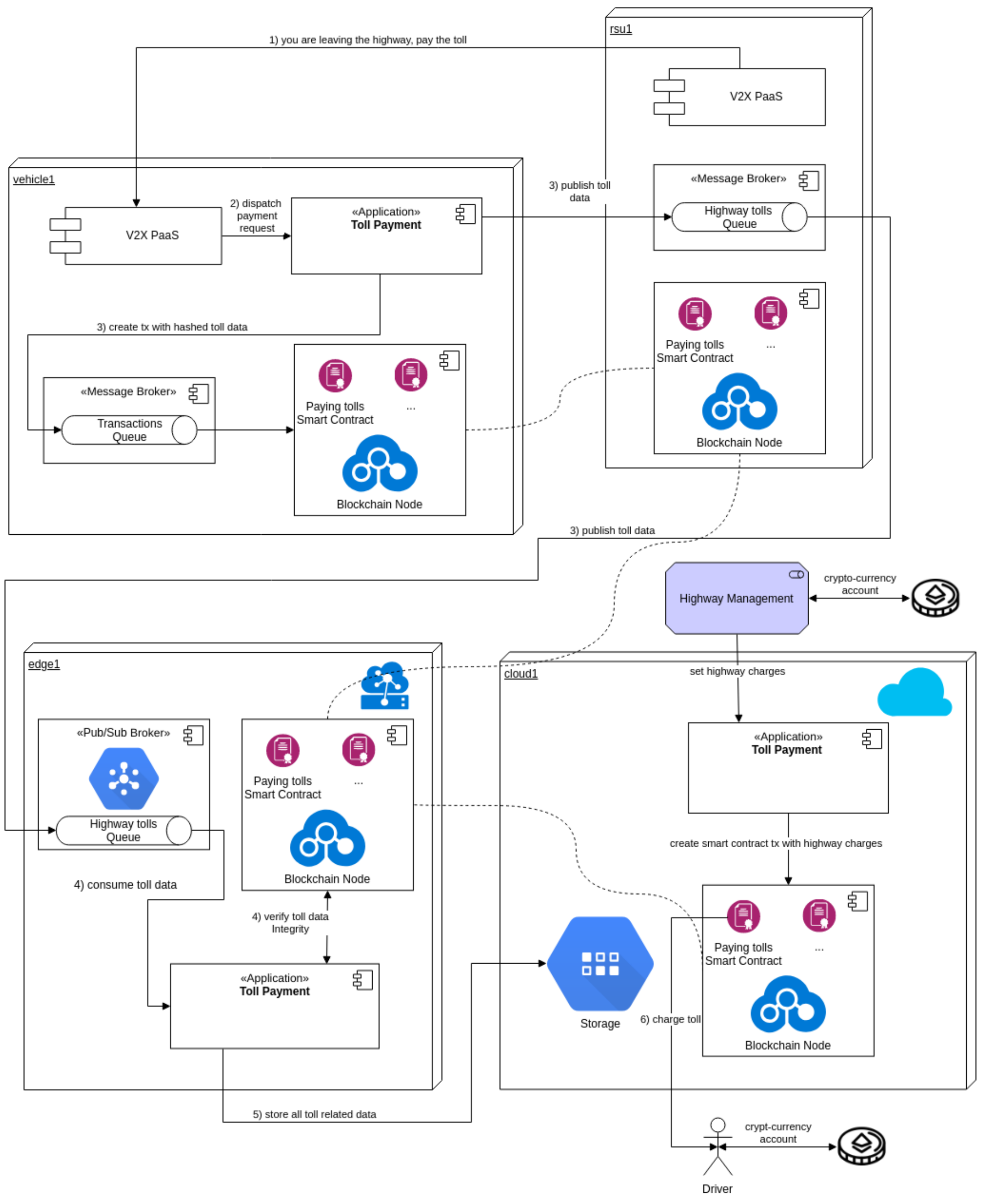


Figure 3.7: Automatic toll payments in blockchain-enabled V2X environment

3.4.7 Summary

Based on our assumptions regarding to the developer's requirements, presented in Section 3.4. We summarize the relevance of the quality metrics on our scenarios in Table 3.2 (Sign +++ stands for the most, + for the least relevant).

Table 3.2: Summary of quality metrics for interactions

	Shortest Tx Acceptance Time	**Highest Tx Acceptance Rate & Highest Sync state**	**Scalability**	**Infrastructure Resource Utilization**
Lane change scenario 3.4.1	++	+++	++	+
Obstacle on the road warning 3.4.2 3.4.3 3.4.4	++	+++	++	+
Insurance scenario 3.4.5	+	++	++	+++
Toll payments 3.4.6	+	++	++	+++

3.5 Blockchain Artefacts

In the preceding sections we identified the main components of MEC and a set of scenarios with corresponding interactions among the MEC components. Blockchain has been used to enable those interactions, but it hasn't been clarified which blockchain features run in the components. The developer has to decide that. To execute the features in the components, we have to create executable artefacts representing the blockchain features. To achieve that we must obtain knowledge about how are those features implemented in the chosen blockchain implementations (refer to Sections 2.2.2.1 and 2.2.2.2).
In Section 2.2.1 we have described blockchain operations, which have to be carried out to enable the interactions. We aim to group those operations into blockchain features. We do the grouping, based on our intention to separate the achieve consensus operation from the other operations. The reason for that is its complexity and is deeper elaborated in Section 2.2.1.
Let's start with grouping the blockchain operations to the features.

- **Creator feature**: We consider the following list of operations within this feature:
  - *Create a transaction*
  - *Sign a transaction*
  - *Submit a transaction*
  - *Verify a transaction*

  - *Accept a block*

- **Consensus feature**: This is an atomic feature composed of the following operation:
  - *Achieve consensus*

Furthermore, we determine how the chosen blockchain implementations implement those features. Therefore, we create a mapping (illustrated in Table 3.3) of the operations included in the features to their corresponding representation in concrete blockchain implementations. Based on the mapping, the developers see implementations of the blockchain features in Ethereum and Hyperledger-Fabric blockchains.
In Section 2.2.1 we explained that there are various types of blockchain nodes and not every operation can be executed by every node (i.e. only a node known as miner can achieve consensus). Based on that, we infer two types of blockchain nodes, such that each node is responsible for running a particular feature. These nodes are marked in bold in Table 3.3. The nodes are:

- *standard node* - responsible for running the creator feature.
- *miner node* - can run consensus feature.

Based on the mapping of operations, we can map the blockchain nodes to their representations in blockchain implementations. The mapping of blockchain nodes is presented in Table 3.4. Deriving from that mapping we conclude that an Ethereum node can be used to execute the creator feature, but only miner node is capable of executing the consensus feature. In Hyperledger-Fabric the peer node is responsible for running the creator feature and orderer node is used to run the consensus feature.
These blockchain nodes represent the executable blockchain artefacts. Please note that in Ethereum, the miner node is simultaneously a standard node. In contrast to Hyperledger-Fabric where the orderer and peer nodes don't share any functionalities regarding to the blockchain operations.
To conclude this section we list the identified blockchain artefacts, which should be deployed by the developer to the MEC components in our scenarios.

- Ethereum: standard node
- Ethereum: miner node
- Hyperledger-Fabric: peer node
- Hyperledger-Fabric: orderer node

Table 3.3: Mapping of blockchain operations

General blockchain	**Hyperledger-Fabric**	**Ethereum**
Create a tx in client	Create a tx proposal in client	Create a tx in client
Sign a tx by client	Sign a tx proposal by client	Sign a tx by client
Submit a tx by client	Send a transaction to the **endorsing peers** by client	Submit and broadcast a tx to the blockchain network by a client
Verify a tx in a **standard blockchain node**	Endorse a transaction by one or more **endorser peers**	Validate a tx by an **Ethereum node**
Reach consensus by a **miner blockchain node**	Order the transactions by **orderer** and add it to the ledger	**Miner node** mines a new block, containing txs.
Accept a block in **standard blockchain node**	**Leading peers** receive the update by the **orderer**	**Ethereum nodes** receive a new block and synchronize their local copy of blockchain

Table 3.4: Mapping of blockchain roles

Generic blockchain	**Hyperledger-Fabric**	**Ethereum**
Standard blockchain node	Peer nodes (endorser, anchor, leader)	Standard node, Miner node
Miner blockchain node	Orderer node	Miner node

3.6 Summary

In this chapter we have shown that, for a single scenario, there might occur different patterns of interaction among MEC components. Further we identified four executable blockchain artefacts, representing blockchain features, which can be deployed to the MEC components. While we left multiple questions opened:

- What interaction does perform the best, in the view of selected quality metrics, for a particular scenario ?
- Which blockchain artefacts should be deployed to which nodes of a topology, representing an interaction, in order to achieve the desired functionality and certain quality metrics for the topology?
- What are the requirements in terms of hardware for the application's underlying infrastructure?

These questions have to be answered by the developer, who creates a blockchain-based application to address a scenario. To help the developer to deal with those questions, we propose a blockchain benchmark framework in the next chapter.

CHAPTER 4

Benchmarks

4.1 Overview

In order to help developers to address the challenges they face, when creating blockchain-based applications in MEC environments, we propose a benchmarking framework in this chapter. The framework can benchmark blockchain interactions among MEC components. The developer creates a benchmark according to the design explained in Section 4.2 and submits it to the framework. We elaborate on the prototype of the framework in Section 4.3. In Section 4.4 we create and benchmark experiments related to our scenarios. We evaluate the results of those experiments. Furthermore, we explain what insights do the results provide to the developers.

4.2 Benchmark Design

4.2.1 Topology

A topology in the context of benchmarks is a graph consisting of MEC's components (identified in Section 3.2.1) presenting nodes of the graph. The edges in the graph represent blockchain interactions among the MEC components. Such that a set of blockchain features (identified in Section 3.5) is deployed to each component.
In this section we aim to propose deployments, which assign a set of the features to each of the MEC components in the topology. To obtain a diversity of the benchmarks, we create multiple different deployments, for each of the interactions (refer to Section 3.4).
Let's formalize the construction of deployments a bit. We group our components to a set $C_{MEC} = \{Cloud, Edge, RSU, vehicle\}$. Furthermore, we define a set $F_{BC} = \{creator, consensus\}$, denoting the blockchain features we would like to deploy. A deployment is then an assignment of a set of elements from F_{BC} to each element from C_{MEC}.

Table 4.1: A deployment of blockchain features for Interaction 1

Interaction id	**Blockchain features deployment**				
	ID	**vehicle**	**RSU**	**Edge**	**Cloud**
1	0	*all*	-	-	-

Table 4.2: A deployment of blockchain features for Interaction 2

Interaction id	**Blockchain features deployment**				
	ID	**vehicle**	**RSU**	**Edge**	**Cloud**
2	0	*creator*	*all*	-	-
	1	*creator*	*consensus*	-	-
	2	*all*	*creator*	-	-
	3	*all*	*consensus*	-	-
	4	*all*	*all*	-	-

Table 4.3: A deployment of blockchain features for Interaction 3

Interaction id	**Blockchain features deployment**				
	ID	**vehicle**	**RSU**	**Edge**	**Cloud**
3	0	*creator*	-	*all*	-
	1	*creator*	-	*consensus*	-
	2	*all*	-	*creator*	-
	3	*all*	-	*consensus*	-
	4	*all*	-	*all*	-

Tables 4.1 to 4.6 propose all deployments, which we assumed might be meaningful for the developers to evaluate, regarding the respective interactions (see Sections from 3.4.1 to 3.4.6). Note we used a keyword *all* to denote a set of $\{creator, consensus\}$). When constructing the deployments, we followed these constraints:

- Always deploy *creator* feature in to the vehicle component, because vehicles are creating data in all considered interactions.
- For each deployment, the union of the features used across the components has to be a super-set of $\{creator, consensus\}$.
- There has to be one deployment for each interaction, which deploys *all* blockchain features to each involved MEC component, while the other deployments assign less features to the components.

Table 4.4: A deployment of blockchain features for Interaction 4

Interaction id	**Blockchain features deployment**				
	ID	**vehicle**	**RSU**	**Edge**	**Cloud**
	0	*creator*	*consensus*	*creator*	-
	1	*creator*	*all*	*creator*	-
	2	*creator*	*creator*	*all*	-
	3	*creator*	*creator*	*consensus*	-
	4	*creator*	*consensus*	*consensus*	-
	5	*creator*	*all*	*consensus*	-
	6	*creator*	*consensus*	*all*	-
	7	*creator*	*all*	*all*	-
4	8	*all*	*all*	*all*	-
	9	*all*	*consensus*	*all*	-
	10	*all*	*creator*	*all*	-
	11	*all*	*creator*	*creator*	-
	12	*all*	*consensus*	*creator*	-
	13	*all*	*all*	*creator*	-
	14	*all*	*creator*	*consensus*	-
	15	*all*	*consensus*	*consensus*	-
	16	*all*	*all*	*consensus*	-

Table 4.5: A deployment of blockchain features for Interaction 5

Interaction id	**Blockchain features deployment**				
	ID	**vehicle**	**RSU**	**Edge**	**Cloud**
	0	*creator*	*consensus*	*creator*	-
	1	*creator*	*all*	*creator*	-
	2	*creator*	*creator*	*all*	-
	3	*creator*	*creator*	*consensus*	-
	4	*creator*	*consensus*	*consensus*	-
	5	*creator*	*all*	*consensus*	-
	6	*creator*	*consensus*	*all*	-
	7	*creator*	*all*	*all*	-
5	8	*all*	*all*	*all*	-
	9	*all*	*consensus*	*all*	-
	10	*all*	*creator*	*all*	-
	11	*all*	*creator*	*creator*	-
	12	*all*	*consensus*	*creator*	-
	13	*all*	*all*	*creator*	-
	14	*all*	*creator*	*consensus*	-
	15	*all*	*consensus*	*consensus*	-
	16	*all*	*all*	*consensus*	-

Table 4.6: A deployment of blockchain features for Interaction 6

Interaction id	**Blockchain features deployment**				
	ID	**vehicle**	**RSU**	**Edge**	**Cloud**
6	0	*creator*	*creator*	*creator*	*consensus*
	1	*creator*	*creator*	*creator*	*all*
	2	*creator*	*creator*	*consensus*	*consensus*
	3	*creator*	*creator*	*consensus*	all
	4	*creator*	*creator*	*consensus*	*creator*
	5	*creator*	*creator*	*all*	*creator*
	6	*creator*	*creator*	*all*	*consensus*
	7	*creator*	*creator*	*all*	*all*
	8	*all*	*creator*	*creator*	*creator*
	9	*all*	*creator*	*creator*	*all*
	10	*all*	*creator*	*consensus*	*consensus*
	11	*all*	*creator*	*consensus*	*creator*
	12	*all*	*creator*	*consensus*	*all*
	13	*all*	*creator*	*all*	*consensus*
	14	*all*	*creator*	*all*	*creator*
	15	*all*	*creator*	*all*	*all*
	16	*all*	*all*	*all*	*all*

4.2.2 Infrastructure Resources

In the context of benchmarks we interpret the infrastructure as resources necessary to build a MEC infrastructure (see Section 3.2.1) and quality of the network connection between nodes of a topology. In Section 3.2.1 we identified MEC components and the resources required to build a real MEC infrastructure. We don't have a real vehicle, RSU and edge node. Therefore, we need to emulate the resources representing those components. However, we do have access to cloud services. The principle of emulating the unavailable resources and combining them with real resources is called symbiosis test specification, sketched on Figure 4.1.

The unavailable resources are emulated by utilizing containers or virtual machine (VM)s. Such that those VMs/containers have to meet certain requirements in terms of hardware configurations to be able to substitute the real resources. Those requirements are listed in Table 4.7. In the list below we present how we emulate the resources.

- *vehicle*: To emulate a single vehicle, we require multiple software components. For example *V2X Communication Emulator* to emulate the V2X scenarios, *Hyperledger-Fabric peer node* and/or *Hyperledger-Fabric orderer node* for blockchain artefacts and possibly other services like *Kafka* [25], *Zookeeper* [26], etc., depending on the deployment (those software components are explained in Section 4.3). All those software components can be executed in a single VM. In that case one vehicle is emulated by that VM and the hardware of the VM is configured according to the Table 4.7. However, there might be a case, when N instances of those services, would be deployed to a single VM. That would mean that the VM emulates N vehicles and the hardware of the VM is N-multiple of a configuration from the Table 4.7. We assume the developers want to benchmark different hardware configurations of emulated vehicle resource. Therefore, we propose light, medium and big configuration in the Table 4.7.

- *RSU*: To emulate a RSU we deploy an emulator container, blockchain artefacts and eventually other required services like *Kafka*, *Zookeeper*, etc. to a VM. In Section 4.3 we discuss further information about those services and the emulator container. However, we utilize the same principle as in the case of vehicle, when we aim to emulate multiple RSUs in a single VM.
- *edge node*: For emulating the edge node we utilize exactly the same principles as in the case of vehicles or RSUs.
- *cloud node*: We don't need to emulate the cloud node, since we have access to real cloud services.

Figure 4.2 illustrates and example of emulated MEC environment. We elaborate how our benchmark framework creates and deploys the emulated resources in Section 4.3.2.

Furthermore, we emulate different network qualities (listed in Table 4.8) by constraining the latency and bandwidth of a network interface, utilized for the communication. We evaluate the qualities of the network within the benchmarks, since we assume it might provide valuable insights for the developers. In the V2X communication scenario there might be a case when vehicle is moving in the city, having a reliable internet connection or driving in the countryside with a weak low-quality connection. Referring back to the interactions in Section 3.4 we can identify all connections between the MEC components. For the purpose of the benchmarks we assume that the connection between cloud and edge nodes is wired and thus always reliable. But all other connections (edge node to vehicle, edge node to RSU, RSU to vehicle and vehicle to vehicle) are wireless and could be low-quality under some circumstances.

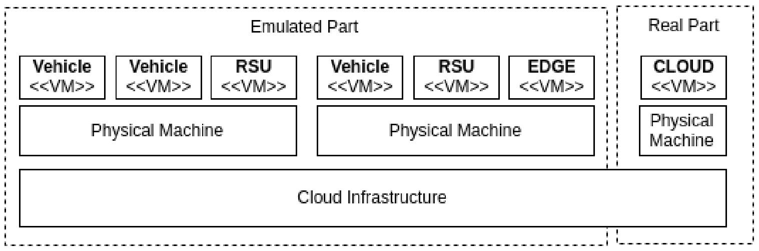


Figure 4.1: Symbiosis Test Specification

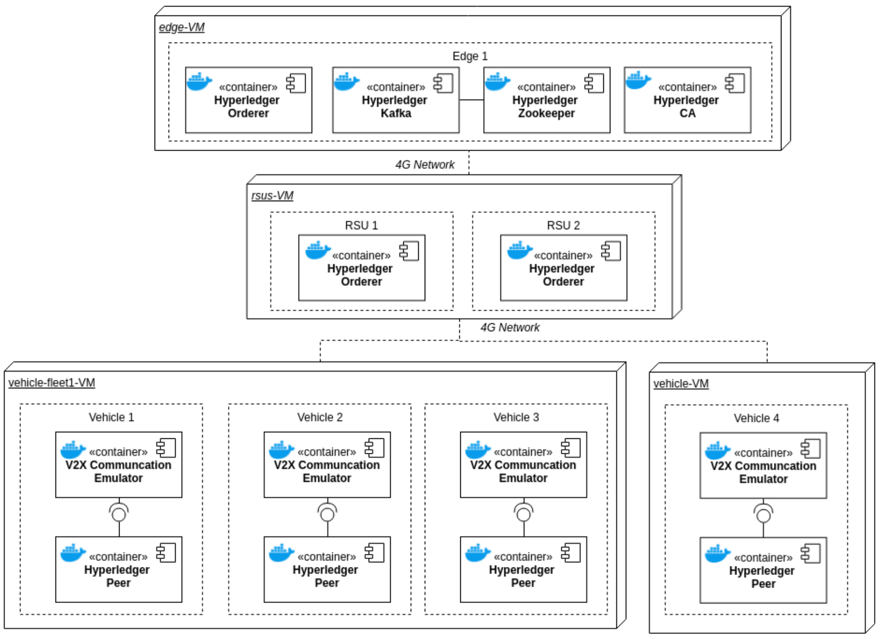

edge-VM
Edge 1
«container» Hyperledger Orderer
«container» Hyperledger Kafka
«container» Hyperledger Zookeeper
«container» Hyperledger CA
4G Network
rsus-VM
RSU 1
«container» Hyperledger Orderer
RSU 2
«container» Hyperledger Orderer
4G Network
vehicle-fleet1-VM
Vehicle 1
«container» V2X Communcation Emulator
«container» Hyperledger Peer
Vehicle 2
«container» V2X Communcation Emulator
«container» Hyperledger Peer
Vehicle 3
«container» V2X Communcation Emulator
«container» Hyperledger Peer
vehicle-VM
Vehicle 4
«container» V2X Communcation Emulator
«container» Hyperledger Peer

Figure 4.2: Example deployment of emulated MEC infrastructure

Table 4.7: The hardware configuration of resources (emulated and real)

