PAVOL JOZEF ŠAFÁRIK UNIVERSITY IN KOŠICE

FACULTY OF SCIENCE

BLOCKCHAIN-BASED ENTERPRISE APPLICATIONS

MASTER THESIS

Field of Study: **Informatics**

Institute: **Institute of Computer Science**

Supervisor: **doc. RNDr. Jozef Jirásek, PhD.**

Košice 2022

Bc. Matúš Revický

Acknowledgments

I would like to express my gratitude towards doc. RNDr. Jozef Jirásek, PhD., who offered valuable insights and guidance on the way forward at various points during this research. His insights, patience and guidance have been second-to-none. I would also like to thank him for constructive feedback, motivating advice and valuable supervision.

Univerzita P. J. Šafárika v Košiciach
Prírodovedecká fakulta

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Matúš Revický
Študijný program:	informatika (jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	Informatika
Typ záverečnej práce:	Diplomová práca
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Blockchain-based enterprise applications.

Názov SK: Podnikové aplikácie s využitím technológie blokových reťazcov.

Cieľ: Compare capabilities of existing enterprise blockchain frameworks.
Design of blockchain-based enterprise application for real-world use case.
Implement and evaluate a proof of concept solution.
Evaluate performance of developed solution.

Literatúra:
1. R. Wattenhofer: Blockchain Science: Distributed Ledger Technology, Independently published, 2019, ISBN-13: 978-1793471734.
2. I. Bashir: Mastering Blockchain: A deep dive into distributed ledgers, consensus protocols, smart contracts, DApps, cryptocurrencies, Ethereum, and more, Packt Publishing, 2020, ISBN-13: 978-1839213199.
3. N. R. Thota: Mastering Hyperledger Fabric: Master The Art of Hyperledger Fabric on Kubernetes, Independently published, 2020, ISBN-13: 979-8650379720.
4. M. A. Khan, K. Salah: IoT security: Review, blockchain solutions, and open challenges, Future Generation Computer Systems 82 (2018) 395–411.
5. H. H. Pajooh, M. Rashid, F. Alam, S. Demidenko: Hyperledger Fabric Blockchain for Securing the Edge Internet of Things, Sensors 2021, 21, 359. https://doi.org/10.3390/s21020359

Vedúci:	doc. RNDr. Jozef Jirásek, PhD.
Oponent:	RNDr. Rastislav Krivoš-Belluš, PhD.
Ústav :	ÚINF - Ústav informatiky
Riaditeľ ústavu:	doc. RNDr. Ondrej Krídlo, PhD.

Dátum schválenia: 29.04.2022

doc. RNDr. Ondrej Krídlo, PhD.
riaditeľ Ústavu informatiky

Abstrakt

Problémom v oblasti distribúcie tepelnej energie je zabezpečenie spoľahlivého a pravidelného odpočtu. Pri rádiovom zbere dát o spotrebe pomocou senzorov je potrebné zabezpečiť auditovateľnosť, bezpečnosť a nemennosť dát. Ako jedna z možností riešenia sa ukázal sofistikovaný monitorovací systém s využitím technológie blockchainu. V práci sa realizuje analýza aplikovateľnosti blockchain frameworkov na monitorovanie spotreby. Výsledkom tejto práce je moderný, progresívny informačný systém založený na blockchain frameworku Hyperledger Fabric v2.3 v súlade s medzinárodnými technickými normami. Zameriava sa na bezpečnosť, nemennosť, transparentnosť dát a integráciu technológie blockchain do komplexných biznisových aplikácií v energetickom sektore. Súčasťou práce sú aj merania a vyhodnotenie výkonu implementovaných smart kontraktov pomocou nástroja Hyperledger Caliper. Experimenty sú zamerané na rýchlosť generovania a posielania požiadaviek v závislosti od počtu lokálnych Caliper klientov a na minimalizáciu neúspešných transakcií pri zachovaní čo najvyššej priepustnosti distribuovanej blockchainovej siete.

Kľúčové slová: *blockchain, Hyperledger Fabric, biznis riešenia*

Abstract

In the energy distribution segment, ensuring safe and regular heat meter readings remains a challenge. Auditability, security and data consistency must be ensured when collecting consumption data by smart energy meters. A sophisticated monitoring system using blockchain technology proved to be one of the possible solutions. First part of research investigates the applicability of blockchain frameworks to build energy consumption monitoring system. This thesis proposes a modern, progressive information system based on the blockchain framework Hyperledger Fabric v2.3 in accordance with international technical standards. It focuses on security, consistency, data transparency and the integration of blockchain technology into complex business applications in the energy sector. The performance of implemented smart contracts is measured using the Hyperledger Caliper tool. The experiments are focused on the speed of generating and sending requests depending on the number of local Caliper clients and on minimizing failed transactions while maintaining the highest possible throughput of the distributed blockchain network.

Keywords: *blockchain, Hyperledger Fabric, enterprise solution*

Contents

List of Figures

List of Tables

Abbreviations

ABAC attribute-based access control

ACL Access Control List

AML anti–money laundering

AMQP Advanced Message Queuing Protocol

API application programming interface

B2B business-to-business

BaaS Blockchain as a Service

BFT Byzantine Fault Tolerance

CA certificate authority

CF Crash faults

CFT Crash Fault Tolerant

CPU Central processing unit

CRUD Create, Read, Update, Delete

DApps Decentralized applications

DHT Distributed Hash Table

DLT Distributed ledger technology

E-CA enrolment certificate authority

E-Certs Enrolment certificates

E-O-V Execute-Order-Validate

ESCC Endorsement System Chaincode

EVM Ethereum Virtual Machine

FBA Federated Byzantine Agreement

HIPAA Health Insurance Portability and Accountability Act

HTML HyperText Markup Language

HTTPS Hypertext Transfer Protocol Secure

IoT Internet of Things

JSON JavaScript Object Notation

JVM Java Virtual Machine

KYC know-your–customer

MSP membership service provider

MVCC Multiversion Concurrency Control

NIST National Institute of Standards and Technology

OS Operating system

P2P peer-to-peer

PBFT Practical Byzantine Fault Tolerance

PC Personal Computer

PKI Public Key Infrastructure

PoA Proof of Authority

PoC Proof of Concept

PoS Proof of Stake

PoW Proof of Work

RAM Random-access memory

RBAC Role-Based Access Control

REST Representational state transfer

RSM replicated state machine

SaaS Software as a Service

SDK Software Development Kit

SMR State machine replication

SoC speed of consensus

SSD Solid-state drive

SSO Single sign-on

SUT System Under Test

T-Certs temporary certificates

TA trust authority

TLS Transport Layer Security

TPS transactions per second

TRXN Transaction

VS Code Visual Studio Code

VSCC Validation System Chaincode

WSL2 Windows Subsystem for Linux 2

Introduction

Sophisticated consumption monitoring is a challenge for the energy sector. Due to lack of reliable consumption data companies may need to over-provision generation thus increasing the cost of energy. The issues related to the security and privacy of consumption data present serious challenges.

Consumers, property management companies, heating distributors and heating companies together form a business network due to their transactive relationships with one another. Such a business network must be decentralized in order to prevent decision-making from becoming concentrated in the hands of a single party.

To facilitate transactions among the members, a system that encapsulates this business network is required. To be trustworthy, such a transaction system must be transparent, immutable, and allow for provenance tracing of assets transacted on the network. Business relationships must be encapsulated in mutually agreed upon self enforcing contracts in order to automate and facilitate transactions.

Blockchain is a decentralized immutable ledger that meets these requirements since it forbids any member from unilaterally processing transactions or making decisions on the network. Transactions are kept in blocks, with each block containing the hash of the previous one, making the ledger verifiable. Any asset's provenance may be easily traced by checking its transaction history on the ledger. Transactions are transparent to all members since the ledger is replicated at each node of the network. The ledger is also immutable due to its distributed nature and the cryptographic linking of blocks, as any unilateral attempts to modify the transactions will produce inconsistency and hence be rejected by the network. For digital assets that can be replicated, double spending is an issue, as unscrupulous parties may attempt to spend the same asset numerous times. The use of a decentralized and immutable ledger that requires consensus for each transaction mitigates this problem.

Permissioned networks, such as Hyperledger Fabric, allow only authenticated parties to join and participate. Identity-based access control mechanisms, as well as

traceability, can be used to define each node's privileges in the network, as well as to establish accountability because the invoker for each transaction is known. To automate transactions, self-enforcing smart contracts can be written to encapsulate agreed-upon business logic. Because network participants are identified and authenticated, resource-intensive approaches like Proof of Work (PoW) are no longer required. The system's operating costs are thereby decreased, and the need to deploy cryptocurrency to motivate nodes to process transactions is eliminated.

This thesis's main topic is a study of the applicability of enterprise permissioned blockchains, to the energy sector and, more specifically, to energy consumption monitoring. We analyze the following research question: Is blockchain applicable to address transaction management for consumption monitoring?

In this thesis, we use Hyperledger Fabric 2.3, a distributed ledger technology framework maintained by the Linux Foundation, to design and develop our solution.

Following research activities are related to performance evaluation: (1) Selection of tool to measure the performance of smart contracts, (2) determination of the highest possible actual send rate of selected tool (Caliper), (3) estimation of the transaction throughput of the network while trying to minimize number of failed transactions, (4) monitoring of hardware resources utilized during experiments.

This thesis is structured as follows:

Chapter 1 provides a comprehensive overview of blockchain technology and important theoretical concepts. This chapter concludes with related work, that provides an insight into the current state of the art in the applicability of blockchain to achieve trust in energy sector.

Chapter 2 explains the use and application of blockchain technology in enterprise settings, covers comparison of public and enterprise blockchains, and describes distributed ledger platforms such as Quorum, Corda and Hyperledger Fabric together with comparison.

Chapter 3 focuses on the architecture and concepts behind the Hyperledger Fabric network.

Chapter 4 presents use-case focused on energy consumption monitoring. Analysis of use case is performed which consists of a participants identification, shared assets identification, lifecycle identification and identification of capabilities and restrictions of each role.

Chapter 5 describes the components forming our solution, functional considerations and additional Hyperledger Fabric tools.

Chapter 6 describes most important parts of implementation process and dives into the technical details relevant to implementing the system.

Chapter 7 presents flow validation, performance and security analysis.

Full implementation source code and additional measurements are available in appendix A. The implemented solution consists of following components: (1) Hyperledger Fabric network (Appendix A, `diploma_thesis_project/fablo-network-generator/`) with chaincodes (Appendix A, `diploma_thesis_project/chaincode-typescript/`) and (2) Applications for organizations (Appendix A, `diploma_thesis_project/apps/`). Appendix A, `diploma_thesis_project/README.md` contains information about additional `diploma_thesis_project/README-*.md` files, which describe how the whole system was created and how to run it.

Additional project was created to measure performance (Appendix A, `caliperworkspace/`), which requires a running Fabric network and identities generated by applications for organizations. Caliper requires the definition of network configurations (Appendix A, `caliperworkspace/networks/networkConfig.json`) to connect to the Fabric network, test configurations (Appendix A, `caliperworkspace/benchmarks`), and test files (A, `caliperworkspace/workload/`) to test performance of chaincode functions. Logs from each benchmark, which contain error messages, can be found in appendix A, `caliperworkspace/reports` in corresponding `*.log` files. Individual reports can be found in appendix A, `caliperworkspace/reports` in corresponding `*.html` files. Resource utilization summary can be found in appendix A, `caliperworkspace/reports/docker_report.xlsx`.

1 Blockchain

The first documented blockchain design was in 2008, and the first open source blockchain implementation was deployed in 2009 as an important component of Bitcoin, the first decentralized digital currency system to distribute bitcoins via the open source release of the Bitcoin peer-to-peer (P2P) software [46].

1.1 The history of blockchain and Bitcoin

Now, we'll look at the early history of computers and computer networks, and how these technologies evolved and contributed to the creation of Bitcoin in 2008. This can be viewed in chronological order [5]:

- 1970s – Cryptographic hash functions.
- 1978 – Invention of public key cryptography.
- 1979 – Invention of Merkle Trees (hashes in a tree form) by Ralph C. Merkle.
- 1982 – The Byzantine Generals Problem (While Bitcoin can be considered a solution to the Byzantine Generals Problem, the Bitcoin network's primary aim was to resolve the previously unsolvable double-spending problem.)
- 2001 – Emergence of BitTorrent and Distributed Hash Tables (DHT).
- 2005 – Prevention of Sybil attacks by using computation puzzles, due to James Aspnes et al.
- 2009 – Bitcoin (first blockchain)

Even if the aforementioned technologies did not directly contribute to the development of Bitcoin, their work is significant to the problem that Bitcoin solved. All prior attempts to build anonymous and decentralized digital currency were partially successful, but they failed to address the issue of avoiding double spending in a truly

trustless or permissionless environment. The Bitcoin blockchain, which launched the Bitcoin cryptocurrency, was eventually able to solve this difficulty [5].

Other concepts, such as State machine replication (SMR), introduced by Leslie Lamport in 1978 and formalized by Fred Schneider in 1980, are also addressable by Bitcoin. Bitcoin (probabilistically) solves the SMR problem by permitting block replication and assuring consistency through its PoW consensus method [5].

1.2 Overview of Blockchain technology

There are several slightly different definitions of blockchain and some very dogmatic opinions. We used the following definition for this thesis [45].

Blockchain is a database encompassing a physical chain of fixed-length blocks containing 1 to N transactions, where each transaction added to a new block is validated and then inserted into the block. When the block is completed, it is appended to the end of the existing chain of blocks. Furthermore, as opposed to traditional CRUD, the only two operations are add transaction and view transaction. As a result, the fundamental blockchain processing consists of the steps listed below [3].

1. Add new and undeletable transactions and organize them into blocks.
2. Cryptographically verify each transaction in the block.
3. Append the new block to the end of the existing immutable blockchain.

Each node keeps a replica of an immutable ledger, which is typically implemented as an append-only file or database. To commit records, blockchain relies on digital signatures and consensus. Distributed ledger technology (DLT) does not employ consensus; instead, DLTs rely on the presence of digital signatures to commit records. The records are committed to the immutable ledger, an append-only database in which each record is ordered in time and each block of records is cryptographically linked to the previous committed block. The block contains additional metadata along with the hash code of the prior block and a set of committed records. The records are typically represented by a hash tree known as a Merkle tree, which is depicted in figure 1.1.

The topmost node of the tree is the root. The nodes at the bottom are known as leaf nodes. Each node is simply a cryptographic hash of a transaction. The Merkle tree is not a list of all transactions, but rather a hash of all transactions in the form

of a tree structure. The most significant advantage is that determining whether a certain transaction has been included within a block is easy and efficient [3].

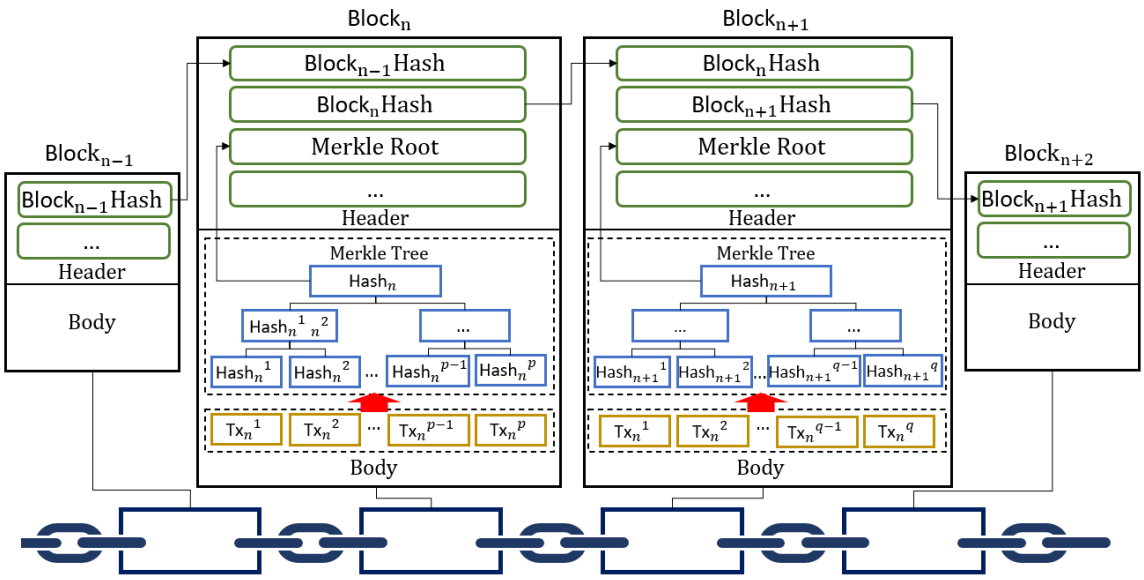

Figure 1.1: Blocks with Merkle tree

1.3 Consensus

The choice of the consensus algorithm to utilize is governed by the type of blockchain in use; that is, not all consensus mechanisms are appropriate for all types of blockchains.

Consensus is the process of achieving agreement on the final state of data between distrusting nodes. Different algorithms are employed to reach a consensus. It is **easy** to obtain **an agreement between two nodes** (e.g., in client-server systems). But when several nodes are participating in a **distributed system** and they need to agree on a single value, it becomes a difficult task to achieve consensus. This process of reaching agreement on a common state or value among several nodes despite the failure of some nodes is known as **distributed consensus**.

All consensus algorithms are designed to deal with faults in a **distributed** system and to allow **distributed** systems to reach a final state of agreement. Replication is used to achieve fault tolerance. This is a commonly used method to achieve fault tolerance. In general, there are two types of faults that a node can encounter [5]:

1. **Crash faults (CF)**: This type of fault occurs when a node merely has crashed. CF are the easier ones to deal with of the two fault types. To cope with this sort of failure, Paxos or the Raft protocol are typically utilized.

2. **Byzantine faults**: The second type of fault is one where the malfunctioning node acts maliciously or inconsistently at random. This type is difficult to handle since it can create confusion due to misleading information. This could be the consequence of an adversary attack, a software error, or data corruption. SMR protocols such as **Practical Byzantine Fault Tolerance (PBFT)** was developed to address this type of faults.

1.3.1 Consensus in blockchain

The two main categories of consensus mechanisms are **roughly described** as follows:

1. **Proof-based, leader-election lottery-based, or the Nakamoto consensus** whereby a leader is elected at random (through an algorithm) and proposes a final value. This category of consensus mechanism is also known as totally decentralized or permissionless. This type is employed in the Bitcoin and Ethereum blockchain in the form of a PoW mechanism.
2. **Byzantine Fault Tolerance (BFT)-based** is a more traditional method based on rounds of votes. This category of consensus is also known as the consortium or permissioned type of consensus mechanism.

When there is a small number of nodes, BFT-based consensus techniques perform well, but they do not scale very well. Leader-election lottery-based (PoW) consensus mechanisms, on the other hand, scale very well but perform very slowly. There are also some additional proposals out there that are attempting to achieve the balance between scalability and performance (e.g., PBFT) [5].

1.3.2 Blockchain Consensus Algorithms

In this section, we explore consensus algorithms that are widely used in existing blockchain solutions. The following is not an exhaustive list [5]:

- **Proof of Work (PoW)**. This type of consensus mechanism relies on proof that appropriate computational resources have been spent before proposing a value for acceptance by the network. PoW involves solving a resource-intensive, difficult-to-solve, yet simple-to-verify cryptographic challenge. The miners need to find a nonce value in a way that the hash of the block content and the nonce has a specific amount of leading zeros. PoW uses a difficulty parameter that

adjusts the amount of leading zeros to limit the number of blocks created in the network to one every 10 minutes. PoW solves the Byzantine generals problem as long as honest participants control the majority of the network.

- **Proof of Stake (PoS)**. To mine blocks using this algorithm, miners must lock certain assets. The mining power of each node is determined by the number of locked assets. The miner who has the most locked assets has a greater weight in mining blocks. PoS is based on the assumption that the nodes that profit the most from the blockchain, i.e., those with the greatest assets invested in it, are less inclined to attack the blockchain (i.e., avoid self-harm). Big businesses, such as Google, can acquire a large share of assets in Internet of Things (IoT), and so PoS may eventually lead to centralization.
- **Proof of Authority (PoA)** is a consensus algorithm that can be thought of as a type of PoS in which each miner's mining power is determined based on its identity in the network rather than the quantity of locked assets. A pre-approved group of nodes act as miner and their identity is known to all network members. The miner with higher reputation have higher chance in mining new blocks.
- **Practical Byzantine Fault Tolerance (PBFT)**: This mechanism achieves SMR, which provides tolerance against Byzantine nodes. Other protocols, including as PBFT, Paxos, Raft, and Federated Byzantine Agreement (FBA), are also utilized or have been suggested for use in a variety of distributed system and blockchain implementations. PBFT provides immediate and deterministic transaction finality. In contrast, the PoW protocol requires a number of confirmations to finalize a transaction with a high probability. PBFT is also more energy efficient than PoW, which uses a lot of electricity.

