Design and Implementation of a Blockchain-Based Task Sharing Service for Edge Computing Servers Using the Hyperledger Fabric Platform

Angelo Vera-Rivera

A Thesis submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

University of Manitoba

Copyright © 2022 by Angelo Vera-Rivera

Design and Implementation of a Blockchain-Based Task Sharing Service for Edge Computing Servers Using the Hyperledger Fabric Platform

University of Manitoba
Faculty of Graduate Studies

We hereby approve the thesis[1] of

Angelo Vera-Rivera

for the degree of

MASTER OF SCIENCE

Dr. Ekram Hossain

Committee Chair, Advisor — Date
Department of Electrical and Computer Engineering

Dr. Robert McLeod

Committee Member — Date
Department of Electrical and Computer Engineering

Dr. Pradeepa Yahampath

Committee Member — Date
Department of Electrical and Computer Engineering

[1] We certify that written approval has been obtained for any proprietary material contained therein.

"Ever tried. Ever failed. No matter.
Try again. Fail again. Fail better."
- Samuel Beckett

Dedicated to Progress in
Communications Engineering

Table of Contents

List of Tables vii

List of Figures viii

Acknowledgements xi

Abstract 1

Chapter 1. Introduction 2
- Author Contributions 4
- Basics of Multi-access Edge Computing (MEC) 6
- The Blockchain Revolution and the Future of Decentralization 13

Chapter 2. Background and Motivation 17
- The Task Sharing Problem in MEC architectures 17
- Basics of Blockchain Networks 23
- The Hyperledger Fabric Framework 31
- Blockchain Technologies and Next-Gen Communication Networks 45

Chapter 3. Methodology and Design Elements 53
- The case for Hyperledger Fabric-based Services for 5G Networks 53
- A Hyperledger Fabric Framework for Task Sharing Collaboration in MEC 57
- Design of the Task Sharing Model at the Blockchain Level 66
- Design of the Task Sharing Model at the Server Level 70

Chapter 4. *EdgeChain*: a Blockchain-based Task Sharing Service Demo 83
- Preliminary analysis 83
- *EdgeChain* network setup 85
- *EdgeChain* smart contracts 95
- *EdgeChain* Client Applications 107
- Performance Evaluation Results 118

Chapter 5. Conclusions 131

References 137

List of Tables

2.1 Summary of examined research work on task sharing schemes for MEC networks 24

2.2 Taxonomy of blockchain systems: membership, consensus, ledger, and smart contract classification 28

2.3 Public vs Private vs Consortium Blockchains 32

2.4 Application of blockchain technologies in next-gen network architecture 48

2.5 Application of blockchain technologies in next-gen network services 50

2.6 Application of blockchain technologies in next-gen user applications 52

3.1 Comparison between traditional and blockchain-based marketplaces for telecommunications operators 54

3.2 Summary of symbols and notations 78

4.1 Hyperledger Fabric software prerequisites 86

4.2 Technical specifications of the available PCs 87

4.3 Names, IPs, and labels of participants inside the *EdgeChain* cluster 89

4.4 Hyperledger Fabric components for the *EdgeChain* network 93

4.5 *Service Terms* negotiation properties 96

4.6 *Computational Task* properties 102

4.7 Fabric parameters for the database experiment. 119

4.8 Fabric parameters for the endorsement policy experiment. 122

4.9 Fabric parameters for the transaction arrival rate & block size experiment. 126

List of Figures

1.1	Illustration of the fog/edge computing paradigm.	10
1.2	(a) Centralized ledger technology (b) distributed ledger technology	15
2.1	(a) Local execution. (b) partial sharing. (c) full sharing.	18
2.2	Basic components of task sharing decision schemes	20
2.3	Illustration of the idea of task sharing at the edge level	22
2.4	Basic layers of the blockchain reference architecture	25
2.5	Transaction flow in the Hyperledger Fabric blockchain framework	36
2.6	Illustration of a fully formed Fabric network with its main components in action	44
3.1	Illustration of the Hyperledger Fabric-based marketplace for network operators	57
3.2	Illustration of the Hyperledger Fabric framework for task sharing collaboration in MEC networks	61
3.3	Variation of the Transaction Latency versus Transaction Arrival Rate for Hyperledger Fabric versions 1.4.0 and 1.4.1.	65
3.4	Variation of memory usage (top) and CPU usage (bottom) versus Transaction Arrival Rate for Hyperledger Fabric versions 1.4.0 and 1.4.1.	66
3.5	Task sharing model at the blockchain level	69
3.6	Graph representation of a task-flow computational job with general dependent tasks.	71
3.7	Representation of the schedule of $\mathcal{A} = \{1, \ldots, A\}$ tasks executed by a subset of $\mathcal{S} = \{1, \ldots, S\}$ edge servers. The makespan of the schedule is $t + 6$.	76
3.8	Results of experiment 1: average task completion time.	80

3.9 Results of experiment 2: average number of service providers. 81

4.1 Blockchain players, their roles, and available transactions in *EdgeChain* network. 84

4.2 Illustration of the *EdgeChain* docker swarm cluster 88

4.3 Arrangement of docker containers inside *EdgeChain* participating organizations 90

4.4 Routine for spinning-up Fabric containers 91

4.5 Routine for channel creation 92

4.6 Lifecycle transitions of *Service Terms* objects 97

4.7 Lifecycle transitions of *Computational Task* objects 103

4.8 Communication between *BusyServerApp* clients and ESP gateway peers 108

4.9 Interaction between *BusyServerApp* clients and *Service Terms Chaincode* (the function being called is *serviceRequest*). 109

4.10 *Communication sequence between BusyServerApp clients and the EdgeChain network* 110

4.11 Communication between *CloudLevelApp* clients and CSP gateway peers 112

4.12 Interaction between *CloudLevelApp* clients and *Service Terms Chaincode* (the function being called is *uploadTask*). 113

4.13 Communication sequence between *CloudLevelApp* clients and the *EdgeChain* network 114

4.14 Communication between *PrivateSharingApp* clients and ESP gateway peers 115

4.15 Interaction between *PrivateSharingApp* clients and *Task Sharing Chaincode* (the function being called is *downloadResults*). 116

4.16 Communication sequence between *PrivateSharingApp* clients and the blockchain network 117

4.17 Average latency of *Service Terms Chaincode* transactions as a function of state database. 119

4.18 Average latency of *Task Sharing Chaincode* transactions as a function of state database. 120

4.19 Average latency of *Service Terms Chaincode* transactions as a function of endorsement policy. 123

4.20 Average latency of *Task Sharing Chaincode* transactions as a function of endorsement policy. 124

4.21 Average Transaction Latency and Average Transaction Throughput versus Transaction Arrival Rate with block size equal to 1. 126

4.22 Average Transaction Latency and Average Transaction Throughput versus Transaction Arrival Rate with block size equal to 3. 127

4.23 Average Transaction Latency and Average Transaction Throughput versus Transaction Arrival Rate with block size equal to 5. 128

4.24 Average Transaction Latency and Average Transaction Throughput versus Transaction Arrival Rate with block size equal to 7. 129

Acknowledgements

I would like to express my infinite gratitude and admiration to Professor Ekram Hossain for his lucid advice and patient guidance. A special recognition to Dr. Ahmed Refaey for his generosity in providing key ideas and feedback to the project. My praise to the whole Wicons team for their continuous collaboration and good vibes. A sincere acknowledgment to the University of Manitoba for providing partial funding for my education. Finally, my biggest appreciation to my beloved family. I could not have done this without their support and encouragement.

Abstract

Design and Implementation of a Blockchain-Based Task Sharing Service for Edge Computing Servers Using the Hyperledger Fabric Platform

Angelo Vera-Rivera

The efficient administration of network resources in Multi-access Edge Computing (MEC) is an active research topic these days. Task sharing, in particular, is one of the fundamental problems regarding MEC architectures although the existing literature approaches the topic mostly from the mobile user point of view. This work explores the use of blockchain technologies for developing a secure and private task sharing model specially designed for edge computing servers. The collaboration scheme aims to increase the utilization of computing resources at the edge layer of MEC networks. The suggested solution operates on a decentralized approach that counts on the blockchain services provided by the Hyperledger Fabric platform. The model offers enhanced security features that leverage the permission-ed nature of Fabric, more specifically, it relies on Fabric's membership service to validate identities and allowed behavior of participant nodes in the network. This design choice restrains external attackers from interfering with the normal operation of the task sharing scheme. The model also offers enhanced privacy features powered by a smart contract design that makes use of multiple decoupled Fabric channels with separate operation rules, ledgers, and peer-to-peer communication networks. This design choice guarantees that the computational tasks circulating in the network are only exposed to servers participating in task sharing services. The trait prevents other actors in the blockchain to have access to private tasks and their data. The contribution of this thesis can be summarized as follows: (1) a conceptual framework for task sharing collaboration in MEC networks using the Hyperledger Fabric platform, (2) a design of the task sharing model at the blockchain and edge server levels, and finally, (3) the implementation details of *"EdgeChain"*, a proof of concept demo for the proposed solution.

1 Introduction

Multi-access Edge Computing (MEC) is a network architecture that relocates cloud services (i.e. processing power, memory, storage, and control functions) geographically close to end-users at the edge of a mobile network in the RAN premises. The intuitive idea behind MEC is that the distribution of computational intelligence across the edge layer may improve the Quality of Service (QoS) offered by the mobile network. This architectural change may provide the grounds for the implementation of high-performance applications and services that respond to users in near real-time. MEC technology has become a key enabler of 5G networks because it envisions a way to provide massive, very-low latency and very-high bandwidth infrastructure that can fulfill the technical requirements of emerging 5G-based business models.

In a real implementation, edge servers are placed in strategic geographical locations to serve a specific coverage area. Upon receiving computing requests from user applications, the servers must provide processing services without the possibility to refuse access to their infrastructure. However, the adoption of massive machine-type communication in 5G networks could potentially overwhelm the processing infrastructure within a service area and decrease the network's ability to meet the required latency and bandwidth targets required. In this scenario, a task sharing model with the capacity to coordinate computer resources at the edge layer may alleviate the distress on constrained servers overloaded with computing

requests. In detail, task sharing refers to the partial or full migration of resource-intensive tasks owned by constrained devices to near infrastructure with plenty of available resources. The migration proceeding intends to solve performance shortcomings (i.e. processing power, memory, storage, and energy deficiencies) at compromised devices in communication networks.

Blockchain is a peer-to-peer computer network that uses advanced cryptography and a distributed database to keep a record of the interactions of a group of independent nodes that do not trust each other. In the blockchain context, the interactions among nodes receive the name of transactions. Transactions are packed and stored in a distributed database called ledger that is hosted by the network and consists of data blocks connected chronologically that resemble the form of a chain. Blockchain transactions are vetted as valid or not valid by a communal verification process called consensus that generally speaking, involves a computationally intensive verification process. Consensus may be viewed as a form of digital democracy that allows nodes in a blockchain to make collective decisions about the network without the supervision of a central authority overseeing the process. A blockchain network must enforce three main characteristics: (1) the ledger must be tamper-resistant, (2) the data in the ledger must be transparent to all (or some) nodes in the blockchain, and (3) transactions circulating in the network must be validated by a consensus algorithm. Blockchain is a powerful concept that provides the technical grounds for the implementation of distributed networks that do not need central supervision to exist and function normally.

The Hyperledger Fabric platform is a permissioned blockchain developed by the Linux Foundation that offers a lightweight and flexible platform for multi-purpose distributed solutions. Fabric stands out from regular blockchains due to its modular architecture. More specifically, Fabric provides a membership service that verifies the identity of network participants and assigns roles in the network that can change depending on the use case. Also, Fabric offers plug-and-play consensus which allows the designer to choose what type of transaction validation

scheme best fits the approached solution. Finally, Fabric supplies smart contract services for the implementation of the use-case logic. Hyperledger Fabric is an interesting option for decentralized applications. It is already very popular in a wide variety of industries such as supply chain, finance, pharmaceuticals, and retail.

As hinted earlier, task sharing is one of the fundamental problems in MEC networks, however, existing literature mostly approaches the topic from the mobile user point of view. MEC is currently a supporting technology of the cloud computing paradigm where network resources at the edge layer cannot be shared. The truth is that traditional MEC frameworks do not consider scenarios where edge servers can coordinate their computer resources to provide decentralized processing services to user applications. The cloud layer is responsible to allocate neglected tasks from strained edge servers to available infrastructure willing to provide processing services. The problem with the central-cloud approach is that it may add a communication overhead that can deteriorate the end-user experience. On the other hand, decentralized approaches might produce reasonable models to handle task sharing problems, however, there exist security and privacy concerns that need to be addressed first before a feasible distributed model for task sharing collaboration can be put in production. For example, what if malicious actors infiltrate the MEC ecosystem? How can edge servers owned by different service providers establish trust during the sharing process? Questions like those are becoming active research topics in the areas of resource management, security, and privacy for MEC networks.

1.1 Author Contributions

This thesis explores the use of the Hyperledger Fabric platform to power a blockchain-based collaboration scheme for edge servers in a MEC environment. The proposed model uses Fabric as a decentralized playing field for the interaction of edge servers that minimizes potential security and privacy breaches. The security traits of our

model rely on the permission-ed nature of Fabric. The platform implements a membership module called Membership Service Provider (MSP) in charge of identity authentication and role management in the network. In fact, the MSP maps identities to roles and also defines permissions over resources, access to data, and allowed behavior of participants in the blockchain. This functionality prevents external malicious actors to interfere with the normal operation of the task sharing scheme. In addition to that, the enhanced privacy features offered by the model rely on the possibility to implement multiple blockchain channels within a single Fabric platform. Fabric permits the coexistence of several blockchain channels with separate rules, ledgers, and peer-to-peer communication networks within a single Fabric ecosystem. By leveraging this feature, the tasks circulating in the network are kept confidential to edge servers participating in task sharing services. The model restrains non-participating actors from accessing private tasks and their data.

The contribution of this work can be summarized in the following items:

(1) A conceptual framework for task sharing collaboration in MEC architectures using the Hyperledger Fabric platform,
(2) A design of the task sharing model at the blockchain and edge server levels, and
(3) The implementation details of "EdgeChain", a proof of concept demo for the proposed solution.

The rest of the thesis is organized as follows. Chapter 2 presents the background and motivation of this work. We include an overview of the task sharing problem in MEC architectures, an analysis of the architecture of blockchain networks, and a revision of the Hyperledger Fabric platform and its potential integration with new generation communication networks. Chapter 3 presents the methodology and design elements of the proposed solution. In detail, we include the case for Hyperledger Fabric-based services for 5G networks, a Hyperledger Fabric framework for task sharing collaboration in MEC networks, the design of the task sharing model at the blockchain level, and the design of the task sharing model at the server level.

Chapter 4 presents the implementation details of "EdgeChain", the blockchain-based task sharing service demo. Finally, Chapter 5 concludes this thesis.

1.2 Basics of Multi-access Edge Computing (MEC)

The new 5G standard is an upgrade from the previous generation of mobile systems that promises to improve access, bandwidth, latency, and performance with respect to its predecessor. 5G networks may become the facilitator of new business models with the potential to revolutionize the way we manage, power, and move the economic life of the planet. The 5G standard envisions a communication infrastructure with very-high bandwidth, very low latency, and the ability to handle a massive amount of subscribers. As a result, new 5G applications will emerge from the infrastructure to serve business models such as autonomous vehicular networks, smart manufacturing, smart grids, e-health services, tactile Internet, and immersive entertainment among others. The industry and academia are currently making huge efforts to finalize the standard so that commercial 5G networks can be deployed massively around the world in less possible time. In fact, a Cisco report shows that by the end of 2022 the number of devices connected to the Internet will reach 28.5 billion and the global data traffic will reach 122 exabytes (10^{18}) per month, 64% of that amount corresponding to mobile and wireless systems[9]. The report also shows that the projected cumulative financial investment in 5G technologies will also peak in 2022 reaching 370 billion dollars[13]. These numbers can only mean one thing: market conditions are favorable and 5G networks are at the doorstep, ready to make a big entrance in our lives.

The 5G New Radio (5G NR) is the global standardization of 5G networks that is currently being developed by the 3rd Generation Partnership Project (3GPP). The standardization derives three relevant use cases for 5G networks: (1) enhanced mobile broadband (EMBB), (2) ultra-reliable low-latency communication (URLLC), and (3) massive machine-type communications (MMTC). Also, the standard states

that 5G will not only support communication, but also content delivery, computation, and control functions[30]. The paradigm shift will power a surging marketplace for products and services characterized by massive volumes of data and whooping requirements of processing power. To meet the demands, 5G relies on several technologies from different fields, such as Multi-access Edge Computing (MEC), Millimeter Wave (MMWA), Non-orthogonal Multiple Access (NOMA), Dense Heterogeneous Networks (HetNets), Machine Learning (ML), and Energy Harvesting (EH) to name a few. MEC in particular is a major enabler of 5G networks because it leverages the idea of distributed computing to locate cloud services (i.e. processing power, memory, storage, and control functions) near end users. The architecture introduced by MEC might increase the available data rate and reduce the latency experienced by subscribers using the infrastructure of 5G networks. A detailed explanation of MEC technology is presented in the next section.

1.2.1 A peek into the past

The first computing paradigm that combined wireless networks and central processing clouds was known as Mobile Cloud Computing (MCC). This paradigm moves the computing power and storage capabilities of mobile networks into centralized clouds to deliver rich applications to end devices with a poor stock of resources. The centralized approach allows the delivery of complex services to users that could not otherwise support them. There are many advantages related to this paradigm including quick development, shared resources, integrated data, and a variety of services ready to use. However, MCC also comes with a few disadvantages. First, the large distances between subscribers and the cloud may translate into high latency which might be a problem for latency-sensitive applications. Second, the large volume of data exchanged between mobile devices and the cloud could create bandwidth shortages and congestion in the network. Finally, the centralized cloud approach creates a single point of attack that puts offloaded data in a vulnerable position compromising the security and privacy of the system.

Multi-access edge computing (MEC) is the evolution of the MCC paradigm that relocates cloud services such as processing power, memory, storage, and control functions, spatially close to end subscribers. In mobile systems, this can be done by deploying intelligence capabilities to the edge of the network within the RAN premises[16]. The intuitive idea behind MEC is that the relocation of cloud services might improve the QoS and overall performance of the network. The distribution of intelligence across the network is vital to deploy highly available, highly reliable, and high-performance applications that can respond to users in near real-time, a frequent requirement for upcoming 5G-based business models.

The first attempt to decentralize computer resources in network architectures was presented by Satyanarayanan et al. in 2009[42]. The model received the name of *Cloudlet*. In simple terms, *Cloudlet* consisted of a set of trusted, resource-robust, Internet-connected clusters of computers located at the edge of a mobile network. The motivation of the arrangement was to support the computation of intensive applications for mobile users with limited resources. *Cloudlet* was equipped with virtual machine capabilities that could function in stand-alone mode without the involvement of the cloud. The mobile users relied on WiFi connections to access *Cloudlet*, however, WiFi limited use of unlicensed frequency bands represented a problem for the security and scalability of the system. In addition to that, the coverage limitations of WiFi made mobile users located at far distances not capable to access the cloudlets because they had to switch from mobile connection to WiFi when using *Cloudlet* services.

Later in 2014, Cisco coined the term *Fog Computing* to label their conceptualization of edge computing architectures. Cisco's Fog describes the migration of cloud computing services from the core to the edge of a network that creates a metaphorical fog of distributed computing infrastructure right outside cloud data centers[8]. *Fog computing* deploys processing power, memory, and storage resources in locations geographically close to subscribers so that they can execute

computing-intensive applications in fog nodes outside the cloud level. This arrangement reduces latency, congestion, and infrastructure costs in the network. Although fog nodes extend cloud infrastructure and services into the edge, they cannot act as self-managed cloud data centers. Indeed, coordination between the central cloud and the fog is still needed for the proper operation of the system. *Cloudlet* and *Fog Computing* are similar in that they both provide services to mobile users, yet, they are not integrated into the mobile network architecture. In fact, *Cloudlet* and *Fog Computing* nodes are typically owned by private service providers separate from mobile network operators.

To upgrade *Fog Computing*, designers attempted to deploy cloud services even deeper into the edge and gave birth to the idea of *Multi-access Edge Computing* (MEC). In the upgraded architecture, MEC nodes provide cloud services to the edge of mobile networks, and *Fog Computing* nodes act as intermediaries between edge nodes and the centralized cloud. Fast forward to recent years, the evolution of MEC technology has been driven by many factors including the increasing number of mobile users, huge volumes of data, and the need for high-bandwidth and low latency infrastructure that may serve 5G-based business models. In general, MEC systems may be described by four technical characteristics[35]. First, MEC networks operate on-premises. They can run isolated from the rest of the network and have access to local resources. Second, MEC networks provide proximity. They are deployed geographically close to end users and may collect (and process) as much data as possible from them. Third, MEC networks supply low latency infrastructure with the potential to shorten communication and propagation times. Finally, due to proximity, MEC networks are capable of providing subscribers with precise location services and contextual information about network conditions for optimization purposes. An illustration of the fog/edge computing paradigm is presented in Figure 1.1.

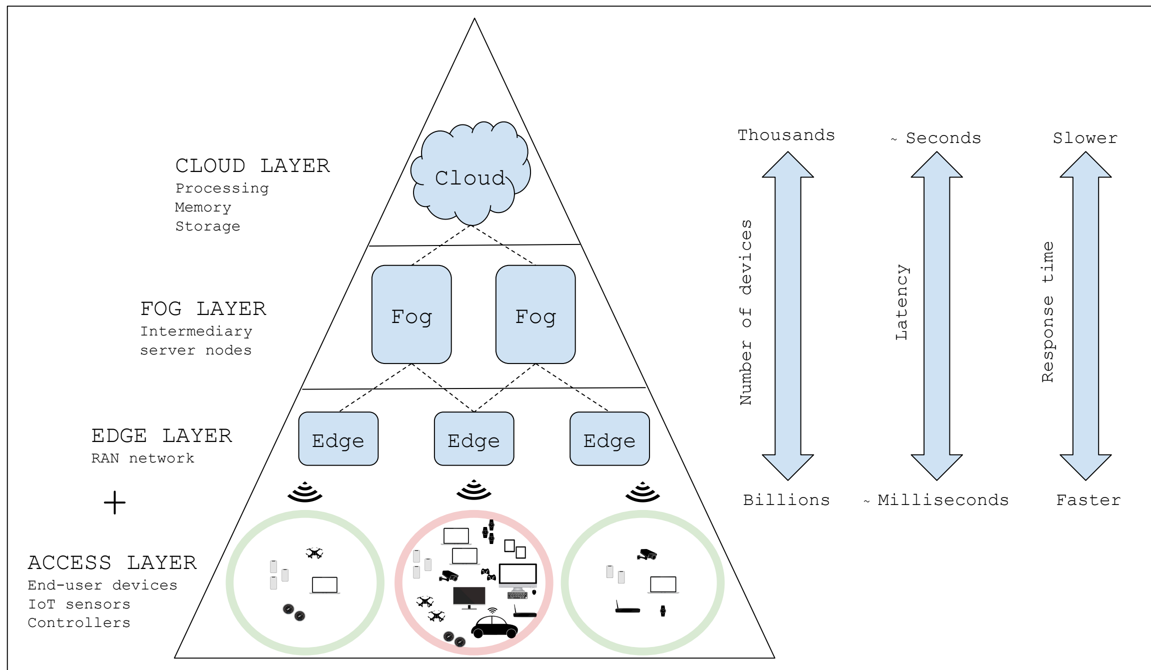

Figure 1.1. Illustration of the fog/edge computing paradigm.

1.2.2 Principles of MEC

The reference architecture of MEC is composed of two planes: (1) system and (2) host planes. The system plane consists of a MEC Orchestrator (MECO) that is responsible for maintaining basic information about the MEC environment, specifically, network topology, available resources, and running applications. The host plane, on the other hand, consists of a host server, a MEC platform manager (MPM), and a virtualization infrastructure manager (VIM). The host server is equipped with the MEC platform and the virtualization infrastructure. The MEC platform consists of a set of functions that support the operation of MEC applications. The virtualization infrastructure provides the grounds for the proper flow of traffic between nodes and applications within the architecture. Concerning the managers, the MPM is in charge of the correct operation of MEC applications and the management of their performance and requirements. The VIM, on the other hand, oversees the operation of virtualized infrastructure, allocation of virtualized resources, and the adequate provisioning of running MEC applications and services. To meet

MEC promises, the architecture needs to deliver on some key principles that are summarized as follows:

Resource Management. The administration of network resources is a critical component in MEC architectures. The tension between constrained infrastructure and the increasing demand for resources makes allocation methods an important research topic in the communications community. The allocation of resources may have multiple objectives and it is subject to the heterogeneity of the network. More specifically, it should account for a massive number of subscribers with individual demands, connecting links with different characteristics, and user applications of different nature. In that context, task sharing is one of the fundamental problems in MEC networks although most of the time it is only approached from the end-users point of view. However, a new trend for task sharing focused on devices closer to the core of the network is emerging rapidly, and that is precisely the area explored in this thesis.

Mobility. Emerging network technologies such as the Internet of Things (IoT) come with the need to accommodate a massive number of subscribers that can move around the service area. User mobility in MEC networks may result in frequent handovers that can cause service disruption and affect the overall performance of the system. For example, a mobile user can change coverage areas in the middle of a task sharing service. In that scenario, MEC architectures must provide a fast and reliable handover mechanism that accounts for changes in the location of the user, dynamic conditions of the access network, and time-varying computation resources.

Heterogeneity. Edge servers may be deployed in different places within the RAN premises to provide coverage to end users with heterogeneous characteristics and requirements. Given that scenario, MEC networks should provide seamless integration of edge devices with the existing access and core infrastructure. Edge servers are deployed in the network to boost available bandwidth and reduce the latency

experienced by user applications. However, the processing capacity of the cloud is still way more robust than that installed at the edge layer. Therefore, it may be a good strategy to let the cloud handle tasks that are tolerant of delays to further optimize the use of computing resources at the edge layer. In that direction, MEC systems must provide means for efficient interaction of the cloud and edge layers so that the two instances can coexist and cooperate to the benefit of the network.

Security and Privacy. Despite the many advantages of MEC networks, some security and privacy downsides must be addressed. First, due to the proximity of edge servers to end users, traditional MCC security methods are not compatible with MEC architectures. And second, mobile-edge or edge-edge task sharing schemes may be insecure especially when they are executed over wireless transmission channels. Although the integration of cryptography algorithms might help solve security issues, the added complexity can be a problem in terms of propagation and execution delays. Fortunately, emerging technologies such as blockchain may help mitigate these issues. Precisely, this work explores the use of the Hyperledger Fabric platform which is a lightweight cryptography-based network that offers blockchain services to help address security and privacy vulnerabilities in MEC networks.

In preparation for the integration of MEC into the 5G standard, 3GPP included MEC in the technical specification report TS 23.501 published by the group in 2019[3]. The report defines 5G network functions, their roles, and how they can seamlessly interact with the MEC reference architecture. For further explanation on this topic, we invite the reader to explore the details of the report. More information on this document can be found in the reference section.

1.3 The Blockchain Revolution and the Future of Decentralization

Blockchain is an emerging technology that is revolutionizing industries across the globe due to its unique ability to power decentralized systems. In short, a blockchain is a peer-to-peer computer network that uses sophisticated cryptography and a consensus mechanism to maintain a distributed database on a set of nodes that do not trust each other. As a result, the nodes in a blockchain network can interact without the need of a central entity dictating the rules of the system. Blockchain technologies were first introduced to the public in 2008 with the appearance of Bitcoin, the first ever known distributed payment system, that uses blockchain as its underlying operating platform. To date, the communications industry is being disrupted by the appearance of new management frameworks for future massive networks that move from centralized to decentralized network administration. Blockchain might be an option to take on this scenario due to its ability to power the implementation of decentralized, self-regulated, secure, and intelligent networks.

1.3.1 Introduction to Blockchain Systems

Blockchain is a peer-to-peer network that uses advanced cryptography to maintain a distributed database of records (i.e. ledger) among a group of independent nodes that do not trust each other. The nodes in a blockchain can interact with the network in the form of transactions validated by a communal verification process known as consensus that normally involves a computing-intensive algorithm. Consensus might be considered a form of digital democracy that allows the nodes in the network to make collective decisions about transactions without the supervision of a central authority. Transactions are packed together inside a structure of data blocks connected chronologically that resemble the form of a chain. By keeping a local copy of the ledger, the nodes in a blockchain become guardians of the history book of the network. This feature provides a high level of transparency

and accountability to the system.

Blockchain networks derive directly from Distributed Ledger technology (DLT). DLT is a database paradigm that defines a set of cryptographic protocols that support the operation of distributed transaction records. More specifically, the DLT paradigm defines a way to access, validate, update, and manage decentralized databases. The protocols guarantee the immutability, consistency, availability, and transparency of the database records. An illustration of the difference between central ledger technology and distributed ledger technology is shown in Figure 1.2. As a general rule, blockchain networks must enforce three fundamental characteristics: (1) immutability of the distributed ledger, (2) transparency of information and (3) consistency of data across the nodes. The origins of blockchain can be traced back forty years ago to the study of consensus problems in the field of distributed computing. For example, the Byzantine Generals problem was formulated in the early eighties by Leslie Lamport et al.[21]. It is a fault-tolerant solution algorithm to reach a consensus on messages transmitted over a network in the presence of faulty messages and adversarial nodes. In 2008, the creators of Bitcoin used this model to develop the consensus mechanism of their platform. Also in the eighties, Koblitz[17] and Miller[32] developed the Elliptic Curve Cryptography (ECC) framework for a public key encryption solution. More than two decades later, ECC was also picked by Bitcoin developers for the encryption model of their system.

The history of blockchain technologies officially started in 2008 when Satoshi Nakamoto (who is believed to be a nickname) invented Bitcoin: an open programmable cryptocurrency along with a distributed payment system to bypass central bank currencies as the means to make monetary transactions[33]. Nakamoto solved the Byzantine Generals Problem for the circulation of Bitcoin currency using a Proof of Work algorithm. Bitcoin makes use of a distributed database to keep the record of all transactions stored inside a particular data structure. More specifically, transactions are saved in a chain of data blocks connected via hash pointers so that it was almost impossible to manipulate the records without being detected and corrected.

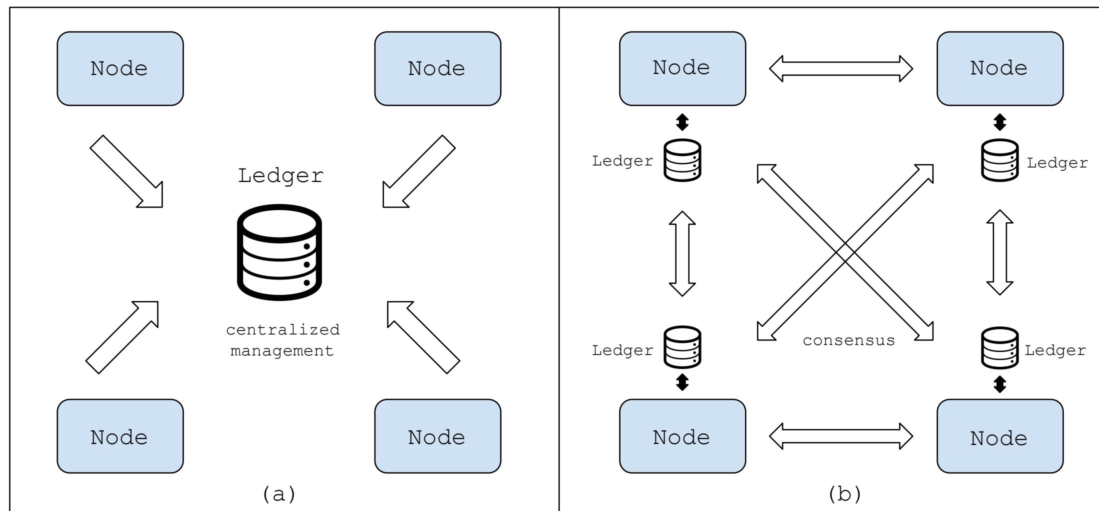

Figure 1.2. (a) Centralized ledger technology (b) distributed ledger technology

At the time, Nakamoto's programmable currency and payment system created a very enthusiastic movement that saw in Bitcoin the opportunity to disrupt the monopoly of governments over national currencies and economic transactions. The Bitcoin movement certainly caught a lot of attention in the financial sector due to the currency's ability to circulate freely on the Internet.

Looking beyond the financial sector, Vitalik Buterin created Ethereum in 2015. Ethereum was a novel blockchain model with the ability to run programmable logic through the use of immutable software installed on the platform[5]. The software running on Ethereum applications receives the name of smart contracts. They support the implementation of multiple data structures and functionality that serve the business logic of decentralized systems. The core idea behind smart contracts is that they are compiled and stored in the blockchain so that network nodes hold a local copy of the software in their ledgers. This feature allows every node in the system to execute their copy of the software and reduce the possibility of malicious actors interfering with the business logic programmed in the network.

In recent years, blockchain technologies have been tailor-designed to serve applications in a variety of economic sectors such as finance, supply chain, energy,

etc. The ability of blockchain to power networks with decentralized governance, immutable data, and transparent information has been a key motivation for the development of those solutions. To date, research projects regarding blockchain integration in engineering systems are very popular. We predict that this trend will continue to grow due to the blockchain's potential to revolutionize the way we design future engineering systems.

2 Background and Motivation

2.1 The Task Sharing Problem in MEC architectures

2.1.1 Essentials of Task Sharing

Task sharing refers to the partial or full migration of computing-intensive from resource-constrained devices to near infrastructure with plenty of available resources. The migration aims to solve resource shortcomings (i.e. processing power, memory, storage, and energy deficiencies) at strained nodes in a network[46]. Task sharing decisions deal with three basic questions: (1) what to share, (2) where to, and (3) at what time. The optimal answers might be based on specific objectives than depend on network conditions and user preferences, such as installed computing power, available bandwidth, and latency requirements.

Computational tasks can be thought of as units of execution of software applications that perform specific activities within the program. Their properties might be defined by data size, required CPU cycles, release times, completion deadlines, node exclusivity, and functional dependencies. It is possible to analyze the sharing profile of a software application in terms of three aspects: (2) *separability*, (2) *a-priori data*, and (3) *dependencies*. First, for applications that enable code or data partitioning, not all the parts are candidates for sharing. For example, input and output events are tasks that typically need to be executed at the mother node, therefore, they most likely cannot be offloaded. Second, for applications that process data, complete apriori knowledge of the amount of information to be processed is not always possible, for example, online gaming software that requires continuous

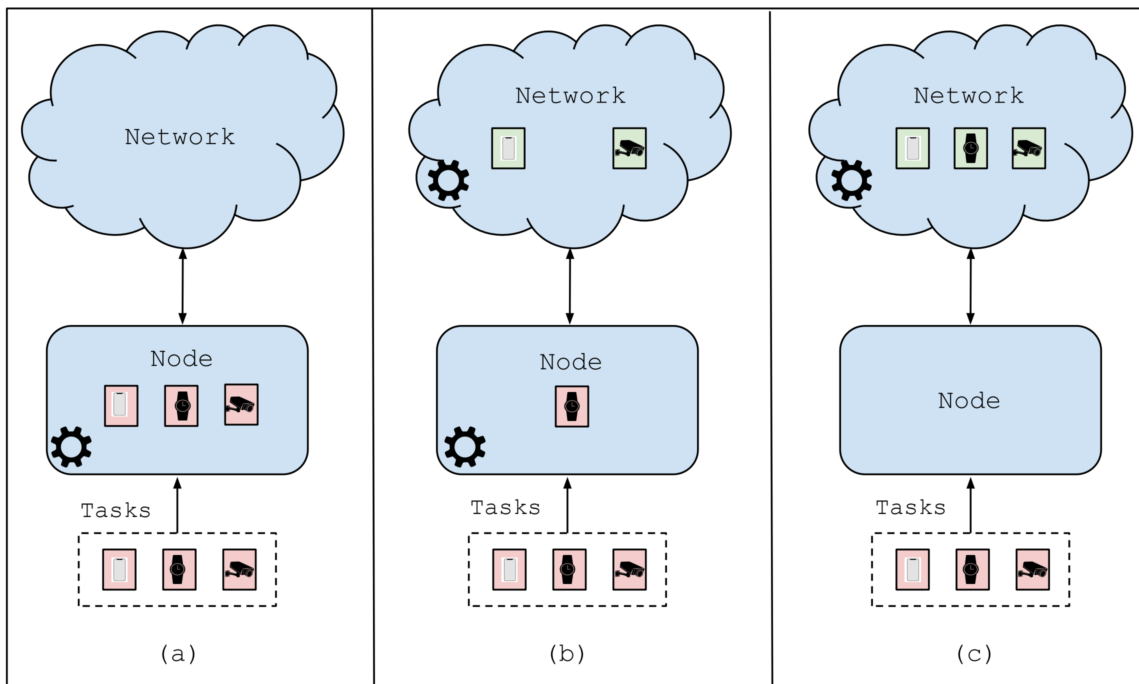

Figure 2.1. (a) Local execution. (b) partial sharing. (c) full sharing.

execution. For this type of application, task sharing can become a very complex process. Finally, for applications composed of tasks that receive (or provide) feedback from (or to) others, parallel sharing might not be possible. In that case, the dependencies among the tasks may play a decisive role in the sharing decisions.