Component	Configuration	CPU	RAM	Storage	OS
Cloud		Intel Xeon E5 Sandy Bridge 2.6GHz 4vCPU	16 GB	60 GB SSD	Ubuntu 18.04
Edge		Intel Xeon E5 Sandy Bridge 2.6GHz 4vCPU	16 GB	60 GB SSD	Ubuntu 18.04
RSU		Intel Xeon E5 Sandy Bridge 2.6GHz 1vCPU	2 GB	16 GB SSD	Ubuntu 18.04
vehicle	light	Intel Xeon E5 Sandy Bridge 2.6GHz 1vCPU	2 GB	20 GB SSD	Ubuntu 18.04
	medium	Intel Xeon E5 Sandy Bridge 2.6GHz 2vCPU	4 GB	20 GB SSD	Ubuntu 18.04
	big	Intel Xeon E5 Sandy Bridge 2.6GHz 4vCPU	8 GB	20 GB SSD	Ubuntu 18.04

Table 4.8: Network configurations

Network	Latency	Bandwidth
3G	200ms	1000kbps
4G	100ms	10000kbps
5G	5ms	54mbps

4.3 Benchmark Framework

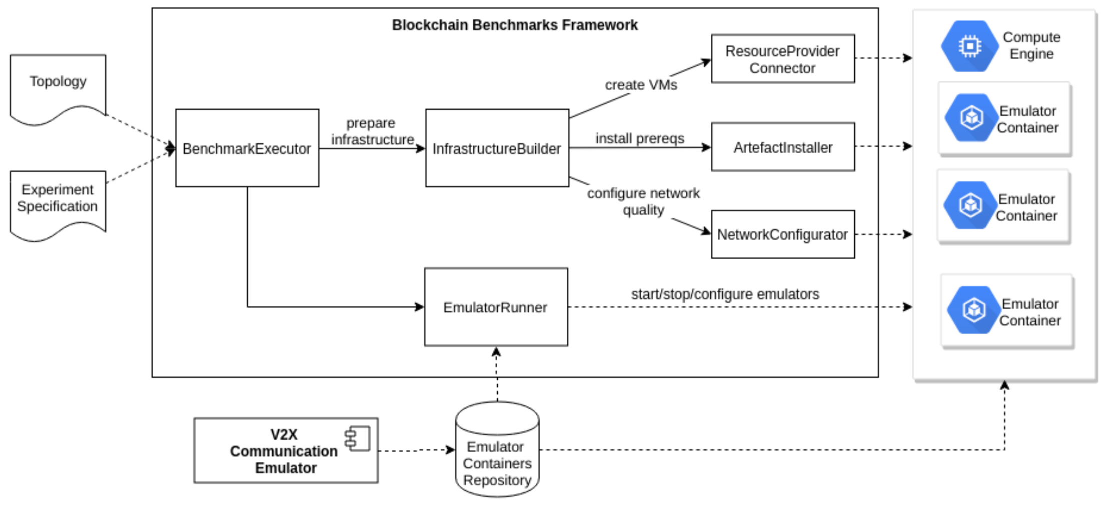


Figure 4.3: Component diagram of the framework

A framework to generate and execute the benchmarks has been developed within the scope of this thesis. Its architecture is illustrated in Figure 4.3. Figure 4.4 provides a class view of the framework. A basic workflow of the framework is following: *BenchmarksExecutor* loads a specification of an experiment and a topology of benchmarks (see Section 4.2.1) at input. *InfrastructureBuilder* creates VMs, necessary for infrastructure of the benchmarks, by utilizing *ResourceProviderConnector*. Then *ArtefactsInstaller* installs prerequisites onto the machines and deploys blockchain artefacts into the topology according to the specification of the experiment. *NetworkConfigurator* setups a quality of network connections between nodes of the topology.
The *EmulatorRunner* deploys and executes *Emulator* containers in every node across the deployed topology. Those containers are responsible for running the benchmarks, by emulating the blockchain interactions in form of message exchange among MEC components. Therefore, the *Emulator* must be connected to a blockchain artefact. That makes the *Emulator* a producer and receiver of the messages. *Emulator* is a customized component and can be provided by the developer to emulate different blockchain interactions. We have used dockerized *V2X Communication Emulator* for our experiments. In

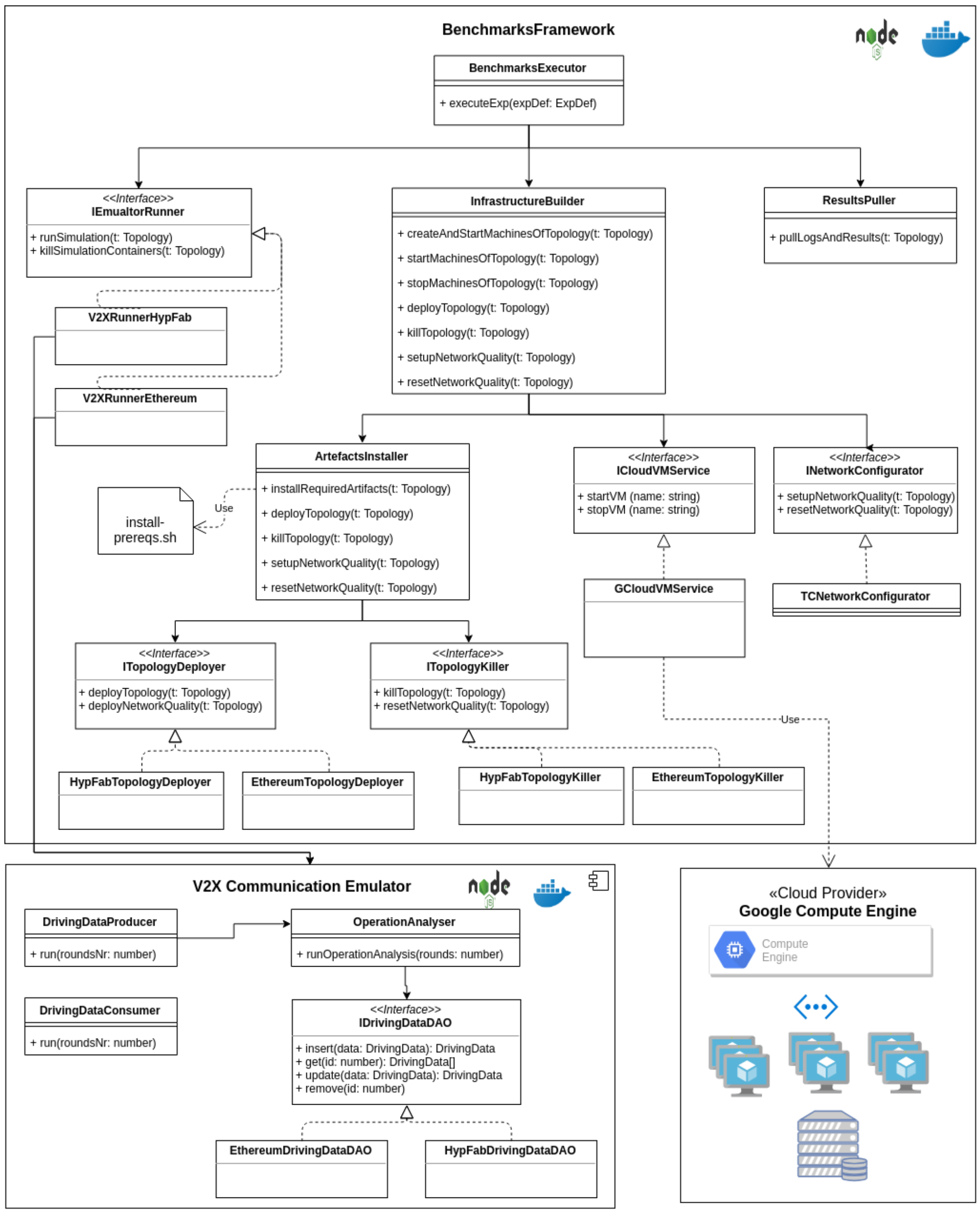

BenchmarksFramework
BenchmarksExecutor
+ executeExp(expDef: ExpDef)
<<Interface>>
IEmualtorRunner
+ runSimulation(t: Topology)
+ killSimulationContainers(t: Topology)
InfrastructureBuilder
+ createAndStartMachinesOfTopology(t: Topology)
+ startMachinesOfTopology(t: Topology)
+ stopMachinesOfTopology(t: Topology)
+ deployTopology(t: Topology)
+ killTopology(t: Topology)
+ setupNetworkQuality(t: Topology)
+ resetNetworkQuality(t: Topology)
ResultsPuller
+ pullLogsAndResults(t: Topology)
V2XRunnerHypFab
V2XRunnerEthereum
ArtefactsInstaller
+ installRequiredArtifacts(t: Topology)
+ deployTopology(t: Topology)
+ killTopology(t: Topology)
+ setupNetworkQuality(t: Topology)
+ resetNetworkQuality(t: Topology)
install-prereqs.sh
Use
<<Interface>>
ICloudVMService
+ startVM (name: string)
+ stopVM (name: string)
<<Interface>>
INetworkConfigurator
+ setupNetworkQuality(t: Topology)
+ resetNetworkQuality(t: Topology)
GCloudVMService
TCNetworkConfigurator
<<Interface>>
ITopologyDeployer
+ deployTopology(t: Topology)
+ deployNetworkQuality(t: Topology)
<<Interface>>
ITopologyKiller
+ killTopology(t: Topology)
+ resetNetworkQuality(t: Topology)
Use
HypFabTopologyDeployer
EthereumTopologyDeployer
HypFabTopologyKiller
EthereumTopologyKiller
V2X Communication Emulator
DrivingDataProducer
+ run(roundsNr: number)
OperationAnalyser
+ runOperationAnalysis(rounds: number)
DrivingDataConsumer
+ run(roundsNr: number)
<<Interface>>
IDrivingDataDAO
+ insert(data: DrivingData): DrivingData
+ get(id: number): DrivingData[]
+ update(data: DrivingData): DrivingData
+ remove(id: number)
EthereumDrivingDataDAO
HypFabDrivingDataDAO
«Cloud Provider»
Google Compute Engine
Compute Engine

Figure 4.4: Class diagram of the framework

the subsequent sections we elaborate deeper on the workflow of the benchmark framework.

4.3.1 Experiment Specification

In this section, we explain the format of experiment specification and topology of benchmarks accepted on input of the benchmark framework. A Listing 4.1 shows an example

of experiment specification. The feature of our benchmark framework is that beside the execution of benchmarks, it can generate benchmarks according to the configuration. The generated benchmarks are obtained by combining values of *bcImplementations*, with *bcDeployments*, *vehicleContainerConfiguration* and *networkQualities*. While each single combination of them is going to be benchmarked. Please note that for the specification in Listing 4.1, we evaluate two blockchain implementations, four deployments, three hardware configurations with three network qualities. That results in 27 generated and executed benchmarks.
Next we explain the parameters, used in Listing 4.1, to specify experiments. The purpose of *name* and *description* parameters is to provide a human-readable identification of benchmarks. *WorkloadEmulator* specifies the Emulator component, which is aimed to emulate interactions among MEC components via blockchain. That component is a separate application. For the benchmarks executed within this thesis, we emulated V2X communication scenarios by utilizing V2X Communication Emulator, running in *docker* and identified via its *imageTag*. The framework currently supports only a docker container for the workload emulator. Parameter *roundsNr* defines how many transactions should be issued by each of the workload emulator instances during an emulation.
The other parameters are optional. They specify the benchmarks, which should be generated and executed by the framework. *bcImplementations* says to the framework, what blockchain implementations should be used for interactions, when running the benchmarks. Currently we support Ethereum and Hyperledger-Fabric blockchains. Via the *bcDeployments* parameter, we define deployment of blockchain features to topologies as explained in Section 4.2.1. Please note that *bcDeployment* is an array, because the goal is to benchmark multiple deployments. Furthermore, there is an array of *vehicle-ContainerConfigurations*, used to specify hardware configurations of vehicles' resources, and *networkQualities* to configure quality of the network connection among the nodes of our topology.

As explained in Section 4.2.1, the topology is represented as a graph. The nodes of the graph are instances of the *Node* class, depicted in Figure 4.5. A JSON representation of the graph is required on input. When the framework is generating benchmarks, it injects the values of *bcImplementations*, *bcDeployment*, *vehicleContainerConfigurations* and *networkQualities* from experiment specification to the respective fields of the topology (*bcArtefact*, *hostMachine*, *netQuality*). Therefore, those topology' fields don't need to be provided. However, there might be use cases, when the developer wants to benchmark a specific topology. For example a topology where `vehicle1` runs *creator* feature, while `vehicle2` runs *consensus* feature. Therefore, the developer provides values for *bcArtefact* field in the topology specification, and doesn't want to let it be overwritten by the framework. To achieve that the parameters *bcImplementations* and *bcDeployment* cannot be defined in the specification of experiment. In another use case, the developer might want to benchmark various deployments of blockchain features to a topology, where `vehicle1` is running in a container with 1vCPU, while `vehicle2` requires 4vCPU. To achieve that the topology' *hostMachine* field has to be defined, while the *vehicleContainerConfigurations* cannot be defined in the experiment specification.

All experiment specifications and topologies, which have been used for the benchmarks in this thesis are available in *experiments* directory of the following GitHub repository: `https://github.com/rdsea/blockchainbenmarkservice`.

Listing 4.1: Definition of experiments 0..26, as accepted by the benchmark framework

```
name: interaction2
description: Experiments for Interaction2

workloadEmulator:
  type: docker
  imageTag: filiprydzi/v2x_communication
roundsNr: 100

bcImplementations:
- eth
- hypfab

bcDeployments:
- id: 4
featuresMapping:
- nodeType: rsu
feature: all
- nodeType: vehicle
feature: all
- id: 2
  featuresMapping:
  - nodeType: rsu
    feature: creator
  - nodeType: vehicle
    feature: all
- id: 0
  featuresMapping:
  - nodeType: rsu
    feature: all
  - nodeType: vehicle
    feature: creator

vehicleContainerConfigurations:
- name: small
  vCPUcount: 1
  memory: 2
  storageSSD: 10
  storageHDD: 0
  os: ubuntu18.04
- name: medium
  vCPUcount: 2
  memory: 4
  storageSSD: 10
  storageHDD: 0
  os: ubuntu18.04
- name: large
  vCPUcount: 4
  memory: 8
```

```
    storageSSD: 10
    storageHDD: 0
    os: ubuntu18.04

networkQualities:
- name: 3G
    latency: 200ms
    bandwidth: 1000kbps
- name: 4G
    latency: 100ms
    bandwidth: 10000kbps
- name: 5G
    latency: 5ms
    bandwidth: 54mbps
```

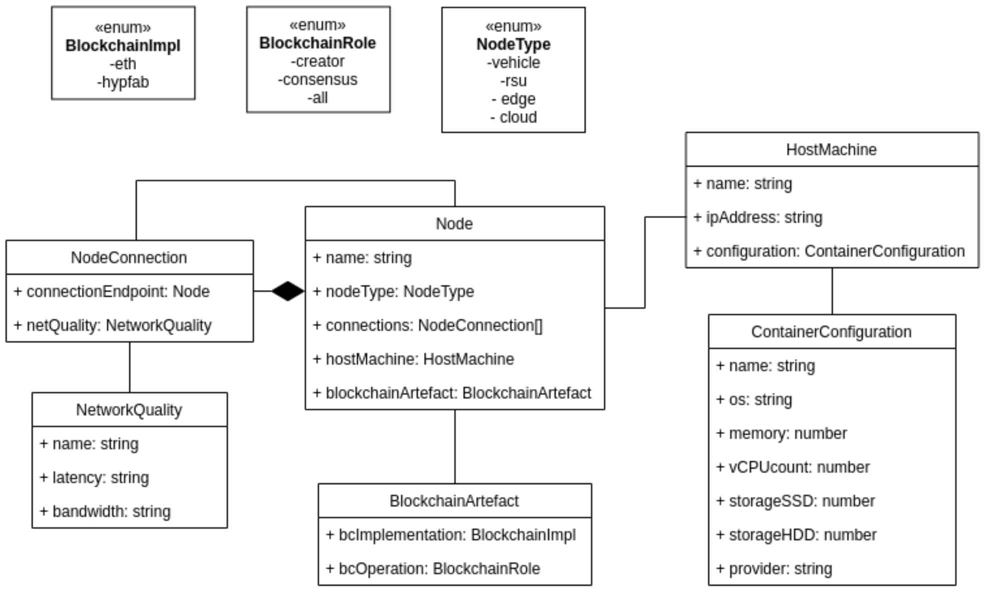


Figure 4.5: Topology format accepted by benchmark framework

4.3.2 Building of Infrastructure

BenchmarkExecutor invokes *InfrastructureBuilder*, which is responsible for building the infrastructure. At first it uses *ResourceProviderConnector* to create and start necessary VMs in the cloud. Currently, the framework supports Google Compute Engine [21] as provider of the resources(VMs, networks), but can be extended to work with other providers as well. *ResourceProviderConnector* utilizes a Node.js client library [1] to interact with the Google Compute Engine. Then the *ArtefactInstaller* installs necessary prerequisites onto the machines. That includes, installing *docker* and *docker-compose*.

[1]https://github.com/googleapis/nodejs-compute

Furthermore, the *ArtefactInstaller* deploys and links blockchain artefacts to the topology of benchmarks, by using respective *ITopologyDeployer* implementation. The currently supported blockchain artefacts are running in docker containers across the topology. We have used geth docker image [2] for Ethereum. The official docker images [3] have been provided by Hyperledger-Fabric to run peer node, orderer node, tools, certificate authority, Kafka and Zookeeper.
Figure 4.2 depicts an example deployment of blockchain artefacts. Please note that a single VM can run multiple software services representing the MEC components. In this case one vehicle is represented by the V2X Communication emulator and Hyperledger-Fabric peer node. Hyperledger-Fabric orderer node is deployed on RSUs and edge, while other software services (Kafka, Zookeeper and certificate authority) are deployed to the edge node as well.
NetworkConfigurator is utilized by *InfrastructureBuilder* to configure quality of network connections among nodes of the benchmark's topology. *NetworkConfigurator* uses a tool called *tc* [4], to manipulate the traffic control.
As soon as the built infrastructure isn't required anymore, *InfrastructureBuilder* invokes *ArtefactInstaller* to kill containers representing blockchain artefacts. Then the *InfrastructureBuilder* invokes *ResourceProviderConnector* to stop VMs in the cloud.

4.3.3 Benchmark Execution

The *EmulatorRunner* deploys and executes *Emulator* containers in every node across the deployed topology. The *Emulator* container connects to a blockchain artefact. Then the *Emulator* starts producing messages and emulates blockchain interactions via exchanging the messages with other nodes of the topology. The blockchain interactions involve invocation of features (refer to Section 3.5) in the blockchain artefact, which are necessary for emulation of the blockchain interactions. *Emulator* can be customized by the developer to emulate specific blockchain interactions.
We have used *V2X Communication Emulator* to address the V2X communication scenarios for our experiments. The *V2X Communication Emulator* is deployed to vehicle nodes of the benchmark's topology. It's a separate NodeJS [43] application, developed in Typescript [56], running in a docker container and is provided to the *EmulatorRunner* by an external container repository (DockerHub [16]). The responsibility of *V2X Communication Emulator* is to emulate interactions by exchanging data among the nodes of benchmark's topology via blockchain. *DrivingDataProducer* in *V2X Communication Emulator* produces a stream of random driving data. Listing 4.2 shows a concrete example of the data stream, consisting of five data instances, produced by a single emulated vehicle (*V2X Communication Emulator* instance). The parameter *roundsNr* has been set in experiment specification (refer to Section 4.3.1) and defines how many times a new driving data instance have to be produced and pushed into the stream. *OperationAnalyser* receives the data from *DrivingDataProducer* and delivers them to

[2]https://github.com/ethereum/go-ethereum
[3]https://github.com/hyperledger/fabric
[4]http://manpages.ubuntu.com/manpages/bionic/man8/tc.8.html

IDrivingDataDAO, which executes the blockchain's *creator* feature to persist the data into blockchain. Then the data become visible for other *V2X Communication Emulator* instances, thus the *V2X Communication Emulator* is used as a data source and data receiver as well. *OperationAnalyser* observes some aspects of the interactions. Those aspects include for example transaction acceptance time, as well other factors important for the quality metrics (see Section 3.3).
As soon as all *V2X Communication Emulators* have finished running the benchmarks. The framework will use *ResultsPuller* to download the logs and results of the benchmarks from all nodes across our topology. The results are mapped to the quality metrics (refer to Section 3.3), supported by the framework.

The developed framework is open-source and is available in the GitHub repository: `https://github.com/rdsea/blockchainbenmarkservice`. It can be used to run other benchmarks and is open for extensions. It can be extended to work with other implementations of blockchain, other cloud providers, or use another component for emulations than the V2X Communication Emulator.

