UNIVERSITY OF BERGEN
DEPARTMENT OF INFORMATICS

Master's thesis

Improve auditing and privacy of electronic health records by using blockchain technology

Author: Kjell-Erik H. Marstein

Supervisor: Chunlei Li

June 3, 2019

Abstract

An ever-increasing amount of sensitive patient data is shared between healthcare institutions. The data is strictly personal and the consequences of unintentional disclosure are severe. Recordkeeping systems embedded in the various healthcare systems must therefore adhere to the highest standards of auditability and privacy. Blockchains allow for immutable recordkeeping, which means that data stored on the blockchain cannot be changed or tampered with. Each block on the blockchain stores the computed hash of the contents of the previous block, which makes each new block dependent on the previous block. Nodes store their own copies of the blockchain and keep them synchronized by using mechanisms for distributed consensus. Distributed consensus mechanisms for blockchains facilitate methods to decide which block is to be added to the blockchain next and essentially decide which version of the blockchain is the correct one. This thesis presents an implementation of a blockchain framework [1] for improving auditing and privacy measures of electronic health record (EHR) systems. The framework was partly presented by Yang et. al in 2018 and submitted for publishing in 2019. The proposed framework presents a new layer that can be implemented on top of existing EHR systems. This makes the process of adopting the system much simpler and less costly. The aim of this thesis is to assess how such an implementation can be created using the Hyperledger Fabric blockchain. The implementation facilitates improved privacy and auditing through a solution of storing access control lists and logs directly on the blockchain. Each attempt to access a record is verified in the access control list and subsequently logged before access is granted to the user. This introduces a standard way of managing access control and auditing across several providers, even if the internal system architecture is different for each provider. The layer can be deployed on top of existing systems and only minor changes to the database interfaces are required for the systems to support the new layer. Although the presented implementation is intended for use in EHR systems, it should also be applicable to other types of recordkeeping systems.

Preface

This thesis is written as a conclusion of my two-year master's degree in informatics at the University of Bergen (UiB). Prior to this study I also hold a bachelor's degree in computer engineering from the Norwegian University of Science and Technology (NTNU). The knowledge acquired from the courses I attended at UiB and NTNU has formed much of the theoretical knowledge employed in this thesis. However, a lot of new knowledge and expertise have also been acquired after I started working on the thesis. The skills to develop software for the chosen blockchain framework have been acquired from extensive studies of the framework's documentation and sample applications. The process of writing this thesis started on August 13, 2018 and ended on June 3, 2019. I would like to thank my supervisor, Associate Professor Chunlei Li of UiB, for excellent assistance and support during this period. His expertise on the subject has been a critical asset to writing the thesis.

Contents

Abstract **i**

Preface **iii**

List of Figures **viii**

List of Tables **xi**

1 Introduction **1**

1.1 Background 1
1.2 Problem Definition 3
1.3 Intended Results 4
1.4 Related Work 6
1.5 Thesis Outline 6

2 Theoretical Background **9**

2.1 E-Health 9
2.1.1 Electronic Health Records 10
2.1.2 Security and Privacy Concerns 11
2.2 Blockchains 12
2.2.1 Cryptography in Blockchains 13
2.2.2 Distributed Consensus 14
2.2.3 Smart Contracts 16
2.2.4 Permissioned vs Permissionless Blockchains 17
2.3 Hyperledger 17
2.3.1 Hyperledger Fabric 19
2.3.2 Chaincode 21

2.3.3 Node Definitions and Domain 22
2.3.4 State Database 24
2.3.5 Block Generation and Consensus 25
2.3.6 Software Development Kits 26

3 Implementing the Framework **29**
3.1 Overview 29
3.1.1 The Proposed Framework 29
3.1.2 The Blockchain Implementation 31
3.2 Hyperledger Fabric 32
3.2.1 Fabric Tools 33
3.2.2 Software Containers 33
3.2.3 Ordering Service and Consensus 34
3.2.4 Certificate Authorities and Membership Services 36
3.2.5 Network Discovery 37
3.3 Java SDK for Hyperledger Fabric 38
3.3.1 Communication Clients 38
3.3.2 Query Requests and Transaction Proposals 39
3.3.3 Collecting Endorsements 39

4 System Description **41**
4.1 Overview 41
4.1.1 HLF Network Package 43
4.1.2 Java Application Package 43
4.2 HLF Network Architecture 44
4.2.1 Configuration 44
4.2.2 Chaincodes 46
4.2.3 Ordering Service 51
4.2.4 Incentive Mechanism 52
4.3 Java Application Architecture 54
4.3.1 Hyperledger Fabric Integration 56
4.3.2 Endorser Selection 57
4.3.3 Graphical User Interface 57
4.4 Deploying the HLF Network Package 58
4.4.1 Initial Configuration of the Network 59

4.4.2 Network Lifecycle Management 60
4.5 Deploying the Java Application Package 62
4.5.1 Required Resource Files 63
4.5.2 User Interface Interaction 64

5 Design and Performance **67**
5.1 Benefits over Traditional Healthcare Systems 67
5.2 Validating the System 68
5.2.1 Block Creation and Policies 69
5.2.2 Resilience to Fault and Misuse 70
5.3 Performance at Scale 70
5.3.1 Increasing Peer to Orderer Ratio 71
5.3.2 Raft vs Kafka Ordering Service 73

6 Conclusion **75**
6.1 Concluding Statement 75
6.2 Further Development 77

A Source Code and Development **85**
A.1 Development Setup 85
A.2 Source Code 86

B Setup Guides **87**
B.1 HLF Network Package 87
B.2 Java Application Package 88

List of Figures

2.1 A traditional EHR system comprising two providers 10
2.2 Three blocks creating a blockchain by storing the hash of the previous block in the header of each succeeding block. 12
2.3 A basic protocol for public-key encryption of a message sent from A to B. . 13
2.4 A typical block implementation with a data section holding transactions and a header section holding related metadata. 14
2.5 A smart contract executing a program that takes an input and produces a matching output. 16
2.6 Illustration of the Fabric transaction flow where each step is represented by activities of a different colour. 20
2.7 The roles and processes applicable to the leader-follower pattern. 24
2.8 Peer nodes are composed of a state database, a blockchain copy, chaincodes and an MSP. 25

3.1 An illustration of the communication hierarchy within a peer organization with four peer nodes, two orderer nodes and multiple clients. 35
3.2 Hyperledger Fabric transaction flow featuring a single endorser. 36

4.1 The architecture used when deploying the system, illustrated with a network of two providers. 42
4.2 HLF Network package directory structure. 45
4.3 The representation of an SC stored on the blockchain. 47
4.4 SC chaincode class diagram. 48
4.5 RRC objects stored on the blockchain, which use the RRC reference as the state database key. 49
4.6 RRC chaincode class diagram. 50

4.7 Incentive Mechanism chaincode class diagram. 53
4.8 Java Application class diagram. 54
4.9 Java Application package directory structure. 55
4.10 Methods provided by the MainController class. 58
4.11 Java Application user interface. 65
4.12 The transaction flow embedded in the Java application. 66

List of Tables

5.1 Average running times of a single iteration of the three-step transaction flow with different numbers of peers and orderers selected per organization. 72
5.2 The average required setup times for the Kafka and Raft ordering services in the development environment specified in Appendix A. 73

Chapter 1

Introduction

1.1 Background

In most countries, the healthcare industry is composed of several healthcare providers, some with private ownership and some public. To provide the best possible healthcare services for their patients, providers must exchange patient information with each other. By exchanging information with other providers, health personnel can get a complete overview of a patient's health. This can be critical in determining which type of treatment a patient requires, especially in emergency situations. For this reason, as much patient data as possible is being stored electronically, so that it can be shared quickly with providers over a network. These health records are termed electronic health records (EHRs) [2].

Most providers operate and govern their own local databases of patient data. These databases are running in each provider's own private network, which means that EHR databases are spread around in multiple physical locations. The different databases might contain EHRs formed from multiple data formats, which means that providers might have to support multiple proprietary formats to be able to interpret the EHRs from every provider's database. Several common data standards and regulations for EHR systems have been proposed to solve this issue [3]. However, interoperability between EHR systems still remains a major challenge for the industry.

As soon as data is made available over a network it is susceptible to eavesdropping and remote attacks. EHRs are regular targets of both malicious attacks and misuse. The contents of an EHR is strictly personal and have great potential in being used to blackmail individuals and institutions. Providers that are not properly managing and securing their recordkeeping systems are penalized with large fines and other repercussions issued by governments overseeing the healthcare industry [4]. Being found responsible for mismanagement of health data could mean that the entire institution must be shut down.

IT systems governing such healthcare records must therefore be able to maintain confidentiality and integrity of the data in all possible scenarios. However, IT systems in the healthcare industry are highly complex with a great amount of legacy code, which means that even a slight change of functionality might require extensive code rewrites. As a result, there is no single easy solution to make these systems secure. If an event does breach confidentially or integrity of the data it is also important that proper auditing facilities are in place, so that it is possible to estimate the extent of the breach, close any exposed vulnerabilities and avoid it from happening again.

The concept of blockchains first gained traction in the form of public cryptocurrencies like Bitcoin [5]. In the last few years, however, blockchain has also sparked interest from other domains. This includes the healthcare domain. Blockchain technology can introduce immutable recordkeeping to an institution. Once data is placed on the blockchain it cannot be changed or replaced without anyone noticing it. This means that as soon as a record is placed on the blockchain neither health personnel nor malicious users are able to tamper with a record.

However, storing complete health records on a distributed blockchain requires a significant amount of data replication and storage. This again would deteriorate the performance of the blockchain network and would require large parts of existing recordkeeping systems to be re-created to work with a blockchain as opposed to a traditional database. In a non-ideal world composed of expensive legacy systems we must consider how we can benefit from new technology without having to re-create already functional systems.

1.2 Problem Definition

The focus in this thesis will be on how blockchain technology can be implemented on top of already existing EHR systems to improve privacy and auditability of such systems. Implementing the technology on top of already existing systems will make a transition to the new technology much easier and affordable for all stakeholders, and one will avoid a situation where all data maintainers would have to start from scratch with new IT systems and databases. The main topic of this thesis is defined by the following problem definition:

> How can blockchain technology be implemented in existing EHR systems to improve privacy and auditability of EHRs?

The thesis will produce a description related to development of a blockchain implementation using the Hyperledger Fabric blockchain [6]. The description and development process build on the work conducted in developing the framework presented in [1]. The article presents a blockchain framework for preventing misuse of EHRs through improving database auditing processes by storing database events and related metadata on the blockchain. The proposed framework in [1] will be referred to as the EHR framework for the remainder of this thesis.

The developed system will implement the smart contracts proposed in the EHR framework and make the necessary modifications and improvements for them to work with the Hyperledger Fabric blockchain. The thesis will also investigate how the proposed incentive mechanism of the EHR framework can be implemented in the system. The proposed measures for log encryption and key distribution will not be assessed in this thesis, but potential future implementation of the measures should be taken into account when designing the system.

Hyperledger Fabric's notion of a smart contract is termed a chaincode and differs in several areas from the smart contract implementations provided by other blockchain projects, e.g. the Ethereum project [7]. For the remainder of this thesis, the term chaincode will be used when referring to actual Fabric smart contract implementations and the term smart contract will be used when discussing high-level framework design.

The following objectives are to be solved in the work on this thesis:

- Create chaincode implementations for the proposed smart contracts
- Create an implementation of the proposed incentive mechanism
- Implement the business logic and processes required for the blockchain network to remain operative
- Create a self-contained application embedding one of the software development kits (SDKs) for Fabric [8] to communicate and interact with the network

The resulting blockchain network will be a self-governing network relying only on a specific number of functional nodes in the network to remain operative. This means that no single authority should be in charge of the network. Any decisions in the network must be made based on majority votes and network-wide policies.

1.3 Intended Results

The EHR framework presented in [1] proposes two smart contracts for representing patient-provider and record-event relationships respectively, as well as a new incentive mechanism. The incentive mechanism uses a value of significance associated with each provider to decide which provider is responsible for generating the next block for the blockchain. As a reward for generating the block, the provider's significance is increased.

The work on this thesis will result in a description of how such a system can be implemented with use of the Hyperledger Fabric blockchain. Together with Ethereum, the Hyperledger projects are some of best known and recognized open source blockchain projects in development today. Hyperledger Fabric is favoured over Ethereum in this thesis because of its design as a permissioned private network, as opposed to Ethereum's public blockchain [9], and because of its unique three-step transaction flow and smart contract implementation [6]. Most components making up a Fabric blockchain are also modular and replaceable, which is an important feature of a system that is intended to be further extended at a later time.

Along with a description of the Fabric implementation of the EHR framework, an actual prototype of the system will be developed. The prototype is intended to showcase how the system will behave in real production systems and allow for testing and measurements to be performed on the system. The prototype will comprise two software packages:

- HLF Network package (Hyperledger Fabric blockchain network)
- Java Application package (Java SE front-end and business logic)

The HLF Network package will comprise the configuration files and source code required to initialize and deploy the Hyperledger Fabric network. This will be a fully scalable and functional network. One must, however, expect some components to be replaced with components that are compatible with existing protocols and components in specific healthcare network implementations. Such components would typically be e.g. network certification authorities for issuing digital identities.

The Java Application package will contain business logic for interacting with the Fabric blockchain. The application will utilize the Fabric software development kit (SDK) for Java [10]. The Fabric SDKs help applications manage chaincodes and events on the blockchain, as well as making the application able to act on behalf of a specific user context. A graphical user interface (GUI) for invoking the application methods will also be created, which makes for easier demonstration of the system.

The business logic part of the application can also be invoked by clients, such as existing applications in the healthcare network, without using the GUI. This offers a way to communicate with the network without extensive knowledge of an SDK. However, in a production environment, existing software should be embedded with a Fabric SDK directly for better performance and less resource engagement. The methods in the Java application can be re-used in other applications for such purposes.

1.4 Related Work

The work conducted in this thesis relates directly to the framework presented in [1], which presents measures to strengthen the auditing and privacy aspects of storing patient health records. The proposed EHR framework will be discussed in more detail in subsequent chapters. Some other related efforts in the field of blockchain for the healthcare industry include a decentralized records management system named MedRec [11], a data management gateway for access requests proposed named Healthcare Data Gateway (HDG) [12] and the scalable ledger OmniLedger [13].

In MedRec, first presented in 2016, the authors use smart contracts to provide patients and providers with addresses linking to existing health records, essentially providing patients with logs and easy access to their health records across providers [11]. Providers are incentivized to participate in the network by receiving aggregated and anonymized data as rewards for validating blocks in the network.

The authors of HDG introduce a data management layer in which the patients are in complete control of approving individual access requests from healthcare providers [12]. The layer works as a gateway, where every entity that requests access to a record must be approved by the patient owning that record. The patient can approve the request through a mobile application.

In the paper on OmniLedger, the authors present a scaleable ledger intended for solving scaling issues without compromising security and decentralization. The intention is that by using a technique known as sharding, performance should be increasing instead of decreasing as the number of users in the network grows [13].

1.5 Thesis Outline

The chapters of this thesis, in chronological order, are:

Abstract - Summary of the most important aspects discussed and introduced in the thesis

Preface - Introduction to the author's background and acknowledgements

1 Introduction - Information about the research background and intended outcome

2 Theoretical Background - An introduction to the theory that the thesis is based on

3 Implementing the Framework - A presentation of the methods used to design and implement the EHR framework

4 System Description - A presentation of the complete system and how it can be deployed

5 Design and Performance - Discussion about the research outcome and further improvements on the system

6 Conclusion - A conclusion derived from the obtained results

Chapter 2 introduces the theory applicable to blockchain and e-health. An introduction of the Hyperledger Fabric blockchain, explaining its implementation of smart contracts and definition of participating nodes, is included here. Chapter 3 provides an explanation of the methods used to obtain the results presented in the thesis. This includes a description of the components that make up a Hyperledger Fabric network and an application made to interact with the network.

In Chapter 4, we present the obtained results along with a description of how the network is to be configured and deployed correctly. We then follow with a discussion of the results in Chapter 5. The discussion further explains design decisions made during the development process and provides an assessment the components that constitute the finished implementation.

Finally, we conclude the thesis with Chapter 6, presenting our conclusion based on the obtained results along with a suggestion for possible future improvements. The appendices provide additional information on how to replicate the development environment and how to use the software that was developed for the thesis.

Chapter 2

Theoretical Background

2.1 E-Health

The term e-health is a broad definition used for denoting the digital processing of health information e.g. digital prescriptions, appointment scheduling and patient data records [14]. E-health is a vital resource to any healthcare system. Being able to exchange health information digitally is key to effective medical help and is especially important in emergency situations. As with all digital processing systems, the challenges in e-health are generally related to maintaining availability, confidentiality and integrity of the data.

The demand for confidentiality and integrity of data within the field of e-health are among the highest in any industry. Much of the health information transmitted is strictly personal and governed by national privacy laws. In Norway, the national policy and standards regarding e-health are administered by the Norwegian Directorate of eHealth (NDE) [15] which is a subordinate to the Norwegian Ministry of Health and Care Services. NDE list their two principal responsibilities as follows:

- National steering and coordination of eHealth through close cooperation with regional health authorities, local authorities, technical organisations, and other interested parties

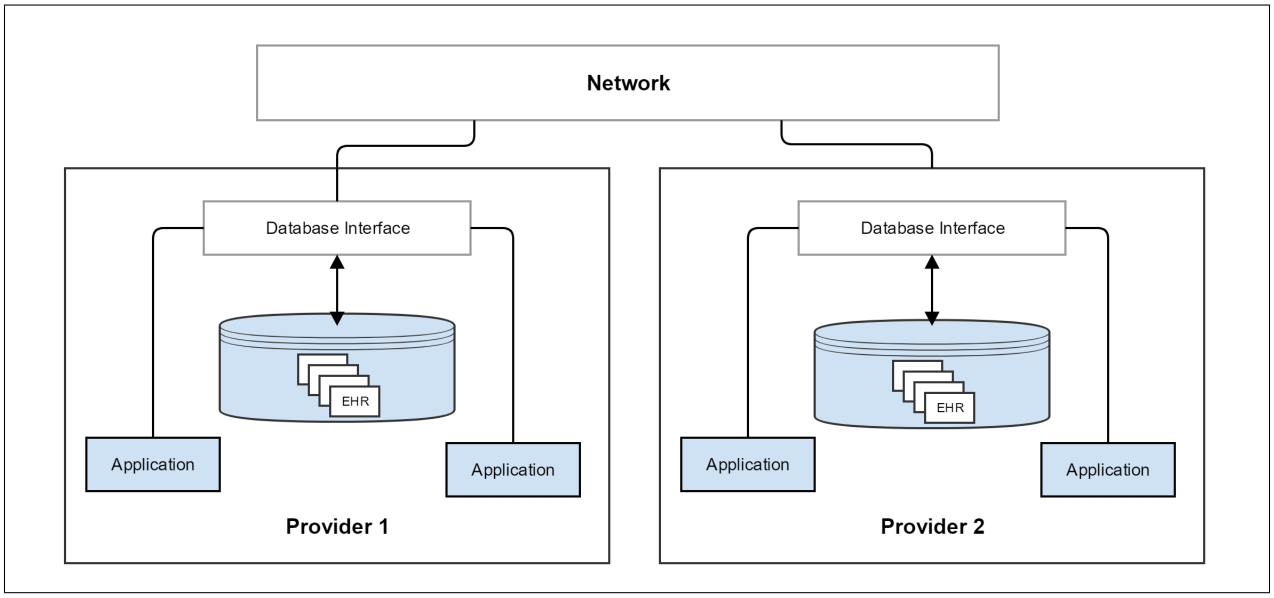


Figure 2.1: A traditional EHR system comprising two providers

- Develop and administrate digital solutions that will improve and simplify our health and care sector

Regulatory institutions ensure that healthcare data is not neglected and that it is stored according to the requirements specified by the law.