Depending on their sharing eligibility, software applications may apply three types of task sharing strategies: (1) local execution (no sharing), (2) full sharing, and (3) partial sharing[27]. In any case, task sharing schemes are strategies to hit certain performance targets defined by the network's designers. The targets may involve the maximization of utility functions with user preferences, minimization of the execution time of software applications and their tasks, minimization of energy consumption, or any other system-defined operation. A further explanation of task sharing alternatives is presented in the next sub-sections. An illustration of the task sharing modes can be found in Figure 2.1.

Local Execution. The computation of tasks is done entirely at local premises. This may happen when software applications cannot be partitioned into shareable tasks or when task sharing schemes do not bring benefits to the network. The execution delay accounts for time spent on the local execution of tasks.

Partial Sharing. The computation of tasks is done partially at local premises and the rest is done at other nodes in the network. This happens when software applications are composed of both, shareable and non-shareable tasks. The execution delay accounts for time spent on local execution of non-shareable tasks plus time spent on transmission, remote execution, and return of results of shareable tasks.

Full Sharing. The computation of tasks is passed entirely to remote nodes in the network. The execution delay accounts for time spent on transmission, remote execution, and return of task results.

To implement a task sharing scheme, a network must be equipped with three essential modules: (1) task profiler, (2) system profiler, and (3) decision engine[11]. The task profiler is the module that determines what parts of a software application can be shared. It can translate applications into a set of tasks, computing requirements, and dependencies. The system profiler is the module responsible for monitoring the performance parameters of the network. The parameters might include CPU power, storage, memory, consumed energy, etc. Finally, the decision engine is the algorithmic approach to making sharing decisions based on the information provided by the task profiler and the system profiler. An illustration of this idea is presented in Figure 2.2.

2.1.2 Task sharing in a MEC context

In connection to MEC architectures, most of the recent work on task sharing has been approached from the perspective of mobile users. For instance, in 2018 the authors Luong et al. proposed a deep neural network architecture to make optimal task sharing decisions in a mobile blockchain network[26]. In their work, they model

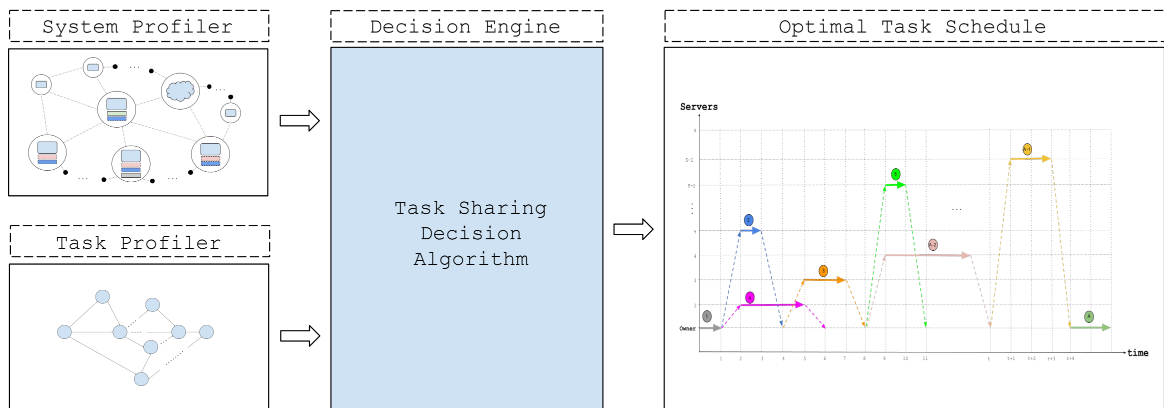

Figure 2.2. Basic components of task sharing decision schemes

the sharing scheme as an auction game between edge servers and mobile users. The computing power of edge servers is the commodity to be auctioned and mobile users must submit bids to *purchase* processing services. The bids become the inputs of the neural network and the outputs determine the task sharing decision along with the associated payments for the computing service.

Also in 2018, the authors Liu et al. proposed a MEC-enabled task sharing framework for a blockchain-based video streaming architecture[25]. They designed an incentive-based collaboration model for communication systems composed of a macro base station, a set of small-cell base stations, and a set of mobile users all running on a blockchain. In the proposed model, the mobile users have the option to offload computation tasks to nearby small-cell base stations or offload them to a group of device-to-device users to avoid the saturation of macro base stations. The authors formulate the sharing scheme using an adaptive block-size blockchain to minimize the overall latency of the video streaming service.

In their 2020 work, the authors' Xiao et al. investigated a blockchain-based task sharing mechanism for mobile users in a MEC network[49]. In their analysis, the mobile users can make task sharing decisions based on the computational performance and latency response of their serving edge server node through a Deep Reinforcement Learning module. The mechanism uses a generic blockchain to

keep an immutable ledger of service records which are later used to calculate the money payment earned by the edge servers providing task sharing services. The authors suggest that their approach suppresses the motivation of selfish behavior and faked service record attacks by malicious edge servers.

In recent years, the focus of task sharing schemes has expanded from models that only serve mobile users to ones that include edge server nodes as well. An illustration of a basic idea of task sharing at the edge layer can be found in Figure 2.3. In 2017, the authors Chen and Xu proposed a collaboration mechanism for small-cell base stations in an ultra-dense user setting[6]. They designed a coalitional game to distribute computation workload among small-cell base stations to summon computing resources at the edge level. The design covers cooperation incentives, fair division of tasks, and a social trust mechanism that helps manage security risks. In 2018, the same authors presented an expansion of their work that aimed to minimize the long-term system-wide latency using Lyapunov optimization. The model proposes an optimization approach that takes into account the capacity and energy constraints of the small-cell base stations[7]. They analyzed two scenarios, one in which a central entity makes task sharing decisions for the small-base stations, and another in which the base stations coordinate their sharing decisions in a distributed style using a coalitional game approach.

In 2020, the authors He and Wang presented a task sharing algorithm for MEC networks that provided a worst-case latency response for latency-sensitive applications in a mobile network[14]. They formulated a task sharing problem for base stations formulated on a stochastic arrival model that maximizes a utility function that depended on the time-average throughput of the system. They built the model based on the assumption that the tasks utilizing sharing services carry on the same workload. This assumption made the problem easier to solve, however, it does not hold in a real network scenario. The same authors upgraded their work in 2021 and presented a task sharing model in which they allow the workload of the

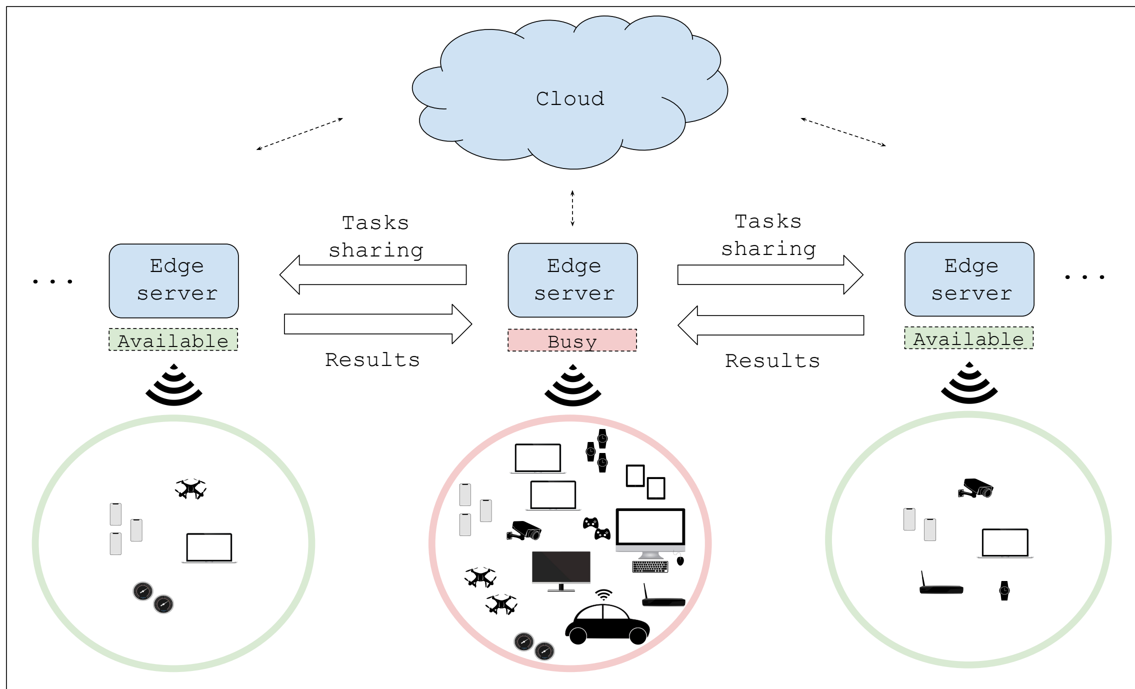

Figure 2.3. Illustration of the idea of task sharing at the edge level

tasks to have different profiles[15]. They presented a scheduler and an online algorithm based on Lyapunov optimization theory that provides a worst-case latency response model for the system. Their algorithm guarantees that all non-dropped tasks are served within a given latency bound. According to the authors, their algorithm tends to overestimate the worst-case latency of the system. In their simulations, it was found that the latency bound provided by their method was considerably larger than the actual value.

Finally, our work in 2020 proposed a framework for secure and private resource collaboration in MEC networks using the Hyperledger Fabric blockchain platform. The integration of blockchain into MEC architectures aims to minimize security and privacy threats[41] against the system. Although our framework was presented in the context of MEC, it can be generalized to cloud/fog computing frameworks. In 2021, we followed up our previous work with a blockchain-based collaborative task offloading model for MEC servers based on the Fabric platform. The model

aims to maximize the utility function of the system that depends on the characteristics of the computational tasks and key performance metrics of Fabric[48]. Fast forward to 2021, we further expanded our work with a blockchain-based task sharing model that takes into account the functional dependencies of the tasks[47]. The task sharing scheme is modeled as a multiprocessor task scheduling problem that allocates a set of precedent-dependent tasks among a set of available edge server nodes by jointly optimizing the utility function of the system and the makespan of the tasks.

To recap, the available literature on task sharing schemes for MEC networks can be divided into two branches: (1) task sharing models centered around mobile users, and (2) task sharing models centered around edge server nodes. The articles in[26],[25], and[49] leverage machine learning, game theory, and blockchain technologies to support efficient and trusted task sharing collaboration schemes for mobile users in MEC networks. On the other hand, the articles in[6],[7],[14], and[15] direct their attention to task sharing schemes for base stations in mobile networks that act as edge servers. To the best of our knowledge, our work in[41],[48], and[47] is currently the only effort to explore the use of the Hyperledger Fabric platform to power task sharing collaboration for servers at the edge layer. A summary of the research work mentioned in this section can be found in Table 2.1.

2.2 Basics of Blockchain Networks

2.2.1 Blockchain Reference Architecture

The reference architecture of a blockchain network can be divided into six interoperable layers as follows: (1) data layer, (2) network layer, (3) consensus layer, (4) incentive layer, (5) contract layer, and (6) application layer. A detailed explanation of every layer can be found in the next subsections. An illustration of the blockchain layered architecture is presented in Figure 2.4.

Focus	Contribution	Authors
Mobile users	Task sharing scheme for mobile users in a MEC network based on auction game theory and deep neural networks.	N. Luong, Z. Xiong, P. Wang and D. Niyato
Mobile users	Task offloading framework for mobile users in a MEC-enabled video streaming network based on a generic blockchain with adaptive block size.	M. Liu, F. Yu, Y. Teng, V. Leung and M. Song
Mobile users	Task sharing mechanism for mobile users in a MEC network that runs on a generic blockchain with deep reinforcement learning intelligence.	L. Xiao, Y. Ding, D. Jiang, J. Huang, D. Wang, J. Li and H. Vincent
Edge servers	Task sharing scheme for small-cell base stations for MEC-enabled ultra-dense networks based on a game theory-based fair share incentive mechanism.	L. Chen and J. Xu
Edge servers	Task sharing algorithm for base stations in a mobile network that provides worst-case latency response based on a stochastic task arrival model and Lyapunov optimization.	X. He and S. Wang
Edge servers	Collaboration framework for secure and private task sharing in MEC networks using the Hyperledger Fabric blockchain platform.	A. Vera-Rivera, A. Refaey, E. Hossain

Table 2.1. Summary of examined research work on task sharing schemes for MEC networks

Data Layer. Located at the bottom of the architecture, the data layer is responsible for tracking and maintaining blockchain blocks. Some of the functions and algorithms implemented in this layer are hash functions, digital signatures, and asymmetric encryption algorithms. A block is a data structure composed of a header with metadata and a body with network transactions. The header contains information about the system including blockchain version, timestamp, Merkle root of

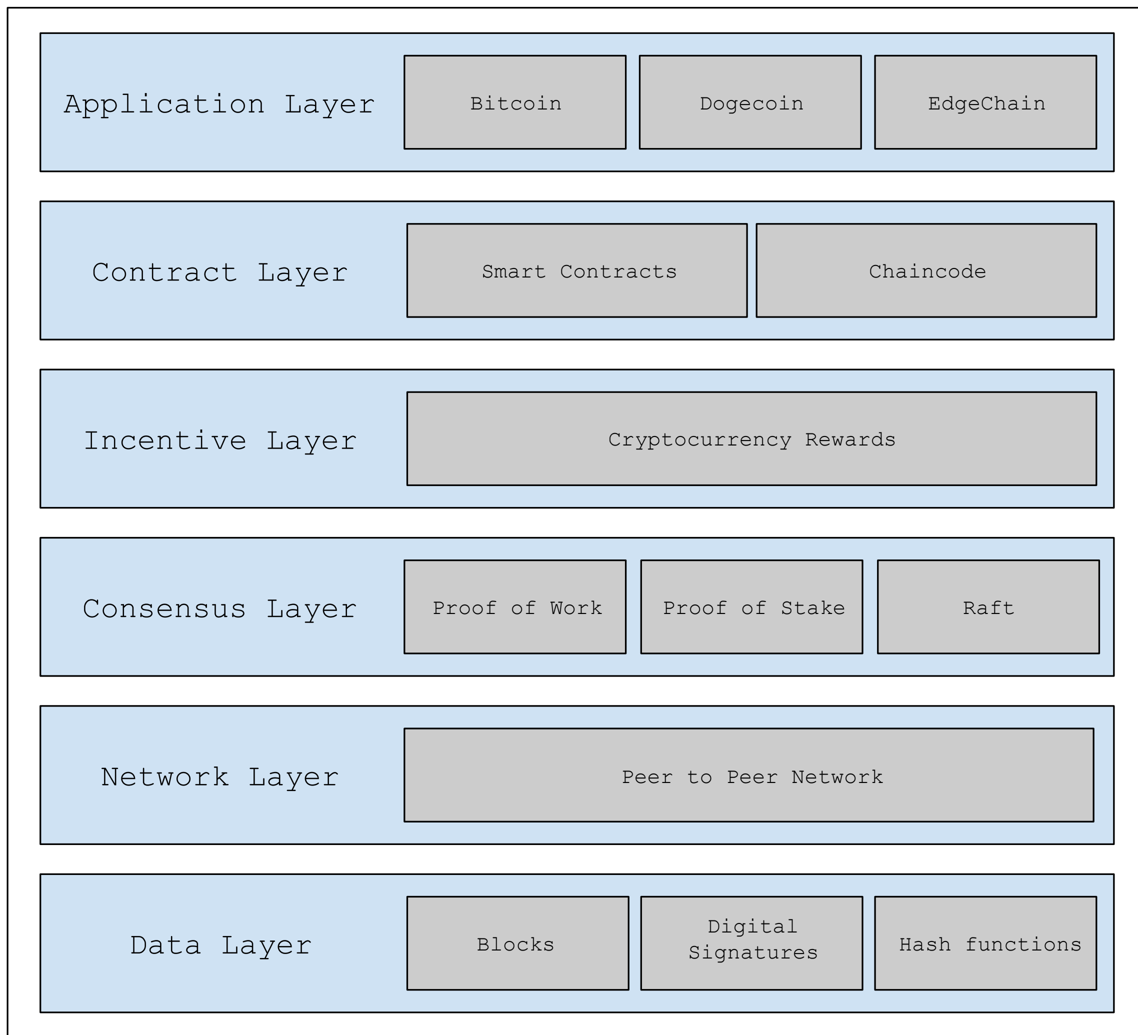

Figure 2.4. Basic layers of the blockchain reference architecture

transactions, and the hash value of the previous block (i.e. hash pointer).

Network Layer. Located right above the data layer, the network layer is responsible for the exchange of information between blockchain nodes. A blockchain is a peer-to-peer network with no central controller where transactions generated by nodes are forwarded to their neighbors. The neighbors then validate the received transactions against the logic implemented in the network. Valid transactions are

forwarded to the subsequent neighbors while invalid transactions are discarded.

Consensus Layer. Next in the stack is the consensus layer which is responsible for the way blockchain nodes reach agreement on the data circulating in the system. There are a handful of consensus protocols available for blockchain systems. Bitcoin, for example, uses Proof of Work (PoW) consensus which relies on the computing power of mining nodes to validate transactions and create new blocks. Ethereum, on the other hand, uses Proof of Stake (PoS) consensus which is a protocol that relies on *wealthy* nodes in the network to validate transactions and create new blocks. *Wealth* is defined as the number of cryptocurrencies (i.e. stake) held by the mining node. Efficient consensus algorithms are one of the fundamental challenges of distributed networks.

Incentive Layer. Located above the consensus layer, the incentive layer supervises incentive mechanisms to balance economic trade-offs between maintenance services required by the network and the blockchain nodes performing these services. The function of reasonable incentive mechanisms is to motivate blockchain nodes (i.e. miners) to participate in maintenance services required by the network. Participation may be rewarded in the form of economic payment or any other form that may be relevant to the network. For example, Bitcoin miners are responsible for data validation and block creation. As compensation, they receive cryptocurrency rewards (i.e. bitcoins) every time they successfully add new blocks to the network.

Contract Layer. Above the incentive layer sits the contract layer who is responsible for managing the programmable logic executed by blockchain nodes. The logic takes the form of immutable smart contracts that follow the operation requirements of the blockchain solution. Smart contracts are compiled, stored, and executed on the blockchain nodes preventing malicious actors from interfering with the execution of the business logic. The contract layer provides compilation, storage, and execution services for the proper integration of smart contracts into

the blockchain network.

Application Layer. Atop the blockchain model sits the application layer that serves as the interface between the outside world and the blockchain infrastructure. In this layer, blockchain architects use Software Development Kits (SDKs), Application Programming Interfaces (APIs), and other necessary software for proper communication between blockchain networks and client applications.

2.2.2 Public vs Private vs Consortium Blockchains

Blockchain networks can be classified in many forms depending on many factors such as type of governance, consensus mechanism, rewarding system, code-base design, identity management, etc[44]. The classification presented in this section breaks down blockchain systems into four distinct categories: (1) membership, (2) consensus, (3) ledger, (4) and smart contract models. The membership category, in particular, can further divide blockchain networks into three different types: (1) Public, (2) Private, and (3) Consortium blockchains. This division is of significant interest to us because it provides core ideas for the construction of the blockchain-based task sharing framework presented in this thesis. The blockchain classification discussed in this section can be seen in Table 2.2.

Public blockchains are un-permissioned platforms with no restriction to access the infrastructure. In fact, anyone with an Internet connection can join the network and become an authorized node. Participants in public blockchains may have transparent access to data, issue transactions, and participate in mining activities like transaction verification and block creation. Public blockchains typically have open source design and use consensus mechanisms with high levels of decentralization such as Proof of Work (PoW) and Proof of Stake (PoS). One of the upsides of public blockchains is that they do not abide by the concept of ownership. In fact, if the founding node leaves the system, the network will continue to run on the remaining nodes with the same operational characteristics. On the other hand, public blockchains are slow and do not scale well due to the use of highly

Level 0	Level 1	Level 2
1. Membership	1.1 Access control	1.1.1 Public un-permissioned blockchain 1.1.2 Private permissioned blockchain 1.1.3 Consortium permissioned blockchain
	1.2 Identity Management	1.2.1 Anonymous 1.2.2 Public Key Infrastructure (PKI)
2. Consensus	2.1 Governance	2.1.1 Decentralized 2.1.2 Hierarchical 2.1.3 Centralized
	2.2 Type of consensus	2.2.1 Proof of work (PoW) 2.2.2 Proof of stake (PoS) 2.2.3 Proof of authority (PoA) 2.2.4 Proof of Burn (PoB) 2.2.5 Proof of Capacity (PoC) 2.2.6 Endorsement based 2.2.7 Hybrid
3. Ledger	3.1 Permissions	3.1.1 Open read/write permissions 3.1.2 Identity-based read/write permissions
	3.2 Type of database	3.2.1 LevelDB 3.2.2 CouchDB
4. Smart contracts	4.1 Logic structure	4.1.1 Hard-coded logic 4.1.2 Programmable logic
	4.2 Coding Language:	4.2.1 Single dedicated language 4.2.2 Multiple general purpose languages
	4.3 License:	4.3.1 Open source 4.3.2 Closed source

Table 2.2. Taxonomy of blockchain systems: membership, consensus, ledger, and smart contract classification

democratic consensus to guarantee the required levels of decentralization. Popular applications that make use of public blockchains are decentralized payment systems and cryptocurrencies. However, public blockchains may also be a good fit

for applications that require fully decentralized, transparent, notarized trust models for open collaboration among the nodes of a network.

Private blockchains are permissioned platforms ruled by a single node inside the network. Likewise, their public counterpart, private blockchains operate as peer-to-peer networks. However, the ruling node is in charge of establishing roles for participants that may limit permissions over resources, access to information, and allowed behavior within the network. Private blockchains are invite-only systems meaning that they typically use a membership service that enforces some kind of access control in the network. On the upside, private blockchains may offer higher transaction throughput compared to public blockchains due to their hierarchical role-based structure. On the downside, private blockchains may also have very limited use cases because of the use of highly centralized consensus that is counterintuitive to the idea of blockchain networks. The source code used in private blockchains is typically closed and proprietary meaning that the participants are unable to audit it. Private blockchains are a good fit for applications that require cryptographic services in combination with access control schemes where transparency is not necessary and the ruling central node can be fully trusted. As a side note, private blockchains operate on a much smaller scale than public blockchains, typically inside a single organization.

Finally, consortium blockchains are platforms with a combination of features that resemble both, public and private blockchains. In consortium blockchains, a set of nodes belonging to multiple organizations (possible competitors with different operational logic and incentive structures) join together to form a decentralized network with a role-based structure and distributed control over access, permissions, security, and resources within the network. The governance of consortium blockchains is shared among the participating organizations moving away from the centralized management approach used in private blockchains. Consensus protocols are controlled by predefined consensus nodes that validate transactions and create new blocks. The consensus nodes implement mechanisms to

enforce data consistency that are normally more efficient than those available in public blockchains. The design of consortium blockchains is typically open source with a modular architecture that offers plug-and-play components and services that adapt easily to many distributed solutions. Consortium blockchains are popular in many industrial applications such as supply chain, healthcare, banking, and more recently communications engineering. The Hyperledger Fabric Project and its multiple blockchain solutions are good examples of that.

When comparing public, private, and consortium blockchains we shall consider four important aspects: (1) access control scheme, (2) consensus mechanism, (3) transparency of the trust model, and (4) level of decentralization. In terms of access control, public blockchains are un-permissioned platforms where participants can freely join the system with an Internet connection. There are neither access control mechanisms nor high-ranked users. On the other hand, private and consortium blockchains are permissioned networks where participants may have to reveal full or part of their identities to gain access to the platform. There is a membership module that verifies identities and accepts or rejects attempts to join the network. There is also a set of high-ranked nodes with veto power to make decisions within the network. Regarding consensus schemes, public blockchains use highly democratic consensus mechanisms such as Proof of Work (PoW) where every node has a vote to make decisions in the network. This, however, comes at the expense of high decision times, low transaction throughput, and high CPU and energy consumption. On the other hand, Private and Consortium blockchains use consolidated power methods such as Proof of Stake (PoS) or Practical Byzantine Fault Tolerant (PBFT) algorithms. Consensus mechanisms of this type are operated by a few designated nodes in the network with no mining process involved. They typically are computationally inexpensive and may provide faster decision times with less CPU and energy consumption. In terms of transparency, data and transactions are completely open to anyone participating in public blockchains. Also, they guarantee complete anonymity within the platform. On the flip side, transactions in private and consortium blockchains are not completely visible to

the public unless some level of clearance is granted. Also, they do not enforce the idea of anonymity since a node needs to reveal full or part of its identity to gain access to the network. In terms of concentration of power, public blockchains have high levels of decentralization emulating a libertarian utopia where all nodes have the same rank, rights, and freedoms with no concentration of power inside the network. On the opposite side, private blockchains are, in practical terms, centralized networks characterized by a single leader that dictates the rules of the system reassembling a digital dictatorship. Finally, a consortium blockchain uses a division of power approach where the main components of the architecture (i.e. identity management, transaction verification, block generation), are managed by a consortium of organizations participating in the network. This approach is reminiscent of a federated digital democracy that is a good compromise between the scalability of the network and the reliability of the trust model when compared to other blockchain technologies. A summary of the main features of public, private, and consortium blockchains can be seen in table 2.3.

2.3 The Hyperledger Fabric Framework

In 2015, the Linux Foundation launched the Hyperledger Project to develop generic blockchain platforms with flexible frameworks that can adapt to different decentralized solutions across the industry spectrum. Rather than pushing for a fixed blockchain standard, the Hyperledger Project supports a community of software developers and industry leaders in charge of developing blockchain technologies with an open source approach and flexible intellectual property rights that will encourage industries to adopt the standard over time. Hyperledger Fabric is one of the blockchain platforms within the Hyperledger Project that consists of a general-purpose blockchain with a consortium design that offers lightweight, fast, scalable, and secure blockchain services[2]. The platform is already very popular among a wide variety of economic sectors that include supply chain, finance, healthcare, and retail commerce.

	Public	**Private**	**Consortium**
Decentralization: Social analogy:	Libertarian society	Dictatorship	Federated democracy
Access control: Access Permission: Membership service: Ranked members:	Un-permissioned No No	Permissioned Yes Yes	Permissioned Yes Yes
Consensus: Governance: Type of consensus: Computation demand: Energy demand: Tx throughput: Tx latency: Scalability:	Distributed Proof-based High High Low High Low	Centralized Voting-based Low Low High Low High	Federated Voting-based Low Low High Low High
Transparency: Data: Tx visibility: Participant's identity:	Public Visible Anonymous	Private Partially visible Disclosed	Private Partially visible Disclosed

Table 2.3. Public vs Private vs Consortium Blockchains

Concretely, Hyperledger Fabric is a permissioned blockchain that provides membership, consensus, ledger, and smart contract services for the implementation of multi-purpose decentralized solutions. The technical design of Fabric relies on (1) an open source approach, (2) modular architecture, (3) identity-based roles, and (4) flexible programmable logic that make it stand out from other blockchain platforms. First, Fabric is open source, meaning that the copyright owners grant users the right to use, modify, and distribute Fabric code without restrictions. There is a community of more than 35 partner companies and thousands of developers around the world working on exploiting Fabric benefits. Second, Fabric has a modular architecture with configurable membership, consensus, ledger, and smart contract services that may fit a wide variety of business cases. Third, Fabric uses

identity-based roles meaning that the nodes in the blockchain hold digital identities that are mapped to permissions over resources, access to data, and allowed behavior in the network. This feature brings a level of trust among network participants even though some of them might be market competitors with antagonistic interests and inherently untrustworthy relationships. Finally, Fabric offers flexible smart contracts that can be programmed using general-purpose languages such as *Java, Go, and Node.js*. This is different from other platforms that may use their specific languages for smart contract code (i.e. Ethereum's Solidity programming language).

2.3.1 Fabric Model

In this section, we discuss the core aspects of the Hyperledger Fabric model. To do that, we have divided the model into four parts: (1) membership, (2) consensus, (3) ledger, and (4) smart contracts. A detailed explanation of these topics is presented next.

Membership. Hyperledger Fabric is a permissioned blockchain in which participant nodes must hold a known identity. Identities are assigned to every network component, namely, organizations, peers, and client applications. Fabric makes use of a Public Key Infrastructure (PKI) framework to create, use, store, and revoke cryptographic identities. A PKI is a combination of hardware, software, and advanced cryptography that assigns a pair of public/private keys to network participants through an enrollment process that results in the creation of digital certificates. The x.509 standard is the most common format for digital certificates in PKI systems. PKIs provide trust services for secure communication over the Internet and are widely used in applications such as online e-commerce, banking, and confidential messaging. Trust services are built upon three fundamental attributes: (1) confidentiality, (2) integrity, and (3) authenticity. Confidentiality guarantees that no external actors can reach data being interchanged between two nodes during peer-to-peer communication. Integrity, on the other hand, guarantees that data circulating in the PKI network is tamper-proof. And also, if data is compromised it

is tamper-evident. Finally, authenticity is the assurance that nodes participating in peer-to-peer communication are legitimate. The authentication process involves a digital certificate and a private key to identify nodes and validate the data being transferred.

The PKI framework consists of four basic components: (1) Certificate Authority (CA), (2) Central Directory, (3) Certificate Management System, and (4) Policy Statement. First, Certificate Authorities are trusted components in the PKI system in charge of approving, issuing, and publishing public keys and digital certificates assigned to network nodes. CAs execute a public key encryption-decryption algorithm (e.g. RSA method) to sign digital certificates with the CA public-private key pair. A public key encryption-decryption algorithm is a cryptographic technique that allows nodes in a network to securely communicate private information over a network using available public information through a scheme of cryptographic signature verification. Second, Central Directories are secured physical locations where key pairs are stored and managed. Third, the Certificate Management System is the module in charge of delivering digital certificates issued by the CAs to their recipients. Finally, the Policy Statement is an official document issued by the PKI and available to the public with an explanation of the requirements, operational details, and lifecycle of the PKI services.

The Hyperledger Fabric framework uses a PKI-based membership service to verify the identities of participants in the blockchain network. The participants (i.e. administrators, peers, and client applications) are required to have a public-private key pair issued by a trusted CA to sign transactions submitted to the network. As a result, Fabric can grant identity-based permissions over resources, access to data, and allowed behavior in the network. The permissioned nature of Fabric is an attractive option for application scenarios where security and privacy are core to the use case being implemented.

Consensus. The core idea of Distributed Ledger Technology (DLT) is that network records must be kept distributed among the participant nodes. To achieve that, it is necessary to use mechanisms that can verify and order the records that are added to the ledger. A consensus mechanism is a transaction validation algorithm to reach agreements on the data circulating in an unreliable network. Consensus must keep track of data flows, build transaction blocks, and give assurance that the data in the network is shielded from security threats. The most well-known consensus mechanism to date is the famous Proof of Work algorithm that was proposed by Satoshi Nakamoto in his 2008 paper "Bitcoin: A Peer-to-Peer Electronic Cash System"[33].

In permission-less blockchains like Bitcoin, any node can take part in the consensus mechanism. They can verify, order, and pack transactions into blocks that will be proposed to the network later. Consensus in permission-less blockchains relies on probabilistic algorithms to guarantee data consistency to a certain level of confidence. However, probabilistic approximations may be vulnerable to ledger forks in which a few nodes may receive blocks with transactions arranged in a different order which may introduce a dissenting view of the history of the network. Hyperledger Fabric, on the other hand, is a permissioned blockchain that operates on a slightly different structure. Consensus in Fabric networks means more than just reaching agreements on data circulating in the network. More specifically, Fabric consensus has a role in the entire lifecycle of transactions from proposals and endorsements to ordering and committing events. Moreover, Fabric consensus can be explained at two different levels: (1) consensus at the peer level, and (2) consensus at the ordering service level.

The consensus at the peer level runs on the idea of endorsements which is a mechanism of checks and validations that take place during the transaction lifecycle before transaction proposals reach the ordering service. More specifically, an endorsement is a validation made by endorsing peers that consists of the simulation of transaction proposals using their copy of the chaincode. If the simulation

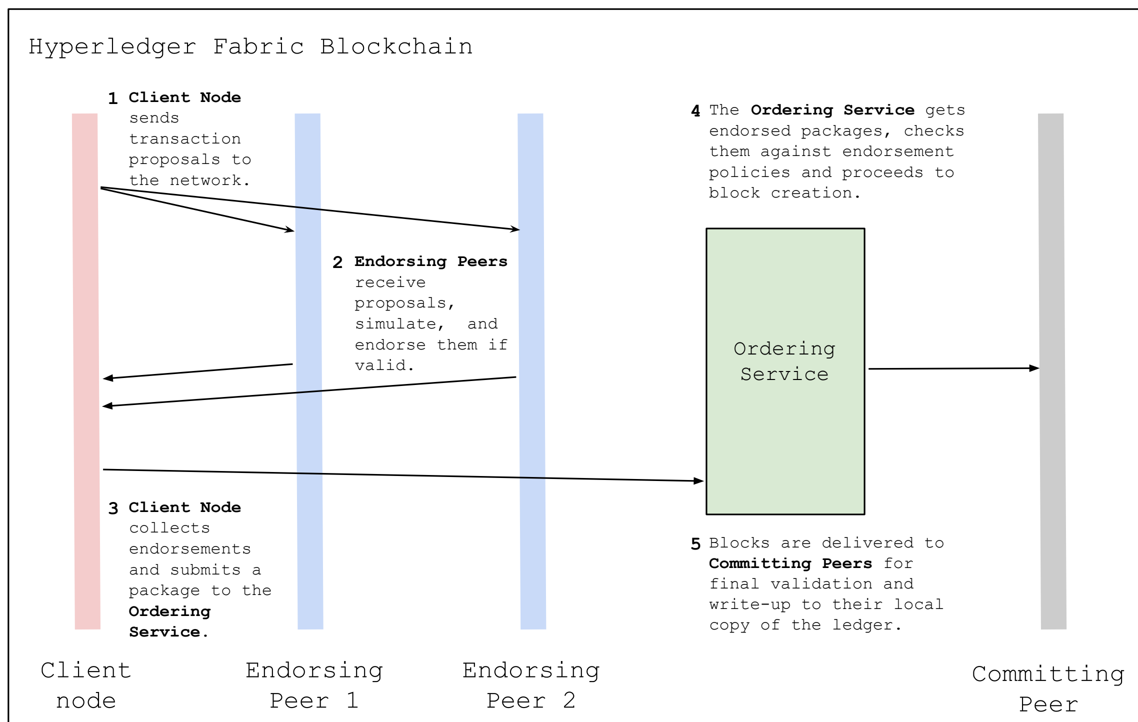

Figure 2.5. Transaction flow in the Hyperledger Fabric blockchain framework

produces the same output as the proposal, endorsing peers then sign the proposal and return the signed transaction to the client that originated it. Endorsement policies are set up in the configuration files of the network where the administrator defines the trusted nodes that get to simulate and sign transactions in the blockchain. The idea of endorsements inside the transaction lifecycle is shown in Figure 2.5.

The consensus at the ordering level starts with the verification of transaction endorsements. After transactions are signed by endorsing peers and returned to invoking clients, they prepare a package that is submitted to the ordering service. The package includes transaction results and collected endorsements. At the ordering service level, the endorsements in the package are validated against the endorsement policies defined in the network. If the transaction complies with the policies, it is declared valid and the consensus process continues with transaction

ordering, block creation, and broadcast of new blocks to committing peers. At this point, the ordering service may look like a centralized module in the network in charge of transaction ordering and block creation. However, the ordering service is composed of a set of ordering peers that operate on a leader-follower structure using the Raft consensus algorithm. The availability of the ordering service module relies on the crash-tolerant nature of Raft. In summary, the consensus model in Fabric networks does not only involve the agreement on the validity of a batch of transactions and their order but rather, it is a multi-stage transaction verification process of checks and balances that guarantees the reliability of transactions circulating in the blockchain network.

Ledger. Ledger is a financial term that describes a book with transaction records arranged in a specific order. In Fabric terms, the ledger is the physical database that holds the indisputable history of the network and stores facts and data about the network. Facts and data may change through the execution of transactions, however, transaction records are unique and immutable. The ledger in a Fabric network is accessible on a channel basis, that is, only peers participating in a channel have access to the channel's ledger. This is a deliberate design choice to keep transactions private in the network. Moreover, the ledger in a Fabric network has two components: (1) the world state database, and (2) the blockchain database.

The world state database holds facts (i.e. states) of objects in the Fabric network. The states are key-value pairs that are updated by the ordering service every time a transaction has completed its lifecycle. The world state is particularly useful because programs can query current object states without looping through the whole history of the network. Client applications can submit transaction proposals to query, update, and delete states in the world state, however, only accepted transactions will induce changes in the database. Depending on the configuration setup, the world state may provide a sophisticated set of operators for efficient queries and storage of state values. Every time a new peer is created in the network, a peer-local instance of the world state is created alongside the peer. If a peer crashes, the

world state can be regenerated from the transaction records of the network when the peer restarts.

The blockchain database keeps the historical records of all transactions (valid and invalid) in the Fabric network. More specifically, the blockchain database holds the record of how Fabric objects arrive at their current states. Transactions in the blockchain database are packed into data blocks connected chronologically to form a blockchain structure. The header of a block contains the cryptographic hash of transactions in the block and the hash of the previous block's header. In this way, blocks are cryptographically connected and their data cannot be changed or tampered with. Fabric makes use of SHA256 hash function for the hash operations. Also, the blockchain database in the Fabric network is implemented as a file in contrast to the world state which is a NoSQL database.

Smart Contracts. Smart contracts are immutable pieces of software running on a blockchain that define objects in the network and instructions on how to modify them. Smart contracts are typically installed, stored, and approved by blockchain participants before they are ready to be used. Smart contracts follow through the business logic of a blockchain application, specifically, it determines how data and transactions are represented in the network. Smart contracts are used to automate the execution of pre-determined agreements in a network without the involvement of a central authority overseeing the process. In a nutshell, smart contracts are the executable logic of a blockchain application that uses transaction functionality to update data in the network.