Listing 4.2: Samples of stream of driving data used for benchmarks

```
{
  id: 7785705911,
  timestamp: 2019-30-3 20:27:58.1
  velocity: 50,
  acceleration: 1
},
{
  id: 7785705912,
  timestamp: 2019-30-3 20:27:58.2
  velocity: 51,
  acceleration: 5
},
{
  id: 7785705913,
  timestamp: 2019-30-3 20:27:59.1
  velocity: 56,
  acceleration: -10
},
{
  id: 7785705914,
  timestamp: 2019-30-3 20:28:01.9
  velocity: 46,
  acceleration: -20
},
{
  id: 7785705915,
  timestamp: 2019-30-3 20:28:03.6
  velocity: 26,
  acceleration: -1
}
...
```

4.4 Experiments

4.4.1 Experiment Design

In this section we create and benchmark a set of experiments for the identified interactions (refer to Section 3.4), which we believe might provide relevant information for the developers. In the list below we explain and justify, what interactions and deployments do we choose to benchmark.

- **Interactions 2 and 3** (explained in Sections 3.4.2 and 3.4.3): In interaction 2, vehicles and RSU have been involved, while in interaction 3, message exchange among vehicles and edge node has taken place. From a developer's point of view, it might be important to know, how does the utilized MEC component (RSU or edge), influence the benchmark's outcome. Since we know that RSU is a computationally weak machine with network capabilities, while edge node is computational powerful. For these interactions the same deployment models have been proposed (refer to Tables 4.2 and 4.3).
  - **Deployment 4**: This one deploys each blockchain feature to every of the involved components. This deployment is interesting to observe, because it should be the most complex one and thus the developer can use it as a baseline for other deployments.
  - **Deployment 0 and 2**: The goal of the developer is to deploy less blockchain features to the components, because it decreases complexity, while simultaneously aiming to achieve better results. One of the lighter deployments is 0: creator feature is executed in vehicles, while RSU|edge node stays the same as in deployment 4. The drawback is that there are only a few nodes achieving consensus, which makes the network partially centralized. Similar principle holds for deployment 2, all vehicles deploy every blockchain's feature, while RSU|edge node deploy only the creator feature.
- **Interaction 4** (explained in Section 3.4.4): This interaction involves vehicles, RSUs and edge. It's the third one identified for the Obstacle on the road warning scenario. Therefore, it's important for the developer to know, what interaction does perform the best for that scenario. That's the reason, why we choose this interaction to be benchmarked. Deployment models for interaction 4 have been proposed in Table 4.4.
  - **Deployment 8**: As in the previous interactions, the developer wants to begin with the most complex deployment to serve as the baseline. The goal is to keep improving the baseline by utilizing other deployments.
  - **Deployment 7, 6 and 11**: Each of these deployments present a less complex deployment. In the deployment 7, the features in RSU and edge node stay the same as in deployment 8, while vehicle is only a creator. In the deployment

6 we proceed to a lighter deployment by removing creator from RSU. The deployment 11 places creator to RSU and edge, while vehicle stays the same as in deployment 8.

- **Interaction 1** (explained in Section 3.4.1): This interaction includes only one single deployment, thus won't be benchmarked in this thesis.
- **Interaction 5 and 6** (explained in Sections 3.4.5 and 3.4.6): These interactions might be interesting to evaluate as well, but their topology is similar to interactions 3 and 4 respectively, thus we favored those within the thesis.

For each of the chosen deployments, multiple hardware configurations of resources emulating the vehicles have been evaluated. As explained in the Table 4.7, there are three such configurations, one another configuration for RSU and one for edge node. Furthermore, all network qualities listed in Table 4.8 have been benchmarked. It means that we have nine experiments for each of the deployments. We chose ten deployments, for each we create nine experiments, totals in 90 experiments.
As we explained in Section 2.2.2, the developer needs to know, what blockchain implementation to use. Therefore, we evaluate how the experiments behave on the chosen blockchain implementations. This doubles the count of experiments we have to create. Since scalability is one of the considered quality metrics relevant to the developers, we include the scale to the experiments as well.

- **Small scale:** This compromises only the nodes mentioned in the interactions, explained in Section 3.4. Meaning that for interaction two we have two vehicles and a RSU. In third there are two vehicles and one edge. And for the fourth one, we consider an edge node with two RSUs and two vehicles.
- **Large scale:** For this scale, we have the same number of edges and RSUs as for small scale, but we increased the count of vehicles to 48. If we think of a situation, having vehicles driving on a four-lane highway, means 12 vehicles are traveling in each lane on average. Assuming that each vehicle needs to occupy ca 50 meters on a highway (according to valid traffic rules for safe driving on highways in Austria [45]). That totals in 600 meters long fleet on a full highway, what is a pretty long distance for a RSU and sufficient long for an edge node. That should be fair enough for a large scale experiment.

When running benchmarks, each of the vehicles created and submitted 100 transactions when emulating the interactions. In total we have 200 transactions for each small scale experiment, and 4800 transactions for large scale case.
Table 4.9 lists all 180 experiments created for Ethereum blockchain. Table 4.10 presents the experiments created for Hyperledger-Fabric, but we have only 144 experiments here since we didn't benchmark large scale for interaction 4 (see Section 3.4.4). That gives us 324 created experiments in total.

Table 4.9: All created experiments for Ethereum blockchain

	Experiment Specification					
Experiment id	**Interaction id**	**Deployment id**	**Scale**	**Blockchain**	**vehicle VM**	**Network quality**
0	2	0	small	Ethereum	light	3G
1						4G
2						5G
3					medium	3G
4						4G
5						5G
6					big	3G
7						4G
8						5G
[9 . . . 17]		2			[light; medium; big]	[3G; 4G; 5G]
[18 .. 26]		4			[light; medium; big]	[3G; 4G; 5G]
[27 .. 35]	3	0			[light; medium; big]	[3G; 4G; 5G]
[36 .. 44]		2			[light; medium; big]	[3G; 4G; 5G]
[45 .. 53]		4			[light; medium; big]	[3G; 4G; 5G]
[54 .. 62]	4	6			[light; medium; big]	[3G; 4G; 5G]
[63 .. 71]		7			[light; medium; big]	[3G; 4G; 5G]
[72 .. 80]		8			[light; medium; big]	[3G; 4G; 5G]
[81 .. 89]		11			[light; medium; big]	[3G; 4G; 5G]
[90 .. 98]	2	0	large		[light; medium; big]	[3G; 4G; 5G]
[99 .. 107]		2			[light; medium; big]	[3G; 4G; 5G]
[108 .. 116]		4			[light; medium; big]	[3G; 4G; 5G]
[117 .. 125]	3	0			[light; medium; big]	[3G; 4G; 5G]
[126 .. 134]		2			[light; medium; big]	[3G; 4G; 5G]
[134 .. 143]		4			[light; medium; big]	[3G; 4G; 5G]
[144 .. 152]	4	6			[light; medium; big]	[3G; 4G; 5G]
[153 .. 161]		7			[light; medium; big]	[3G; 4G; 5G]
[162 .. 170]		8			[light; medium; big]	[3G; 4G; 5G]
[171 .. 179]		11			[light; medium; big]	[3G; 4G; 5G]

Table 4.10: All created experiments for Hyperledger-Fabric blockchain

	Experiment Specification					
Experiment id	**Interaction id**	**Deployment id**	**Scale**	**Blockchain**	**vehicle VM**	**Network quality**
[180 .. 188]	2	0	small	Hyperledger-Fabric	[light; medium; big]	[3G; 4G; 5G]
[189 .. 197]		2			[light; medium; big]	[3G; 4G; 5G]
[198 .. 206]		4			[light; medium; big]	[3G; 4G; 5G]
[207 .. 215]	3	0			[light; medium; big]	[3G; 4G; 5G]
[216 .. 224]		2			[light; medium; big]	[3G; 4G; 5G]
[225 .. 233]		4			[light; medium; big]	[3G; 4G; 5G]
[234 .. 242]	4	6			[light; medium; big]	[3G; 4G; 5G]
[243 .. 251]		7			[light; medium; big]	[3G; 4G; 5G]
[252 .. 260]		8			[light; medium; big]	[3G; 4G; 5G]
[261 .. 269]		11			[light; medium; big]	[3G; 4G; 5G]
[270 .. 278]	2	0	large		[light; medium; big]	[3G; 4G; 5G]
[279 .. 287]		2			[light; medium; big]	[3G; 4G; 5G]
[288 .. 296]		4			[light; medium; big]	[3G; 4G; 5G]
[297 .. 305]	3	0			[light; medium; big]	[3G; 4G; 5G]
[306 .. 314]		2			[light; medium; big]	[3G; 4G; 5G]
[315 .. 323]		4			[light; medium; big]	[3G; 4G; 5G]

All experiment specifications and topologies, which have been used for the benchmarks in this thesis are available in *experiments* directory of the following GitHub repository: `https://github.com/rdsea/blockchainbenmarkservice`.

4.4.2 Evaluation

A subset of benchmarks' results is depicted in Table 4.11. The table shows the experiments, which performed the best, regarding to the quality metrics, among all deployments and infrastructures for each interaction, blockchain implementation and scale. The semantic of the first row is following: when running the benchmarking experiment 24, all nodes executed 200 transactions altogether, all of them have been accepted, achieved stated acceptance times. Each node was synchronized, when simulation has finished. A vehicle node fully utilized all 4 CPU's cores, while a single CPU core is an Intel Xeon E5 Sandy Bridge 2.6 GHz as defined in Table 4.7, 1192.4 MB of RAM. A RSU node fully utilized one CPU core and 235.5 MB of RAM, edge node isn't involved in interaction 2.
In this section we present how the developers could benefit from the framework by explaining the results of benchmarks, executed by the framework. When choosing the best results we interpret the requirements of developers, formulated in Section 3.4.7, in form of the quality metrics as follows:

- **High Synchronization State and High Transaction Acceptance Rate**: We assumed, the developers would prefer benchmarks with the least number of nodes, which lost their synchronization during simulations. In case of equality, they would choose the one with least number of failed transactions.
- **Short Transaction Acceptance Time**: The developers pick benchmark achieving the shortest median of transaction acceptance times. In case two benchmarks have achieved equal medians, they choose the one with smaller average time.
- **Low Infrastructure Resources Utilization**: Benchmarks' results with the smallest CPU utilization for vehicles have been preferred.

Table 4.11: Best benchmarks results according to our interpretations

Input			**Result**	**Performance**				**Reliability**		
				Transaction Acceptance Time				**Transaction Acceptance Rate**		
Interaction Id	**Blockchain implementation**	**Scale**	**Experiment ID**	*Minimum Time*	*Maximum Time*	*Median Time*	*Average Time*	*Accepted count*	*Failed count*	**Not-sync nodes count**
2	eth	small	24	240	6510	2316.5	2507.7	200	0	0
2	eth	large	94	979	23013	5691.5	6617.78	4800	0	0
2	hypfab	small	188	2044	2072	2054	2054.45	200	0	0
2	hypfab	large	272	50	6956	124	487.18	4800	0	0
3	eth	small	53	474	5120	1862	2153.24	200	0	0
3	eth	large	125	429	16774	3698.5	4444.62	4800	0	0
3	hypfab	small	233	2038	2064	2049	2049.55	200	0	0
3	hypfab	large	302	49	5708	154	494.91	4800	0	0
4	eth	small	61	199	4914	2151.5	2188.34	200	0	0
4	eth	large	149	306	47711	5218.5	6529.1	4618	182	0
4	hypfab	small	260	2041	2063	2048.5	2050.21	200	0	0

4.4.2.1 Results of interaction 2 (Vehicle to RSU)

These results relate to the obstacle warning scenario explained in Section 3.4.2. Plots in Figures 4.6 and 4.7 depict a dependency between the infrastructure and medians of transaction (tx) acceptance times; infrastructure and tx acceptance rate respectively, among all blockchain deployments for this interaction. For small scale topology, the benchmarks have shown that 100% tx acceptance rate has been achieved for both blockchain implementations. Concerning the median times deployments 2 and 4 in Ethereum seem to depend on a used infrastructure in a direct proportion. Although for deployment 0 a more powerful infrastructure results to longer median times. That might be caused by a centralized nature of the deployment, meaning that we deploy every blockchain feature to the RSU, but vehicles are executing only the creator feature. Based on that, we assume that the better hardware we use for the vehicles, the more txs are submitted. While the RSU is not powerful enough to accept them all. In contrast to Ethereum, for Hyperledger-Fabric the plots follow the same pattern for every deployment and vehicle resource configuration. Concretely, the plots are showing that better quality of the network leads to shorter acceptance times of the txs. The developer can conclude that experiment number 188 performed the best concerning the preferred quality metrics for small scale (deriving from Table 4.11). That's Hyperledger-Fabric blockchain, deployment 0, *big VM configuration type* for the *vehicles* and *5G network* for connections between vehicles and RSUs. However, the deployment 0 has a downside, because it makes the network partially centralized. Since only RSUs are used to run consensus feature and we expect to have only a few RSUs and more vehicles. While Ethereum experiments suggested deployment 4, which leaves the network decentralized as the principles of blockchain are saying, but decentralization isn't one of our preferred quality metrics.
For large scale topology, the only deployment for which we measured 100% tx acceptance rate is deployment 0. The reason is that deployment 0 is centralized (consensus feature is deployed only to RSU), thus it's easier to synchronize. Nevertheless, the acceptance rate of deployment 2 and 4 seems to depend on the used infrastructure in a direct proportion. Regarding the median times we witness a similar pattern as in small scale for Ethereum. While Hyperledger-Fabric in large scale reports the same behavior as for small scale concerning the network quality. However, it's interesting that the better hardware we use for the vehicle node, the worse performance we get. That might be caused that the vehicles create more txs when utilizing more powerful hardware, thus it takes longer to synchronize. Hyperledger-Fabric has shown much better results for large scale in the interaction. Thus the best deployment and infrastructure is the one used for experiment 272 on large scale.

Deriving from all results and the above observations, the developer might choose *Hyperledger-Fabric blockchain, deployment 0, small machine type for vehicles and 5G network*. As we obtained the best results among all experiments for both scales concerning the reliability and performance for the interaction. However, deployment 0 makes the network partially centralized, which might break the principles of blockchain.

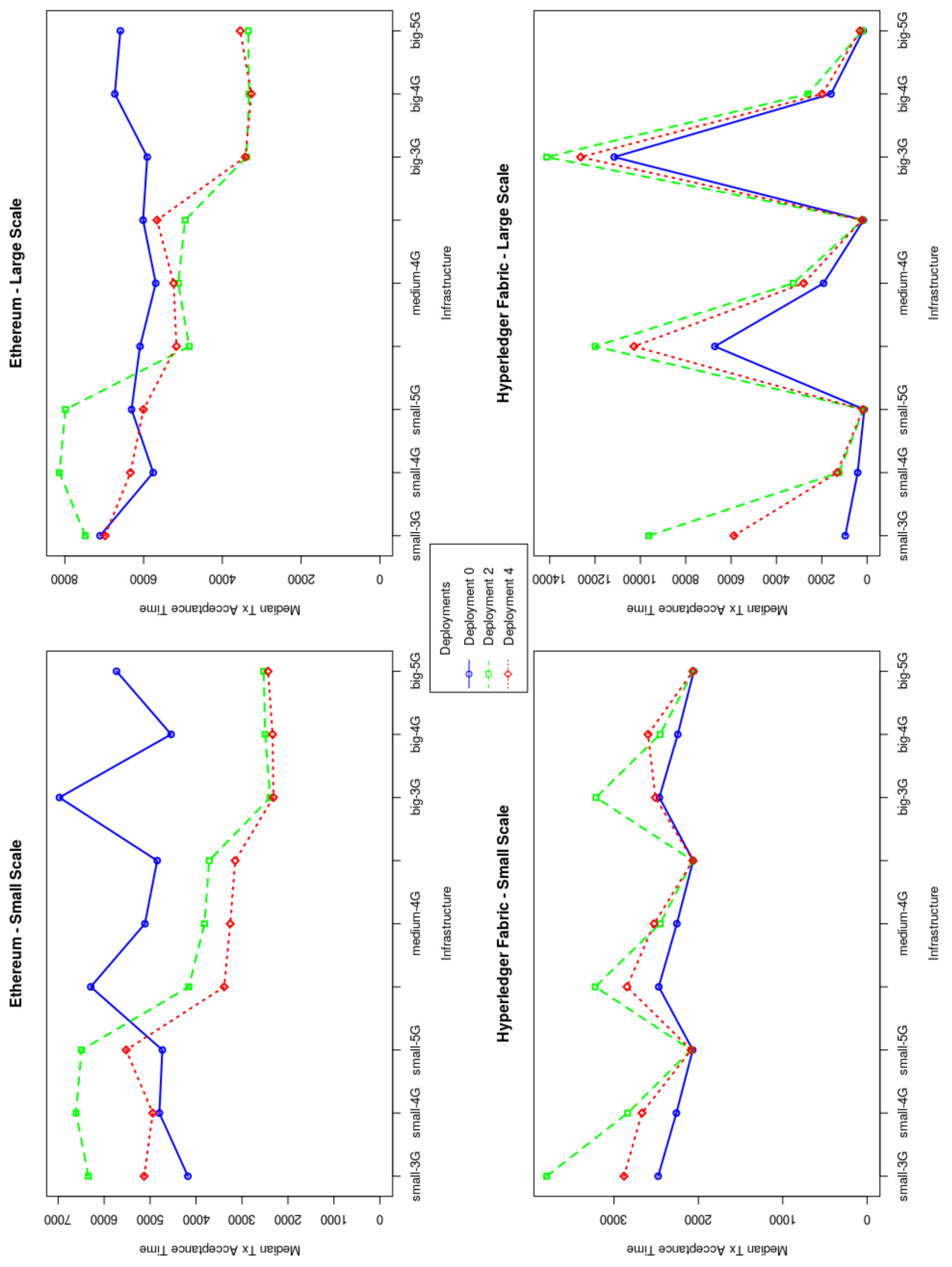


Figure 4.6: Medians of tx acceptance times for interaction 2

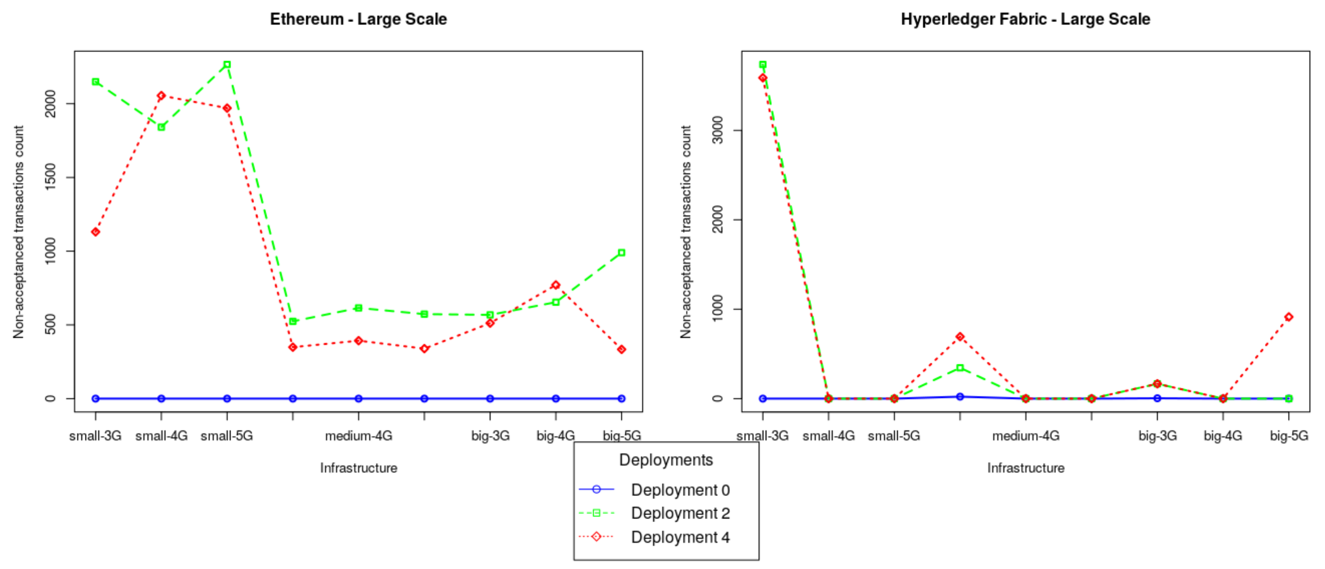


Figure 4.7: Number of rejected txs for interaction 2

4.4.2.2 Results of interaction 3 (Vehicle to Edge)

The benchmarks we executed for this interaction are the same as in the previous interaction. The only difference is that we have used edge node instead of RSU in the interaction's topology. Thus we assume to obtain better results, since edge node has more computational power than RSU (this is shown in Table 4.7).
The median times depicted in Figure 4.8 follow similar patterns as in the previous interaction for both blockchain implementations and scales. For the developer, an interesting observation is that even the deployment 0 in Ethereum seems to depend on the infrastructure in contrast to the previous interaction. This matches our assumption that we should obtain better results than before, because edge node is more powerful than RSU. Deployment 0 is partially centralized here as well, thus running consensus only in the edge node takes shorter time than in RSU.
In this interaction we face a non-reliable behavior for deployment 2 when using Ethereum in small scale. That is surprising, but a reason is probably that the edge node cannot accept txs in time. Altogether, we have 19920 rejected txs for Ethereum in large scale for this interaction. However, we had 18034 rejected transactions in the previous. Although we assumed that we would have had less rejected txs in this interaction. But in deployment 2 we observed an increased number of rejected txs for the big machine types for vehicles. That might be again caused by too many created txs, which cannot be accepted in time. For all other deployments in large scale, we faced the same patterns as in previous interaction.