  PBFT is not very scalable. That's why it is better suited to consortium networks rather than public blockchains. It is considerably faster than PoW protocols. Sybil attacks can be carried out on a PBFT network, where a single entity can control multiple identities in order to influence voting and, subsequently, decision. However, the fix is straightforward, and in fact, this is not particularly practical in consortium networks when all identities are known on the network. This issue can easily be solved by increasing the number of network nodes.

1.4 Types of Blockchain

Based on how blockchain has evolved over the last few years, it can be divided into multiple categories with different, although somewhat overlapping characteristics. In this section, we will examine the different types of blockchains from a technical and business use perspective [5].

1.4.1 Public Blockchains

A public blockchain is one that its creators intended for **everyone** to be able to access and transact with; a blockchain where transactions are included if and only if they are valid; a blockchain where **everyone** can contribute to the consensus process. As previously stated, the consensus mechanism determines which blocks get added to the chain and what the current state is. Instead of relying on a central server, the public blockchain is secured by cryptographic verification, which is supported by incentives for miners. Anyone can be a miner to aggregate and publish those transactions. In the public blockchain, because no user is implicitly trusted to verify transactions, all users follow an algorithm that verifies transactions by committing software and hardware resources to solving a problem by brute force (i.e., by solving the cryptographic puzzle). The miner who finds the solution first gets rewarded, and each new solution, together with the transactions necessary to verify it, serves as the foundation for the next challenge to be solved. The verification concepts are PoW or PoS [3].

1.4.2 Consortium Blockchains

A consortium blockchain, such as R3 [41], is a distributed ledger in which the consensus process is **controlled by a predefined set of nodes**, such as a consortium of nine financial institutions, each of which operates a node, and of which five (such as the US Supreme Court) must sign every block in order for the block to be valid. The right to read the blockchain can be open to the public or restricted to participants, and there are also hybrid options, such as making the root hashes of the blocks public while also providing an application programming interface (API) that allows members of the public to make a limited number of queries and receive cryptographic proofs of some parts of the blockchain state. These blockchains are distributed ledgers that may be described as "partially decentralized" [3].

1.4.3 Private Blockchains

A fully private blockchain is one in which write permissions are centralized to a **single organization**. Read permissions can be public or restricted to any level. Likely applications include database management and auditing internal to a single company, so public readability may not be necessary in many cases at all, though in other cases public auditability is desired. Private blockchains might give answers to financial enterprise problems, such as compliance agents for regulations like the Health Insurance Portability and Accountability Act (HIPAA), anti–money laundering (AML), and know-your–customer (KYC) laws. The Linux Foundation's Hyperledger project and the Gem Health network are both private blockchain projects under development [3].

1.4.4 Summary

Although they differ in terms of security and performance in terms of the speed of consensus (SoC), if any trust authority (TA) is used, and how many TAs are necessary, as detailed in table 1.1, these three blockchain categories have several common properties [46]:

(1) they all employ decentralized peer-to-peer networks for transactions,

(2) they all demand that each transaction be digitally signed and appended only to the blockchain,

(3) they all rely on consensus to synchronize the replicas across the network.

Types	**Description**	**# Trust authorities**	**Speed of consensus**	**Scenarios**
Public	Anyone can participate and is accessible worldwide	0	Slow	Global decentralized scenarios
Consortium	Controlled by pre-selected nodes within the consortium	>=1	Normal	Businesses among selected organizations
Private	Write rights are controlled by an organization	1	Fast	Information sharing and management in an organization

Table 1.1: Types of blochchain [46]

1.5 Related Work

The use of blockchain in the energy sector has attracted the interest of academic researchers, utility corporations, and energy decision makers. This section briefly

discusses related work.

Kieron O'Hara [36] demonstrates the essential functions of data trust. He identifies eight technical criteria that are required for trustworthy data sharing: discovery, provenance, access controls, access, identity management, usage auditing, accountability, and impact. His work is focused on theoretical properties of data trust and is not blockchain specific. Blockchain and its innovative features supply several of these necessary properties, such as provenance and auditing. Access control restrictions and data impact calculations, for example, are two more desirable features for data trust that can be written into blockchain as smart contracts.

Gür et al. [13] propose a scalable and energy efficient energy tracking system with blockchain and IoT devices. In their solution Raspberry Pi is used to simulate metering, Hyperledger Fabric 1.3 is selected as a blockchain system. Authors explain the need for a system where all users have access to the current usage levels, all users trust the system and the privacy of their personal data is preserved. Their work briefly points out benefits of smart sensors in consumption metering. Their Proof of Concept (PoC) solution is implemented using Hyperledger Composer (deprecated on Aug 29th 2019), which dramatically simplified blockchain application development. Their solution was tested on single computer inside Virtualbox. No performance analysis was performed.

Yue Qi et al. [39] examines the use of blockchain technology to build an energy consumption monitoring system for key energy-consuming units, analyzes the fit between business needs and blockchain technology, designs high-level blockchain platform architecture, and discusses the issues of energy consumption monitoring, blockchain terminal equipment authentication and collection of trusted data. Research focuses mostly on comparison with the existing energy consumption monitoring system for energy-consuming units in China. Proposed solution is not implemented and it is not known which technologies should be used.

Claudia Pop et al. [38] are dealing with real-time energy data collected by smart energy meters. They propose a scalable second tier solution which combines the blockchain ledger with distributed queuing systems and NoSQL databases to allow for less frequent registration of energy transactions on the chain while retaining the tamper-evident features of blockchain technology. Authors also propose a solution for tamper-evident registration of smartmeter' energy data and related energy transactions using digital fingerprinting that allows the energy transaction to be linked hashed-back on-chain while the sensor data is stored off-chain. For the on-chain components, authors

used Ethereum and smart contracts, while for the off-chain components, authors used Cassandra database and RabbitMQ messaging broker.

Esther Mengelkamp et al. [33] propose a blockchain-based microgrid energy market without the need for central intermediaries. Common blockchain use in the energy industry is to transform the power grid into a peer-to-peer network allowing prosumers to trade electricity with one another (e.g., purchase and/or sell energy generating excess to neighbors). In general, smart meters are used to identify energy surplus, which is then transformed into equivalent energy tokens that may be swapped in a local grid-level marketplace. Proposed solution was implemented using Ethereum and project is available at [34].

Based on the academic articles reviewed, we can conclude that trustworthy energy consumption monitoring systems are possible to implement and would provide clear added value. The key benefits of these implementations include traceability, security, and auditability.

2 Enterprise Blockchain

In this chapter, we will look into enterprise blockchains. We will also attempt to answer the major issue of why existing public blockchains are not always a suitable candidate for enterprise use cases. We will also present a number of enterprise blockchain platforms. We'll cover the following topics along the way:

- enterprise solutions and blockchain,
- limiting factors,
- requirements,
- enterprise blockchain versus public blockchain,
- blockchain in the cloud,
- currently available enterprise blockchains.

Enterprise blockchains are a form of **permissioned consortium chain** that primarily addresses enterprise needs. Enterprise blockchains are typically **private** and **permissioned**, and are run among consortium members. While public blockchains offer integrity, consistency, immutability, and security, they lack certain features, making them less appropriate for enterprise use [5].

2.1 Enterprise solutions and blockchain

Enterprise solutions integrate several aspects of a business and enable it to meet its goal by providing stakeholders with business-critical information. Therefore, an enterprise blockchain system should be capable of achieving this result.

We can ask certain questions to determine whether or not a blockchain is suitable for an enterprise. The following are a few of these questions:

- Does my use case require data to be shared among participants?

- Does my use case have people from other organizations that aren't always trustworthy and have conflicting interests?
- Does my use case require strict auditing? A blockchain can help with this since it is an immutable, tamper-proof chain of records that can offer a clear audit trail of all transactions made on the chain (enterprise system).
- Does my use case need to keep transactions confidential?
- Is it necessary for my use case to maintain participant anonymity?
- Does my use case require restricted but transparent ledger updates?
- No need for a single trusted authority in my use case; instead, network decisions (ledger updates) should be made by consortium members and agreed upon by them?

In addition to the questions outlined here, we need to additionally address the following questions when introducing an enterprise blockchain solution:

- What is the proposed blockchain solution's overall goal? Is it in accordance with the enterprise's business goals? Is it a Proof of Concept (PoC) solution to just illustrate a concept, or a production project with real-world commercial deliverables?
- Where would it be hosted – in the cloud or on-premises? Who will be in charge of the blockchain solution once it is deployed?
- Risk management concerns – does the solution follow any known risk management guidelines? National Institute of Standards and Technology (NIST) 800-37, for example, is a risk management framework that provides a strategy for managing security and privacy risk in information technology systems and organizations.
- Is there any other enterprise blockchain project that this organization has already implemented? Can we learn from previous experiences and apply some of the tools and best practices that have already been developed?

The challenge now is how to design blockchain solutions that provide actual business value.

The major advantage of enterprise blockchains is in its ability to serve as a sharable, replicable, permissioned ledger amongst organizations, resulting in immediate cost

savings by removing the requirement for data exchange. In doing so, we also eliminate the requirement for infrastructure and tools to facilitate such exchanges. Furthermore, because a blockchain provides inherent security, immutability, and auditing, there is no need to invest individually for these requirements [5].

2.2 Limiting factors in public blockchain

These **limitations in public blockchains**, as well as special requirements in any organization, have sparked interest in enterprise blockchain. Next, we'll go through some of the most typical problems.

- **Slow performance**. Public blockchains are slow, with just a few transactions per second being processed. Bitcoin is capable of processing 3-4 transactions per second.
- **Lack of access governance**. All participants in an enterprise must be identified so that everyone knows who they are dealing with. Because public blockchains lack an identification and access control system, they are inappropriate for business use.
- **Lack of privacy**. Public blockchains are inherently transparent, with everyone able to see everything on the ledger. This implies that anyone can access transaction data and participants involved in a transaction.
- **Probabilistic consensus**. Public blockchains often employ a PoW consensus algorithm, which is inherently a probabilistic protocol with probabilistic finality. Enterprise blockchains solve this limitation by employing deterministic consensus algorithms that guarantee immediate finality.
- **Transaction fees**. For each transaction executed on Ethereum or other similar blockchain, a transaction fee is charged in the native cryptocurrency. While this technique offers incentives for miners and guards against spam, it is not required in enterprise blockchains.

The issues raised here are considered limitations in public blockchains, but they can also be viewed as requirements in enterprise blockchains [5].

2.3 Enterprise blockchain requirements

In certain circumstances, enterprise blockchains have stricter requirements than public blockchains. In public blockchains, for example, eventual consistency is permitted. However, in enterprise blockchains, once a transaction is committed, it should immediately finalize and irrevocably become part of the global record (state).

Now, we'll go through three key requirements that a blockchain must meet in order to be suitable for enterprise use. These requirements are for privacy, performance, and access governance.

- **Privacy**. In enterprise blockchains, privacy is of the utmost significance. Privacy has two aspects: confidentiality and anonymity.
- **Performance**. Enterprise blockchains must be able to execute transactions at a fast rate due to the high-speed requirements of businesses.
- **Access governance**. From another perspective, because enterprise blockchains are permissioned, enterprise-grade access control (in the form of either a new mechanism on the chain or control driven by an enterprise Single sign-on (SSO) already in place) is a fundamental requirement. Since all members on a consortium chain must be identified, it is critical to design an access control system that enables that process. This capability is achievable through the use of enterprise-grade access control techniques such as Role-Based Access Control (RBAC).

We'll go through several additional requirements that can improve the suitability/efficacy of corporate blockchain systems [5]:

- **Ease of use**. Enterprise system deployment is a well-studied, known, and mature field. Enterprise deployment has gotten simple over the years thanks to enterprise orchestration technologies and established techniques. Blockchains, on the other hand, are not as simple to deploy as other enterprise systems. This is changing with the adoption of Blockchain as a Service (BaaS) and deployment automation tools. There is still some work to be done.
- **Better tools**. Typically, various supporting tools and utilities are supplied with the main product to operate the software in enterprise systems. Administration tools, deployment tools, developer utilities, visualization tools, management tools, and end user tools are examples of these. Blockchain platforms with

better tooling are considerably more desirable because of better user support. Tools such as block explorers, user administration modules, and Decentralized applications (DApps) to manage smart contracts are quite beneficial. They are gradually becoming more mature as the blockchain ecosystem is growing.

2.4 Enterprise blockchain versus public blockchain

This section will present a comparison between public and enterprise blockchains. Consider the table 2.1, which compares various aspects of the two blockchain types [5].

Aspect	**Public chains**	**Enterprise chains**
Confidentiality	No	Yes
Anonymity	No	Yes
Membership	Permissionless	Permissioned via voting, KYC, usually under an enterprise blockchain.
Identity	Anonymous	Known users
Consensus	PoW/PoS	BFT
Finality	Mostly probabilistic	Requires immediate/instant finality.
Transaction speed	Slower	Faster (usually, should be).
Scalability	Better	Not very scalable, mainly due to consensus choice. Usually a much smaller number of nodes compared to public chains.
Regulatory compliance	Not usually required	Required at times.
Trust	Fully decentralized	Semi-centralized and managed via consortium and voting mechanisms.
Smart contracts	Not strictly required; for example, in the Bitcoin chain	Strictly required to support arbitrary business functions.

Table 2.1: Public and enterprise blockchains comparison [5]

Evidently, enterprise blockchains have a wide range of applications, including but not limited to payments, insurance, KYC, and supply chain monitoring [5]. It is, however, important to note that these solutions lack details about implementation.

2.5 Blockchain in the cloud

Cloud computing offers numerous advantages to businesses, including increased efficiency, cost savings, scalability, high availability, and security.

Blockchain as a Service, or BaaS, is an extension of Software as a Service (SaaS) in which an organization's blockchain platform is implemented in the cloud. The organization maintains its applications on the blockchain, while the cloud provider manages the rest of the software, infrastructure, and other aspects such as security and operating systems. This means that the cloud provider provides and maintains the blockchain's software and infrastructure. The customer or enterprise can concentrate on their business applications without having to worry about infrastructure issues.

There are many BaaS providers. A few of them are listed as follows [5]:

- **AWS** [6]
- **Azure** [2]
- **Oracle** [37]
- **IBM** [27]

2.6 Currently available enterprise blockchains

As this is a very rapidly growing field for research, and the market is quite active in this domain, several enterprise blockchain solutions have been developed and made available in the previous few years. Notably, 2019 has been labeled "the year of the enterprise blockchain" due to the emergence of several startups and enterprises focusing on this area.

In this chapter, we will provide a brief description of Corda and Quorum. Hyperledger Fabric is described in more detail as our solution leverages a mix of Hyperledger technologies.

2.6.1 Corda

Corda, by definition, is not a blockchain because it does not use blocks to batch transactions. Nonetheless, it is a distributed ledger that gives all of the benefits that

a blockchain can. Traditional blockchain systems use the concept of transactions being grouped together in a block, with each block cryptographically linked back to its parent block, creating an immutable record of transactions. Corda, on the other hand, is not like this.

Corda was developed from ground up with a unique approach for giving all blockchain benefits without the need for a traditional blockchain with blocks. It was first designed for the financial industry to address issues that arose as a result of each organization maintaining its own local ledgers. This means that each organization has its own "view of truth". This situation frequently results in inconsistencies and operational risk. Furthermore, data is duplicated at each organization, resulting in increased costs and complexity in managing individual infrastructures. These are the kinds of issues in the financial industry that Corda set out to solve by developing a decentralized database platform. Corda initially focused solely on financial applications, but now has extended into other sectors such as government, healthcare, insurance, and supply chains.

Corda is available in two implementations: Corda enterprise and Corda open source. Corda enterprise and Corda open source are both interoperable and compatible, with the same functionality. The enterprise edition, on the other hand, is more focused on enterprise requirements. It is a commercial version of the Corda platform that focuses on business needs such as privacy, security, and performance. It also contains the Corda firewall, high availability nodes and notaries, and hardware security module support. It offers an enterprise-grade platform for business use.

The Corda platform, on the other hand, enables direct business-to-business (B2B) transactions. This network delivers benefits such as privacy and easy data sharing, which reduces complexity and costs.

Corda's source code can be found at [9]. It is written in Kotlin, which is a statically typed language that targets the Java Virtual Machine (JVM).

The Corda platform's core components are the Corda network, state objects (contract code and legal prose), transactions, consensus, and flows.

The Corda network is defined as a fully connected graph. It is a permissioned network that provides direct P2P communication on a "need to know" basis. There is no global broadcast or gossip mechanism, unlike other traditional blockchains/distributed ledgers. Communication takes place directly between interested parties on a point-to-point basis. The Advanced Message Queuing Protocol (AMQP) serialization protocol is used for communication between peers or nodes. A service known as

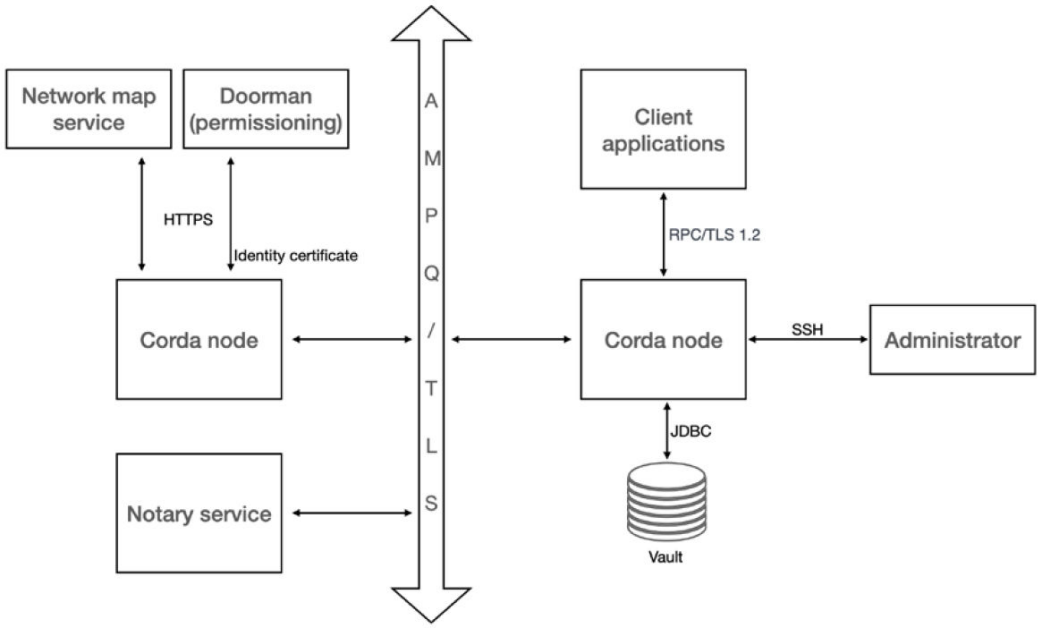

Figure 2.2: Corda high-level network architecture [5]

network map service is in charge of publishing a list of peers [5].

Figure 2.2 depicts a high-level architectural design of the Corda network.

2.6.2 Quorum

Quorum [40] is an open source enterprise blockchain platform. It is a lightweight Ethereum fork. Quorum not only benefits from the upstream public Ethereum (Geth) project's innovation and research, but it has also offered numerous great enterprise features. These enterprise features are mainly concerned with delivering enterprise-level privacy, performance, and permissioning (access control).

Quorum addresses three main concerns with public blockchains, making it an ideal alternative for corporate use cases:

- Privacy
- Performance
- Enterprise governance

At a high level, the Quorum architecture is composed of nodes and associated privacy managers. The Quorum node is a modified and improved version of public geth that enables private transactions. Quorum nodes communicate with privacy managers via Hypertext Transfer Protocol Secure (HTTPS), and privacy managers

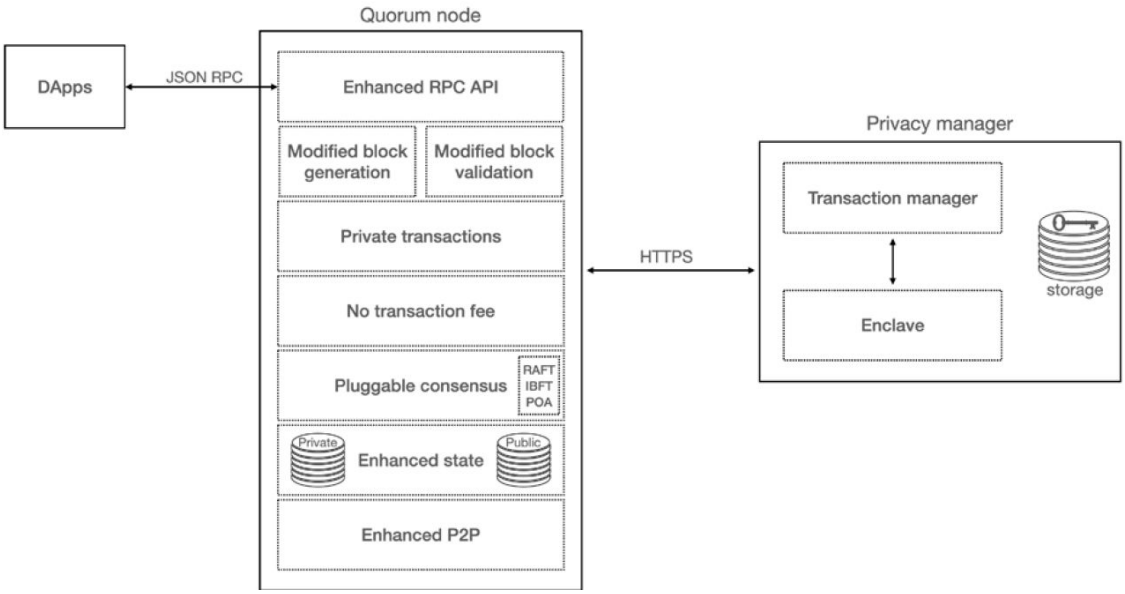

Figure 2.3: Quorum architecture [5]

are responsible for providing privacy by storing the payload in encrypted format in their local storage. These Quorum nodes separate the public and private states.