2.1.1 Electronic Health Records

An electronic health record (EHR) is a collection of a patient's electronically stored health data [16], e.g. test results and medications. The benefit of an EHR over a traditional physical record is that an EHR can be shared electronically and thereby be available to health personnel at other locations much quicker, as illustrated in Figure 2.1. Sharing EHRs between different health institutions over a network means that a patient's health information is available for use immediately when it is needed, no matter which institution maintains the original record.

Several EHR specifications, standards and regulations exist [3]. An EHR specification comprises both data models and communication standards. Communication standards are intended to support interoperability between different systems and to maintain confidentiality of the data. Institutions sharing EHRs must either agree on a common EHR standard or implement measures to interpret data in multiple formats.

2.1.2 Security and Privacy Concerns

The challenges in respect to sharing EHRs are many. The great benefits of electronic access to records come with an increased security risk. Essentially, EHRs facilitate for rapid sharing of records to possibly a large number of people. As a result of potential system errors or malicious entities present in the network, privacy breaches and unauthorized access to EHRs are known to occur and are hard to completely safeguard against.

It is important to realize that privacy can be violated without involvement of any malicious non-authorized entities. Even health personnel that are authorized to access EHRs, are not supposed to access an EHR of a patient if it is not a strict requirement for them to perform their job. In [3], Fernández-Alemán et al. concludes that a harmonisation of security and privacy standards found in EHR systems are required, and that auditing is particularly useful to identify suspicious access and common access practice.

Typically, patients are not intended to directly access EHRs, as EHRs are merely used by health personnel to decide how to treat a patient. In some systems, however, EHRs can be partly presented to patients [17] through various web-based services. Depending on the country where the system is deployed, patients might have the legal right to access their complete health information. However, to get hold of a complete EHR one must make an official request.

The benefits and concerns in allowing patients to access their EHRs, e.g. immediate access to test results without waiting for a practitioner to assess the results, are discussed by Beard et al. in [17]. One of the concerns raised with respect to this is that patients are not qualified to assess test results and might as a result of this misinterpret critical test results, which is why the approach of only partially presenting EHRs to patients is used in most of such systems.

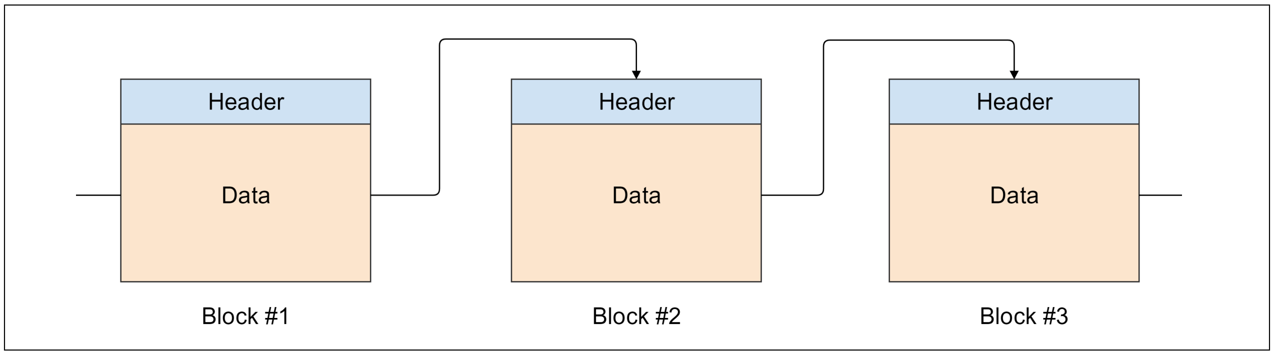


Figure 2.2: Three blocks creating a blockchain by storing the hash of the previous block in the header of each succeeding block.

2.2 Blockchains

Blockchain is the term used for a distributed ledger constructed as a chain of blocks, as illustrated in Figure 2.2. The concept was first introduced by Satoshi Nakamoto in 2008 and is best known for its implementation in the Bitcoin cryptocurrency network [5]. The blockchain technology has received large interest in the recent years and many of the world's leading IT companies, such as IBM and Oracle, have devoted substantial amounts of resources to work on the technology.

A distributed ledger is characterized by the fact that information is not stored in a single location governed by a single entity, but instead replicated by every node in the network. Each node holds its own copy of the ledger, meaning that malicious changes made to the ledger on one of the nodes will not be replicated to other nodes, as long as the malicious nodes do not outnumber the functional nodes [5]. The number of malicious or faulty nodes required to break the network depends on the specific method for consensus used in the network.

Information stored on the ledger is placed in blocks. A block consists of a header section and a data section. The data section will typically be composed of several data entries, also known as transactions. A batch of multiple transactions can be placed in a block, instead of creating new blocks for each transaction.

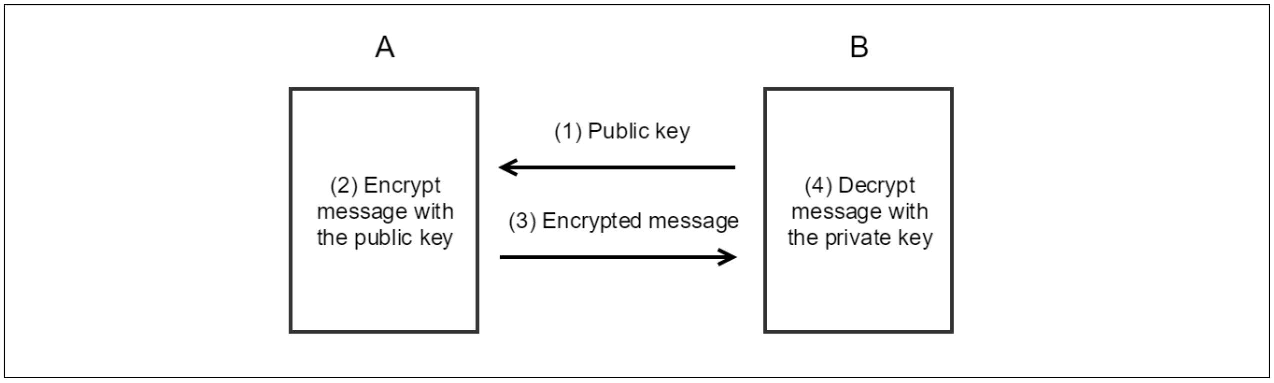

A
B
(1) Public key
(2) Encrypt message with the public key
(3) Encrypted message
(4) Decrypt message with the private key

Figure 2.3: A basic protocol for public-key encryption of a message sent from A to B.

2.2.1 Cryptography in Blockchains

Cryptographic methods are used for a wide range of operations in blockchain networks, e.g. securing blocks and signing transactions. These operations are supported by the use of hash functions and public-key cryptography [18]. A hash function can be thought of as a one-way mapping from a specific input to a specific output. The chances of collisions are extremely small and computationally infeasible to find. We can therefore consider each input to produce a unique output.

Transactions are signed using public-key cryptography, also known as asymmetric cryptography. A basic protocol for public-key encryption is illustrated in Figure 2.3. Authorized entities in the network are supplied with private keys, while the public key is publicly available. An entity would sign its proposed transaction using its private key. Other entities in the network can then decrypt the transaction by using the public key and verify that the transaction was indeed created by an entity with access to a valid private key [19].

The exact information held by a block in the blockchain varies from implementation to implementation. In addition to a section holding the actual data of the transactions, a block would typically include a header holding these fields of information [19], also illustrated in Figure 2.4:

- The block number
- The hash value of the previous block's header

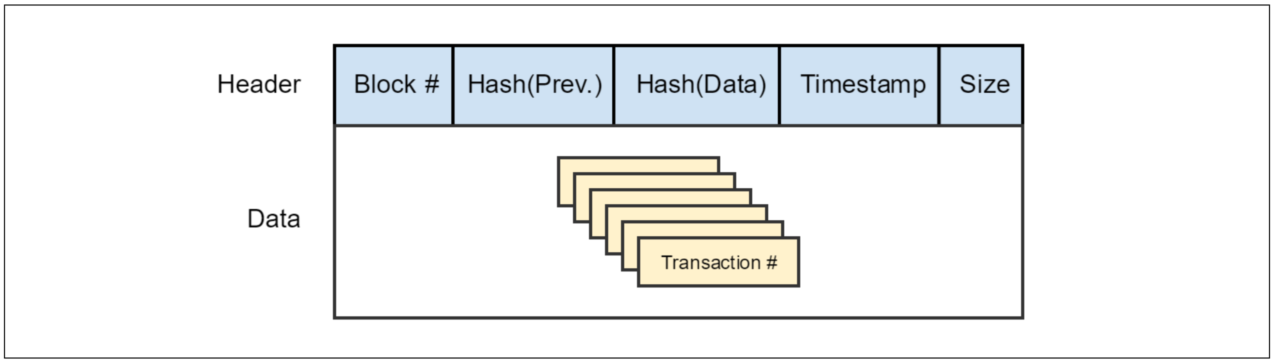


Figure 2.4: A typical block implementation with a data section holding transactions and a header section holding related metadata.

- The hash representation of the data in the block
- A timestamp
- The block size

The hash computed for a block is dependent on the data stored in the block, which means that a unique hash is computed for each block. This means that if you modify a block that is already on the blockchain, the hash for this block will change. All subsequent blocks would therefore have to regenerate their previous hash field. Regenerating all the subsequent blocks is a computationally expensive operation and requires a substantial amount of time and resources to succeed.

More importantly, due to the security characteristics of the hash function, it is computationally infeasible to generate blocks that match hashes in the existing blocks. In other words, a malicious node will not be able to recompute all succeeding blocks and it will not be able to convince every other node in the network that its version of the blockchain is the correct version, without having acquired some necessary majority share in the network.

2.2.2 Distributed Consensus

Consensus mechanisms are essential for reliable distributed computing systems and are important components in blockchain networks. Consensus algorithms are responsible for reaching and maintaining consensus in a distributed network, such as a blockchain network. Consensus in a blockchain network is concerned with making sure that the next block that is added to the blockchain is a valid block and that all attempts from malicious or malfunctioning nodes to spoof participants with false blocks are disregarded [19]. When the majority of nodes in a network agree on a version of the blockchain, consensus is reached.

There are several situations in which a distributed network might not be able to reach consensus. A consensus algorithm might be vulnerable to some of these situations and tolerant to other. When choosing a consensus algorithm, it is therefore important to be fully aware of which situations might occur in a specific network. One of these situations is known as Byzantine faults. Byzantine faults are conditions where it cannot be determined if a component has failed or not [20]. A Byzantine fault tolerant system is generally able to operate as long as the number of faulty nodes do not exceed one third of the total number of nodes in the network. A typical starting point for an implementation of a BFT algorithm is the Practical BFT (PBFT) algorithm [20].

There are several different consensus algorithms deployed in various blockchain networks. The best known algorithm is perhaps the Proof of Work algorithm [5] used in Bitcoin and several other cryptocurrencies. Proof of Work implements the task of block generation through a process in which nodes are required to solve complex cryptographic tasks, e.g. finding a specific value of which the hash output begins with a specified number of zero bits before the block can be successfully added to the blockchain [5]. When a node finds the correct value, other nodes in the network verify that the value is correct before adding the block to their version of the blockchain. As an incentive to complete the task, nodes typically get a certain amount of cryptocurrency as a reward for successfully completing it.

Although Proof of Work is still a popular consensus algorithm, it is criticised for its huge computational resource requirements which result in a huge waste of energy [21]. Another popular algorithm which requires considerably less computational resources is the Proof of Stake algorithm. In a Proof of Stake algorithm, validators are selected based on their economic stake in the network, e.g. the amount of cryptocurrencies they are in possession of and how long the currency has been in their possession [21], and not on their ability to complete cryptographic tasks.

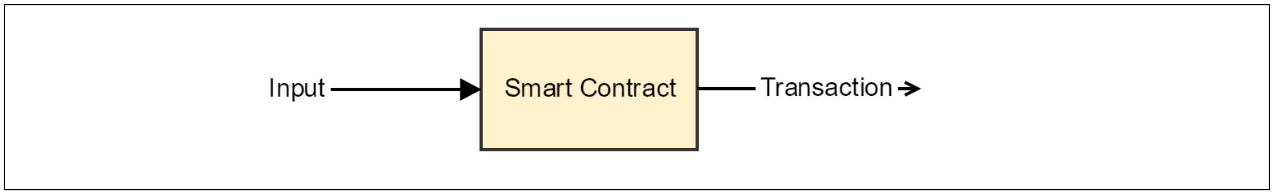


Figure 2.5: A smart contract executing a program that takes an input and produces a matching output.

2.2.3 Smart Contracts

Smart contracts are small programs installed on the blockchain, typically executing some sort of business logic in an automatic response to a change in the blockchain or the network topology [19]. Most blockchain platforms offer some implementation of smart contracts. The original purpose of a smart contract was to represent traditional written contracts in a way that removes the need for trusted third parties, such as a lawyer, to make certain that the criteria of a contract is fulfilled [19]. In practice, however, a smart contract can be any kind of program executing business logic that makes sense to install on a blockchain.

Most commonly, a smart contract specifies a set of constraints that must be fulfilled in order for the program to execute. Figure 2.5 shows a smart contract taking a set of inputs and producing an output in the form of a transaction. When installing the program on the immutable blockchain, we ensure that these constraints cannot be tampered with [19]. This erases the need for a trusted third-party to validate that the requirements of a contract have been successfully fulfilled. However, the fact that the smart contract is installed on a blockchain also means that it can be difficult to correct bugs in the program, especially in public blockchains where it can be difficult to get all involved parties to agree on a new version of the program.

Ethereum [9] is an open source public blockchain platform that became popular mainly due to its implementation of smart contracts. Ethereum smart contracts are written in a programming language called Solidity and allow users to add their own functionality to the Ethereum blockchain [9]. Although Solidity is developed by a team of Ethereum project developers, it is also used as the programming language for smart contracts in several other blockchain platforms. Several general-purpose programming languages, such as Java and Python, can also be used for writing smart contracts in some blockchain platforms [22].

2.2.4 Permissioned vs Permissionless Blockchains

Permissioned blockchains do not allow for public unauthorized access to the blockchain, which means that every node in the network must be authorized before they can access it, as opposed to in a permissionless blockchain network where everyone is free to participate [19]. In all blockchain networks, data stored in a block is visible to every node that is part of the network. This stems from the fact that to verify a block, nodes must be able to view the data within the block. This means that all non-encrypted data placed on the blockchain can be viewed by every node in the network. For permissioned blockchains, the nodes seeing the data will of course be limited to authorized nodes.

Permissioned blockchains are usually domain specific and aimed towards a smaller group of participants, which also means that the length of the blockchains are typically much shorter than for public blockchains. This allows enterprises to store larger amounts of data in each block without harming network performance. On a public blockchain, the amount of data must be limited to avoid storage and processing issues as the blockchain grows exceptionally large.

The integrity of both permissioned and permissionless blockchains are maintained by consensus algorithms, which provide a measure for deciding which is the correct version of the blockchain and prevent any attempt from malicious or malfunctioning nodes to corrupt the network [19].

2.3 Hyperledger

Hyperledger is an open source blockchain project consisting of several blockchain related frameworks and tools managed by the Linux Foundation. By the start of 2019, the project comprises six different frameworks for deploying blockchains, along with seven tools for benchmarking, deployment, modelling, analysing, ledger interoperability and cryptography [23].

The frameworks are described by Hyperledger as follows:

- Hyperledger Burrow - Permissionable smart contract machine (EVM)
- Hyperledger Fabric - Permissioned with channel support
- Hyperledger Grid - WebAssembly-based project for building supply chain solutions
- Hyperledger Indy - Decentralized identity
- Hyperledger Iroha - Mobile application focus
- Hyperledger Sawtooth - Permissioned and permissionless support, EVM transaction family

Each Hyperledger framework is targeted for different use cases and user groups. Common for all the projects is that they bring something unique to the group and that they are applicable to companies operating in vastly different business sectors. The tools provide additional functionality to the frameworks and are described as follows [23]:

- Hyperledger Aries - Infrastructure for peer-to-peer interactions
- Hyperledger Caliper - Blockchain framework benchmark platform
- Hyperledger Cello - As-a-service deployment
- Hyperledger Composer - Model and build blockchain networks
- Hyperledger Explorer - View and explore data on the blockchain
- Hyperledger Quilt - Ledger interoperability
- Hyperledger Ursa - Shared cryptographic library

2.3.1 Hyperledger Fabric

Hyperledger Fabric was originally initiated by IBM and is currently one of the frameworks under the Hyperledger umbrella. The framework specifies a permissioned blockchain. The main features of the Fabric framework, that for the most parts are not found in other frameworks, are as follows [6]:

- Modular support for consensus protocols
- A three-step transaction flow where each step can be run on a different entity
- Support for smart contracts written in standard general-purpose programming languages
- Support for private sub-ledgers, known as channels
- Configurable and modular membership services

Hyperledger Fabric incorporates a unique three-step transaction flow that is not found in other blockchains. The transaction flow is made up of an execute-order-validate architecture, comprising an endorsement step, a block creation step and a validation step [6]. The tree steps are illustrated in Figure 2.6, where each step is indicated by a different colour. In the endorsement step, nodes are required to replicate the transaction in their version of the blockchain, to see if the same output is produced. If the selected nodes return the same result, the transaction is endorsed. The next step is to place the transaction within a block. Nodes in the network then validate the block before adding it to their blockchain [6].

Another unique concept in Fabric is the channel. A channel is a private ledger which provides data isolation and confidentiality [6]. Only authorized nodes can interact with a specific channel. There can be several channels in a single Fabric blockchain network, meaning that nodes that require private information to be exchanged between themselves can create their own separate private channel in addition to being part of the main channel.