In Fabric context, smart contracts implement the terms, definitions, and processes that rule the interaction of network participants. Smart contracts may submit transaction proposals to query, create, update, or delete states of an object in the network. The blockchain database will register transaction records whether proposals are valid or not. The world state database, on the other hand, will only get updated if transactions are valid. Smart contracts have configurable endorsement

policies that define organizations and peers in the network that must approve (i.e. digitally sign) transaction proposals to be declared valid. The endorsement scheme in Fabric is different from other blockchains in which endorsements can be issued by any node in the network. Endorsement policies are configurable so that only *trusted* organizations can approve transactions according to the business logic of the network. In Fabric blockchains, the term chaincode is used to identify smart contracts that have completed their implementation lifecycle and are available to be used in the network. However, the distinction between the terms is often overlooked so they are mostly used interchangeably. Smart contracts in Fabric can translate business models into blockchain software using Software Development Kits (SDK). Fabric SDKs support a wide set of Application Program Interfaces (APIs) and a few general-purpose programming languages (i.e. Java, Javascript, Golang) for the implementation of Fabric-based solutions.

2.3.2 Fabric components

Peers. Fabric peers are software containers that package code, tools, libraries, and dependencies that make the blockchain network run reliably. Peers are the physical hosts of ledgers and smart contracts which are the core components of any blockchain-based solution. Peers are equipped with a set of APIs that allow external applications to access the services they provide. Peers, smart contracts, and ledgers hold a very close relationship with each other in the sense smart contracts are pieces of code with the ability to access ledgers that physically reside on peers. Moreover, the job of a peer is to host instances of smart contracts and ledgers to provide redundancy protection for business logic and data in the network. Peers are instrumental in eliminating the single point of failure problems in Fabric networks, one of the big promises of any distributed ledger technology. Peers can physically handle multiple ledgers and smart contracts at the same time. However, it is also possible to have peers in the network with no installed smart contracts or running ledgers.

Organizations. Organizations are collections of peers that may operate under individual incentive schemes that do not necessarily complement one another. Typically, organizations are enterprises that own infrastructure in a marketplace and execute individual business logic. Fabric is designed to provide a trust model for organizations with competing business structures (possibly market competitors) so that they may transact with each other in a transparent and secure platform. In a typical Fabric setup, multiple organizations may come together to form a consortium that runs and maintain a blockchain network. Fabric enforces the idea of decentralized management by allowing networks to be formed and managed by the participating organizations rather than a single central administration point.

Channels. When multiple organizations and their peers join together to interact and trade objects in a Fabric network, they do so within the limits of a communication space called a blockchain channel. In Fabric context, a channel is a network mechanism by which a set of peers can communicate and interact privately. Within a channel, peers agree to run and maintain a distributed ledger of records with isolated rules and transaction logic (i.e. smart contracts) although not all peers in an organization may join the same channel. However, it is typical that all peers join at least one channel. We can think of channels as private blockchains within a bigger decentralized ecosystem. In fact, Fabric can be seen as a collection of multiple independent blockchain channels that are put together to serve the logic of a distributed solution.

Applications. The outside world can interact with Fabric blockchains through the use of client applications. In Fabric context, client applications are pieces of software that make use of Software Development Kit (SDKs) and Application Programming Interfaces (APIs) to connect to peers in a blockchain channel and execute chaincode transactions that may induce changes on the channel ledger. SDKs power applications with the necessary functionality to connect to peers, generate transactions and submit proposals to the network. SDKs also allow client applications to receive alerts when transactions get ordered, validated, and committed

to a channel. Applications are typically developed to serve one organization in a channel but can also connect to peers inside other organizations in the network. The Fabric platform currently supports SDKs for developing client applications in three general-purpose programming languages: Java, Javascript, and Golang.

Membership Service Provider. The Membership Service Provider (MSP) is the component of the Fabric framework in charge of identity validation in the network. The MSP not only provides identity verification services but also manages roles in the blockchain. The default implementation of MSP makes use of the traditional Public Key Infrastructure model in which *X.509* certificates issued by trusted Certificate Authorities are used as digital identities in the network. In standard setups, organizations in a Fabric consortium can configure their MSPs. Moreover, the configuration parameters (i.e. MSP identifier, CA root of trust, intermediate CA certificates, the path to cryptographic material, etc) must be specified in the genesis block of the channel when the network is being bootstrapped to life. The MSP does not provide conventional web services but rather implements a record of trust in the blockchain to authenticate who is allowed to participate in the network. The MSP is also capable of mapping accepted identities into roles in the network with predetermined permission clearance to perform actions at the peer, channel, and organization levels.

Ordering Service. As discussed earlier, consensus is the mechanism used in distributed systems to agree on the validity of transactions and their order in the ledger. To do that, Fabric introduces a component called Ordering Service which consists of a set of designated peers in the network (i.e. ordering peers) in charge of transaction ordering and block generation. To avoid single-point-of-failure problems, the ordering peers form a cluster with a leader-follower structure. If the leader peer crashes for any reason, a new leader is elected using the Raft algorithm. The consistency of the service is guaranteed by the fault-tolerant nature of the Raft method. Unlike public blockchains, Fabric relies on deterministic consensus meaning that any set of transactions that have been ordered by the ordering service

is guaranteed to be unique when added to the ledger. The duties of the ordering service are separated from chaincode execution and endorsement routines that happen at the (regular) peer level. The separation prevents bottlenecks in the network and contributes to the good performance and scalability of the system.

State Database. Fabric supports two types of state databases for ledger implementation: (1) LevelDB and (2) CouchDB. LevelDB is the default implementation of ledgers in Fabric networks that come embedded inside peer nodes. LevelDB ledgers store data as key-value pairs but only support simple operations such as key, key range, and composite queries. LevelDB might be a simple, lightweight, and convenient database for some applications, however, it comes with limitations to manage large and complex data. CouchDB, on the other hand, is an open-source NoSQL database that runs completely decoupled from peers in Fabric networks. It offers advanced query options based on JSON data modeling and chaincode indexing that make queries efficient for large data sets with complex structures. The type of database must be consistent across network peers and the setup must be decided before launching the network. In fact, switching databases is currently not supported by Fabric due to incompatibility issues between the binary-based data format used in LevelDB and the JSON-based data model used in CouchDB. Regardless of the type, a state database must keep two important records: the current state of objects in the network (i.e. world-state database), and the history of transactions that determined them (i.e. blockchain database).

Chaincode. Chaincode is an inside term coined by the Hyperledger Fabric project to identify *smart contracts* in blockchains with programmable logic such as Fabric. In Fabric terms, chaincode is a piece of immutable software that contains the business logic of blockchain solutions. Chaincode controls the lifecycle of network objects and implements a set of common terms, data, rules, definitions, and processes agreed by network participants before they start transacting with one another. Every piece of chaincode has an associated endorsement policy that defines trusted organizations that must sign chaincode transaction proposals before

they are declared valid. The chaincode model in Fabric supports the development of a wide variety of use cases in multi-purpose programming languages such as Java, Javascript, and Golang. After development, a chaincode becomes available in a channel following a process called chaincode lifecycle. The chaincode lifecycle includes packaging, installation, approval, commitment, and initialization of the software on all the network peers. Chaincode is at the core of Fabric architecture and is normally the main focus of solution architects and developers.

Management tools. The Hyperledger Fabric project supplies a set of management tools for the administration and maintenance of Fabric networks. Hyperledger Cello (1) is an application that provides a visual interface to facilitate the creation and administration of Fabric ecosystems. Cello runs on top of container-based clusters and is commonly integrated into commercially available blockchain-as-a-service platforms. Hyperledger Explorer (2) is a web application to monitor performance metrics in Fabric networks. Explorer has an overview of the components of the network: organizations, peers, chaincode, etc. In fact, Explorer can view blocks, query data, and submit transaction proposals to the blockchain ecosystem. Finally, Hyperledger Composer (3) is a modeling framework that facilitates the integration of existing business logic with Fabric blockchains. Composer framework helps network architects in modeling Fabric-based solutions with a friendly tool-set that speeds up the design and delivery process of Fabric-solutions.

Fabric components in action. In this section, we will examine how the components of the Fabric architecture come together to form a blockchain network. First, let's suppose that three organizations, ***R0, R1*** and ***R2***, agree to form the consortium ***X*** to collaborate in a new Hyperledger Fabric blockchain network ***N***. Organization ***R0*** will contribute with the ordering peer ***O***, and the other two organizations, ***R1 and R2***, will contribute with peers **P1 and P2** respectively. The ordering service is the first administration point of the network and it should be present in every proposed consortium. The identities of the organizations and their peers are validated by certificate authorities ***CAO, CA1*** and ***CA2***. The configuration and policies of the

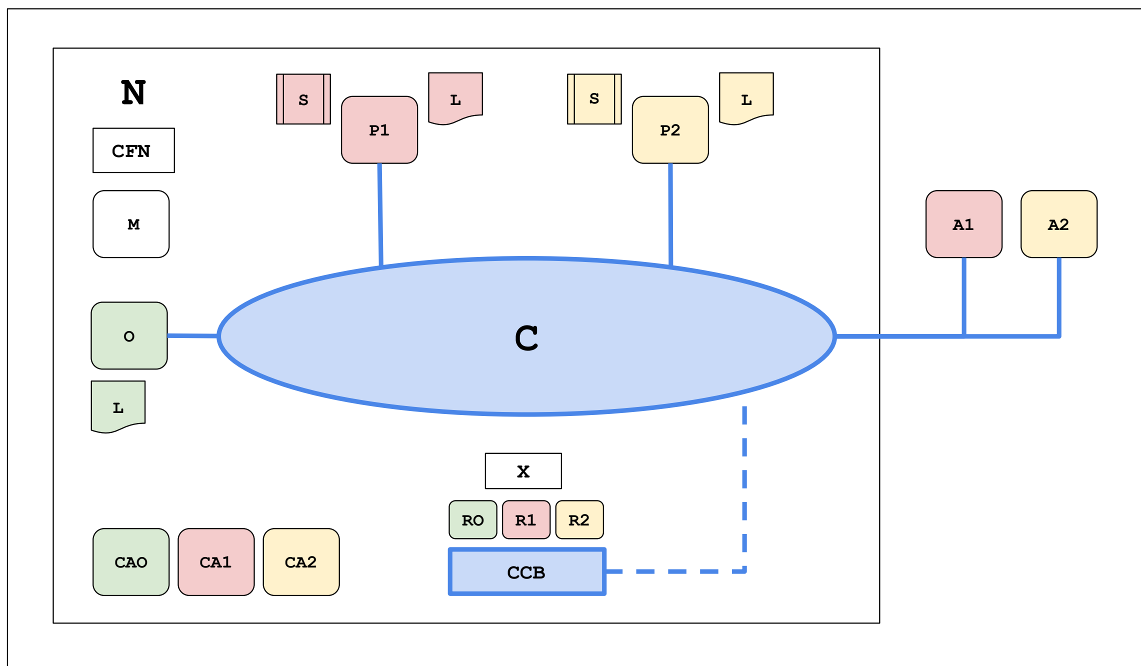

Figure 2.6. Illustration of a fully formed Fabric network with its main components in action

consortium are agreed upon by the organizations beforehand and put together in the network configuration file ***CFN***. Next, organizations ***R0, R1*** and ***R2*** create an application channel ***C*** so that participant peers can interact with each other through transaction proposals. The configuration of ***C*** is agreed by ***R0, R1*** and ***R2*** and is written into the channel configuration block ***CCB***. Although at this point no peers have physically joined channel ***C***, it is considered to be created by the only existence of the channel configuration block. Peers ***P0, P1*** and ***P2*** can join ***C*** after ***CCB*** is ready. When a peer joins a channel, it automatically receives a copy of the channel ledger (i.e. world state database and blockchain database) that holds the current state and history of the network. In fact, when ***P0, P1*** and ***P2*** join ***C***, they receive a copy of ***L***, the ledger associated to ***C***. It is important to note that there is no restriction for peers to join multiple channels. Indeed, in production scenarios, it is common to have peers join multiple channels and keep copies of multiple ledgers. After peers have joined channel ***C***, chaincode ***S*** (i.e. smart contract) can

be packaged, installed, approved, committed, and initialized on the multiple organizations and peers of network ***N***. We can think of **S** to be physically installed on the peers but logically hosted on ***C***. After ***S*** is deployed and ready for production, client applications ***A1*** and ***A2*** (physically installed on ***R1*** and ***R2*** respectively) can be used to interact with the Fabric ecosystem. Although ***A1*** and ***A2*** are technically not part of the blockchain network, they have recognized identities issued by ***CA1*** and ***CA2*** and are allowed to invoke transactions proposals to query or update ***L***. Finally, a management tool **M** (e.g. Hyperledger Explorer and/or Hyperledger Cello) can be installed on top of the Fabric network for administration and maintenance purposes. An illustration of the fully formed network explained in this section is shown in Figure 2.6.

2.4 Blockchain Technologies and Next-Gen Communication Networks

As we move towards next-gen communication networks in which massive machine-type communication, ultra-low latency, and enhanced broadband capabilities will be the dominant trend, the complexity of the emerging standard will prompt fundamental engineering problems related to resource management, heterogeneous collaboration, access control, and network security models. The integration of blockchain technologies with next-gen networks may bring to the picture prime features of blockchain such as decentralization, self-governance, tamper-proof data, transparency, and security attributes to solve some of the problems mentioned earlier. In this section, we review current research work related to the intersection of blockchain technologies with communication networks and the impact that distributed ledgers may have on the design of network architecture, network services, and user applications for the envisioned 5G standard.

Blockchain and network architecture: Blockchain-based architecture for 5G networks opens new possibilities for the implementation of new architecture models that may fulfill the specifications of the standard. Blockchain networks may bring

intelligence and cryptographic services into architecture technologies such as Network Virtualization, Software-Defined Networking, Cloud Computing, and Edge Computing models. In the cloud computing department, the authors' Zhou et al. propose a *Clean-room Security Service Protocol* based on a consortium blockchain that monitors the implementation of illegal software on the user's end in a cloud computing setting. The authors claim that their protocol reduces the security threats of the network and has major potential in trusted network computing systems[52]. Next on, the authors Sharma et al. propose a blockchain-over-cloud solution designed for cloud service providers so they can share their computing resources through smart contracts implementation[43]. Also, the authors Ricci et al. propose a decentralized cloud storage mechanism based on STORJ blockchain platform. The mechanism breaks down the files that need storage services and then distributes the pieces of data to multiple nodes in the blockchain[40]. Finally, the authors' Xu et al. designed a machine-learned-based energy management framework embedded in blockchain smart contracts to minimize the energy consumption of cloud data centers. The authors test their system on Google cloud clusters and show that their proposed system is able to reduce the energy cost of the data center when compared to other benchmark approaches[50].

Blockchain technologies also present an opportunity to introduce improvements into edge computing architectures. For example, the authors Kotobi and Sartipi[20] propose a blockchain-based architecture for wireless networks in smart city environments. The system introduces a caching mechanism to enhance the communication capabilities of the network and enable massive data collection for smart city applications. A distributed ledger is used to secure communication between smart city applications and IoT sensors deployed in the network. Moving on, The authors Rawat et al. design a blockchain-based architecture for IoT networks that runs on a wireless network virtualization model that allows multiple virtual network operators with different resource requirements to be accommodated in a blockchain-based ecosystem. The model leverages edge computing to provide processing and storage services to limited IoT devices in the network. The authors

focus their analysis on the double spending problem of wireless resources (i.e. frequency) for which they propose a blockchain-based solution approach[23]. Finally, the authors propose a decentralized data management protocol that implements a distributed access control manager for end users in mobile edge networks[53]. The authors make use of blockchain and off-blockchain data storage to give end users decentralized control of their data without the need for a third trusted party in charge of storage services. A summary of the papers discussed in this section is presented in Table 2.4.

Blockchain and network services. Efficient administration of available infrastructure in communication networks must balance tensions between the growing demand for resources (forced by the massive number of connected devices) and the limited infrastructure owned by network operators. As seen in earlier sections, blockchain can provide a trust model for resource collaboration among a consortium of network operators. The authors Mafakheri et al. present a blockchain-based distributed authentication mechanism and infrastructure sharing model for small cells in a 5G network[28]. They use a consortium blockchain to store user subscription information in a distributed ledger maintained by the core nodes of the participant operators. In this way, the operators have a distributed access control list to securely provide services to subscribers in the consortium. The proposed model aims to reduce CAPEX and OPEX indexes and optimize the economics of the consortium. In the same realm, the authors Rawat et al. propose a spectrum sharing model for virtual wireless networks based on blockchain that allows primary spectrum owners to lease their available resources to virtual mobile operators[38]. The blockchain network provides a trust model for device-to-device interactions of primary spectrum owners and virtual mobile operators. Next on, the authors Ling et al. propose a blockchain-based radio access network (BRAN) with a decentralized architecture to manage access and infrastructure sharing in mobile networks[24]. The BRAN architecture uses smart contract logic for network authentication and a variation of PoW consensus based on the identity of the participant to validate interactions between nodes. The authors present a set of test results

Use case	Focus	Contribution	Authors
Security	Cloud computing	Blockchain-based *Clean-room Security Service Protocol* for software implementation in cloud settings.	L. Zhou, G. Wang, and X. Xing
Resource management	Cloud computing	Blockchain-based solution for cloud datacenters to share computing resources over a smart contract-enabled mechanism.	S. Sharma, L. Ahuja, and D. Goyal
Storage	Cloud computing	Decentralized cloud storage mechanism based on the STORJ blockchain platform.	J. Ricci, I. Baggili, and F. Breitinger
Energy management	Cloud computing	Smart contract-based energy management framework to minimize energy consumption in cloud data centers.	C. Xu, K. Wang, and M. Guo
Smart Cities	Edge computing	Blockchain-based caching mechanism for processing massive data in smart city wireless applications.	K. Kotobi and M. Sartipi
IoT networks	Edge computing	IoT architectrure based on edge computing and blockchain that implements a wireless virtualization model for network operators.	D. Rawat, and Md. Parwez, and A. Alshammari
Data management	Edge computing	Decentralized data management protocol for mobile edge networks.	G. Zyskind, D. Zekrifa, A. Pentland and O. Nathan

Table 2.4. Application of blockchain technologies in next-gen network architecture

that suggest that their decentralized model performs better than the centralized counterpart in terms of latency and throughput provided by the network.

The authors Kotobi et al. propose a medium access control protocol based on blockchain in which multiple cognitive networks compete in an auction game for a priced portion of the available spectrum. They introduce the use of a cryptocurrency in the network for spectrum transactions and a distributed database that is visible to all participants with the intention to enforce transparency in the auction mechanism [18, 19]. Moving on, the authors Fukumitsu et al. propose a distributed online data storage mechanism based on a peer-to-peer network that does not need central storage services. In the proposed model, the data generated by users is divided into many parts and distributed to the blockchain network using anonymous communication so that no data remains in the user node. The security features provided by blockchain prevent attacks from malicious nodes who may want to tamper with user information stored in the blockchain ledger [12]. Similar to that, the authors Wang et al. propose a decentralized storage model framework for cloud datacenters based on Ethereum blockchain and attribute-based encryption. They presented a demo implemented on the Ethereum test network Rinkeby to show that their proposed scheme is feasible. Finally, next-gen networks can leverage the cryptographic nature of blockchain for security and privacy services. In this context, the authors Kroonmaa et al. filed a US Patent for a data authentication system based on a distributed hash tree infrastructure supported by blockchain. The model uses a blockchain-based hash tree structure formed from digital input records that reduces the possibility to make unauthorized changes in the data structure. A summary of the work discussed in this section is presented in Table 2.5

Blockchain and end-user applications: 5G networks open the opportunity for developing new end-user applications with the potential to become game changers for many economic sectors. On the other hand, blockchain technology could be the facilitator to solving some of the challenges related to the implementation of 5G applications, especially in areas such as decentralized management, security, and privacy. For example, the authors Biswas and Muthukkumarasamy propose a blockchain-based security framework for IoT applications in smart city ecosystems [4]. Their framework runs on a four-layer model that includes a database layer

Use case	Focus	Contribution	Authors
Network economics	Resource management	Infrastructure sharing model based on consortium blockchain for network operators. The model optimizes CAPEX and OPEX indexes of the network consortium.	B. Mafakheri, T. Subramanya, L.Goratti and R. Riggio
Network economics	Spectrum management	Spectrum leasing model based on blockchain for virtual network operators.	D. Rawat, Md. Parwez, and A. Alshammari
Infrastructure sharing	Resource management	Blockchain-based ratio access network (BRAN) for secure access control and resource sharing in mobile networks.	X. Ling, J. Wang, T. Bouchoucha, C. Levy and Z. Ding
Network transparency	Spectrum management	Auction-based mechanism for spectrum sharing in mobile networks. The model runs on a blockchain network with a cryptocurrentcy model for spectrum transactions.	K. Kotobi and S. G. Bilen
Data Storage	Storage management	Distributed online data storage mechanism based on a peer-to-peer network. The model breaks down user data in several parts that are stored in a distributed ledger maintained by the network.	M. Fukumitsu, S. Hasegawa, J. Iwazaki, M. Sakai and D. Takahashi
Data Storage	Storage management	Storage model for cloud datacenters based on Ethereum blockchain for secure data encryption.	S. Wang, Y. Zhang, and Y. Zhang
Network Security	Authentication management	US Patent: Data authentication system based on a distributed hash tree infrastructure supported by blockchain.	A. Kroonmaa, A. Buldas, and J. Pearce

Table 2.5. Application of blockchain technologies in next-gen network services

for the integration of blockchain ledgers. The blockchain network serves as a common platform for secure and distributed interaction of smart city devices. The authors claim that their framework could deliver the grounds for direct collaboration between citizens and local governments. Next on, the authors Li et al. introduce a decentralized on-demand energy supply model based on microgrids that provides energy to miner devices in blockchain-based IoT networks. The energy allocation problem is modeled as a Stackelberg game to find optimal profit strategies for both the microgrids and the miners. The proposed system is supposed to alleviate energy limitations for decentralized IoT networks in the presence of different consensus mechanisms.

In the smart industry sector, the authors Fernandez-Caramés et. al propose a smart inventory management system that runs on Unmanned Aerial Vehicle (UAV) infrastructure[10]. The UAVs can scan products with Radio-Frequency IDentification (RFID) tags and send items data to a consortium blockchain that aggregates multiple actors in the supply chain structure. The network validates entries using smart contracts and enforces traceability of the items. The system automates tedious inventory tasks that are typically performed manually by humans. Moving on to smart transportation, the authors Zhang et al. propose a Vehicular Ad-hoc Network (VANET) based on blockchain for self-organizing data transmission for autonomous driving vehicles[51]. The model uses smart contracts installed on a consortium blockchain network to securely process and store vehicle data that feed assisted driving and safety applications. The authors present performance evaluation data that suggest a significant reduction in the consensus convergence time of their blockchain network compared to other traditional blockchain platforms. Finally, the authors Ramani et al. put forward a distributed data management model for the healthcare industry. The system runs on Ethereum blockchain that provides privacy services for patients' data so that it can only be accessed by authorized parties[37]. The authors present a demo network with the logic of the proposed data model.

Use case	Focus	Contribution	Authors
Secure smart cities	IoT networks	Blockchain-based framework for secure IoT applications in smart city ecosystems.	K. Biswas and V. Muthukkumarasamy
Smart energy	IoT networks	Decentralized on-demand energy supply model based on microgrids and blockchain for IoT networks.	J. Li, Z. Zhou, J. Wu, J. Li, S. Mumtaz, X. Lin, H. Gacanin, S. Alotaibi
Smart industry	Unmanned aerial vehicles	Blockchain-based inventory management model that runs on Unmanned Aerial Vehicle infrastructure for smart factories.	T. Fernández-Caramés, O. Blanco-Novoa, M. Suárez-Albela, P. Fraga-Lamas
Smart transportation	Vehicular Ad-hoc Networks	Blockchain-based VANET for secure and self-organizing data sharing between among autonomous driving vehicles.	X. Zhang and X. Chen
Smart healthcare	Data management	Ethereum-based distributed data management model for the healthcare industry.	V. Ramani, T. Kumar, A. Bracken, M. Liyanage and M. Ylianttila

Table 2.6. Application of blockchain technologies in next-gen user applications

3 Methodology and Design Elements

3.1 The case for Hyperledger Fabric-based Services for 5G Networks

The deployment of 5G networks will require a huge amount of financial investment from telecommunication companies. To cope with that, network operators must find new revenue streams to level off the capital expenditure on infrastructure and technical support. Fortunately, the adoption of the 5G standard will also open the door to rethinking the ownership model for network operators that, in our opinion, is outdated and not well equipped to handle emerging business cases in the IT sector. Given the intense capital expenditure, infrastructure cooperation seems to be an alternative to lower infrastructure costs without affecting the proper delivery of applications and services for businesses and their consumers. In traditional ownership frameworks, telecommunication companies may reach rigid agreements in a centralized style where senior management teams get together and discuss static cooperation mechanisms that, in many cases, may include infrastructure sharing in any form or shape. Unfortunately, centralized frameworks for infrastructure ownership do not adapt very well to the needs of emerging 5G business models that may need more dynamic and self-organizing approaches.

The decentralized services offered by blockchain technologies may become the facilitator for automated resource management schemes that adapt better to the complexities of sophisticated 5G applications. As discussed earlier, blockchain networks provide a trust model for (1) record-keeping the truth of a network (i.e.

tamper-proof distributed ledgers), and (2) secure interactions for network participants (i.e. immutable smart contracts). Unlike centralized networks in which trust comes directly from the reputation of a central node, the trust model in blockchain networks relies on distributed consensus algorithms and advanced cryptography to maintain truthfulness in the system. Smart contracts, on the other hand, are pieces of immutable software installed on the blockchain. They hold definitions, rules, data, and processes agreed by network participants to generate transactions that update the ledger. The self-execute nature of smart contracts allows network operators to automatically enforce cooperation agreements previously negotiated by participant organizations. Blockchain technologies might represent the opportunity to empower companies with a surfeit of resources to participate in automatic on-demand trading schemes that may otherwise be done statically. A comparison between traditional and blockchain-based marketplaces for telecommunications operators is summarized in Table 3.1.

	Traditional Marketplace	**Blockchain-based Marketplace**
Type of governance	Centralized trade	Decentralized trade
Source of trust	Reputation of a central entity	Distributed consensus algorithms and advanced cryptography
Type of contracts	Mutual and static	Consortium and on-demand
Negotiation terms	Subject to interpretation by the involved parties	Pre-negotiated and immutable
Trade execution	Manual	Self-execute

Table 3.1. Comparison between traditional and blockchain-based marketplaces for telecommunications operators

The Hyperledger Fabric platform is a permissioned blockchain that offers membership, ledger, consensus, and chaincode services that support the implementation of multi-purpose distributed solutions. Fabric is a lightweight, low latency, and high throughput blockchain that may open opportunities for the implementation of distributed trading markets for network operators in the telecommunications sector. In blockchain-based marketplaces, network operators can trade scarce resources under competitive conditions in self-governed and secure platforms. From a market perspective, network resources might become scarce either because the resource supply is short as in the case of the frequency spectrum, or because production costs are high as in the case of the manufacture of computer processors. Moreover, the marketization of network resources in the telecommunications sector may impact network services such as wireless spectrum allocation, network slicing, and the lease of computer resources. Moving from static methods to on-demand smart contracts for infrastructure sharing may require blockchain-based market models that can summon multiple market players to trade network resources in both seller (provider) and buyer (requester) roles.

A Hyperledger Fabric-based multilateral marketplace would consist of a set of network operators willing to participate in the system and a set of smart contracts that facilitates the negotiation and trading of network resources. The Hyperledger Fabric platform brings in two key features to support the proper operation of the marketplace. First, it maintains a distributed record of all transactions in the network, and second, it supports the distributed execution of on-demand cooperation agreements through the use of immutable smart contracts. The low latency and high throughput characteristics of Fabric would make the execution of smart contracts in the marketplace highly dynamic and ductile. In a Fabric-based marketplace, market applications would send trading proposals to the blockchain network. The proposals contain requests for resources whose details depend on the implemented market model (i.e. regular auctions, reverse auctions, or any other form of resource allocation scheme). When trading transactions are received by the blockchain, the marketplace logic is automatically activated through the means

of installed smart contracts. The terms of trading schemes are pre-negotiated by the market players and are subsequently embedded into smart contracts available in the market network. Trading models might be based on traditional resource allocation approaches that maximize global objectives defined by the players (e.g. multiprocessor task scheduling problems), or game theory approaches where individually rational players seek to maximize their utility functions (e.g. auction problems). In this work, we use the first approach although the latter may seem more intuitive for distributed market models like the one discussed in this section. If trading proposals are successful, market players will endorse the transactions so that they become ready to be added to the blockchain network. Moreover, the ordering service will pack successful transactions into blocks that are later appended to the distributed ledger of the network. Once appended, trading transactions become part of the official truth of the market. This trading framework guarantees that market transactions are transparent and immutable. Also, it might provide an effective trust model for market players to deal with the contentious nature of the relationships among participants in the marketplace.

Although resource sharing is not a new idea and has been used extensively in previous generations of communication networks, sharing models typically consisted of long-term static agreements where operators accept to share network infrastructure without considering the short-term dynamic variations of resource demands. Also, the aggressive competition among market players in the telecommunications sector makes it almost impossible to have a trusted central intermediary to deal with tensions among market competitors. This is true especially when it comes to decisions on pricing and allocation of resources. On the other hand, the idea of decentralized marketplaces for infrastructure sharing has the potential to relieve market tensions, however, it requires the active participation of multiple actors in the industry including vendors, network operators, and regulators. In summary, blockchain platforms, Hyperledger Fabric, in particular, may power decentralized, transparent, and secure ownership models for emerging business applications based on 5G networks that require flexible and on-demand resource

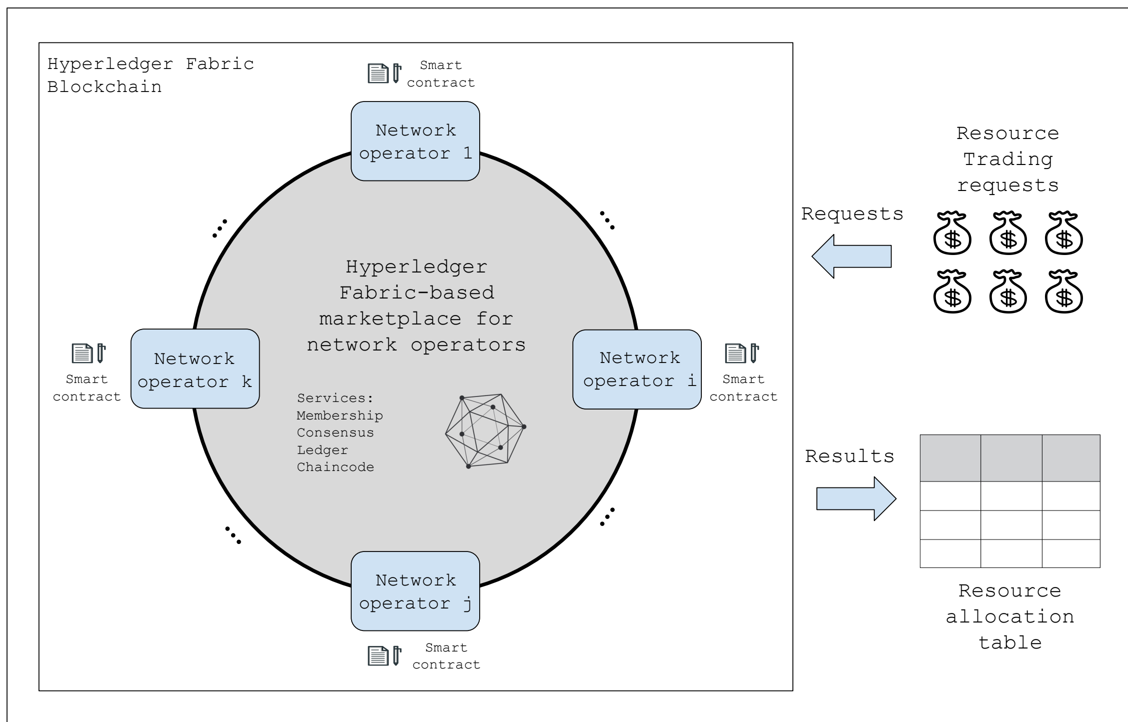

Figure 3.1. Illustration of the Hyperledger Fabric-based marketplace for network operators

delivery. An illustration of the Hyperledger Fabric-based marketplace for network operators is presented in Figure 3.1.

3.2 A Hyperledger Fabric Framework for Task Sharing Collaboration in MEC

With the popularization of Open-RAN and network function virtualization technologies in 4G and 5G systems, the current trend in regards to MEC deployment tends to favor the utilization of vendor-neutral Commercial Off-The-Shelf (COTS) servers at the edge level. Typically, each COTS unit has a limited amount of resources in terms of computing power, memory, and storage capabilities. In real implementations, MEC networks are often filled with a massive number of low-performance COTS nodes that may struggle to deal with a massive number of

end-user requests during peak load hours. In fact, the utilization of network resources at the edge level may still be unbalanced to a great deal when compared to their total installed capacity. There is an increasing interest to develop efficient mechanisms for resource administration at the edge level so that the vision of MEC networks can be fully exploited. In that direction, the benefits of the integration of blockchain technologies with MEC networks can be summarized as follows:

Notarization. The tamper-proof nature of blockchain records allows authenticity and integrity confirmation of network participants and their actions.

Ownership. The double-spending resilience provided by blockchain networks guarantees consistency of the ownership history meaning that for every blockchain object, there is always a unique owner at any given point in time.

Provenance and Chain of Custody. Cryptographic identifiers for blockchain objects hold origin and chain of custody information in a way that allows subdivision and combination of items.

The proposed framework integrates the open standards and management technology of Hyperledger Fabric into the architecture of MEC to put forward a powerful, lightweight, and extensible technology foundation for secure management of computing resources in MEC networks. The framework is built on top of the Fabric platform for a few reasons. First, Fabric is a permissioned blockchain that guarantees identity verification of network participants before granting access permissions to the network. Second, Fabric allows peer-to-peer communication between network nodes that eliminates the need for a central intermediary with control of network interactions. Third, every node holds a local ledger that contains transaction records. The ledger content may be different across the network depending on the access rights granted to a node. More specifically, only nodes with the same access rights can share a ledger with a particular set of transaction records. Finally, Fabric uses smart contracts to submit transaction proposals to the

network and a notary service to avoid double-spending problems. Moreover, Fabric does not rely on mining techniques to verify transactions and reach consensus in the network. In fact, transaction proposals are verified by a separate set of peers with special hierarchy called the ordering service.

The main goal of our framework is to provide a Service-Oriented Architecture (SOA) for secure and private task sharing collaboration specially designed for MEC networks. To achieve that, edge server nodes are instrumented in such a way that Hyperledger Fabric is responsible for their monitoring and management. The MEC network integrates Fabric protocols for peer-to-peer communication between edge server nodes and remote management tools that monitor the network resources. Specifically, the system is adapted to dynamically discover servers, allow peer-to-peer communication, and provision the logic for task sharing services among server nodes. The proposed MEC framework supports decentralized computing services with dynamic task scheduling and unified management of resources while maintaining security capabilities with a special focus on the integrity and confidentiality of tasks and their data. In summary, our framework proposes a decentralized, model-driven, and heterogeneous communication architecture that abides by the following principles:

Node Autonomy: edge server nodes can perform autonomous connection, discovery, learning, and execution of tasks through the integration of Hyperledger Fabric services.

Open Collaboration: edge server nodes can share tasks among themselves and also between them and the cloud. This provides elastic networking, computing, and storage capabilities.

In our framework, the relationship between any two edge server nodes changes from master-slave to equal partnership through the peer-to-peer communication nature of the Hyperledger Fabric platform. Also, the nodes selected to process tasks

in the network are dynamically chosen based on their availability and the number of computing resources locked in for that purpose. The selection procedure involves a series of optimization routines running on the busy server end. The cloud level will have an overview of the candidate nodes that volunteer to accept tasks. In addition to that, the system will leverage Fabric privacy features so that tasks themselves are not released to candidate nodes or the cloud. Rather, they are only shared between busy servers and the nodes selected to complete the tasks.

The framework is built under the assumption that a collection of $S + 1$ nodes, namely, $\mathcal{S} = \{1, \ldots, S\}$ edge server nodes plus the cloud level, get together to form a consortium in the Hyperledger Fabric platform. At any given time, a busy server node $i \in \mathcal{S}$ submits a task sharing service request to the blockchain network to get additional resources to process pending computational tasks. After the request is in place, the cloud level shortlists a set of candidate servers $\{1, \ldots, j, \ldots, k\} \in \mathcal{S}$ willing to process the pending tasks based on the information provided by the monitoring tool running at this location. With the information in hand, the busy server solves an optimization problem to select a few nodes from the candidate list that will become the service providers. After node selection is complete, the task sharing scheme (i.e. upload, remote execution, download of results) is overseen by the blockchain network through the execution of smart contracts available on the platform. The illustration of the proposed framework can be seen in Figure 3.2. The task sharing request and the sharing scheme that comes after will follow the sequence below.