The figure 2.3 illustrates the Quorum architecture [5].

2.6.3 Hyperledger Fabric

Hyperledger Foundation has published a white paper [26] that presents a reference architecture model that can serve as a guideline to build permissioned distributed ledgers. These high-level components are shown in figure 2.4.

We have five top-level components that deliver different services. The first is identity, which offers membership services such as authorization, identification, and authentication. Then there's the policy component. Following that, we have blockchain and transactions, which include the Consensus Manager, Distributed Ledger, network P2P protocol, and Ledger Storage services. Finally, there is the smart contracts layer, which offers chaincode services in Hyperledger and hosts smart contracts using Secure Container technology.

Fabric [19] is a well-known open-source permissioned blockchain technology developed by the Linux Foundation [30]. It is the first blockchain technology to enable the generation of smart contracts in general-purpose programming languages. Fabric allows clients to submit transactions to a blockchain system that provides decentralized control of a shared, distributed state, i.e., there is no single trusted authority that decides on the system's current state. A smart contract, also known as chain-

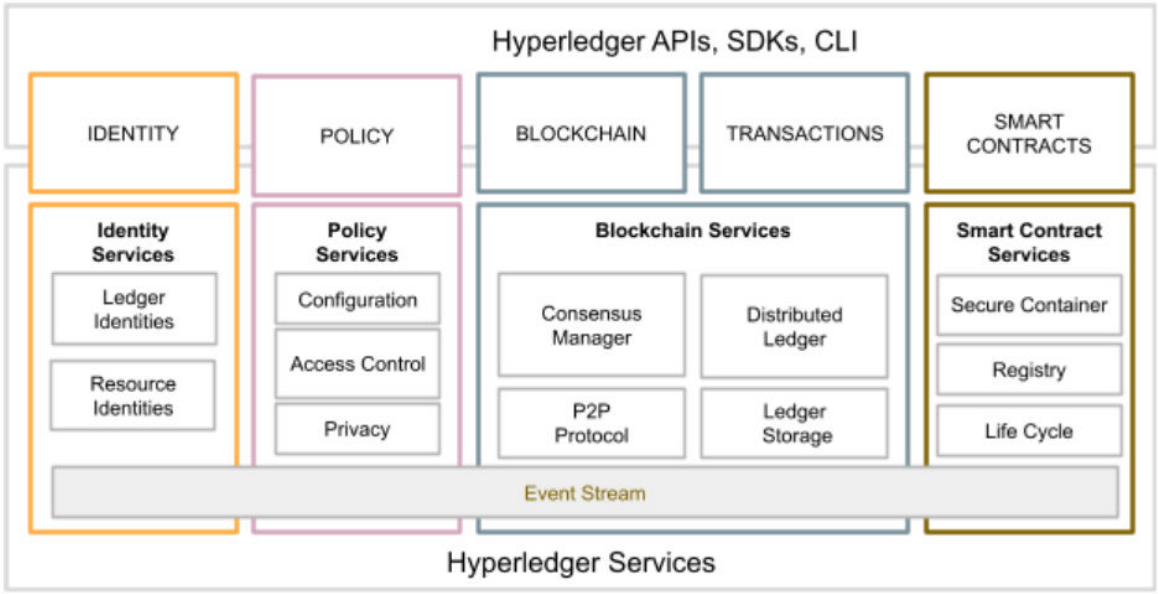

Figure 2.4: Hyperledger Fabric architecture overview [26]

code in Fabric terminology, defines all possible functions that may be invoked by a transaction.

The distributed state, known as the world state, is stored as a versioned key-value store - Hyperledger Fabric natively supports LevelDB [29], as well as CouchDB [1]. Each key has a version number that is changed with each write. The distributed ledger stores the whole history of all transactions (both successful and failed) on the network, which are organized into blocks. Both the distributed ledger and the world state are replicated on a set of distributed nodes, called peers, that are registered on the Fabric network.

Peers retrieve blocks of transactions from an ordering service, which ensures that transactions are delivered in correct order; more specifically, all peers receive all transactions in the same order. They independently validate each transaction and update their copy of the world state and the ledger accordingly. The ordering service can be replicated for fault tolerance and employs a consensus technique (e.g., Raft) to agree on the order of all transactions.

The ordering service manages numerous channels, which are private communication paths between Fabric components. Endorsers are a subset of peers who have the extra task of endorsing transactions submitted by clients, i.e., they simulate transaction execution to create read and write sets depending on the current world state. The ordering service manages numerous channels, which are private communication paths between Fabric components.

Endorsers are a subset of peers who have the extra task of endorsing transactions submitted by clients, i.e., they simulate transaction execution to create read and write sets depending on the current world state. An endorsement policy specifies the amount of endorsements necessary for a transaction to be considered legitimate. Finally, peers are organized into organizations that often correlate to real organizations. These organizations tend to play a major role in the endorsement policy [7].

2.6.4 Comparison of main platforms

In table 2.2, we will present a brief comparison of the most popular enterprise blockchain platforms. We'll go through the majority of the intriguing features of enterprise blockchains. This can be used as a quick reference for comparing these platforms. However, because this is a quickly changing area, certain features may alter over time, or new features may be implemented or improved [5].

Feature	**Quorum**	**Fabric**	**Corda**
Target industry	Cross-industry	Cross-industry	Cross-industry
Performance (approximate transactions per second)	700	560	600
Consensus mechanism	Pluggable multiple Raft, IBFT, PoA	Pluggable Raft, unofficial SmartBFT	Pluggable, notarybased
Tooling	Rich enterprise tooling	SDK	Rich enterprise tooling
Smart contract language	Solidity	Golang/Java/Javascript/ Typescript	Kotlin/Java
Finality	Immediate	Immediate	Immediate
Privacy	Yes (restricted private transactions)	Yes (restricted private transactions)	Yes (restricted private transactions)
Access control	Enterprise-grade permissioning mechanism	Membership service provides/certificate based	Doorman service/ KYC. Certificate based
Implementation language	Golang, Java	Golang	Kotlin
Node membership	Smart contract and node software managed	Via membership service provider	Node software managed using configuration files, certificate authority controlled
Member identification	Public keys/ addresses	PKI-based via membership service provider, supports organization identity	PKI-based, support organization identity

Table 2.2 continued from previous page

Feature	Quorum	Fabric	Corda
Cryptography used	SECP256K1, AES, CURVE25519 + XSALSA20 + POLY13050, PBKDF2, SCRYPT	SECP256R1	ED255519 SECP256R1 RSA – PKCS1
Smart contract runtime	EVM	Sandboxed in Docker containers	Deterministic JVM
Upgradeable smart contract	Possible with some patterns, not inherently supported	Allowed via upgrade transactions	Allowed via administrator privileges and auto update allowed under administrative checks
Tokenization support	Flexible - inherited from public Ethereum standards	Programmable	Corda token SDK

Table 2.2: Comparison of blockchain frameworks [5]

This comparison may also be used to quickly assess the feasibility of different platforms for an enterprise use case.

3 Hyperledger Fabric

We chose Hyperledger Fabric as the underlying blockchain platform. For this reason we provide a more detailed overview of Hyperledger Fabric. We have covered the fundamentals of Hyperledger Fabric in the section 2.6.3. The following factors were used to make the decision: functional extent, support, cost, smart contract language, and interfaces.

3.1 Transaction flow

Fabric's transaction flow is divided into three stages: execution, ordering, and validation. This is known as the Execute-Order-Validate (E-O-V) model, and it is depicted in figure 3.5. Each phase is described in more detail below [7].

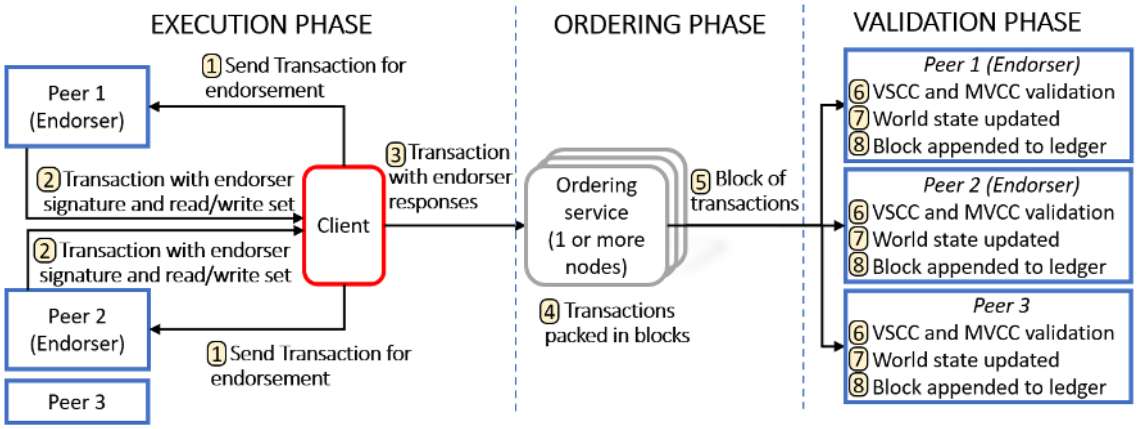

Figure 3.5: Hyperledger Fabric transaction flow [7]

1) **Execution phase**

Step 1: The client sends a transaction to every endorser. The transaction can include multiple reads and writes to one or more keys in the world state.

Step 2: The endorsing peers simulate execution of the transaction on the world state and generate a read/write set that corresponds to the current world state

of each key in the transaction. Then, the endorsing peers send a response back to the client that contains their own signature as well as the read/write set. This distributed execution of transactions on the endorsing peers contributes to maintaining trust without a centralized authority.

Step 3: The client collects the responses from endorsing peers and sends them to the ordering service nodes. The client may optionally validate the signatures of the endorsing peers and the consistency of the read/write set received from different peers. Later in the validation phase, these must be checked. By performing this check, the client can assist in detecting transaction failures early in the transaction flow, hence reducing overhead.

2) **Ordering phase**

Step 4: The ordering service uses a consensus protocol to order the transactions received from the client. A transaction block is generated when one of three conditions is met: a fixed amount of time has elapsed (block timeout), a fixed number of transactions has been received (block size), or the overall size of transactions has reached a fixed limit (block max bytes).

Step 5: The block of transactions is subsequently distributed to all peers.

3) **Validation phase**

Step 6: When a peer receives a block of transactions from the ordering service, each transaction in the block is validated separately. A peer verifies that a sufficient number of valid endorsing peer signatures have been obtained in accordance with the endorsement policy (Validation System Chaincode (VSCC) validation). The peer then validates whether the version of every key in the read set of each transaction is the same as the version of the same key in the current world state (Multiversion Concurrency Control (MVCC) validation). MVCC mechanism ensures consistency in the ledger and prevents double spending.

Step 7: If the VSCC and MVCC validation checks succeed, the transaction write sets are applied to the world state. If any of the validation tests fails, the client is notified that the transaction has been aborted, and the world state remains unchanged.

Step 8: The validated block, which includes both aborted and committed transactions, is appended to the ledger. Every transaction's commit or abort status is logged.

3.2 Chaincode lifecycle

Chaincode represents the executable logic associated with ledger, that serves as storage. Fabric offers domain-independent system chaincode that provide low-level functionality for interacting with the ledger.

The Fabric chaincode lifecycle, illustrated in figure 3.6, is a process that allows organizations to agree on how a chaincode will be operated before it can be used on a channel. Before chaincode can be used it is required to perform the following tasks [24]:

1) **Package the chaincode**. Chaincode needs to be packaged in a tar file, which contains chaincode files and metadata which specify the chaincode language, code path, and package label.

2) **Install the chaincode on peers**. Chaincode package needs to be installed on every peer that will execute and endorse transactions using peer administrator. Install command will return a chaincode package identifier, which is the package label combined with a hash of the package.

3) **Approve chaincode definition**. The approval of chaincode definition acts as a vote by an organization on the chaincode parameters it accepts. Every organization needs to approve chaincode definition. The chaincode definition includes the following parameters:

    - **Package identifier**, which is required for organizations that wants to use the chaincode. An organization can approve a chaincode definition without installing a chaincode package or including the identifier in the definition.
    - **Name**
    - **Version number** assigned to the package
    - **Sequence**, which indicates how many time was chaincode has been upgraded.
    - **Endorsement policy** specifying which organizations need to execute and validate the transaction output. By default, the endorsement policy requires that a majority of organizations in the channel endorse a transaction.
    - **Collection configuration file**, which contains private data collection definition file associated with chaincode.

- **Endorsement System Chaincode (ESCC)/VSCC Plugins**. Optionally a custom endorsement or validation plugin can be used in this chaincode.
- **Initialization**, which indicates if it is required to call `init` function.

4) **Commit the chaincode definition to the channel**. Once a sufficient number of channel members have approved a chaincode definition, it can be committed to the channel by one organization. Before the definition is sent to the ordering service, it is sent to the channel's peers, who endorse it based on whether or not their organization endorsed it. The number of organizations that need to approve a definition is governed by the `LifecycleEndorsement` policy, which is separate from the chaincode endorsement policy. After the commit of the chaincode definition to the channel, the chaincode container will launch on all peers where the chaincode has been installed, allowing channel members to start using the chaincode.

Figure 3.6: Hyperledger Fabric chaincode lifecycle [20]

Applications interact with peers in order to access the ledger. In order to do so, application have to use Fabric SDK. Ledger-query interactions are as easy as a three-step communication between an application and a peer; ledger-update interactions are a little more complicated, requiring two more steps [25].

In figure 3.7 application A connects to P1 and invokes chaincode S1 to query or update the ledger L1. P1 invokes S1 to generate a proposal response containing either a query result or a proposed ledger update. Application A receives the proposal response and, for queries, the process is now complete. For updates, A creates a transaction out of all of the responses and sends it to O1 for ordering. O1 collects transactions from across the network into blocks, and distributes these to all peers, including P1. P1 validates the transaction before committing to L1. Once L1 is updated, P1 generates an event, received by A, to signify completion [25].

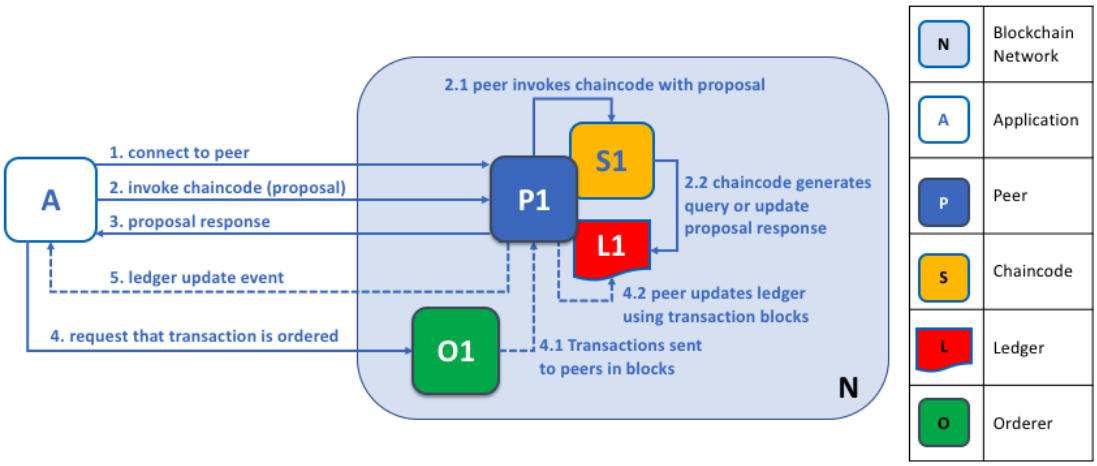

Figure 3.7: Hyperledger Fabric invoke query [25]

3.3 Membership services

These services are used to give access control functionality to Fabric network users. The following functions are carried out via the membership services [5]:

- User identity verification
- User registration
- Assign appropriate permissions to the users depending on their roles

A certificate authority (CA) is used by membership services to provide identity management and authorization operations. This CA might be internal (such as Fabric CA, which is the default interface in Hyperledger Fabric) or external (such as a certificate authority). Enrolment certificates (E-Certs) are issued by Fabric CA and are produced by an enrolment certificate authority (E-CA). Peers are permitted to join the blockchain network once they have been assigned an identity. Temporary certificates (T-Certs) are also issued and are used for one-time transactions [5].

3.4 The Ordering Service

There are several ways for reaching agreement on the strict ordering of transactions between ordering service nodes. Orderers also enforce basic channel access control, limiting who can read and write data to them and who can configure them. Hyperledger Fabric 2.3 ordering services [22]:

- **Raft** (recommended) - CFT, described in more detail in section 3.4.1. Used in proposed solution.
- **Kafka** (deprecated in v2.x) - CFT, similar to Raft with additional administrative overhead of managing a Kafka cluster. It will not be described any further because it is deprecated.
- **Solo** (deprecated in v2.x) - for test only and consists only of a single ordering node. It has been deprecated and may be removed entirely in a future release. It will not be described any further.
- **BFT-SMART** - is **not** part of official Hyperledger Fabric v2.3 release. Barger et al. describe in their paper [4] efforts to transform Fabric into a end-to-end BFT system. They created a (1) stand-alone Byzantine fault-tolerant consensus library, based on BFT-SMART, (2) a full integration of the library with Hyperledger Fabric, which addresses BFT concerns of all its components publicly available at [43], with an accompanying SDK [44].

3.4.1 Raft

The **Raft protocol** is a Crash Fault Tolerant (CFT) consensus mechanism. In Raft, the leader is always assumed to be honest. At the most basic level, it is a replicated log for a replicated state machine (RSM) in which a unique leader is elected every "term" (time division) and whose log is replicated to all follower nodes [5].

Raft is comprised of three sub-problems:

- **Leader election** (a new leader election in case the existing one fails)
- **Log replication** (leader to follower log synch)
- **Safety** (no conflicting log entries (index) between servers)

Election safety, leader append only, log matching, leader completeness, and state machine safety are all guaranteed by the Raft protocol. Each server in Raft can have either a follower, leader, or candidate state. The protocol ensures election safety (i.e., just one winner every election term) as well as liveness (that is, some candidate must eventually win) [5].

The Raft protocol is described in the following steps. At a fundamental level, the protocol is fairly straightforward and may be expressed in the following sequence:

Node starts up –> Leader election –> Log replication

1. The node starts up.
2. The leader election process starts. Once a node is elected as leader, all changes go through that leader.
3. Each change is entered into the node's log.
4. Log entry remains uncommitted until the entry is replicated to follower nodes and the leader receives write confirmation votes from a majority of the nodes, then it is committed locally.
5. The leader notifies the followers regarding the committed entry.
6. Once this process ends, agreement is achieved.

The state transition of the Raft algorithm can be visualized in figure 3.8.

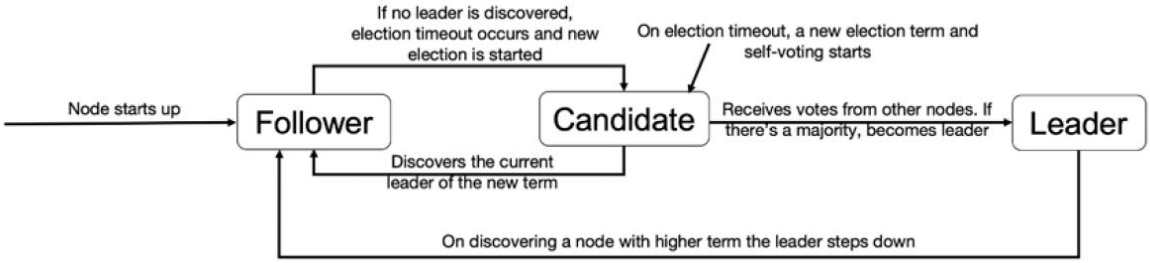

Figure 3.8: Raft state transition [5]

Raft eventually replicates the log (data) across all nodes. The goal of log replication is to keep nodes in sync with one another. Log replication logic can be visualized in figure 3.9.

