Faculty of Science and Technology

BACHELOR'S THESIS

Study program/Specialization:	Spring semester, 2021
Bachelor in Computer Science	Open / Restricted access
Writers:	
Fredrik Woie, Magnus Glenna	
Subject leader:	
Antorweep Chakravorty	
Faculty supervisors:	
Nikita Rajendra Karandikar, Antorweep Chakravorty	
Thesis title:	
Blockchain based hospital data management using Hyperledger Fabric	
Credits (ECTS): 20	
Key words:	Pages: 37
Blockchain Hospital journals Data management	+ attachments: 45 Stavanger 15. mai 2021

Universitetet
i Stavanger

Blockchain based hospital data management using Hyperledger Fabric

Fredrik Woie
Magnus Glenna

Faculty of Science and Technology
University of Stavanger

May, 2021

Abstract

Technological progress has pushed blockchain technology to cover a multitude of use cases and environments with different types of technology and frameworks. *HyperLedger Fabric* [10] is a framework for creating a permissioned blockchain. Blockchain [1] is defined as a decentralized, distributed and immutable ledger. Permissioned indicates that all nodes and users need permission to take part in the network, meaning it is within a controlled and centralized environment.

We propose *HyperLedger Fabric* for use in a hospital as a data storage. Security and reliability are vital parts of a hospital data storage, but this often complicates the sharing of data between network participants. Using *Fabric*, we want to achieve secure, shareable and authorized data storage backed by blockchain technology. Permissions should be set so only authorized peers can read secure data; Only the doctor of a patient, the patient if desired and anyone given access by the patient should be able to read a patient's data.

Acknowledgements

We want to thank our supervisors, Nikita Rajendra Karandikar and Antorweep Chakravorty, for guidance and expertise throughout the process of this thesis.

Contents

1 Introduction **1**
1.1 Background and motivation 1
1.2 Outline 2

2 Background **3**
2.1 The birth of Blockchain 3
2.1.1 Double Spending 4
2.2 New ways of using blockchains 5
2.3 The elements of blockchain 5
2.3.1 Decentralization 6
2.3.2 Distribution 6
2.3.3 Encryption 7
2.3.4 Immutability 7
2.3.5 Tokenization 7
2.4 Development of blockchain 7
2.4.1 Public and Private Blockchains 8
2.5 Use cases and downsides with blockchain 8
2.6 Standardization 9

3 Hyperledger Fabric **11**
3.1 Background 11
3.2 Networks and Channels 11
3.3 Identity 12
3.4 Membership Service Providers 13
3.5 Access Control Lists 13
3.6 Smart Contracts 15
3.7 Peers and Orderers 15
3.8 Raft 17

3.9 Update Transaction 18
3.10 Query Transaction 19
3.11 Private Data Collection 20
3.12 Docker 20
3.12.1 Docker Compose 21

4 Project: Using Fabric for hospital data collection 22
4.1 Requirements 22
4.2 Proposed Architecture 23
4.2.1 Network entities 23
4.2.2 Confidentiality and authentication 23
4.2.3 Network structure 25
4.2.4 Certificate Authorities 27
4.2.5 Chaincode 28
4.2.6 JavaScript API 32
4.3 Running the application 34
4.3.1 Prerequisites and binaries 34
4.3.2 Starting the network 35

5 Conclusion 36
5.1 Summary 36
5.2 Further work 37

Acronyms

ACL **A**ccess **C**ontrol **L**ist. 4, 14, 24, 30, 36
API **A**pplication **P**rogramming **I**nterface. 4, 32

CA **C**ertificate **A**uthority. 4, 12, 13, 25, 27, 32, 35
CFT **C**rash **F**ault **T**olerant. 4, 17
CLI **C**ommand **L**ine **I**nterface. 4, 34
CRL **C**ertificate **R**evocation **L**ist. 4, 13