(1) The edge server node $i \in \mathcal{S}$ owns a set of computational tasks that require processing. The server is short of resources and requests the task sharing service to the blockchain network.
(2) The cloud level runs a monitoring tool with an overview of the network. It can shortlist a set of candidate nodes $\{1, \ldots, j, \ldots, k\} \in \mathcal{S}$ to take on the tasks owned by server i.
(3) With the information provided by the cloud, the busy server i allocates the queued tasks among a few selected candidate nodes. The selection

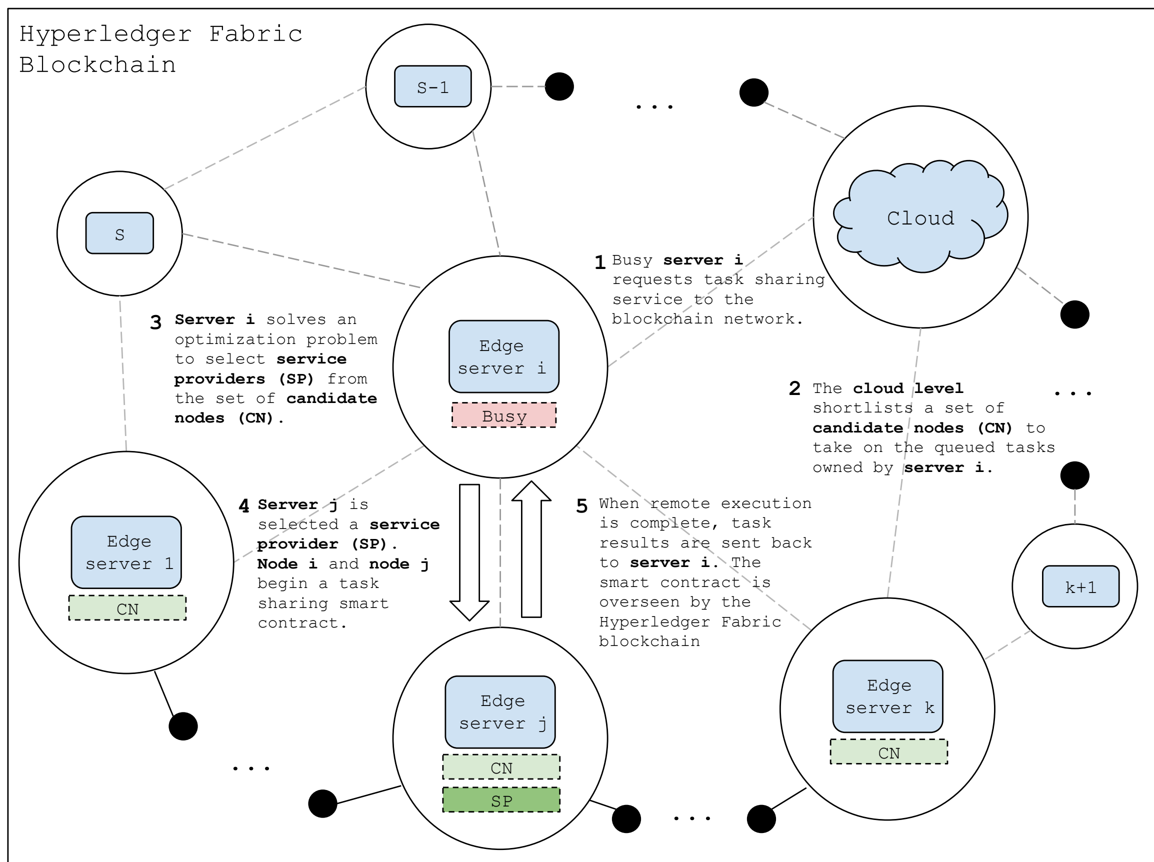

Figure 3.2. Illustration of the Hyperledger Fabric framework for task sharing collaboration in MEC networks

procedure is based on the optimization of particular objectives at the busy server end. The selected nodes become service providers.

(4) Finally, the busy server i and every selected candidate j invoke Fabric smart contracts to carry through the sharing process.

(5) The sharing process will complete its lifecycle following the rules of the Hyperledger Fabric framework.

It is important to note that the nodes involved in task sharing transactions must follow the ledger rules dictated by the Fabric platform. More specifically, ledgers that receive updates as a result of task sharing transactions are only visible to nodes participating in the associated sharing service. This feature guarantees confidentiality and privacy of tasks and their data during the process. On a separate note,

the smart contracts installed on the blockchain will implement the task sharing logic in the system and be responsible for processing transaction requests. Any edge server node can activate a smart contract by submitting an initiation transaction to the network. When the initiation is approved, the smart contract enters into an active state and starts receiving transaction proposals from the server nodes in the network.

Fabric peers are separated into two different run-times and three distinct types: endorsing, committing, and ordering peers. Endorsing and committing peers are executed on the same run-time. Ordering peers, on the other hand, are run on a completely separate run-time. This feature is consequential with the modular architecture of Hyperledger Fabric that separates chaincode execution and consensus. The three types of Fabric peers and their run-times are described as follows:

Run-time 1. This run-time involves endorsing and committing peers. Committing peers may not have a copy of the chaincode installed on their premises meaning that they are not capable of running smart contract functions. However, they do maintain a ledger with all records. In contrast, endorsing peers must have a copy of the chaincode installed on their premises to simulate transactions using smart contract functions and prepare transaction endorsements with signature verifications based on the results.

Run-time 2. This run-time involves ordering peers responsible for receiving endorsed transactions and packaging them into blocks. They broadcast transaction blocks to all other peers in the network to update their local copy of the ledger. The ordering peers keep track of all transactions (valid and not valid), however, the ledgers belonging to endorsing and committing peers only contain valid transactions.

As discussed earlier, the Fabric platform assigns network roles based on node types (i.e. endorsing, committing, and ordering peers). In fact, execution, ordering, and commitment of transactions are completely decoupled thanks to the node hierarchy introduced by Fabric. In particular, the separation of execution and ordering duties enables peer nodes to handle multiple task sharing requests simultaneously. Parallel execution increases the efficiency of processes running at the peer level and accelerates the delivery of transaction proposals to the ordering service. The division of labor saves peer nodes from transaction ordering and consensus. At the same time, ordering peers are freed from transaction execution and ledger maintenance duties. This kind of parallelism based on role bifurcation allows peers to run independently of verification by ordering nodes. The trick effectively caps the processing power required for authentication given that peer nodes do not have to trust all ordering nodes in the network and vice-versa.

3.2.1 Preliminary Performance Evaluation

In blockchain context, performance evaluation is the process of measuring relevant performance metrics of a system under test[39]. Typical metrics of interest are related to transaction response time and transaction throughput. Benchmarking, on the other hand, is the process of defining standard metrics and testing scenarios to make fair comparisons across different systems. In this section, we use the Caliper Blockchain Evaluation Tool to measure the performance of different versions of the Hyperledger Fabric platform. Caliper supports the testing of different blockchain systems such as Hyperledger Fabric, Hyperledger Burrow, Hyperledger Composer, Ethereum, FISCO BCOS, Hyperledger Iroha, and Hyperledger Sawtooth. The benchmark metrics used by Caliper are Read Latency, Read Throughput, Transaction Latency and Transaction Throughput[36]. A brief explanation of these metrics is presented next:

Read Latency (RL). RL is defined as the time difference between the submission time of a transaction request and the time when the reply is received. This metric is expressed in seconds $[s]$.

$$RL = ReplyReceivedTime - SubmissionTime \tag{3.1}$$

Read Throughput (RT). RT can be defined as the ratio of total read operations on the ledger per unit of time. This value is expressed in reads per second $[rps]$.

$$RT = \frac{TotalReadTransactions}{TotalTime} \tag{3.2}$$

Transaction Latency (TL). TL is the time difference between the transaction confirmation time and the submission time. It is important to note that confirmation happens when transactions get appended to the blockchain ledger. In that case, they are said to be committed to the network. TL is expressed in seconds $[s]$.

$$TL = TransactionConfirmationTime - SubmissionTime \tag{3.3}$$

Transaction Throughput (TT). TT is defined as the ratio of total committed transactions per unit of time. This rate is expressed in transactions per second $[tps]$.

$$TT = \frac{TotalCommittedTransactions}{TotalTime} \tag{3.4}$$

To finalize this section, Figures 3.3 and 3.4 show the results of a benchmark evaluation of Hyperledger Fabric versions 1.4.0 and 1.4.1 using the Caliper Benchmark Tool. The benchmark for the tests is Fabric Marbles with state database set to GoLevelDB. The total number of transactions pushed into the Fabric network is 10000 and the transaction arrival rate increases from 500 tps to 6000 tps in 50 unit increments. The parameters being measured are transaction latency, CPU usage, and memory usage.

Observation 1: Figure 3.3 shows the behavior of transaction latency in seconds $[s]$ with respect to the transaction arrival rate given in transactions per second $[tps]$. For every tps point, the average of the latency of transactions is measured. It can be observed that transaction latency values for Hyperledger Fabric 1.4.0 varies from 15 seconds to 25 seconds in unstable form, whereas the transaction latency values for Hyperledger Fabric 1.4.1 remains more stable in this regard. The average of the

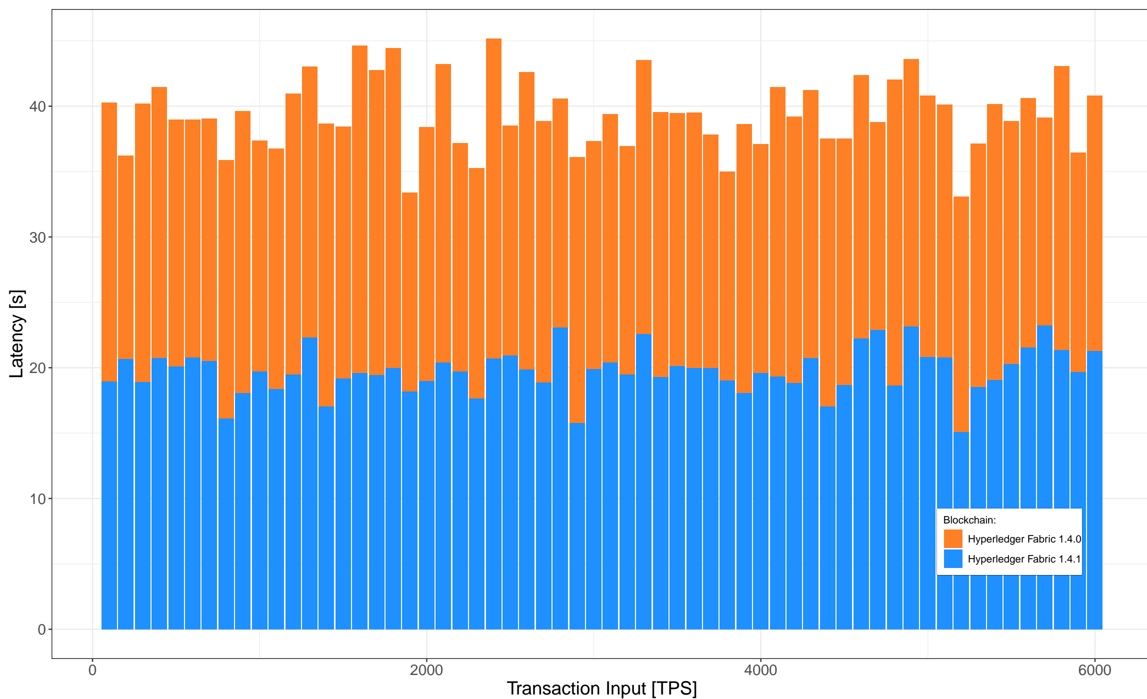

Figure 3.3. Variation of the Transaction Latency versus Transaction Arrival Rate for Hyperledger Fabric versions 1.4.0 and 1.4.1.

latency of transactions is approximately 20 seconds for both systems under test.

Observation 2: Figure 3.4 (top) shows the evaluation of the system's memory usage in megabytes [MB] versus transaction arrival rate given in transactions per second [tps]. For every tps point, the average of used memory is measured. It can be observed that for both Hyperledger Fabric versions, memory usage behaves similarly across the evaluation interval with a very stable form and an average of approximately 158 MB for both systems under test.

Observation 3: Figure 3.4 (bottom) shows the evaluation of CPU Usage measured as a percentage [$\%$] versus transaction arrival rate given in transactions per second [tps]. For every tps point, the average of CPU usage is measured. Similar to Observation 2, it can be examined that for both Hyperledger Fabric 1.4.0 and 1.4.1, CPU usage behaves almost identically with an average of 5.9% for both systems under test.

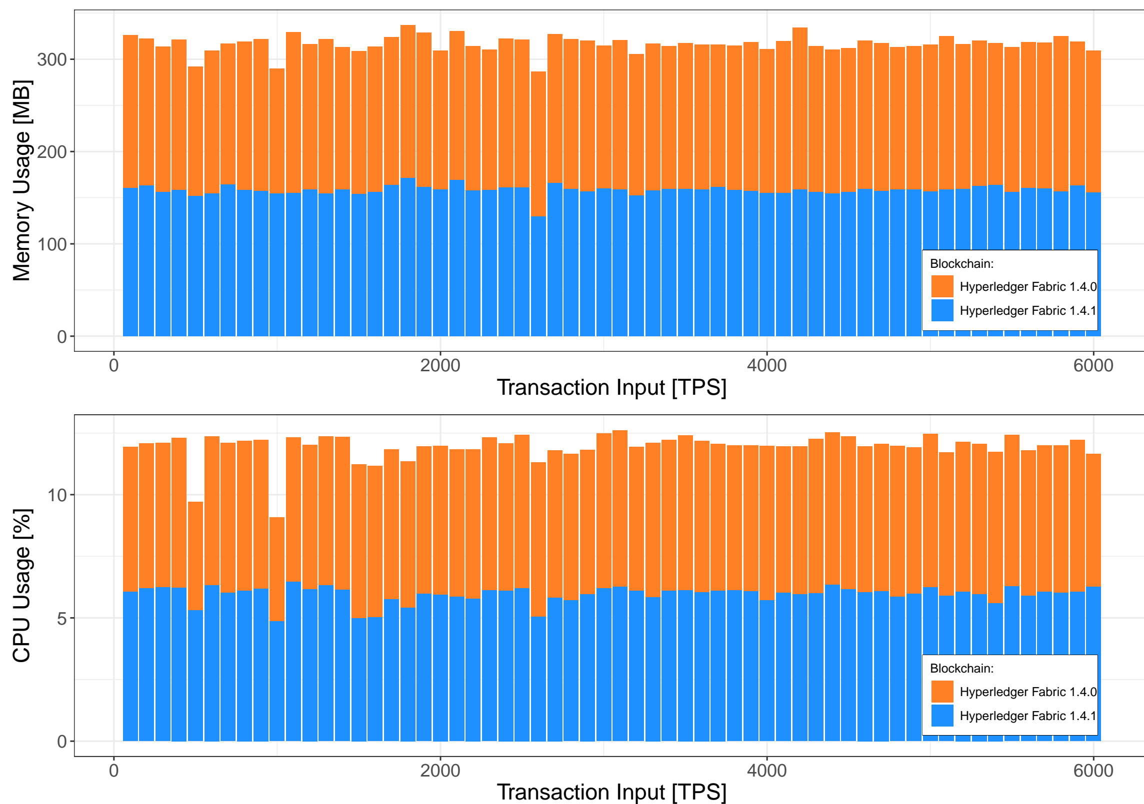

Figure 3.4. Variation of memory usage (top) and CPU usage (bottom) versus Transaction Arrival Rate for Hyperledger Fabric versions 1.4.0 and 1.4.1.

3.3 Design of the Task Sharing Model at the Blockchain Level

As discussed in previous sections, Hyperledger Fabric is an open-source platform that offers blockchain services for multi-purpose distributed solutions. Fabric offers a modular architecture of peers, membership, ledger, consensus, and smart contracts with enhanced cryptographic attributes for decentralized applications. In our model, a set of edge servers and the cloud level get together to form a consortium in the Hyperledger Fabric platform. A set of computational tasks that require processing services are Fabric objects, and the logic of the proposed sharing scheme is embedded in the chaincode installed on the network peers.

The proposed solution accommodates a Membership Service Provider (MSP) in charge of identity authentication and role management in the network. The players participating in the network (i.e. organizations, peers, and client applications) receive standard *x.509* identities and a public-private key pair from trusted Certificate Authorities (CAs). The MSP maps identities to network roles that define permissions over resources, access to information, and allowed behavior in the blockchain. In addition to the MSP, the network implements an Ordering Service module (OS) in charge of transaction verification and consensus scheme. Finally, the logic of the task sharing scheme is implemented by smart contracts installed on the network peers. The smart contracts are governed by the Hyperledger Fabric architecture and the policies defined by the task sharing logic. Together, they guarantee secure and private communication between nodes involved in the sharing scheme. The dynamics of our task sharing solution can be described as follows:

Service Request: A resource-constrained edge server node owns a task-flow computational job with a set of precedent-dependent tasks. The server requests the task sharing service to the blockchain network.

Discovery of Candidate Nodes: Upon receiving the task sharing request, the monitoring tool sitting at the cloud level shortlists a set of candidate nodes that may offer the task sharing service.

Selection of Service Providers: With the help of optimization tools, the busy server determines the optimal allocation of tasks among the set of candidate nodes. The allocation maximizes the utility function of the server and minimizes the makespan of the schedule of the set of tasks. The allocated servers are called service providers.

Task Sharing Contract. The busy server and every service provider start a task sharing contract that oversees uploading, remote execution, and return of results of tasks subject to the sharing service. The contracts will use the blockchain services

of the Fabric platform.

Right after the Fabric network is launched, the ordering service takes the lead as the first point of administration to set up the network according to the configuration policy agreed upon by participant organizations (i.e. edge service providers and cloud service providers). The configuration policy contains information regarding organizations and peers, the setup of the membership service, and the type of consensus running at the ordering service. Next, the Fabric consortium creates a global channel where the plenary of server nodes and the cloud level may communicate with each other and request task sharing services. The channel implements a distributed ledger of records with the history of submitted transactions to the channel space. Once the global channel is configured, the *Service Terms Chaincode* is installed and committed to the channel. The *Service Terms Chaincode* will track *(i) request, (j) reply, and (k) decision* transactions related to the terms of task sharing services, however, it will not execute the task sharing itself. This feature is an intentional design choice to protect the privacy of the tasks and their data by not publicly exposing them to the network.

The monitoring tool sitting at the cloud level has access to information related to network resources and performance. The module can shortlist a set of available servers that may offer task sharing services to the network. On the other hand, busy servers have an optimization module that can map pending tasks on their end to a few available nodes from the candidate list. Technically, the monitoring and optimization modules (i.e. client applications) are not part of the blockchain, however, they also receive digital identities from Certificate Authorities and are integrated into the network through the use Fabric APIs. Once integrated, the client applications may use the global channel to submit transaction proposals that follow the logic of the *Service Terms Chaincode*. When a busy server requests the task sharing service, it invokes a request transaction on the global channel. The cloud node then uses its monitoring tool to discover available servers that might become service providers after the selection process. With that information in hand, the

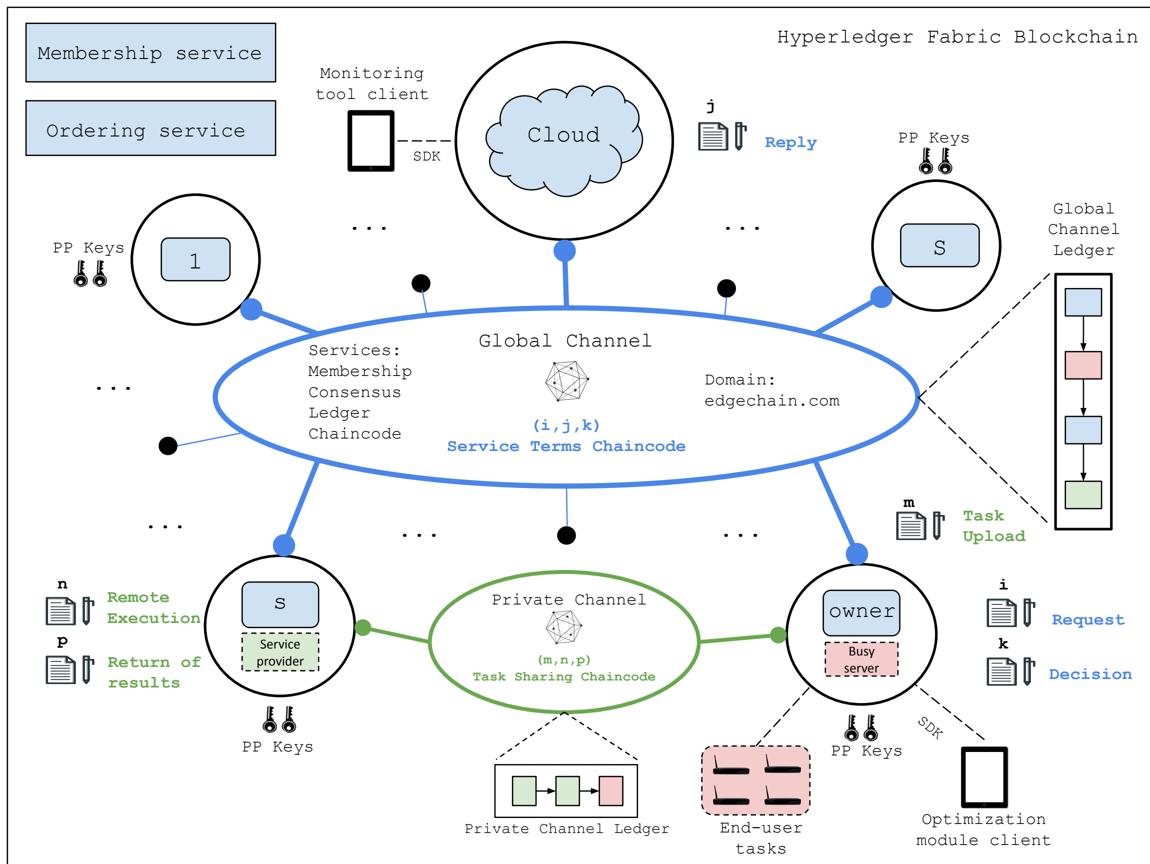

Figure 3.5. Task sharing model at the blockchain level

busy server uses the optimization module to find the optimal subset of servers from the candidate list that will be in charge of remote task execution. Finally, the busy server and the service providers make use of separate private channels equipped with the *Task Sharing Chaincode* to oversee *(m) task upload, (n) remote execution, and (p) return of results* transactions that complete the lifecycle of the task sharing service. The global and private channels are completely decoupled from one another, meaning that they implement their own policy rules, ledgers, and peer-to-peer communication. The decoupled channels in our model separate the two pieces of chaincode in the network (i.e. *Service Terms* and *Task Sharing* Chaincode) to preserve the privacy of tasks and their data. In fact, under the rules of our model, the tasks are only visible to servers partaking in the sharing service

restricting other nodes in the blockchain to have access to this information. The task sharing model at the blockchain level is summarized in Figure 3.5.

3.4 Design of the Task Sharing Model at the Server Level

3.4.1 Mathematical Representation of Task-flow Computational Jobs

As a general rule, computer applications are a combination of software code and relevant data designed to perform specific jobs for end users or other applications. They can be viewed as a set of computational jobs that need to be sequentially executed to produce the desired output. More specifically, a computational job is an independent unit of execution of a computer application that performs computational work with a specific pre-defined objective and is normally associated with a computer process. Jobs can be further divided into tasks that are smaller execution units that may hold a relational dependency with other tasks inside the same job. Computational tasks can range from a single instruction to complicated code structures that may involve data manipulation and optimization problems. The execution of tasks is normally controlled by a job scheduler and once tasks are finished successfully, the computational job is said to be complete.

One of the fundamental design postulates of software development is known as the Acyclic Dependencies Principle (ADP). ADP states that "the dependency graph of packages or components in a software application should have no cycles"[34]. The principle implies that the relational dependencies among components in software applications must flow only in one direction and may never form closed loops. In that context, a task-flow computational job is a unit of execution of a software application that contains a set of tasks and a collection of data that requires a number of computing resources to be completed. Formally, a task-flow computational job is the tuple $\mathcal{J} = \{\mathcal{A}, \mathcal{G}(\mathcal{A})\}$, where $\mathcal{A} = \{1, \ldots, A\}$ is a set of tasks contained by $\mathcal{J}$ and every $a \in \mathcal{A}$ is portrayed by the tuple $(\delta_a, \rho_a, \tau_a)$ where δ_a is size of the task in bytes, ρ_a represents the CPU cycles needed for task execution, and τ_a is the deadline in seconds. $\mathcal{G}(\mathcal{A})$, on the other hand, is a direct control-flow graph with no cycles that

captures the functional relationships (i.e. dependencies) of the set of tasks $\mathcal{A}$ inside the job $\mathcal{J}$. Moreover, the vertices of $\mathcal{G}$ represent the tasks themselves, and the edges represent their precedence dependencies. Every task $a \in \mathcal{A}$ is labeled by its corresponding data size, CPU requirement, and deadline time. At the same time, every edge dependency in $\mathcal{G}$ represents the fact that any task $a \in \mathcal{A}$ must be completed before the execution of the successor task $b \in \mathcal{A}$. Finally, the edge weight $\delta_{a,b}$ represents the size of the data transferred from task a to task b. Figure 3.6 shows the graph representation of a task-flow computational job with general dependent tasks. The mathematical framework of our model is built upon the assumptions laid out in this section.

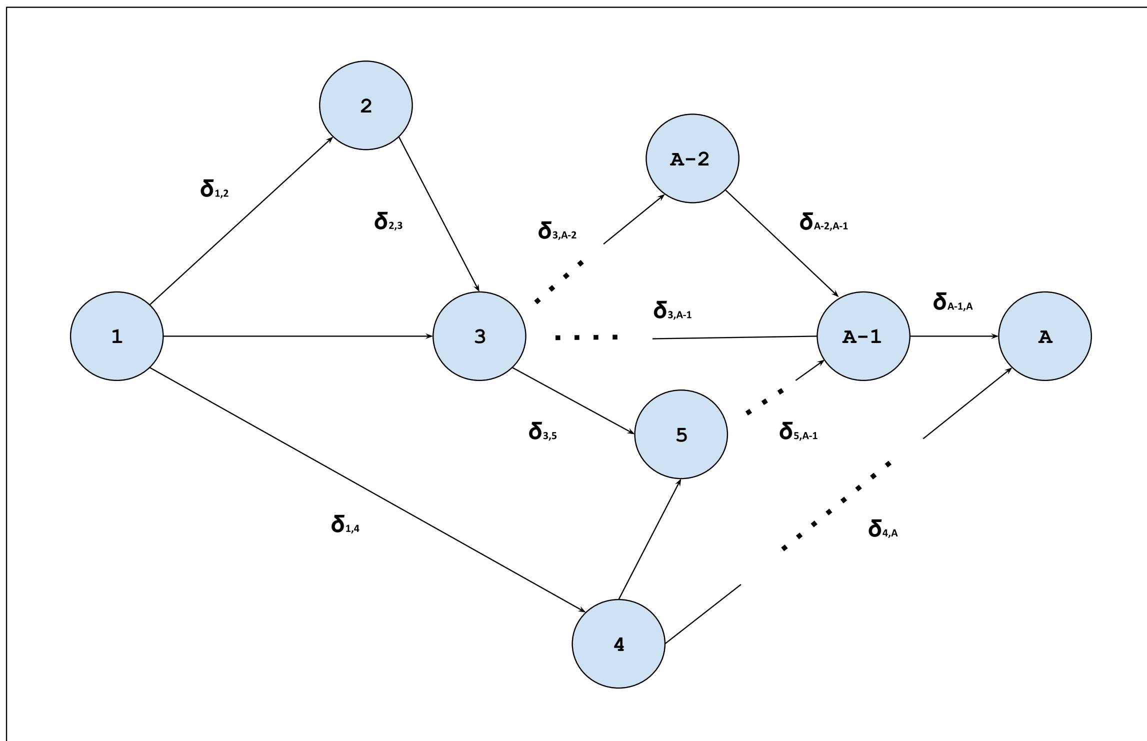

Figure 3.6. Graph representation of a task-flow computational job with general dependent tasks.

3.4.2 Mathematical Framework for the Task Sharing Scheme

Let us consider a MEC network composed of a set of edge servers $\mathcal{S} = \{1, \ldots, S\}$ and a cloud level all running as independent nodes on the Hyperledger Fabric blockchain. Thanks to a decentralized architecture and the use of advanced cryptography, Fabric provides a trust model for a secure and private task sharing service in the network that does not need a central authority in charge of sharing decisions. Let us also consider a server node from the set $\mathcal{S}$ that owns a task-flow computational job $\mathcal{J} = \{\mathcal{A}, \mathcal{G}(\mathcal{A})\}$ that may be decomposed in a set of indivisible tasks $\mathcal{A} = \{1, \ldots, A\}$, and the acyclic graph $\mathcal{G}(\mathcal{A})$. Each $a \in \mathcal{A}$ is the tuple $(\delta_a, \rho_a, \tau_a)$ where δ_a is the size of the task in bytes, ρ_a is the CPU cycles needed for task execution, and τ_a is the deadline in seconds. On the other hand, the acyclic task-call graph $\mathcal{G}$ captures the task precedence dependencies in $\mathcal{A}$.

From $\mathcal{G}$, the general task precedence dependency matrix G is defined in equation 3.5.

$$G^{a,b} = \begin{cases} 1, & \text{if task } a \text{ precedes task } b \\ 0, & \text{otherwise.} \end{cases} \tag{3.5}$$

Also from graph $\mathcal{G}$, the immediate task precedence dependency matrix I is defined as follows in equation 3.6.

$$I^{a,b} = \begin{cases} 1, & \text{if task } a \text{ immediately precedes task } b \\ 0, & \text{otherwise.} \end{cases} \tag{3.6}$$

If a busy server is not able to provide computing services to job $\mathcal{J}$, the server is given the possibility to outsource the execution of the tasks inside $\mathcal{J}$ to available infrastructure in the network. In that case, the busy server can submit a task sharing request to the blockchain asking for computing resources. After the request is in place, the cloud shortlists a set of available nodes $s \in \mathcal{S}$ that are well conditioned to provide task sharing services. The process of shortlisting servers is registered in the availability matrix B defined in equation (3.7).

$$B^{a,s} = \begin{cases} 1, & \text{if server } s \text{ can execute task } a \\ 0, & \text{otherwise.} \end{cases} \tag{3.7}$$

With the information provided by the cloud, the busy server finds the optimal allocation of tasks among available servers that maximizes its utility function and minimizes the makespan of the tasks. Once a sharing decision is in place, the rest of the task sharing process (i.e. *task upload*, *remote execution*, and *return of results*) is managed by the blockchain through the execution of task sharing contracts. More specifically, if a server with CPU-clock frequency f and hardware constant c is set to process a computational task a locally, the execution time and consumed energy can be calculated according to equations (3.8) and (3.9). The term c in equation (3.9) is the server's hardware constant introduced by the Dynamic Voltage and Frequency Scaling (DVFS) computation model for MEC servers[29].

$$t^a = \frac{\rho_a}{f}. \tag{3.8}$$

$$e^a = c\rho_a f^2. \tag{3.9}$$

On the other hand, if a busy server is set to outsource task a to another server s given that the two nodes share a communication channel with bandwidth bw^s, then the total round-trip time of a is defined in equation (3.10). The first term o the equation represents the execution time of a in s. The second term is the uploading time of a from the owner to s given that δ_a is the size of the task. The third term constitutes the downloading time of the results of a from s back to the task owner given that $\sum_b \delta_{ab}$ is the aggregated size of the results of a that will be fed into all its immediate successors b. Finally, the fourth term accounts for the communications overhead introduced by the Hyperledger Fabric platform. Specifically, the overhead is the latency injected by the smart contracts in charge of the task sharing process. The equation (3.12) shows the energy consumed by the owner during the communication process where P is the average power of the busy server.

$$t^{a,s} = \frac{\rho_a}{f_s} + \frac{\delta_a}{bw^s} + \frac{\sum_b \delta_{a,b}}{bw^s} + t_{HFB}. \tag{3.10}$$

$$e^{a,s} = \frac{\delta_a}{bw^s} P. \tag{3.11}$$

From equations (3.8) and (3.10), we define the execution delay matrix D with elements $D^{a,s} \in \mathcal{R}$ that represent the execution time of the tasks $a \in \mathcal{A}$ by servers $s \in \mathcal{S}$. We also define the communication delay matrix C with elements $C^{a,s} \in \mathcal{R}$ that represent the round-trip communication time of $a \in \mathcal{A}$ from the busy server to any other server $s \in \mathcal{S}$ in the system. The utility that a busy server gets by outsourcing task execution to other server nodes is represented by the weighted average of four ratios. The first two ratios are time and energy savings as a result of the task sharing service; They are defined in equations (3.12) and (3.13).

$$r_1 = \frac{t^a - t^{a,s}}{t^a}. \tag{3.12}$$

$$r_2 = \frac{e^a - e^{a,s}}{e^a}. \tag{3.13}$$

Moreover, the last two ratios capture the reputation of the server nodes inside the blockchain network. More specifically, r_3 is the transaction success ratio that represents the portion of transactions successfully endorsed by peers compared to the total number of transactions submitted to the network. This ratio can be interpreted as a measure of the reliability of a node in the blockchain network. On the other hand, r_4 is the normalized block size ratio that measures the relative block size used by a peer compared to the maximum block size allowed by the blockchain. As stated in[45], bigger blocks may lead to a reduction of the latency introduced by Fabric. This holds under the assumption that the transaction throughput in the blockchain is not saturated. In short, this ratio is a measure of how fast a server node manages transactions inside the blockchain. At this point, the elements of the utility matrix U are defined as the scalar product presented in equation (3.14). The entries $U^{a,s} \in R$ are calculated as the weighted average of the performance

ratios $\mathbf{r^{a,s}} = [r_i]$. The weight vector $\mathbf{w}$ represents the performance preferences of the busy server. The sum of the elements of $\mathbf{w}$ is equal to one, $\sum_i w_i = 1$.

$$U^{a,s} = \mathbf{w} \cdot \mathbf{r^{a,s}} \tag{3.14}$$

The allocation variable $X_{a,s} \in \{0, 1\}$ is defined in equation (3.15).

$$X_{a,s} = \begin{cases} 1, & \text{if task } a \text{ is allocated to server } s \\ 0, & \text{otherwise.} \end{cases} \tag{3.15}$$

The time at which every task must be released for remote execution is represented by S_a. Then, the makespan of the schedule of the tasks, $C_{max} \in \mathcal{R}$, is defined in equation (3.16).

$$C_{max} = \max_{a,s}(S_a + (D^{a,s} + C^{a,s})X_{a,s}) \tag{3.16}$$

Also, the aggregated utility of a schedule, $\mathcal{U} \in \mathcal{R}$, is defined in equation (3.17).

$$\mathcal{U} = \sum_{a \in \mathcal{A}} \sum_{s \in \mathcal{S}} U^{a,s} X_{a,s} \tag{3.17}$$

Finally, the dual objective function of the task sharing model at the server level, $\mathcal{F} \in \mathcal{R}$, is defined in equation (3.18).

$$\mathcal{F} = \mathcal{U} - C_{max} \tag{3.18}$$

3.4.3 Problem Formulation

Scheduling Theory is the branch of mathematics that studies algorithmic solutions for task allocation problems in sets composed of autonomous agents. The task sharing problem analyzed in this section considers a set of autonomous edge servers $\mathcal{S} = \{1, \ldots, S\}$ running as independent nodes on the Hyperledger Fabric blockchain. Some of the servers are available to share their processing resources to execute a computational job $\mathcal{J} = \{\mathcal{A}, \mathcal{G}(\mathcal{A})\}$ owned by one of the agents in the system. The job is composed of a set of discrete computational tasks $\mathcal{A} = \{1, \ldots, A\}$. The tasks may have different sizes, CPU requirements, and deadlines. Also, their

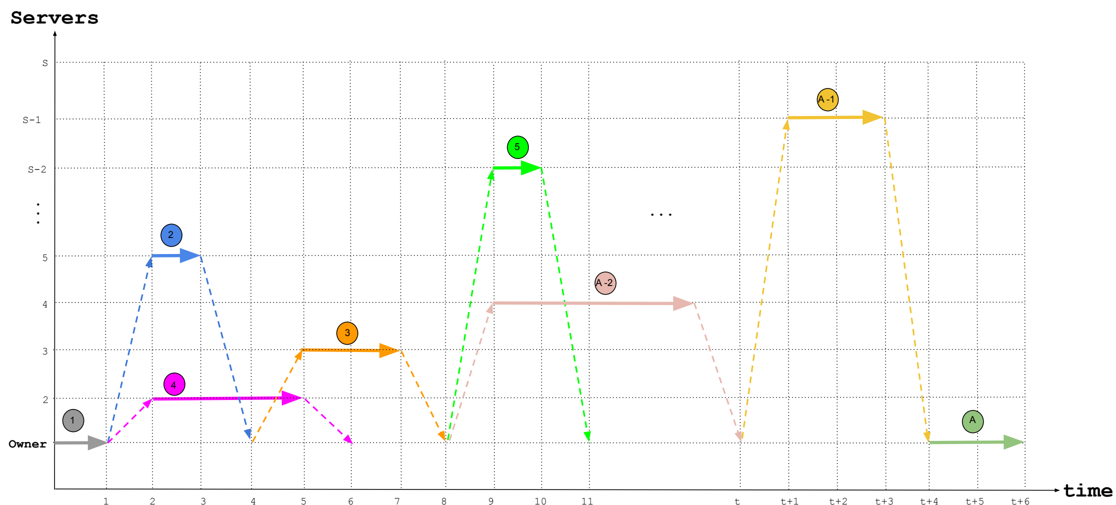

Figure 3.7. Representation of the schedule of $\mathcal{A} = \{1, \ldots, A\}$ tasks executed by a subset of $\mathcal{S} = \{1, \ldots, S\}$ edge servers. The makespan of the schedule is $t + 6$.

precedence dependencies are modeled by the directed acyclic graph $\mathcal{G}$, with no specific release times. The servers may also have different computing power and hardware characteristics. The physical medium and available bandwidth between any pair of servers may be different as well. The joint objective of the problem is to determine an optimal mapping of tasks and servers to maximize the aggregated utility of the busy server and minimize the completion time (i.e makespan) of the resulting schedule. An illustration of the schedule of $\mathcal{A} = \{1 \ldots A\}$ tasks executed by a subset of $\mathcal{S} = \{1 \ldots S\}$ edge servers is presented in Figure 3.7.