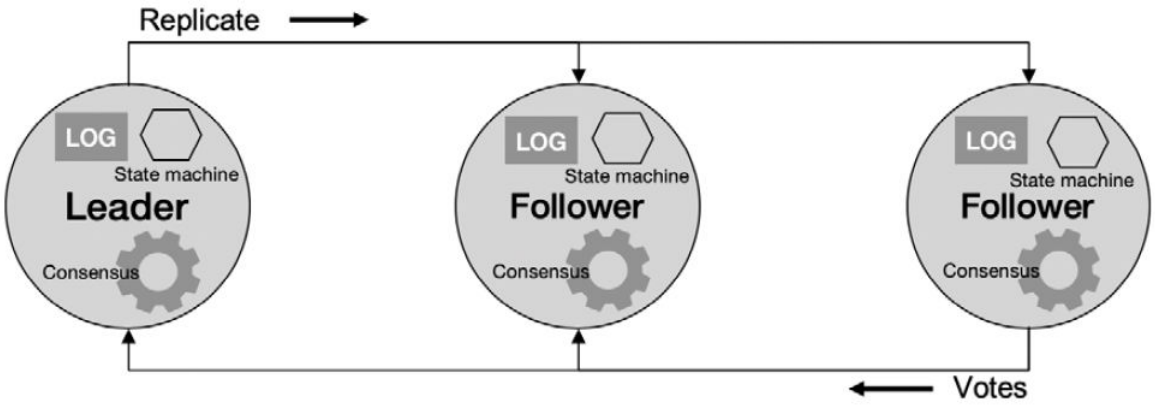

Figure 3.9: Log replication mechanism [5]

Log replication is a straightforward mechanism. The leader is in charge of log replication. Once the leader has a new entry in its log, it sends replication requests

to the follower nodes. When the leader receives enough confirmation votes from the followers showing that the replicate request was accepted and processed, the leader commits that entry to its local state machine. The entry is considered committed at this point [5].

The proposed method utilizes Hyperledger Fabric as a blockchain framework, with a Raft-based CFT ordering service.

4 Business Scenario

This chapter is dedicated to the main aim of this thesis, which is to develop a trustworthy energy consumption monitoring solution to enable accurate billing, already mentioned in Introduction. We present a use-case, which considers consumption data sharing between multiple organizations and final consumer. As the underlying blockchain platform, we propose to use Hyperledger Fabric 2.3.

4.1 Real-world processes

Due to lack of publicly available information of processes regarding heat meters installation and monitoring, the following situation does not necessarily mimic reality. Our prototype will work with the following situation.

There is only one heat meter installed by the **heat distributor** for the entire building. The device readings are taken and based on these measurements the bill for the entire building is calculated. **Heat distributor's** consumer is the **property management** company, which needs to calculate heat bill for all registered **tenants** of the apartment building in a fair way. To do so **property management** company installs heat meters in all apartments.

Manual read is carried out by a meter reader (usually a person working in the property management company) who visits each apartment and takes the read. If it is not possible to take a read (when apartment owner is not present) they will leave a skipped read card explaining why and the read may be estimated. Once the read is taken, the meter reader will upload the reading to their database. Information about consumption reaches apartment owner usually only in annual statement.

Currently, **property management** companies are switching to advanced wireless technologies as there is no need to manually submit meter readings, it protects the privacy of apartment owner, it eliminates errors in meter readings and tampering with meters, allows meter readings at any time, owner can monitor current data via

application. Based on available consumption, apartment owners can better regulate their consumer behavior, thereby optimizing housing costs.

Thanks to smart heat meters, malfunctions can be detected in time and unpleasant surprises in the annual bill can be avoided. More frequent monitoring and notifications allow apartment owners and property management company to take immediate measures.

When implementing a smart meter solution data security and personal data protection must also be taken into account. Directive 2012/27/EU on energy efficiency aims to provide final customers with information on heat and hot water consumption at shorter intervals, so that they can react more quickly to that consumption and, if necessary, regulate it. Meters installed in accordance with Directives 2009/72/EC and 2009/73/EC shall enable accurate billing information based on actual consumption. Member States shall ensure that final customers have the possibility of easy access to complementary information on historical consumption allowing detailed self-checks [10].

Device identity and security will be a key factor in maintaining overall security of buildings and their systems against cyber attacks. Blockchains can be utilized to authenticate device identity, protect and supply data, handle access to facilities, and process transactions in the operations of facilities, all in a distributed way. For example, the IBM Watson IoT platform can include a private blockchain integration that enables IoT devices to send data to private blockchains for inclusion in shared transactions with tamper-resistant records. Distributed replication of the blockchain allows customers and business partners to access and supply IoT data without the need for central control and management. The blockchain can also be used to release payment for services or use of facilities once certain agreed upon conditions are met and tracked [31].

4.2 Shared process workflow

The development of our prototype will initially focus on the communication between:

- Property management company with a certain amount of properties to administer.
- Heating distributor distributes heat to those properties.
- Final heat consumer

Every participant is looking for ways to secure low cost opportunities to cut their bills, save on maintenance, and reduce their environmental impacts.

In this use case we can see that there is need for shared data, strict auditing (billing), need to ensure the confidentiality of transactions (cannot be public), participants have to be known (know who to bill). And this is where blockchain comes in. Blockchain can also handle non-trusting participants, non-trusted intermediary. This technology may be used to make system more resilient to tampering. By processing data from meters using blockchain and smart contracts trust between all parties would be established as we would have single source of truth.

In current process trust issues occurs as final heat consumer has to trust property management company to read the current value from heat meter correctly and property management company has to trust heating distribution company to read correct value from main heat meter.

To demonstrate the value of blockchain, we will use a simplified version of the preceding process with certain modifications. In this series of events, we assume a straight, linear narrative in which all participants are in agreement and nothing out of the ordinary occurs; guards are built into the process merely to catch errors. The following are the transactions in our workflow which is depicted in in figure 4.10.

Figure 4.10: Heat consumption metering workflow

1. The property manager requests registration of main heat meter for the whole building from the heating distributor.
2. The heating distributor accepts registration request of the main heat meter.
3. The property manager regularly updates main meter value.
4. The final consumer requests registration of local heat meter.
5. The property manager accepts registration of local heat meter.
6. The final consumer regularly updates own local meter value.

4.3 Shared assets and data

Participants in our workflow must share some information that allows them to see the consumption monitoring process and its status at any given time.

By evaluating the aforementioned simplified workflow, we can define one asset possessed by each participant and shared with one another to drive the process from one step to the next. Recognized assets, along with their attributes are depicted in table 4.1.

Asset type	Asset attributes
Sensor	sensorID, consumer, consumerMSP, approver, approverMSP, value, status

Table 4.1: Assets with attributes

Table 4.2 depicts data elements that circumscribe the options available to participants in each stage.

Data type	Data attributes	
Sensor	Request and acceptance status:	by heating distributor and property manager company respectively by final consumer and property manager respectively
	Current value(energy consumed):	by final consumer and property manager

Table 4.2: Options available to participants in each stage

4.4 Participants' roles and capabilities

There are 3 categories of participants in our scenario: heating distributor, property management company, final heat consumer. The terms in this set refer to the roles an entity can assume in a deal. Each role's capabilities and restrictions are also defined in the following list:

- Only property manager may request main heat meter registration for the entire apartment building.
- Only heating distributor may accept main heat meter registration request.
- Only property manager may update main heat meter value and only if main heat meter registration was accepted.
- Only final heat consumer may request personal meter registration for owned apartment.

- Only property manager may accept personal heat meter registration request.
- Only final consumer may update main heat meter value and only if main heat meter registration was accepted.
- Heating distributor may see all registered main meters with history.
- Property manager may see all registered personal heat meters with history and main heat meter on the entire building with history.
- Final consumer may see only his own meter with history.

The following chapters will go into greater detail about the development of a solution based on Hyperledger Fabric that will allow for a reliable energy consumption monitoring solution.

5 System architecture

The designed trustworthy energy consumption monitoring solution consists of the following components.

- **Hyperledger Fabric network** responsible for business logic and providing desired characteristics such as immutability, security, and traceability.
- **Applications for organizations** responsible for exposing the blockchain component functionalities.

5.1 Functional considerations

Considering the preliminary considerations, limiting factors, and requirements, which we've discussed in chapter 2, we have decided to employ an permissioned enterprise blockchain in our already discussed use case from section 4.2:

At the beginning of the development 2021, the options for an enterprise blockchain that would meet our requirements were restricted to a few providers, including Hyperledger, Corda, and Quorum. Choice was made based on the following criteria: functional extent, support, cost and interfaces. We finally chose **Hyperledger Fabric**, which is part of the Hyperledger Project founded by the Linux Foundation and is intended to develop enterprise applications based on a permissioned blockchain. It is built on a modular architecture that provides various components such as membership and ordering services.

It should be noted that for our first working prototype there is no need for external storage because document based database **CouchDB**, which is already implemented in Hyperledger Fabric, supports rich queries and works well with JavaScript Object Notation (JSON) format.

Another issue arises when we realize that we need some distributed testing environment. For development purposes we plan to use simple testing network, provided by Hyperledger. Setting up hyperledger test network, which is based on multiple

Docker images, is not trivial and is described in official documentation [21]. On the other hand, production deployment is described only roughly.

The process for deploying a **Fabric network is complex** and presumes an understanding of **Public Key Infrastructure (PKI)** and **managing distributed systems**. We need to be aware of how networks are deployed in order to develop effective smart contracts and applications. It will be required for evaluating the performance of our solution.

5.2 Hyperledger Fabric - tools

Hyperledger also incubates and promotes a range of business blockchain technologies that can be used with various distributed ledger frameworks. The main tools in Hyperledger project which are of interest in this research are:

- **Hyperledger Caliper** [15]. Hyperledger Caliper is a blockchain benchmark tool, which allows users to measure the performance of a specific blockchain implementation with a set of predefined use cases. Hyperledger Caliper is a benchmark tool for blockchain frameworks and relies on a functioning blockchain implementation as the benchmarking target. There are different performance indicators supported in the framework. These include success rate, transaction read rate, throughput, latency, and hardware resource consumption, such as CPU, memory, and I/O.
- **Hyperledger Composer** [17]. Hyperledger Composer is an extensive, open development toolset and framework to make developing blockchain applications easier. The major purpose of Hyperledger Composer was to reduce time to value and make it easier to integrate blockchain applications with existing business systems. However, on August 29th 2019 Hyperledger Composer project was officially **deprecated**. The toolset became too difficult to maintain as the architecture grew and the underlying distributed ledger technology frameworks were updated on a regular basis.
- **Hyperledger Explorer** [18]. Hyperledger Explorer is a user-friendly Web application tool used to view, invoke, deploy or query blocks, transactions and associated data, network information (name, status, list of nodes), chain codes and transaction families, as well as any other relevant information stored in the ledger.

- **Fablo** [11]. Fablo is a simple tool to generate the Hyperledger Fabric blockchain network and run it on Docker. It supports Raft and solo consensus protocols, multiple organizations and channels, chaincode installation and upgrade. On September 21st, 2021, the Hyperledger Foundation posted a video highlighting Fablo in version 0.2, which brought this tool to our notice.

5.2.1 Designing a Hyperledger Fabric network

To design and run a blockchain application, the first step is to determine how many channels are required. For our application we will use one channel, which will maintain the history of meter operations among the different participants. To install an application and run transactions on our smart contract, we will describe how to create and launch a network on which the application will be installed.

The process of building a Hyperledger Fabric network for an application begins with a list of participating organizations. An organization is a security domain as well as a unit of identity and credentials. It governs one or more network peers and relies on a membership service provider (MSP) to provide identities and certificates to peers and clients for smart contract access privileges. Since an MSP instance serves as a CA, it must host a root CA and one or more intermediate CAs within an organization. The ordering service, which acts as a foundation of a Fabric network, is usually assigned its own set of organizations.

Our sample network will consist of two organizations, representing the property manager and heating distributor, and one individual participant, which is heat consumer. The heating distributor represents the heating distributor entity. The property management organization, however, represents both the property management entity and final consumers. An organization represents both a security domain and a business entity; in other words, it provides a trust boundary around a set of entities using an MSP and attests to the veracity of transactions submitted by it.

Running a Fabric peer is a heavy and costly business. Hyperledger fabric focuses on B2B communication and as we can see a final consumer entity represents consumer not a business. So there is a need to group those two in a single organization. A final consumer entity obtains the right to submit transactions or read the ledger state from property management organization in the role of a client.

A single peer within an organization is sufficient to maintain a ledger replica and execute smart contracts, therefore that is how we will create our initial network. For

redundancy, more peers can be added. Aside from the peers, our network includes one root CA, two intermediate CA (one for Org1 and one for Org2 acting as MSP) and an ordering service. The ordering service runs in a single organization. The three organizations with MSPs, peers, and clients of our network are illustrated in figure 5.11.

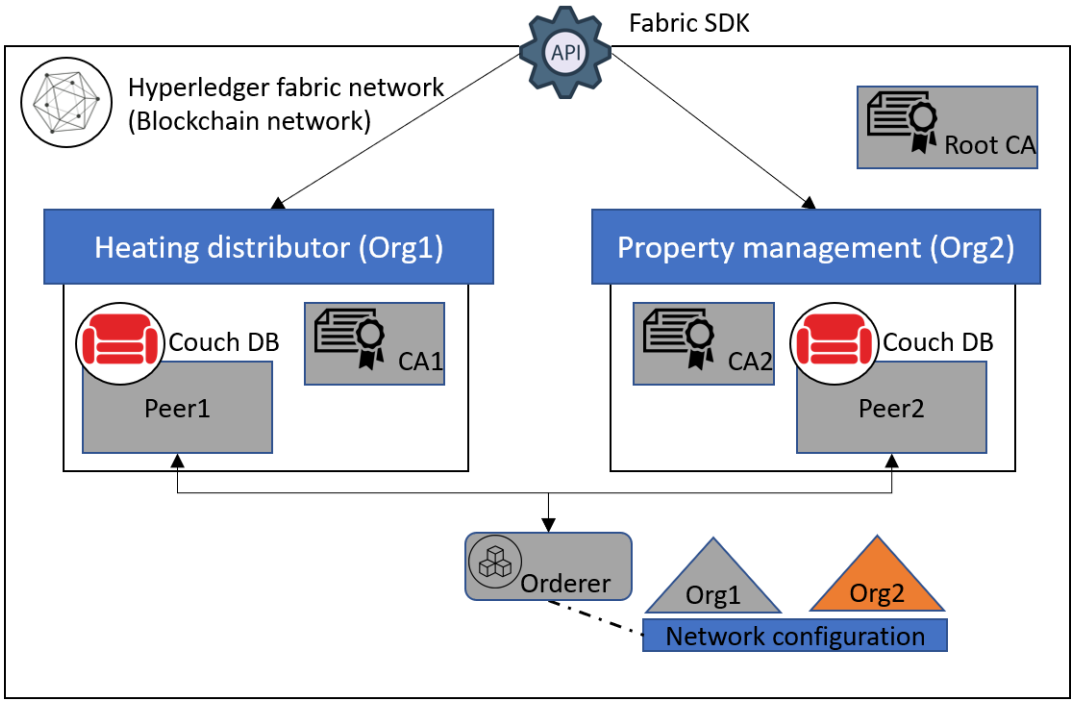

Figure 5.11: Initial Hyperledger Fabric network design

Since the final consumer and property manager are now part of the same organization, the application can distinguish between the two for the purpose of controlling access to the smart contract and ledger. The access control mechanism is implemented as follows:

- Having an organization's MSP, operating as a CA server, distinguishing attributes can be embedded within the certificates it issues to its members. The access control logic is implemented in the contract to parse the attributes and permit or disallow an operation based on business logic requirements.
- Additional access control mechanisms are built into application layers interacting with the contract.

A key feature of a secure and permissioned blockchain is access control. The access control mechanism is typically implemented within the contract and thus enforced

during transaction processing on multiple endorsing peers, and the result validated through transaction consensus.

5.2.2 Applications for organizations

Each service-layer application must implement the operations that are expected to be done at the runtime stage, namely submitting transactions to and querying smart contracts.

Application runtime life cycle is often started by registering and enrolling a user in an organization's MSP (Fabric CA server). As a result, credentials are obtained, which can be stored in wallets on the disk or in a database. A user can submit service requests to a gateway, along with the proper wallet credentials, and the gateway will transform the requests into contract transactions or queries and synchronously return responses. We chose to describe the `PropertyManagementOrg` application because, in addition to implementing enrollment and contract invocation features, it also distinguishes users based on their roles within the organization. Our contracts' Access Control List (ACL) rules require these different roles to be specified in certificate attributes.

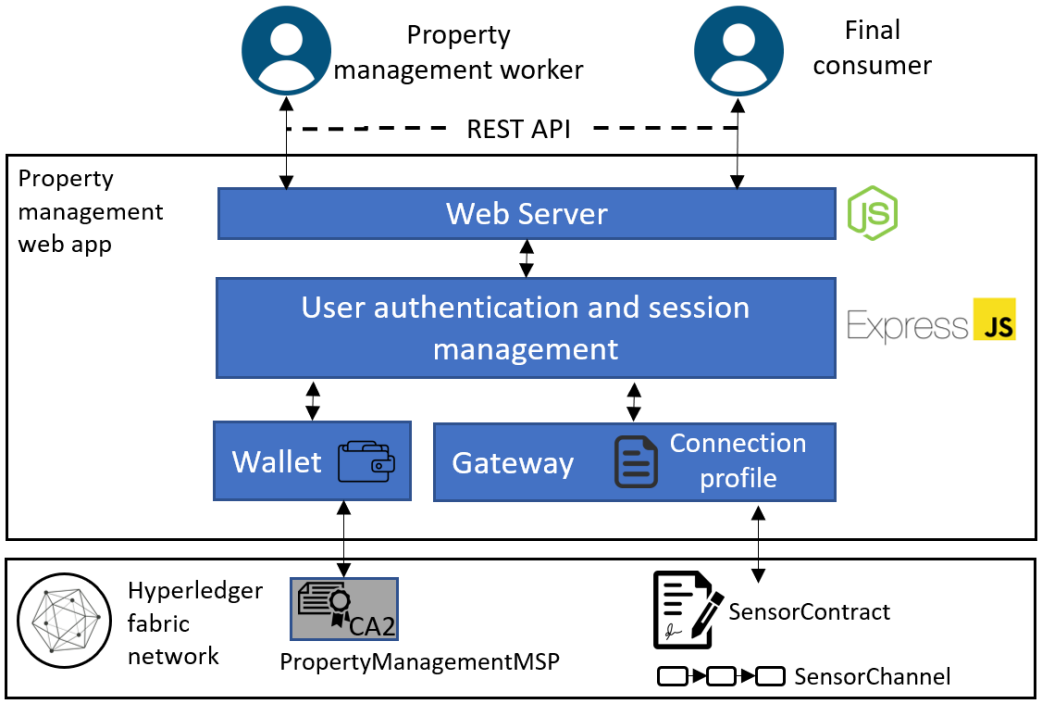

Figure 5.12: Property management web application architecture

This property management application, depicted on figure 5.12, is written in JavaScript and developed using NodeJs framework. It maintains a wallet for the

identities of property management workers and final consumers, obtained through communication with the `PropertyManagementMSP`. All users have access to the same Representational state transfer (REST) API on the web server, which includes functions for registration, login, and smart contract operations. The User Authentication and Session Manager module restricts access to various REST API functions and manages the creation of user identities and roles. This module also maintains sessions using JSON web tokens, allowing a logged-in authenticated user to perform multiple operations with a dynamically generated token.

As we were not able to find any existing solution, this part had to be implemented from scratch.

6 Implementation

In this chapter, we will discuss the designed system from the standpoint of implementation. The first section will discuss how the Fabric network was created, the chaincode development process, and the obstacles encountered throughout this process. Then there will be some technical information on the REST server implementation and chaincode integration into applications.

It is vital to note that this was the most time-consuming area of the thesis. Due to the complexity of Hyperledger Fabric the implementation process was slow. This becomes even more problematic because the framework is still in its early stages and the release of version 2.x that brings breaking changes. We decided not to discuss all of the encountered errors.