DLT **D**istributed **L**edger **T**echnology. 4, 5, 8–10

FSM **F**inite-State **M**achine. 4, 17

ID **I**dentification. 4, 13, 14

MSP **M**embership **S**ervice **P**rovider. 4, 13, 16, 23, 24

PKI **P**ublic **K**ey **I**nfrastructure. 4, 12, 13
PoS **P**roof **o**f **S**take. 1, 4, 7
PoW **P**roof **o**f **W**ork. 1, 4, 7

SHA **S**ecure **H**ash **A**lgorithm. 4, 7

TLS **T**ransport **L**ayer **S**ecurity. 4, 26, 27

Chapter 1

Introduction

1.1 Background and motivation

Since the first blockchain, Bitcoin, were introduced, the technology has improved both in popularity and in use cases. It has revolutionized the way digital assets are stored. While the Bitcoin blockchain only let one transfer currency, new blockchains has been made to make it possible to transfer all types of digital assets. A transaction within a blockchain does not require a third party outside the network, making the distribution of assets decentralized. By not needing a third party to verify the transaction, the cost efficiency is increased compared with banks. Instead of a third party, blockchains use consensus algorithms to prevent malicious activity.

There are two types of blockchains, permissioned and permissionless. In a blockchain which is permissionless everyone has access to every transaction keeping the transactions tamper proof. As this type of blockchain does not have a requirement to become a member, it often uses more energy consuming consensus algorithms to verify transactions.

To prevent bad actors from tamper with transactions in a public blockchain, consensus algorithms like *Proof of Work* [7] and *Proof of Stake* [20] has been implemented. The permissioned blockchain requires participants to be authorized to access the network. This can allow for less energy consuming consensus algorithms as the network actors could be more trusted. It offers the participants to share assets within the permissioned network compared with permissionless where everyone can see the transaction. A framework called *HyperLedger Fabric* [10] is often used to create blockchains which are

based on permissions. It is often used as a ledger for business transactions between cross-industry collaborations.

As of the start of the project our knowledge of blockchain technology were limited, with some interest in cryptocurrencies like Bitcoin, Ethereum etc. We chose the subject of the thesis to get a better understanding of blockchains and its use cases. Based on experience with the lack of journals being shared between hospitals and doctors, we wanted to create a blockchain which made it possible for doctors be up to date with each patient's journal. Since sensitive data had to be stored in the blockchain, we needed a way to create a permission based blockchain. After a thorough research we found that the framework, *HyperLedger Fabric*, met our requirements. We used it in our project to define the permissions of each node, patient and doctor, in the network. Only the doctor of a patient and the patient itself should be able to read the patient's data. If anyone else would want to read the data, they must gain access from the patient. By storing the hospital data in a blockchain, it would be more secure and reliable compared to traditional databases.

1.2 Outline

The remaining part of the thesis are outlined as follows:

Chapter 2 introduces the reader to the blockchain technology and concepts used in the thesis.

Chapter 3 lets the reader get a better knowledge of the blockchain framework, *HyperLedger Fabric*, before it is later introduced in the project.

Chapter 4 covers the project part of the thesis. It explains the requirements, entities, and architecture of the network.

Chapter 5 concludes the thesis.

Chapter 2

Background

2.1 The birth of Blockchain

In 1991, Stuart Haber and W. Scott Stornetta described a technology that used cryptographically secured chain of blocks to make the timestamp of a document tamper-proof [1]. Three years later a data scientist called Nick Szabo proposed a digital contract called smart contract [2]. It was designed to determine when to execute a transaction of digital assets, based on terms of the contract. In 1998, Szabo designed a cryptocurrency that he called bit gold. The currency did not get past the design phase since Szabo was not able to prevent double spending [3].

Satoshi Nakamoto realized that the technology described by Haber and Stornetta could be used for more than preventing tampering of documents. With inspiration from the work of Stuart Haber, W. Scott Stornetta and the cryptocurrency designed by Szabo, Nakamoto created a digital currency called Bitcoin. The currency made it possible to perform a money transaction from one person to another, without the need of a third party [4], p.57.