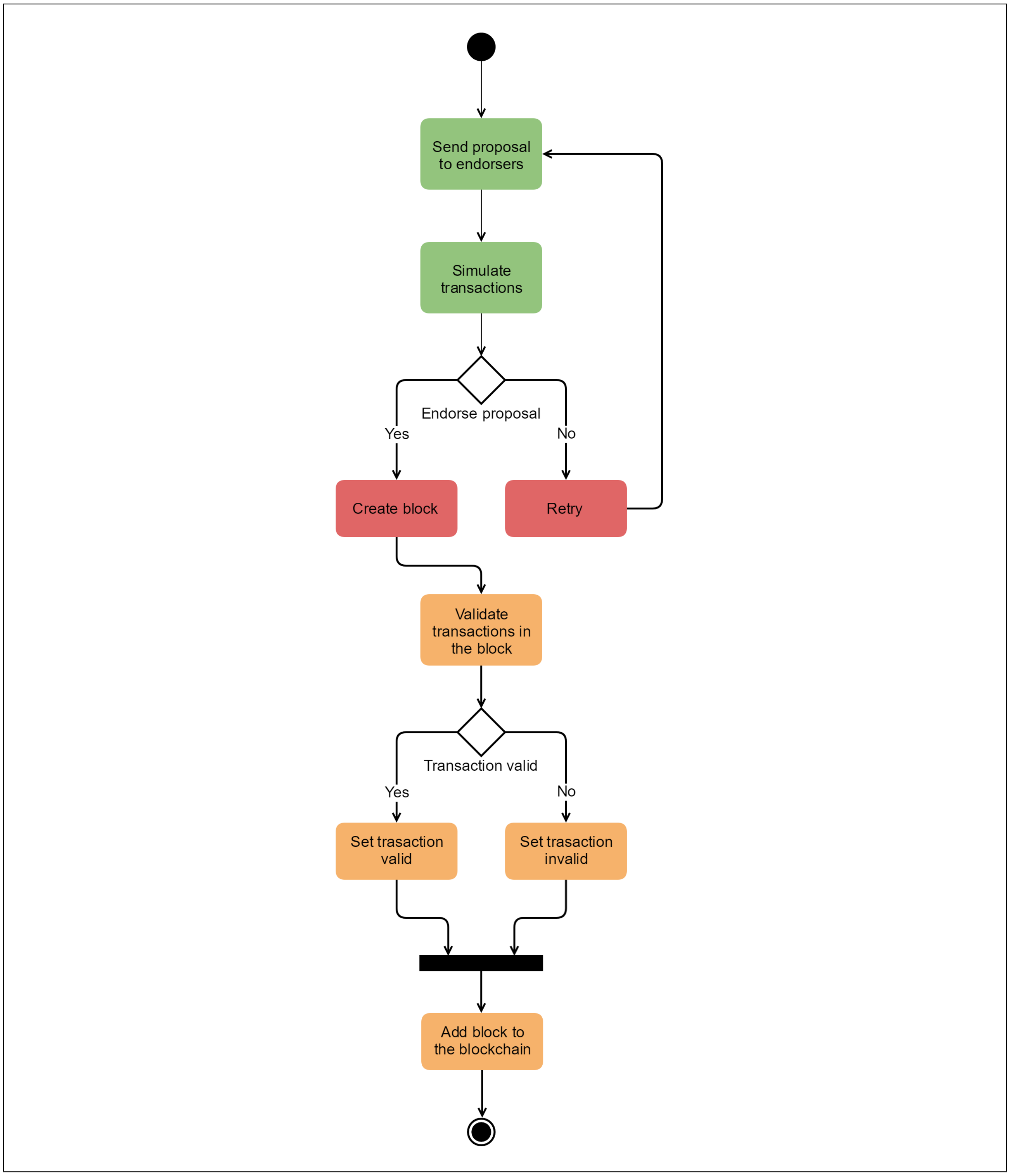


Figure 2.6: Illustration of the Fabric transaction flow where each step is represented by activities of a different colour.

Fabric's modular architecture causes many of its components, such as the mechanism of consensus and membership services, to be pluggable and configurable [6]. All main components of a Fabric network operate in their own separate environment, such as a Docker container [24]. A malfunctioning component can be replaced simply by stopping and tearing down its container, and thereafter bring up a new container to replace it.

Another Hyperledger framework sharing parts of the same design philosophy as Fabric is the Hyperledger Sawtooth framework [25]. Hyperledger Sawtooth was initially under Intel development but was incubated in the Hyperledger project in 2016 [26]. Both Sawtooth and Fabric appeal to multiple use cases because of their modular architecture, meaning that they can be used in networks developed for a wide range of different business cases.

2.3.2 Chaincode

Smart contracts in Hyperledger Fabric are known as chaincodes. Currently supported programming languages for chaincode are Go, Node.js and Java. However, chaincodes running on the same channel must all be written in the same language [6]. Nodes with chaincode installed on them are only aware of the name and version number of the chaincode and not which programming language it is written in.

A chaincode executes in its own container, separate from the node where the chaincode is installed [6]. However, the chaincode container is not created until the node receives its first chaincode request. This generally results in a significant delay for the first chaincode call, but reduces the computational resources required occupied by the network. Before we instantiate or upgrade chaincode on a channel, we must make sure that the chaincode is installed on the required number of nodes [27]. Multiple versions of a chaincode might be installed on a node at the same time.

The Fabric chaincode libraries for Go, Node.js and Java provide methods managing transactions proposed by applications [27]. Methods to invoke chaincode functions from within another chaincode are also provided.

There are three chaincodes, called system chaincodes, used in Fabric by default [27]:

- Lifecycle Chaincode (LSCC)
- Configuration Chaincode (CSCC)
- Query Chaincode (QCSS)

These chaincodes control various system functionality, e.g. controlling the process of installing user-created chaincode.

2.3.3 Node Definitions and Domain

There are three types of nodes in a Hyperledger Fabric network [6]:

- Client
- Peer
- Orderer

Nodes are defined based on the different roles they play in the network. Client nodes invoke blockchain events and transactions through peer nodes on behalf of the applications they represent [6]. Peer nodes hold the chaincodes instantiated on the channel and execute chaincodes involved in a transaction to validate that the proposed transaction produces the same chaincode output on each peer [6].

Each peer in the network is provided with a membership service provider (MSP). MSPs are used for managing identities for the nodes in the network. The peer uses the MSP to sign and validate endorsements when issuing a transaction or when verifying transaction proposals coming from other peers [6]. After enough peer nodes have signed off on the transaction, the client that proposed the transaction sends it to the orderer nodes for block creation. A block can contain a single transaction or a batch of multiple transactions. Peer nodes also validate the transactions constituting a block after the block has been created [6].

Nodes are operating under different organizations. An organization owns and operates a set of nodes in the network. For each organization there is at least one node operating as an anchor peer [28]. The anchor peer is visible for all organizations on the channel, allowing nodes from other organizations to discover it and communicate with other nodes in the organization as well [8]. To avoid single point of failure it is advised to have several redundant nodes of each type.

The orderer nodes are known collectively as the ordering service [6]. Orderer nodes are responsible for creating blocks. After a block has been created, the elected leading peer of each organization pulls the block from the ordering service and distributes it to each peer in its organization [6]. The leading peer of an organization can be set manually or dynamically. Dynamic leader election initially elects one peer for each organization as the leading peer. The leading peer sends updates to the rest of the peers in its organization regularly to show that it is still alive [28]. If peers stop receiving updates from the leading peer, they will elect a new leading peer.

The ordering service component is designed so that it is pluggable, meaning it can be changed based on the needs of the specific Fabric implementation. Currently, there are three types of ordering services officially implemented in Hyperledger Fabric: Solo, Kafka and Raft [29]. Several unofficial ordering service implementations also exist. The Solo ordering service rely on a single orderer node to create blocks and is not intended to be used in production environments [29].

The Kafka ordering service relies on an Apache Kafka cluster [30] to preserve data while the orderer nodes work on creating new blocks. The orderer nodes pull data from the Kafka nodes when they are ready to receive new data. The Kafka cluster relies on an ensemble of Apache ZooKeeper data nodes [31] to track the status of each node in the cluster. Data is replicated to all nodes in the cluster from a node selected as the cluster leader. If the cluster leader goes offline, the ZooKeeper ensamble is used to elect a new leader. A network using the Kafka ordering service would typically organize all orderer nodes within a single orderer organization, as the decentralized benefit of spreading the nodes in multiple organizations will be violated by communication with the Kafka cluster [29].

Raft offers the same crash-fault tolerant leader-follower pattern [32] as Kafka but gives

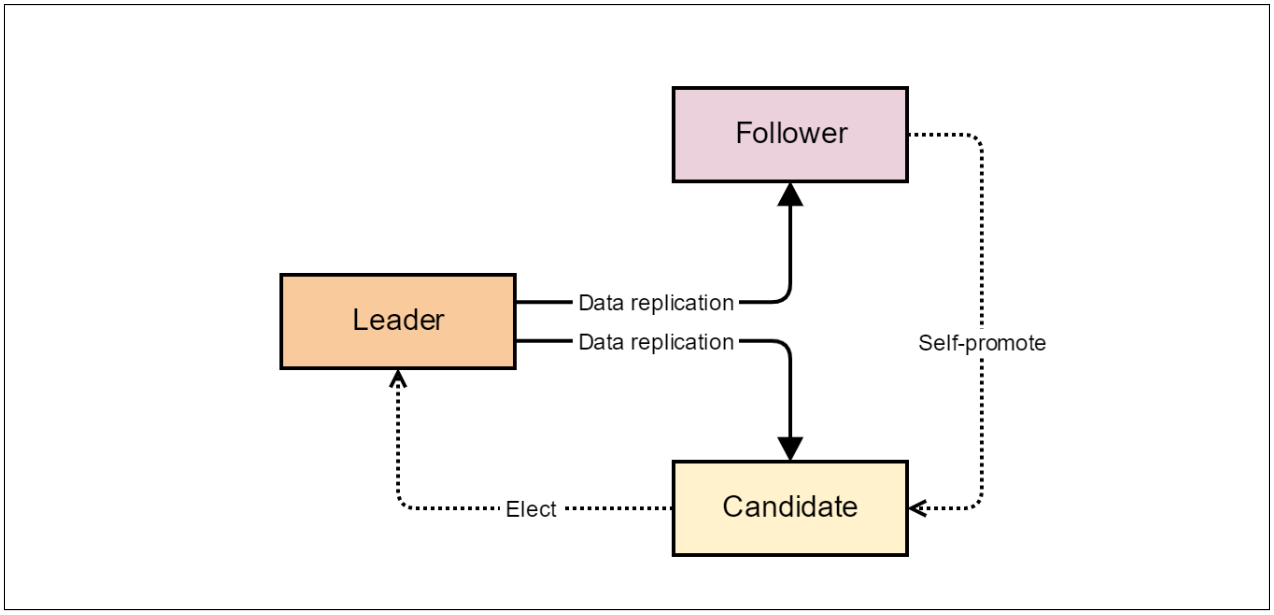


Figure 2.7: The roles and processes applicable to the leader-follower pattern.

more in terms of decentralization and less administrative overhead. With Raft, orderer nodes are typically placed within each peer organization and dynamically assigned as a leader, follower or candidate [29]. Nodes initially start out as followers and self-promote to candidate if the leader is no longer communicating. Nodes then vote for one of the candidates to be the new leader. The process is illustrated in Figure 2.7. A new Byzantine fault tolerant (BFT) ordering service based on the current Raft implementation is also in development [29].

2.3.4 State Database

A Fabric network stores data in key-value pairs. Peer nodes store this data in two places: on the blockchain and in the state database. The state database holds the latest value associated with every key, while the blockchain, which is the original immutable source of data, maintains the complete story of the key-value pair [33]. Each new transaction represents an update to a value in the state database. To find the original value of a key, we would search for the block containing the first transaction related to that key.

When a peer has successfully verified a new transaction, it updates the value of the key in the database. It is the value in the state database that is returned when a client queries the

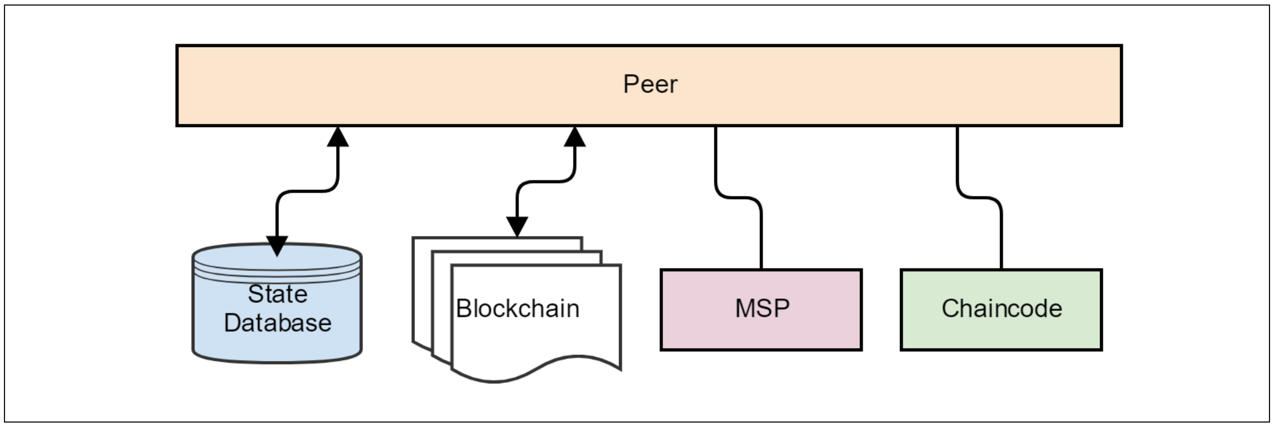


Figure 2.8: Peer nodes are composed of a state database, a blockchain copy, chaincodes and an MSP.

Fabric network [33]. A query on the blockchain is only executed when the state database must be re-created or if a historic value for the key is required. The current version of Hyperledger Fabric supports two options for state database: LevelDB and CouchDB [33].

LevelDB is the default state database implementation in Fabric. It is a simple database capable of storing simple key-value pairs. A query to the LevelDB is a traditional query on the key. CouchDB, on the other hand, is an open source noSQL document database [34] built to handle large amounts of data. Data in CouchDB is stored in JSON [35] format and rich queries using the CouchDB JSON query language are supported [33]. However, CouchDB instances run in separate containers and thereby impose more overhead on the system than the embedded LevelDB implementation.

2.3.5 Block Generation and Consensus

Blocks in a Fabric blockchain network are created by orderer nodes. However, these n-odes simply create and distribute the blocks, and are not involved in validating them [6]. The task of validating transactions of a block is placed with the peer nodes. Consensus in a Fabric network is achieved in a process of validating a set of transactions to an en-dorsement policy [6]. As opposed to other blockchain implementations, consensus is not governed by a single algorithm but by the complete process of proposing a transaction and validating the created block.

Peers initially validate that a transaction has not already been submitted, that the signature is valid and that the client proposing the transaction is authorized to perform the transaction. This process of transaction validation is known as endorsing a transaction [6] and is performed by selected peer nodes. These peers execute the same chaincodes used to generate the transaction, to check that the same output is produced. Which nodes are required to endorse a transaction is governed by the endorsement policy associated with the chaincode [6]. The endorsing peers execute the chaincode and send the results back to the client. The client then verifies the peer signatures and checks if all peers return the same result.

If the endorsement is successful, the client sends the transaction to the ordering service. No validation of the transaction is performed by the ordering service. The ordering service simply receives transactions and orders them in a block [6]. The block is then distributed via each organization's leading peer to all peers on the channel. To make sure that no changes have been made to the blockchain since the transactions were proposed and that the endorsement policy is fulfilled, transactions are validated by each peer when the block is distributed, and the individual transactions are tagged as valid or invalid. Peers then append the block to their blockchain, update their state database and alert the client if a transaction is deemed invalid [33].

2.3.6 Software Development Kits

Hyperledger Fabric currently offers SDKs for applications developed in Java and Node.js, while SDKs for Python, REST and Go are also in development [8]. The names given to the SDKs in Java and Node.js are as follows:

- Java SDK for Hyperledger Fabric
- Hyperledger Fabric Client SDK for Node.js

The SDKs provide methods for applications to manage Hyperledger Fabric channels and chaincode, e.g. ordering transactions, querying blocks, listening for events and discovering other nodes in the network. Without the use of an SDK embedded application, these features must be invoked by accessing the application programming interface (API) of the Fabric components directly from the command line [27].

The SDKs do not provide features for persistence and application developers must therefore implement such features themselves, e.g. the embedded application must implement its own method to listen for endorsing peers before ordering a transaction and for peers to validate transactions in a block. If an application sends a transaction request that is not correctly endorsed to the orderer service, the transaction will be deemed invalid in the validation phase after the block has been created and distributed to the peers [6].

Chapter 3

Implementing the Framework

3.1 Overview

The results presented in this thesis are produced by combining a theoretical blockchain framework [1] with an open source blockchain implementation. The chosen blockchain implementation is Hyperledger Fabric v1.4.1 [36]. The design of Hyperledger Fabric induces several design alterations and adjustments to the proposed EHR framework. Any adjustments made to the framework will be contemplated and discussed in the thesis.

In this section we present the EHR framework and the options of implementation that are available. Essentially, we present the methods required to produce the results presented in Chapter 4. Further discussion and justification on the chosen implementation design and framework alterations will be presented in Chapter 5.

3.1.1 The Proposed Framework

The EHR framework proposed in [1] provides an interesting starting point for implementing a blockchain network to provide improved auditing and privacy of already deployed EHR systems. The thesis will focus on the first draft of the EHR framework, which was presented in 2018. The main features to look at in this first draft are the smart contracts and incentive mechanism. The proposed procedures for encryption and key distribution through a collective authority were added to the extended version of the article, which was submitted for review with a journal in March 2019. These additional features will not be assessed in this thesis.

There are two smart contracts proposed in the article: the Record Relationship Contract and the Summary Contract. Both these contracts must be implemented as Hyperledger Fabric chaincodes. The Record Relationship Contract holds metadata for each record in a database, e.g. information about the owner and maintainer of the record, an access control list of who is authorized to access the record and a log of events that has happened to the record. Meanwhile, the Summary Contract holds a list of user-provider relationships and references to the corresponding metadata in the Record Relationship Contract. A user-provider relationship exists if a user has a record stored with the provider.

Since the EHR framework is defined with the traditional order-execute transaction flow and regular consensus algorithms in mind [19], the proposed incentive mechanism must be altered to work with the unique three-step transaction flow and consensus mechanism effectuated in Hyperledger Fabric [6].

The amount of significance associated with each provider is inherent to the proposed incentive mechanism. This value must be placed on the blockchain, which means we must develop a chaincode to do so. The three Java chaincodes to be implemented in Hyperledger Fabric are then as follows:

- Record Relationship Contract
- Summary Contract
- Incentive Mechanism

The names listed above will be used consistently for any components in the source code that relates to these specific chaincodes.

3.1.2 The Blockchain Implementation

The relevant open source blockchain projects currently available are the Hyperledger projects [23] and Ethereum [9]. The EHR framework on which this thesis is based on calls for a domain-specific permissioned blockchain with an incentive mechanism that is not driven by rewards in form of cryptocurrency or other economic stakes. This means that the framework is best suited to be implemented in one of the Hyperledger projects, as opposed to on the public and economically incentivized blockchain provided by Ethereum.

All Hyperledger open source projects are under continuous development, which makes much of the projects' documentation rapidly outdated. Documentation for all Hyperledger projects and their different versions can be found online [37]. This thesis makes use of Hyperledger Fabric v1.4.1, released April 11th, 2019 [36]. The most interesting feature introduced in v1.4.1 is the new Raft ordering service.

Information on open development issues and potential vulnerabilities of the various Fabric versions are found on the Hyperledger Fabric issue tracking website [38]. The result-s presented in this thesis are produced using some features that were introduced with v1.4.1 and the results can therefore not be reproduced in earlier versions.