6.1 Implementation overview

The following tasks were involved in the overall system's implementation:

- **Hyperledger Fabric network implementation** - began by deploying a Fabric test network capable of deploying and invoking chaincodes. Access control in Hyperledger Fabric depends on network architecture. The created network was then modified to achieve the desired network topology and fit the design. Following the network's deployment, the chaincode's development began with the implementation of the designed system specification. While functionalities were being built, they were being tested using unit tests. When all of the action flows defined in our use-case could be completed via transaction invocation, the procedure was considered complete.
- **REST server implementation** - smart contracts that operate on data shared by many organizations are sensitive bits of code that should only be accessed over secure channels with built-in protections against misuse. As a result, it's

important to add applications on top of contracts, exposing the contracts' capabilities in secure ways. Unlike smart contracts, these apps may be developed as traditional enterprise applications utilizing well-known technology stacks.

6.2 Hyperledger Fabric network

We'll be developing on a Lenovo ThinkPad W541 with Windows 10 pro as the operating system. Only Unix-based operating systems can run Hyperledger Fabric. For this reason, we have installed Ubuntu 20.04 in Windows Subsystem for Linux 2 (WSL2), which allows developers to run a Linux environment as a subsystem on a Windows PC.

6.2.1 Prerequisites

Following the test network setup guide from the Hyperledger Fabric 2.3 documentation [21], the test network was set up. Once a network has been started using shell scripts, we will use VS Code to connect to it and begin writing contracts and applications. A number of requirements must be installed in order to do so, including:

- Git
- cURL
- Docker and Docker Compose
- NodeJS and NPM

As indicated in section 5.2, a tool known as Fablo became available after the majority of the solution had already been implemented. However, analyzing the influence of Fabric network architecture on performance was still ahead of us, so we chose to use Fablo for network building.

6.2.2 Preparing the network

The entire network will be deployed on a single physical machine, with the various network components operating in appropriately configured Docker containers.

The process of setting up a Fabric network may be summarized in the stages below, which will be detailed later in this chapter.

1. **Generating network cryptographic material** - for peers, orderers, TLS based communication...

2. **Generating channel artifacts** - including genesis block and configuration update transactions.
3. **Bringing up network components** (orderers, peers, ...) - using `docker-compose.yaml`.

After completing these steps, we'll have a Fabric network up and running, allowing us to focus on the chaincode. After chaincode implementation, the chaincode lifecycle must be followed, as stated in section 3.2.

Configuration

Several configuration files are essential to the network setup. These are the most significant, and we'll go over them in detail later in this chapter:

- `configtx.yaml` (appendix A, `diploma_thesis_project/fablo-network-generator/fablo-target/fabric-config/configtx.yaml`) - used during generating channel artifacts
- `crypto-config-*.yaml` (appendix A, `diploma_thesis_project/fablo-network-generator/fablo-target/fabric-config/`) - heavily utilized during generating network cryptographic material
- `docker-compose.yaml` (appendix A, `diploma_thesis_project/fablo-network-generator/fablo-target/fabric-docker/docker-compose.yaml`) - responsible for bringing up network components

The Fabric network is created and deployed using the `fabric-docker.sh` (appendix A, `diploma_thesis_project/fablo-network-generator/fablo-target/fabric-docker.sh`) script's commands. The appendix A, `diploma_thesis_project/README_fablo_development_deployment.md` explains how to start a Fabric network.

Generating network cryptographic material

The generation of X.509 certificates and signing keys for the MSP of each peer and orderer organization, as well as for TLS-based communication, is the initial stage in network configuration. We'll also need to make certificates for each peer and orderer node, which will be signed by their respective MSPs. Finally, we'll need to generate additional certificates and keys, as well as TLS root certificates, enabling

TLS-based communication among these peers, orderers, and MSPs. File `crypto-config-org1.yaml` contains the configuration needed to build these cryptographic artifacts. The structure of the organization, the number of peers in each organization, and the default number of users (admin user is created additionally by default) for whom certificates and keys must be issued are all contained in `URL`.

```
PeerOrgs:
  - Name: Org1
    Domain: org1.com
    Specs:
    Template:
      Count: 1
    Users:
      Count: 1
```

Listing 6.1: Contents of crypto-config-org1.yaml

According to the configuration 6.1 of the heating distributor's (Org1) organization, Org1 will have one peer (Template section) and one non-admin user (Users section). The peer's organization domain name and its CA are also specified.

Appendix A, `diploma_thesis_project/fablo-network-generator/fablo-target/fabric-config/` contains similar `crypto-config-*` files for the orderer organization and the other peer organization.

We want to clarify that ordinary and administrator users may be created dynamically by submitting requests to the MSPs of organizations.

Generating channel artifacts

To build a network based on an organization's structure and bootstrap a channel, we'll need to provide the following artifacts:

- A genesis block that contains consortium specifications and organization-specific certificates that define who can create and manage network channels. This block will act as the initial block of the orderer system channel, which is maintained by the ordering service nodes to track multiple application channels created inside the network.
- A channel configuration transaction.
- Anchor peer configuration transactions for each organization. An anchor peer acts as a fulcrum inside an organization enabling Fabric gossip-based cross-organization ledger synchronization.

The `configtxgen` tool from Fabric binaries was used to perform the actions in this section. This phase is dependent on the `configtx.yaml` configuration file, which

will be described more below. The following are the most important sections of `configtx.yaml`:

- **Channel profiles** - The profiles section 6.2 describes the organizational structure of our sensor network.

```
Profiles:
    # Profile used to create Genesis block for group group1 #
    Group1Genesis:
        <<: *ChannelDefaults
        Orderer:
            <<: *Group1Defaults
            Organizations:
                - *Orderer
            Capabilities:
                <<: *OrdererCapabilities
        Consortiums:
            SampleConsortium:
                Organizations:
                    - *Orderer
                    - *Org1
                    - *Org2

    # Profile used to create channeltx for my-channel1 #
    MyChannel1:
        <<: *ChannelDefaults
        Orderer:
            <<: *Group1Defaults
            Organizations:
                - *Orderer
            Capabilities:
                <<: *ApplicationCapabilities
        Consortium: SampleConsortium
        Consortiums:
            SampleConsortium:
                Organizations:
                    - *Org1
                    - *Org2
        Application:
            <<: *ApplicationDefaults
            Organizations:
                - *Org1
                - *Org2
```

Listing 6.2: Profile section of configtx.yaml

The configuration of the genesis block is provided in the `Group1Genesis` section, which specifies one consortium, `SampleConsortium`. This consortium is made up of a group of organizations that work together to maintain a channel. `SampleConsortium` is comprised of `Org1`, `Org2`, and `Orderer`, each of which is specified in its own subsection in the Organizations section of the file. The orderer is a member of its own organization named `Orderer`. `MyChannel1` section describes the channel settings for our distributed application. This channel is associated with `SampleConsortium`.

- **Organization configurations** - Each organization section includes information about its MSP, hostname and port information for its anchor peers.

```
Organizations:
    - &Org1
      Name: Org1MSP
      ID: Org1MSP
      MSPDir: crypto-config/peerOrganizations/org1.com/msp

      Policies:
          Readers:
              Type: Signature
              Rule: "OR('Org1MSP.member')"
          Writers:
              Type: Signature
              Rule: "OR('Org1MSP.member')"
          Admins:
              Type: Signature
              Rule: "OR('Org1MSP.admin')"
          Endorsement:
              Type: Signature
              Rule: "OR('Org1MSP.member')"

      AnchorPeers:
          - Host: peer0.org1.com
            Port: 7041
```

Listing 6.3: Organizations section of configtx.yaml

Listing 6.3 depicts the configuration of `Org1`. The names of the organization is `Org1` and it's MSP is `Org1MSP`. The `MSPDir` variable refers to the location of this organization's cryptographic material, which we produced previously using the cryptogen command. Policies for reading, writing, and administering the channel are described in the Policies section (privileges are proven by certificates given by the Org1 organization's MSP). The organization's policy for endorsing (signing) a smart contract transaction is also defined.

- **Ordering service configuration** - Orderer configuration is in listing 6.4.

```
Orderer: &Group1Defaults
    OrdererType: etcdraft
    Addresses:
        - orderer0.group1.orderer.com:7030
    EtcdRaft:
        Consenters:
            - Host: orderer0.group1.orderer.com
              Port: 7030
              ClientTLSCert: crypto-config/peerOrganizations/orderer.com/
    peers/orderer0.group1.orderer.com/tls/server.crt
              ServerTLSCert: crypto-config/peerOrganizations/orderer.com/
    peers/orderer0.group1.orderer.com/tls/server.crt

    BatchTimeout: 2s
    BatchSize:
        MaxMessageCount: 10
        AbsoluteMaxBytes: 99 MB
        PreferredMaxBytes: 512 KB
    Organizations:
    # Policies defines the set of policies at this level of the config tree
    # For Orderer policies, their canonical path is
    #   /Channel/Orderer/<PolicyName>
    Policies:
        Readers:
```

```
            Type: ImplicitMeta
            Rule: "ANY Readers"
        Writers:
            Type: ImplicitMeta
            Rule: "ANY Writers"
        Admins:
            Type: ImplicitMeta
            Rule: "MAJORITY Admins"
        # BlockValidation specifies what signatures must be included in the
     block
        # from the orderer for the peer to validate it.
        BlockValidation:
            Type: ImplicitMeta
            Rule: "ANY Writers"
    Capabilities:
        <<: *OrdererCapabilities
```

Listing 6.4: Orderer section of configtx.yaml

The Addresses section lists the orderer nodes, along with their hostnames and listening ports. The `BatchTimeout` and `BatchSize` parameters control the production of blocks. The maximum number of messages included inside a block is specified by `MaxMessageCount`, while the maximum length of time to wait for new transactions before forming a block is specified by `BatchTimeout`. The Policies section is similar to the Organizations section. To avoid explicit calling out of signatories, we describe `ImplicitMeta` policies, which are compositions of simpler (Signature) policies. `"ANYWriters,"` for example, specifies that every member of the orderer organizations has write access for block creation. `BlockValidation` is a policy rule that allows ordering nodes to sign blocks. Finally, the `OrdererType` option specifies how our ordering service is configured. It is set to `etcdraft` in this case, which gives fault-tolerance guarantees.

- Other parts of the `configtx.yaml` file, such as Channel and Application, are used to define channel- and application-specific rules. The Capabilities section outlines version compatibility criteria. We will not go into specifics regarding these options.

Bringing up network components

The `docker-compose.yaml` network configuration file is used to start the network as a group of Docker containers. This file defines peers, peer databases, orderers, and CAs as services and sets of attributes, allowing us to deploy them as interconnected Docker containers all at once rather than manually running instances of these services on one or more machines.

The services we need to correspond to our sensor network's nodes are:

- Two instances of a Fabric peer, one for each organization.

- One instance of a Fabric orderer running in Raft mode
- Three instances of a Fabric CA, corresponding to the MSPs of Org1, Org2 and Orderer.

Full content of `docker-compose.yaml` is available in appendix A. Any Fabric peer configuration parameter can be set in this file using environment variables. It is out of scope of this thesis to describe this file. Configuration files `core.yaml`, `orderer.yaml` are described in [23].

6.2.3 Chaincode implementation, testing and deployment

The data structure and its fields are implemented in accordance with the specifications provided in chapter 4. The `sensorContract` is characterized by the data structure shown in table 4.1. This contract is primarily in charge of tracking the states of smart meters. This is accomplished by retaining state and specified rules that determine whether or not a state change is possible. In our use case, the sensor begins with the status REQUESTED and progresses to ACCEPTED. Only after the status is ACCEPTED may the value that defines the quantity of consumed energy be updated.

The access control mechanism is implemented into the smart contract and hence enforced throughout transaction processing on several endorsing peers, with the outcome certified via transaction consensus. It is implemented using ACLs in compliance with the requirements specified in section 4.4. One example is visualized in listing 6.5. ACLs define which roles from which organizations may execute specific functions. Additional access control mechanisms are inbuilt into application layers interacting with the contract.

```
this.aclRules[SensorContract.getAclSubject('Org1MSP', 'finalConsumer')] = ['init'
    , 'exists', 'requestSensor', 'uploadDataSensor', 'getSensor', '
    getSensorStatus', 'deleteSensor', 'listSensor', 'getSensorHistory'];
```

Listing 6.5: Example of ACLs

The `beforeTransaction` function, in listing 6.6, includes the enforcement of our policy which uses an `aclRules` to verify authorization of the transaction invoking identity.

```
public async beforeTransaction(ctx: Context) {
    const id = ctx.clientIdentity.getID();
    const mspId = ctx.clientIdentity.getMSPID();
    const role = this.getUserRole(ctx);
    const tx = ctx.stub.getFunctionAndParameters().fcn;
    const aclSubject = SensorContract.getAclSubject(mspId, role);
    if (!this.aclRules.hasOwnProperty(aclSubject)) {
```

```
8           throw new Error(`The participant with id: ${id} belonging to MSP: ${mspId
    } and role: ${role} is not recognized`);
9       }
10      if (!this.aclRules[aclSubject].includes(tx)) {
11          throw new Error(`The participant with id: ${id} belonging to MSP: ${mspId
    } and role: ${role} cannot invoke transaction ${tx}`);
12      }
13  }
```

Listing 6.6: Implementation of beforeTransaction

E-Certs are issued by the Fabric CA to network users. When a user submits to Fabric, the E-Certs represents the user's identity and is used as a signed transaction. Before invoking a transaction, the user must first register and get an E-Certs from Fabric CA.

Fabric uses attribute-based access control (ABAC). The ABAC enables the contract to make access control decisions based on user identification attributes. Users that have an E-Certs have access to a number of additional attributes (key-value pairs). In this case we used additional attribute `BUSINESS_ROLE`.

Based on our use case, the table 6.1 describes the collection of functions that record and retrieve data to and from the ledger to provide the contracts' business logic. The table also defines the access control definitions of organization members, which are required to execute the appropriate functions.

Function	**Roles permitted to invoke**	**Description**
exists	Org1MSP.finalConsumer Org1MSP.propertyManagementWorker Org2MSP.heatingDistributorWorker	Checks if a sensor exists
requestSensor	Org1MSP.finalConsumer Org1MSP.propertyManagementWorker	Requests sensor registration.
acceptSensor	Org1MSP.propertyManagementWorker Org2MSP.heatingDistributorWorker	Accepts sensor registration request.
uploadDataSensor	Org1MSP.finalConsumer Org1MSP.propertyManagementWorker	Uploads new value from sensor.
getSensor	Org1MSP.finalConsumer Org1MSP.propertyManagementWorker Org2MSP.heatingDistributorWorker	Gets sensor data.

Table 6.1 continued from previous page

Function	Roles permitted to invoke	Description
getSensorStatus	Org1MSP.finalConsumer Org1MSP.propertyManagementWorker Org2MSP.heatingDistributorWorker	Gets sensor status.
listSensor	Org1MSP.finalConsumer Org1MSP.propertyManagementWorker Org2MSP.heatingDistributorWorker	Gets list of sensors, where the requester is involved.
getSensorHistory	Org1MSP.finalConsumer Org1MSP.propertyManagementWorker Org2MSP.heatingDistributorWorker	Gets full history for sensor.

Table 6.1: Contract functions of SensorContract

All functions from table 6.1 are implemented in appendix A in file `diploma_thesis_project/chaincode-typescript/src/sensor-contract.ts`.

Implementation of unit tests for our contract functions is available in appendix A `diploma_thesis_project/chaincode-typescript/src/sensor-contract.spec.ts`. Following technologies were used: the Mocha testing framework [35] with the Chai assertion library [8], which enables to write unit tests as sentences, and the Sinon mocking library [42], which allows to create stubs of objects.

Chaincode deployment

After completing the preceding steps, our network is operational, but no channels exist between the participants. For this, Hyperledger Fabric provides binaries. The `configtx.yaml` file is used to define channel creation transaction as well as transactions that update peers to be anchor peers. Then, using peer binary, the channel is created and the organization's peers are added as anchor peers.

There is now a channel between the organizations, but no chaincode is deployed on it. To deploy a chaincode, follow the steps outlined in section 3.2. Binaries provided by Hyperledger Fabric allowed us to carry out those steps.

6.3 REST server implementation

Because the REST server is traditional enterprise application, already discussed in section 5.2.2, the implementation process was straightforward. The server is built using NodeJS and ExpressJS. `Fabric-network` and `fabric-ca-client` were used to connect to the Fabric network.

Fabric identity creation is noteworthy. Fabric provides the ability to dynamically create identities and credentials using the Fabric CA server. To enforce access control rules, our deployed contract depends on special attributes to be present in callers' certificates. Such properties are not present in certificates generated statically.

In this use case, each unique `finalConsumer`, `propertyManagementWorker`, and `heatingDistributorWorker` is assigned a unique Fabric identity and mapped to it. Appendix A contains the code for Fabric identity creation and maintenance, which can be found in `diploma_thesis_project/apps/org2backendapp/src/models/identityModel.js`.

The process of creating a new identity consists of three steps:

1. **Enrollment of registrar** - When a Fabric CA server is created for each organization, it includes an administrator account. This administrator user serves as the CA's registrar. To do so, it must first enroll with the CA, which involves generating a public-private key pair and receiving an X.509 certificate from the CA by sending the public key along with a certificate signing request. The certificate, together with other metadata, is saved in the local wallet.
2. **Registration of user** - The enrolled registrar submits a registration request for a user to the CA by providing a username. The CA creates an identity for the user internally and returns a secret to the registrar. In the `PropertyManagementOrg` application, we need to separate regular client users into `finalConsumer` and `propertyManagementWorker`. This is accomplished by mapping the proper role. The registration request is sent to the CA specified in the connection profile.
3. **Enrollment of user** - The user can now enroll with the CA in the same manner as the registrar, using its username and the secret obtained in the previous stage as a password. An identity is formed and saved in a file in the wallet directory.

When a user calls the `/register` endpoint, the three-step preceding procedure outlined is initiated.

6.4 Summary

Implementing a full blockchain application is an ambitious and difficult task. A wide variety of skills is required - systems, networking, security, web application development, distributed deployment, ...

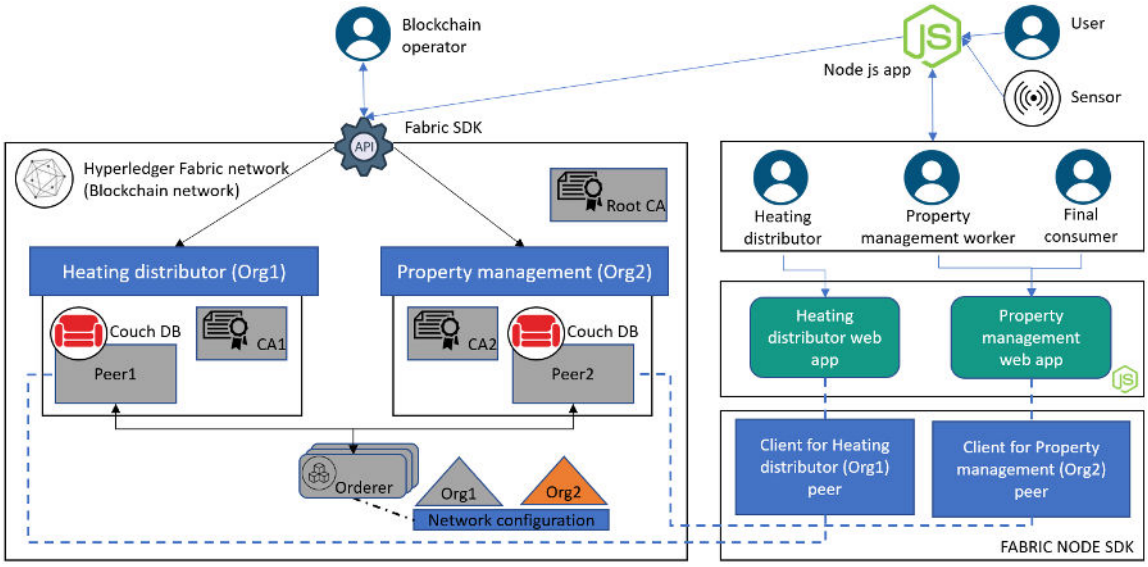

Figure 6.13: Full solution architecture

Using our consumption monitoring use case, we implemented a distributed application over a Fabric network. This proof of concept solution consists of a network of 2 organizations. Both of these are represented by an Hyperledger Fabric organization holding one peer node, one orderer node and certificate authorities to issue certificates to its members as depicted in figure 6.13.

Our distributed application allows three independent personas from two organizations to manage consumption monitoring workflow. The attributes of sensor-related artifacts, as well as the history of this workflow, were stored in a tamper-resistant, shared, replicated ledger. Once the smart contracts were complete, we used REST APIs to offer their capabilities to different organizations' members in different ways.

7 Evaluation

7.1 Flow validation

Now we'll complete the whole workflow depicted in chapter 4.2, changing between different users to drive the workflow. Curl commands with different parameters are used to invoke services. These commands can also be found in appendix A.