For the developer is interesting to observe that experiment number 53 performs the best for small scale. That gives a recommendation to use Ethereum blockchain, deployment 4 (defined in Table 4.3), big machine type for vehicle nodes and 5G network. Hyperledger-Fabric blockchain (experiment 302) outperformed Ethereum when running large scale benchmarks for the interaction. Precisely for deployment 0, medium machine types for vehicles and 5G network we got the best results.
After analyzing all benchmarks for the interaction 3. Our framework could provide a hint to the developer that *Hyperledger-Fabric blockchain, deployment 0, medium machine type and 5G network* are considered as best deployment and configuration among all evaluated for interaction 3.

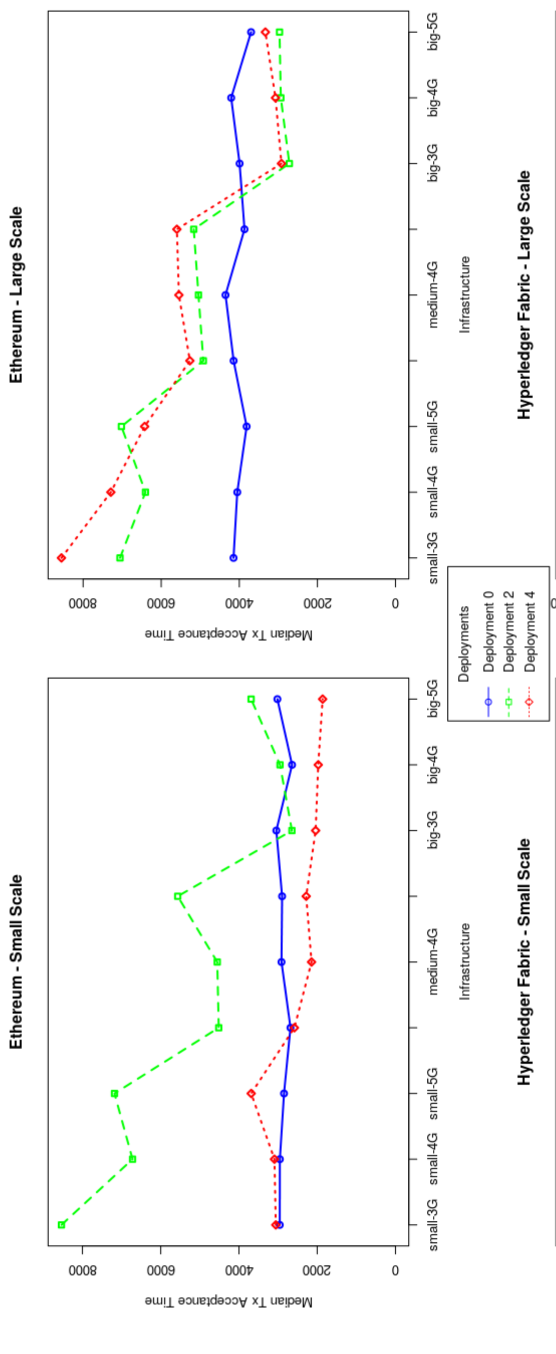


Figure 4.8: Medians of tx acceptance times for interaction 3

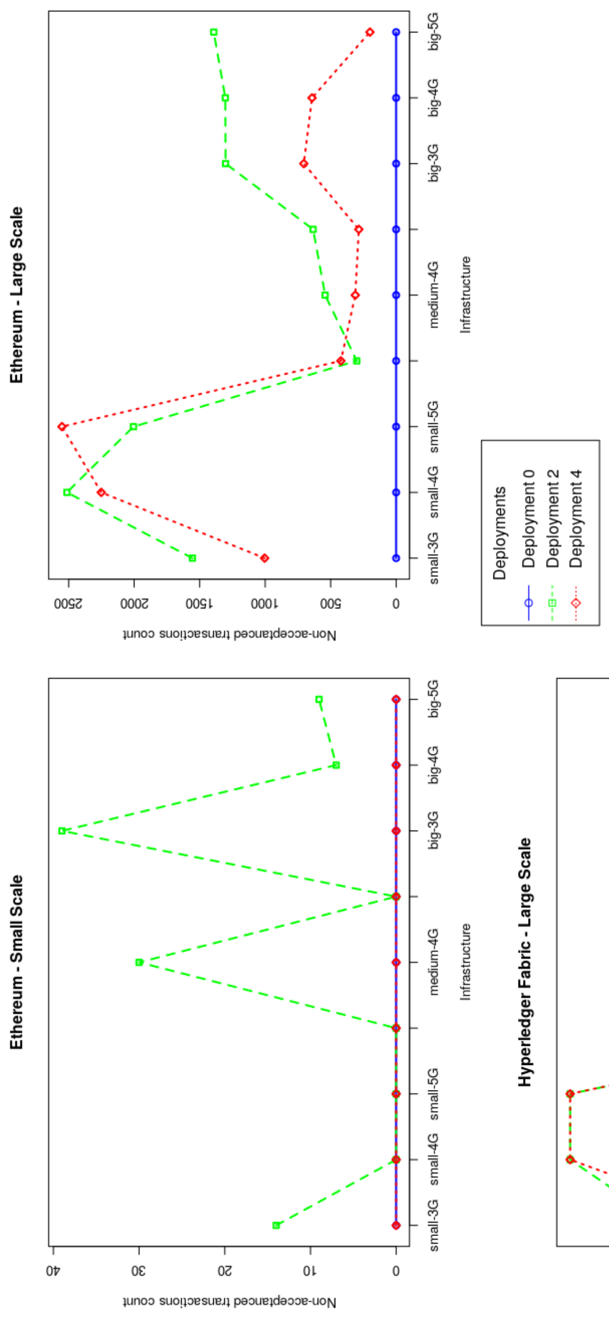


Figure 4.9: Number of rejected txs for interaction 3

4.4.2.3 Results of interaction 4 (Vehicle to RSU and Edge)

The results we present here refer to the interaction explained in Section 3.4.4. Let's discuss the median times (depicted in Figure 4.10) and tx acceptance rates (Figure 4.11) of this interaction. We begin with the simplest case, namely plot showing results for Ethereum in large scale. We observe the same dependency in the plot as we observed in the interaction 2. There is a direct dependency between the infrastructure and median times. The same pattern as in previous interaction is shown in Hyperledger-Fabric for small scale, and it achieves 100% acceptance rate in this interaction. The median times of accepted txs in Ethereum, small scale seems to depend on the used infrastructure, while for the small machine types, no txs have been accepted. All other experiments for Ethereum-small scale achieved almost 100% rates as well. Concerning the acceptance rates of large scale Ethereum, the plots seems to show a dependency between infrastructure and number of rejected txs.

For experiment number 260 we achieved the best results according to the benchmarks. In the small scale case Hyperledger-Fabric performed better as in the previous two interactions. It utilized big machine type for vehicle nodes, 5G network and deployment 6. That places every blockchain feature in the edge node, while RSU is responsible only for consensus and vehicles for creating txs. We benchmarked only Ethereum blockchain for large scale in this interaction and achieved the best results for experiment 149. We get those for deployment 8, medium machine types for vehicle and 5G network. These results provide insights from the benchmarks of interaction 4 to the developers.

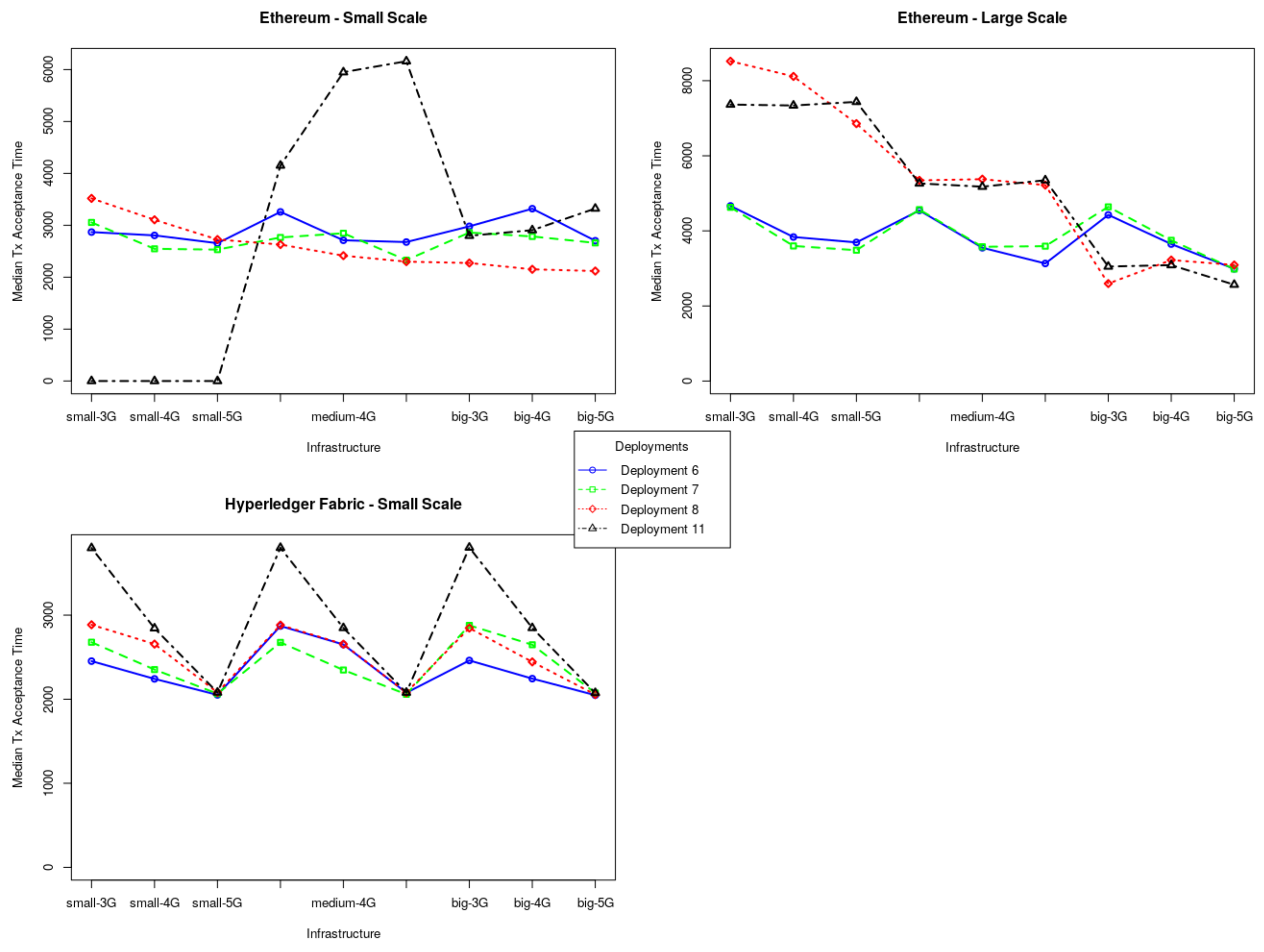


Figure 4.10: Medians of tx acceptance times for interaction 4

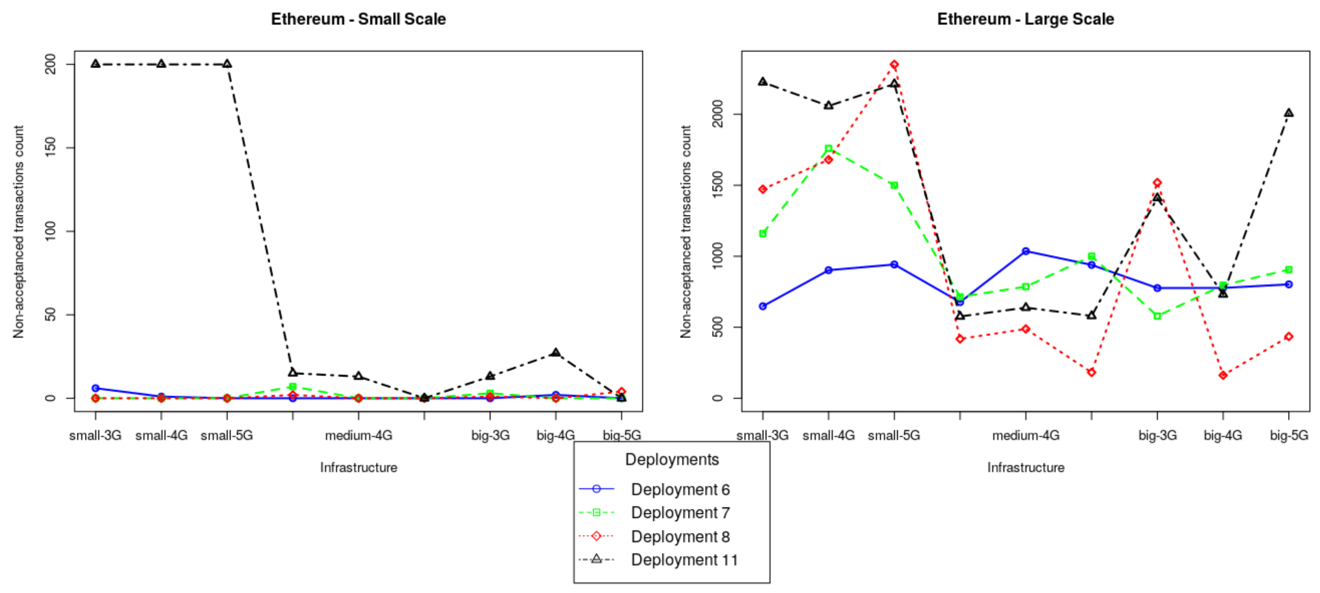


Figure 4.11: Number of rejected txs for interaction 4

4.4.2.4 Discussion

Since all three interactions above are related to the obstacle on the road warning scenario. The developer might be interested in interaction, which performed the best among all for that scenario.

- **small scale** - We see that all interactions achieved 100% acceptance rates, while for interaction 2 we got the best results regarding median times. That matches our assumption that interaction with edge node should perform better than the one with RSU. That may be the reason why interaction 3 outperformed interaction 2, and surprisingly interaction 4 as well, although it uses both RSU and edge.
- **large scale** - For the large scale, interactions 2 and 3 achieved 100% acceptance rates, unlike interaction 4, where we got 182 rejected txs. That might be caused by too many nodes in the topology, which weren't able to accept all txs, created in a short time-span. Regarding the median times, interaction 2 was better than 3. That is in contrast to small scale and is surprisingly against our assumption that we should get better results for interaction 3 than 2.

All in all, deriving from the benchmarks the developer can conclude that interaction 2 performed the best. Namely, *Hyperledger-Fabric blockchain, deployment 0, small machine type for vehicles and 5G network* achieved the best results concerning reliability and performance among all interactions for the scenario. Nevertheless, the suggested deployment and infrastructure might still not be sufficient for the real obstacle warning scenario. Because as Bettisworth [6] et al. defined a maximum latency to 100ms in safety application. None of the executed experiments managed to meet that requirement.

4.5 Summary

In this chapter we contribute a blockchain benchmark framework, which is able to benchmark blockchain interactions among MEC components. We present an architecture and prototype of the framework. To demonstrate flexibility of our framework we created and benchmarked 324 experiments for interactions identified and described in Chapter 3. We have shown what insights do the benchmarks bring for the developers, dealing with blockchain-based applications addressing our scenarios.

CHAPTER 5

Experiments Knowledge Service

5.1 Overview

In the previous chapter we created and benchmarked a set of experiments against quality metrics (see Section 3.3). We want to provide the achieved benchmarks along with their topologies and infrastructures to developers, such that they can reuse knowledge gathered by the benchmarks. Furthermore, developers will execute another benchmarks in the future, which might add relevant information to the knowledge. Therefore, we propose and develop *Experiments Knowledge Service*, which stores and manages data related to the benchmarks. By means of the service, the developers can gain insights about infrastructures, blockchain interactions among the MEC components and software artefacts, which have been used for the benchmarks. We consider blockchain artefacts and eventually artefacts representing other services like Kafka or Zookeeper (see Section 4.3.2) as the software artefacts. Furthermore, the service utilizes the stored benchmarks data to give recommendations about deployment of blockchain artefacts to topology of a blockchain-based application running in MEC. Via the recommendations, the developers reuse the knowledge, stored by the service, so they don't need to run benchmarks again. In this chapter we propose a structure for the data in Section 5.2. The service exposes a set of APIs enabling to manage the data and we explain operations exposed via the APIs in Section 5.3. In Section 5.4 we elaborate on the prototype of the service. We show concrete examples on how the developers can benefit by utilizing the Experiment Knowledge Service in Section 5.5.

5.2 Architecture and Models

5.2.1 Experiments Knowledge Service Architecture

Figure 5.1 depicts a high-level architectural overview of the Experiments Knowledge Service. The Experiments Knowledge Service is internally composed of multiple services, each has a specific responsibility. In this section we explain their functionality.
Via *Deployment Pattern Service* a developer can manage all deployment patterns stored by the Experiments Knowledge Service. A deployment pattern is a graph consisting of MEC components, presenting graph's nodes, while edges of the graph represent blockchain interactions among the MEC components. Consider the following example: a deployment pattern for interaction identified in Section 3.4.2 is vehicle-RSU-vehicle. The deployment pattern is basically a topology of a benchmark (see Section 4.2.1). However it's composed by pure MEC components without any blockchain features deployed on the components. This service exposes a set of APIs enabling to list all, delete or create new deployment patterns in the Experiments Knowledge Service.
Infrastructure Service manages metadata about resources, which have been utilized for infrastructure of a benchmark (see Section 4.2.2), stored in by the Experiments Knowledge Service. The resources, we have used for the benchmarks have been provided by an external provider. The resources's metadata reference to that provider. This service provides APIs to operate on the stored metadata.
The responsibility of *Software Artefact Service* is to manage metadata, stored in Experiments Knowledge Service, of software artefacts, used in the benchmarks. We consider blockchain artefacts, emulator and other software services (see Section 4.3), which have been deployed to the MEC components when running the benchmarks. The Experiments Knowledge Service doesn't own any repository of the software artefacts. Therefore, we store only metadata concerning those artefacts. The metadata provide enough information for developers to execute those artefacts. The artefacts themselves are stored at an external provider (e.g. DockerHub [16]). The Software Artefact Service exposes APIs enabling to manage the stored metadata.
Blockchain Benchmark DaaS provides benchmarks' data, stored in the Experiments Knowledge Service. These data include an experiment, which is going to be/has been benchmarked, a topology of the experiment and quality metrics measured when benchmarking the experiment. Please note that experiments, which haven't been benchmarked yet, might be stored. A topology of the experiment is a deployment pattern, such that software artefacts (from *Software Artefact Service*) are deployed to the MEC components involved in the pattern. Each of MEC components in an experiment's topology has a resource (from *Infrastructure Service*) associated with it, which is used as a container for the MEC component. This service provides a set of APIs which enables the developers to create, delete and list all experiments with their topologies and achieved quality metrics.
The purpose of *Recommendation Service* is to give recommendations about deployment of blockchain artefacts to MEC components involved in a topology of blockchain-based application. The application is represented by a model, submitted to the Recommendation Service by a developer. The Recommendation Service identifies blockchain interactions

among the MEC components in the model and gives a recommendation based on those interactions. Furthermore, the service advises what infrastructure should be used for the application. The recommendations are proposed based on preferred quality metrics of the developer. This service utilizes other services to provide data for the recommendations. We elaborate further on the supported format of the application's model and on the process of giving recommendations in Section 5.3.

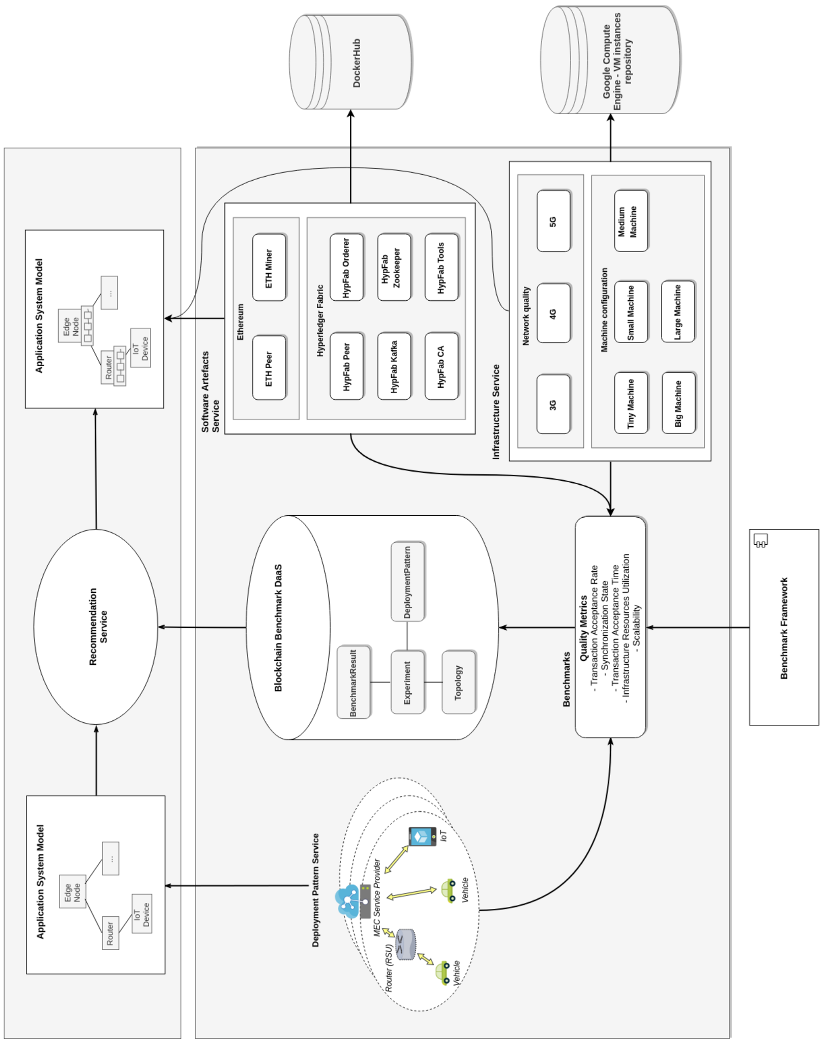


Figure 5.1: High-level overview of Experiments Knowledge Service

5.2.2 Experiment Knowledge Service Data Model

Figure 5.2 illustrates a model of data managed by Experiment Knowledge Service. In Sections 5.2.3 to 5.2.7 we provide an extensive explanation of the classes in the model.