The Fabric and Sawtooth projects were the first two codebases selected for incubation in Hyperledger [26]. Both projects provide implementations that are mature and production ready. Some of the most prominent differences between the current versions of the two projects are [6], [25]:

- Fabric supports strictly permissioned blockchains, whereas Sawtooth supports both permissioned and permissionless blockchains
- Fabric implements a unique transaction flow for achieving consensus in the network, whereas Sawtooth implements a traditional transaction flow and consensus algorithm
- Fabric supports channels for private transaction data between subgroup of nodes, whereas in Sawtooth data from every transaction is visible to all nodes

From studying the description of both blockchain projects, it is reasonable to suggest that the proposed EHR framework can be implemented effectively using any of the two blockchain implementations. However, Fabric's flexibility regarding consensus and incentive mechanisms, as well as pliant membership governance, make it useful for our implementation of the EHR framework and the prototyping application. Support for private channels is also a valued feature for potential future development, e.g. allowing analytics companies to analyse only parts of the data through private sub-ledgers instead of the full ledger.

3.2 Hyperledger Fabric

The binaries for Hyperledger Fabric are hosted in a GitHub repository [36]. There is currently no proper installer provided with the binaries. Instead, a script named bootstrap.sh, which is included in the repository, can be used to install the binaries along with some sample applications. See more about the requirements for installing the Fabric tools and binaries in the repository's README file [36] or in the prerequisites section of the Fabric documentations website [39]. For information on the development setup used in this thesis, see Appendix A.

A Fabric blockchain network includes three modular and pluggable components of special interest for developers [6]:

- An Ordering Service
- A Certificate Authority
- Membership Service Providers

In addition to these channel-wide components, each peer node in the network is composed of several other modular and pluggable components, e.g. state databases and chaincodes, which are presented in Section 2.3.4 and 2.3.2.

3.2.1 Fabric Tools

Two software tools are supplied with the Fabric binaries:

- Crypto Generator (cryptogen)
- Configuration Transaction Generator (configtxgen)

The Crypto Generator tool generates certificates and signing keys for the identities participating in the network [40]. These certificates and keys enable entities to sign transactions and verify identities. The tool can be configured in a YAML [41] configuration file, which is consumed by the tool upon execution. This provides a quick and simple way to produce cryptographic material for use in a development environment. In a production environment, however, a certificate authority (CA) will typically be used for generating the cryptographic material [6].

The Configuration Transaction Generator tool creates our genesis block and other subsequent configuration blocks [42]. Configuration blocks hold only configuration transactions, not regular transactions. The tool is configured in a YAML configuration file, where we specify the ordering service, anchor peers, MSPs, organizational policies and channel-wide policies, which were introduced in Section 2.3.3. The policies specified in this file are base policies and may be overridden by e.g. specific chaincode policies. Essentially, the policies specify which certificates are required to sign the data for a signature to be valid.

3.2.2 Software Containers

Containers for Fabric entities are created with Docker [24]. Running entities in isolated environments provided by container software is a good way to simulate distributed behaviour, even if all entities are in fact running on the same physical machine in a development environment. Each container simulates an entity that could just as well be running on another machine in another physical location, as it would in a production environment. Running entities in containers also eases administration and maintenance of entities, as faulty entities can be removed and replaced quickly.

Hyperledger Fabric provides ready-made Docker images for starting the different types of entities making up a network [36]. The types of Fabric entities used for this thesis are (Docker image names):

- fabric-peer: A peer node
- fabric-orderer: An orderer node
- fabric-ca: A Fabric CA
- fabric-couchdb: A CouchDB instance
- fabric-ccenv: System environment used to build chaincode
- fabric-javaenv: System environment used for Java chaincode
- fabric-tools: System environment for running software tools

The initial configuration of the individual entities is described in YAML configuration files used as input to the Docker Compose tool [43] when the network is first initialized. Docker Compose consumes the files and creates the specified Docker containers. The Docker images provided for each type of Fabric entity ensure that entities are ready to join the blockchain as soon as the required containers are up and running, without needing to install any additional software.

3.2.3 Ordering Service and Consensus

The ordering service comprises a set of orderer nodes, collectively known as the ordering service. Orderer nodes can be organized in a single orderer organization or as members of peer organizations. Figure 3.1 shows the various nodes making up a peer organization. Configuration of the orderer nodes depends on the type of ordering service used in the network. More information on ordering services is provided in Section 2.3.3.

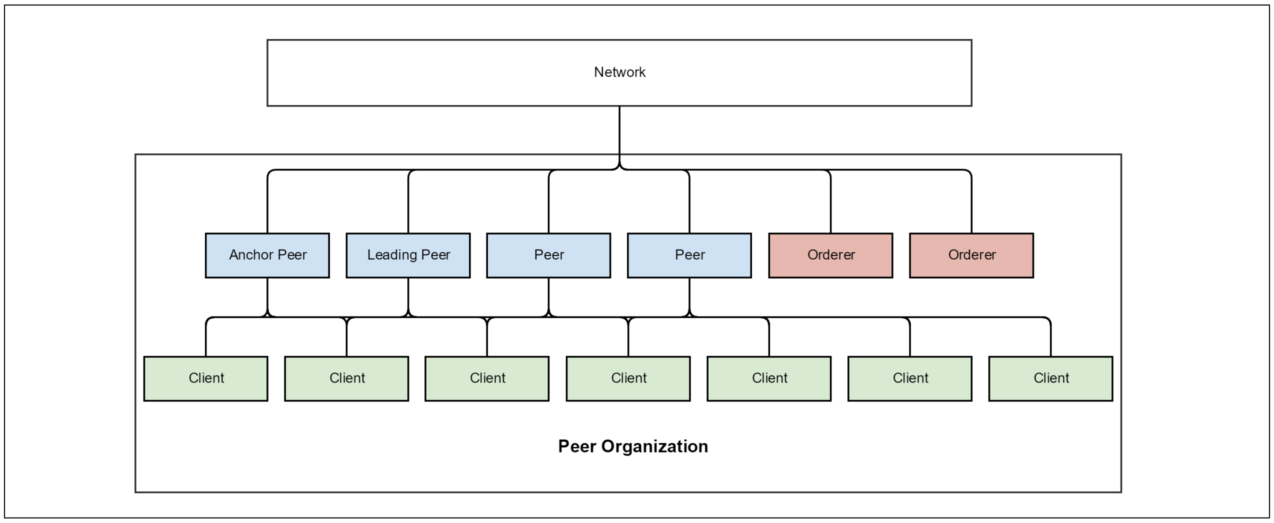


Figure 3.1: An illustration of the communication hierarchy within a peer organization with four peer nodes, two orderer nodes and multiple clients.

For a network using the Kafka ordering service, which until recently was the default ordering service implementation for production-ready systems, orderer nodes are typically placed within a single organization. This is due to the fact that orderer nodes must communicate with the Kafka cluster. This breaks any decentralization benefits gained from multiple orderer organizations. Although organizing nodes in a single organization does not limit where the actual physical nodes are placed, it is considered a centralized approach in terms of policy specifications and administration.

With the introduction of the Raft ordering service in Hyperledger Fabric v1.4.1 [29], communication with an intermediate node cluster is no longer required. It therefore makes sense to place orderer nodes in peer organizations, as opposed to organizing every orderer node in a single orderer organization. Spreading orderer nodes in different organizations increases the decentralization aspect of the network. All organizations that regularly participate in the network should provide orderer nodes to the ordering service.

As described in Section 2.3.5 and visualized in Figure 3.2, the transaction flow of a Fabric blockchain network, for any type of ordering service, is as follows:

1. The client issues a transaction proposal

2. The peer representing the client sends the proposal to required endorsers

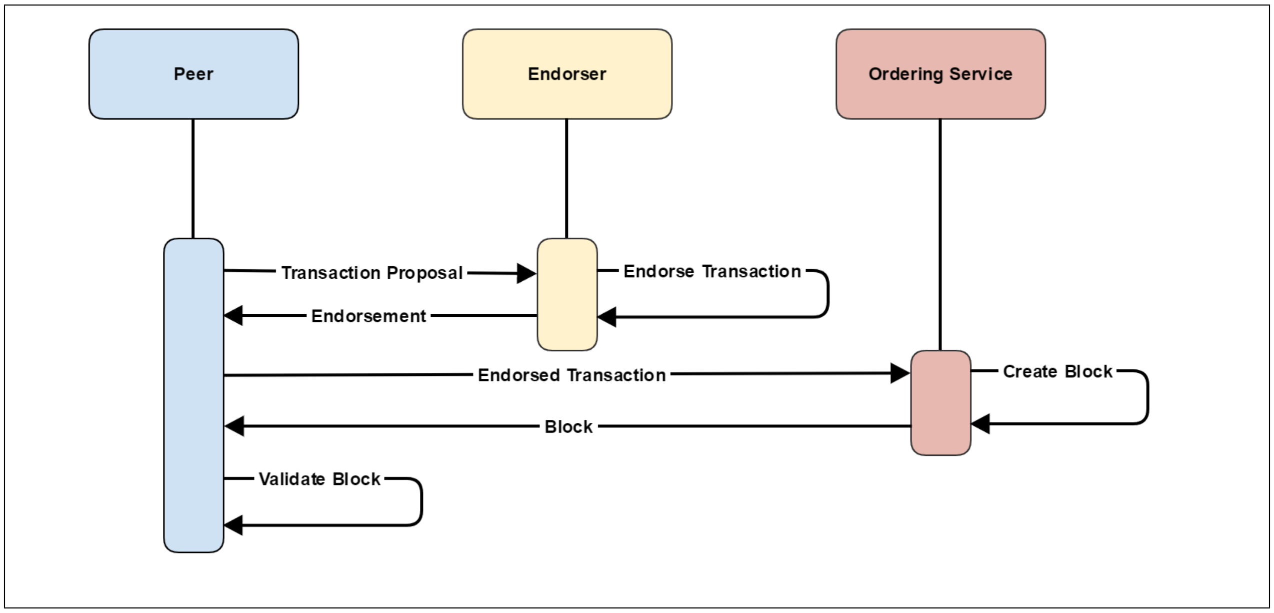


Figure 3.2: Hyperledger Fabric transaction flow featuring a single endorser.

3. The client checks if the proposal is correctly endorsed
4. The peer representing the client sends the transaction to the ordering service
5. The ordering service creates a block, likely containing several transactions
6. The block is distributed to the leading peer of each organization
7. Leading peers distribute the block to the rest of the peers on the channel
8. Peers validate the transactions of the block before adding it to the blockchain

If a transaction is not validated, it is marked as invalid when the peer places the block on its ledger. Clients invoking a transaction must therefore listen to transaction events even after the transaction has been sent to the ordering service, to make sure that the transaction was verified by the peers.

3.2.4 Certificate Authorities and Membership Services

Certificate authorities (CAs) handle identity registration and digital certificates for Fabric networks [6]. Entities communicating in the network identify themselves using certificates issued by one of the CAs in the network. The entities validating the certificates are the MSPs, as introduced in Section 2.3.3.

The CA is a pluggable component and multiple CAs can be used in a network at the same time, e.g. one for each organization. Fabric provides a default CA implementation known as the Fabric CA [44]. Typically, if the blockchain is to be implemented in existing systems, there will already exist a CA in the network. This existing CA would then be used instead of the Fabric CA component.

The configuration of each MSP is what enforces the policies specified in the network. Whereas a CA generates the required keys and certificates for an entity, the MSP is used to validate the credentials when an entity communicates with the network [6]. MSPs also enforce role checks on whether an entity is e.g. a client, member or admin of the domain. Policies might require that a certain number of entities of each role signs off on a transaction. The trust domains for each organization is specified by the MSPs based on which CA is authorized to issue credentials to members of that specific trust domain.

MSPs are part of the channel configuration and are kept synchronized with the consensus mechanism. There is one MSP for each organization in the channel. Local MSPs are also defined on each node in the network. These local MSPs control e.g. which entities can install chaincode on a peer.

3.2.5 Network Discovery

Peers in the network discover each other using the service discovery [45]. The discovery process uses anchor peers associated with each organization to explore the network and discover peers belonging to other organizations. This eliminates the need to provide static information about each peer in the network. For a peer to be visible to the service discovery process, it must have an external endpoint set in its configuration [45]. It must also know the address of at least one other node in its organization, which again must know the address of a different node. In this way, every node in the network is known to at least one other node, and the complete network can therefore be discovered.

The service discovery process uses information from the gossip protocol to identify connected peers. The gossip protocol continuously identifies which peers in the network are online or offline. It also broadcasts ledger data to other peers on the channel, so that peers that are out of sync can copy any missing blocks[28]. When a new block is created by the ordering service, the leading peer of each organization gossips this block to the rest of the peers in its organization. The protocol also allows for new peers on the channel to transfer ledger data over peer-to-peer connections [28].

3.3 Java SDK for Hyperledger Fabric

The Java SDK for Hyperledger Fabric [10] provides developers with an API for developing Java applications for interaction with Hyperledger Fabric networks. The API offers routines related to service discovery and invoking chaincode methods through transaction proposals or query requests.

3.3.1 Communication Clients

The Java SDK provides two types of client classes [10]:

- HFClient - Hyperledger Fabric Client
- HFCAClient - Hyperledger Fabric CA Client

The client classes are used for invoking methods to communicate with Fabric networks. An HFClient object comprises several methods for invoking the chaincodes installed on a channel. To represent a channel in the blockchain network, the object holds a reference to a channel object, which again holds a collection of node objects that construct the channel. Methods in the client are invoked from user context. The user context is associated with an object that is of a class implementing the Java SDK User interface [10]. A class implementing the User interface must hold information about the associated username, roles and affiliations. These fields of information are used by the CA when enrolling the user.

The HFCAClient class is used for handling events related to the CAs in the network, such as enrolling a new user or registrar. The Fabric CA implementation associate users with affiliations and departments, not with organizations. However, an organization is often mapped to a single affiliation and its departments. An affiliation can be broken down to a set of departments and sub-departments, typically on the form "department.sub-department". For instance, "hospital1.surgery" implicates that the user is part of the surgery department of hospital1. This affiliation would typically be mapped with an organization called hospital1 in the Fabric blockchain.

3.3.2 Query Requests and Transaction Proposals

Transaction proposal requests are constructed from the TransactionProposalRequest class and invoked through an HFClient object [10]. The requests hold the name of the chaincode and method that is to be invoked, along with any arguments required by the chaincode method. Queries are created from the QueryByChaincodeRequest class [10], which hold the same information as specified for the TransactionProposalRequest above.

The SDK do not provide methods for persistence [10]. As soon as a proposal is invoked through the HFClient, no other measures are invoked by the SDK. The application developer must therefore develop methods for listening to responses from endorsers and nodes that validate the transaction.

3.3.3 Collecting Endorsements

The SDK provide an interface for discovering nodes in the network [10]. This means that addresses and hostnames of nodes in the network do not have to be supplied manually to the application. Whenever a peer needs to discover nodes for endorsing a transaction, it simply utilizes the network's service discovery mechanism, which is discussed in Section 3.2.5. Service discovery then returns the names of the installed chaincodes, the selected endorsement policies and the name of available orderer nodes and endorsing peers.

The combination of endorsing peers can often be chosen in multiple configurations, depending on the endorsement policy selected for the chaincode. For instance, in a channel with two organizations maintaining two peers each, the following combinations of endorsing nodes can be selected for a chaincode that requires endorsement from at least one node from each organization:

(1) Organization 1: Peer 1 - Organization 2: Peer 1

(2) Organization 1: Peer 1 - Organization 2: Peer 2

(3) Organization 1: Peer 2 - Organization 2: Peer 1

(4) Organization 1: Peer 2 - Organization 2: Peer 2

The service discovery denotes these configurations as layouts. Each layout holds a list of groups, where each group holds a list of peers. Typically for most implementations, all peers within a group will be from a single organization. The layout also states how many endorsements are needed from each group. The SDK embedded application can then decide which layout it prefers and issue a transaction proposal to selected endorsers from this layout.

The application must adhere to the transaction flow presented in Section 2.3.5, which means that after sending the transaction proposal to the endorsing peers, the application must wait for the endorsement responses before sending the transaction to the ordering service. If the application sends a transaction that is not correctly endorsed to the ordering service, the transaction will be marked invalid by the peers validating the block that has been created. To make sure that the transaction is validated by the peers, applications must listen to the channel for events even after the block has been distributed in the network.

Additional documentation of each class and method provided by the SDK can be found in the Java SDK for Hyperledger Fabric GitHub repository [10].

Chapter 4

System Description

4.1 Overview

The resulting software system consists of two stand-alone software packages:

- HLF Network package
- Java Application package

The HLF Network package comprises the Hyperledger Fabric blockchain configuration and chaincode files. By running the scripts provided in this package, a functional blockchain network can be created. The package's intended place in a system is illustrated in Figure 4.1, where it is termed as the blockchain layer. As is shown in the figure, the package is deployed next to the existing system, not replacing it. Only minor parts of the existing system must be altered or extended for being able to operate with the new blockchain layer.

The Java Application package is merely developed to demonstrate the features of the HLF Network package and is not intended to be deployed in a production environment. It provides, essentially, a simulation of the database interface and can be replaced by

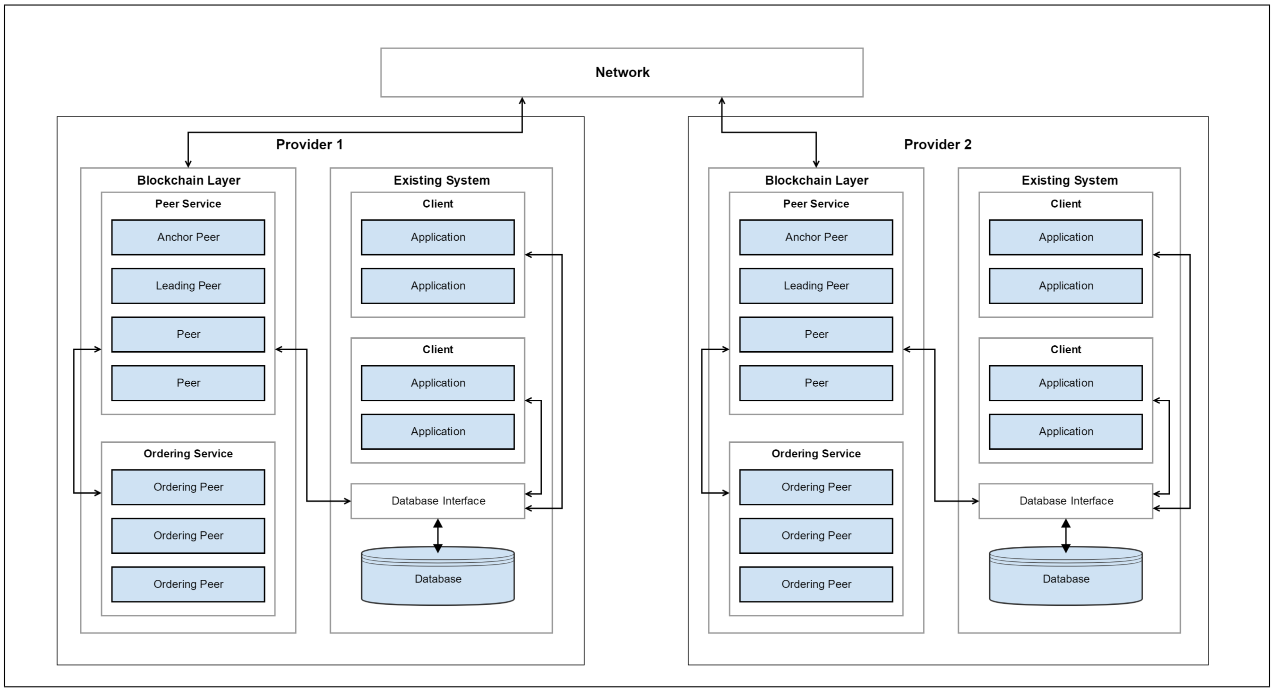


Figure 4.1: The architecture used when deploying the system, illustrated with a network of two providers.

any other application implementing a Fabric SDK. Not having to deal with a database and database management component makes demonstration of the blockchain network easier. However, the methods provided for communication with the HLF network can be used as a basis when implementing the Java SDK in an existing database system.