Identities creation

The following users have to be registered (passwords are set to `password` for all users):

- In the `PropertyManagementOrg` application running on `http://localhost:4000`, we create:

  a user with username `finalconsumer` with role `finalconsumer`

  ```
  curl -X POST http://localhost:4000/register -H "content-type:application/x-www-form-urlencoded" -d 'registrarUser=admin&registrarPassword=adminpw&username=finalconsumer&password=password&role=finalconsumer'
  ```

  with response

  ```
  true
  ```

  a user with username `propertymanagementworker` with role `propertymanagementworker`

  ```
  curl -X POST http://localhost:4000/register -H "content-type:application/x-www-form-urlencoded" -d 'registrarUser=admin&registrarPassword=adminpw&username=propertymanagementworker&password=password&role=propertymanagementworker'
  ```

  with response

  ```
  true
  ```

- In the `HeatingDistributorOrg` application running on `http://localhost:4001`, we create:

  a user with username `heatingdistributorworker` with role `heatingdistributorworker`

```
curl -X POST http://localhost:4001/register -H "content-type:application/x-
    www-form-urlencoded" -d 'registrarUser=admin&registrarPassword=adminpw&
    username=heatingdistributorworker&password=password&role=
    heatingdistributorworker'
```

with response

```
true
```

This can be run on a newly created network with no users currently registered.

The wallets folder `diploma_thesis_project/apps/org1backendapp/org1.example.com/wallets` now contains identities `admin.id`, `finalconsumer.id`, `propertymanagementworker.id`.

The wallets folder `diploma_thesis_project/apps/org2backendapp/org2.example.com/wallets` now contains identities `admin.id` and `heatingdistributorworker.id`.

Those identities are stored as JSON objects and some of their attributes are the MSP ID of the organization, the client's private signing key, the certificate issued for the client by the MSP.

Users logging in

Following the registration of users, we have to log them into their respective applications and receive tokens for each of them. Because we will be reusing these tokens for several actions, we decided to save them in environment variables. This step requires the jq tool [28].

- `finalconsumer` logging in

```
JWT_FIN=$(curl -X POST http://localhost:4000/login -H "content-type:
    application/x-www-form-urlencoded" -d 'username=finalconsumer&password=
    password' 2>/dev/null | jq .token)
JWT_FIN=${JWT_FIN:1:${#JWT_FIN}-2}
export JWT_FIN
```

- `propertymanagementworker` logging in

```
JWT_PRO=$(curl -X POST http://localhost:4000/login -H "content-type:
    application/x-www-form-urlencoded" -d 'username=propertymanagementworker&
    password=password' 2>/dev/null | jq .token)
JWT_PRO=${JWT_PRO:1:${#JWT_PRO}-2}
export JWT_PRO
```

- `heatingdistributorworker` logging in

```
JWT_HEA=$(curl -X POST http://localhost:4001/login -H "content-type:
    application/x-www-form-urlencoded" -d 'username=heatingdistributorworker&
    password=password' 2>/dev/null | jq .token)
JWT_HEA=${JWT_HEA:1:${#JWT_HEA}-2}
export JWT_HEA
```

Workflow

After logging users into their individual applications and getting tokens for each of them, we can begin the workflow described in chapter 4.2.

1. The property manager requests registration of main heat meter for the whole building from the heating distributor.

```
curl -X POST http://localhost:4000/sensor/request -H "content-type:
    application/x-www-form-urlencoded" -H "authorization: Bearer ${JWT_PRO}"
    -d 'sensorId=sensor1&value=0&approverMSP=Org2MSP'
```

   with response

```
true
```

   verify the existence of sensor

```
curl -X GET http://localhost:4000/sensor/sensor1 -H "authorization: Bearer ${
    JWT_PRO}"
```

   with response

```
{
 "approver": null,
 "approverMSP": "Org2MSP",
 "consumer": "x509::/OU=client/CN=propertymanagementworker::/C=US/ST=
    California/L=San Francisco/O=org1.com/CN=ca.org1.com",
 "consumerMSP": "Org1MSP",
 "sensorID": "sensor1",
 "status": "REQUESTED",
 "value": 0
}
```

2. The heating distributor accepts registration request of the main heat meter.

```
curl -X GET http://localhost:4001/sensor/sensor1/accept_sensor -H "
    authorization: Bearer ${JWT_HEA}"
```

   with response

```
true
```

   verify the existence of a sensor

```
curl -X GET http://localhost:4001/sensor/sensor1 -H "authorization: Bearer ${
    JWT_HEA}"
```

   with response

```
{
 "approver": "x509::/OU=client/CN=heatingdistributorworker::/C=US/ST=
    California/L=San Francisco/O=org2.com/CN=ca.org2.com",
 "approverMSP": "Org2MSP",
 "consumer": "x509::/OU=client/CN=propertymanagementworker::/C=US/ST=
    California/L=San Francisco/O=org1.com/CN=ca.org1.com",
 "consumerMSP": "Org1MSP",
 "sensorID": "sensor1",
 "status": "ACCEPTED",
 "value": 0
}
```

3. The property manager regularly updates main meter value.

```
curl -X POST http://localhost:4000/sensor/upload -H "content-type:
    application/x-www-form-urlencoded" -H "authorization: Bearer ${JWT_PRO}"
    -d 'sensorId=sensor1&value=15'
```

with response

```
true
```

verify the existence of a sensor

```
curl -X GET http://localhost:4001/sensor/sensor1 -H "authorization: Bearer ${
    JWT_HEA}"
```

with response

```
{
 "approver": "x509::/OU=client/CN=heatingdistributorworker::/C=US/ST=
    California/L=San Francisco/O=org2.com/CN=ca.org2.com",
 "approverMSP": "Org2MSP",
 "consumer": "x509::/OU=client/CN=propertymanagementworker::/C=US/ST=
    California/L=San Francisco/O=org1.com/CN=ca.org1.com",
 "consumerMSP": "Org1MSP",
 "sensorID": "sensor1",
 "status": "ACCEPTED",
 "value": 15
}
```

4. The final consumer requests registration of local heat meter.

```
curl -X POST http://localhost:4000/sensor/request -H "content-type:
    application/x-www-form-urlencoded" -H "authorization: Bearer ${JWT_FIN}"
    -d 'sensorId=sensor2&value=0&approverMSP=Org1MSP'
```

with response

```
true
```

verify the existence of a sensor

```
curl -X GET http://localhost:4000/sensor/sensor2 -H "authorization: Bearer ${
    JWT_FIN}"
```

with response

```
{
 "approver": null,
 "approverMSP": "Org1MSP",
 "consumer": "x509::/OU=client/CN=finalconsumer::/C=US/ST=California/L=San
    Francisco/O=org1.com/CN=ca.org1.com",
 "consumerMSP": "Org1MSP",
 "sensorID": "sensor2",
 "status": "REQUESTED",
 "value": 0
}
```

5. The property manager accepts registration of local heat meter.

```
curl -X GET http://localhost:4000/sensor/sensor2/accept_sensor -H "
    authorization: Bearer ${JWT_PRO}"
```

with response

```
true
```

verify the existence of a sensor

```
curl -X GET http://localhost:4000/sensor/sensor2 -H "authorization: Bearer ${
    JWT_PRO}"
```

with response

```
{
 "approver": "x509::/OU=client/CN=propertymanagementworker::/C=US/ST=
    California/L=San Francisco/O=org1.com/CN=ca.org1.com",
 "approverMSP": "Org1MSP",
 "consumer": "x509::/OU=client/CN=finalconsumer::/C=US/ST=California/L=San
    Francisco/O=org1.com/CN=ca.org1.com",
 "consumerMSP": "Org1MSP",
 "sensorID": "sensor2",
 "status": "ACCEPTED",
 "value": 0
}
```

6. The final consumer regularly updates own local meter value.

```
curl -X POST http://localhost:4000/sensor/upload -H "content-type:
    application/x-www-form-urlencoded" -H "authorization: Bearer ${JWT_FIN}"
    -d 'sensorId=sensor2&value=25'
```

with response

```
true
```

verify the existence of a sensor

```
curl -X GET http://localhost:4000/sensor/sensor2 -H "authorization: Bearer ${
    JWT_FIN}"
```

with response

```
{
 "approver": "x509::/OU=client/CN=propertymanagementworker::/C=US/ST=
    California/L=San Francisco/O=org1.com/CN=ca.org1.com",
 "approverMSP": "Org1MSP",
 "consumer": "x509::/OU=client/CN=finalconsumer::/C=US/ST=California/L=San
    Francisco/O=org1.com/CN=ca.org1.com",
 "consumerMSP": "Org1MSP",
 "sensorID": "sensor2",
 "status": "ACCEPTED",
 "value": 25
}
```

Additional queries

1. Viewing list of sensors that are associated with the user

```
 curl -X GET http://localhost:4000/sensor -H "authorization: Bearer ${JWT_PRO
    }"
```

with response

```
[
 {
```

```
  "approver": "x509::/OU=client/CN=propertymanagementworker::/C=US/ST=
    California/L=San Francisco/O=org1.com/CN=ca.org1.com",
  "approverMSP": "Org1MSP",
  "consumer": "x509::/OU=client/CN=finalconsumer::/C=US/ST=California/L=San
    Francisco/O=org1.com/CN=ca.org1.com",
  "consumerMSP": "Org1MSP",
  "sensorID": "sensor2",
  "status": "ACCEPTED",
  "value": 25
 },
 {
  "approver": "x509::/OU=client/CN=heatingdistributorworker::/C=US/ST=
    California/L=San Francisco/O=org2.com/CN=ca.org2.com",
  "approverMSP": "Org2MSP",
  "consumer": "x509::/OU=client/CN=propertymanagementworker::/C=US/ST=
    California/L=San Francisco/O=org1.com/CN=ca.org1.com",
  "consumerMSP": "Org1MSP",
  "sensorID": "sensor1",
  "status": "ACCEPTED",
  "value": 15
 }
]
```

2. Viewing sensor history for auditability reasons

```
curl -X GET http://localhost:4000/sensor/sensor1/history -H "authorization:
    Bearer ${JWT_PRO}"
```

with response

```
[
 {
  "txId": "da7f0c1c6ab8843f2a0958e70a85e04e52e84d44046bf0f4c5676c8e01513747",
  "timestamp": "Mon Jan 03 2022 17:24:12 GMT+0000 (Coordinated Universal Time
    )",
  "isDelete": "false",
  "sensorAgreement": {
   "approver": "x509::/OU=client/CN=heatingdistributorworker::/C=US/ST=
    California/L=San Francisco/O=org2.com/CN=ca.org2.com",
   "approverMSP": "Org2MSP",
   "consumer": "x509::/OU=client/CN=propertymanagementworker::/C=US/ST=
    California/L=San Francisco/O=org1.com/CN=ca.org1.com",
   "consumerMSP": "Org1MSP",
   "sensorID": "sensor1",
   "status": "ACCEPTED",
   "value": 15
  }
 },
 {
  "txId": "cc54aa005677c91c99ed421e09009243051673f749fe7cbef8547795d67e6508",
  "timestamp": "Mon Jan 03 2022 17:24:09 GMT+0000 (Coordinated Universal Time
    )",
  "isDelete": "false",
  "sensorAgreement": {
   "approver": "x509::/OU=client/CN=heatingdistributorworker::/C=US/ST=
    California/L=San Francisco/O=org2.com/CN=ca.org2.com",
   "approverMSP": "Org2MSP",
   "consumer": "x509::/OU=client/CN=propertymanagementworker::/C=US/ST=
    California/L=San Francisco/O=org1.com/CN=ca.org1.com",
   "consumerMSP": "Org1MSP",
```

```
  "sensorID": "sensor1",
  "status": "ACCEPTED",
  "value": 0
 }
},
{
 "txId": "9ea756e35984bb10934235caf5e3b609d7895205810f7ceaa602e1ddeb3f2b2a",
 "timestamp": "Mon Jan 03 2022 17:24:07 GMT+0000 (Coordinated Universal Time
  )",
 "isDelete": "false",
 "sensorAgreement": {
  "sensorID": "sensor1",
  "consumerMSP": "Org1MSP",
  "consumer": "x509::/OU=client/CN=propertymanagementworker::/C=US/ST=
   California/L=San Francisco/O=org1.com/CN=ca.org1.com",
  "approverMSP": "Org2MSP",
  "approver": null,
  "value": 0,
  "status": "REQUESTED"
 }
}
]
```


7.2 Performance evaluation

This section focuses on the application's performance on selected experiments and evaluation. The tests are run locally with Hyperledger Caliper v0.4.2, which is already described in section 5.2, and the configuration files provided in appendix A.

Caliper requires the definition of network configurations (Appendix A, `caliperworkspace/network/`), test configurations (Appendix A, `caliperworkspace/benchmarks/`), and test files (Appendix A, `caliperworkspace/workload/`) in order to function with new chaincodes and existing networks.

To measure blockchain performance with Hyperledger Caliper, the following steps were performed:

1. Fabric network was created, initialized and chaincode was deployed.
2. Both organizations' applications are launched, and an identity is created for each role using `/register` endpoint. This is required because our chaincode functions expect specific attributes to be present in the caller certificate.
3. Caliper was configured to connect to Fabric network using the network configurations file (Appendix A, `caliperworkspace/network/`).
4. Test workload modules were written to test performance of following chaincode functions: `requestSensor`, `uploadDataSensor`, `getSensor`, `listSensor`,

`getSensorHistory` (Appendix A, `caliperworkspace/workload/`).

5. Test configurations were written (Appendix A, `caliperworkspace/benchmarks/`).
6. Caliper was launched, and the findings were saved in the form of auto-generated HTML reports.

The following setup was used for development and testing, some information was already provided in section 6.2:

- Host machine: Notebook Lenovo W541
  - OS: Windows 10 Pro 64-bit (10.0, Build 19043)
  - RAM: 4x8GB DDR3 1600MHz Processor: 2.8GHz Intel core i7 4810MQ
  - SSD: SAMSUNG SSD 860 EVO 500 GB Architecture: x86_64
  - CPU(s): 8
  - On-line CPU(s) list: 0-7
  - Core(s) per socket: 4
- Fabric network, REST servers and Caliper
  - WSL2 with Ubuntu 20.04
  - Docker desktop 4.3.1

The transaction throughput and latency of the Hyperledger Fabric network is measured in this section. The **throughput** is defined as follows [14]:

$$Transaction\ Throughput\ =\ \frac{Total\ committed\ transactions}{total\ time\ in\ seconds} @\ \#committed\ nodes$$

Transaction throughput is the rate at which valid transactions are committed by the blockchain System Under Test (SUT) in a defined time period. Note that this is not the rate at a single node, but across the entire SUT, i.e. committed at all nodes of the network. This rate is expressed as transactions per second (TPS) at a network size. To get the total committed transactions, total number of invalid transactions is subtracted from the total number of transactions. Even if two networks have the same throughput, one will be more effective if it has a greater success rate [14].

Transaction Latency is defined as follows [14]:

$$Transaction\ latency\ =\ (Confirmation\ time\ @\ network\ treshold)\ -\ submit\ time$$

Transaction latency is the amount of time it takes for the effect of a transaction to be usable throughout the network. This includes the propagation time as well as any settling time produced by the consensus mechanism. All nodes in the SUT should be used to measure delay. The only meaningful network threshold for non-probabilistic protocols like Raft is 100% [14].

7.2.1 Experiment 1

The transaction send rate can be configured in Caliper's benchmark configuration file. This is referred to as the expected send rate, and it is not the same as the actual send rate. The expected send rate can be set to anything, but Caliper's resources restrict the actual send rate.

First experiment determines Caliper's highest possible actual send rate. The send rate should be unaffected by the Fabric network architecture and hence should provide an independent measurement of Caliper's performance. In this case, when Caliper and Fabric network run on the same computer, measurement is not completely independent.

During this experiment, the **fixed-rate rate controller** is used. It is the most basic controller. It sends input transactions at the specified TPS interval [16]. Tests were run with 1, 2, 3, 4, 5 workers. We believe that higher amount of workers should lead to higher send rate. The table 7.1 contains these measurements.

Observation

As seen from table 7.1, not every chaincode function has the same **send rate**. Caliper generates transactions in the test files, and the transaction generation speed is affected by complexity of tasks. `ReadSensor` has the highest send rate, because it require only generation of sensor identification. `GetSensorHistory` also requires only generation of sensor identification, but its send rate is lower. In the case of `readSensor`, `updateSensor`, `getSensorHistory` and `listSensor`, the highest send rate is achieved when 5 workers are used. Highest send rate for `createSensor` and is achieved when employing 4 workers.

Name	Workers	TRXN succ	TRXN Fail	Send Rate (TPS)	Avg Latency (s)	Throughput (TPS)
readSensorFixedRate	1	10000	0	280.9	0.01	280.8
readSensorFixedRate	2	10000	0	398	0.03	397.9
readSensorFixedRate	3	9130	869	485	4.23	461.8
readSensorFixedRate	4	7676	2324	485.1	6.08	463.6
readSensorFixedRate	5	7510	2490	486.1	6.54	447.8
updateSensorFixedRate	1	1635	8365	87.8	2.78	86.1
updateSensorFixedRate	2	460	9540	130.2	21.12	82.7
updateSensorFixedRate	3	257	9742	288.8	31.53	83.7
updateSensorFixedRate	4	241	9759	283.4	41.64	96.5
updateSensorFixedRate	5	263	9737	291.1	41.22	123.6
createSensorFixedRate	1	9999	1	81.2	34.53	60.4
createSensorFixedRate	2	9883	117	204.5	87.78	58.8
createSensorFixedRate	3	6007	3992	217.9	64.45	82.4
createSensorFixedRate	4	4728	5272	232.1	60.37	97.6
createSensorFixedRate	5	3236	6764	209.8	47.81	120.9
listSensorFixedRate	1	56	9944	156.5	17.45	106.5
listSensorFixedRate	2	0	10000	218.3	-	142.6
listSensorFixedRate	3	0	9999	260.9	-	146.8
listSensorFixedRate	4	0	10000	272.9	-	149.9
listSensorFixedRate	5	0	10000	276.9	-	151.1
getSensorHistoryFixedRate	1	9216	784	170.7	13.3	143.2
getSensorHistoryFixedRate	2	5729	4271	241	19.99	185.8
getSensorHistoryFixedRate	3	6273	3726	348.5	11.08	317.1
getSensorHistoryFixedRate	4	5318	4682	345.4	15.19	288.9
getSensorHistoryFixedRate	5	4943	5057	391.3	16.07	291.9

Table 7.1: Measurements using fixed-rate rate controller

Caliper creates a new child process for each client, and by increasing the number of workers the process resource overhead increases. Because Caliper and the Fabric network are both running on the same machine, they compete for the same resources. These are probably the reasons for only small increase when adding more than 3 workers. In general, we can see that by increasing amount of workers from 1 to 2 significantly increases send rate.

Another finding is that having a larger number of workers lowers the **transaction success rate**, which was observed in `readSensor` and `createSensor`. `UpdateSensor` transaction success rate decreased from 16.3% to 4.6% when utilizing 2 workers instead of 1, by adding more workers the success rate remained roughly the same. The success rate for `getSensorHistory` continued to fall from 1 to 2 workers, then increased for 3 workers and continue to decrease for more workers. `ListSensor` had the lowest success rate, with just 54 successful transactions while using one worker and none when using more workers.

The table 7.2 summarizes reasons for failed transactions when using fixed-rate rate controller. Logs from each benchmark, which contain error messages, can be found in appendix A, `caliperworkspace/reports/` in corresponding `*.log` files. Docker containers resource utilization was monitored during benchmarking. Indi-

Error/Function	readSensor FixedRate	updateSensor FixedRate	createSensor FixedRate	listSensor FixedRate	getSensorHistory FixedRate
peer=undefined, status=grpc, message=Peer endorsements do not match	NO	**YES**	**YES**	NO	NO
peer=undefined, status=grpc, message=Endorsement has failed	NO	**YES**	**YES**	NO	NO
no endorsement plan available	NO	NO	**YES**	NO	NO
timeout expired while executing transaction	NO	NO	**YES**	**YES**	NO
FabricError: Query failed. Errors: ["Error: 2 UNKNOWN: too many requests for /protos.Endorser, exceeding concurrency limit (2500)"]	**YES**	NO	NO	**YES**	**YES**
TransactionError: Commit of transaction failed on peer peer0.org1.com:7041 with status MVCC_READ_CONFLICT	NO	**YES**	NO	NO	NO

Table 7.2: Reasons for failed transactions summary

vidual reports can be found in appendix A, `caliperworkspace/reports/` in corresponding `*.html` files. Resource utilization summary can be found in appendix A, `caliperworkspace/reports/docker_report.xlsx`

Conclusion

Caliper performance is affected by the number of workers. The highest send rate wasn't achieved with the same amount of workers for every chaincode function. However, we could consider 3 workers to be the optimal number for maximizing the send rate as there is only small increase in send rate when using more than 3 workers. Our highest measured send rates for `readSensor`, `updateSensor`, `createSensor`, `listSensor`, `getSensorHistory` were in the same order 486 TPS, 291 TPS, 232 TPS, 276 TPS, 391 TPS. A potential transaction throughput higher than previously mentioned cannot be measured since Caliper's host machine is unable to generate send transactions faster.