As stated by Lawler and Lenstra[22], a collaboration scheme of this type can be modeled as a **R|PREC|F|JP** multiprocessor task scheduling problem. The notation is explained in detail next:

- The unrelated servers condition **R** states that the processing time of any task is a function of the chosen executing server and the task itself.
- The precedence constraint **PREC** assumes that the dependencies of the set of tasks form a directed acyclic graph $\mathcal{G}$. It is also assumed that the tasks may have distinct execution times and also no specific release times. They may be scheduled as soon as their predecessors are completed.

- The function **F** represents the optimal criteria of the model. The dual objective of our task sharing problem aims to maximize the aggregated utility of the busy server and minimize the completion time of the schedule.
- The communication type **JP** states that the cost (i.e. delay) of communicating a task between a pair of servers in the network depends solely on the servers and the task itself.

In short, the problem is set to allocate a finite set of precedence dependent tasks $\mathcal{A} = \{1, \ldots, A\}$ among a set of available edge servers $\mathcal{S} = \{1, \ldots, S\}$ with the objective to maximize the dual function $\mathcal{F} = \mathcal{U} - C_{max}$. The edge servers and the cloud level run as independent nodes on the Hyperledger Fabric blockchain. The blockchain metrics used in the model capture the health of the nodes in terms of their transaction reliability, and the communication overhead introduced by the blockchain platform. The resources of the system and the performance of Fabric are tracked by a monitoring tool sitting at the cloud level. The monitoring tool provides input information to servers requesting the sharing service so that they can make appropriate allocation decisions. The formulation of the task sharing problem at the server level takes the form of a Mixed Integer Linear Program (MILP) presented in equations 3.19x. The summary of symbols and notations can be found in table 3.2.

The problem formulation has a MILP structure which is by definition a non-convex problem. However, the relaxed version of the model may have convex properties. If this is the case, problem solutions may be approximated in tractable time using well-established methods such as branch-and-bound or heuristic techniques. Constraint 3.19c guarantees the one-to-one task-server mapping condition. Constraints 3.19d and 3.19e force task sharing solutions to available servers offering time and energy savings. Constraint 3.19f gives the assurance that a task a cannot start before the processing of all its predecessors b given that $I^{a,b} = 1$. Finally, constraints 3.19g to 3.19i in concert with the overlapping variable θ and the constant M specify that the execution time of any two tasks a and b in a server do not overlap given that $G^{a,b} = 0$.

Table 3.2. Summary of symbols and notations

Notation	**Definition**
Input parameters:	
$\mathcal{J}$	Task-flow computational job
$\mathcal{A}$	Set of computational tasks in $\mathcal{J}$
$a, b \in \mathcal{A}$	Individual tasks in $\mathcal{A}$
d_a	Data size of a
ρ_a	CPU requirements of a
τ_a	Deadline of a
$\mathcal{G}$	Precedence dependency graph of $\mathcal{A}$
$\mathcal{S}$	Set of edge servers in the system
$s \in \mathcal{S}$	Individual server in $\mathcal{S}$
f	CPU-clock frequency of the busy server
c	Hardware constant of the busy server
P	Power consumption of the busy server
t_{HFB}	Latency introduced by HFB*
Indicator variables:	
G	General task precedence dependency matrix
$G^{a,b}$	General precedence of a with respect to b
I	Immediate task precedence dependency matrix
$I^{a,b}$	Immediate precedence of a with respect to b
B	Server availability matrix
$B^{a,s}$	Availability of s to process a
Data variables:	
t^a	Local execution time of a
e^a	Local energy consumption of a
$t^{a,s}$	Outsource roundtrip time (a, s)
$e^{a,s}$	Outsource energy consumption (a, s)
$r^{a,s}$	Performance ratios (a, s)
w	Performance preferences
D	Execution delay matrix
$D^{a,s}$	Execution time (a, s)
C	Communication delay matrix
$C^{a,s}$	Communication time (a, s)
U	Utility matrix
$U^{a,s}$	Utility of the sharing service (a, s)
Support variables:	
$\theta_{a,b}$	Overlapping between a and b
M	Default upper bound of the makespan
Decision variables:	
$X_{a,s}$	Allocation variable (a, s)
S_a	Scheduled start time of a
Objective functions:	
$\mathcal{U}$	Aggregated utility of the task schedule
C_{max}	Makespan of the schedule
F	Objective function of the task sharing model

* HFB: Hyperledger Fabric Blockchain

$$\max_{S_a, X_{as}} \quad \mathcal{U} - C_{max} \tag{3.19a}$$

s.t.

$$X_{a,s} \in \{0, 1\} \tag{3.19b}$$

$$\sum_{s \in \mathcal{S}} X_{a,s} = 1 \tag{3.19c}$$

$$D^{a,s} X_{a,s} \leq t^a \tag{3.19d}$$

$$e^{a,s} X_{a,s} \leq e^a \tag{3.19e}$$

$$S_b \geq S_a + (D^{a,s} + C^{a,s}) X_{a,s}, \qquad |I^{a,b} = 1 \tag{3.19f}$$

$$S_b - \sum_{s \in \mathcal{S}} D^{a,s} X_{a,s} - S_a \leq M(1 - \theta_{a,b}) \quad |G^{a,b} = 0 \tag{3.19g}$$

$$S_b - \sum_{s \in \mathcal{S}} D^{a,s} X_{a,s} - S_a \geq -M\theta_{a,b} \qquad |G^{a,b} = 0 \tag{3.19h}$$

$$X_{a,s} + X_{b,s} + \theta_{a,b} + \theta_{b,a} \leq 3 \qquad |G^{a,b} = 0 \tag{3.19i}$$

$$\theta_{a,b} \in \{0, 1\} \tag{3.19j}$$

3.4.4 Numerical Results

In this section, the task sharing model at the server level is put to the test under different task loads and available resources to assess the behavior of the allocation scheme. The experiments stand on a set of available edge servers with CPU-clock frequencies drawn uniformly from [6.00, 8.00] GHz and average power consumption equal to 100 Watts. The busy server is made intentionally slow compared to the others with CPU-clock frequency set to 1 GHz. The busy server owns a set of precedence-dependent tasks with CPU requirements and sizes are also drawn uniformly from [2.50, 10.00] MCPU-clocks and [50 300] MB, respectively. The precedence dependencies of the tasks are modeled as a random acyclic graph generated from an upper triangular binary random matrix. The weights of the edges are drawn uniformly from the interval [50 150] MB. The performance preferences of

the owner are all set to 0.25. All the servers are connected through a Gigabit Ethernet network with 10 Gbps of available bandwidth, Fabric is set to introduce a latency of 0.02 seconds. The numerical evaluation was powered by Matlab and the CVX optimization toolbox. The results are presented in the following subsections.

Experiment 1: evaluation of the average completion time of a set of tasks with dependencies versus the number of available servers. As shown in Figure 3.8, the makespan of the tasks decreases almost linearly as the number of available servers increases from 1 to 14. On average, the set with $N = 20$ tasks benefits the most from the sharing scheme experiencing a 34% makespan reduction from 0.72 to 0.49 seconds. For the set with $N = 15$ tasks, the reduction is 24% from 0.45 to 0.34 seconds, however, for the set with $N = 10$ tasks, the benefit is almost negligible.

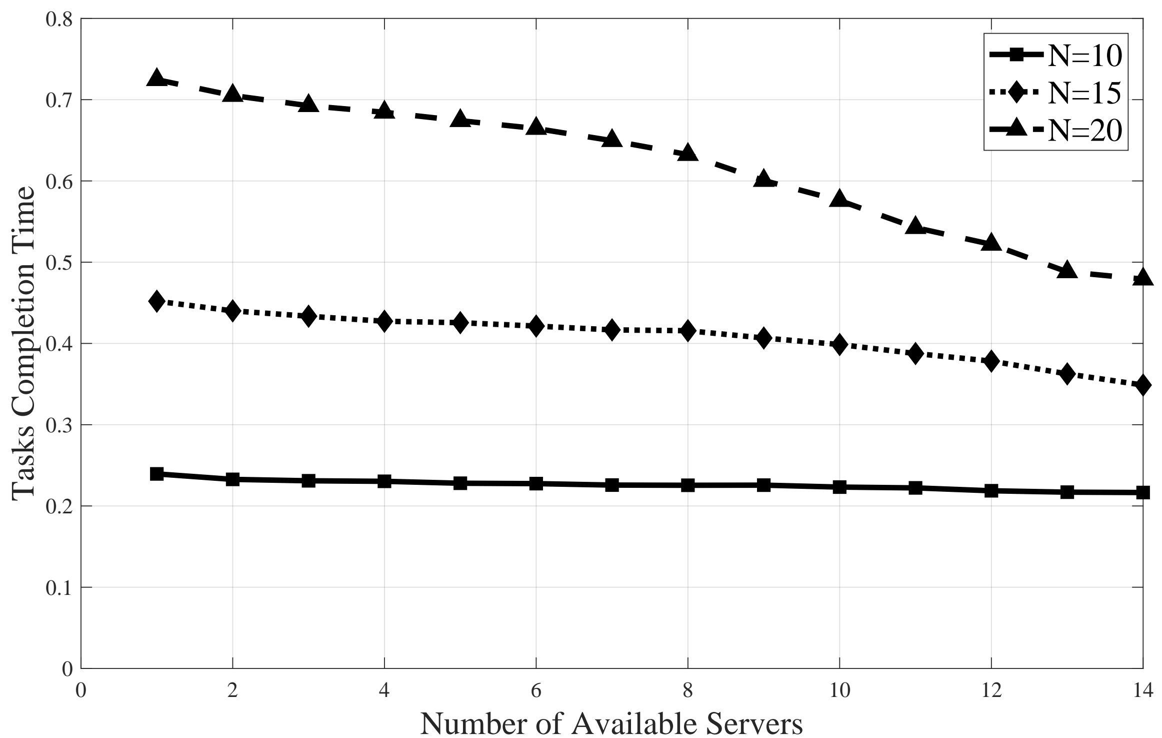

Figure 3.8. Results of experiment 1: average task completion time.

Experiment 2: evaluation of the average number of service providers versus the number of available servers. As shown in Figure 3.9, the network occupation measured by the number of service providers, increases as the number of available servers raises from 1 to 14. For the set with $N = 10$ tasks, a saturation point is reached close to 9 available servers and sits at about 2.5 service providers or 28% occupation ratio. For the set with $N = 15$ tasks, saturation is reached at around 11 available servers and sits approaching 2.6 service providers or 24% occupation ratio. Finally, for the set with $N = 20$ tasks, saturation is reached at around 13 available servers and sits roughly above 3 service providers or 23% occupation ratio.

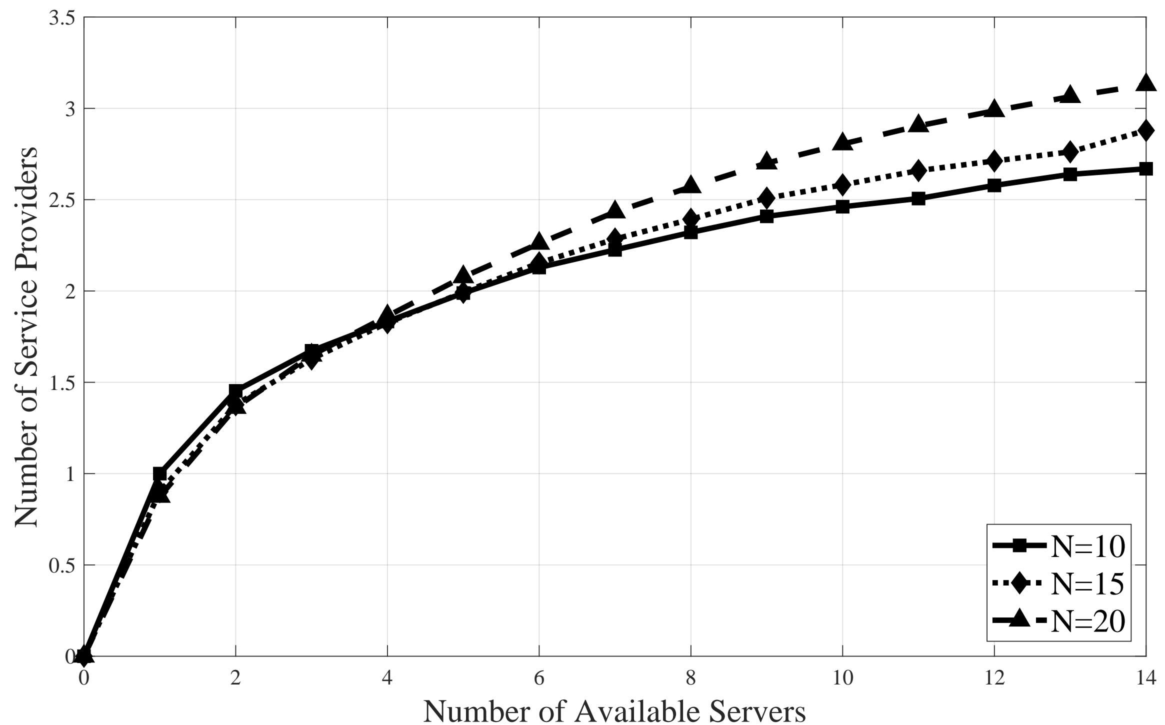

Figure 3.9. Results of experiment 2: average number of service providers.

The preliminary experiments show that the proposed task sharing model might increase the utilization of resources in the network although we believe that there is still much room for improvements in this domain. Although the focus of this thesis has always been the implementation of a demo showing that our blockchain model for task sharing is possible in real life, we believe that the natural progression

of this work must include the evolution of the proposed communication model and task sharing scheme. In fact, there is still much room for more in-depth mathematical analysis of the task allocation problem and solution algorithms. For now, we're offering results that lie more on the experimental side of the idea, however, they are the grounds for future development of more sophisticated collaboration models that may bring better performance results. It is also important to mention that the solution to the task sharing problem at the server level is implemented as an optimization module in our blockchain model. The module is included in the network as a client application running on the nodes of our demo. It uses the Google Operations Research library (Google OR Tools) to solve the task sharing problem discussed in this section. For more details on the implementation of the optimization module, please refer to chapter 4, section 4.4.

4 *EdgeChain*: a Blockchain-based Task Sharing Service Demo

4.1 Preliminary analysis

EdgeChain is a blockchain-based solution designed for a consortium of edge service providers to share computational tasks between organizations within a blockchain network. *EdgeChain* runs on the Hyperledger Fabric platform which is an open source permissioned blockchain with a trust model that consists of membership, consensus, ledger, and chaincode services so that server nodes in the network can engage in reliable, secure, and private task sharing services. Edge Service Providers (ESPs) are typically private tech companies (i.e. Google, Amazon, Microsoft, IBM) with computing infrastructure deployed in strategic locations near the end users they serve. ESPs might offer content delivery, web services, and application services to users located at the bottom layer of a communications network. In our *EdgeChain* demo, five separate organizations, specifically, four Edge Service Providers and one Cloud Service Provider, get into an infrastructure cluster to form a blockchain consortium to pool computer infrastructure and process computational tasks coming from stranded servers in the network. The whole premise of *EdgeChain* relies on the idea that a consortium of ESPs willing to share their idle resources would effectively reduce the latency associated with the execution of queued tasks in the system, and also increase the utilization efficiency of the computer infrastructure installed on the consortium.

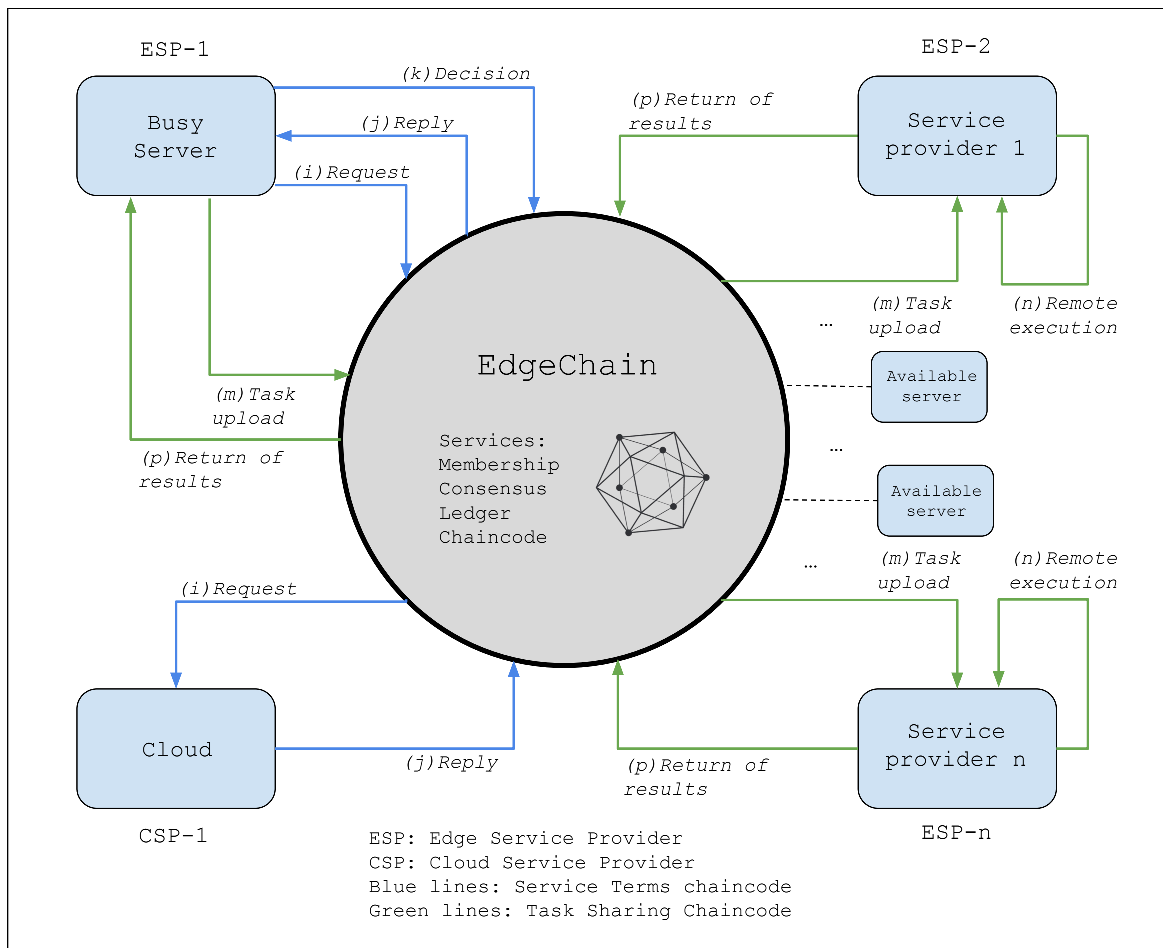

Figure 4.1. Blockchain players, their roles, and available transactions in *EdgeChain* network.

There are four distinct roles in *EdgeChain*: (1) busy server, (2) clould level, (3) available server, and (4) service provider. The interactions between busy servers and the cloud are overseen by the *Service Terms Chaincode* available on the global channel. More specifically, busy servers can request the task sharing service using the *(i) request* transaction. The cloud level, on the other hand, is equipped with a monitoring tool with an overview of the resources in the network and is in charge of replying to service requests with a set of candidate servers to take on the tasks. To do that, the cloud makes use of the *(j) reply* transaction. Finally, when busy servers make task sharing decisions to allocate queued computational tasks to a

subset of available servers, they make use of the *(k) decision* to communicate their decisions to the global channel. After task sharing decisions have been reached, the interactions between busy servers and service providers are overseen by the *Task Sharing Chaincode*. In particular, the servers make use of the *(m) task upload*, *(n) remote execution*, and *(p) return of results* transactions to complete the lifecycle of the task sharing service on separate private channels. Figure 4.1 shows an illustration of the blockchain players, their roles, and the available transactions in the *EdgeChain* network. *EdgeChain* relies on the membership, consensus, ledger, and chaincode services provided by Hyperledger Fabric services to support the implementation of the task sharing solution. The deployment strategy presented in this chapter will reproduce the following sequence: (1) install the pre-requisites and components of the Fabric platform, (2) develop and deploy the chaincode with the logic of the task sharing solution, and (3) develop and install the necessary client applications that serve as the connection between the outside world and the blockchain network.

4.2 *EdgeChain* network setup

The first step towards launching a Fabric network consists in the installation of a set of software prerequisites on the nodes partaking in the network. Hyperledger Fabric is a container-based platform that uses container management tools, programming languages, packet managers, and binaries to facilitate the operation of the blockchain network. Fabric binaries, in particular, are the platform-specific machine language executable files to set up and run the Fabric blockchain. The complete list of Fabric pre-requisites and their recommended versions is shown in Table 4.1. Once the required software is installed on the blockchain nodes, the Fabric network is ready to be launched. For the deployment of *EdgeChain*, we have four PCs to simulate the infrastructure of the organizations taking part in the blockchain consortium. Every computer will host a set of docker containers with the assigned Fabric components according to the Fabric configuration file (i.e. *config.tx*). The containers include certificate authorities, ordering service peers,

organization peers, peer databases, CLIs, chaincode packages, and client application packages. A summary of the specifications of the computers is shown in Table 4.2.

Software	Version	Remarks
Linux Ubuntu	20.04	Unix-based operating system
Curl	7.68.0	Software tool for getting and sending data over the Internet using URL syntax
Docker	20.10.12	Software tool to manage container units for OS-level virtualization.
Docker Compose	1.25.0	Software tool for defining and running multi-container Docker applications.
Docker Swarm	20.10.12	Software tool for orchestration of container-based cluster networks.
Go	1.13.8	Programming language
Node js	10.19.0	JavaScript runtime environment.
Python	2.7.18	Programming language
NPM	6.14.4	Default package manager for the JavaScript runtime environment Node.js
Fabric binaries	2.2	Hyperledger Fabric executable files

Table 4.1. Hyperledger Fabric software prerequisites

	PC-1	PC-2	PC-3	PC-4
Name	ECE-Sturgeons	ECE-Gudgeon	ECE-Agnatha	ECE-Dipnol
Operating System	Linux Ubuntu	Linux Ubuntu	Linux Ubuntu	Linux Ubuntu
System Type	64 bits	64 bits	64 bits	64 bits
Processor	3.40 GHz x 8	3.40 GHz x 8	3.40 GHz x 8	3.40 GHz x 8
Memory	7.6 GiB	7.6 GiB	7.6 GiB	7.6 GiB
Storage	500 GB	500 GB	500 GB	500 GB
Network capabilities	100 Mbps	100 Mbps	100 Mbps	100 Mbps

Table 4.2. Technical specifications of the available PCs

4.2.1 Physical connections

The four *EdgeChain* PCs are physically connected to a hardwired Local Area Network with a data-rate capacity of 1 Gbps. The computers are logically organized in a cluster with the help of Docker Swarm orchestration software. Docker-swarm powers the implementation of container-based networks across multiple physical or virtual hosts. This is a deliberate design choice so that the set of available PCs can be managed from a single administration point. In the case of *EdgeChain*, the cluster is controlled by one node that acts as cluster manager and another three nodes that act as followers or workers. The manager node is the single point of administration of the physical network, therefore, all the Fabric components and their docker containers can be deployed from this location. An illustration of the *EdgeChain* docker swarm cluster can be found in Figure 4.2. The names of the servers, their IPs, and their labels inside the swarm cluster are shown in Table 4.3.

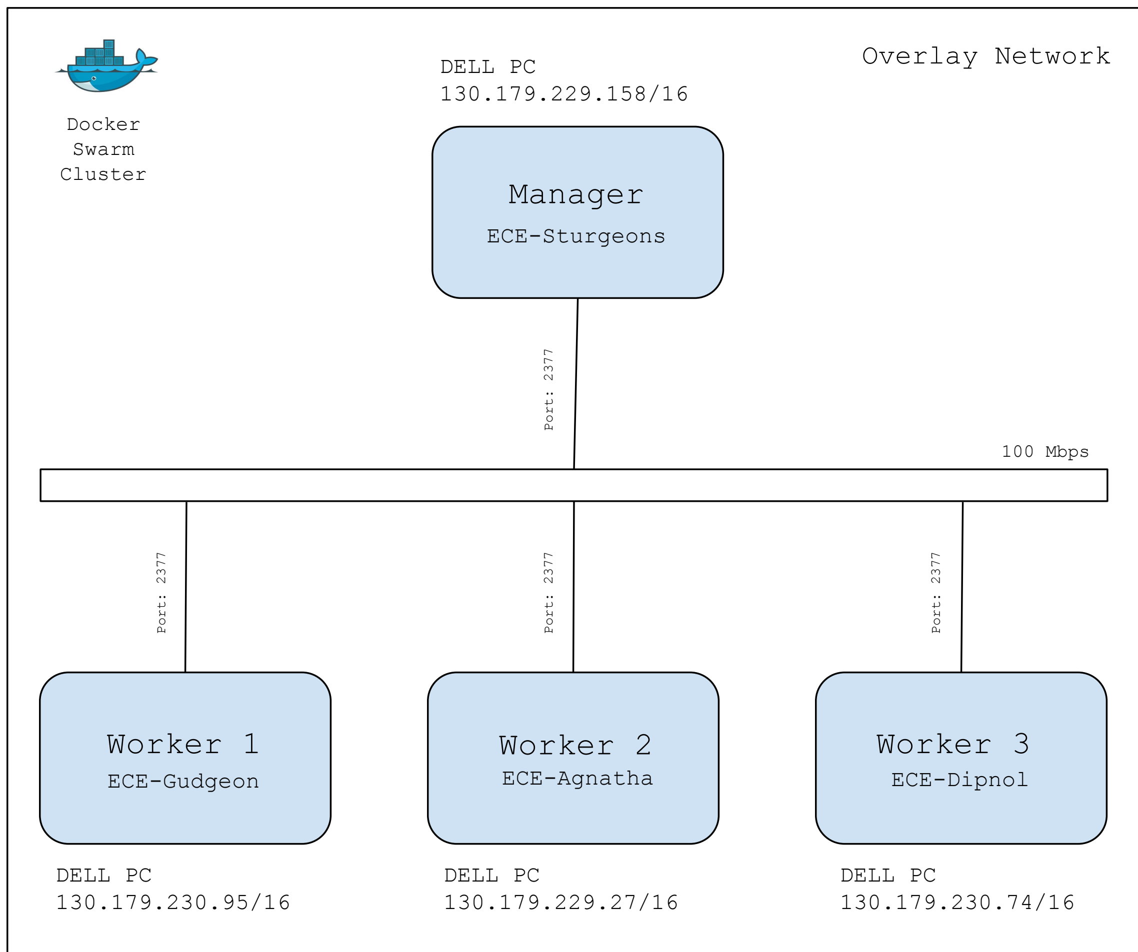

Figure 4.2. Illustration of the *EdgeChain* docker swarm cluster

The implementation of the cluster should produce an output similar to what is shown in the terminal output screen down below:

```
ID                             HOSTNAME         STATUS       MANAGER STATUS
pr93s85uv1qeo779l78ualz3e  *   ECE—Sturgeons    Ready        Leader
u0j1hxzfy8c6nqr46b9rzeb2w      ECE—Gudgeon      Ready
rr7fg7ygd63ozbcscorh21vxi      ECE—Agnatha      Ready
qb23m0dx0fbpolvtn745ndmy6      ECE—Dipnol       Ready
```

4.2.2 Network components

Hyperledger Fabric is a container-based blockchain that uses isolated software units (i.e. software containers) to deploy network components on the physical

Name	**IP**	**Swarm label**
ECE-Sturgeons	130.179.229.158	Manager
ECE-Gudgeon	130.179.230.95	Worker 1
ECE-Agnatha	130.179.229.27	Worker 2
ECE-Dipnol	130.179.230.74	Worker 3

Table 4.3. Names, IPs, and labels of participants inside the *EdgeChain* cluster

infrastructure. *EdgeChain* is a Fabric-based network that recreates a distributed version of a cloud/edge architecture. More specifically, four Edge Service Providers (ESP) and one Cloud Service Provider (CSP) get together to form a blockchain consortium to launch a decentralized task sharing service that delivers computing services to software applications located behind the edge infrastructure. The consortium accommodates five organizations (i.e. ESPs and CSP) across the four computers available for *EdgeChain*. Every organization is composed of one Certificate Authority (CA) in charge of generating cryptographic identities inside the organization, one organization peer that hosts the chaincode with the task sharing logic, one instance of a Couch Database (CouchDB) that keeps a copy of the world state and blockchain, and a Command Line Interface (CLI) to interact with the peers. In addition, *EdgeChain* implements ordering peers for the ordering service, and the membership service to validate identities within the network. An illustration of the arrangement of docker containers inside *EdgeChain* participating organizations (i.e. CSP and ESPs) can be found in Figure 4.3.

Fabric provides a Certificate Authority tool called *Fabric-CA* that reproduces a Public Key Infrastructure (PKI) to create digital identities in the network. *Fabric-CA*

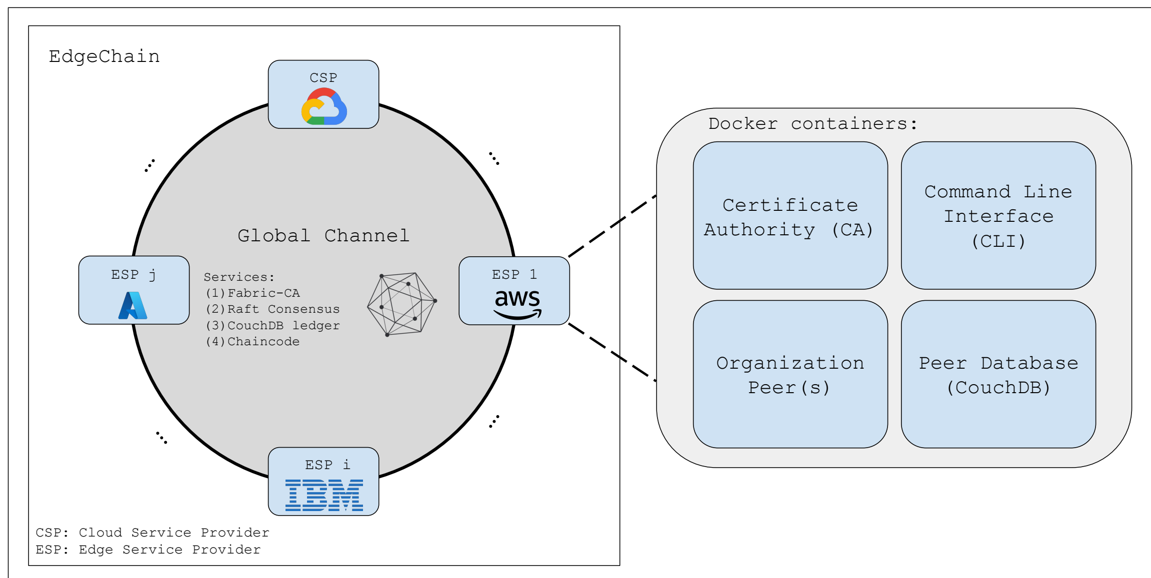

Figure 4.3. Arrangement of docker containers inside *EdgeChain* participating organizations

generates cryptographic identities through a process called *enrollment* that is configurable to fit the amount of crypto material required by the network. *Fabric-CA* might be used during development stages, however, when moved to production, it should be replaced with commercially available CAs that are trusted by the Internet industry[31]. Following the spin-up of containers with the Certificate Authorities (CAs), *Fabric-CA* tool generates cryptographic identities for network participants (i.e. organizations, peers, client users, etc). They use the crypto identities to sign their attempts to interact with the blockchain network. After crypto material generation, the rest of Fabric components are brought up to complete the formation of the network. The spin-up of network containers is initiated by the manager node in the docker swarm cluster using <*docker stack deploy*> commands. At this point, ordering peers, organization peers, CouchDB instances, and CLIs should be deployed across network nodes. The container profiles are available in *YAML* configuration files that are used by docker commands during the spin-up process. The typical building routine for Fabric networks is presented in Figure 4.5.

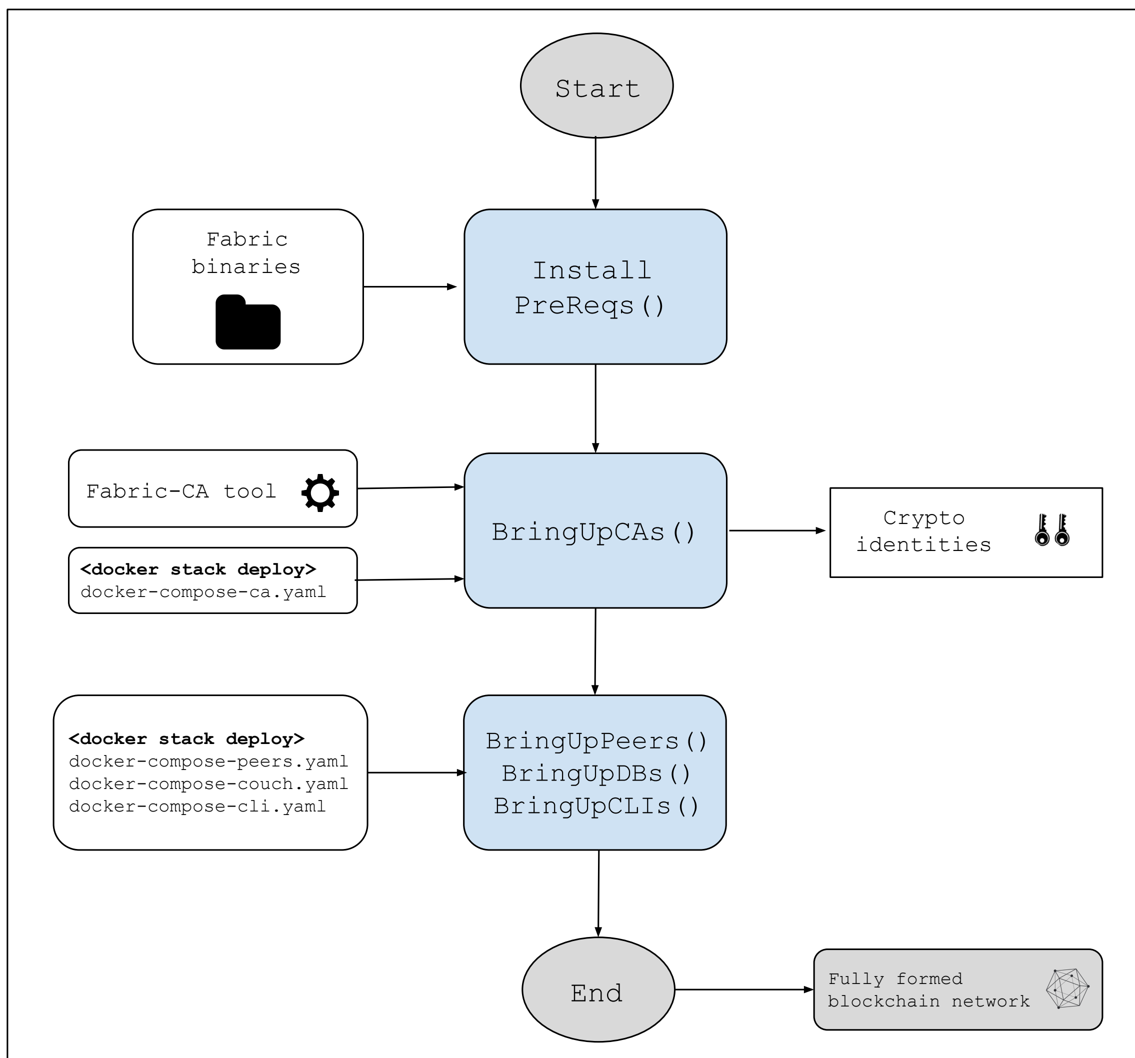

Figure 4.4. Routine for spinning-up Fabric containers

Following the creation of Fabric components, EdgeChain consortium is ready to create application channels. To do that, Fabric provides the *ConfigTxGen* tool that allows the creation of channel configuration blocks and related channel artifacts (i.e. channel update transaction and anchor peers information). For channel creation, the *ConfigTxGen* tool receives the configuration file of the network (i.e. *configtx.yaml*) as an input. The network contains one global channel with the plenary of edge server nodes, and $\binom{n}{2}$ private channels (n is the number of server nodes) to privately connect every combination of two servers in the network. The channel

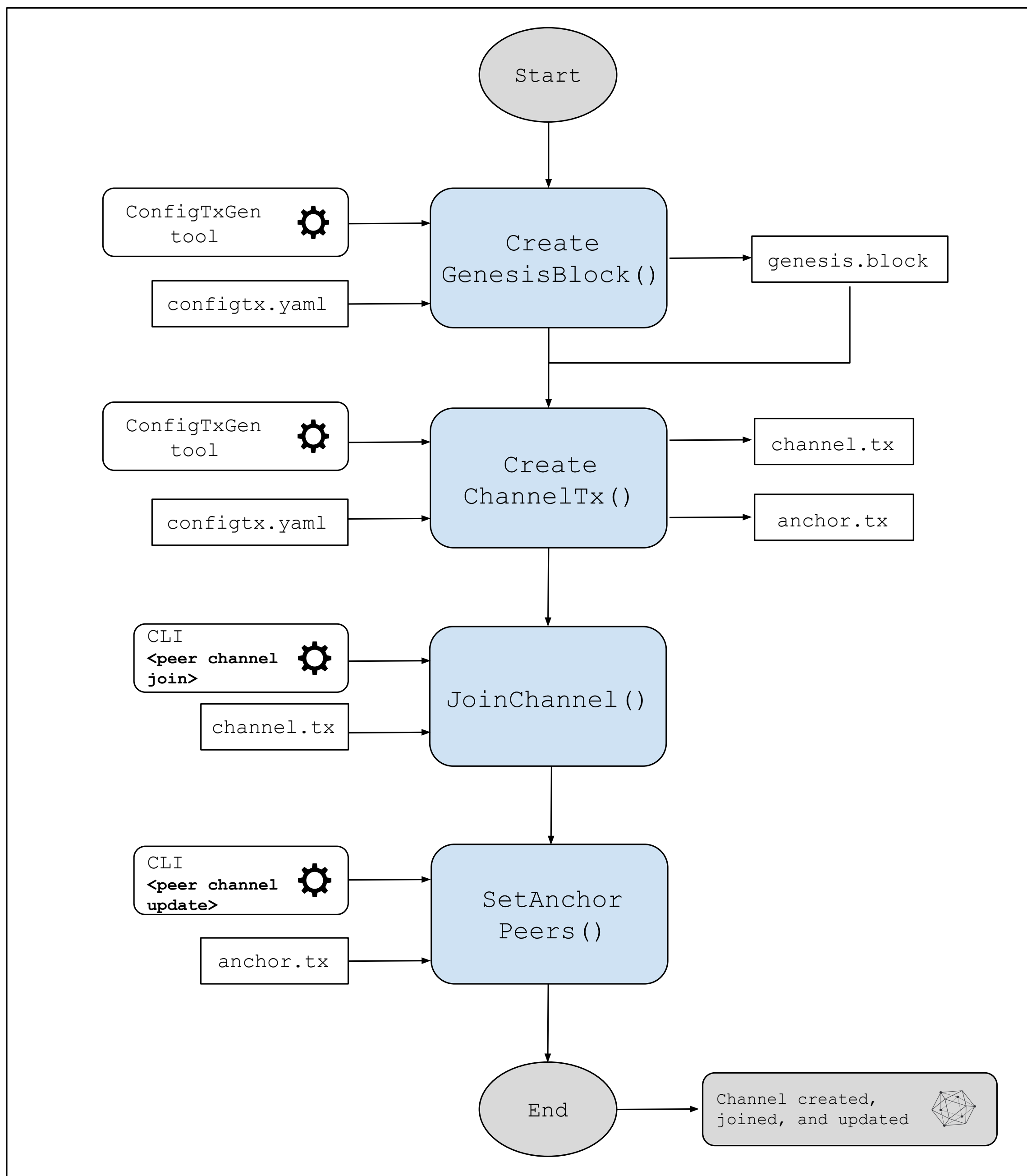

Figure 4.5. Routine for channel creation

creation routine is presented in Figure 4.5. A summary of the components of the *EdgeChain* network can be found in Table 4.4.