The SDK can be embedded in the database interface, essentially operating as a blockchain gateway to the database. It can be integrated either directly in a modified version of the interface or as a new wrapper application around the existing interface. Clients within the same organization can communicate directly with the SDK embedded interface, while requests coming from peers in other organizations are routed through the peer nodes.

In addition to the components illustrated in Figure 4.1, a CA must be present in the network. Most commonly, the package would be configured to be using an existing CA in the system. If such a CA is not available in the network, a new CA can be deployed in the blockchain layer for merely accommodating the HLF Network package.

Together, the HLF Network and Java Application packages demonstrate the core functionality obtained by the EHR framework [1] when using the Hyperledger Fabric blockchain model for implementation. Both packages are made available under the A-pache License Version 2 (Apache-2.0) [46].

4.1.1 HLF Network Package

The configuration files in the HLF Network package comprise both container and channel configuration, as well as configuration for creating cryptographic material with the Crypto Generator tool provided with the Fabric binaries. In production environments, material created by the Crypto Generator can be replaced by cryptographic material created by a CA. Even if the package is mainly intended for systems managing EHRs, it is also applicable to other similar recordkeeping use cases.

Any authorized application utilizing one of the Fabric SDKs, e.g. the Java Application package, can interact with the network and invoke chaincodes. Communication can also be initiated directly from the command line APIs of the various nodes in the network.

All nodes in the network, except the default Fabric CAs, have TLS enabled for secure communication and therefore only accept communication using the TLS protocol [47]. The CAs provided with the package are configured for development and testing purposes only. In a production environment, the CA implementation should be replaced with CAs that are already provided in the existing network.

4.1.2 Java Application Package

The Java Application package comprises both a front-end GUI and back-end business logic for interacting with the Fabric blockchain. The part of the application executing the business logic required to communication with the Fabric network relies on the Java SDK for Hyperledger Fabric version 1.4.2 [10].

The GUI is developed with JavaFX and is loosely tied with the business logic part of the application. This means that the business logic can be re-used for purposes where a GUI is not needed or where the GUI is replaced by some other form of interface. The application is created as a demonstration tool for showcasing the features of the HLF

Network package.

4.2 HLF Network Architecture

This section provides a design and implementation description of the components utilized in the HLF Network package. All main components are pluggable and can be replaced or edited, generally without rewriting other components. This makes the package flexible so that it can accommodate existing systems in the best possible way.

All entities are running in individual docker containers to simulate physically separated environments. For simplicity in testing for the thesis, all containers have been running on a single physical machine. The entities can, by specifying correct host information during configuration, be placed on any machine and communicate with each other over a network using TLS.

4.2.1 Configuration

The package contains six YAML files. These files specify the architecture and configuration details of the network. Figure 4.2 shows the directory structure of the package. The network is generated by the following configuration files:

- crypto-config.yaml
- configtx.yaml
- compose-with-raft.yaml
- compose-with-couchdb.yaml
- base/compose-base.yaml
- base/peer-base.yaml

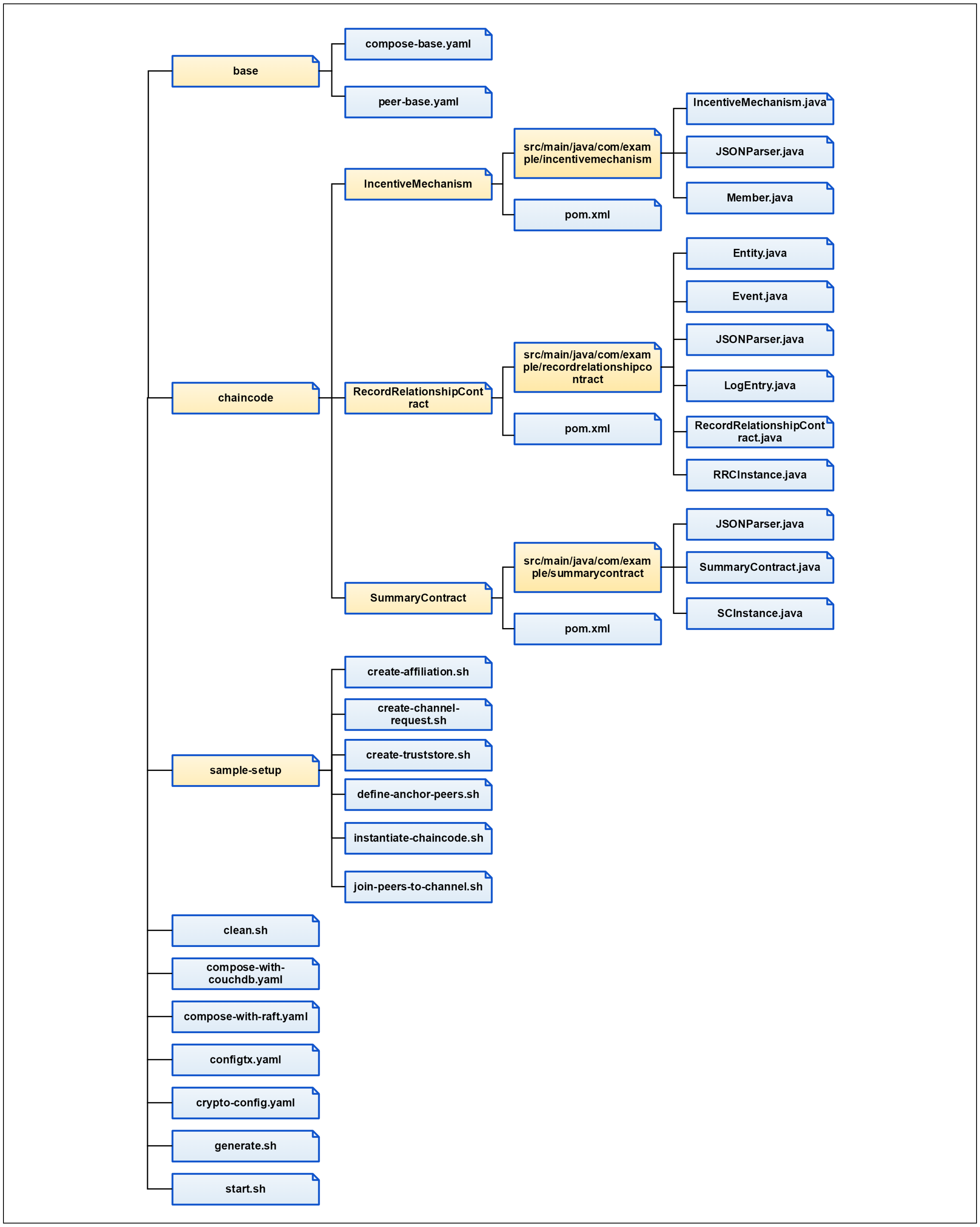


Figure 4.2: HLF Network package directory structure.

crypto-config.yaml governs the creation of cryptographic material to be used by the nodes in the network and is consumed by the Crypto Generator tool. The file includes hostnames and alternative names for all peer and orderer nodes in the network. Material for specified CAs are also generated. However, a CA can also generate this material by itself. The CA would typically also be used to create cryptographic material for other nodes in the network, thereby making the material created by the Crypto Generator unnecessary. The cryptographic material created by the Crypto Generator is placed in a directory named crypto-config.

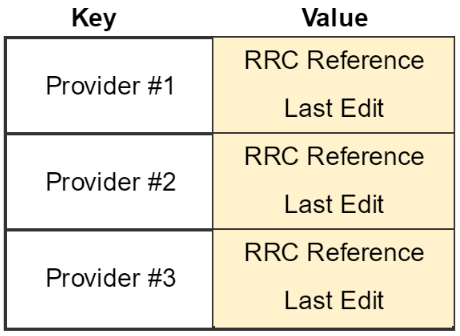
configtxgen.yaml specifies channel configuration details, such as the type of ordering service, policies and MSPs, as well as addresses for each organization's anchor peers. The MSP configuration must provide the path to the directory holding the generated certificates. If the Crypto Generator tool is used for creating the certificates, this directory will be located in a sub-directory of crypto-config.

The remaining four configuration files are used with Docker Compose, a tool for configuring and initiating Docker containers [43]. compose-with-raft.yaml specifies names, network addresses and dependencies for the required Docker volumes and services, while compose-with-couchdb.yaml provides additional configuration for CouchDB containers. Each peer in the network requires an associated CouchDB container for the state database. The CouchDB configuration file should be consumed by Docker Compose together with compose-with-raft.yaml, if the network is to use CouchDB as the state database.

The files located in the directory named base are extensions to compose-with-raft.yaml. peer-base.yaml specify configurations that are common for all nodes of a specific type in the network, while compose-base.yaml provides individually dependent container settings. This include unique names and addresses for all containers. Environment variables for MSPs and file paths to the cryptographic material, including certificates for TLS, must also be provided in this file.

4.2.2 Chaincodes

Chaincodes for the HLF Network package are written in the Java programming language. Java objects used to represent transaction data in the chaincodes are stored as



Key	Value
Provider #1	RRC Reference Last Edit
Provider #2	RRC Reference Last Edit
Provider #3	RRC Reference Last Edit

Figure 4.3: The representation of an SC stored on the blockchain.

JSON strings on the blockchain. The open source Gson Java library is used for converting JSON strings to Java objects and vice versa [48]. Writing both the chaincodes and the application in the same programming language allows for the same classes to be used in both packages if necessary.

The following three chaincodes have been implemented:

- Summary Contract
- Record Relationship Contract
- Incentive Mechanism

The Summary Contract (SC) and Record Relationship Contract (RRC) chaincodes represent the two smart contracts proposed in the EHR framework [1] and discussed in Section 3.1.1. The Incentive Mechanism chaincode implements the functionality required for the proposed incentive mechanism of the EHR framework and will be presented on its own in the next section.

In the SC chaincode, a Java class named SCInstance is used to represent the data that is stored on the blockchain. An object of this Java class holds the following information:

- The ID of the provider maintaining the record
- A unique reference to the RRC

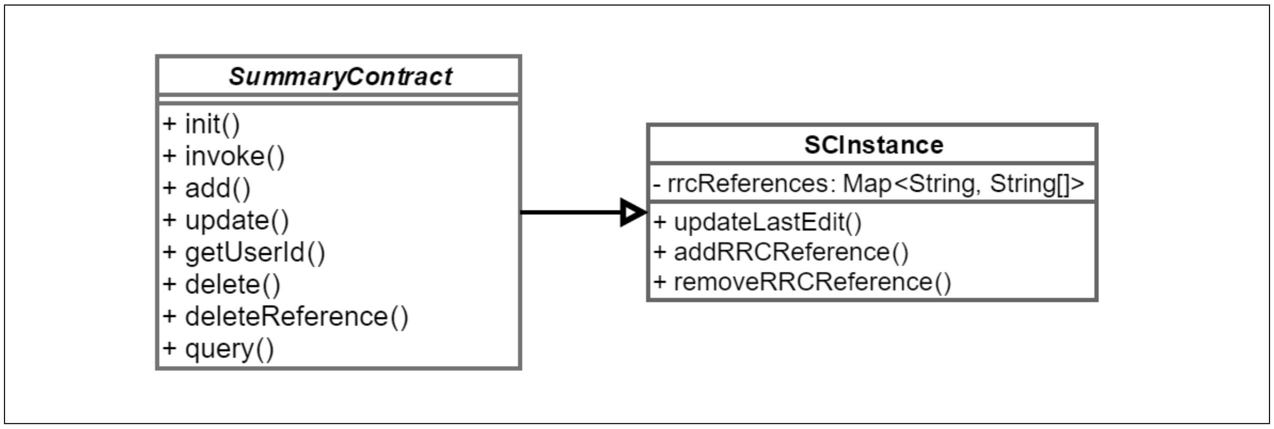


Figure 4.4: SC chaincode class diagram.

- The timestamp of the last edit to the RRC

The object contains a map with the provider ID as the key. The value of each entry in the map is a one-dimensional array of length 2, storing a reference to the RRC associated with the user-provider pair and a last edit timestamp of the RRC. Figure 4.3 shows a graphical representation for an SC stored on the blockchain. The figure shows the SC of a user with three user-provider relationships. The SC object is stored in the state database with the user ID as the key. The Java object is serialized to JSON before being placed on the blockchain. Figure 4.4 shows the class diagram for the SC chaincode.

The results of a query can be presented in the form of the returned JSON string or as a Java object after de-serializing the JSON string. A query for all provider relationships for a patient would simply include the user ID for the patient. The returned object would then hold the RRC reference and last edit timestamps for all providers where the patient has an RRC. The RRC reference is used to query the RRC chaincode to get the RRC associated with the reference.

The RRCInstance class is used to represent the data of an RRC. Objects of the class are stored on the blockchain with a unique RRC reference as the key. An RRCInstance object holds the following three pieces of information:

- An access control list of which entities are authorized to access the record
- The amount of significance associated with the record

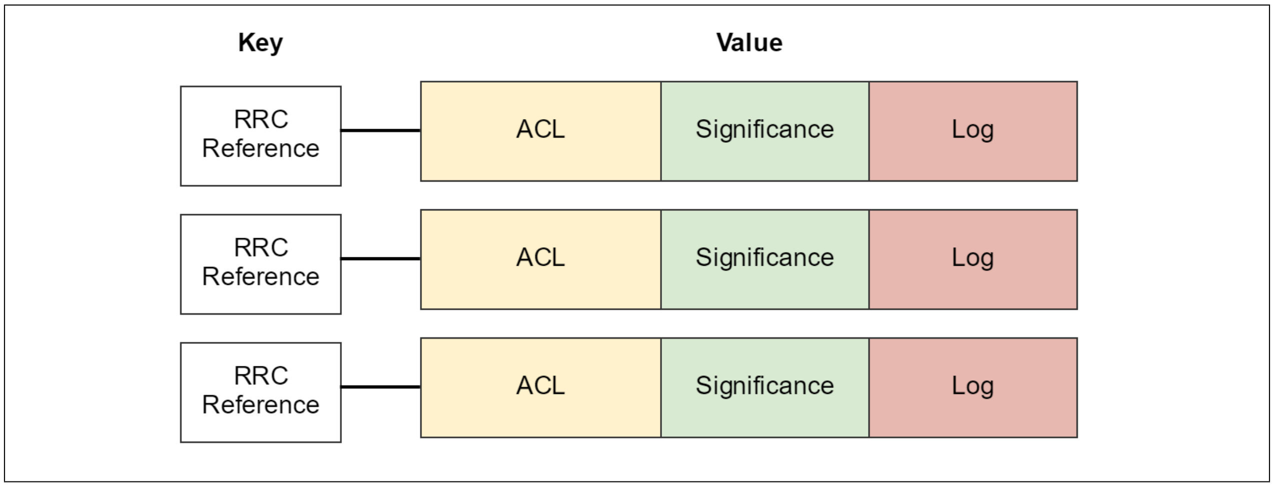


Figure 4.5: RRC objects stored on the blockchain, which use the RRC reference as the state database key.

- A log of all events that has happened to the record

The access control list (ACL) is constructed as a map. Strings holding either a client ID or an MSP ID are used as keys and lists of events as the value. The types of events are defined in a Java enum class and include events such as READ, WRITE and CREATE. The events listed for a client or MSP in the ACL control which types of events the entity is authorized to perform on the record.

Both individual client access and MSP-wide group access can be granted to a record. An MSP is typically associated with a single provider in a one-to-one relationship. To grant read access to all clients belonging to a provider named hospital1, we must add MSP hospital1 to the ACL with event type READ. To distinguish client IDs from MSP IDs, a suffix of "CLIENT" or "MSP" is added to the map key.

The RRCInstance object is saved in the state database using the RRC reference as the key, as illustrated in Figure 4.5.. This unique reference, however, cannot be created by a randomized generator in the chaincode, as this would cause each endorsing node to end up with a different reference and endorsement would therefore fail. The reference must instead be supplied by the application creating the RRC. Figure 4.6 shows the class diagram for the RRC chaincode.

The event log within the RRCInstance is a list of LogEntry objects. Each LogEntry object

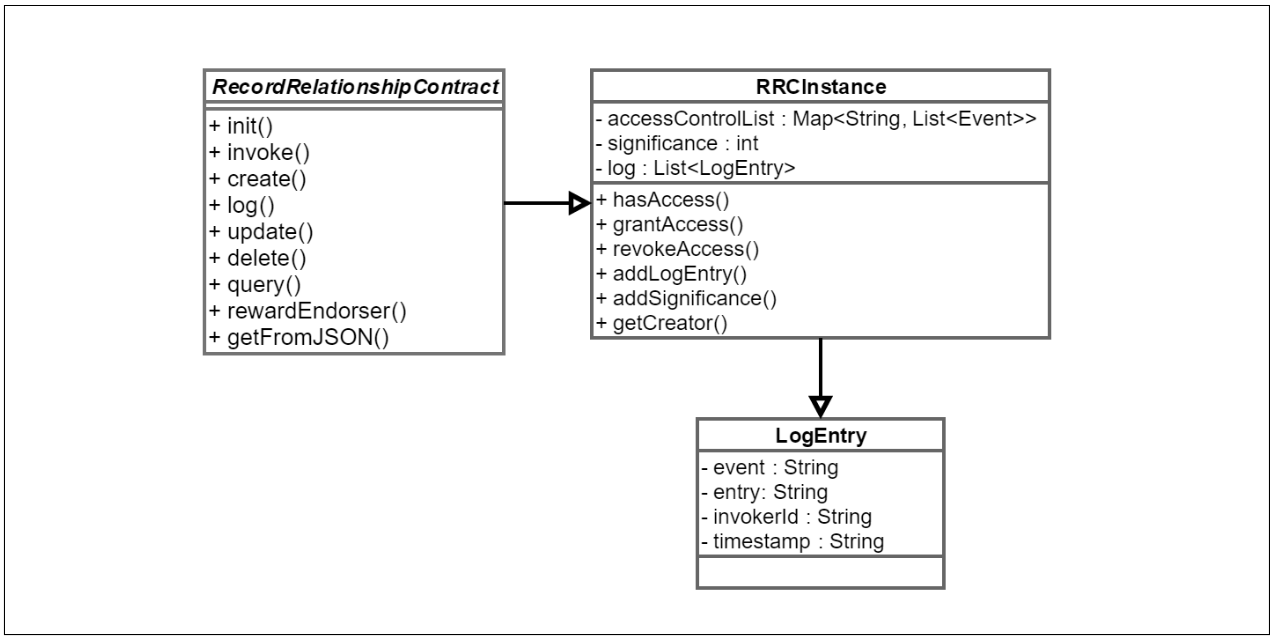


Figure 4.6: RRC chaincode class diagram.

comprises the following information:

- The type of event that occured
- The modifications made to the record
- The ID of the client that invoked the event
- A timestamp of when the event occured

The type of event that was executed must correspond with the events found in the ACL of the RRCInstance object. If a provider makes a query for reading the RRC of a patient, the chaincode will first check if the ID of the client is found in the ACL associated with the RRC. If the client is found in the ACL, the chaincode then checks if the client is authorized for the READ event.