7.2.2 Experiment 2

When evaluating the Fabric network, the aim was to obtain a measurable metric for the Fabric network's performance. The network's transaction throughput is one such metric.

In some cases, the success rate for the first experiment was relatively low. We believe that by using another rate controller we can improve our success rate. Transaction throughput is affected by the success rate, and we want to look at transaction throughput when the majority of transactions are successful.

During this experiment, the **fixed-load rate controller** is used to drive the tests at a target loading (backlog transactions). By changing the driven TPS, this controller

will attempt to maintain a defined backlog of transactions inside the system. As a result, the system achieves the highest possible TPS while maintaining the pending transaction load [16]. Tests were run with 1, 2, 3, 4, 5 workers. The table 7.3 contains these measurements.

Name	Workers	TRXN succ	TRXN Fail	Send Rate (TPS)	Avg Latency (s)	Throughput (TPS)
readSensorFixedLoad	1	10000	0	292.9	0.01	292.8
readSensorFixedLoad	2	10000	0	451.3	0.16	449.6
readSensorFixedLoad	3	9999	0	499.1	1.06	487.4
readSensorFixedLoad	4	10000	0	411.1	1.22	408.5
readSensorFixedLoad	5	10000	0	416.9	1.2	416.7
updateSensorFixedLoad	1	1799	8201	94	2.56	92
updateSensorFixedLoad	2	1303	8697	85.8	5.97	84.1
updateSensorFixedLoad	3	1831	8168	82.8	7.73	80.8
updateSensorFixedLoad	4	2126	7874	76.7	7.41	75.5
updateSensorFixedLoad	5	2717	7283	72.2	7.75	71
createSensorFixedLoad	1	10000	0	65.4	9.21	62
createSensorFixedLoad	2	10000	0	63.2	10.22	62.1
createSensorFixedLoad	3	9999	0	59.6	11.24	58.8
createSensorFixedLoad	4	10000	0	55.9	11.33	55.2
createSensorFixedLoad	5	10000	0	58.1	11.07	57.3
listSensorFixedLoad	1	10000	0	81.3	8.71	76.7
listSensorFixedLoad	2	10000	0	28.4	12.49	28.1
listSensorFixedLoad	3	9999	0	27.4	11.63	26.9
listSensorFixedLoad	4	9005	995	19	9.97	18.8
listSensorFixedLoad	5	5	9995	34.7	0.23	31.4
getSensorHistoryFixedLoad	1	10000	0	176.7	2.92	170.3
getSensorHistoryFixedLoad	2	10000	0	157.1	3.82	153.9
getSensorHistoryFixedLoad	3	9999	0	140.2	4.61	138.7
getSensorHistoryFixedLoad	4	10000	0	133.4	4.24	132.3
getSensorHistoryFixedLoad	5	10000	0	131.1	4.08	130.7

Table 7.3: Measurements using fixed-load rate controller

Observation

As seen from table 7.3 **send rate** is in most cases lower when compared to fixed-rate rate controller, except for some situations in `readSensor`. But **transaction success rate** is much higher. When using fixed-load rate controller transaction send rate is very similar to transaction throughput.

Not every chaincode function has the same **throughput** when using the same amount of workers. `ReadSensor` has highest throughput when using 3 workers, `updateSensor` when using 1 worker, `createSensor` when using 2 workers, `getSensorHistory` when using 1 worker. `ListSensor` benchmark response size grows with amount of workers by the size of 50 sensors, which means that when using 1 worker response is array with 50 sensors, but when using 5 workers response is array with 250 sensors. This probably explains why throughput is the highest for 1 worker and low success rate for 5 workers.

The table 7.4 summarizes reasons for failed transactions when using fixed-load rate controller. `MVCC_READ_CONFLICTS` cannot be easily removed and would require structural changes to prevent multiple attempts to update value of the same key. Logs from each benchmark, which contain error messages, can be found in appendix A, `caliperworkspace/reports/` in corresponding `*.log` files. Docker containers resource utilization was monitored during benchmarking. Individual reports can be found in appendix A, `caliperworkspace/reports/` in corresponding `*.html` files. Resource utilization summary can be found in appendix A, `caliperworkspace/reports/docker_report.xlsx`

Error/function	readSensor FixedLoad	updateSensor FixedLoad	createSensor FixedLoad	listSensor FixedLoad	getSensorHistory FixedLoad
peer=undefined, status=grpc, message=Peer endorsements do not match	NO	**YES**	NO	NO	NO
peer=undefined, status=grpc, message=Endorsement has failed	NO	NO	NO	NO	NO
no endorsement plan available	NO	NO	NO	NO	NO
timeout expired while executing transaction	NO	NO	NO	**YES**	NO
FabricError: Query failed. Errors: ["Error: 2 UNKNOWN: too many requests for /protos.Endorser, exceeding concurrency limit (2500)"]	NO	NO	NO	NO	NO
TransactionError: Commit of transaction failed on peer peer0.org1.com:7041 with status MVCC_READ_CONFLICT	NO	**YES**	NO	NO	NO

Table 7.4: Reasons for failed transactions summary

Conclusion

The choice of rate controller affects success rate, send rate and throughput. **Transaction success rate** is much higher when using fixed-load rate controller and multiple errors were eliminated. The highest throughput wasn't achieved with the same amount of workers for every chaincode function. Our highest measured throughput for `readSensor`, `updateSensor`, `createSensor`, `listSensor`, `getSensorHistory` were respectively 487 TPS, 92 TPS, 62 TPS, 76 TPS, 170 TPS.

7.3 Security of proposed solution

The suggested design's security features are derived from two main sources. The first is the security features inherited from Hyperledger Fabric. The second layer of security is provided by design choices made to guarantee privacy, access control, and data provenance.

The system gains following security features by being built on top of the Hyperledger Fabric blockchain network model [32]:

- **Immutability** - The ordering service must sign blocks before they can be added to the ledger. After that, transactions are sorted in blocks and sent to peer nodes, which can validate the blocks by having access to the ordering node's CAs. If orderer is compromised, the malicious party will have a different ledger than other peers and that is not enough to convince other peers that their version is the right one. Endorsement policy that requires all parties to sign the endorsement ensures that the states of the participants match.
- **Privacy and Confidentiality** - Privacy aspects such as asymmetric cryptography and zero-knowledge proofs serve to separate transaction data from ledger records. As a result, the data is protected from the underlying algorithm. Thus, the orderers have no knowledge of the transaction data. To prevent fraudulent organizations from joining the blockchain and stealing user privacy, all organizations joining the blockchain must be authenticated at the CA. MSP also separates roles between different organizations.
- **Consensus** - Consensus is in charge of checking the correctness of all transactions in a block and agreeing on an order for them. Every node on the channel is guaranteed to process the transactions in the same manner, and every non-faulty node on the channel should eventually receive the submitted transactions. Consensus is a sophisticated process in Hyperledger Fabric, and it is present in the transaction flow mechanism outlined in section 3.1.

The following assumptions are considered when analyzing the security aspects of the proposed system:

(i) Data measurements performed by smart meters are correct.

(ii) No human errors when entering data are considered as this is out of scope of this thesis.

(iii) Sufficient amount of honest peers and orderers is present in blockchain network.

(iv) Correct endorsement policy is enforced.

The developed solution achieves, based on the aforementioned assumptions, the following security features:

1. **Separation from uninvolved** - participants must have a valid certificate issued by MSP services to access developed system and its data.
2. **Asset traceability/auditability/provenance** - each activity done on the ledger is recorded and kept in the asset's state. This allows to track all changes up to asset's creation.
3. **Soundness** - endorsement policy requires majority of organizations to execute chaincode and endorse the execution results in order for the transaction to be considered valid. This is fulfilled, because we assumed sufficient amount of honest nodes.
4. **States validity** - business logic implemented in smart contracts ensures that meters do not end up in illegal states by performing checks on the validity of the operations in smart contracts.
5. **Restricted data modification** - To alter an asset, it must be part of a valid transaction that takes access and business requirements into consideration.
6. **Accountability** - Organizations are not permitted to act on behalf of other participants. Because Fabric is permissioned, the identity of the caller is checked and validated for each operation, and the transaction is signed by him. It is possible to prove that he submitted the corresponding transaction using Fabric's PKI.
7. **Dispute validity** - it is really hard to open disputes when all interested parties have access to the same smart meter history, faking a problem with smart meter becomes increasingly difficult.

The system also mitigates common threats by being built on top of the Hyperledger Fabric. In the table 7.5, we will briefly review the most common security threats and how to address them.

Threat with description	**Hyperledger Fabric**	**Network/Node Operator**
Spoofing - Using credentials to impersonate a legitimate user or compromising a user's private key.	Generates X.509 certificates for its users.	Manages the distribution of certificate revocation lists across network participants to guarantee that revoked users can no longer access the system.
Tampering - Modification of information.	To make tampering unfeasible, cryptographic methods are used.	Derived from Fabric.

Table 7.5 continued from previous page

Threat with description	Hyperledger Fabric	Network/Node Operator
Repudiation - It is impossible for an entity to deny who did what.	Uses digital signatures to keep track of who did what.	Derived from Fabric.
Replay attacks - Replaying transactions to corrupt the ledger.	To validate the transaction, read/write sets are used. A transaction replay will fail because of to an invalid read set.	Derived from Fabric.
Information disclosure - Data exposed through intentional breach or accidental exposure.	Allows for the use of TLS for in-transit encryption.	The operator's responsibility is to avoid information disclosure by adhering to information security best practices for Fabric nodes as well as applications that communicate with the ledger. Organizations must specify the right Fabric mechanisms for sharing data as part of data governance, especially when adding new users or apps.
Denial of service - Making genuine users' access to the system difficult.	The operator's responsibility.	The operator's duty to prevent denial of service to the system.
Elevation of privileges - Obtaining high-level application access.	Without a manual review of access, issued identities cannot be upgraded.	It is the network/node operator's duty to audit a smart contract, limit access, and operate smart contract containers with proper limitations. (Typically Docker containers in Fabric deployments.)
Ransomware - Using cryptography or other methods to block access to filesystem data.	The compromising of one peer in a distributed system should not have an impact on the data integrity of other peers.	The operator's duty is to ensure that ransomware does not prevent access to a node's ledger.

Table 7.5: Threats [12]

Conclusion

Blockchain technology is a game-changing breakthrough that has the potential to significantly enhance many existing systems by making them more transparent, secure, and efficient.

Throughout this thesis, we described relevant aspects of blockchain technology, with focus on Hyperledger Fabric. Next, we provided a review on selected academic papers in the energy sector. More importantly, the focus was put on the applicability of enterprise blockchain technology in the energy sector. We present a use-case, which considers consumption data sharing between multiple organizations and final consumer.

We designed and implemented an enterprise blockchain-based system focused on aforementioned use-case. The designed trustworthy energy consumption monitoring solution consists of two main components. Hyperledger Fabric network responsible for business logic and providing desired characteristics such as immutability, security, and traceability, with all chaincode functions unit tested. And applications for organizations responsible for exposing the blockchain component functionalities.

Evaluation consisted of flow validation, performance and security analysis. We conducted several research efforts with the purpose of better understanding blockchain network performance.

Performance test were developed using framework Hyperledger Caliper while aim of the first experiment was to determine Caliper's highest possible actual send rate. During this experiment we found out that Caliper performance is affected by amount of workers and every chaincode function has different highest send rate when using different amount of workers. During first experiment when using only basic controller we also noticed low success rate. We realized that using only TPS as metric is not enough and we need to consider also success rate.

During the second experiment we measure transaction throughput of blockchain network, which is affected by the success rate when the majority of transactions are

successful. We found that, when using different rate controller we were able to improve success rate. Our highest measured throughput for `readSensor`, `updateSensor`, `createSensor`, `listSensor`, `getSensorHistory` were respectively 487 TPS, 92 TPS, 62 TPS, 76 TPS, 170 TPS.

As for future work, we could perform more complex performance experiments, provide formal definition of MVCC and explain why MVCC read conflicts occur. Redesign solution to eliminate MVCC read conflicts which would enable realtime monitoring.

Additional research activities, trials and projects will demonstrate if blockchain can achieve its full potential, prove economic feasibility, and eventually be embraced in the mainstream.

Resumé

Táto práca sa zaoberá zabezpečením spoľahlivého a pravidelného odpočtu v oblasti distribúcie tepelnej energie. Problém spočíva v zabezpečení auditovateľnosti, bezpečnosti a nemennosti dát pri rádiovom zbere dát o spotrebe pomocou senzorov. Ako jedno z možných riešení sa ukázal sofistikovaný monitorovací systém s využitím blockchain technológie. Výskum sa zaoberá otázkou, či je blockchain aplikovateľný na riešenie správy transakcií pri monitorovaní spotreby v energetike.

Kapitola 1 poskytuje prehľad technológie blockchain a dôležitých teoretických konceptov. Túto kapitolu uzatvára pohľad na súčasný stav v oblasti aplikovateľnosti blockchainu na dosiahnutie dôvery v energetickom sektore.

Kapitola 2 vysvetľuje použitie technológie blockchain v podnikovom prostredí. Obsahuje porovnanie verejných a podnikových (enterprise) blockchainov. Popisuje a porovnáva blockchain frameworky Quorum, Corda a Hyperledger Fabric. Kapitola 3 sa zameriava na architektúru a koncepty frameworku Hyperledger Fabric.

Proces v praxi

Vzhľadom na nedostatok verejne dostupných informácií o procesoch inštalácie a monitorovania meračov tepla, nasledujúca situácia nemusí nevyhnutne vystihovať realitu.

Merač tepla **distribučnej spoločnosti** je spoločný pre celú bytovku. **Distribučná spoločnosť** si účtuje teplo, ktoré prejde fakturačným meradlom na vstupe do bytovky. Keďže budova sa považuje za jedného spotrebiteľa tepla, zákazníkom **distribučnej spoločnosti** je **správca**, nie jednotlivé domácnosti. **Správcovia** (správcovské spoločnosti, bytové družstvá a spoločenstvá vlastníkov bytov) následne rozúčtovávajú náklady pre **koncových spotrebiteľov** podľa ich konkrétnej spotreby. Na to, aby rozúčtovanie bolo čo najspravodlivejšie, **správca** nainštaluje pomerové merače tepla do jednotlivých bytov.

Pri **klasickom pochôdzkovom fyzickom** odpočte je potrebné, aby užívatelia

bytov boli v určitý deň a hodinu doma a čakali na zamestnanca **správcovskej spoločnosti**, ktorý **fyzicky vykoná** odpočet spotreby tepla z meračov umiestnených v byte. Navyše, k informáciám o spotrebe tepla sa väčšina vlastníkov bytov dostane až v ročnom vyúčtovaní.

V súčasnosti **správcovské spoločnosti, bytové družstvá a spoločenstvá vlastníkov bytov** prechádzajú na vyspelé rádiové technológie, kde nie je potrebné manuálne zadávanie meraní do systému. Zároveň chráni súkromie užívateľov bytov, eliminuje časť chýb pri odpočtoch a neoprávnenú manipuláciu s meračmi. Umožňuje realizáciu odpočtov kedykoľvek v priebehu rozúčtovacieho obdobia, pričom vlastník či správca môže aktuálne údaje sledovať cez internetový portál. Na základe jednoduchého prístupu k informáciám o spotrebe, môže užívateľ lepšie regulovať svoje spotrebiteľské správanie a tým optimalizovať náklady na bývanie.

Nežiaduce stavy možno vďaka rádiovej technológii včas odhaliť a vyhnúť sa nepríjemným prekvapeniam v ročnom vyúčtovaní. Kontrola spotreby cez internetový portál, notifikácie a hlásenia umožňujú koncovým užívateľom a správcom vykonať okamžitú nápravu.

Pri implementácii rádiového riešenia treba zohľadniť aj spôsob zabezpečenia dát a ochrany osobných údajov. Smernica Európskeho parlamentu a Rady 2012/27/EÚ o energetickej efektívnosti, ktorej požiadavky sa postupne implementujú do legislatívy Slovenskej republiky, má za cieľ zabezpečiť koncovým odberateľom informácie o spotrebe tepla a teplej vody v kratších časových intervaloch, aby mohli na uvedenú spotrebu rýchlejšie reagovať a prípadne ju včas regulovať. Smernica (článok 10 Informácie o vyúčtovaní) zaväzuje členské štáty zabezpečiť pre koncových odberateľov, aby údaje o vyúčtovaní boli presné a založené na skutočnej spotrebe, a umožniť im jednoduchý prístup k doplňujúcim informáciám o histórii spotreby. [kapitola 4]

Postup zdieľaného procesu

V súčasnom procese dochádza k problémom s dôverou. Koncový odberateľ tepla je odkázaný dôverovať odpočtom správcovskej spoločnosti. Správcovská spoločnosť je odkázaná dôverovať odpočtom hlavného merača distribučnej spoločnosti.

Vývoj prototypu sa spočiatku zameriava na komunikáciu medzi nasledujúcimi účastníkmi:

- Správcovská spoločnosť (Property management company), ktorá spravuje určitý

počet nehnuteľností.

- Distribučná spoločnosť (Heating distributor), ktorá distribuuje teplo do daných nehnuteľností.
- Koncový odberateľ tepla (Final consumer)

V tomto prípade použitia vidíme, že sú potrebné zdieľané údaje, prísny audit (účtovanie), potreba zabezpečiť dôvernosť transakcií (nemôže byť verejná), účastníci musia byť známi (vedieť, komu účtovať). Blockchain je použitý na zvýšenie odolnosti systému voči neoprávnenej manipulácii. Spracovaním údajov z meračov pomocou blockchainu a smart kontraktov sa vytvorí dôvera medzi jednotlivými účastníkmi, pretože budú mať prístup k jedinému zdroju dát (single source of truth). Detailne popísaný proces sa nachádza v sekcii 4.2.

Architektúra a implementácia systému

Kapitola 5 predstavuje navrhnuté riešenie monitorovania spotreby energie, ktoré pozostáva z nasledujúcich komponentov.

- **Hyperledger Fabric sieť** zodpovedná za business logiku a poskytovanie požadovaných vlastnosti, ako je nemennosť, bezpečnosť a vysledovateľnosť.

  Naša blockchainová sieť pozostáva z dvoch organizácií reprezentujúcich správcovskú spoločnosť a distribučná spoločnosť. Sieť obsahuje aj jedného individuálneho účastníka, ktorým je koncový odberateľ tepla. Správcovská spoločnosť reprezentuje entitu správcovskej spoločnosti aj koncových spotrebiteľov.

  Jeden peer v rámci organizácie postačuje na udržiavanie repliky účtovnej knihy (ledger) a vykonávanie smart kontraktov. Okrem peerov naša sieť obsahuje jednu koreňovú (root) CA, dve intermediárne CA (jednu pre Org1 a jednu pre Org2 fungujúce ako MSP) a zoraďovaciu službu (ordering service). Zoraďovacia služba beží v jednej organizácii. Popísaná blockchainová sieť je znázornená na obrázku 5.11.

- **Aplikácie pre organizácie**, ktoré umožňujú používanie funkcií blockchain komponentu.

  Životný cyklus aplikácie sa začína registráciou používateľa na MSP (Fabric CA serveri) organizácie. Tým sa získajú poverenia (credentials), ktoré

môžu byť uložené v peňaženkách na disku alebo v databáze. Aplikácia `PropertyManagementOrg` tiež rozlišuje používateľov na základe ich rolí v rámci organizácie. Pravidlá ACL našich smart kontraktov vyžadujú, aby boli tieto rôzne roly špecifikované v atribútoch certifikátu.

Používatelia majú prístup k rovnakému REST API na webovom serveri, ktoré obsahuje funkcie pre registráciu, prihlásenie a funkcie smart kontraktov. Modul autentifikácie používateľov (User Authentication) a manažér relácií (Session Manager) obmedzuje prístup k rôznym funkciám REST API a riadi vytváranie identít a rolí používateľov. Tento modul tiež udržiava relácie pomocou webových tokenov JSON, čo umožňuje prihlásenému používateľovi vykonávať viaceré operácie s dynamicky generovaným tokenom.

Implementácia Hyperledger Fabric siete a aplikácii pre organizácie je detailne popísaná v kapitole 6.