Component	Container Name	Physical Node	Network Alias
System:			
Orderer CA	ca.orderer	Manager	ca.orderer.edgechain.com:10054
Orderer Peer	orderer	Manager	orderer.edgechain.com:7050
ESP1:			
CA	ca.org1	Worker1	ca.org1.edgechain.com:7054
Peer	peer0.org1	Worker1	peer0.org1.edgechain.com:7051
Peer DB	couch.peer0.org1	Worker1	couch.peer0.org1.edgechain.com:5984
CLI	cli.org1	Worker1	cli.org1.edgechain.com
ESP2:			
CA	ca.org2	Worker2	ca.org2.edgechain.com:8054
Peer	peer0.org2	Worker2	peer0.org2.edgechain.com:9051
Peer DB	couch.peer0.org2	Worker2	couch.peer0.org2.edgechain.com:7984
CLI	cli.org2	Worker2	cli.org2.edgechain.com
ESP3:			
CA	ca.org3	Worker3	ca.org3.edgechain.com:9054
Peer	peer0.org3	Worker3	peer0.org3.edgechain.com:11051
Peer DB	couch.peer0.org3	Worker3	couch.peer0.org3.edgechain.com:9984
CLI	cli.org3	Worker3	cli.org3.edgechain.com
ESP4:			
CA	ca.org4	Manager	ca.org4.edgechain.com:11054
Peer	peer0.org4	Manager	peer0.org4.edgechain.com:13051
Peer DB	couch.peer0.org4	Manager	couch.peer0.org4.edgechain.com:11984
CLI	cli.org4	Manager	cli.org4.edgechain.com
CSP:			
CA	ca.org5	Manager	ca.org5.edgechain.com:12054
Peer	peer0.org5	Manager	peer0.org5.edgechain.com:15051
Peer DB	couch.peer0.org5	Manager	couch.peer0.org5.edgechain.com:13984
CLI	cli.org5	Manager	cli.org5.edgechain.com

Table 4.4. Hyperledger Fabric components for the *EdgeChain* network

The manager node hosts the Ordering Service, Org4 (ESP4), and Org5 (CSP).On the other hand, worker 1 hosts Org1 (ESP1), worker 2 hosts Org2 (ESP2), and worker

3 hosts Org3 (ESP3). A query command on the docker containers installed on the network nodes should produce results similar to the terminal output snippets presented next.

Manager node:

```
CONTAINER ID    IMAGE                STATUS    PORTS    NAMES
d68ff1327324    fabric—ca:latest     Up        7054     ca.orderer
c5cc200803cc    fabric—orderer:2.3   Up        7050     orderer
70c18fe9fae1    fabric—ca:latest     Up        7054     ca.org4
25a027d1d26f    fabric—peer:2.3      Up        7051     peer0.org4
88ae0c5f8ce9    couchdb:3.1.1.       Up        5984     couch.peer0.org4
4fa4d58e8d4e    fabric—tools:2.3     Up                 cli.org4
7aa9b3ae1fa2    fabric—ca:latest     Up        7054     ca.org5
8336e89af664    fabric—peer:2.3      Up        7051     peer0.org5
ef8d47792da7    couchdb:3.1.1        Up        5984     couch.peer0.org5
7451af8ce170    fabric—tools:2.3     Up                 cli.org5
```

Worker 1 node:

```
CONTAINER ID    IMAGE                STATUS    PORTS    NAMES
d53330921492    fabric—ca/latest     Up        7054     ca.org1
09ccc93a13e7    fabric—peer:2.3      Up        7051     peer0.org1
671bfd7a3b21    couchdb:3.1.1        Up        5984     couch.peer0.org1
d09c56fdb545    fabric—tools/2.3     Up                 cli.org1
```

Worker 2 node:

```
CONTAINER ID    IMAGE                STATUS    PORTS    NAMES
d41674361613    fabric—ca/latest     Up        7054     ca.org2
b66aa8494d6e    fabric—peer:2.3      Up        7051     peer0.org2
3f8e08aac31e    couchdb:3.1.1        Up        5984     couch.peer0.org2
e40773486676    fabric—tools/2.3     Up                 cli.org2
```

Worker 3 node:

```
CONTAINER ID    IMAGE                STATUS    PORTS    NAMES
12b0aaceda7d    fabric—ca/latest     Up        7054     ca.org3
f630368adba2    fabric—peer:2.3      Up        7051     peer0.org3
cee6636664d7    couchdb:3.1.1        Up        5984     couch.peer0.org3
b13edce497d6    fabric—tools/2.3     Up                 cli.org3
```

4.3 *EdgeChain* smart contracts

EdgeChain participants (i.e. CSPs and ESPs) use chaincode available in the network to circulate computational tasks queued in busy servers. The blockchain model presented in section 3.3 states that the dynamics of the task sharing process may be described by the following sequence: (1) service request, (2) discovery of candidate nodes, (3) selection of service providers, and (4) task sharing contract. In this section, we present the details of the smart contracts that make up the core logic of the task sharing solution presented in this thesis.

4.3.1 Service Terms Chaincode

Service Terms Chaincode contains the logic of the first three stages of the task sharing model (i.e. service request, the discovery of candidate nodes, and selection of service providers) where the terms of task sharing services are negotiated. We make use of Java programming language and Fabric Contract Shim API for coding the logic inside *Service Terms Chaincode*. Fabric Contract API provides a series of classes, methods, and interfaces to code contracts, transactions, and perform ledger operations. The smart contract is shared on the global channel and is accessible to the plenary of server nodes and the cloud level. In fact, busy servers and the cloud level might invoke transactions to negotiate computing resources in the network. On the global channel, the objects of value are named *Service Terms* which may be described by the set of negotiation properties presented in Table 4.5. The Java class declaration of *Service Terms* objects can be found in the code snippet down below.

```
/* Details: ServiceTerms class */
@DataType()
public final class ServiceTerms {
    @Property()
    private final String serviceID;
    @Property()
    private final String serviceOwner;
    @Property()
    private final String requestedResources;
    @Property()
```

```
    private final String requestTime;
    @Property()
    private final String availableServers;
    @Property()
    private final String replyTime;
    @Property()
    private final String decisionStartTime;
    @Property()
    private final String serviceProviders;
    @Property()
    private final String decisionFinishTime;
}
```

Property	Description
serviceID	Alpha-numeric identifier of the task sharing request. Type: string.
serviceOwner	Name of requesting server. Type: string.
requestedResources	Tasks profile information. Type: JSON object.
requestTime	Request timestamp. Type: string.
availableServers	Available servers in the network. Type: JSON object.
replyTime	Request timestamp. Type: string.
decisionStartTime	Start decision timestamp. Type: string.
serviceProviders	Selected servers for remote execution. Type: JSON object
decisionFinishTime	Finish decision timestamp. Type: string.

Table 4.5. *Service Terms* negotiation properties

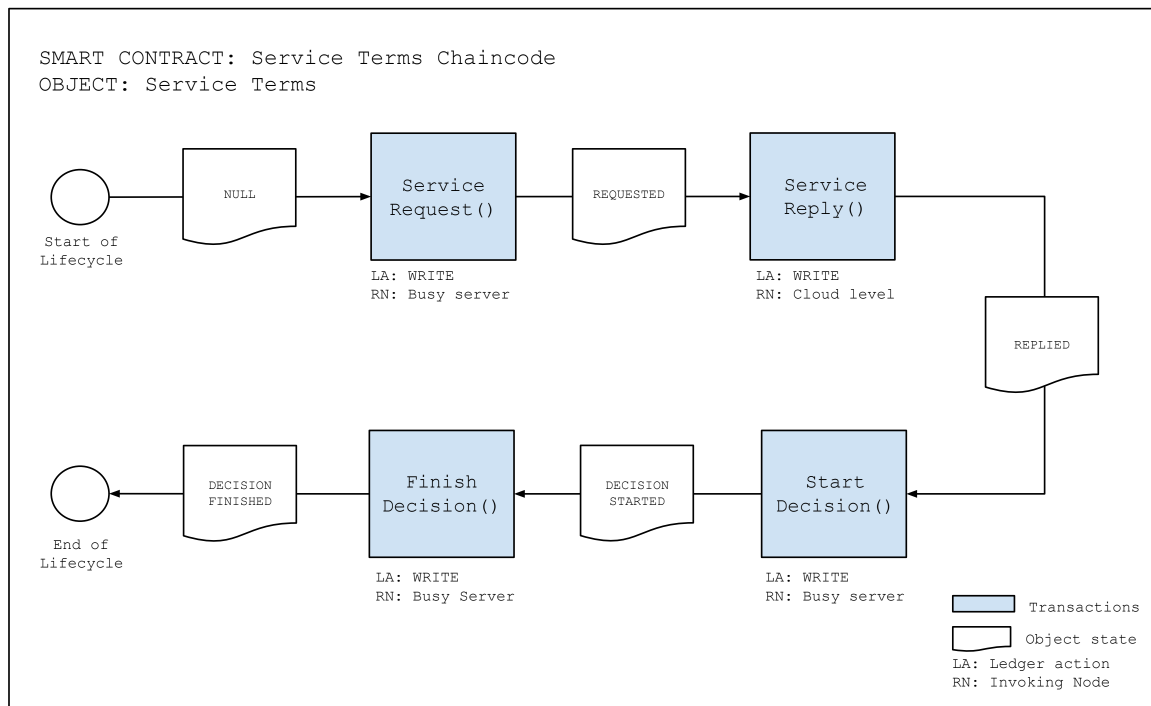

Figure 4.6. Lifecycle transitions of *Service Terms* objects

Service Terms Chaincode governs the process that handles *Service Terms* objects from initial creation to successful completion of their lifecycle. The transitions can be described by transactions that change the state of *Service Terms* objects circulating on the global channel. An illustration of the lifecycle of *Service Terms* objects is presented in Figure 4.6. From the lifecycle diagram, *Service Terms Chaincode* logic can be explained in terms of four transactions described as follows:

Service Request. Busy servers with queued computational tasks can initiate task sharing services with *serviceRequest()* transactions. When this function is invoked, busy servers hand over the strings *serviceID*, *serviceOwner*, *requestTime* and the JSON object *requestedResources* to put together transaction proposals for the network. If proposals are approved, new *Service Terms* objects are created and written to the channel ledger. The initial state of objects is set to *REQUESTED*. The Java implementation of *serviceRequest* transactions is presented in the code snippet down below.

```java
/**
* Details: Busy servers use tx to request task sharing services
* Prototype: serviceRequest(@param1,@param2,@param3,@param4)
* @param1: ctx transaction context
* @param2: serviceID
* @param3: serviceOwner
* @param4: requestedResources
* return tye: ServiceTerms
*/
@Transaction()
public ServiceTerms serviceRequest(final Context ctx, final String
    ↪ serviceID, final String serviceOwner, final String
    ↪ requestedResources, final String requestTime) {
    ChaincodeStub stub = ctx.getStub();
    String serviceState = stub.getStringState(serviceID);
    ServiceTerms service = new ServiceTerms(serviceID, serviceOwner,
        ↪ requestedResources, requestTime, null, null, null, null,
        ↪ null);
    serviceState = genson.serialize(service);
    stub.putStringState(serviceID, serviceState);
    return ServiceTerms;
}
```

Service Reply. The cloud level can reply to service requests with *serviceReply()* transactions. When this function is called, the cloud level hands over the strings *serviceID*, *serviceOwner* and the JSON object *availableResources* to create transaction proposals for the network. If transactions are approved, the *Service Terms* objects associated with provided *serviceID* parameters get updated on the channel ledger with the list of available resources in the network. Also, the state of the objects is moved from *REQUESTED* to *REPLIED*. The Java implementation of this transaction is presented in the code snippet shown next.

```java
/**
* Details: Cloud level uses tx to reply to task sharing requests
* Prototype: serviceReply(@param1,@param2,@param3,@param4)
* @param1: ctx transaction context
* @param2: serviceID
* @param3: availableServers
* @param4: replyTime
* return type: ServiceTerms
```

```
*/
@Transaction()
public ServiceTerms serviceReply(final Context ctx, final String
   ↪ serviceID, final String availableServers, final String
   ↪ replyTime) {
   ChaincodeStub stub = ctx.getStub();
   String serviceState = stub.getStringState(serviceID);
   ServiceTerms service = genson.deserialize(serviceState,
      ↪ ServiceTerms.class);
   ServiceTerms newService = new
      ↪ ServiceTerms(service.getServiceID(),service.getServiceOwner(),
      ↪ service.getRequestedResources(),service.getRequestTime(),
      ↪ availableServers, replyTime, null, null, null);
   String newServiceState = genson.serialize(newService);
   stub.putStringState(serviceID, newServiceState);
   return newService;
}
```

Start Decision. When busy servers get notified that the cloud level has replied to their service requests, they must inform the network that they are ready to make task allocation decisions. To do that, they use *startDecision()* transactions. When this function is called, busy servers hand over the strings *serviceID* and *decision-StartTime* to construct transaction proposals for the network. After transactions are approved, *Service Terms* objects get updated on the ledger with timestamps that inform the network that busy servers have started to solve their task allocation problems. At this stage, the state of *Service Terms* objects is moved from *REPLIED* to *DECISION STARTED*. The Java implementation of this transaction is presented in the code snippet shown next.

```
/**
* Details: Busy servers use tx to start task allocation decisions
* startDecision(@param1,@param2,@param3)
* @param1 ctx transaction context
* @param2 serviceID
* @param3 decisionStartTime
* return ServiceTerms
*/
@Transaction()
public ServiceTerms startDecision(final Context ctx, final String
   ↪ serviceID, final String decisionStartTime) {
```

```
    ChaincodeStub stub = ctx.getStub();
    String serviceState = stub.getStringState(serviceID);
    ServiceTerms service = genson.deserialize(serviceState,
        ↪ ServiceTerms.class);
    ServiceTerms newService = new
        ↪ ServiceTerms(service.getServiceID(),
        ↪ service.getServiceOwner(),
        ↪ service.getRequestedResources(), service.getRequestTime(),
        ↪ service.getAvailableServers(),
    service.getReplyTime(), decisionStartTime, null, null);
    String newServiceState = genson.serialize(newService);
    stub.putStringState(serviceID, newServiceState);
    return newService;
}
```

Finish Decision. When busy servers have made their task allocation decisions, they use *finishDecision()* transactions to notify the blockchain network about this event. When this function is invoked, busy servers hand over the strings *serviceID*, *finishDecisionTime* and the JSON object *serviceproviders* to create transaction proposals and send them to the network. When transactions get approved, *Service Terms* objects get updated on the ledger with the set of servers that were selected to provide task sharing services. In other words, *finishDecision()* transactions culminate the negotiation process between busy servers and the cloud level. At this stage, *Service Terms* object states are moved from *DECISION STARTED* to *DECISION FINISHED* which also means the end to their lifecycle. The Java implementation of this transaction is presented in the code snippet shown next.

```
/**
* Details: Busy servers use tx to finish task allocation decisions
* finishDecision(@param1,@param2,@param3,@param4)
* @param1 ctx transaction context
* @param2 serviceID
* @param3 serviceProviders
* @param4 decisionFinishTime
* return ServiceTerms
*/
@Transaction()
```

```
public ServiceTerms finishDecision(final Context ctx, final String
    ↪ serviceID, final String serviceProviders, final String
    ↪ decisionFinishTime) {
    ChaincodeStub stub = ctx.getStub();
    String serviceState = stub.getStringState(serviceID);
    ServiceTerms service = genson.deserialize(serviceState,
        ↪ ServiceTerms.class);
    ServiceTerms newService = new
        ↪ ServiceTerms(service.getServiceID(),
        ↪ service.getServiceOwner(),
        ↪ service.getRequestedResources(), service.getRequestTime(),
        ↪ service.getAvailableServers(), service.getReplyTime(),
        ↪ service.getDecisionStartTime(),serviceProviders,
        ↪ decisionFinishTime);
    String newServiceState = genson.serialize(newService);
    stub.putStringState(serviceID, newServiceState);
    return newService;
}
```

4.3.2 Task Sharing Chaincode

Task Sharing Chaincode oversees the logic of the final stage of the task sharing model (i.e. the task sharing contract) where *Computational Tasks* objects are privately shared between busy servers and server nodes acting as service providers. As in *Service Terms Chaincode*, we use Java as the contract programming language and Fabric Contract API for the development of the logic inside *Task Sharing Chaincode*. This smart contract is shared on private channels and is accessible in pairs of busy servers and service providers. On private channels, *Computational Tasks* are the objects of value and the properties that describe them are shown in Table 4.6. The Java class declaration of *Computational Task* objects can be found in the code snippet down below.

```
/* Details: ComputationalTask class */
@DataType()
public final class ComputationalTask {
    @Property()
    private final String taskID;
    @Property()
    private final String uploadTime;
```

```
    @Property()
    private final String taskCode;
    @Property()
    private final String execStartTime;
    @Property()
    private final String execFinishTime;
    @Property()
    private final String downloadTime;
    @Property()
    private final String taskResults;
}
```

Property	**Description**
taskID	Alpha-numeric identifier of the task object. Type: string.
uploadTime	Task upload timestamp. Type: string.
taskCode	Task script code. JSON object.
execStartTime	Start execution timestamp. Type: string.
execFinishTime	Finish execution timestamp. Type: string.
downloadTime	Results download timestamp. Type: string.
taskResults	Task results data. Type: JSON object.

Table 4.6. *Computational Task* properties

Task Sharing Chaincode controls the process that moves *Computational Task* objects from creation to successful completion of their lifecycle. Task transitions can be described by transactions that change the state of the the tasks circulating on private channels. An illustration of the lifecycle of *Computational Task* objects

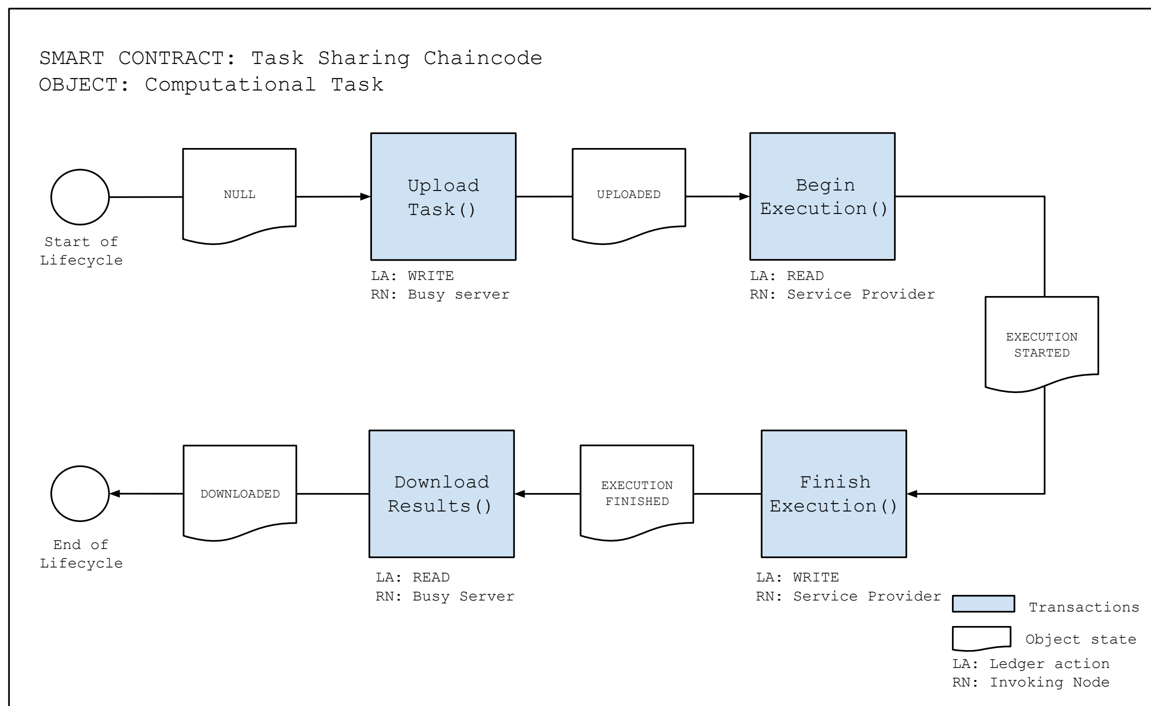

Figure 4.7. Lifecycle transitions of *Computational Task* objects

is presented in Figure 4.7. From the lifecycle diagram, the logic in *Task Sharing Chaincode* can be fully explained with the four transactions described next:

Upload Task. Busy servers use *uploadTask()* transactions to start the sharing process with service providers over private channels. When this function is invoked, busy servers hand over the strings *taskID*, *uploadTime* and the JSON object *taskCode* to create a transaction proposal to the network. If proposals are approved, new *Computational Task* objects are created and written to the channel ledger. The initial state of task objects is set to *UPLOADED*. The Java implementation of this transaction is presented in the code snippet shown next.

```
/**
* Details: Busy servers use tx to upload tasks
* Prototype: uploadTask(@param1,@param2,@param3,@param4)
* @param1: ctx transaction context
* @param2: taskID
* @param3: uploadTime
```

```
* @param4: taskCode
* return type: ComputationalTask
*/
@Transaction()
public ComputationalTask uploadTask(final Context ctx, final String
    ↪ taskID, final String uploadTime, final String taskCode) {
    ChaincodeStub stub = ctx.getStub();
    String taskState = stub.getStringState(taskID);
    Task task = new Task(taskID, uploadTime, taskCode, null, null,
        ↪ null, null);
    taskState = genson.serialize(task);
    stub.putStringState(taskID, taskState);
    return Task;
}
```

Begin Execution. Service providers use *beginExecution* transactions to read *Computational Task* objects that were previously uploaded by busy servers to private channel ledgers. After tasks are read, they become available for remote execution. When this function is called, service providers hand over the strings *taskID* and *execStartTime* to create transaction proposals for the network. If transactions are approved, the *Computational Task* objects associated with the provided *taskID* strings get updated with timestamps to notify the blockchain network that remote execution have started. Also, the state of the objects is moved from *UPLOADED* to *EXECUTION STARTED*. The Java implementation of this transaction is presented in the code snippet down below.

```
/**
* Details: Service providers use tx to begin remote execution
* Prototype: beginExecution(@param1,@param2,@param3)
* @param1: ctx transaction context
* @param2: taskID
* @param3: execStartTime
* return ComputationalTask
*/
@Transaction()
public ComputationalTask beginExecution(final Context ctx, final
    ↪ String taskID, final String execStartTime) {
    ChaincodeStub stub = ctx.getStub();
    String taskState = stub.getStringState(taskID);
    Task task = genson.deserialize(taskState, Task.class);
```

```
    Task newTask = new Task(task.getTaskID(), task.getUploadTime(),
        ↪ task.getTaskCode(), task.getExecStartTime(), null, null,
        ↪ null);
    String newTaskState = genson.serialize(newTask);
    stub.putStringState(taskID, newTaskState);
    return newTask;
}
```

Finish Execution. Service providers use *finishExecution* transactions to notify the blockchain that task execution have finished at their premises and also upload task results to the channel ledger. When this function is called, service providers hand over the strings *taskID*, *execFinishTime* and the JSON object *taskResults* to create transaction proposals for the network. If transactions are approved, *Computational Task* objects get updated on the ledger with timestamps that inform the network that remote execution have finished. At this point, the state of *Computational Task* objects is changed from *EXECUTION STARTED* to *EXECUTION FINISHED*. The Java implementation of this transaction is presented in the code snippet shown next.

```
/**
* Details: Service providers use tx to finish remote execution
* Prototype: finishExecution(@param1,@param2,@param3.@param4)
* @param1: ctx
* @param2: taskID
* @param3: execFinishTime
* @param4: taskResults
* return: ComputationalTask
*/
@Transaction()
public ComputationalTask finishExecution(final Context ctx, final
    ↪ String taskID, final String execFinishTime, final String
    ↪ taskResults) {
    ChaincodeStub stub = ctx.getStub();
    String taskState = stub.getStringState(taskID);
    Task task = genson.deserialize(taskState, Task.class);
    Task newTask = new Task(task.getTaskID(), task.getUploadTime(),
        ↪ task.getTaskCode(), task.getExecStartTime(),
        ↪ execFinishTime, taskResults, null);
    String newTaskState = genson.serialize(newTask);
    stub.putStringState(taskID, newTaskState);
```

```
    return newTask;
}
```

Download Task Results. When busy servers get notified that remote execution of tasks have finished, they use *downloadResults()* transactions to download task results from the channel ledger. When this function is invoked, busy servers hand over strings *taskID* and *downloadTime* to create transaction proposals for the network. If proposals are approved, *Computational Task* objects get updated on the ledger with timestamps that inform the network that task results have been downloaded. Also, *downloadResults()* transactions end the lifecycle *Computational Task* objects and move their states from *EXECUTION FINISHED* to *DOWNLOADED*. The Java implementation of this transaction is presented in the code snippet shown next.

```
/**
* Details: Busy servers use tx to download task results
* Prototype: downloadResults(@param1,@param2,@param3)
* @param1: ctx transaction context
* @param2: taskID
* @param3: downloadTime
* return ComputationalTask
*/
@Transaction()
public ComputationalTask downloadResults(final Context ctx, final
    ↪ String taskID, final String downloadTime) {
    ChaincodeStub stub = ctx.getStub();
    String taskState = stub.getStringState(taskID);
    Task task = genson.deserialize(taskState, Task.class);
    Task newTask = new Task(task.getTaskID(), task.getUploadTime(),
        ↪ task.getTaskCode(), task.getExecStartTime(),
        ↪ task.getExecFinishTime(), task.getTaskResults(),
        ↪ downloadTime);
    String newTaskState = genson.serialize(newTask);
    stub.putStringState(taskID, newTaskState);
    return newTask;
}
```

4.4 *EdgeChain* Client Applications

Client applications can interact with *EdgeChain* to submit transactions and execute operations in the platform ledgers. More precisely, Edge Service Providers (ESPs) and Cloud Service Providers in *EdgeChain* can invoke transactions available in *Service Terms Chaincode* and *Task Sharing Chaincode* to circulate computational tasks in the edge network. Client applications are a key part of the *EdgeChain* solution due to their ability to hook up the outside world to the blockchain infrastructure. Depending on their physical location in the network, client applications may monitor the performance of edge servers, manage queued tasks, and solve task allocation problems. This section presents the details of the three client applications developed for *EdgeChain*: (1) *BusyServerApp*, (2) *CloudLevelApp*, and (3) *PrivateSharingApp*.

4.4.1 Busy Server Client Application

BusyServerApp clients are Java applications installed on multiple ESP servers to monitor incoming application requests (i.e. computational jobs) from end-users and request computer resources on their behalf of them. *BusyServerApp* clients may call the *Service Terms Chaincode* to start the negotiation of task sharing services with the blockchain network when the server nodes have queued tasks that need processing services. *BusyServerApp* clients will connect to the blockchain platform using the Fabric Java SDK module that supplies the functionality to reach other nodes in the global channel. *BusyServerApp* clients use gateway peers (i.e. peer0.org$N$, for N={1,2,3,4}) inside their parent organizations to submit transaction proposals to the network. The configuration of gateway peers may be found in the connection profile generated by parent organizations before joining the network consortium. Gateway peers make use of their local copy of the *Service Terms Chaincode* and the Fabric Contract API to assemble transaction proposals that will be submitted to the network. An illustration of the way *BusyServerApp* clients communicate with gateway peers inside their parent organizations is presented in Figure 4.8.

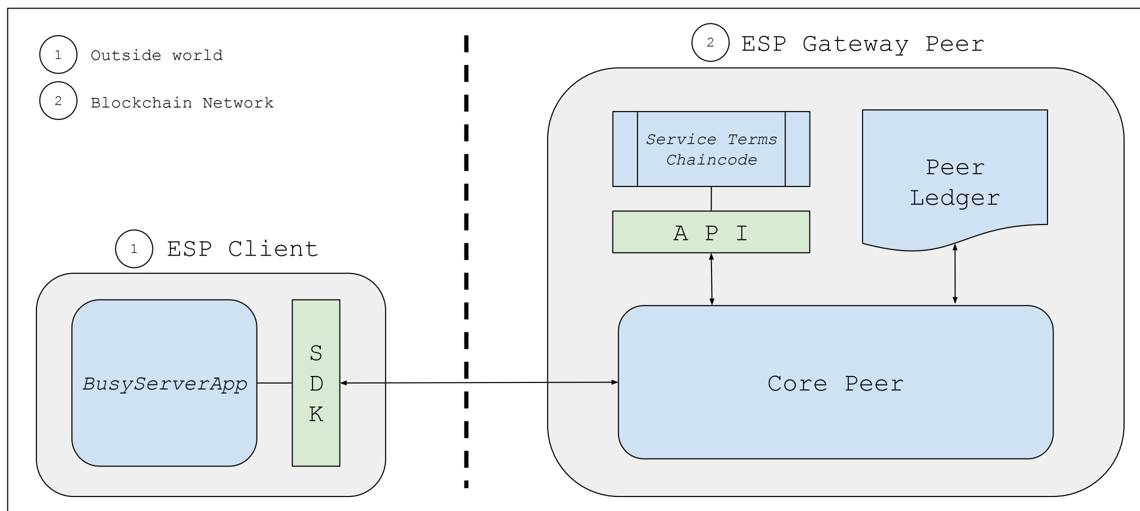

Figure 4.8. Communication between *BusyServerApp* clients and ESP gateway peers

BusyServerApp clients can submit transactions on behalf of stranded servers looking to get resources from the edge network. To do that, they start by retrieving *x.509* digital certificates from their wallets corresponding to pre-authorized users with clearance to access the network. Then, they load the connection profile with designated gateway peers for their parent ESP organizations. Both, identity wallets and connection profiles, are available on the local file systems of the ESP nodes the *BusyServerApp* clients run on. The gateway peers provide an entry point for clients to access the global channel network and the functionality inside *ServiceTermsChaincode*. At that stage, clients may call the functionality inside the chaincode to prepare transaction requests for the network and set off the Fabric consensus mechanism. *BusyServerApp* clients are, in fact, the middle man between busy servers and the blockchain network for the negotiation process of task sharing services. In particular, they can request computing resources, solve task allocation problems, and notify the blockchain about allocation decisions. A simplified diagram of how *BusyServerApp* clients interact with *Service Terms Chaincode* can be found in Figure 4.9. A summary of the labor inside *BusyServerApp* clients is presented next:

(1) Retrieve *x.509* client identities from their local ESP wallets

(2) Connect to designated ESP gateway peers using the information in the connection profiles of their parent organizations
(3) Access the global channel using Java SDK functionality
(4) Connect to *Service Terms Chaincode* and submit transaction proposals to the network
(5) Stay alert of transaction notifications and process responses accordingly

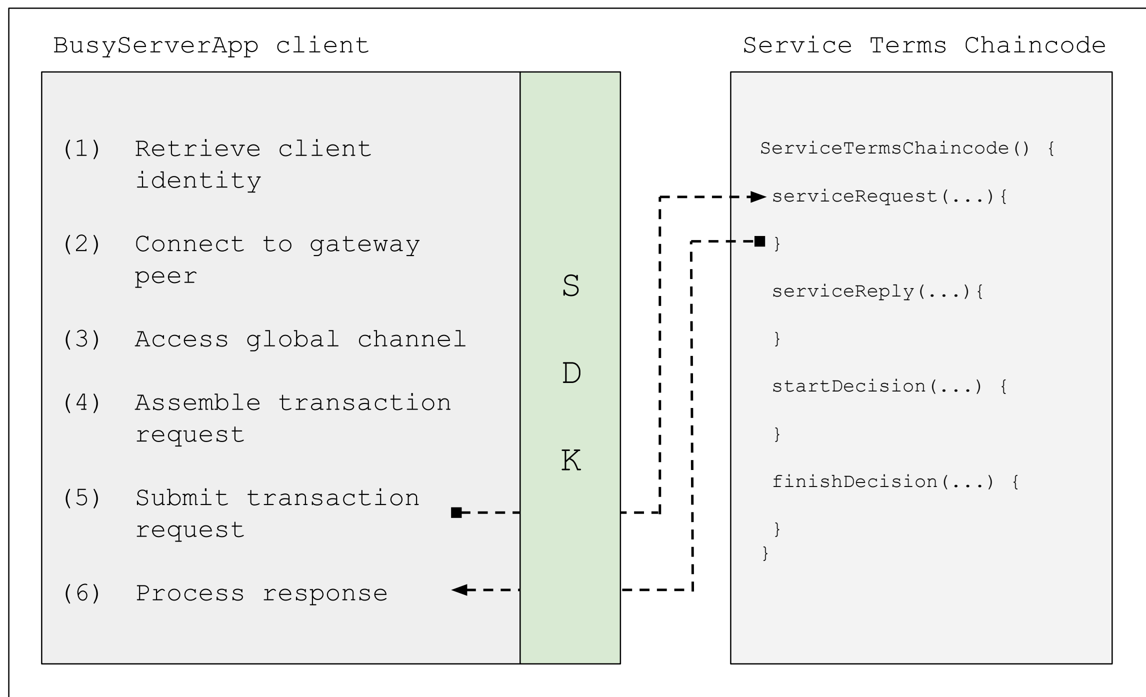

Figure 4.9. Interaction between *BusyServerApp* clients and *Service Terms Chaincode* (the function being called is *serviceRequest*).