If the client is authorized, a LogEntry object containing the type of event, the client ID and the timestamp of the invoked query is created and placed on the blockchain. Only when this procedure is fully completed is the RRC returned to the client. Measures for encrypting the LogEntry should be implemented in future versions of the software package and are discussed briefly in Section 6.2.

If the event was a WRITE, e.g. a practitioner enters a few sentences about the latest session with a patient to the patient's record, the new data entered is included in the LogEntry. Each time a WRITE event is added to the log of an RRC, the last edit timestamp of the associated user-provider pair in the SC is updated. To find the creator of an RRC, we query the RRCInstance for the LogEntry where the type of event equals CREATE.

Several internal calls between the chaincodes are used to invoke the required methods. For instance, when adding a new RRC to the blockchain, the RRC chaincode first sends a call to the SC chaincode. The SC chaincode checks if the user has an existing SCInstance object on the blockchain. If the user does not have an existing SCInstance on the blockchain, the SC chaincode must create a new SCInstance object.

The one-to-one relationship between patient records and providers means that there can only exist one RRC reference for each patient-provider relationship. If there already exist an RRC for the relationship, the SC chaincode will return an error. In a typical scenario where we want to check the log of an RRC, we must first query the SC chaincode to get the RRC reference and thereafter query the RRC chaincode to get the LogEntry object.

4.2.3 Ordering Service

The network is configured with Raft as the ordering service. As mentioned in Section 2.3.3, Raft provides less administrative overhead compared with Kafka, as there are no additional Kafka and ZooKeeper nodes to manage. The lack of a Kafka cluster also improves the decentralization aspect of the network, as orderer nodes do not have to communicate with a single-organization Kafka cluster. For the Raft ordering service, it makes sense to place orderer nodes within peer organizations in the network, as opposed to in a single orderer organization, and each provider can then register as many orderer nodes as it finds necessary.

To ensure that each organization provide the same amount of resources to the network, a policy on how many orderer nodes each organization should provide to the network must be specified. A larger organization would typically invoke more transactions and a heavier load on the network and could therefore be required to provide more orderer nodes to the network. The three organizations specified in the HLF Network package are by default assigned one orderer node each.

The network requires at least three active orderer nodes to achieve crash fault tolerance for one orderer node. Increasing the total number of orderer nodes in the network to e.g. five, will increase the crash fault tolerance to two orderer nodes. However, with the decentralized ordering service approach used in the HLF Network package, it is important that the property of crash fault tolerance do not depend on a single provider being present in the network. For instance, in a network with five orderer nodes, an organization maintaining three of these nodes might compromise the network if it goes offline or decides to leave the network for good.

4.2.4 Incentive Mechanism

The EHR framework [1] introduces a concept of significance associated with each provider. The concept is intended to be used in the decision on which node is responsible for creating the next block. A high value of significance indicates that the provider has a smaller chance to be selected with the task than a provider with a smaller amount of significance. As a reward for creating a block, the provider will receive an increase in its significance.

Hyperledger Fabric's three-step transaction process places the computational load of validating a transaction with the endorsing node, not with the node that creates the block, which is the case in most other blockchain frameworks. In the HLF Network package, we therefore let significance decide the endorsing nodes, not the orderer nodes.

The endorsement policies specified for the chaincodes in the HLF Network package require endorsements from at least one peer from two different organizations. This means that we need an endorsement from a peer in the organization that creates the transaction, as well as from a peer in one of the other organizations. However, the organization of the peer that created the transaction proposal will not receive a reward for the endorsement. When a block contains more than one transaction, the increase in significance is granted to every organization that has endorsed at least one transaction in the block, if the organization is not the same as the one that proposed the transaction.

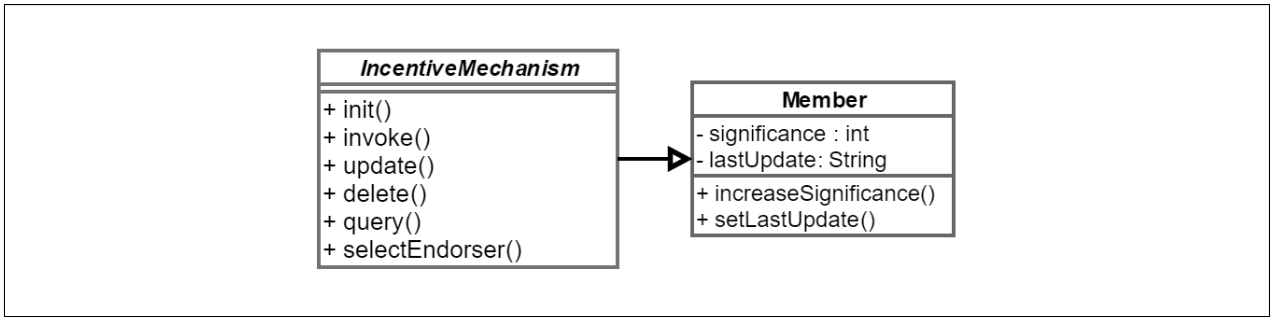


Figure 4.7: Incentive Mechanism chaincode class diagram.

The proposed concept has been implemented in the network with the Incentive Mechanism chaincode. The chaincode provides methods to select the next provider, to update the significance associated with a provider and to place the updated significance value on the blockchain. Figure 4.7 shows the class diagram for the chaincode.

The criteria used by the chaincode for selecting the next endorsers are as follows:

1. The organization has not been selected for endorsement in the past 10 minutes
2. The organization has the lowest amount of significance

Criterion 1 will always select the organization with the longest time idling. This ensures that even organizations with relatively high significance values will have to endorse transactions once in a while. If criterion 1 does not apply for any organization, criterion 2 will decide which organization is selected. The 10-minute criterion can be increased or decreased as to what will suit the network best.

The RRC chaincode automatically invokes the method to update significance, which is found in the Incentive Mechanism chaincode, when a transaction is executed. This update in significance constitutes the reward for the organization that endorsed the transaction.

Each RRC is associated with a significance reflecting the importance and uniqueness of the data in the record. This value of significance is added to the total amount of significance associated with the provider maintaining the record. The specific formula and implementation of how this value is to be calculated must be agreed upon by the providers in the network and is not part of the HLF Network package.

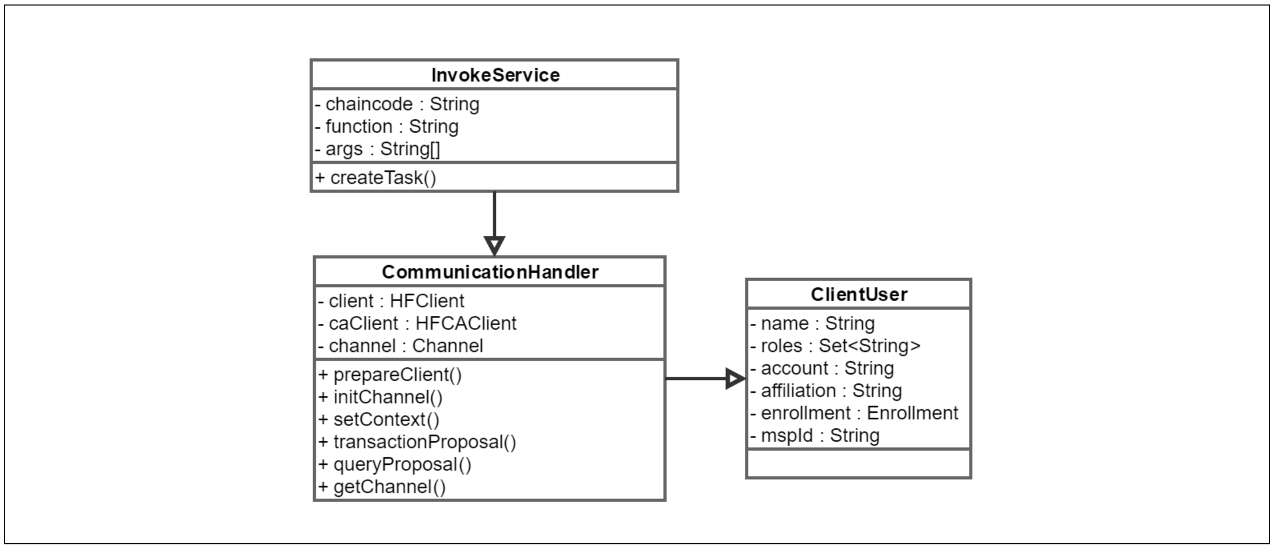


Figure 4.8: Java Application class diagram.

4.3 Java Application Architecture

This section provides a description of the Java Application package. The application uses Apache Maven [49] as build tool and requires the following dependencies for JavaFX [50], JAXB [51], Gson [48] and the Java SDK for Hyperledger Fabric [10]:

- org.openjfx - javafx-controls 11.0.1 - JavaFX
- org.openjfx - javafx-media 11.0.1 - JavaFX
- org.openjfx - javafx-graphics 11.0.1 - JavaFX
- org.openjfx - javafx-fxml 11.0.1 - JavaFX
- javax.xml.bind - jaxb-api 2.3.1 - JAXB
- com.google.code.gson - gson 2.8.5 - Gson
- org.hyperledger.fabric-sdk-java 1.4.2 - Java SDK for Hyperledger Fabric

JavaFX is used for creating the graphical user interface, while Gson is used for pretty-printing the JSON data so that it is easier to interpret when printing it for the user. JAXB

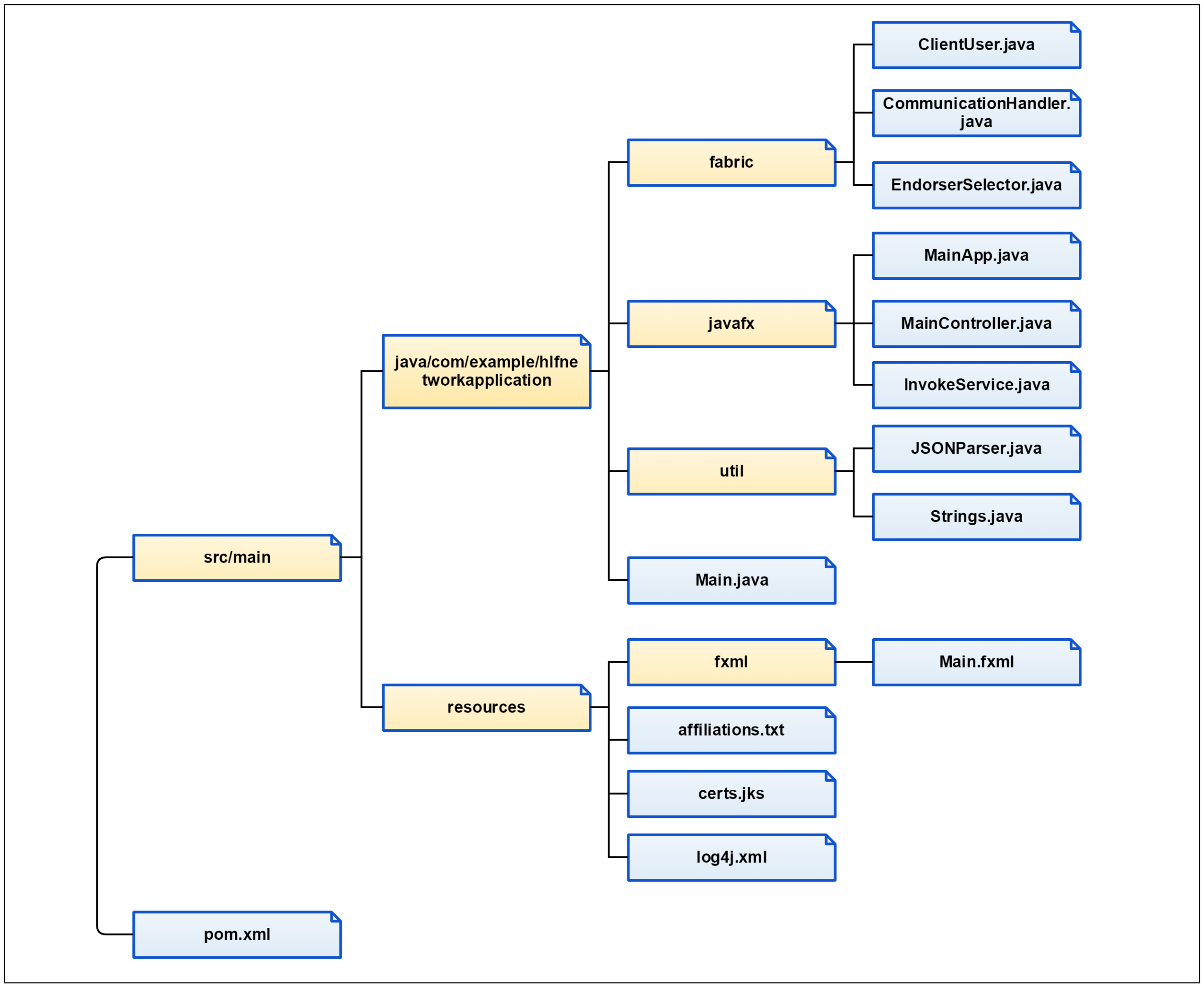


Figure 4.9: Java Application package directory structure.

is used for general mapping from Java objects to XML. The Java SDK for Hyperledger Fabric is used for communication with components in the HLF Network package.

Figure 4.8 shows a class diagram of the classes executing the business logic in the application. GUI operations are controlled by the MainController class, while communication with the Fabric network is handled in the CommunicationHandler class. Utility and intermediate classes are used to facilitate operations in these two classes. The main tasks of each class is further explained in the following subsections. For specific descriptions of each class method, see the Java class and method documentation provided in the source code. The complete file hierarchy is illustrated in Figure 4.9.

4.3.1 Hyperledger Fabric Integration

The Java Application package uses the Java SDK for Hyperledger Fabric for communication with the Fabric network in the HLF Network package. The SDK is included in the project files as a Maven dependency, specified in the project's POM file located in the root directory. All dependencies used in the package are specified in the POM file. Specific classes of the SDK are imported to classes in the application that require methods from the SDK.

Methods for communicating with the network are provided solely in the CommunicationHandler class. Before communication methods can be executed, an HFClient (Hyperledger Fabric Client) object, an HFCAClient (Hyperledger Fabric CA Client) and a Channel object must be initialized. These objects are created by running the prepareClient and initChannel methods provided in the class. Before initChannel is called, the application must enroll a user with the CA by calling the setContext method and provide the required enrollment details.

The User interface provided by the SDK is implemented in the ClientUser class. Objects of this class are intended to store the user details of a single user enrolled with the CA. ClientUser objects are stored to file so that the client can re-use the enrollment information even if the application has been restarted. The client can also re-enroll to a different user context whitout restarting the application. When re-enrolling, the previously enrolled user stored in a file.

The InvokeService class is called from the MainController class when a user requests communication with the Fabric network. The method provided in the service class runs in a separate thread, so that the GUI is still responsive to the user while it waits for communication with the network to finish. The class first checks whether the endorsing peers return the same results and thereafter waits for a transaction event to be broadcast on the channel, indicating if the transaction was accepted by the peers or not. For applications that do not use JavaFX, the InvokeService class can be used without extending the JavaFX Service abstract class.

4.3.2 Endorser Selection

By default, the implementation of service discovery provided in the Java SDK provides two methods for endorsement selection: either Endorsement Selection Random or Endorsement Selection Least Required Blockheight. The former selects endorsing peers randomly, as long as the peers comply with the chaincode endorsement policy, while the latter prefers endorsers with a smaller block height.

The Java Application package implements its own method for endorsement selection named Endorsement Selection Significance. The implementation selects peers from the layouts provided by the service discovery. A peer from the organization selected by the Incentive Mechanism chaincode and a peer from the organization invoking the transaction are selected for endorsement.

The layouts received from the service discovery are shuffled, so that if the same transaction is invoked twice, a new set of endorsing peers should be selected. A transaction would typically be invoked again if the first endorsement fails. This might happen if one of the endorsing peers does not manage to finish the endorsement process because of some internal error. Shuffling the layouts ensures that a different peer is likely selected for the next round of endorsement. Source code for Endorsement Selection Significance is found in the EndorserSelector class, which implements the SDK's EndorsementSelector class [10].

4.3.3 Graphical User Interface

JavaFX 11 [50] is used for all elements comprising the GUI. The GUI layout is set by the FXML file in the Java resources directory. Control and creation of GUI elements are separated from the business logic of the application and are executed in calls to the methods in the MainController class. Figure 4.10 shows the class diagram for the MainController class. Since the state database relies on JSON data representation, a JSONParser class is used for formatting the JSON data returned from state database queries to a humanly readable format. The JSONParser class uses methods from the Gson library [48].

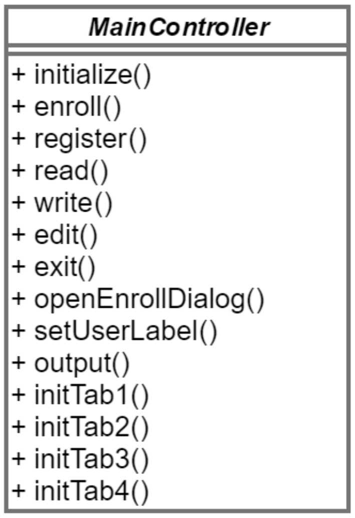


Figure 4.10: Methods provided by the MainController class.

When invoking business logic from the GUI, a JavaFX service class [52] is started. This class runs the logic in a new thread without blocking the user interface. Information about whether the methods have executed correctly or not is displayed in the text area at the bottom of the GUI. The service class calls methods in the CommunicationHandler class, which is the only class that utilizes methods from the SDK and initiates communication with the network directly.

4.4 Deploying the HLF Network Package

The HLF Network package can be deployed on any platform that satisfy the requirements listed in the prerequisites in the Hyperledger documentation [39]. A general description of how to run and configure the package is provided in this section, while a step-by-step user guide to set up and start the software is found in Appendix B. For configuration details of the Fabric network that we do not touch upon in this thesis, we refer to the original documentation supplied in the Fabric GitHub repository [36] and on the Fabric Read the Docs webpage [37].