Vyhodnotenie

Vyhodnotenie aplikácie spočíva v nasledujúcich aktivitách:

- **Overenie toku dát**, ktoré spočíva v kontrole jednotlivých krokov procesu popísaného v sekcii 4.2, pričom dochádza k prepínaniu medzi jednotlivými užívateľmi [kapitola 7.1].
- **Vyhodnotenie bezpečnosti** kde sa formou diskusie prechádzajú jednotlivé bezpečnostné aspekty riešenia. Bezpečnostné prvky navrhovaného dizajnu sú odvodené z dvoch hlavných zdrojov. Prvým sú bezpečnostné prvky zdedené z Hyperledger-u Fabric. Druhým zdrojom sú architektonické rozhodnutia, ktoré garantujú súkromie, kontrolu prístupu a pôvod údajov [kapitola 7.3].
- **Vyhodnotenie výkonu implementovaného riešenia** sa zameriava na nasledujúce výskumné aktivity: (1) výber nástroja na meranie výkonu smart kontraktov, (2) určenie najvyššej možnej skutočnej rýchlosti odosielania (actual send rate) vybraného nástroja (Caliper), (3) odhad priepustnosti blockchainovej siete, pričom sa snažíme minimalizovať počet neúspešných transakcií, (4) monitorovanie hardvérových prostriedkov využívaných počas experimentov. Experimenty sú zamerané na rýchlosť generovania a posielania požiadaviek v závislosti od počtu lokálnych Caliper klientov a na minimalizáciu neúspešných transakcií pri

zachovaní čo najvyššej priepustnosti distribuovanej blockchainovej siete [kapitola 7.2].

Výkon bol testovaný pomocou frameworku Hyperledger Caliper, pričom cieľom prvého experimentu bolo zistiť maximálnu možnú rýchlosť generovania a posielania požiadaviek (actual send rate). Počas tohto experimentu sme zistili, že výkon Caliperu je ovplyvnený počtom lokálnych Caliper klientov. Každá testovaná funkcia smart kontraktu dosiahne rôznu najvyššiu prenosovú rýchlosť (send rate) pri použití rôzneho počtu lokálnych Caliper klientov. V prvom experimente pri použití iba základného ovládača rýchlosti (rate controller) sme tiež zaznamenali nízku úspešnosť (success rate). Uvedomili sme si, že použitie iba TPS ako metriky nestačí a je potrebné brať do úvahy aj ďalšiu metriku, úspešnosť (success rate) [kapitola 7.2.1].

Počas druhého experimentu meriame priepustnosť (throughput) distribuovanej blockchainovej siete, ktorá je ovplyvnená mierou úspešnosti (success rate). Zistili sme, že pri použití iného ovládača rýchlosti (rate controller) sme zlepšili mieru úspešnosti (success rate). Naša najvyššia nameraná priepustnosť pre `readSensor`, `updateSensor`, `createSensor`, `listSensor`, `getSensorHistory` bola 487 TPS, 92 TPS, 62 TPS, 76 TPS, 170 TPS [kapitola 7.2.2].

Do budúcna je prácu možné rozšíriť o komplexnejšie výkonnostné experimenty, poskytnutie formálnej definície MVCC a vysvetlenie, prečo dochádza ku konfliktom čítania MVCC (MVCC read conflicts). Upraviť navrhnuté riešenie systému tak, aby došlo k odstráneniu konfliktov pri čítaní MVCC, čo by umožnilo monitorovanie v reálnom čase.

Bibliography

[1] *Apache CouchDB*. URL: https://couchdb.apache.org/ [visited on 04. 01. 2022].

[2] *Azure blockchain solutions*. URL: https://azure.microsoft.com/en-gb/solutions/blockchain/ [visited on 04. 01. 2022].

[3] BAMBARA, J. J. et al. *Blockchain: A Practical Guide to Developing Business, Law, and Technology Solutions*. McGraw-Hill, February 2018. ISBN: 978-1-260-11586-4.

[4] BARGER, A. et al. *A Byzantine Fault-Tolerant Consensus Library for Hyperledger Fabric*. URL: https://arxiv.org/pdf/2107.06922.pdf [visited on 15. 02. 2022].

[5] BASHIR, I. *Mastering Blockchain. A deep dive into distributed ledgers, consensus protocols, smart contracts, DApps, cryptocurrencies, Ethereum, and more*. Third Edition. Birmingham-Mumbai: Packt Publishing, 2020. ISBN: 978-1-83921-319-9.

[6] *Blockchain on AWS*. URL: https://aws.amazon.com/blockchain/ [visited on 04. 01. 2022].

[7] CHACKO, J. A., MAYER, R. and JACOBSEN, H.-A. *Why Do My Blockchain Transactions Fail? A Study of Hyperledger Fabric (Extended version)**. 2021. URL: https://arxiv.org/abs/2103.04681 [visited on 04. 01. 2022].

[8] *Chai*. URL: https://www.chaijs.com/ [visited on 05. 01. 2022].

[9] *Corda*. URL: https://github.com/corda/corda [visited on 04. 01. 2022].

[10] *Directive 2012/27/EU of the European parliament and of the council of 25 October 2012. on energy efficiency, amending Directives 2009/125/EC and 2010/30/EU and repealing Directives 2004/8/EC and 2006/32/EC (Text with*

EEA relevance). 2019. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32012L0027\&from=EN.

[11] *Fablo. Source code*. URL: https://github.com/hyperledger-labs/fablo [visited on 05. 01. 2022].

[12] GAUR, N. et al. *Blockchain with Hyperledger Fabric. Build decentralized applications using Hyperledger Fabric 2*. Second Edition. BIRMINGHAM - MUMBAI: Packt Publishing, 2020. ISBN: 978-1-83921-875-0.

[13] GUR, A. O., OKSUZER, S. and KARAARSLAN, E. "Blockchain Based Metering and Billing System Proposal with Privacy Protection for the Electric Network". In: *2019 7th International Istanbul Smart Grids and Cities Congress and Fair (ICSG)*. IEEE, 2019, pp. 204–208. ISBN: 978-1-7281-1315-9. DOI: 10.1109/SGCF.2019.8782375. URL: https://ieeexplore.ieee.org/document/8782375/ [visited on 04. 01. 2022].

[14] *Hyperledger Blockchain Performance Metrics*. URL: https://www.hyperledger.org/wp-content/uploads/2018/10/HL_Whitepaper_Metrics_PDF_V1.01.pdf [visited on 05. 01. 2022].

[15] *Hyperledger Caliper. Source code*. URL: https://github.com/hyperledger/caliper [visited on 05. 01. 2022].

[16] *Hyperledger Caliper. rate controllers*. URL: https://hyperledger.github.io/caliper/v0.4.2/rate-controllers [visited on 05. 01. 2022].

[17] *Hyperledger Composer. Source code*. URL: https://github.com/hyperledger-archives/composer [visited on 05. 01. 2022].

[18] *Hyperledger Explorer. Source code*. URL: https://github.com/hyperledger/blockchain-explorer [visited on 05. 01. 2022].

[19] *Hyperledger Fabric. Source code*. URL: https://github.com/hyperledger/fabric [visited on 05. 01. 2022].

[20] *Hyperledger Fabric v2. lifecycle image*. URL: https://programmer.help/images/blog/608a49b858e8d91821b7009d4354ead5.jpg [visited on 05. 01. 2022].

[21] *Hyperledger Fabric v2.2 documentation. Test network*. 2021. URL: https://hyperledger-fabric.readthedocs.io/en/release-2.2/test_network.html [visited on 04. 01. 2022].

[22] *Hyperledger Fabric v2.3. The Ordering Service.* URL: https://hyperledger-fabric.readthedocs.io/en/release-2.3/orderer/ordering_service.html [visited on 15. 02. 2022].

[23] *Hyperledger Fabric v2.3. Sample configurations.* URL: https://github.com/hyperledger/fabric/tree/release-2.3/sampleconfig/ [visited on 05. 01. 2022].

[24] *Hyperledger Fabric v2.3 documentation. Chaincode lifecycle.* URL: https://hyperledger-fabric.readthedocs.io/en/release-2.3/chaincode_lifecycle.html [visited on 04. 01. 2022].

[25] *Hyperledger Fabric v2.3 documentation. Peers.* URL: https://hyperledger-fabric.readthedocs.io/en/release-2.3/peers/peers.html [visited on 05. 01. 2022].

[26] *Hyperledger Whitepaper.* URL: https://docs.google.com/document/d/1Z4M_qwILLRehPbVRUsJ3OF8Iir-gqS-ZYe7W-LE9gnE/edit\#heading=h.m6iml6hqrnm2 [visited on 04. 01. 2022].

[27] *IBM Blockchain Platform.* URL: https://www.ibm.com/uk-en/cloud/blockchain-platform [visited on 04. 01. 2022].

[28] *Jq. a lightweight and flexible command-line JSON processor.* URL: https://stedolan.github.io/jq/ [visited on 05. 01. 2022].

[29] *LevelDB.* URL: https://github.com/google/leveldb [visited on 04. 01. 2022].

[30] *Linux Foundation.* URL: https://www.linuxfoundation.org/ [visited on 04. 01. 2022].

[31] LOMBARDO, H. *Using Blockchains for IoT in Facilities Management.* 2016. URL: https://www.chainofthings.com/news/2016/8/8/blockchains-ideal-for-iot-in-facilities-management [visited on 05. 01. 2022].

[32] MA, C. et al. "The privacy protection mechanism of Hyperledger Fabric and its application in supply chain finance". In: *Cybersecurity* 2.1 (2019). ISSN: 2523-3246. DOI: 10.1186/s42400-019-0022-2. URL: https://cybersecurity.springeropen.com/articles/10.1186/s42400-019-0022-2 [visited on 05. 01. 2022].

[33] MENGELKAMP, E. et al. “Designing microgrid energy markets”. In: *Applied Energy* 210 (2018), pp. 870–880. ISSN: 0306-2619. DOI: 10.1016/j.apenergy.2017.06.054. URL: https://linkinghub.elsevier.com/retrieve/pii/S030626191730805X [visited on 04. 01. 2022].

[34] *Microgrid-Blockchain-Project*. URL: https://github.com/guibvieira/Microgrid-Blockchain-Project [visited on 04. 01. 2022].

[35] *Mocha*. URL: https://mochajs.org/ [visited on 05. 01. 2022].

[36] O’HARA, K. *Data Trusts: Ethics, Architecture and Governance for Trustworthy Data Stewardship*. 2019. URL: https://eprints.soton.ac.uk/428276/1/WSI_White_Paper_1.pdf [visited on 15. 02. 2022].

[37] *Oracle Blockchain Platform Cloud Service*. URL: https://www.oracle.com/uk/application-development/cloud-services/blockchain-platform/ [visited on 04. 01. 2022].

[38] POP, C. et al. “Blockchain-Based Scalable and Tamper-Evident Solution for Registering Energy Data”. In: *Sensors* 19.14 (2019). ISSN: 1424-8220. DOI: 10.3390/s19143033. URL: https://www.mdpi.com/1424-8220/19/14/3033 [visited on 04. 01. 2022].

[39] QI, Y. et al. “Research of Energy Consumption Monitoring System Based on IoT and Blockchain Technology”. In: *Journal of Physics: Conference Series* 1757.1 (2021-01-01). ISSN: 1742-6588. DOI: 10.1088/1742-6596/1757/1/012154. URL: https://iopscience.iop.org/article/10.1088/1742-6596/1757/1/012154 [visited on 04. 01. 2022].

[40] *Quorum*. URL: https://github.com/ConsenSys/quorum [visited on 04. 01. 2022].

[41] *R3 trust technology*. URL: https://www.r3.com/trust-technology/ [visited on 04. 01. 2022].

[42] *Sinon.JS*. URL: https://sinonjs.org/ [visited on 05. 01. 2022].

[43] *SmartBFT. Integration of the BFT consensus library into Fabric*. URL: https://github.com/SmartBFT-Go/fabric [visited on 15. 02. 2022].

[44] *SmartBFT. Java SDK for Hyperledger Fabric*. URL: https://github.com/SmartBFT-Go/fabric-sdk-java [visited on 15. 02. 2022].

[45] ZAND, M., WU, X. and MORRIS, M. A. *Hands-On Smart Contract Development with Hyperledger Fabric V2*. O'Reilly Media, Inc., September 2021. ISBN: 978-1-4920-8612-3.

[46] ZHANG, R., XUE, R. and LIU, L. "Security and Privacy on Blockchain". In: *ACM Computing Surveys* 52.3 (2020-05-31), pp. 1–34. ISSN: 0360-0300. DOI: 10.1145/3316481. URL: https://dl.acm.org/doi/10.1145/3316481 [visited on 04. 01. 2022].

Appendices

Appendix A: CD medium - an electronic copy of this thesis, source code of final solution together with user manual.

A CD medium

CD contents:

```
blockchain_based_enterprise_applications_revicky.pdf
diploma_thesis_project/
├── README.md
├── README_before_fablo_development_deployment.md
├── README_caliper_benchmark_development.md
├── README_fablo_development_deployment.md
├── apps
│   ├── org1backendapp
│   │   ├── package.json
│   │   ├── README.md
│   │   ├── org1.example.com
│   │   │   └── connection-org1.json
│   │   └── src
│   │       ├── app.js
│   │       ├── app.spec.js
│   │       ├── controllers
│   │       │   ├── identityController.js
│   │       │   └── sensorController.js
│   │       ├── models
│   │       │   ├── identityModel.js
│   │       │   └── sensorModel.js
│   │       └── routes
│   │           └── routes.js
│   └── org2backendapp
│       ├── package-lock.json
│       ├── package.json
│       ├── README.md
│       ├── org2.example.com
│       │   └── connection-org2.json
│       └── src
│           ├── app.js
│           ├── app.spec.js
│           ├── controllers
│           │   ├── identityController.js
│           │   └── sensorController.js
│           ├── models
│           │   ├── identityModel.js
│           │   └── sensorModel.js
│           └── routes
│               └── routes.js
└── chaincode-typescript
    ├── package-lock.json
    ├── package.json
    ├── tsconfig.json
    ├── tslint.json
    ├── META-INF
    │   └── statedb
    │       └── couchdb
    │           └── indexes
    │               ├── approverIndex.json
    │               ├── consumerIndex.json
    │               └── finalConsumerIndex.json
```

```
│   ├── src
│   │   ├── emptyObject.ts
│   │   ├── index.ts
│   │   ├── sensor-contract.spec.ts
│   │   ├── sensor-contract.ts
│   │   ├── sensorAgreement.ts
│   │   ├── sensorAgreementHistory.ts
│   │   └── sensorAgreementStatus.ts
│   └── fablo-network-generator
│       ├── fablo-config.json
│       └── fablo-target
│           ├── fabric-docker.sh
│           ├── fabric-config
│           │   ├── configtx.yaml
│           │   ├── crypto-config-orderer.yaml
│           │   ├── crypto-config-org1.yaml
│           │   └── crypto-config-org2.yaml
│           ├── fabric-docker
│           │   ├── .env
│           │   ├── channel-query-scripts.sh
│           │   ├── commands-generated.sh
│           │   ├── docker-compose.yaml
│           │   └── scripts
│           │       ├── base-functions.sh
│           │       ├── base-help.sh
│           │       ├── chaincode-functions.sh
│           │       ├── channel-query-functions.sh
│           │       └── cli
│           │           └── channel_fns.sh
│           └── hooks
│               └── post-generate.sh
caliperworkspace/
├── package-lock.json
├── package.json
├── benchmarks
│   └── thesis_allSensorOperationsBenchmark.yaml
├── networks
│   └── networkConfig.json
└── reports
    ├── docker_report.xlsx
    ├── caliper-createSensorFixedLoad-1-worker.log
    ├── caliper-createSensorFixedLoad-2-worker.log
    ├── caliper-createSensorFixedLoad-3-worker.log
    ├── caliper-createSensorFixedLoad-4-worker.log
    ├── caliper-createSensorFixedLoad-5-worker.log
    ├── caliper-createSensorFixedRate-1-worker.log
    ├── caliper-createSensorFixedRate-2-worker.log
    ├── caliper-createSensorFixedRate-3-worker.log
    ├── caliper-createSensorFixedRate-4-worker.log
    ├── caliper-createSensorFixedRate-5-worker.log
    ├── caliper-getSensorHistoryFixedLoad-1-worker.log
    ├── caliper-getSensorHistoryFixedLoad-2-worker.log
    ├── caliper-getSensorHistoryFixedLoad-3-worker.log
    ├── caliper-getSensorHistoryFixedLoad-4-worker.log
    ├── caliper-getSensorHistoryFixedLoad-5-worker.log
    ├── caliper-getSensorHistoryFixedRate-1-worker.log
    ├── caliper-getSensorHistoryFixedRate-2-worker.log
    ├── caliper-getSensorHistoryFixedRate-3-worker.log
    ├── caliper-getSensorHistoryFixedRate-4-worker.log
    ├── caliper-getSensorHistoryFixedRate-5-worker.log
    ├── caliper-listSensorFixedLoad-1-worker.log
    ├── caliper-listSensorFixedLoad-2-worker.log
    ├── caliper-listSensorFixedLoad-3-worker.log
    ├── caliper-listSensorFixedLoad-4-worker.log
    ├── caliper-listSensorFixedLoad-5-worker.log
    ├── caliper-listSensorFixedRate-1-worker.log
    ├── caliper-listSensorFixedRate-2-worker.log
    ├── caliper-listSensorFixedRate-3-worker.log
    └── caliper-listSensorFixedRate-4-worker.log
```

```
├── caliper-listSensorFixedRate-5-worker.log
├── caliper-readSensorFixedLoad-1-worker.log
├── caliper-readSensorFixedLoad-2-worker.log
├── caliper-readSensorFixedLoad-3-worker.log
├── caliper-readSensorFixedLoad-4-worker.log
├── caliper-readSensorFixedLoad-5-worker.log
├── caliper-readSensorFixedRate-1-worker.log
├── caliper-readSensorFixedRate-2-worker.log
├── caliper-readSensorFixedRate-3-worker.log
├── caliper-readSensorFixedRate-4-worker.log
├── caliper-readSensorFixedRate-5-worker.log
├── caliper-updateSensorFixedLoad-1-worker.log
├── caliper-updateSensorFixedLoad-2-worker.log
├── caliper-updateSensorFixedLoad-3-worker.log
├── caliper-updateSensorFixedLoad-4-worker.log
├── caliper-updateSensorFixedLoad-5-worker.log
├── caliper-updateSensorFixedRate-1-worker.log
├── caliper-updateSensorFixedRate-2-worker.log
├── caliper-updateSensorFixedRate-3-worker.log
├── caliper-updateSensorFixedRate-4-worker.log
├── caliper-updateSensorFixedRate-5-worker.log
├── createSensorFixedLoad-1-worker.html
├── createSensorFixedLoad-2-worker.html
├── createSensorFixedLoad-3-worker.html
├── createSensorFixedLoad-4-worker.html
├── createSensorFixedLoad-5-worker.html
├── createSensorFixedRate-1-worker.html
├── createSensorFixedRate-2-worker.html
├── createSensorFixedRate-3-worker.html
├── createSensorFixedRate-4-worker.html
├── createSensorFixedRate-5-worker.html
├── getSensorHistoryFixedLoad-1-worker.html
├── getSensorHistoryFixedLoad-2-worker.html
├── getSensorHistoryFixedLoad-3-worker.html
├── getSensorHistoryFixedLoad-4-worker.html
├── getSensorHistoryFixedLoad-5-worker.html
├── getSensorHistoryFixedRate-1-worker.html
├── getSensorHistoryFixedRate-2-worker.html
├── getSensorHistoryFixedRate-3-worker.html
├── getSensorHistoryFixedRate-4-worker.html
├── getSensorHistoryFixedRate-5-worker.html
├── listSensorFixedLoad-1-worker.html
├── listSensorFixedLoad-2-worker.html
├── listSensorFixedLoad-3-worker.html
├── listSensorFixedLoad-4-worker.html
├── listSensorFixedLoad-5-worker.html
├── listSensorFixedRate-1-worker.html
├── listSensorFixedRate-2-worker.html
├── listSensorFixedRate-3-worker.html
├── listSensorFixedRate-4-worker.html
├── listSensorFixedRate-5-worker.html
├── readSensorFixedLoad-1-worker.html
├── readSensorFixedLoad-2-worker.html
├── readSensorFixedLoad-3-worker.html
├── readSensorFixedLoad-4-worker.html
├── readSensorFixedLoad-5-worker.html
├── readSensorFixedRate-1-worker.html
├── readSensorFixedRate-2-worker.html
├── readSensorFixedRate-3-worker.html
├── readSensorFixedRate-4-worker.html
├── readSensorFixedRate-5-worker.html
├── updateSensorFixedLoad-1-worker.html
├── updateSensorFixedLoad-2-worker.html
├── updateSensorFixedLoad-3-worker.html
├── updateSensorFixedLoad-4-worker.html
├── updateSensorFixedLoad-5-worker.html
├── updateSensorFixedRate-1-worker.html
├── updateSensorFixedRate-2-worker.html
├── updateSensorFixedRate-3-worker.html
└── updateSensorFixedRate-4-worker.html
```

```
│   ├── updateSensorFixedRate-5-worker.html
└── workload
    ├── createSensor.js
    ├── getSensorHistory.js
    ├── listSensor.js
    ├── readSensor.js
    └── updateSensor.js
```