In 2008, Satoshi Nakamoto published the Bitcoin white paper. The paper defined an electronic coin as a chain of digital signatures. The coin was meant to be transferred by digitally signing a hash of the previous owner and the public key of the coin receiver. The hash and public key are then added to a block of transactions, embedded with a hash of the previous block, and added to the chain of blocks. This hash made the ledger immutable [5], p.2. This type of transaction provides a strong control over the ownership of the coins, but do not prevent double spending[6] from happening[5], p.8.

2.1.1 Double Spending

Double spending occurs when a digital currency gets spent twice. A digital currency can get used twice if a person with knowledge of the blockchain network and with the necessary resources to manipulate the blockchain, manage to duplicate and use a digital token [6]. An illustration of a double spending is show in figure 2.1. Since Bitcoin is a decentralized cryptocurrency, without any third-party to verify each transaction, Nakamoto needed a way to prevent double spending. For this he implemented the *Proof of Work* consensus algorithm. As new blocks are added to the chain, a 64 digit hexadecimal number(a *hash*) representing the previous block is added to the block. This was achieved by having users, so-called *miners*, brute force calculate the hash of a block which is less or equal to a target hash. The winner of this race to a valid hash added the new block to the chain and was rewarded with Bitcoin. With this it would be impossible to change any previous transaction as you would have to recalculate the hash before a new block is added.[7].

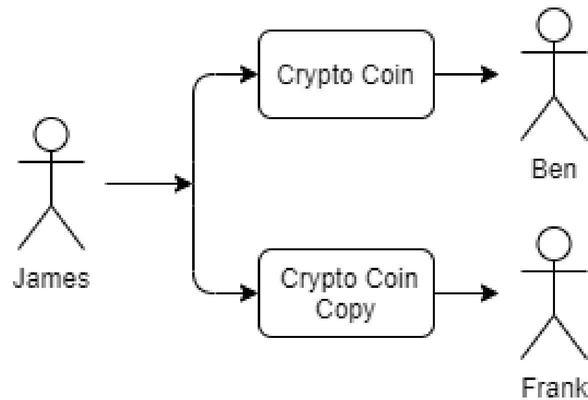

Figure 2.1: Illustration of double spending

2.2 New ways of using blockchains

It took a few years for developers to use blockchain as more than a ledger for cryptocurrency transactions. In 2013 Vitalik Buterin created Ethereum, a cryptocurrency and tool to create more blockchain applications. Buterin adopted the technology, smart contracts, invented by Szabo. It was designed to exchange digital assets directly without the need of a middleman. Smart contracts became on-chain applications which users could build, deploy and use to execute transactions[8].

Companies like Microsoft and IBM uses the Ethereum technology in their blockchain applications; collaborating in a project called *HyperLedger*, created by the Linux Foundation in 2015. The project goal is to implement blockchains as ledgers for the business transactions in intra-industry collaborations and to increase the performance and reliability of the ledger systems used [9].

A framework called *HyperLedger Fabric* is often used for distribution of digital assets in intra-industry collaborations. It is a blockchain technology based on permissions and roles between the nodes in the network. It can separate private business data from data intended to be distributed to other businesses using channels and private data collections [10]. For users to transact on the network, *Fabric* uses what they call a *chaincode* [11]. An in-depth description of *HyperLedger Fabric* will be given later in chapter 3.

2.3 The elements of blockchain

Blockchain is a type of database where data are stored in blocks and then chained together. Each new block stores a hash of the previous block creating a chain of immutable blocks [12]. Blockchains are mostly used as a ledger for transactions of digital assets and are therefore often referred to as *Distributed Ledger Technology* [13]. The five elements of blockchain are: decentralization, distribution, encryption, immutability, and tokenization [14]. Illustrated in figure 2.2. The use cases of the elements and how they should be implemented in a blockchain was described in Nakamoto's white paper [5].