4.4.1 Initial Configuration of the Network

The HLF Network is comprised of several configuration files. This section provides a summary of how to configure the files and the network properly. The configuration files are explained in the order they are consumed by the setup scripts, which are also provided in the package. Further documentation on the semantics of the configuration files is found in the Fabric GitHub repository, as well as in the documentation on Docker Compose [43].

The Crypto Generator, which is introduced in Section 3.2.1, generates the cryptographic material that is configured in the file named crypto-config.yaml. All peers and orderer nodes in the network must be listed in this file, so that corresponding cryptographic material can be created. Nodes must be listed under their associated organization. Hostname, common name and subject alternative names (SANs) must be specified according to the environment where nodes are deployed. This information is included in the certificates that are being generated. If a node operates from another address than the ones specified in this configuration, the certificate will be deemed invalid.

The next file that requires configuration is configtx.yaml. The path to the MSPs' cryptographic material must be specified in this file, along with at least one anchor peer for each organization. The ordering service used for the network must also be specified, along with hostnames, addresses and paths to cryptographic material for the orderer nodes in the ordering service. The network is governed by signature policies and implicit policies. Signature policies are used by MSPs to evaluate if signatures are valid, while implicit policies aggregate the results of signature policies in context of configuring the network. Both types of policies must be specified in configtx.yaml.

The remaining four configuration files are used for configuring the Docker containers and are consumed by the Docker Compose tool:

- compose-with-raft.yaml
- compose-with-couchdb.yaml
- base/compose-base.yaml
- base/peer-base.yaml

Each node in the network must be listed and configured as a volume and service in compose-with-raft.yaml. The same goes for CouchDB instances in the compose-with-couchdb.yaml file. If an IP address is omitted from a container configuration, it will be assigned an IP address dynamically. This requires that an application contacting the container uses DNS for hostname to IP mapping.

Common configuration details for all peers and orderer nodes are provided in base/peer-base.yaml. Paths to cryptographic materials and configuration details for TLS and gossip protocols must be set in this file. Finally, base/compose-base.yaml provides individual environment configuration details such as addresses and endpoints for each node. These configuration details must match with the details provided in the cryptographic material for TLS handshakes to succeed.

4.4.2 Network Lifecycle Management

Three bash scripts for quick setup and break down of the network on a Linux installation are provided. It is advised to run the network on a fresh virtual machine to avoid other programs and custom configurations from affecting the network. See Appendix B for steps on how to successfully run the network and Appendix A for information on the software environment used during development.

Run the scripts in the order listed below to successfully create and thereafter break down a network using the HLF Network package:

- generate.sh - Generate network artifacts and cryptographic elements
- start.sh [seconds] - Create docker containers and request channel creation from an orderer node
- clean.sh - Stop and remove docker containers, and remove all generated artifacts and elements (Note: this script removes all Docker containers in the system)

The first script invokes the Configuration Transaction Generator and Crypto Generator tools. Use of the Crypto Generator is for testing purposes only and should be replaced by a CA in a production environment. For testing, the Crypto Generator offers an easy way to generate the necessary cryptographic certificates and keys for each identity in the network before the network is brought up.

The next script starts by creating the necessary Docker containers. The required amount of time for initializing the containers may vary depending on the system's hardware configuration. If container services are invoked before they have been initialized, an error will occur. The start.sh script takes the number of seconds to wait for initialization as an argument when running the script. In general, no more than 20 seconds should be required for all containers to get ready for receiving communication requests. See Section 5.3.2 for the average amount of time required for intialization during testing.

Docker containers offer a simple way to run several isolated entities on a single machine, communicating in the same manner as if they were on separate physical machines. To connect with entities running on other machines, the corresponding hostnames and IP addresses must be configured in the YAML configuration files.

The configuration files bundled with the HLF Network package are initially configured with three organizations representing each of the made-up providers named hospital1, pharmacy1 and practitioner1. The organizations are configured with two peers each, as well as a single orderer node per organization. New organizations and peers can be configured either in the configuration files before the network is started or by utilizing one of the Fabric SDKs while the network is already up and running. The Fabric binaries also provide some tools for adding organizations and peers using the command line.

Secure end-to-end communication is achieved with TLS. All entities in the network, except the CAs, are configured for communication over TLS. The default Fabric CAs have TLS disabled since they are provided for demonstration and testing purposes only. However, TLS can be enabled for CAs as well by adding the tls.enabled flag to the CA start command in compose-base.yaml, as well as the file paths to the CA's TLS certificate and key. If the default CA registrar name or password is changed, these details must also be updated in compose-base.yaml. Note that the CA certificates created by the Crypto Generator require the full hostname of the CA to be used in communication.

After the first two scripts have been executed, the network is up and ready for communication. Note, however, that a chaincode is not instantiated on a node until the first time the chaincode is called on that specific node. This instantiation process may cause transaction requests to timeout if it takes too long. The failed transaction request can then be re-initiated later. This behaviour ensures that system resources are not wasted on chaincode containers that are not in use, as redundant peers only install chaincode after the main peer is down and new peers start to receive chaincode invocations and endorsement requests.

The scripts used by generate.sh and start.sh to bring up the network are located in a directory named sample-setup:

- create-truststore.sh - Creates a trust store to be used with the Java Application package
- create-channel-request.sh - Sends a create channel request to the ordering service
- join-peers-to-channel.sh - Joins the listed peers to the channel
- define-anchor-peers.sh - Defines anchor peers for each organization
- instantiate-chaincode.sh - Installs chaincodes on the listed peers and instantiates chaincodes on the channel
- create-affiliations.sh - Adds the listed affiliations to the Fabric CAs

The files are invoked in the order listed above. When new peers are added to the configuration files, they must also be added to join-peers-to-channel.sh and instantiate-chaincode.sh, if they are to join the channel and run chaincode. If new affiliations are required, they should be added to create-affiliations.sh or added manually by using the command line interface of the CA container.

4.5 Deploying the Java Application Package

The source code for the Java Application package is organized in a Maven hierarchy with the POM file located in the root directory. Java classes are located in the packages found in src/main/java/com/example/hlfnetworkapplication, while resource files are located in src/main/resources. Classes are organized in the following packages:

- fabric - Classes used in communication with the Fabric network
- javafx - Classes for GUI elements and interaction
- util - Utility classes such as JSON manipulation and String constants

With the Java Application package, the user can simulate database actions and thereafter query the blockchain to check if it updates correctly. The user can operate with different client identities from the same application instance, to see how access control is enforced.

4.5.1 Required Resource Files

The application requires a text file and a trust store to be placed in src/main/resources before the application is loaded:

- affiliations.txt - List of affiliations managed by the CAs
- certs.jks - Trust store generated by generate.sh in the HLF network package

The list of affiliations is required when the application is enrolling a user with the CA and must correspond with the affiliations added to the CAs. Each line in the text file represents an affiliation in the following format, where semicolon is used as a word separator:

```
affiliation-name;msp-name;ca-name;ca-url;registrar-name;registrar-password
```

The trust store contains certificates created by the Crypto Generator and is required for presenting certificates on behalf of the entities enrolled in the application. Without the trust store, the Java application will not be able to communicate over TLS.

The configuration files in the HLF Network package are configured with static IPs for peer and orderer nodes. This makes it possible to create cryptographic material with the

Crypto Generator before the Docker containers are created. If not, cryptographic material must be created after each container has been assigned an IP dynamically.

If the hostname to IP address mapping is not available over DNS, the Java application must read hostnames and addresses from the system's hosts configuration file. In Linux operating systems, the hosts file is located in /etc/hosts. This file should be updated with the IP address of each Docker container and the associated hostname mapping. Hostnames for the Docker containers are specified in base/compose-base.yaml in the HLF Network package, which was discussed in Section 4.2.1. The mapping for the initial network configured in the HLF Network package is as follows:

172.18.0.40 peer0.hospital1.example.com

172.18.0.50 peer1.hospital1.example.com

172.18.0.60 peer0.pharmacy1.example.com

172.18.0.70 peer1.pharmacy1.example.com

172.18.0.80 peer0.practitioner1.example.com

172.18.0.90 peer1.practitioner1.example.com

172.18.0.100 orderer0.hospital1.example.com

172.18.0.110 orderer0.pharmacy1.example.com

172.18.0.120 orderer0.practitioner1.example.com

Make sure that the Docker containers are assigned IP addresses that are not already in use and that they are correctly included in the node's certificate.

4.5.2 User Interface Interaction

The user will meet a dialog box to enroll a user on startup. The application will enroll the selected username with the CA using the registrar credentials specified in affiliations.txt.

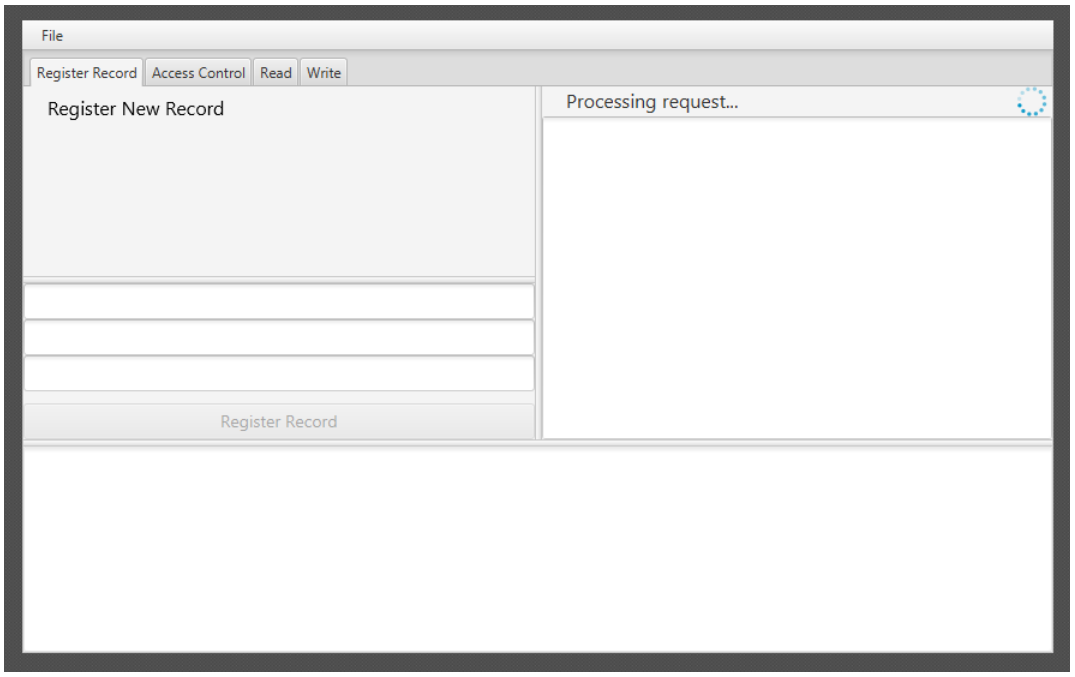


Figure 4.11: Java Application user interface.

A ClientUser object is created and saved in a directory named users, so that user information is preserved on application exit. If a selected user is already stored in the users directory, the application will read the ClientUser object from the file instead of trying to re-enroll the user with the CA. The enrollment dialog can also be opened from the File menu after the application has been started.

The application features a simple and intuitive user interface shown in Figure 4.11. The interface presents the user with four different tabs, each representing a different use case:

- Register Record - Add a new RRC to the blockchain
- Access Control - Edit the ACL of an already existing RRC
- Read - Query the state database for an RRC and its associated Log
- Write - Execute a new WRITE event on an RRC

Each tab has the same layout, consisting of a horizontal splitplane that holds a vertical splitpane in the upper section and a simple text area for system output in the lower section. The vertical splitpane contains user interaction elements in the left section and a text

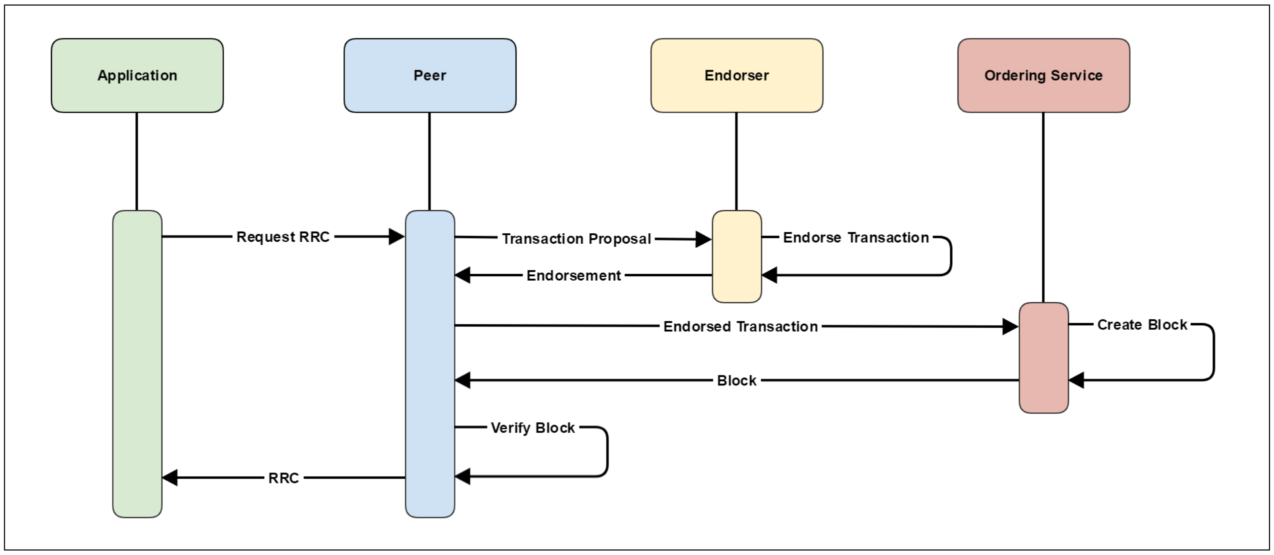


Figure 4.12: The transaction flow embedded in the Java application.

area in the right section. A progress indicator is displayed in the top of the text area when the application is waiting on response from the Fabric network. Any returned payload of interest for the user is displayed in the text area below.

Figure 4.12 shows the event flow when the user issues a query for an RRC. Before the RRC is displayed to the user, a new transaction adding the READ event to the RRC log is created. The application must receive endorsements from all required endorsers before the transaction can be sent to the ordering service. The block created by the ordering service is then distributed to all peers on the channel. Each peer validates the transactions in the block and marks each transaction as valid or invalid.

The same flow is executed when a new RRC is added to the blockchain or when an RRC is updated, however, without a return value of interest. Information on whether transactions have completed successfully or failed is printed in the text area at the bottom of the user interface.

Chapter 5

Design and Performance

5.1 Benefits over Traditional Healthcare Systems

The HLF Network developed in this thesis brings several improvements in the areas of privacy and auditability over traditional EHR systems that do not embed blockchain technology. Traditional systems rely on providers' local systems to provide auditing of record events. The implementation and quality of such auditing measures might therefore be inconsistent. This places much trust with a provider's ability to maintain its own systems securely.

In the HLF Network package, the auditing process is a decentralized process executed in co-operation between all participants in the network. This means that auditing is unbiased and does not rely on access to a sole provider's system logs. This increases transparency in the system and eliminates the possibility of faulty auditing processes.

The package also introduces improved privacy measures in the form of common ACLs for every record. Combined with the logs that provide auditing features to the network, a patient has full insight in which entities have accessed its logs, as well as which entities are authorized to do so.

In short, some of the benefits of the developed blockchain solution over traditional EHR systems for auditing and privacy concerns are:

- Auditing of the EHR system can be performed with data collected from any node in the network
- Since all nodes agree on the auditing process and what data is logged, the auditing process is unaffected by providers' internal auditing policies
- A single provider cannot tamper with the logs of neither its own nor other providers' records
- Authentication of entities invoking various database operations can be performed in both the blockchain layer and in the database interface
- Lists of authorized entities, ACLs, are placed on the blockchain to avoid unauthorized changes of access rights and to ensure that only authorized personnel can access a patient's record

5.2 Validating the System

The HLF Network package presented in Chapter 4 is a functional blockchain network that can be configured and implemented in existing recordkeeping systems, e.g. EHR systems. The integrity of the system has been verified during testing with the Java Application package. Testing with the Java Application package comprises the following tests:

- Add a new RRC to the blockchain
- Edit the ACL of an RRC that is already added to the blockchain
- Read an RRC stored on the blockchain
- Simulate writing to a record, which triggers the event to be logged

No signs of data corruption or other malfunctions were discovered during testing with the Java application.

5.2.1 Block Creation and Policies

The framework proposes a status field indicating if the RRC is successfully added to the blockchain to be included in the RRC [1]. However, this field is not part of the RRC implementation of the HLF Network package, as applications interacting with the Fabric network can utilize a flag associated with each transaction for the same purpose. The embedded three-step transaction flow of Hyperledger Fabric ensures that a transaction that is not endorsed correctly is marked as invalid when peers place the block on the blockchain.

The three chaincodes developed for this thesis set an endorsement policy that requires one peer from the invoker organization and one peer from another organization to endorse a transaction. The peer within the organization invoking the transaction is used to verify that the invoking peer is not faulty, just as endorsers from other organizations also do. Meanwhile, the peer selected from the other organization ensures that no organization is trying to spoof the network. The organization invoking the transaction does not get a significance increase. This means that there is no incentive to create transactions, only to endorse transactions made by other organizations.

The chosen endorsement policy is tolerant to misbehaving organizations, as long as two or more of the organizations do not perform a coordinated effort to spoof the network. If two coordinating organizations select each other's peers for endorsement, they can successfully propose and thereafter endorse any transaction they would like to. However, in a network composed of essentially trusted organizations, it is not expected that two organizations would operate in such a way.

It is possible to restrict peers of a specific role to endorse transactions on behalf of an organization. These roles are specified during channel configuration. Such roles could denote the various physical locations of peers or the physical location of the clients assigned with connecting to that specific peer.

5.2.2 Resilience to Fault and Misuse

The Raft ordering service is crash fault tolerant. New leaders are elected when the current leader node goes offline, e.g. with three nodes in the ordering service, the network can tolerate to lose one node and still be operational with the two remaining nodes. For a five-node ordering service, the network can withstand the loss of two nodes. In other words, if the majority of orderer nodes are still active, the network can withstand to lose a node. The Kafka ordering service is also a crash fault tolerant service.

The Raft ordering service will serve as a starting point for the implementation of an official byzantine fault tolerant (BFT) ordering service for Hyperledger Fabric. Some unofficial BFT ordering services have been developed, but none are currently included in the official Fabric releases.

The lack of BFT in the current ordering services offered with Hyperledger Fabric means that the system does not sustain the robustness to handle malicious responses from compromised nodes in the network. However, for the closed healthcare system use case targeted with this thesis, the lack of BFT does not impose an immediate threat to the system. The combination of strict access control and network monitoring of any node in the network of a healthcare provider and the fact that the Fabric blockchain is permissioned means that the appearance of malicious nodes in the network is unlikely.

5.3 Performance at Scale

In versions prior to v1.1.0, Hyperledger Fabric voting-based consensus performed worse when scaling for an increased number of nodes than comparable blockchain implementations. However, for versions after v1.1.0 there are no indications that the framework has problems with scaling [53].