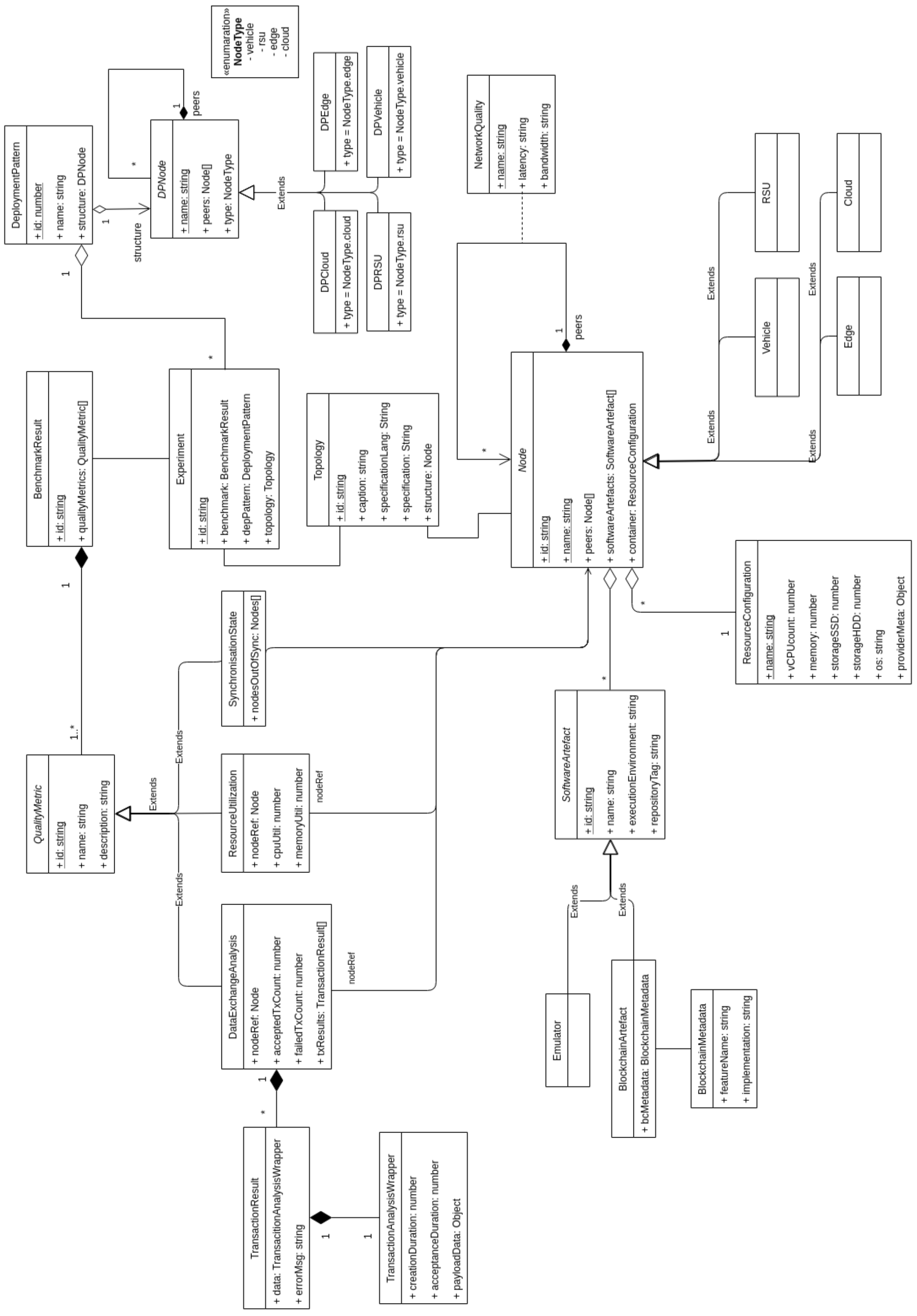


Figure 5.2: Model of data stored in the Experiments Knowledge Service

5.2.3 Specification of Deployment Patterns

The class diagram in Figure 5.3 defines a data model, used to represent a deployment pattern in our Experiments Knowledge Service. The *DPNode* class stands for a MEC component participating in a deployment pattern. In Section 3.2.1 we explained that there are four different MEC components. These are represented by *DPNode*'s subclasses (*DPCloud*, *DPEdge*, *DPRSU* and *DPVehicle*). These inherit all properties from *DPNode*. The model is extendable to support new MEC components, which can be introduced by a new *DPNode*'s subclass. Each *DPNode* is identified by its *id*. Attribute *name* presents its caption. A *DPNode*'s *peers* are other *DPNodes*, which are connected to the *DPNode* by an edge in the graph representing a deployment pattern. *DeploymentPattern* is a class representing deployment pattern. It has a reference to root *DPNode*, representing root node in the graph, of a particular deployment pattern.

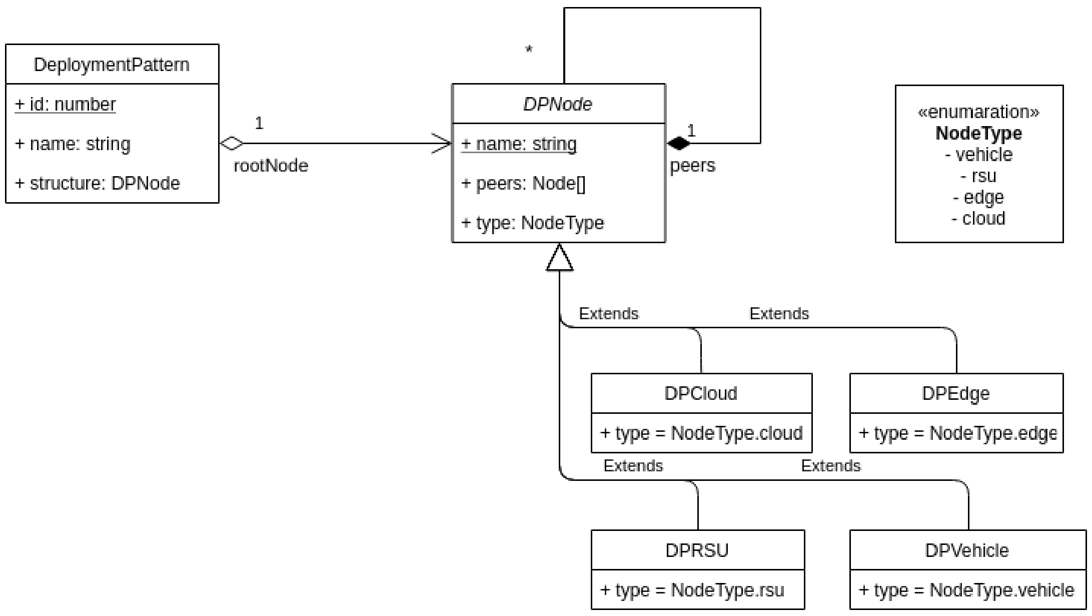


Figure 5.3: Deployment Pattern data model

5.2.4 Specification of Software Artefacts Information

A class *SoftwareArtefact* depicted in Figure 5.4 presents a data model representation of metadata concerning software artefacts in the Experiments Knowledge Service. A software artefact in general is an executable piece of software. The artefact is deployed and runs in an execution environment, which might be a docker container, operating system, etc. That is specified via *executionEnvironment* property. The Experiments Knowledge Service doesn't own any repository of the artefacts, therefore we use an external one. Variable *repositoryTag* is an identification of the artefact within the external repository.

For example if we use a docker container as our execution environment, the *repositoryTag* might be the id of container's image in DockerHub.
We create *BlockchainArtefact* class, representing blockchain artefacts, as a subtype of *SoftwareArtefact*. Figure 5.5 illustrates an Ethereum's miner node as an instance of the *BlockchainArtefact*. Other specific software artefact is the *Emulator*. There might be other software artefacts represented as instances of the *SoftwareArtefact* class in the Experiments Knowledge Service (see Section 4.3).

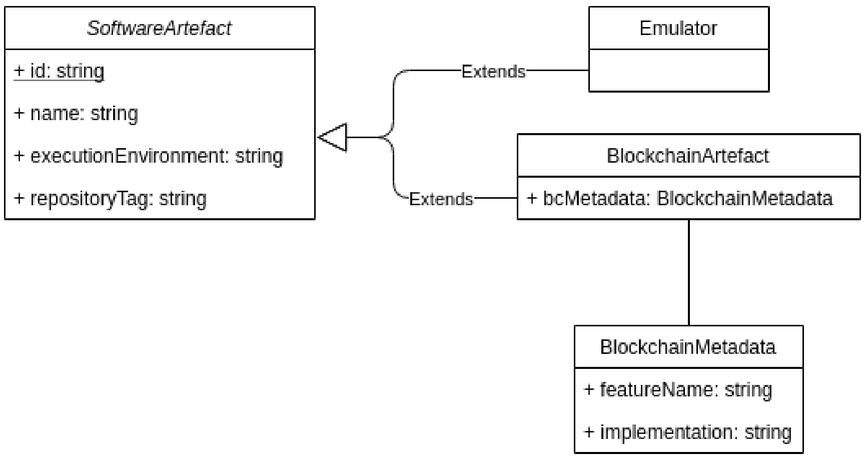
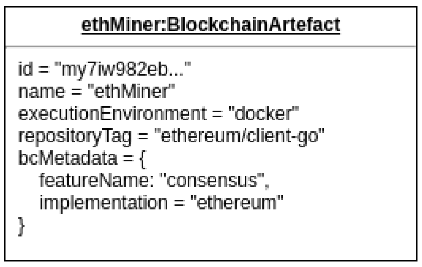


Figure 5.4: Software artefacts data model



ethMiner:BlockchainArtefact

```
id = "my7iw982eb..."
name = "ethMiner"
executionEnvironment = "docker"
repositoryTag = "ethereum/client-go"
bcMetadata = {
   featureName: "consensus",
   implementation = "ethereum"
}
```

Figure 5.5: Ethereum's miner node represented as *BlockchainArtefact* in the Experiments Knowledge Service

5.2.5 Infrastructures Specification

The data models depicted in Figure 5.6 show what metadata do we store for an infrastructure. In Section 4.2.2 we explained that the infrastructure, in our work, is composed of resources for the MEC components and configuration of network between the MEC components. The *ResourceConfiguration* class is identified via its name. We store the key hardware properties including: number of virtual CPU cores, memory in GBs and storages of the resources. Some metadata about a provider of the resource might be stored as well. That might include information about whether it's a real or emulated MEC resource. If it's an emulated one then it might be stored whether it's a virtual machine or a docker container. The concrete structure of the data, which is stored under infrastructure, is left to the developer. Furthermore, we persist operating system running on the resource. The *NetworkQuality* identified by a *name* and characterized via its *bandwidth* and *latency*.

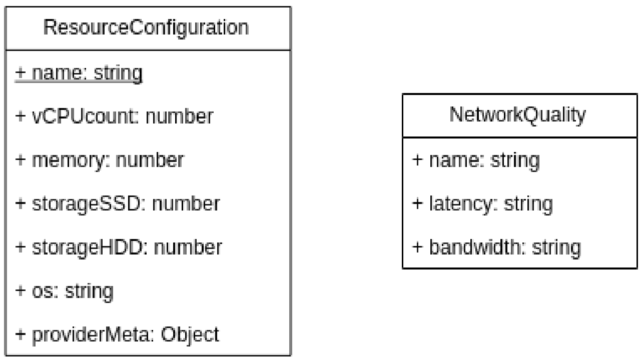


Figure 5.6: Infrastructure data model

5.2.6 Experiments Information Model

An experiment is represented by the *Experiment* class, depicted in Figure 5.7. An experiment is associated with a topology, represented by the *Topology* class. A topology of experiment is a graph composed of instances of *Node* class, presenting graph's nodes, with a root stored under *structure*. Instance of the *Node* class is a representation of MEC component. The edges of the graph represent blockchain interactions among the *Node*'s instances (MEC components). The *Node* class is the same as *DPNode*, however, there are *softwareArtefacts* deployed to the MEC components, represented by instances of *Node*, and there is a known infrastructure specified via *container* of the *Node* class. *NetworkQuality* as an association class representing quality of a network connection between two *Node* instances is also considered. Therefore, the topology of an experiment is the same as a deployment pattern, however, software artefacts are deployed to the *Node* instances (MEC components) and there is a known infrastructure of the topology. A *Node* instance is globally identified via its *id*, while the *name* attribute must be unique per *Experiment* instance. Beside the *structure*, we store *specificationLang* for the topology, that indicates the deployment language (TOSCA [13], AWS Cloud Formation [4], etc.) used to specify the topology. While the specification itself is stored in the *specification* attribute.
Furthermore, we store reference to a *DeploymentPattern* in the *Experiment* class. That's the deployment pattern, which is related the topology of the experiment. By the definition of the model, it implies that a single deployment pattern might have multiple topologies. But we always have to ensure that topology's structure matches the structure of the

deployment pattern, related to the topology. Since that is not constrained on the design level, we have to assure that in a service layer. Please note that we decided to store the same structure twice (once as a structure of a deployment pattern and the second time as structure of the topology) to make it easier for developers to see an exact topology which has been/is going to be benchmarked within an experiment.

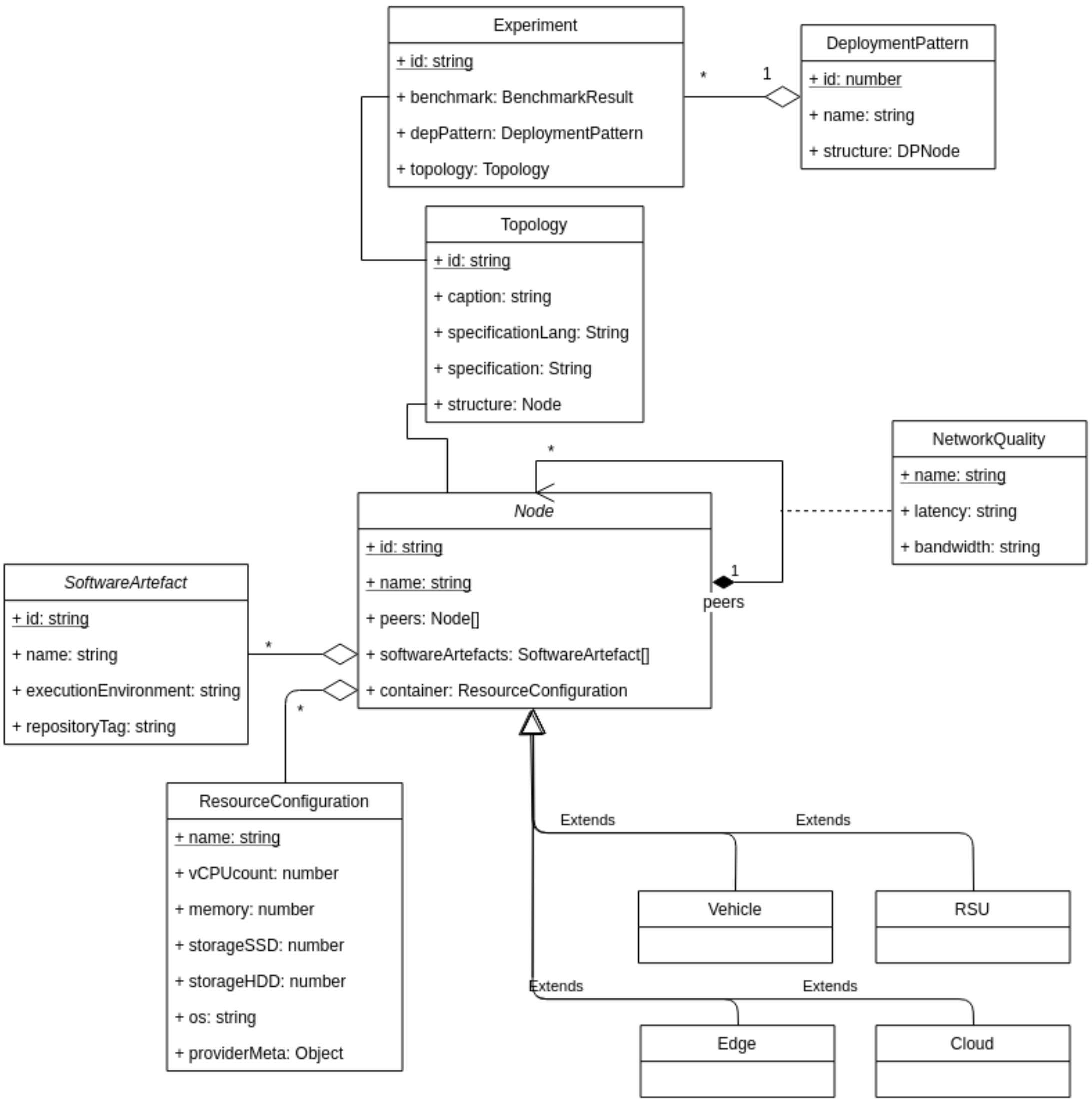


Figure 5.7: Experiment data model

5.2.7 Specification of benchmark result

When an experiment is benchmarked, a result represented as *BenchmarkResult* in the model depicted in Figure 5.8 is obtained. A *BenchmarkResult* is a set of quality metrics (see Section 3.3) which have been measured when benchmarking the experiment. A *QualityMetric* is an abstract class identified via its *id*, having a *description* explaining the metric. The subclasses of *QualityMetric* are concrete quality metrics. Let's look at the metrics (*QualityMetric* subclasses) supported by the Experiments Knowledge Service:

- *DataExchangeAnalysis*: This is the accepted transaction rate and time metric. Within a benchmark multiple transactions have been created and some results have been achieved (*txResults*). We count the transactions, which have been accepted (*acceptedTxCount* attribute) and which failed (*failedTxCount*). If a transaction has been accepted we store its metadata in *data* attribute of *TransactionResult*. Otherwise, we store an error message in *errorMsg*. *TransactionAnalysisWrapper* stores *acceptanceDuration* and *creationDuration* meaning how long did it take to accept and create the transaction. Attribute *payloadData* is an object, which represents a payload of an interaction, carried out via the transaction. The *DataExchangeAnalysis* class references a *Node* class, which represents an instance of *Node* where the transactions have been created.
- *ResourceUtilization*: *cpuUtil* is a number in percentage stating the utilization of a CPU core, *memoryUtil* presents how many MBs of RAM have been consumed when running a benchmark. The *nodeRef* is a *Node*' instance in which this metric have been measured.
- *SynchronisationState*: this is the Synchronization State metric. It references *Node*' instances via the *nodesOutOfSync* field, which lost the synchronization with the rest of a blockchain network during execution of a benchmark.

The data model is extendable. If a new quality metric is introduced, new subclass has to be defined.