A code snippet with the Java implementation of the *BusyServerApp* class is shown next. The six communication stages between clients and the *Service Terms Chaincode* are clearly defined by the comments inside the code.

```
public class BusyServerApp {
    public static void main(String[] args) throws Exception {
        /* Load wallet with digital identites */
        Path walletPath = Paths.get("wallet");
        Wallet wallet = Wallets.newFileSystemWallet(walletPath);
        /* Load connection profile */
```

```
        Path networkConfigPath = Paths.get("connection-org1.yaml");
        Gateway.Builder builder = Gateway.createBuilder();
        builder.identity(wallet,"appUser");
        /* Create a gateway connection */
        try (Gateway gateway = builder.connect()) {
            /* Get network channel and chaincode */
            Network network = gateway.getNetwork("globalchannel");
            Contract contract = network.getContract("ServiceTerms");
            /* Submit transactions */
            contract.submitTransaction("serviceRequest",...);
        }
    }
}
```

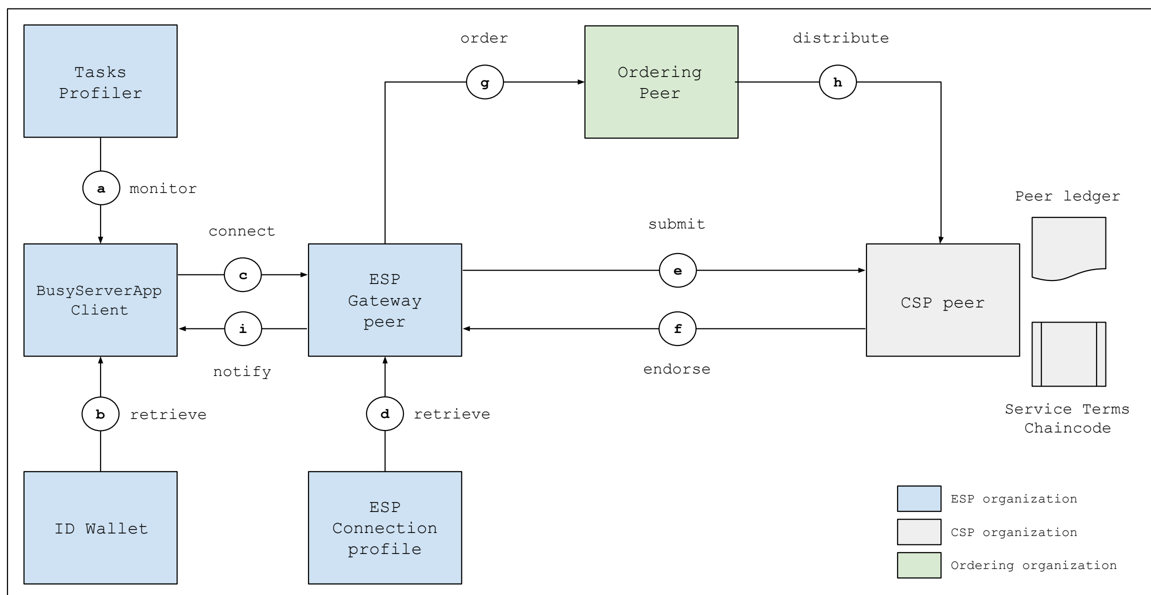

Figure 4.10. *Communication sequence between BusyServerApp clients and the EdgeChain network*

With respect to the decision engine of our model, *BusyServerApp* clients integrate the Google optimization tool (*Google OR-Tools*) [1] to solve the task scheduling problem designed for the *EdgeChain* solution (the problem is explained in section 3.4 *"Task Sharing Model at the Server Level"*). *Google OR-Tools* is an open source optimization software available for multiple programming languages including Java

that offers a variety of solvers for combinatorial optimization. *Google OR-Tools* specializes in finding optimal schedules for allocation problems composed of a set of task objects that need to be distributed among a fixed set of resources that provide labor. *EdgeChain* makes use of the GLOP solver to find optimal solutions for the linear program in 3.19x. An illustration of the communication sequence between *BusyServerApp* clients and the *EdgeChain* network can be found in Figure 4.10.

4.4.2 Cloud Level Client Application

The *CloudLevelApp* client is a Java application installed on the cloud datacenter (i.e. CSP) to negotiate task sharing service requests with busy servers. The client calls functions inside *Service Terms Chaincode* to reply to service requests and notify the network with available computer resources in *EdgeChain* network. As in *BusyServerApp* clients, the *CloudLevelApp* client will connect to the blockchain infrastructure using the Fabric Java SDK. The SDK provides functionality to reach other nodes in the global channel. The *CloudLevelApp* client uses a gateway peer (i.e. peer0.org5) and the connection profile inside the CSP organization to submit transaction proposals to the network. Moreover, the gateway peer makes use of its local copy of *Service Terms Chaincode* and the Fabric Contract API to construct transaction proposals that will be sent to the network. An illustration of the connection scheme between *CloudLevelApp* and the gateway peer in CSP is presented in Figure 4.11.

The *CloudLevelApp* client can submit transactions on behalf of the monitoring tool sitting at the cloud level. This module has an overview of the computer resources in the network and can shortlist a set of candidate servers to take on computational tasks queued in the system. To do that, the client starts by retrieving *x.509* certificates from the local wallet that correspond to pre-authorized users with clearance to access the network. The connection profile with the configuration of the gateway peer for the parent CSP organization is loaded next. The gateway peer provides an entry point for the *CloudLevelApp* client to access the global channel and the *Service Terms Chaincode*. From the gateway peer location, the

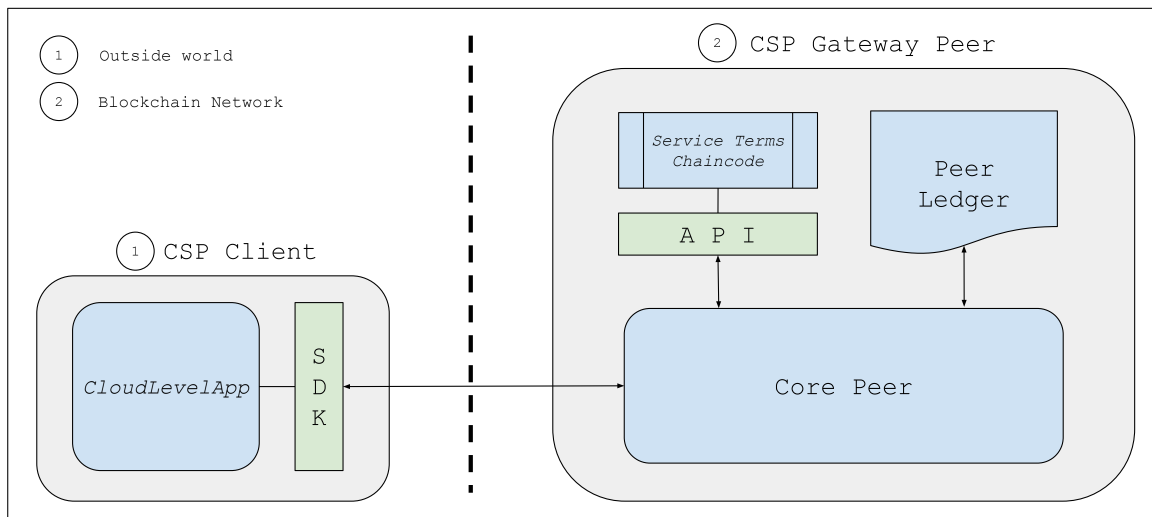

Figure 4.11. Communication between *CloudLevelApp* clients and CSP gateway peers

CloudLevelApp client may call *serviceReply* transactions to update the blockchain ledger with the information about available computer resources in the edge network. A simplified diagram of how the *CloudLevelApp* client interacts with the *Service Terms Chaincode* can be found in Figure 4.12. A summary of the labor inside *CloudLevelApp* client is presented next:

(1) Retrieve *x.509* client identities from the local CSP wallet
(2) Connect to the designated CSP gateway peer using the information in the connection profile of the parent organization
(3) Access the global channel using Java SDK functionality
(4) Connect to *Service Terms Chaincode* and submit transaction proposals to the network
(5) Stay alert of transaction notifications and process responses accordingly

A code snippet with the Java implementation of the *CloudLevelApp* class is shown next. The six communication stages between clients and the *Service Terms Chaincode* are clearly defined by the comments inside the code.

```
public class CloudLevelApp {
    public static void main(String[] args) throws Exception {
        /* Load wallet with digital identites */
        Path walletPath = Paths.get("wallet");
```

```
        Wallet wallet = Wallets.newFileSystemWallet(walletPath);
        /* Load connection profile */
        Path networkConfigPath = Paths.get("connection-org1.yaml");
        Gateway.Builder builder = Gateway.createBuilder();
        builder.identity(wallet,"appUser");
        /* Create a gateway connection */
        try (Gateway gateway = builder.connect()) {
            /* Get network channel and chaincode */
            Network network = gateway.getNetwork("globalchannel");
            Contract contract = network.getContract("ServiceTerms");
            /* Submit transactions */
            contract.submitTransaction("serviceReply",...);
        }
    }
}
```

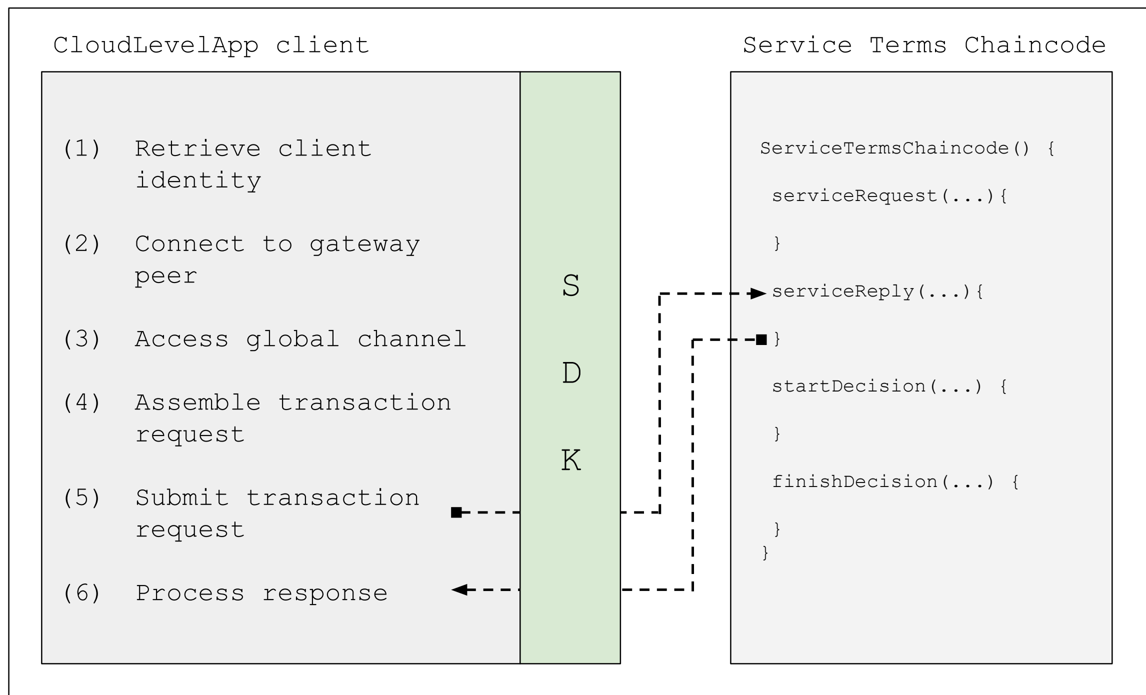

Figure 4.12. Interaction between *CloudLevelApp* clients and *Service Terms Chaincode* (the function being called is *uploadTask*).

For our demo, we assume that all the servers in *EdgeChain* are potential candidates to accept queued tasks in the network. However, in production scenarios,

there is a variety of open-source applications available that might be installed on the cloud datacenter to monitor computer resources and blockchain performance in the edge network. This information may be used as input for the allocation decision engine of our model. An illustration of the communication sequence between the *CloudLevelApp* client and the blockchain network can be found in Figure 4.13.

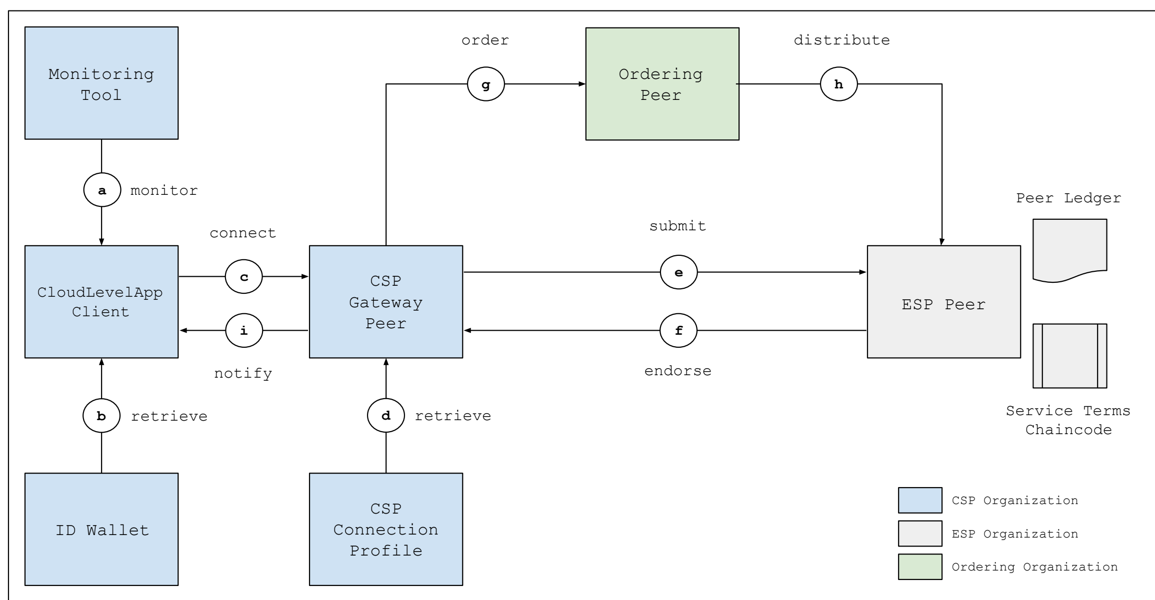

Figure 4.13. Communication sequence between *CloudLevelApp* clients and the *EdgeChain* network

4.4.3 Private Sharing Client Application

PrivateSharingApp clients are Java applications installed on multiple ESP servers to oversee the task sharing process between busy servers and service providers. They may call the functionality available in the *Task Sharing Chaincode* to upload-/download data of queued tasks and their results. *PrivateSharingApp* clients use the Fabric Java SDK module to reach other nodes through available private channels. The gateway peers in their parent organizations (i.e. peer$N$, N={1,2,3,4}) are used to submit transaction proposals to the network. In fact, gateway peers make use of their local copy of the *Task Sharing Chaincode* and the Fabric Contract API to construct transactions that will be submitted to the network. An illustration of the

way *PrivateSharingApp* clients communicate with gateway peers in their parent organizations is presented in Figure 4.14.

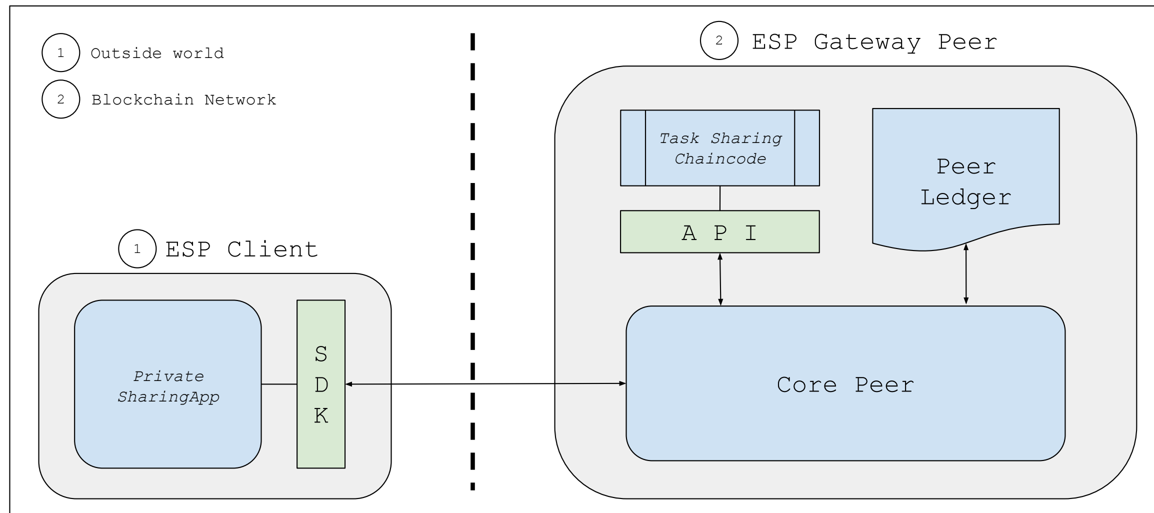

Figure 4.14. Communication between *PrivateSharingApp* clients and ESP gateway peers

PrivateSharingApp clients can submit transactions on behalf of busy servers and service providers to privately manage the task sharing process between each other. In fact, they can upload, remote execute, and download the results of queued computational tasks in the network. *PrivateSharingApp* clients have two versions, one for busy servers and one for service providers that get used depending on the server's role in the task sharing service. They start their duties by retrieving *x.509* digital certificates from their local wallets. Then, they load the connection profiles with the configuration of designated gateway peers for their parent ESP organizations. The gateway peers provide an entry point for *PrivateSharingApp* clients to access the private channels and submit transactions to the network. The gateway peers make use of their local copy of the *Task Sharing Chaincode* and the Fabric Contract API to construct a transaction proposal that will go through the lifecycle of *Computationa Task* objects. An illustration of the interaction scheme between *PrivateSharingApp* and the *Task Sharing Chaincode* can be found in Figure 4.15. A summary of the labor inside *PrivateSharingApp* clients is presented next:

(1) Retrieve *x.509* client identities from local ESP wallets

(2) Connect to designated ESP gateway peers using the information in the connection profiles of the ESP organizations

(3) Access private channels using Java SDK functionality

(4) Connect to *Task Sharing Chaincode* and submit transaction proposals to the network

(5) Stay alert of transaction notifications and process responses accordingly

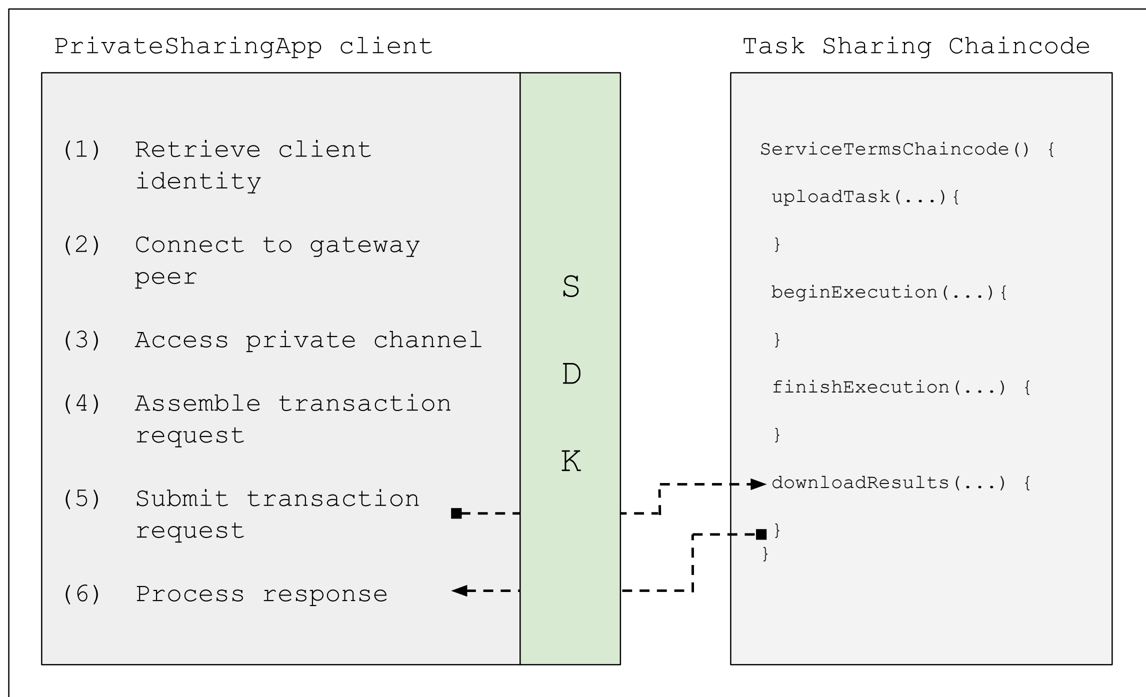

Figure 4.15. Interaction between *PrivateSharingApp* clients and *Task Sharing Chaincode* (the function being called is *downloadResults*).

A code snippet with the Java implementation of the *PrivateSharingApp* class is shown next. The six communication stages between clients and the *Task Sharing Chaincode* are clearly defined by the comments inside the code.

```
public class PrivateSharingApp {
    public static void main(String[] args) throws Exception {
        /* Load the wallet with digital identities*/
        Path walletPath = Paths.get("wallet");
        Wallet wallet = Wallets.newFileSystemWallet(walletPath);
        /* Load connection profile */
```

```
        Path networkConfigPath = Paths.get("connection-org1.yaml");
        Gateway.Builder builder = Gateway.createBuilder();
        builder.identity(wallet,"appUser");
        /* Create a gateway connection */
        try (Gateway gateway = builder.connect()) {
            /* Get network channel and chaincode */
            Network network = gateway.getNetwork("privatechannel");
            Contract contract = network.getContract("TaskSharing");
            /* Submit transactions */
            contract.submitTransaction("uploadTask",...);
        }
    }
}
```

An illustration of the communication sequence between the *PrivateSharingApp* clients and the blockchain network can be found in Figure 4.16.

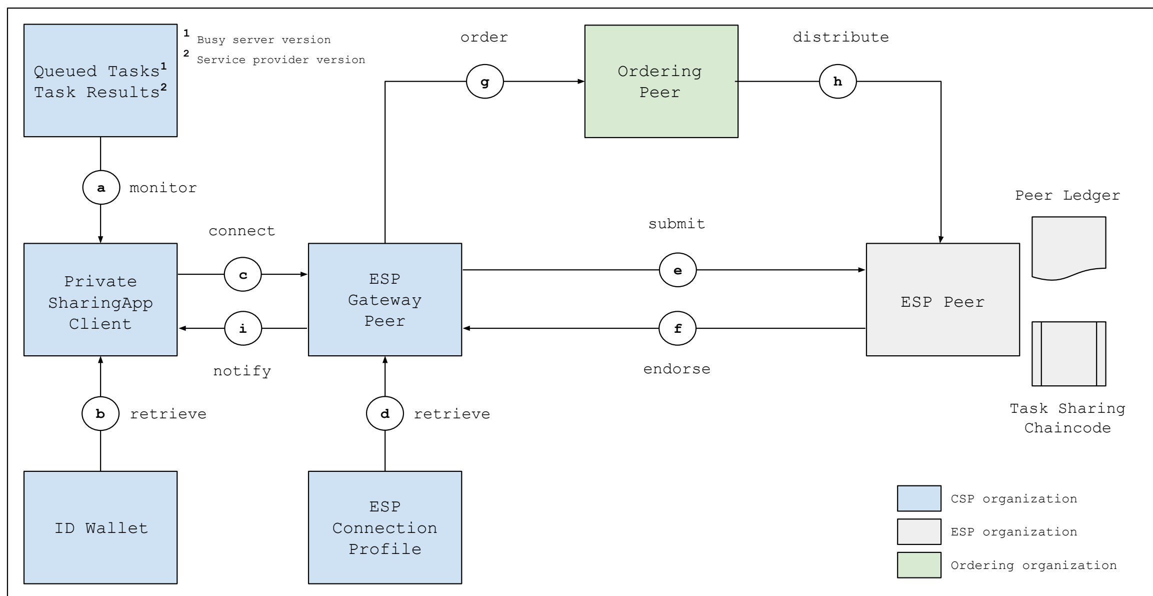

Figure 4.16. Communication sequence between *PrivateSharingApp* clients and the blockchain network

4.5 Performance Evaluation Results

Let's recap the fact that *EdgeChain* network runs on a physical cluster of four identical computers. The network consists of a Fabric-CA PKI instance, a Raft-based Ordering Service module, a consortium of five organizations, seven blockchain channels, two pieces of chaincode, and three client applications that make up the blockchain-based task sharing solution. Every computer has a 3.40 GHz x 8 i7 core processor, 7.16 GiB of available memory, and 500 GBs of storage. In this section, we evaluate the impact of a few configurable parameters of the Fabric platform on the transaction latency and transaction throughput induced by the blockchain network. Latency is defined as the time (in seconds) taken by transactions from when they leave client applications to when they get committed to the blockchain ledger. Throughput, on the other hand, is defined as the rate (in transactions per second) at which transactions are committed to the blockchain ledger. The evaluation of transaction latency and transaction throughput is the standard practice to measure the response time and scalability of a blockchain solution. We use these two parameters as the primary metrics to analyze the performance of *EdgeChain*. If not otherwise specified, the results presented in this section are averaged over multiple runs of the same experiment.

4.5.1 Impact of State Database

Let's recall that Hyperledger Fabric currently offers two ledger implementations to store key-value pair (KV) data related to blockchain objects: (1) LevelDB and (2) CouchDB. LevelDB ledger is the default implementation of a state database in Fabric nodes that comes embedded inside core peers. It is lightweight but supports only a limited number of simple operations on the database. CouchDB ledger, on the other hand, is an SQL database that runs separately from peer nodes. It supports advanced operations on the database, complex indexing methods, and complex data structures. For this latency experiment, all the Fabric configuration parameters are fixed except for the ledger state database. The parameter values set for this experiment can be seen in Table 4.7. Figures 4.17 and 4.18 show the impact

of the type of ledger database on the average transaction latency experienced by transactions. *Service Terms Chaincode* and Task Sharing Chaincode transactions are taken into account. The number of transaction runs is 5000.

Parameter	**Value**
Endorsement policy	Majority (default)
Block size	1 transaction
Block timeout	1 second

Table 4.7. Fabric parameters for the database experiment.

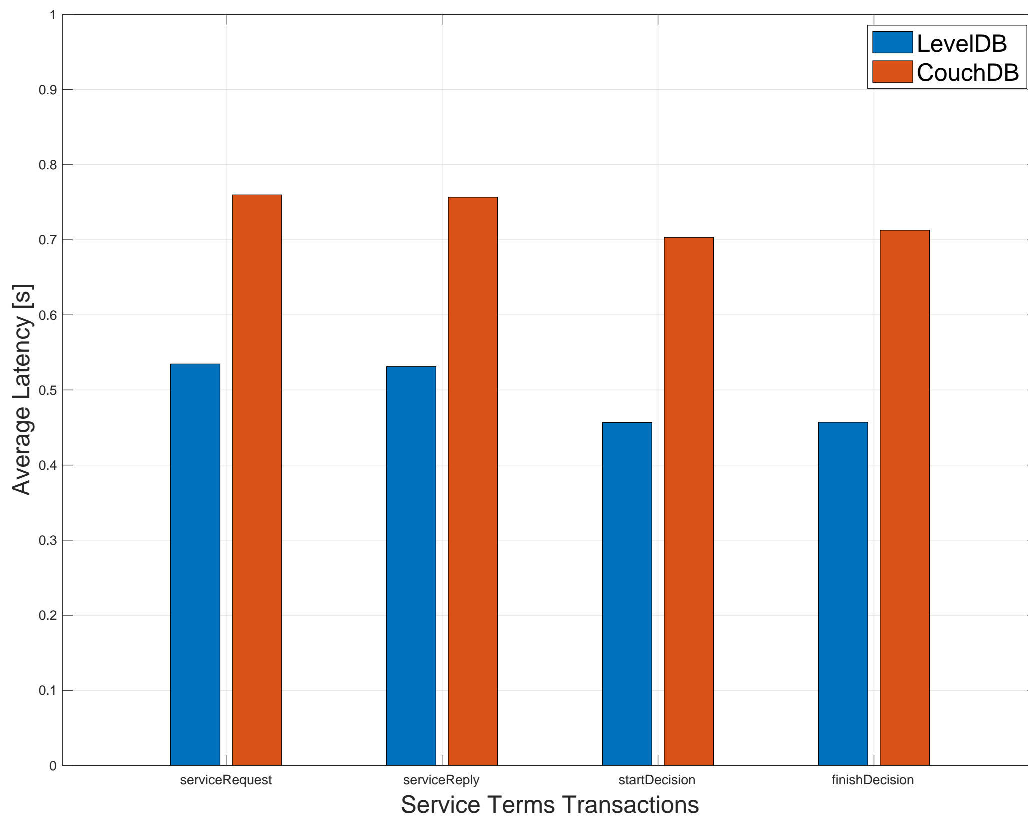

Figure 4.17. Average latency of *Service Terms Chaincode* transactions as a function of state database.

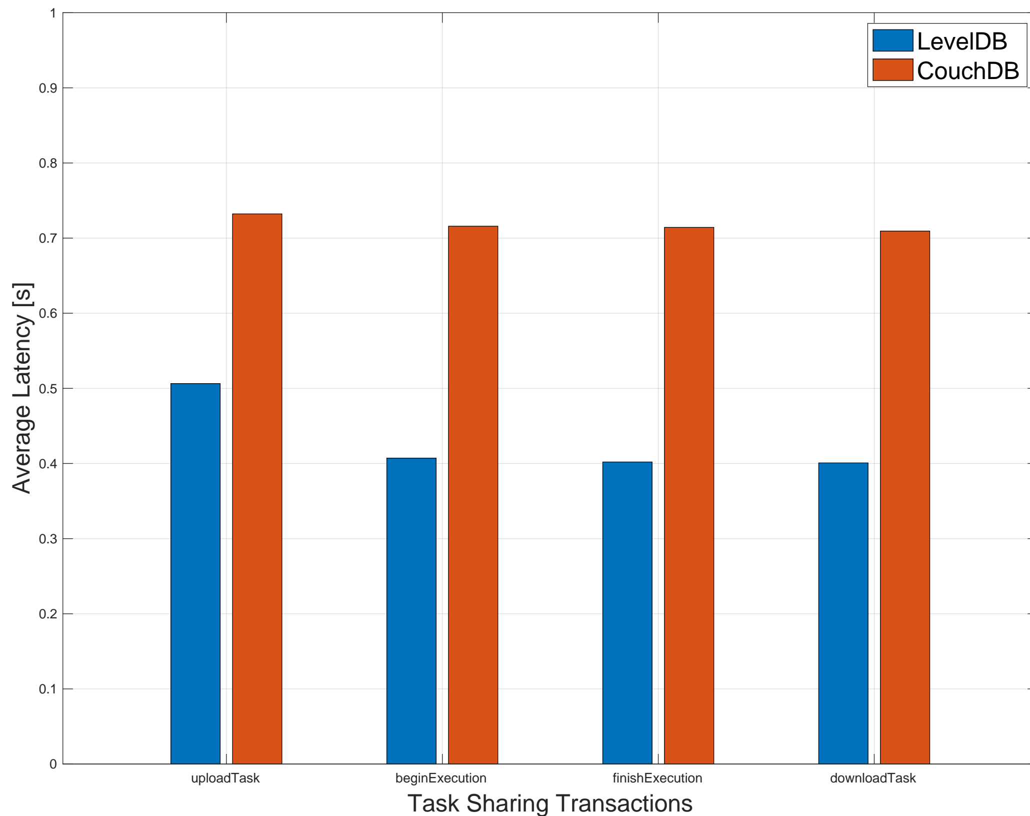

Figure 4.18. Average latency of *Task Sharing Chaincode* transactions as a function of state database.

Observation 1: The results show that the latency experienced by transactions for the CouchDB ledger implementation is significantly greater when compared to the latency experienced when LevelDB ledger is installed. The main reason for the difference is that LevelDB is a state database embedded in the peer nodes while CouchDB is a database that runs separately from peer nodes using REST APIs over a secure HTTP connection. The communication between peer nodes and the REST APIs adds additional steps to the write/read operations that transactions force on the CouchDB ledger. Across transactions, there is also a slight difference in the latencies introduced by the two pieces of chaincode. As for the *Service Terms Chaincode*, the complexity of *ServiceRequest* and *ServiceReply* transactions is higher than that of *startDecision* and *finishDecision.* The higher number of Key-Value (KV) ledger operations introduced by the first two has a slight impact

on the latency being measured. This effect can be seen in both, CouchDB and LevelDB, implementations. As for the *Task Sharing Chaincode*, the effect of KV ledger operations can also be seen in the latency introduced by *uploadTask* transaction with respect to the other three (i.e. *beginExecution*, *finishExecution*, and *downloadTask*) although the effect is more noticeable in the LevelDB implementation.

Guideline: LevelDB ledger is a simpler and faster option than CouchDB ledger and might be a better design choice for latency-sensitive applications that manage data with low complexity structures. CouchDB, on the other hand, is slower but can handle more sophisticated data structures. It may be a good choice for applications that can tolerate higher response latencies.

Action: Move on to the next experiments setting LevelDB as the state database for *EdgeChain* network.

4.5.2 Impact of Endorsement Policy

Let's recall that, at the peer level, the consensus mechanism in Fabric networks rely on the idea of endorsements. Endorsement policies are the rules that define the minimum number of digital signatures that must be collected from the blockchain network before transaction proposals get approved and moved to the ordering service. When client peers submit transaction proposals to a blockchain channel, they are sent to endorsing peers defined on the channel. Endorsing peers receive the proposals and simulate them with their local copy of the chaincode. If the simulation process produces the same output as the proposals, transactions are signed with the endorser's private key and are sent back to the client that created them. Endorsing signatures are a method to validate that transaction results strictly follow the installed logic on the blockchain. From a macro perspective, endorsement policies dictate the level of democratization for the voting system to approve transactions, that is, minority, majority, or unanimity approvals. The policies are the result of the combination of three logic operators: (1) AND, (2) *OR*,

and (3) *NOutOf*. The operands are always members of the organizations participating in the blockchain consortium. For example, an endorsement policy defined by the expression $AND(Org1MSP.peer, Org2MSP.peer)$ means that any peer in both, Org1 and Org2, must sign a transaction proposal before it is declared valid. $NOutOf(1, Org1MSP.peer, Org2MSP.peer, Org3MSP.peer)$, on the other hand, means that at least 1 out of the 3 organizations in the policy (i.e. Org1, Org2, and Org3) must endorse a transaction proposal to get it approved. For this latency experiment, all the Fabric configuration parameters are fixed except for channel endorsement policies. The parameter values set for this experiment can be seen in Table 4.8. Figures 4.19 and 4.20 show the impact of endorsement policies in the average transaction latency experienced by transactions. *Service Terms Chaincode* and Task Sharing Chaincode transactions are taken into account. The number of transaction runs is 5000.

Parameter	**Value**
State database	LevelDB
Block size	1 transaction
Block timeout	1 second

Table 4.8. Fabric parameters for the endorsement policy experiment.