Performance in the Fabric blockchain has two potential bottlenecks:

- Peer node endorsements

- Ordering service throughput

Our chosen endorsement policy of requiring only one external node to endorse our transaction ensures that the endorsements bottleneck is minimized. The peer node endorsements bottleneck would only increase if we add too many clients compared to peer nodes in the network. If this happens, we must introduce additional peer nodes to handle the endorsements bottleneck.

As the ordering service is a queuing system that batches transactions into blocks, increasing the batch size might help for throughput issues. As each query of the blockchain will result in a log transaction, we might end up with a large amount of transactions in the system as the number of clients increase. However, the workload involved in creating a block is small, as no validation of the data is performed at the ordering service. The computationally intensive tasks are solely executed at peer nodes. The potential bottleneck imposed by the ordering service should in most cases be negligible.

Testing and measurements obtained in regard to the implementation created for this thesis are limited by physical constraints in the testing environment. The testing simulates virtual nodes with the use of containers running on a single physical machine. Each container presents an isolated environment where software can operate as if it was running on a separate physical machine and communicate with other containers through loopback network interface.

5.3.1 Increasing Peer to Orderer Ratio

The configuration files supplied with the HLF Network package are initially configured with two peer nodes and a single orderer node for each organization. This gives a 2:1 peer to order ratio, which for larger networks means that we will end up with way too many orderer nodes in the network. For larger networks, a 10:1 peer to orderer ratio would be a more reasonable configuration.

For a small organization with a relatively low amount of significance compared with other organizations in the network, a larger number of endorsing peers is required. This requirement will slowly deteriorate as the organization increases its significance com-

# of Peers	# of Orderers	Running time
2	1	3.7 seconds
2	2	3.8 seconds
4	1	4.6 seconds
4	2	4.6 seconds
6	1	5.1 seconds
6	2	4.9 seconds

Table 5.1: Average running times of a single iteration of the three-step transaction flow with different numbers of peers and orderers selected per organization.

pared to other new organizations, and the organization will be able to reduce its number of peers. However, a larger organization with a higher amount of significance might also need a large number of peers for both endorsing its own transactions as well as providing endpoints for its many client applications that are running at the same time.

In general, no more than two or three orderer nodes provided by each organization are required. The exact number of orderer nodes per organization will depend on the number of peers per organization in the network. In a network with a large number of peers but few organizations, each organization will have to provide more orderers than in a network with fewer peers and a higher number of organizations. In general, a significantly higher number of endorsing peers than orderers are needed in the network.

The final decision on how many peer nodes and orderer nodes are required in the network will always depend on the expected connectivity of the nodes in the network. If nodes in the network have low connectivity and regularly experience connection issues, more redundant nodes must be added to the network.

The network has been tested in several peer and orderer configurations. The running time for executing a query for an RRC from the Java Application package is shown in Table 5.1. The query was executed by one peer in each organization at the same time. The listed running time is the average value for all the peers, in three attempts. The results show an average variation in running time of about 25 percent. It is likely that most of these variations come from system overhead and are not caused by the performance of the actual blockchain network.

The number of nodes feasible for testing is bounded by the constraints induced by the

# of nodes	Setup time for Kafka	Setup time for Raft
9	66 seconds	10.6 seconds
15	71 seconds	12.2 seconds
21	88 seconds	12.8 seconds

Table 5.2: The average required setup times for the Kafka and Raft ordering services in the development environment specified in Appendix A.

machine running the containers. For a realistic full-scale test, nodes should be placed in physical disparate locations as to simulate organizational setup, administration and overhead.

5.3.2 Raft vs Kafka Ordering Service

The initial version of the HLF Network package used a Kafka ordering service in Hyperledger Fabric v1.4.0. As of v1.4.1, the new Raft ordering service was introduced as an option. The final version of the HLF Network package uses the Raft ordering service. In terms of operational and administrative complexity, using the Raft service over Kafka reduces the complexity significantly, as discussed in Section 2.3.3. This is especially noteworthy for large systems spanning multiple organizations. The Raft service requires significantly less inter-node ordering communication and system overhead.

Another drawback of the Kafka implementation is that the Kafka cluster must be run as a single organization in the network. This means that all orderer nodes will communicate with the same centralized cluster. This also introduce implications when scaling the network to support a large number of nodes.

For a system running on the setup described in Appendix A, the initial time for all specified nodes to be up and ready to accept communication is reduced from 1-2 minutes to only 10-15 seconds when applying the Raft ordering service over Kafka. This means that the Raft setup is about six times quicker than the Kafka setup. This is bound to the fact that there is no need for any Kafka or ZooKeeper nodes to be initialized. The average time measured for the Docker Compose setup to finish for the environment described in Appendix A is shown in Table 5.2. The result is an average of three attempts for each configuration.

Chapter 6

Conclusion

6.1 Concluding Statement

The thesis introduces a cost effective and adaptable blockchain implementation for improving auditability and privacy of EHR systems. The proposed blockchain system builds on the EHR framework presented in [1] and has been successfully implemented in Hyperledger Fabric. The implementation is verified to work according to the description provided in this thesis and it can be concluded that the framework can draw benefits from an implementation in Hyperledger Fabric.

In the thesis we have described chaincode implementations for the proposed smart contracts and incentive mechanism. The implementation of the incentive mechanism has been tailored to the three-step transaction flow embedded by Hyperledger Fabric, which means that instead of selecting an orderer to create the block as would typically be done in a traditional transaction flow, the mechanism selects peers for the more computational heavy task of endorsing a transaction. If a transaction is successfully added to the blockchain, the incentive mechanism rewards the endorsers with a value of significance.

Essentially, the significance value indicates an organization's value to the network. The criteria used for endorser selection make for a combined decision based on a provider's significance and the time it has been idling without endorsing transactions. The result is that mainly new providers with a low amount of significance are selected for the task of endorsement. However, due to the propsed criteria, organizations with large values of significance will also have to carry out endorsement tasks every now and then. In time, the value of significance associated with each organization in the network will even out, creating an even work-balance in the network.

The proposed chaincode implementations place logs of events happening to an EHR on the blockchain, which make for immutable auditing capabilities. Chaincodes also embed access control features directly on the blockchain. The ACLs governing which entities are authorized to access a record is placed on the blockchain to avoid malicious edits to an ACL, while the process of authenticating a client for access to an RRC is conducted in chaincode checking for entries in the ACLs. This makes for strict enforcement of access control policies, improving privacy of the EHR system.

One of the main benefits of the implementation is that it can be deployed on top of already existing systems, requiring changes only to the database interface embedded in the existing system. The various entities and components of the implementation can be changed in plug-and-play fashion due to the use of modular components and individual containers for each entity and component. The modular architecture of the implementation eases the adoption process and is a big incentive when assessing whether to adopt the system or not.

The Hyperledger Fabric blockchain implementation does, however, impose some constraints on the framework. One of the imposed constraints is that every transaction must be deterministic, which means we cannot generate non-deterministic values within chaincodes, and that a transaction must be executed at every peer that validates it, which imposes some computational load. A discussion of potential future improvements of the two proposed software packages are discussed in the following section.

6.2 Further Development

The HLF Network package currently comprises the features described in the first draft of the EHR framework [1] presented in 2018. The framework was updated in March 2019, mainly introducing new features of encrypting the RRC log and using a collective authority for distributing the secret key. Hyperledger Fabric already provides support for encrypting objects and it should be a feasible task to implement a collective authority for key distribution as well. Implementing these features should be the next step in any further development of the package.

As mentioned in Chapter 2.3.3, a new Byzantine fault tolerant ordering service for Hyperledger Fabric is in development. As this new service is being built on top of the Raft ordering service, which is already used in the HLF Network package, it is likely that changing to the new service will only be a minor task.

Unique references for the RRCs must currently be supplied to the chaincode from client applications. As mentioned in Section 4.2.2, randomized references cannot be created within the chaincode, as chaincodes have to be deterministic. However, creating unique references from some sort of pattern could be a possible solution to being able to create references inside the chaincode.

Yang et al. [1] also propose a method to calculate the significance associated with each provider's EHRs as they are added to the blockchain. This amount of significance is intended to indicate the value and quality of the record. The value is associated with the amount of new information the record brings to the network and the relative importance of each new field of information.

However, an implementation of this feature would require that knowledge about the specific data standard used for EHRs within the system is available in advance. This is typically not the case, as systems are currently using a wide range of formats. It is therefore not feasible to implement this feature in a general manner for inclusion in any of the packages developed for this thesis. The feature must instead be implemented specifically for each system.

Although methods in the Java Application package can be invoked from existing EHR systems without using the GUI, and therefore be used as a back-end for existing applications, systems should instead embed one of the Fabric SDKs directly in the database interface. The communication methods provided in the Java Application package, including the method for endorsement selection, can be re-used in a database interface.

Bibliography

[1] G. Yang, C. Li, and K. E. Marstein, "A design of blockchain-based architecture for the security of electronic health record (EHR) systems (in review)," *Concurrency and Computation: Practice and Experience*, 2019.

[2] Office of the National Coordinator for Health Information Technology, "What is an electronic health record (EHR)?." `https://www.healthit.gov/faq/what-electronic-health-record-ehr`. Retrieved April 25, 2019.

[3] J. L. Fernández-Alemán, I. C. Señor, P. n. O. Lozoya, and A. Toval, "Security and privacy in electronic health records: A systematic literature review," *Journal of Biomedical Informatics*, vol. 46, no. 3, pp. 541–562, 2013.

[4] Roberts, Lucien W. and Towey, Emily W. G., "Risk management: Medical records." `https://www.physicianspractice.com/pearls/risk-management-medical-records`. Retrieved March 1, 2019.

[5] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." `https://bitcoin.org/bitcoin.pdf`. Retrieved March 25, 2019.

[6] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, (New York, NY, USA), pp. 30:1–30:15, ACM, 2018.

[7] Ethereum Foundation, "Ethereum project." `https://www.ethereum.org`. Retrieved April 25, 2019.

[8] Hyperledger, "Glossary." `https://hyperledger-fabric.readthedocs.io/en/release-1.4/glossary.html`. Retrieved May 7, 2019.

[9] Ethereum Community, "White paper." `https://github.com/ethereum/wiki/wiki/White-Paper`. Retrieved May 20, 2019.

[10] Hyperledger, "Java SDK for hyperledger fabric 2.0 pre-release." `https://github.com/hyperledger/fabric-sdk-java`. Retrieved May 4, 2019.

[11] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman, "Medrec: Using blockchain for medical data access and permission management," in *2016 2nd International Conference on Open and Big Data (OBD)*, pp. 25–30, IEEE, 2016.

[12] X. Yue, H. Wang, D. Jin, M. Li, and W. Jiang, "Healthcare data gateways: Found healthcare intelligence on blockchain with novel privacy risk control," *Journal of medical systems*, vol. 40, no. 10, p. 218, 2016.

[13] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 583–598, IEEE, 2018.

[14] H. Oh, C. Rizo, M. Enkin, and A. Jadad, "What is ehealth (3): A systematic review of published definitions," *J Med Internet Res*, 2005.

[15] The Norwegian Directorate of eHealth (NDE), "The Norwegian directorate of e-health (NDE)." `https://ehelse.no/english`. Retrieved May 1, 2019.

[16] HealthIT.gov, "What is an electronic health record(EHR)?." `https://www.healthit.gov/faq/what-electronic-health-record-ehr`. Retrieved May 25, 2019.

[17] L. Beard, R. Schein, D. Morra, K. Wilson, and J. Keelan, "The challenges in making electronic health records accessible to patients," *Journal of the American Medical Informatics Association*, vol. 19, no. 1, pp. 116–120, 2012.

[18] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer, 2nd ed., 2010.

[19] Yaga, Dylan and Mell, Peter and Roby, Nik and Scarfone, Karen, "Blockchain technology overview." https://doi.org/10.6028/NIST.IR.8202. Retrieved May 30, 2019.

[20] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI '99 Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pp. 173–186, USENIX Association Berkeley, 1999.

[21] Nxt community, "Nxt whitepaper." https://www.dropbox.com/s/cbuwrorf672c0yy/NxtWhitepaper_v122_rev4.pdf. Retrieved March 25, 2019.

[22] Hyperledger, "Hyperledger fabric SDKs." https://hyperledger-fabric.readthedocs.io/en/release-1.4/fabric-sdks.html. Retrieved May 4, 2019.

[23] The Linux Foundation, "Hyperledger." https://www.hyperledger.org. Retrieved March 25, 2019.

[24] Docker Inc., "What is a container? a standardized unit of software." https://www.docker.com/resources/what-container. Retrieved May 1, 2019.

[25] K. Olson, M. Bowman, J. Mitchell, S. Amundson, D. Middleton, and C. Montgomery, "Sawtooth: An introduction." https://www.hyperledger.org/wp-content/uploads/2018/01/Hyperledger_Sawtooth_WhitePaper.pdf. Retrieved May 25, 2019.

[26] The Linux Foundation, "About hyperledger." https://www.hyperledger.org/about. Retrieved March 1, 2019.

[27] Hyperledger, "Chaincode for operators." https://hyperledger-fabric.readthedocs.io/en/release-1.4/chaincode4noah.html. Retrieved May 15, 2019.

[28] Hyperledger, "Gossip data dissemination protocol." https://hyperledger-fabric.readthedocs.io/en/release-1.4/gossip.html. Retrieved May 7, 2019.

[29] Hyperledger, "The ordering service." https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer/ordering_service.html. Retrieved May 25, 2019.

[30] Apache Software Foundation, "Introduction." https://kafka.apache.org/intro.html. Retrieved April 7, 2019.

[31] Apache Software Foundation, "Apache zookeeper." https://zookeeper.apache.org. Retrieved April 7, 2019.

[32] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference*, pp. 305–319, USENIX Association, 2014.

[33] Hyperledger, "Ledger." https://hyperledger-fabric.readthedocs.io/en/release-1.4/ledger.html. Retrieved May 25, 2019.

[34] I. MongoDB, "NoSQL databases explained." https://www.mongodb.com/nosql-explained. Retrieved May 6, 2019.

[35] Ecma International, "The JSON data interchange syntax." https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf. Retrieved May 30, 2019.

[36] Hyperledger, "v1.4.1 release notes - april 11, 2019." https://github.com/hyperledger/fabric/releases/tag/v1.4.1. Retrieved April 25, 2019.

[37] Hyperledger, "A blockchain platform for the enterprise." https://hyperledger-fabric.readthedocs.io/en/release-1.4/. Retrieved May 1, 2019.

[38] Hyperledger, "Activity." https://jira.hyperledger.org/projects/FAB/summary. Retrieved May 1, 2019.

[39] Hyperledger, "Prerequisites." https://hyperledger-fabric.readthedocs.io/en/release-1.4/prereqs.html. Retrieved May 1, 2019.

[40] Hyperledger, "cryptogen." https://hyperledger-fabric.readthedocs.io/en/release-1.4/commands/cryptogen.html. Retrieved May 6, 2019.

[41] Ben-Kiki, Oren and Evans, Chris and döt Net, Ingy, "Yaml ain't markup language (YAML™) version 1.2." `https://yaml.org/spec/1.2/spec.pdf`. Retrieved March 1, 2019.

[42] Hyperledger, "configtxgen." `https://hyperledger-fabric.readthedocs.io/en/release-1.4/commands/configtxgen.html`. Retrieved May 6, 2019.

[43] Docker Inc., "Overview of docker compose." `https://docs.docker.com/compose/overview/`. Retrieved May 1, 2019.

[44] Hyperledger, "Fabric CA user's guide." `https://hyperledger-fabric-ca.readthedocs.io/en/release-1.4/users-guide.html`. Retrieved May 10, 2019.

[45] Hyperledger, "Service discovery." `https://hyperledger-fabric.readthedocs.io/en/release-1.4/discovery-overview.html`. Retrieved May 7, 2019.

[46] Apache Software Foundation, "Apache license, version 2.0." `https://www.apache.org/licenses/LICENSE-2.0.html`. Retrieved May 7, 2019.

[47] Rescorla, E., "The transport layer security (TLS) protocol version 1.3." `https://tools.ietf.org/pdf/rfc8446.pdf`. Retrieved May 30, 2019.

[48] Gson Community, "Gson." `https://github.com/google/gson`. Retrieved May 1, 2019.

[49] Apache Software Foundation, "Apache maven 3.x." `https://maven.apache.org/ref/3.6.1/`. Retrieved May 1, 2019.

[50] Gluon, "Javafx." `https://gluonhq.com/products/javafx/`. Retrieved May 15, 2019.

[51] Oracle, "Lesson: Introduction to JAXB." `https://docs.oracle.com/javase/tutorial/jaxb/intro/index.html`. Retrieved May 1, 2019.

[52] Oracle, "Service (javafx 11)." `https://openjfx.io/javadoc/11/javafx.graphics/javafx/concurrent/Service.html`. Retrieved May 15, 2019.

[53] C. Ferris, “Does hyperledger fabric perform at scale?.” `https://www.ibm.com/blogs/blockchain/2019/04/does-hyperledger-fabric-perform-at-scale/`. Retrieved April 15, 2019.

Appendix A

Source Code and Development

A.1 Development Setup

The system used for development and testing of the HLF Network package and the Java Application package is configured with the following software setup:

- Ubuntu 18.04.1
- OpenJDK 11.0.2
- Docker 18.09.4
- Docker Compose 1.17.1
- Go 1.11.4
- Hyperledger Fabric v1.4.1

Java SE JDK 11 or higher is required for running the Java Application package.

A.2 Source Code

The source code for the software packages developed in this thesis are located in private Git repositories. Contact the author for access to the repositories.

Appendix B

Setup Guides

B.1 HLF Network Package

To set up the Hyperledger Fabric blockchain network provided by the HLF Network package, follow these steps in the order they are listed below:

1. Place the HLF Network package in the Fabric binaries root directory
2. Edit the configuration files if necessary:
   - compose-with-couchdb.yaml
   - compose-with-raft.yaml
   - configtx.yaml
   - crypto-config.yaml
   - base/compose-base.yaml
   - base/peer-base.yaml
3. Remove any existing cryptographic material or artifacts
4. Run generate.sh to create new cryptographic material and channel artifacts

5. Run start.sh [seconds] to start the docker containers and configure the peers and channel
6. Check that all containers have started successfully and are running

To tear down the network, run clean.sh. Note that clean.sh requires root access and will remove all docker volumes and containers in your system. If you do not want all volumes and containers to be removed, remove the volumes and containers associated with the HLF Network manually and delete the following files and directories created in the root directory before trying to set up a new network:

- certs.jks
- channel-artifacts
- crypto-config

B.2 Java Application Package

The HLF Network must be started before the application can be configured. After the network has been started, follow the steps below to start the Java Application:

1. Copy the trust store named certs.jks, which is created by generate.sh in the HLF Network package, to the resource directory of the Java Application package
2. Configure affiliations.txt and list the affiliations that should be available to the user when creating a user to enroll with the CA
3. If the system is not using DNS, add hostname address mappings in the system hosts file (located at /etc/hosts in Linux)
4. Delete the directory named user if it already exists in the root directory
5. Run the application

For testing purposes, it is advised to run the application from an integrated development environment (IDE), as opposed to packaging it as a stand-alone application.