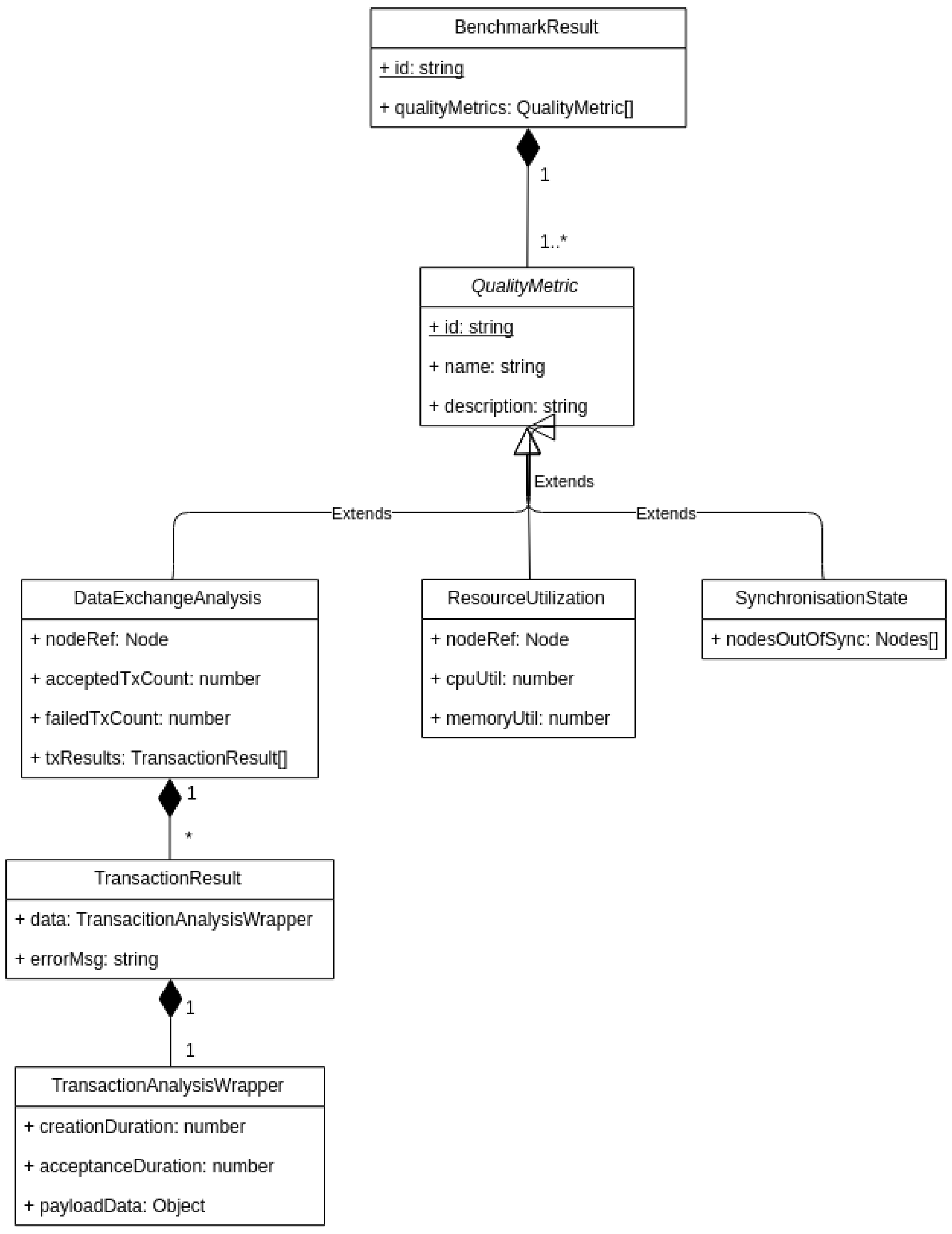


Figure 5.8: Benchmark result data model

5.3 Service Operations

In this section we explain design and exposed operations of services, which compose the Experiment Knowledge Service.

5.3.1 Deployment Pattern Service

Via *Deployment Pattern Service* a developer can manage all deployment patterns stored by the Experiments Knowledge Service. *Deployment Pattern Service* exposes a set of REST endpoints to enable the developer to interact with the service. Those endpoints are listed in Table 5.1. Figure 5.9 presents a traditional three-tier architecture used in the implementation of the service. We use a Neo4J [42] database to store the deployment patterns. We have chosen the Neo4J because it provides a clear visual overview of the interacting MEC components involved in a deployment pattern for the developer.

Table 5.1: Operations of Deployment Pattern Service

Endpoint	**Input**	**Output**	**Description**
POST /dep_pattern	DeploymentPattern	DeploymentPattern	Create a new DeploymentPattern in the service.
GET /dep_pattern	-	DeploymentPattern[]	Retrieves all DeploymentPattern stored in the service.
GET /dep_pattern/:id	string	DeploymentPattern	Retrieves a DeploymentPattern with the id stored in the service.
DELETE /dep_pattern/:id	string	boolean	Deletes a DeploymentPattern with the id from the service.

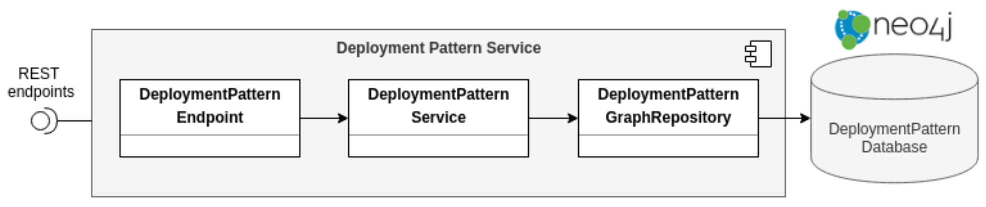

neo4j
Deployment Pattern Service
REST endpoints
DeploymentPattern Endpoint
DeploymentPattern Service
DeploymentPattern GraphRepository
DeploymentPattern Database

Figure 5.9: Architectural overview of Deployment Pattern Service

5.3.2 Software Artefacts Service

The responsibility of *Software Artefact Service* is to manage metadata of software artefacts, used in benchmarks, stored by the Experiments Knowledge Service. A set of REST endpoints (listed in Table 5.2) is provided by the service to enable interactions with the service. The service uses a collection in the Experiment Knowledge Service's MongoDB [40] database to store metadata concerning the software artefacts. However, the artefacts themselves are stored in an external repository of artefacts (e.g. DockerHub [16]). This service implements a three-tier architecture, depicted in Figure 5.10.

Table 5.2: Operations of Software Artefact Service

Endpoint	**Input**	**Output**	**Description**
POST /software_artefact	SoftwareArtefact	SoftwareArtefact	Create a new SoftwareArtefact in the service.
GET /software_artefact	-	SoftwareArtefact[]	Retrieves all SoftwareArtefact stored in the service.
GET /software_artefact/:id	string	SoftwareArtefact	Retrieves a SoftwareArtefact with the id stored in the service.
DELETE /software_artefact/:id	string	boolean	Deletes a SoftwareArtefact with the id from the service.

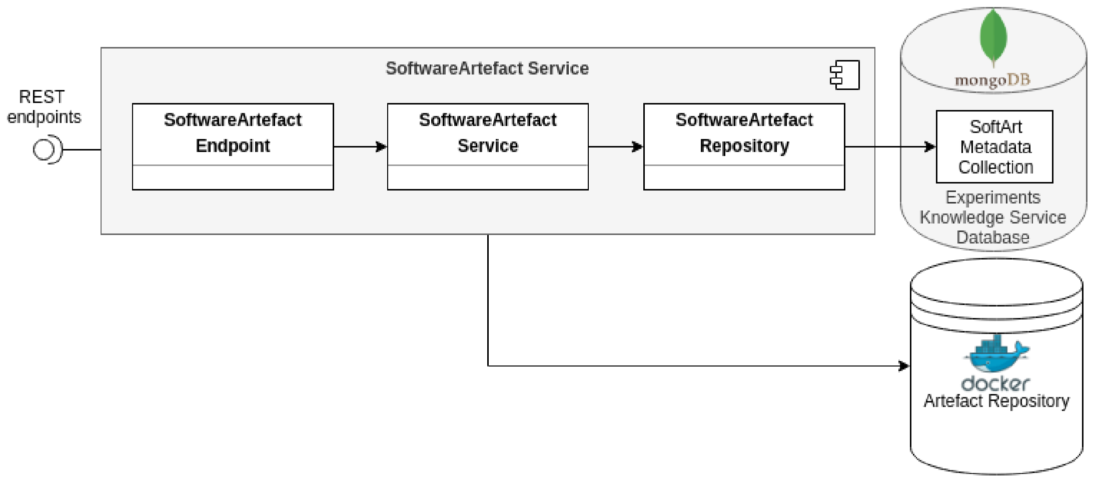

SoftwareArtefact Service
REST endpoints
SoftwareArtefact Endpoint
SoftwareArtefact Service
SoftwareArtefact Repository
mongoDB
SoftArt Metadata Collection
Experiments Knowledge Service Database
docker
Artefact Repository

Figure 5.10: Architectural overview of Software Artefact Service

5.3.3 Infrastructure Service

Infrastructure Service manages metadata about infrastructure resources and quality of network connections, which have been used for a benchmark, stored by the Experiments Knowledge Service. The Infrastructure Service exposes REST endpoints, listed in Table 5.3, to enable management of the metadata. Only metadata of the resources are stored and the resources themselves are provided by an external provider. The metadata are stored in a collection of the MongoDB database. We have followed a three tier architecture when implementing this service, as it's illustrated in Figure 5.11.

Table 5.3: Operations of Infrastructure Service

Endpoint	**Input**	**Output**	**Description**
POST /resource_config	ResourceConfiguration	ResourceConfiguration	Create a new ResourceConfiguration in the service.
POST /net_quality	NetworkQuality	NetworkQuality	Create a new NetworkQuality in the service.
GET /resource_config	-	ResourceConfiguration[]	Retrieves all ResourceConfiguration stored in the service.
GET /net_quality	-	NetworkQuality[]	Retrieves all NetworkQuality stored in the service.
GET /resource_config/:id	string	ResourceConfiguration	Retrieves a ResourceConfiguration with the id stored in the service.
GET /net_quality/:name	string	NetworkQuality	Retrieves a NetworkQuality with the name stored in the service.
DELETE /resource_config/:id	string	boolean	Deletes a ResourceConfiguration with the id from the service.
DELETE /resource_config/:id	string	boolean	Deletes a NetworkQuality with the id from the service.

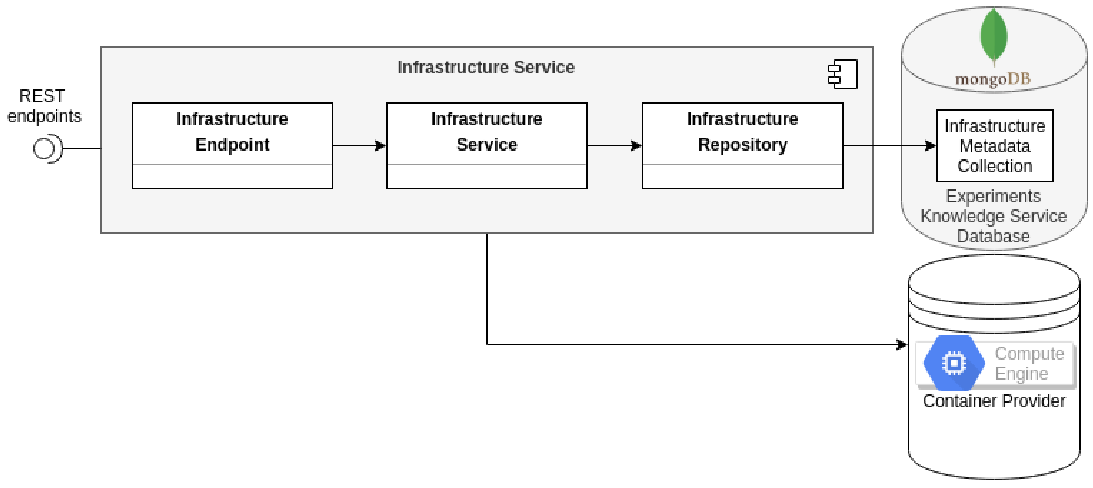


Figure 5.11: Architectural overview of Infrastructure Service

5.3.4 Blockchain Benchmark DaaS

The Blockchain Benchmark DaaS provides REST endpoints, listed in Table 5.4, to enable management of experiments stored in the Experiments Knowledge Service. As depicted in Figure 5.12, the service internally manages experiments' topologies and eventually benchmarks achieved by the experiments. The experiment, topology and achieved benchmark are stored in separate collections of the MongoDB database.
This service requires functionality provided by *DeploymentPatternService*, *SoftwareArtefactService* and *InfrastructureService* because, as shown on Figure 5.2, an experiment object is associated with a deployment pattern. Topology of the experiment is composed of *Node* instances, which are associated with software artefacts and infrastructure.
If a developer utilized our benchmark framework (refer to Section 4.3) for running the benchmarks. Then the benchmarks produced by the framework can be converted by *Results Parser* to the format accepted by the endpoints of *The Blockchain Benchmark DaaS*. The *ResultsParser* is an utility application, which has been developed within this thesis and is available in GitHub repository: `https://github.com/rdsea/blockchainbenmarkservice`.

Table 5.4: Operations of Blockchain Benchmark DaaS

Endpoint	Input	Output	Description
POST /experiment	Experiment	Experiment	Create a new Experiment in the service.
GET /experiment	-	Experiment[]	Retrieves all Experiments stored in the service.
GET /experiment/:id	string	Experiment	Retrieves a Experiment with the id stored in the service.
DELETE /experiment/:id	string	boolean	Deletes a Experiment with the id from the service.

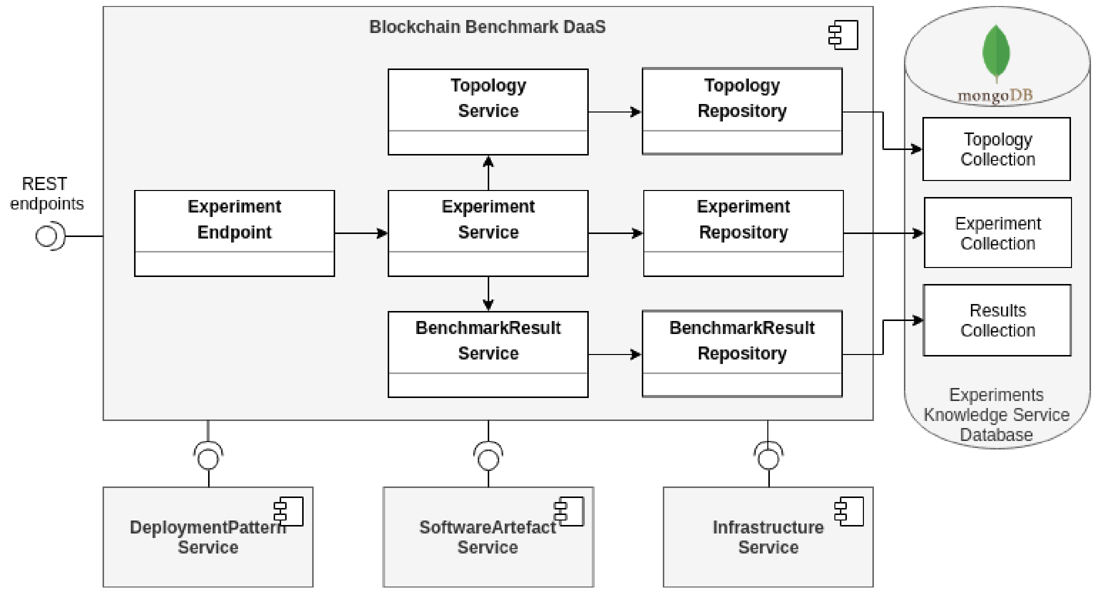


Figure 5.12: Architectural overview of Blockchain Benchmark DaaS

5.3.5 Recommendation Service

The workflow of *Recommendation Service* when giving recommendations involves the following three key steps:

1. Load model of a blockchain-based application in MEC, for which a developer wants to obtain recommendation, and preferences on quality metrics.

2. Find one among all deployment patterns stored by the Experiments Knowledge Service, which is most similar to topology of the application.
3. Look for a benchmarked experiment, which achieved best preferred quality metrics and is associated with the most similar deployment pattern.

A developer can submit a specification of the model to the *Recommendation Service* via the REST endpoints, listed in Table 5.5. The developer can set priorities, which describe his/her preferences on quality metrics. Those are submitted as parameters of an url query in form of numbers, higher number means higher priority. Those priorities will be considered by the service when looking for a benchmarked experiment, which achieved quality metrics according to the priorities.
Currently the service supports two types of specifications of the model:

- Model specified via an instance of the *Topology* class (see Section 5.2.6): Since the purpose of the Recommendation Service is to give recommendations about deployments of blockchain artefacts to MEC components involved in the model and to propose an infrastructure for the application. The submitted topology doesn't have to include any *BlockchainArtefact* instances in *softwareArtefacts* attribute of involved *Node* instances. Neither the *container* field in the *Node* instances, nor *latency* and *bandwidth* of *NetworkQuality* have to be provided.
- Topology and Orchestration Specification for Cloud Applications (TOSCA) [7] specification of the model: We have decided to use TOSCA, because via TOSCA the developer can specify what nodes are involved in an application's topology, what's underlying infrastructure of the application and what software artefacts are deployed to nodes of the topology. Since the Recommendation Service has to find a deployment pattern stored by the Experiments Knowledge Service, which is most similar to topology of the application, and deployment pattern is a graph of MEC components, the model in TOSCA has to contain the MEC components as well. Therefore, we provide TOSCA definitions of node types representing the MEC components. Figure 5.14 depicts the node types, these are derived from node types defined by Cloudify [11]. The node types, which are interacting among each other must be connected via *filip.relationships.nodes_network* connection type. Please note that those node types are custom TOSCA containers implementing properties we considered in infrastructure's resources (see Section 5.2.5). Those properties don't have to be included in the submitted model. The same holds for *BlockchainArtefacts* contained in the node types of application's topology. The explanation why those properties don't have to be included is given in the previous item of this list. Internally, the service converts the TOSCA specification to an instance of the *Topology* class. That conversion from/to TOSCA representation is done by *TOSCATopologyAdapter*, depicted in Figure 5.13.

As soon as the model is loaded to the Recommendation Service, the next step is to find a most similar deployment pattern. We have to develop an algorithm to address that.

We know that both deployment pattern and application's topology are graphs, composed of MEC components representing nodes of the graph. Therefore, a trustworthy solution would be to find a largest common subgraph. Nevertheless, that problem is known to be NP-hard [1]. Thus, we aim to look for a more feasible approach.
In Listing 5.1 we propose our idea, described in pseudo-code, used to find the most similar deployment pattern. Firstly, it iterates over all stored deployment patterns and splits them to interaction pairs. An interaction pair is a pair of nodes (representing MEC components) connected by an edge in the graph. Consider following example: a deployment pattern "vehicle - RSU - edge node - RSU - vehicle", has the following list of interaction pairs: (vehicle, RSU), (RSU, edge node), (edge node, RSU), (RSU, vehicle). The deployment pattern for which we find the largest number of matching interaction pairs with the topology of the submitted application model is considered to be the most similar one. This algorithm is implemented in *DeploymentPatternMatcher*, depicted in Figure 5.13.
When the most similar deployment pattern is obtained, the Recommendation Service can proceed to the second step of the workflow. In the second step, the service looks among all experiments associated with the deployment pattern to find one, which benchmarks have shown best results according to the preferred quality metrics, set by developer. That is done by employing respective MongoDB queries.
When the experiment is found then its topology is returned on output. Eventually, if the developer submitted a topology in TOSCA, then TOSCA representation of the experiment's topology is returned. The service adds software artefacts represented by *filip.nodes.SoftwareArtefact* to nodes of the topology. That is specified via Cloudify relationship of type *cloudify.relationships.contained_in*. It would be advisable to propose an algorithm that maps the experiment's topology to the one submitted on input, such that the resulting topology could be further used instead of the submitted one. Such an algorithm is out of the scope of this thesis, but can be added within a future work.

Table 5.5: Operations of Recommendation Service

Endpoint	Input	Query Parameters	Output	Description
POST /recommendTopology	Topology	syncState, txAcceptRate, txAcceptTime, infRes	Topology	Return a topology associated with an experiment, which achieved best benchmarks according to preferred quality metrics set via URL query. This endpoint accepts a Topology as defined in the Experiments Information Model (see Section 5.2.6) on input.
POST /recommendTopologyTOSCA	string	syncState, txAcceptRate, txAcceptTime, infRes	string	Return a TOSCA representation of a topology associated with an experiment, which achieved best benchmarks according to preferred quality metrics set via URL query. This endpoint accepts a topology specified in TOSCA on input. We elaborate on an exact format in this section.