Observation 2: For the *Service Terms Chaincode*, we evaluate three endorsement policies: (1) minority approval (20% endorsement signatures, 1 out of 5 organizations), (2) simple majority approval (>50% endorsement signatures, 3 out of 5 organizations), and (3) unanimity approval (100% endorsement signatures, 5 out of 5 organizations). For the *Task Sharing Chaincode*, we evaluate two endorsement policies: (1) weak majority approval (50% endorsement signatures, 1 out of two organizations), and (2) unanimity approval (100% endorsement signatures, 2 out of 2 organizations). The results show that the latency experienced by transactions

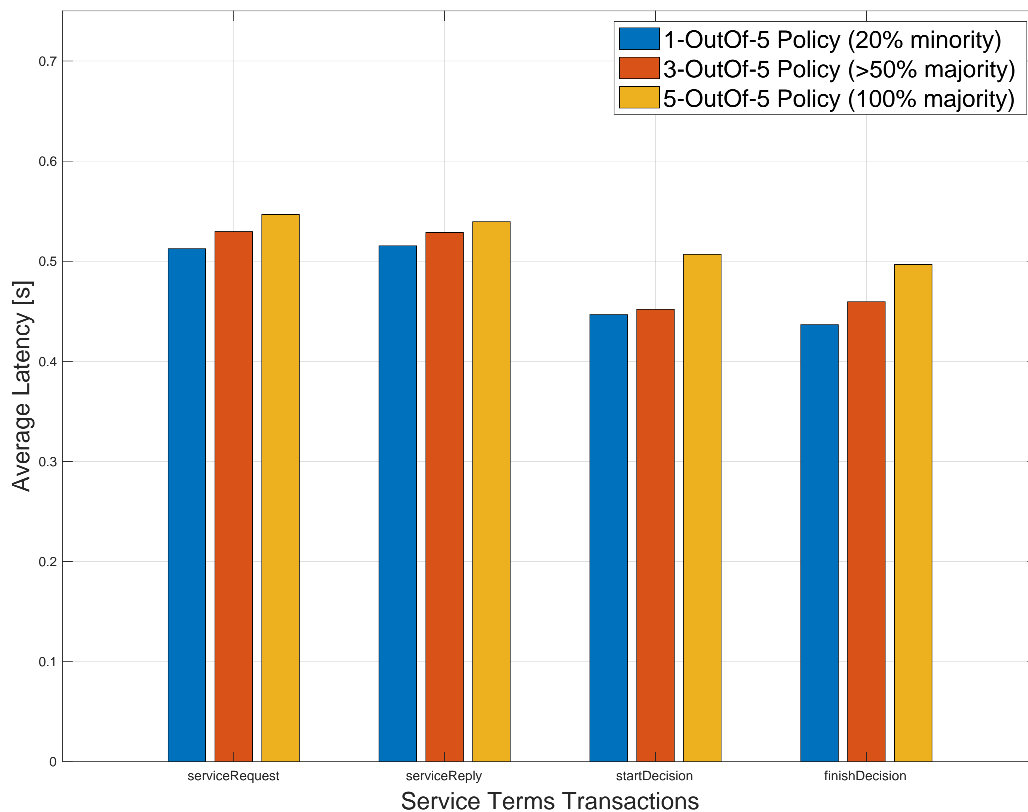

Figure 4.19. Average latency of *Service Terms Chaincode* transactions as a function of endorsement policy.

is indeed affected by the number of signatures required by the endorsement policy. As for the *Service Terms Chaincode*, the minority approval and simple majority approval show similar latency times. The unanimity approval, on the other hand, slightly increases the latency compared to the other two policies. As for the *Task Sharing Chaincode*, the unanimity approval forces a slight increment in the latency for the weak majority approval, however, the increment is almost negligible. In general, the increment in transaction latency can be explained in terms of three main operations that take place during the endorsement validation phase. First, *x.509* identity certificates of endorsing peers are de-serialized from the membership module back to the proposing peer. Second, once de-serialized, the certificates are validated against the list of organization MSP identifiers. And third, the signatures of endorsed transactions have to be verified to confirm the identity of

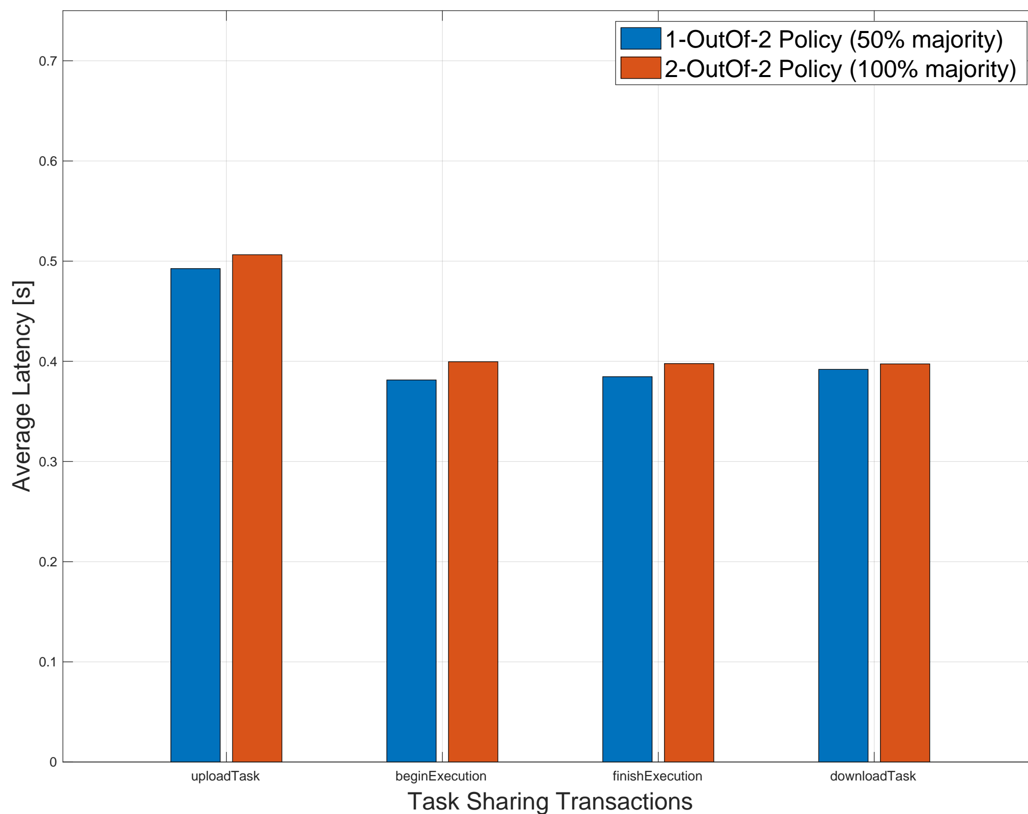

Figure 4.20. Average latency of *Task Sharing Chaincode* transactions as a function of endorsement policy.

endorsing peers. The three operations are executed by the proposing peers every time endorsed transactions enter their premises. We can see that the effect of different endorsement policies is mostly experienced by the *Service Terms Chaincode* which operates in the *global channel*. The *global channel* contains 5 organizations and an equal number of peers, therefore, a change in the number of required signatures to approve transactions in this channel has greater latency impact than similar changes in channels with less number of participants. This is true simply by the lesser number of operations that must take place in smaller channels during the endorsement validation phase. This can be corroborated with the behavior of transactions in the *Task Sharing Chaincode* which operates in private channels with only two organizations and two peers.

Guideline: The complexity of endorsement policies is a factor that impacts the scalability of the blockchain network. When a network requires a larger number of endorsements, transactions must be simulated and signed at a higher number of peer nodes during the endorsement phase. Therefore, the higher the number of required signatures, the higher the latency experienced by transactions. For better performance, set endorsement policies with only a few required signatures.

Action: Move on to the next experiments setting the endorsement policy to one required signature ($NOutOf$, $N = 1$) for all *EdgeChain* smart contracts.

4.5.3 Impact of Transaction Arrival Rate and Block Size

In a Fabric network, transactions are ordered and packed into blocks at the ordering service. Once created, blocks are sent back to organization peers for final verification and committing to their local copies of the blockchain ledger. The block size is a configurable parameter of the blockchain network that defines the number of packed transactions in a block. Unlike the endorsement phase in which processing is done in a per-transaction style, the processing of blocks at the peer level is done one block at a time. In fact, there is a tight relationship between block size, transaction latency, and transaction throughput inside the network. A deficient pick of block size may impact the speed at which blocks are created when the arrival rate is not sufficiently. This may have transactions wait too long at the ordering service causing bottlenecks in the network. For a better picture of this latency and throughput experiment, we study these parameters in conjunction with the transaction arrival rate. The parameter values set for this experiment can be seen in Table 4.9. All the Fabric configuration parameters are fixed except for the block size. Figures 4.21 to 4.24 show the impact of block size on the average transaction latency and the average transaction throughput experienced by *EdgeChain*. The network is tested under multiple block sizes and transaction arrival rates.

Observation 3: We evaluate four block size policies ($blocksize = 1, 3, 5, 7$) at different transaction arrival rates to catch potential bottlenecks in the network. The tension

Parameter	**Value**
State database	LevelDB
Endorsement Policy	*NOutOf*, $N = 1$
Block timeout	1 second

Table 4.9. Fabric parameters for the transaction arrival rate & block size experiment.

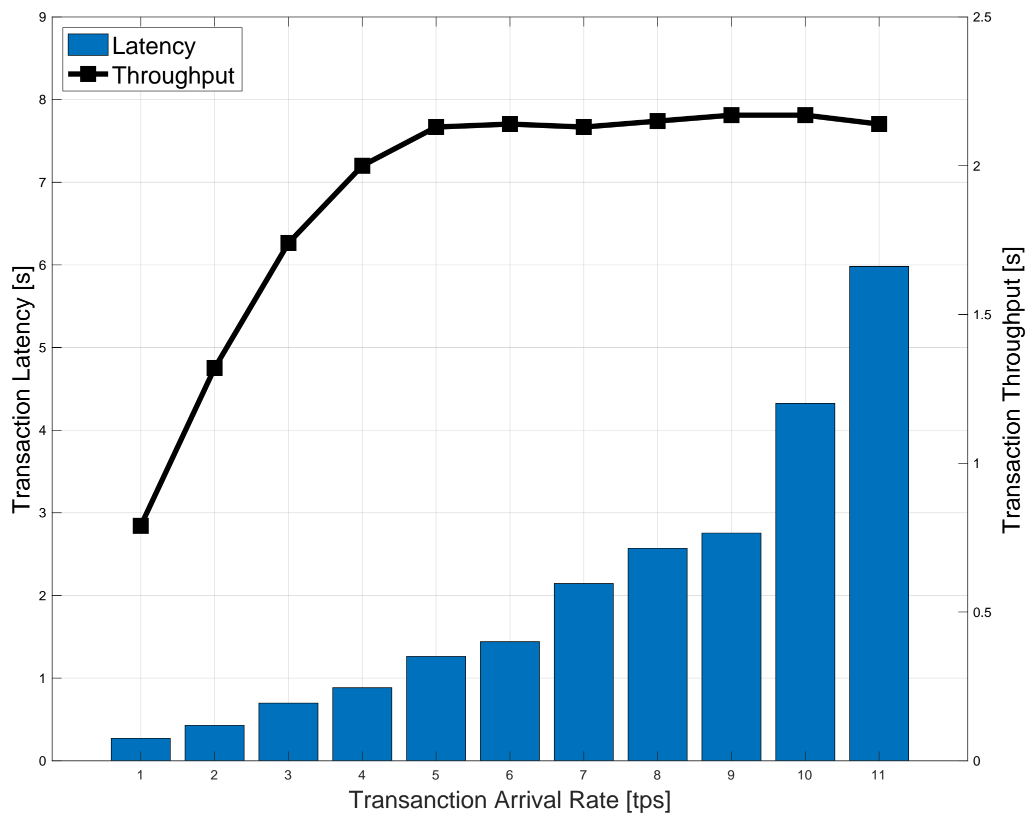

Figure 4.21. Average Transaction Latency and Average Transaction Throughput versus Transaction Arrival Rate with block size equal to 1.

between latency and throughput becomes clear when the block size policy and the transaction arrival rate vary. For $blocksize = 1$, the throughput increases linearly

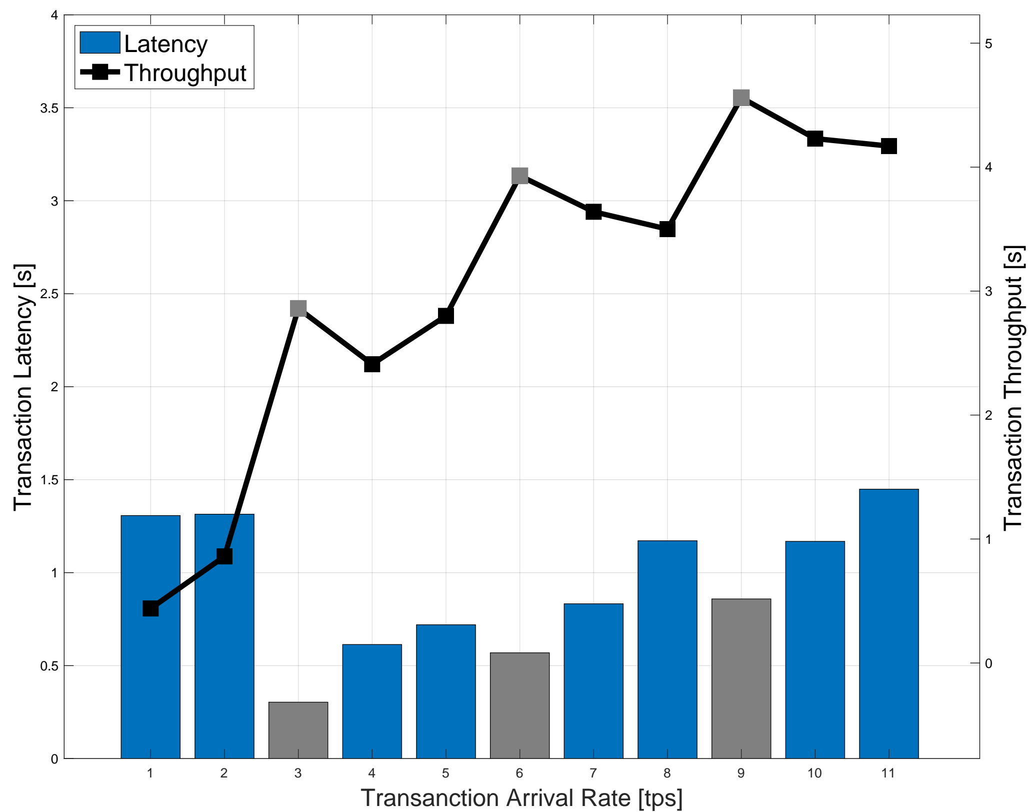

Figure 4.22. Average Transaction Latency and Average Transaction Throughput versus Transaction Arrival Rate with block size equal to 3.

as expected before reaching the saturation point. When the arrival rate gets close to the saturation point, the latency increases significantly from less than half of a second to a few seconds. This effect can be explained by the growing number of service transactions waiting at the ordering service queue when the arrival rate increases. The waiting time at the queue affects the ultimate commit latency time and becomes a bottleneck in the system. The behavior can be seen in Figure 4.21.

Observation 4: When we increase the block size to 3, 5, and 7, the transaction latency remains unaltered at approximately 1.3 seconds when the arrival rate is below the block size. This can be explained due to the block generation time being conditioned by the *block timeout* parameter that gains prominence when the rate

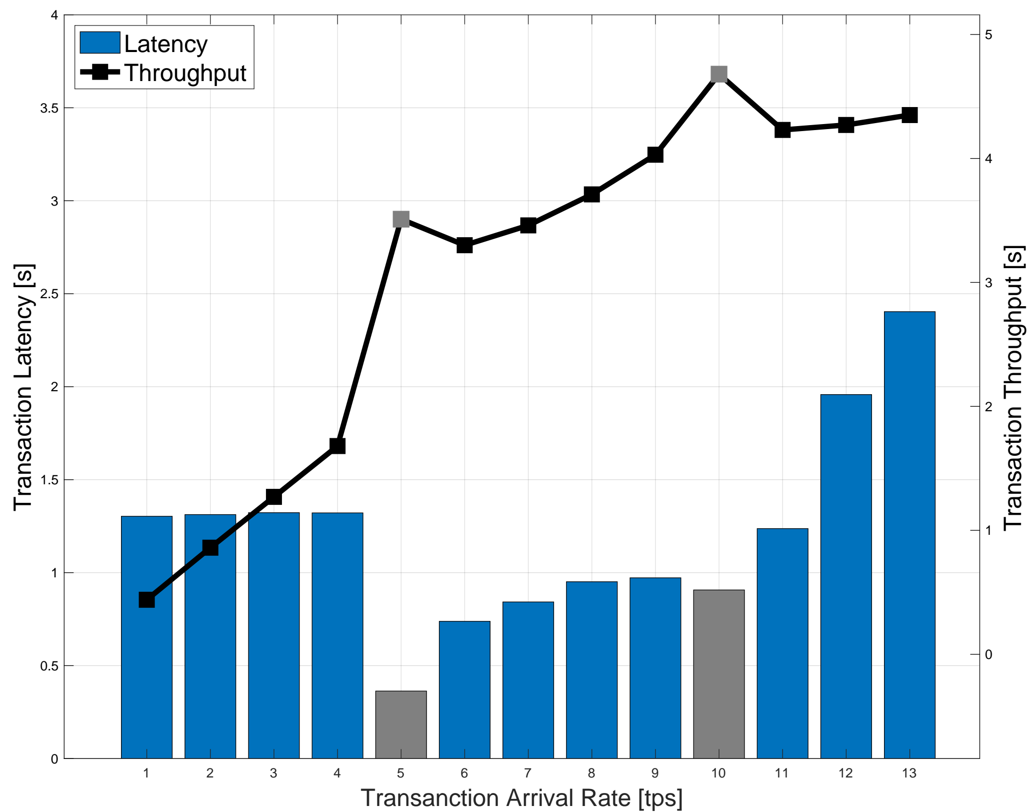

Figure 4.23. Average Transaction Latency and Average Transaction Throughput versus Transaction Arrival Rate with block size equal to 5.

of incoming transactions is below the size of a block. In this scenario, ordered transactions have to wait for at least *"block timeout"* seconds before getting committed to the network. In our tests, the *block timeout* is set to 1 second. For arrival rates above the block size, the transaction latency increases with the arrival rate. An interesting behavior is spotted at the points where the arrival rate matches the block size or its multipliers ($blocksize = N, multipliers = 2*N, 3*N, \ldots$). At these points, the network experiences a decrease in transaction latency that can be explained by a faster generation of blocks given that the number of incoming transactions in a batch meets the block transaction quota exactly hence no remaining transactions have to wait in the ordering service queue for the next round of block generation and commit phase. On average, this effect translates translates in a slight decrease in committing times that can be spotted on the gray bars in Figures 4.22, 4.23, and

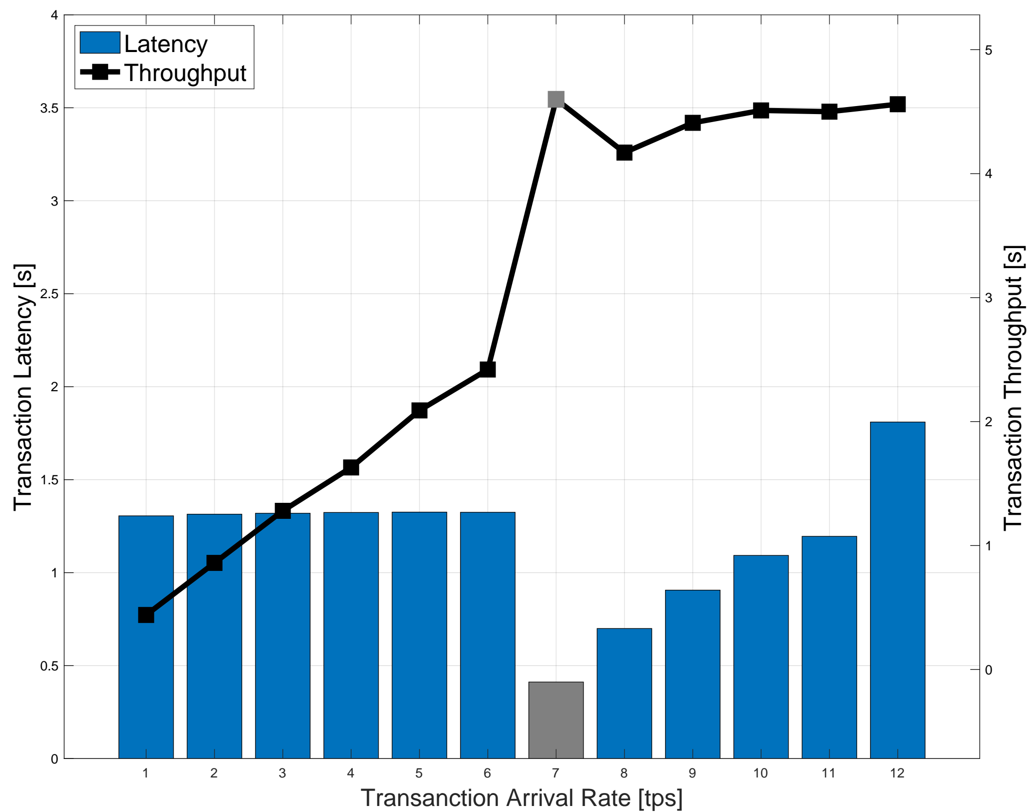

Figure 4.24. Average Transaction Latency and Average Transaction Throughput versus Transaction Arrival Rate with block size equal to 7.

4.24.

Observation 5: Same as transaction latency, the transaction throughput increases as we increase the block size ($blocksize = 3, 5, 7$). Again, for arrival rates equal to the block size or its multipliers, the transaction throughput experiments a boost that can be explained in similar terms as the latency case. At these points, blocks are created at a faster rate because of the synchronization between the transaction arrival rate and the transaction quota at the block level that, on average, reduces the waiting time of service transactions at the ordering service hence increasing the throughput produced by the blockchain. The effect is highlighted with gray square markers in Figures 4.22, 4.23, and 4.24.

Guideline: To achieve higher transaction throughput and lower transaction latency, always match the block size to the transaction arrival rate when the network sits below the saturation point. We recognize that when the blockchain finds optimal latency and throughput levels, the network may offer faster and more scalable task sharing services.

5 Conclusions

Multi-access Edge Computing (MEC) is a network architecture that relocates cloud services such as processing power, memory, storage, and control functions geographically close to end-users at the edge of mobile networks. MEC is one of the key enablers of the 5G standard and its promises of future communication networks with high bandwidth, low latency, and the ability to handle a massive amount of subscribers. The efficient administration of resources is one of the fundamental problems regarding MEC networks due to the tension between expensive communication infrastructure and the increasing demand for resources. Although the existing literature tends to approach task allocation problems from the mobile user perspective, a new trend for task sharing focused on devices closer to the core of the network is emerging rapidly. However, there exist security and privacy concerns that need to be addressed before a feasible model for task sharing collaboration can be put into production. Our thesis work explores the use of blockchain technologies to develop a secure and private task sharing solution for edge computing servers. The proposed prototype operates on a decentralized approach that relies on the blockchain services provided by the Hyperledger Fabric platform, the services are (1) membership, (2) consensus, (3) ledger, and (4) smart contract modules.

Hyperledger Fabric is a membership-only blockchain platform that integrates Public Key Infrastructure (PKI) to validate cryptographic identities and the allowed behavior of participating nodes in the network. Fabric's membership service prevents external attackers from having access to the edge network and interfering

with the normal operation of the task sharing mechanism. The consensus mechanism in Fabric platforms consists of a signature-based system of endorsements that guarantees that the computing tasks circulating in the network strictly follow the pre-approved rules of the network. Ledger services in Fabric maintain modular databases with the historical records of all blockchain transactions in the edge network. The ledger design is flexible and permits the coexistence of several blockchain databases with different rules and access permissions. Our task sharing model leverage this feature to keep the confidentiality of tasks circulating in the network and expose them only to servers participating in task sharing services. In addition to that, Fabric smart contract services allow our task sharing model to implement the logic behind it. Our design decouples the task sharing model into two separate smart contracts: (1) Service Terms Chaincode for service negotiation, and (2) *Task Sharing Chaincode* for the actual sharing of computational tasks. The smart contracts operate in separate blockchain channels with different rules, ledgers, and peer-to-peer communication networks so that tasks and their data are kept private to service providers only at any time. We run by the premise of public negotiation terms but private task sharing services. The premise may serve as a collaboration incentive for actors with contentious relationships such as market competitors.

The adoption of the 5G standard will open the opportunity for new decentralized ownership models that may phase out centralized traditional models that may be unfit to handle emerging business cases in the IT sector. Given the intense capital expenditure directed towards the deployment of 5G networks, the adoption of infrastructure cooperation may be an alternative to lower infrastructure implementation costs without affecting the delivery of applications and services to end users. The decentralized services offered by blockchain technologies may become the facilitator for automated resource management schemes that adapt better to the complexities of sophisticated 5G applications. Blockchain-based networks provide a trust model for (1) record-keeping transactions in distributed ledgers, and (2) secure interactions for network participants through immutable smart contract

services. The marketization of network resources in the telecommunications sector may impact network services such as wireless spectrum allocation, network slicing, and the lease of computer resources. Shifting from centralized ownership models to on-demand smart-contract-based infrastructure sharing may require market models based on blockchain that can incentivise consortiums of sellers, buyers, and regulators for secure and private market services, more specifically, trading services.

The premise of this thesis consists in the idea that MEC networks can benefit from the notarization, ownership, and chain-of-custody services offered by the Hyperledger Fabric blockchain to address security and privacy concerns related to collaboration mechanisms for edge computing servers. We propose a conceptual framework for task sharing collaboration in MEC architectures using the Hyperledger Fabric platform. The framework puts up lightweight and extensible Service-Oriented Architecture (SOA) for secure and private task sharing collaboration specially designed for MEC networks. The system is adapted to dynamically discover servers, allow peer-to-peer communication, and provides the logic for task sharing services among server nodes. The proposed MEC framework supports decentralized computing services with dynamic task scheduling and unified management of resources while maintaining security capabilities with a special focus on the integrity and confidentiality of tasks and their data.

At the blockchain level, we propose a task sharing design that accommodates a set of edge servers and the cloud level that get together to form a consortium in the Hyperledger Fabric platform. A set of computational tasks that require processing services are Fabric objects, and the logic of the proposed task sharing scheme is embedded in the chaincode installed on the network peers. The dynamics of the solution can be described in four phases: (1) service request, (2) discovery of candidate nodes, (3) selection of service providers, and (4) task sharing contract. The model offers enhanced security features that leverage the permission-ed nature of Fabric to validate identities and allowed behavior within the network. The model

also offers enhanced privacy features powered by a smart contract design that makes use of multiple decoupled Fabric channels with separate operation rules, ledgers, and peer-to-peer communication networks. This design choice guarantees that the computational tasks circulating in the network are only exposed to servers participating in task sharing services. At the server level, the task sharing scheme is modeled as a multiprocessor task scheduling problem that allocates a set of precedent-dependent computational tasks among a set of available servers by jointly optimizing the aggregated utility and the makespan of the tasks. In summary, we introduce a reference task-sharing model at the blockchain and server levels that establish a block-based agreement among multiple edge service providers that enables macro deployment and cross-collaboration among them.

We also present the implementation details of *EdgeChain*, a proof of concept demo of the task-sharing model. *EdgeChain* is a Hyperledger Fabric-based solution designed for a consortium of edge service providers to share computational tasks between organizations within a blockchain network. The whole premise of EdgeChain relies on the idea that a consortium of ESPs willing to share their idle resources would reduce the latency associated with the execution of stranded tasks in the system, and also increase the utilization of computer infrastructure installed on the consortium. EdgeChain relies on the membership, consensus, ledger, and chaincode services provided by Hyperledger Fabric to deliver the task sharing solution. *EdgeChain* network runs on a physical cluster of four computers and consists of the combination of Fabric-CA PKI, a Raft-based Ordering Service, a consortium of five organizations (four Edge Server Providers and one Cloud Service Provider), seven blockchain channels, two smart contracts, and three client applications that make up the task sharing solution. The implementation of *EdgeChain* shows that our task sharing model is feasible. The experiments show that our proof of concept generates reasonable results, however, there is still much room for improvement to reduce the latency introduced by service transactions, increase the throughput produced by the blockchain, and improve the overall features offered by the task

sharing model.

The preliminary results suggest that the implementation of our model might not be fast enough to serve near real-time applications. The latency overhead introduced by task sharing transactions exceeds response limits for latency-sensitive applications that fall in the order of the tenths of milliseconds at worst. One possible countermeasure to this is the optimization of the configurable parameters of Fabric (i.e. endorsement policy, state database, block size, batch timeout, etc) so that the latency introduced by the blockchain model is minimized to tolerable levels. Another potential measure consists in making adjustments to the task sharing model at the blockchain level to minimize the number of required transactions. A priori, the transactions in our model could be re-organized to reduce the number of operations on the ledger. Let's recall that transactions in a blockchain network ultimately translate into KV write/read operations on the ledger which add up to the latency overhead introduced by the system, write operations in particular.

At the server level, our current model runs on a decentralized approach where busy servers have full control over their task sharing decisions (i.e. what to share, when, and where to). However, shifting the approach to one where the cloud makes the allocation decisions could accelerate the sharing process. Our original blockchain-based task sharing framework published in 2021 [41] presented a centralized model with the cloud taking control of the allocation process. We believe that the centralized model approach is worth exploring in the future. In a slightly different direction, sharing computational tasks can also be seen as a trading problem rather than a multiprocessor scheduling problem. Our current approach does not bring market tensions to the picture although we believe that market models such as auctions and their variations might capture the nuances of the sharing process more adequately. Let's recall that Edge Service Providers (ESPs) are normally private companies with profit-driven economic incentives. In that case, a viable collaboration model for a consortium of ESPs should include a pricing model that captures the economic reward for the providers of sharing services. A market model can

seemingly introduce a pricing mechanism for the payments and it can be easily programmed as a cryptocurrency in the logic of the Hyperledger Fabric blockchain.

Even though Hyperledger Fabric introduces a strong access control and authentication mechanism for servers at the edge level, security and privacy protection at the end-user level is still an open problem. Also, the mobility of end users is an issue that our model does not yet address. Service break-offs may happen if end-users move across coverage areas of different edge servers. In that scenario, the provision of smooth task ownership handover between server nodes is one of the challenges for the evolution of our task sharing model. Also, the experiments conducted in the results section relied on a load generator that handles the creation of dummy computational jobs using the Acyclic Dependencies Principle. However, that assumption may not necessarily coincide with the structure and complexity of real end-user applications. To conclude this discussion, we believe that any future evolution of our model should focus on two tasks: (1) reduce the latency introduced by the blockchain network, and (2) increase the overall features offered by the task sharing framework.

Before the end, we would like the readers to note that although all of the elements discussed in this work are presented in the context of MEC, the proposed framework is general and can also be used for task sharing problems in a cloud computing setting. Finally, we are hopeful that the telecommunications community can see value in some of the ideas presented in this thesis and build on top of this work.

References

[1] Google optimization tool (*Google OR-Tools*). https://developers.google.com/optimization/introduction/overview. Accessed: 2022-07-22.

[2] Introduction to hyperledger fabric white paper. *The Hyperledger White Paper Working Group, The Linux Foundation Project*, 2018.

[3] System architecture for the 5g system 3gpp ts 23.501 v16.1.0. Technical report, The 3rd Generation Partnership Project, 06 2019.

[4] Kamanashis Biswas and Vallipuram Muthukkumarasamy. Securing smart cities using blockchain technology. 12 2016. doi: 10.1109/HPCC-SmartCity-DSS.2016.0198.

[5] Vitalik Buterin. Ethereum white paper. https://ethereum.org/whitepaper/, 2013.

[6] L. Chen and J. Xu. Socially trusted collaborative edge computing in ultra dense networks. *Procededings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–11, 2017.

[7] L. Chen, S. Zhou, and J. Xu. Computation peer offloading for energy-constrained mobile edge computing in small-cell networks. *IEEE/ACM Transactions on Networking*, 36(3):574–586, 2018.

[8] M. Chiang and T. Zhang. Fog and iot: An overview of research opportunities. *IEEE Internet of Things Journal*, 3(6):854–864, 2016.

[9] Cisco. Cisco visual networking index: Global mobile data traffic forecast update, 2017-2022, white paper. 19, 2019. URL www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html.

[10] Tiago M. Fernández-Caramés, Oscar Blanco-Novoa, Manuel Suárez-Albela, and Paula Fraga-Lamas. A uav and blockchain-based system for industry 4.0 inventory and traceability applications. *Proceedings*, 4(1), 2019. ISSN 2504-3900. doi: 10.3390/ecsa-5-05758. URL https://www.mdpi.com/2504-3900/4/1/26.

[11] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya. Mobile code offloading: From concept to practice and beyond. *IEEE Communication Magazine*, 3(53):80–88, 2015.

[12] M. Fukumitsu, S. Hasegawa, J. Iwazaki, M. Sakai, and D. Takahashi. A proposal of a secure p2p-type storage scheme by using the secret sharing and the blockchain. *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, page 803–810, 2017.

[13] M. Gantz and P. Miller. White paper: The salesforce economy: Enabling 1.9 million new jobs and 389 billion dollars in new revenue over the next five years. *International Data Corporation Technical Report*, 19, 2016.

[14] X. He and S. Wang. Peer offloading in mobile edge computing with worst-case response time guarantees. *IEEE Internet of Things Journal*.

[15] Xingqiu He, Sheng Wang, and Xiong Wang. Providing worst-case latency guarantees with collaborative edge servers. *IEEE Transactions on Mobile Computing*, 2021. doi: 10.1109/TMC.2021.3133306.

[16] W. Johnston, K. Sparks, B. Daly, and R. Gyurek. 5g edge computing white paper. *FCC Technological Advisory Council, 5G IoT Workign Group*, 2018.

[17] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48 (177):203–209, 1987.

[18] K. Kotobi and S. G. Bilen. Blockchain-enabled spectrum access in cognitive radio networks. *2017 Wireless Telecommunications Symposium (WTS)*, page 1–6, 2017.

[19] K. Kotobi and S. G. Bilen. Secure blockchains for dynamic spectrum access: A decentralized database in moving cognitive radio networks enhances security and user access. *IEEE Vehicular Technology Magazine*, page 32–39, 2018.

[20] Khashayar Kotobi and Mina Sartipi. Efficient and secure communications in smart cities using edge, caching, and blockchain. In *2018 IEEE International Smart Cities Conference (ISC2)*, pages 1–6, 2018. doi: 10.1109/ISC2.2018.8656946.

[21] L. Lamport, E. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[22] E. Lawler, J. Lenstra, and K. Kan. Recent developments in deterministic sequencing and scheduling: A survey. *Deterministic and Stochastic Scheduling*, pages 35–73, 1982.

[23] Cheng Li and Liang-Jie Zhang. A blockchain based new secure multi-layer network model for internet of things. In *2017 IEEE International Congress on Internet of Things (ICIOT)*, pages 33–41, 2017. doi: 10.1109/IEEE.ICIOT.2017.34.

[24] X. Ling, J. Wang, T. Bouchoucha, C. Levy, and Z. Ding. Blockchain radio access network (b-ran): Towards decentralized secure radio access paradigm. *IEEE Access*, page 9714–9723, 2019.

[25] M. Liu, F. Yu, Y. Teng, V. Leung, and M. Song. Distributed resource allocation in blockchain-based video streaming systems with mobile edge computing. *IEEE Transactions in Wireless Communications*, 18(1):695–708, 2018.

[26] N. Luong, Z. Xiong, P. Wang, and D. Niyato. Optimal auction for edge computing resource management in mobile blockchain networks: A deep learning approach. *Proceedings IEEE International Conference on Communications*, pages 1–6, 2018.

[27] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys Tutorials*, 19, 2017.

[28] B. Mafakheri, T. Subramanya, L.Goratti, and R. Riggio. Blockchain-based infrastructure sharing in 5g small cell networks. *14th International Conference on Network and Service Management*, page 313–317, 2018.

[29] Y. Mao, C. You, J. Zhang, K. Huang, and K. Letaief. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys and Tutorials*, 19(4):2322–2358, 2017.

[30] Y. Mao, C. You You, J. Zhang, K. Huang, and K. B. Letaief. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys Tutorials*, 19:2322–2358, 2019.

[31] M. Masdari, S. Jabbehdari, and M. Ahmadi. A survey and taxonomy of distributed certificate authorities in mobile ad hoc networks. *Journal Wireless Communications Network 2011*, 112, 2011. doi: https://doi.org/10.1186/1687-1499-2011-112.

[32] V. Miller. Use of elliptic curves in cryptography. *Springer Conference on the Theory and Application of Cryptographic Techniques*, pages 417–426, 2000.

[33] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[34] Matthias Noback. *The Acyclic Dependencies Principle*, pages 185–216. Apress, Berkeley, CA, 2018. ISBN 978-1-4842-4119-6. doi: 10.1007/978-1-4842-4119-6_9. URL https://doi.org/10.1007/978-1-4842-4119-6_9.

[35] M. Patel, C. Chan, N. Sprecher, S. Abeta, and N. Neal. Mobile-edge computing introductory technical white paper. *White Paper, Mobile-edge Computing (MEC) industry initiative*, 2014.

[36] Hyperledger Performance and Scale Working Group. White paper: Hyperledger blockchain performance metrics. 2018.

[37] Vidhya Ramani, Tanesh Kumar, An Bracken, Madhusanka Liyanage, and Mika Ylianttila. Secure and efficient data accessibility in blockchain based healthcare systems. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 206–212, 2018. doi: 10.1109/GLOCOM.2018.8647221.

[38] Danda B. Rawat, Md. Salik Parwez, and Abdullah Alshammari. Edge computing enabled resilient wireless network virtualization for internet of things. In *2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC)*, pages 155–162, 2017. doi: 10.1109/CIC.2017.00030.

[39] A. Refaey, K. Hammad, S. Magierowski, and E. Hossain. A blockchain policy and charging control framework for roaming in cellular networks. *IEEE Network*, 34(3):170–177, 2020.

[40] J. Ricci, I. Baggili, and F. Breitinger. Blockchain-based distributed cloud storage digital forensics: Where's the beef? *IEEE Security Privacy*, 17(1):34–42, 2019.

[41] A. V. Rivera, A. Refaey, and E. Hossain. A blockchain framework for secure task sharing in multi-access edge computing. *IEEE Network*, pages 1–8, 2020. doi: 10.1109/MNET.011.2000497.

[42] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8:14–23, 2009.

[43] S. Sharma, L. Ahuja, and D. Goyal. Building secure infrastructure for cloud computing using blockchain. In *International Conference on Intelligent Computing and Control Systems*, 2018.

[44] Paolo Tasca and Claudio J. Tessone. A taxonomy of blockchain technologies: Principles of identification and classification. *Ledger*, 4, Feb. 2019. doi: 10.5195/ledger.2019.140. URL https://ledger.pitt.edu/ojs/ledger/article/view/140.

[45] P. Thakkar, S. Nathan, and B. Viswanathan. Performance benchmarking and optimizing hyperledger fabric blockchain platform. *arXiv:1805.11390v1*, 2018.

[46] T. Tran, A. Hajisami, P. Pandey, and D. Pompili. Collaborative mobile edge computing in 5g networks: New paradigms, scnearios and challenges. *IEEE Communications Magazine*, 55(4):54–67, 2017.

[47] Angelo Vera, Ahmed Refaey, and Ekram Hossain. Paper presented at IEEE GLOBECOM Global Communications Conference, Edge Computing Workshop, 2021.

[48] Angelo Vera, Ahmed Refaey, and Ekram Hossain. Blockchain-based collaborative task offloading in mec: A hyperledger fabric framework. Paper presented at the IEEE ICC International Conference on Communications, 6G Workshop, 2021.

[49] L. Xiao, Y. Ding, D. Jiang, J. Huang, D. Wang, J. Li, and H. Vincent. A reinforcement learning and blockchain-based trust mechanism for edge networks. *IEEE Transactions on Communications*, 68(9):5460 – 5470, 2020.

[50] C. Xu, K. Wang, and M. Guo. Intelligent resource management in blockchain-based cloud datacenters. *IEEE Cloud Computing*, 4(6):50–59, 2017.

[51] Xiaohong Zhang and Xiaofeng Chen. Data security sharing and storage based on a consortium blockchain in a vehicular ad-hoc network. *IEEE Access*, 7: 58241–58254, 2019. doi: 10.1109/ACCESS.2018.2890736.

[52] L. Zhou, G. Wang, and X. Xing. Cssp: The consortium blockchain model for improving the trustworthiness of network software services. In *2017 IEEE International Conference on Ubiquitous Computing and Communications IUCC*, pages 101–107, 2017.

[53] Guy Zyskind, Djabeur Zekrifa, Pentland Alex, and Oz Nathan. Decentralizing privacy: Using blockchain to protect personal data. pages 180–184, 05 2015. doi: 10.1109/SPW.2015.27.