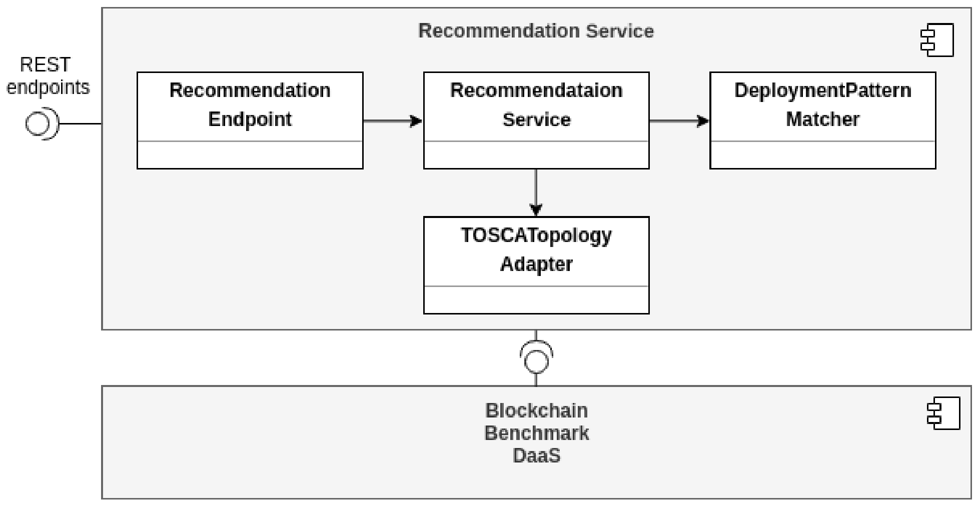


Figure 5.13: Architectural overview of Recommendation Service

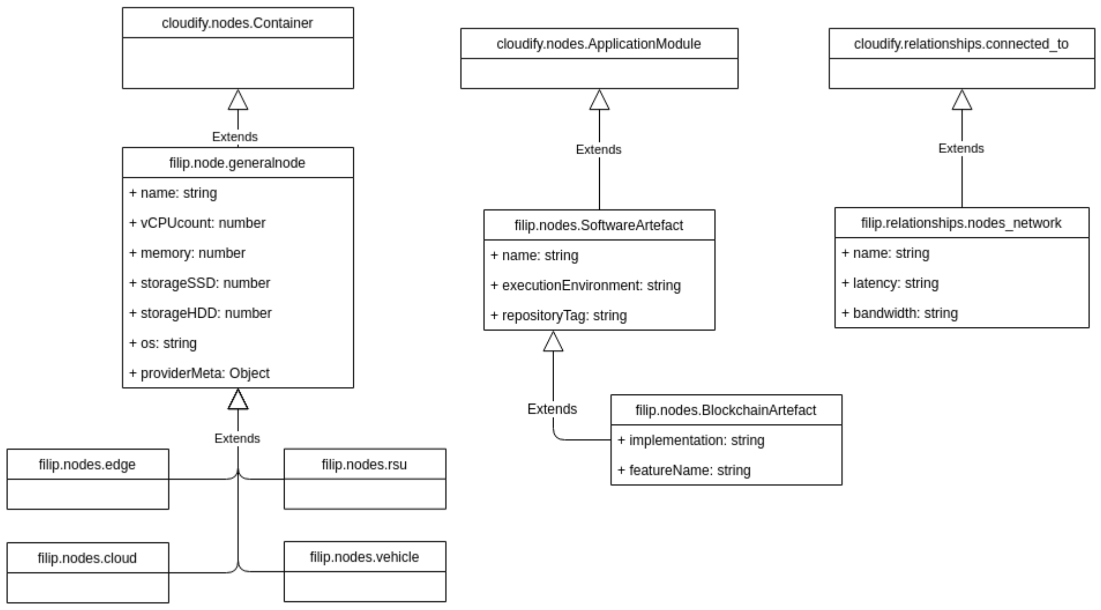


Figure 5.14: Definitions of infrastructure and software artefacts in TOSCA Cloudify

Listing 5.1: Finding most similar deployment pattern

```
Node[] deployment_patterns;
Node input;

findSimilarDepPattern(deployment_patterns, input) {
  InteractionPair[] inputPairs = splitToInteractionPairs(input);
  InteractionPair[] patternPairs;

  maxMatchesCount = 0;
  bestPattern = null;

  for each pattern in deployment_patterns {

    matchesCount = 0;
    patternPairs = splitToInteractionPairs(pattern);
    boolean[] usedPair = new boolean[patternPairs.length];

    for each inputPair in inputPairs {
      j = 0;
      for each patternPair in patternPairs {
        if ( inputPair.equals(patternPair) && !usedPair[j] ) {
          usedPair[j] = true;
          matchesCount++;
          break;
        }
        j++;
      }
    }
    if (maxMatchesCount < matchesCount) {
      maxMatchesCount = matchesCount;
      bestPattern = pattern;
    }
  }
  return bestPattern;
}

InteractionPair[] splitToInteractionPairs(Node node) {
  if (isVisited(node)) {
    return;
  }
  markNodeAsVisited(node);
  for each peer in node.peers {
    NodeTypePair pair = (node.type, peer.type);
    if (peer.peers == null || peer.peers.length == 0) {
      return [ pair ];
    } else {
      return splitToInteractionPairs(peer).push ( [ pair ] );
    }
  }
}

class InteractionPair {
  NodeType a;
  NodeType b;

  boolean equals(InteractionPair that) {
    return (this.a == that.a && this.b == that.b) ||
      (this.b == that.a && this.a == that.b);
  }
}
```

5.4 Prototype

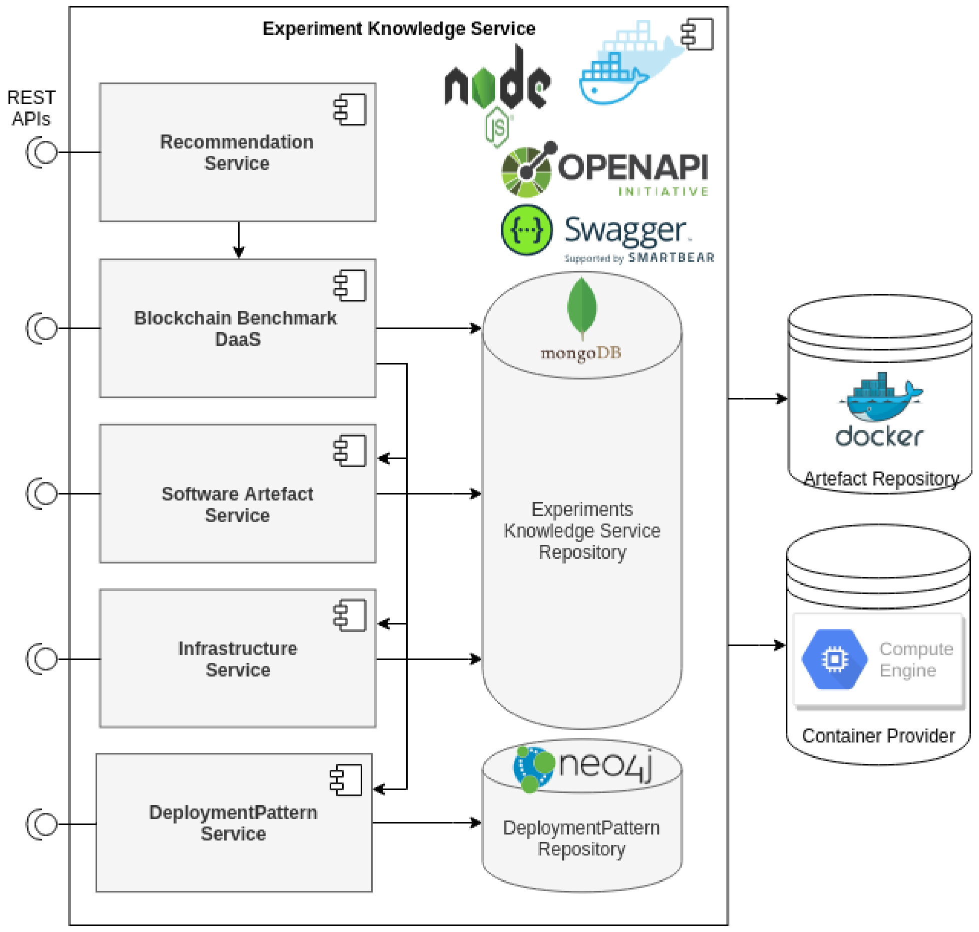

Experiment Knowledge Service
REST APIs
Recommendation Service
Blockchain Benchmark DaaS
Software Artefact Service
Infrastructure Service
DeploymentPattern Service
OPENAPI INITIATIVE
Swagger Supported by SMARTBEAR
mongoDB
Experiments Knowledge Service Repository
neo4j
DeploymentPattern Repository
docker
Artefact Repository
Compute Engine
Container Provider

Figure 5.15: Architectural overview of Experiments Knowledge Service prototype

The *Experiment Knowledge Service* prototype encapsulates and implements all operations described in the previous section. An architectural overview of the prototype is depicted in Figure 5.15. The prototype has been developed in Typescript [56], runs in a NodeJS [43] environment and can be executed inside a docker container. It uses a single

MongoDB and a Neo4J database for the repository layer. The Experiments Knowledge Service exposes a set of REST APIs to manipulate the data (see Section 5.3). The REST endpoints are documented with Swagger.io [53] (according to OpenAPI Specification 2.0 [33]).
The *Blockchain Benchmark*, *Infrastructure* and *Software Artefact* services store their data in the MongoDB database, each of the services in a separate collection. While the *Deployment Pattern Service* is utilizing the Neo4J database.
The implementation and documentation of the prototype is published in the GitHub repository: `https://github.com/rdsea/blockchainbenmarkservice`. The definition of required TOSCA node types is provided in
experiments_knowledge_service/tosca_node_types.yaml file in the repository.

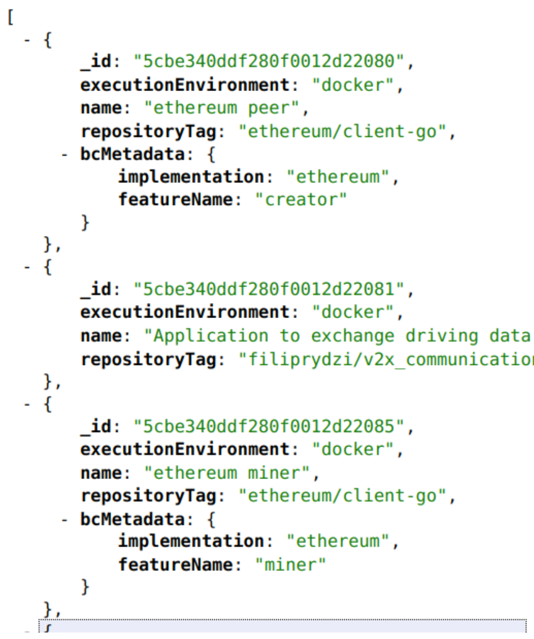
5.5 Examples

5.5.1 Search benchmarking information

Let's assume the developer wants to use Ethereum blockchain to enable interactions in his/her application. He/she wants to use the Experiment Knowledge Service for the recommendations (see Section 5.3.5). Since the developer wants to use Ethereum, he/she is interested in whether our Experiment Knowledge Service stores some benchmarks, which utilized Ethereum. To determine that, he/she invokes /software_artefact endpoint (see Section 5.3.2) to obtain a list of all software artefacts used in the benchmarks. Figure 5.16 depicts a subset of returned software artefacts. Now the developer can see that the Experiments Knowledge Service might contain some benchmarks, which utilized Ethereum blockchain. Therefore, the developer can proceed to actually obtaining recommendations. There are plenty of similar use cases, which involve searching some benchmarking information. Those might help the developers in the same way, as we have shown it for this use case.

5.5.2 Search benchmarking pattern

A developer might be interested in whether any patterns stored by the service occur in a system model of his/her blockchain-based application in MEC. If some patterns from the service match with the application's model. Then the developer can conclude that he/she can use the Experiments Knowledge Service to obtain recommendations. Otherwise, he/she has to use a benchmark framework to benchmark the interactions in his application. To list all deployment patterns, the developer can invoke /dep_pattern endpoint (see Section 5.3.1). There is another possibility that the developer connect directly to the Neo4J database, execute the following query `MATCH (n) RETURN n` and obtain a visual overview about the deployment patterns (as depicted in Figure 5.17).



```
[
  - {
        _id: "5cbe340ddf280f0012d22080",
        executionEnvironment: "docker",
        name: "ethereum peer",
        repositoryTag: "ethereum/client-go",
      - bcMetadata: {
            implementation: "ethereum",
            featureName: "creator"
        }
    },
  - {
        _id: "5cbe340ddf280f0012d22081",
        executionEnvironment: "docker",
        name: "Application to exchange driving data within V2X",
        repositoryTag: "filiprydzi/v2x_communication"
    },
  - {
        _id: "5cbe340ddf280f0012d22085",
        executionEnvironment: "docker",
        name: "ethereum miner",
        repositoryTag: "ethereum/client-go",
      - bcMetadata: {
            implementation: "ethereum",
            featureName: "miner"
        }
    },
  - {
```

Figure 5.16: A subset of software artefacts stored by the Experiment Knowledge Service

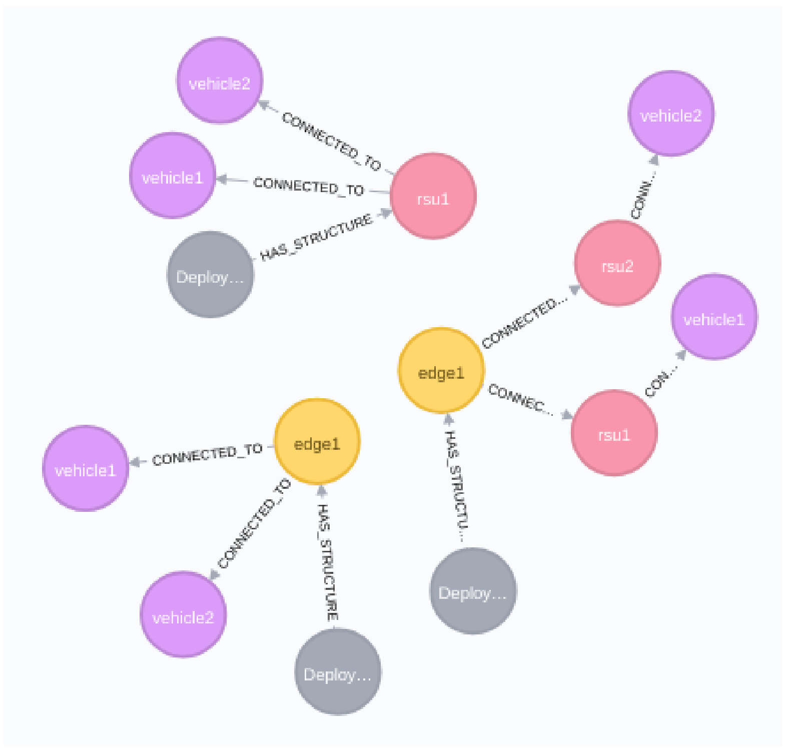


Figure 5.17: Deployment patterns, as stored in Neo4J database

5.5.3 Obtain recommendations

Suppose a developer wants to utilize blockchain for interactions among the instances of his application, which are running across a MEC infrastructure. However, he/she doesn't know which deployment of blockchain artefacts to involved MEC would be most suitable for him/her. Let's assume for the developer, the highest transaction acceptance rate and shortest acceptance time are most important quality metrics. Therefore, the developer can benefit from the Experiment Knowledge Service by obtaining recommendations about the deployment.
The developer has TOSCA specification of an application's model, depicted in Figure 5.18. Figure 5.19 shows visualization of the specification in Cloudify Composer [12]. He/she executes POST request to /recommendTopologyTOSCA endpoint (see Section 5.3.5) with the following parameters: `txAcceptRate=2&txAcceptTime=1` and the TOSCA specification as the request's payload. The Experiment Knowledge Service carries out the operations, which are involved in obtaining recommendations (those are explained in Section 5.3.5) and return TOSCA specification containing deployed blockchain artefacts and properties for underlying infrastructure. The returned TOSCA specification is depicted in Figures 5.20 and 5.21. A visualization via Cloudify Composer [12] is illustrated in Figure 5.22. In this case the returned TOSCA specification can be directly used by the developer to deploy the blockchain artefacts into topology of the application.

```
node_templates:
  vehicle1:
    type: filip.nodes.vehicle
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      infrastructure: ''
    relationships:
      - target: rsu1
        type: filip.relationships.nodes_network
  vehicle2:
    type: filip.nodes.vehicle
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      infrastructure: ''
    relationships: []
  LaneChangeApp:
    type: filip.nodes.SoftwareArtefact
    relationships:
      - type: cloudify.relationships.contained_in
        target: vehicle1
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      name: ''
  rsu1:
    type: filip.nodes.rsu
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      infrastructure: ''
    relationships:
      - target: vehicle2
        type: filip.relationships.nodes_network
  LaneChangeApp1:
    type: filip.nodes.SoftwareArtefact
    relationships:
      - type: cloudify.relationships.contained_in
        target: rsu1
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      name: ''
      executionEnvironment: ''
  LaneChangeApp2:
    type: filip.nodes.SoftwareArtefact
    relationships:
      - type: cloudify.relationships.contained_in
        target: vehicle2
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      executionEnvironment: ''
```

Figure 5.18: TOSCA representation of an application system model, used as input for Recommendation Service

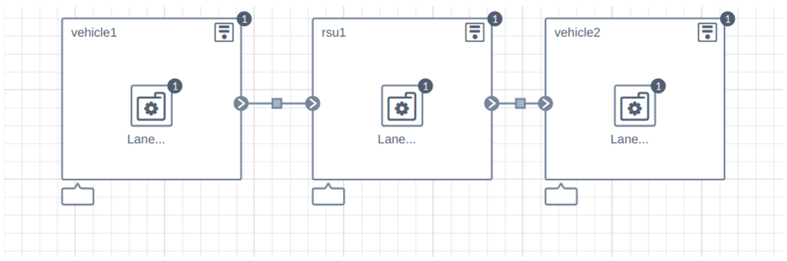
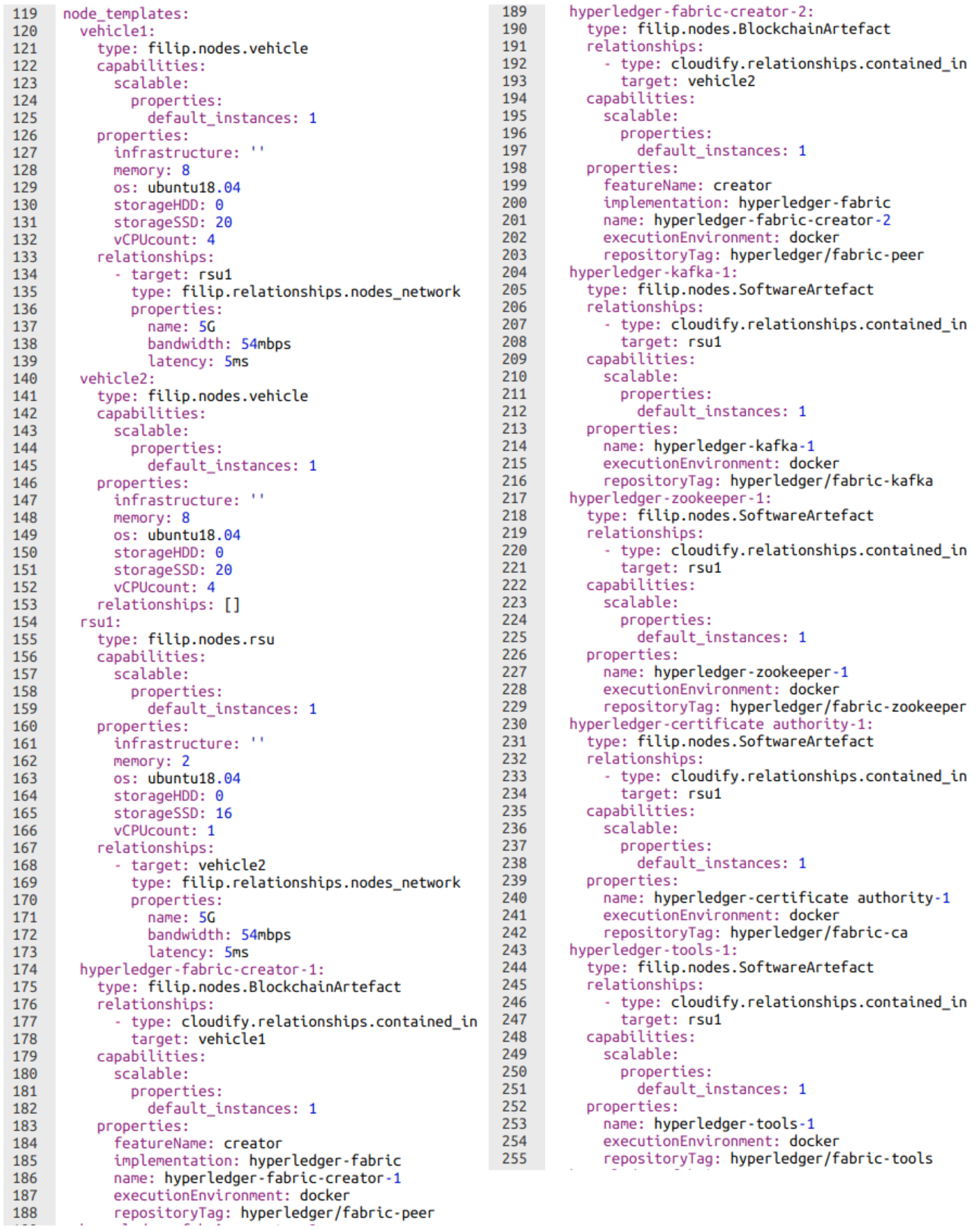


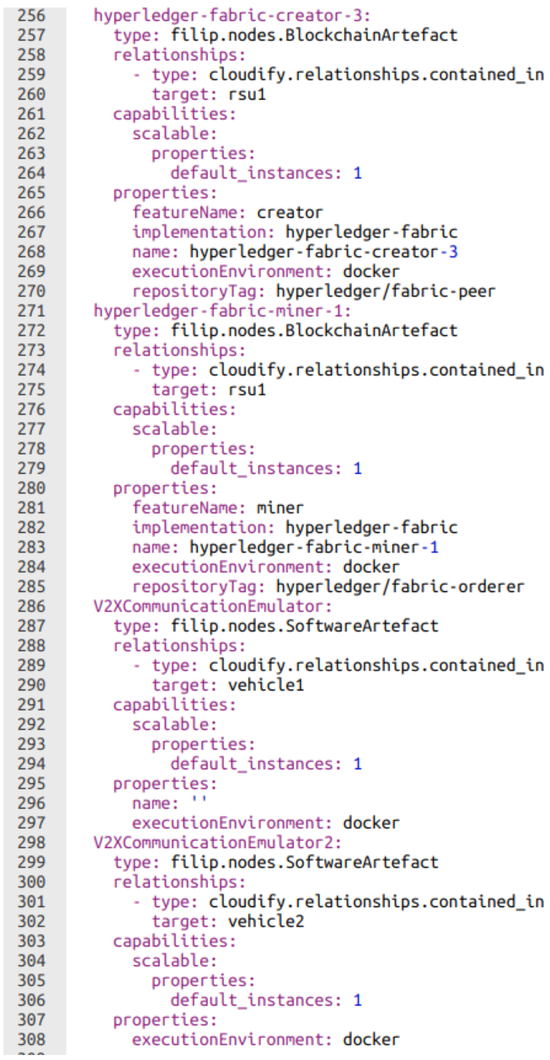
Figure 5.19: Input for Recommendation Service, as depicted by Cloudify Composer [12]



```yaml
node_templates:
  vehicle1:
    type: filip.nodes.vehicle
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      infrastructure: ''
      memory: 8
      os: ubuntu18.04
      storageHDD: 0
      storageSSD: 20
      vCPUcount: 4
    relationships:
      - target: rsu1
        type: filip.relationships.nodes_network
        properties:
          name: 5G
          bandwidth: 54mbps
          latency: 5ms
  vehicle2:
    type: filip.nodes.vehicle
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      infrastructure: ''
      memory: 8
      os: ubuntu18.04
      storageHDD: 0
      storageSSD: 20
      vCPUcount: 4
    relationships: []
  rsu1:
    type: filip.nodes.rsu
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      infrastructure: ''
      memory: 2
      os: ubuntu18.04
      storageHDD: 0
      storageSSD: 16
      vCPUcount: 1
    relationships:
      - target: vehicle2
        type: filip.relationships.nodes_network
        properties:
          name: 5G
          bandwidth: 54mbps
          latency: 5ms
  hyperledger-fabric-creator-1:
    type: filip.nodes.BlockchainArtefact
    relationships:
      - type: cloudify.relationships.contained_in
        target: vehicle1
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      featureName: creator
      implementation: hyperledger-fabric
      name: hyperledger-fabric-creator-1
      executionEnvironment: docker
      repositoryTag: hyperledger/fabric-peer
```

```yaml
  hyperledger-fabric-creator-2:
    type: filip.nodes.BlockchainArtefact
    relationships:
      - type: cloudify.relationships.contained_in
        target: vehicle2
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      featureName: creator
      implementation: hyperledger-fabric
      name: hyperledger-fabric-creator-2
      executionEnvironment: docker
      repositoryTag: hyperledger/fabric-peer
  hyperledger-kafka-1:
    type: filip.nodes.SoftwareArtefact
    relationships:
      - type: cloudify.relationships.contained_in
        target: rsu1
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      name: hyperledger-kafka-1
      executionEnvironment: docker
      repositoryTag: hyperledger/fabric-kafka
  hyperledger-zookeeper-1:
    type: filip.nodes.SoftwareArtefact
    relationships:
      - type: cloudify.relationships.contained_in
        target: rsu1
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      name: hyperledger-zookeeper-1
      executionEnvironment: docker
      repositoryTag: hyperledger/fabric-zookeeper
  hyperledger-certificate authority-1:
    type: filip.nodes.SoftwareArtefact
    relationships:
      - type: cloudify.relationships.contained_in
        target: rsu1
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      name: hyperledger-certificate authority-1
      executionEnvironment: docker
      repositoryTag: hyperledger/fabric-ca
  hyperledger-tools-1:
    type: filip.nodes.SoftwareArtefact
    relationships:
      - type: cloudify.relationships.contained_in
        target: rsu1
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      name: hyperledger-tools-1
      executionEnvironment: docker
      repositoryTag: hyperledger/fabric-tools
```

Figure 5.20: TOSCA representation of an application system model, as output by the Recommendation Service



```
  hyperledger-fabric-creator-3:
    type: filip.nodes.BlockchainArtefact
    relationships:
      - type: cloudify.relationships.contained_in
        target: rsu1
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      featureName: creator
      implementation: hyperledger-fabric
      name: hyperledger-fabric-creator-3
      executionEnvironment: docker
      repositoryTag: hyperledger/fabric-peer
  hyperledger-fabric-miner-1:
    type: filip.nodes.BlockchainArtefact
    relationships:
      - type: cloudify.relationships.contained_in
        target: rsu1
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      featureName: miner
      implementation: hyperledger-fabric
      name: hyperledger-fabric-miner-1
      executionEnvironment: docker
      repositoryTag: hyperledger/fabric-orderer
  V2XCommunicationEmulator:
    type: filip.nodes.SoftwareArtefact
    relationships:
      - type: cloudify.relationships.contained_in
        target: vehicle1
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      name: ''
      executionEnvironment: docker
  V2XCommunicationEmulator2:
    type: filip.nodes.SoftwareArtefact
    relationships:
      - type: cloudify.relationships.contained_in
        target: vehicle2
    capabilities:
      scalable:
        properties:
          default_instances: 1
    properties:
      executionEnvironment: docker
```

Figure 5.21: TOSCA representation of an application system model, as output by the Recommendation Service cont.

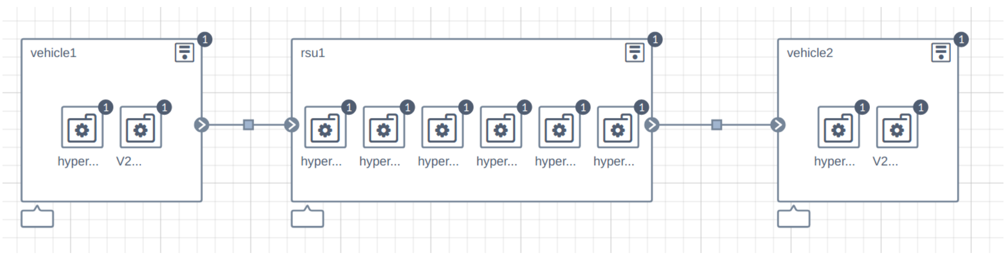


Figure 5.22: Output from Recommendation Service, as depicted by Cloudify Composer [12]

5.6 Summary

In this chapter we propose and implement a prototype of Experiments Knowledge Service, which manages data related to benchmarks. It's main purpose is to provide knowledge gathered from benchmarks to developers of blockchain-based applications in MEC. Such that the developers can reuse the knowledge from benchmarks for design of their applications.

CHAPTER 6

Conclusion and Future Work

6.1 Conclusion

In this thesis we contribute a blockchain benchmark framework, which is able to benchmark blockchain interactions among MEC components. We have emphasized a requirement of introducing the framework by discussing challenges faced by developers when implementing blockchain-based application in MEC. There are numerous works and studies centered around utilization of blockchain in MEC and benchmarking frameworks able to evaluate different blockchain solutions. However, none of the related works focuses on benchmarking blockchain interactions among MEC components and, therefore, don't addresses our requirements. To obtain a better understanding of the challenges, we have reused scenarios in the V2X domain and have looked on those scenarios from a developer's point of view. We have assumed that the developer implements an application, which addresses one of the scenarios and interacts over blockchain. Based on that, we have identified different interactions among MEC components, which might arise for the scenarios. Furthermore, we have identified blockchain features and have mapped them to executable blockchain artefacts, which can be deployed to MEC components by the developers, to enable the interactions.
A prototype of the framework has been implemented. To demonstrate flexibility of the framework, 324 experiments, based on the identified interactions, have been created and benchmarked. However, we haven't created all those experiments manually, since the framework implements a feature, which enables to generate and benchmark experiments automatically based on a submitted specification. Deriving from the benchmarks' result we have elaborated on what insights do the results bring for the developers and how does the framework can help towards addressing the challenges.
In most cases, our benchmarks have shown a direct proportion between the underlying infrastructure, used for the benchmarks and observed quality metrics. Although, we witness a surprising found out that an interaction employing RSU instead of an edge

node has performed better. In general, for the considered scenarios Hyperledger-Fabric blockchain outperformed Ethereum in terms of the quality metrics.
Furthermore, we introduce an Experiment Knowledge Service, which manages data related to the benchmarks. The developers can benefit from the service by obtaining information about infrastructures, software artefacts or topologies which have been utilized in the benchmarks. Furthermore, the developers can obtain recommendations about deployment of blockchain artefacts to MEC components involved in a blockchain-based application, and infrastructure for the application without having to run the benchmarks. The main purpose of the service is to reuse knowledge gathered by benchmarks to help developers during design phase of the application. In our work we have shown concrete examples on how developers can benefit from the service. A prototype of the service has been developed within this thesis as well.

6.2 Future Work

There are numerous opportunities for future works in our benchmark framework. Currently, it supports two well-known blockchain systems: Ethereum and Hyperledger-Fabric, however it can be extended to work with other blockchain systems as well. We have described quality metrics which are considered and measured by the framework. Those metrics measure only some basic qualities thus it would be beneficial if the framework would support other metrics e.g. fault tolerance i.e. how does a system behave on a blockchain's node failure. A developer can implement other emulators of blockchain interactions than the one we used, which emulates interactions in form of exchanging small pure text data.

The Experiment Knowledge Service can extended as well. An interesting feature would be to enable autonomous collaboration of the benchmark framework with the Experiment Knowledge Service. When an experiment is benchmarked by a benchmark framework, then the experiment and achieved benchmarks can be automatically stored to the service. Another very interesting feature would be that if a developer adds blockchain artefacts of a new blockchain implementation to the Experiments Knowledge Service, then the experiments stored in the service would be automatically benchmarked again while utilizing the new blockchain implementation. There are numerous use cases which could be addressed and implemented by adding the collaboration between the framework and the service.
A future work improving the algorithm used to find the most similar deployment pattern implemented in Recommendation Service (see Section 5.3.5) would be appreciated as well. Furthermore, in Section 5.3.5 we left a possibility for future work open. That is the algorithm mapping a topology of experiment to the one submitted to the Recommendation Service.

Bibliography

[1] Tatsuya Akutsu. A polynomial time algorithm for finding a largest common subgraph of almost trees of bounded degree. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 76(9):1488–1493, 1993.

[2] Mani Amoozadeh, Arun Raghuramu, Chen-Nee Chuah, Dipak Ghosal, H Michael Zhang, Jeff Rowe, and Karl Levitt. Security vulnerabilities of connected vehicle streams and their impact on cooperative driving. *IEEE Communications Magazine*, 53(6):126–132, 2015.

[3] ASFINAG. Digital section toll. `https://www.asfinag.at/toll/route-and-digital-section-toll/digital-section-toll/`. [Online; accessed 20-February-2019].

[4] AWS. Aws cloudformation. `https://aws.amazon.com/cloudformation/`. [Online; accessed 21-April-2019].

[5] Roberto Baldessari, Bert Bödekker, Matthias Deegener, Andreas Festag, Walter Franz, C Christopher Kellum, Timo Kosch, Andras Kovacs, Massimiliano Lenardi, Cornelius Menig, Timo Peichl, Matthias Röckl, Dieter Seeberger, Markus Straßberger, Hannes Stratil, Hans-Jörg Vögel, Benjamin Weyl, and Wenhui Zhang. Car-2-car communication consortium - manifesto. 01 2007.

[6] Caitlin Bettisworth, Matthew Burt, Alan Chachich, Ryan Harrington, Joshua Hassol, Anita Kim, Katie Lamoureux, Dawn LaFrance-Linden, Cynthia Maloney, David Perlman, et al. Status of the dedicated short-range communications technology and applications: Report to congress. Technical report, 2015.

[7] Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. Opentosca – a runtime for tosca-based cloud applications. In Samik Basu, Cesare Pautasso, Liang Zhang, and Xiang Fu, editors, *Service-Oriented Computing*, pages 692–695, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[8] P. Brody and V. Pureswaran. Device democracy: Saving the future of the internet of things. `http://www-935.ibm.com/services/us/gbs/thoughtleadership/internetofthings/`. [Online; accessed 08-April-2019].

[9] K. Christidis and M. Devetsikiotis. Blockchains and smart contracts for the internet of things. *IEEE Access*, 4:2292–2303, 2016.

[10] Paul Clements, Rick Kazman, Mark Klein, et al. *Evaluating software architectures*. Tsinghua University Press Beijing, 2003.

[11] Cloudify. Cloudify. `https://cloudify.co/`. [Online; accessed 21-March-2019].

[12] Cloudify. Cloudify composer. `https://docs.cloudify.co/4.5.0/developer/composer/`. [Online; accessed 23-April-2019].

[13] Cloudify. Tosca cloud orchestration for beginners. `https://cloudify.co/2015/07/21/what-is-TOSCA-cloud-application-orchestration-tutorial-cloudify.html`. [Online; accessed 21-April-2019].

[14] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, Vignesh Kalyanaraman, et al. Blockchain technology: Beyond bitcoin. *Applied Innovation*, 2(6-10):71, 2016.

[15] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1085–1100. ACM, 2017.

[16] DockerHub. Dockerhub. `https://hub.docker.com/`. [Online; accessed 22-April-2019].

[17] K. Dolui and S. K. Datta. Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing. In *2017 Global Internet of Things Summit (GIoTS)*, pages 1–6, June 2017.

[18] A. Dorri, S. S. Kanhere, R. Jurdak, and P. Gauravaram. Blockchain for iot security and privacy: The case study of a smart home. In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 618–623, March 2017.

[19] Ali Dorri, Salil S. Kanhere, and Raja Jurdak. Blockchain in internet of things: Challenges and solutions. *CoRR*, abs/1608.05187, 2016.

[20] Ali Dorri, Marco Steger, Salil S. Kanhere, and Raja Jurdak. Blockchain: A distributed solution to automotive security and privacy. *CoRR*, abs/1704.00073, 2017.

[21] Google Compute Engine. Google compute engine. `https://cloud.google.com/compute/`. [Online; accessed 22-April-2019].

[22] EtherAPI. Apis for the future: decentralized, trustless, secure. `https://etherapis.io/`. [Online; accessed 26-April-2019].

[23] Ethereum. Ethereum white paper. `https://github.com/ethereum/wiki/wiki/White-Paper`. [Online; accessed 27-March-2019].

[24] Miad Faezipour, Mehrdad Nourani, Adnan Saeed, and Sateesh Addepalli. Progress and challenges in intelligent vehicle area networks. *Commun. ACM*, 55(2):90–100, February 2012.

[25] Apache Software Foundation. Apache kafka - a distributed streaming platform. `https://kafka.apache.org/`. [Online; accessed 24-April-2019].

[26] Apache Software Foundation. Welcome to apache zookeeper. `https://zookeeper.apache.org/`. [Online; accessed 24-April-2019].

[27] John Harding, Gregory Powell, Rebecca Yoon, Joshua Fikentscher, Charlene Doyle, Dana Sade, Mike Lukuc, Jim Simons, Jing Wang, et al. Vehicle-to-vehicle communications: readiness of v2v technology for application. Technical report, United States. National Highway Traffic Safety Administration, 2014.

[28] Hyperledger. Hyperledger architecture, volume 1. `https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf`. [Online; accessed 23-March-2019].

[29] Hyperledger. Hyperledger architecture, volume 2. `https://www.hyperledger.org/wp-content/uploads/2018/04/Hyperledger_Arch_WG_Paper_2_SmartContracts.pdf`. [Online; accessed 27-March-2019].

[30] Hyperledger. Hyperledger caliper: Blockchain performance benchmarking for hyperledger burrow, fabric, iroha and sawtooth. `https://hyperledger.github.io/caliper/`. [Online; accessed 29-March-2019].

[31] Hyperledger. Hyperledger introduction. `https://www.hyperledger.org/wp-content/uploads/2018/08/HL_Whitepaper_IntroductiontoHyperledger.pdf`. [Online; accessed 27-March-2019].

[32] IHS. Internet of things (iot) connected devices installed base worldwide from 2015 to 2025 (in billions). 2015.

[33] OpenAPI Initiative. Openapi. `https://www.openapis.org/`. [Online; accessed 22-April-2019].

[34] Bojana Koteska, Elena Karafiloski, and Anastas Mishev. Blockchain implementation quality challenges: A literature review. 09 2017.

[35] N. Kshetri. Can blockchain strengthen the internet of things? *IT Professional*, 19(4):68–72, 2017.

[36] Benjamin Leiding, Parisa Memarmoshrefi, and Dieter Hogrefe. Self-managed and blockchain-based vehicular ad-hoc networks. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, UbiComp '16, pages 137–140, New York, NY, USA, 2016. ACM.

[37] L. Li, J. Liu, L. Cheng, S. Qiu, W. Wang, X. Zhang, and Z. Zhang. Creditcoin: A privacy-preserving blockchain-based incentive announcement network for communications of smart vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 19(7):2204–2220, July 2018.

[38] Peng Liu, Thomas F. LaPorta, and Kameswari Kotapati. Chapter 12 - cellular network security. In John R. Vacca, editor, *Computer and Information Security Handbook*, pages 183 – 203. Morgan Kaufmann, Boston, 2009.

[39] N. Lu, N. Cheng, N. Zhang, X. Shen, and J. W. Mark. Connected vehicles: Solutions and challenges. *IEEE Internet of Things Journal*, 1(4):289–299, Aug 2014.

[40] MongoDB. Mongodb. `https://www.mongodb.com/`. [Online; accessed 21-April-2019].

[41] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`. [Online; accessed 27-March-2019].

[42] Neo4J. Neo4j. `https://neo4j.com/`. [Online; accessed 21-April-2019].

[43] NodeJS. Nodejs. `https://nodejs.org/en/`. [Online; accessed 22-April-2019].

[44] Liam O'Brien, Paulo Merson, and Len Bass. Quality attributes for service-oriented architectures. In *Proceedings of the International Workshop on Systems Development in SOA Environments*, SDSOA '07, pages 3–, Washington, DC, USA, 2007. IEEE Computer Society.

[45] Oeamtc. Oeamtc. `https://www.oeamtc.at/thema/vorschriften-strafen/vormerkdelikt-das-blueht-dem-draengler-16186758`. [Online; accessed 23-March-2019].

[46] D. Puthal, N. Malik, S. P. Mohanty, E. Kougianos, and C. Yang. The blockchain as a decentralized security framework [future directions]. *IEEE Consumer Electronics Magazine*, 7(2):18–21, March 2018.

[47] M. Samaniego and R. Deters. Blockchain as a service for iot. In *2016 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 433–436, Dec 2016.

[48] L. S. Sankar, M. Sindhu, and M. Sethumadhavan. Survey of consensus protocols on blockchain applications. In *2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS)*, pages 1–5, Jan 2017.

[49] P. K. Sharma, M. Y. Chen, and J. H. Park. A software defined fog node based distributed blockchain cloud architecture for iot. *IEEE Access*, 6:115–124, 2018.

[50] Madhusudan Singh and Shiho Kim. Blockchain based intelligent vehicle data sharing framework. *CoRR*, abs/1708.09721, 2017.

[51] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In Rainer Böhme and Tatsuaki Okamoto, editors, *Financial Cryptography and Data Security*, pages 507–527, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[52] Xiang Sun and Nirwan Ansari. Edgeiot: Mobile edge computing for the internet of things. *IEEE Communications Magazine*, 54(12):22–29, 2016.

[53] SwaggerIO. Swaggerio. `https://swagger.io/`. [Online; accessed 22-April-2019].

[54] Road Traffic Technology. Lkw-maut electronic toll collection system. `https://www.roadtraffic-technology.com/projects/lkw-maut/`. [Online; accessed 20-February-2019].

[55] Parth Thakkar, Senthil Nathan, and Balaji Vishwanathan. Performance benchmarking and optimizing hyperledger fabric blockchain platform. *CoRR*, abs/1805.11390, 2018.

[56] Typescript. Typescript. `https://www.typescriptlang.org/`. [Online; accessed 22-April-2019].

[57] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In Jan Camenisch and Doğan Kesdoğan, editors, *Open Problems in Network Security*, pages 112–125, Cham, 2016. Springer International Publishing.

[58] Zehui Xiong, Yang Zhang, Dusit Niyato, Ping Wang, and Zhu Han. When mobile blockchain meets edge computing: Challenges and applications. *CoRR*, abs/1711.05938, 2017.

[59] Z. Yang, K. Yang, L. Lei, K. Zheng, and V. C. M. Leung. Blockchain-based decentralized trust management in vehicular networks. *IEEE Internet of Things Journal*, pages 1–1, 2019.

[60] Y. Yuan and F. Y. Wang. Towards blockchain-based intelligent transportation systems. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, pages 2663–2668, Nov 2016.

[61] MI US) Zachary, Christopher L. (TROY. Method and system using a blockchain database for data exchange between vehicles and entities, November 2018.