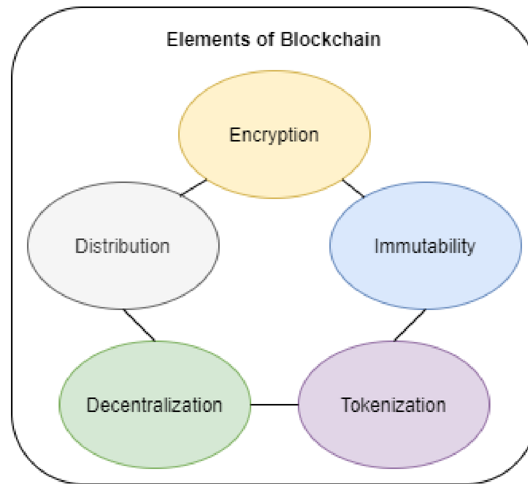

Figure 2.2: The five main elements of blockchain [14]

2.3.1 Decentralization

Decentralization means that no central body holds control over the network and is instead maintained by a consensus mechanism. If a node controls over 50 percent of the networks computing power they could forcibly approve transactions.

To reach a consensus on which transactions are valid or not, nodes use an algorithm to issue, validate and commit transactions on the network. This requires all transactions to be open on the network so that all nodes can participate in the validation process.

2.3.2 Distribution

The network is distributed over physically apart nodes. Each node has an equal role in the network, takes part of validation, and holds a full copy of the blocks and ledger stored.

2.3.3 Encryption

The Bitcoin blockchain uses *SHA-256* encryption [17] for both the *Proof of Work* algorithm and transaction verification [18]. *SHA* stands for *Secure Hash Algorithm*. A small change in the input data will give a completely different hash. Therefore an attempt to change previous blocks will require a recalculation of the hash or the change will be easily discovered if not blocked [17]. Transactions within a blockchain are immutable because of systems like *Proof of Work*, making blockchain exceptional for storing sensitive data.

2.3.4 Immutability

Immutability refers to transactions being cryptographically signed, timestamped, and added sequentially to the ledger making it unchangeable once committed unless there is an agreement to do so.

2.3.5 Tokenization

The Bitcoin blockchain uses tokenization, meaning that digital assets get a value based on real currencies like dollar. At the start of Bitcoin, one coin was worth the same as one dollar. The goal was to create a value to Bitcoin as a currency so that the miners could get paid for managing the network.

2.4 Development of blockchain

Almost all existing blockchains today are based on the five basic pillars, suggested by Nakamoto. Newer blockchain technology take inspiration from Bitcoin, but have new methods to implement and manage blockchains. For instance, the new Ethereum 2.0 uses the *Proof of Stake* algorithm, in an attempt to replace *PoW* as the consensus algorithm used to verify and create new blocks.

It is designed to randomly pick nodes as validators, with a bias based on how many tokens the node possesses. This reduces the overall computing power needed as less nodes take part of the validation. However this does have an impact on security of the blockchain

2.4.1 Public and Private Blockchains

There are two types of blockchains, public and private. Also known as permissioned and permissionless blockchains. Bitcoin is a public blockchain meaning that every transaction within the network is visible to all participants. Even though Nakamoto argued for the importance of having the blockchain public to prevent any tampering of transactions, new private blockchains have been developed.

Private blockchains are permissioned, meaning that network participants must be registered and authorized. These types of blockchains are often used for sensitive data making it important to establish who can participate and what actions they can utilise. Since nodes on the network can often be more trusted, private ones can use less power consuming consensus mechanisms to validate transactions. To join a private blockchain you must either be invited by the founder of the network or get validated by a set of rules determined in a smart contract [22].

Since public blockchains allow anyone to join the network, the amount of computer power needed to maintain the large-scale network can be enormous. For this reason, businesses who are looking to set up a blockchain at a minimal cost, should consider using a private network.[23].

2.5 Use cases and downsides with blockchain

Blockchain technology can provide a secure and reliable way of storing digital assets, which is a driving force for many companies to implement the technology. IBM has created a Food Trust Blockchain to make it possible for suppliers to trace products back to their origin. It can be used to verify a product location of origin, which could for example be used to track contaminated food to the source of an outbreak. *DLT*s can also be used to authenticate votes, which could have eliminated any suspicion of fraud during the 2020 USA presidential election because of the immutability of a blockchain [12].

By using a blockchain to store hospital data, the journals will be safe from hackers. There are cases where countries have hacked into hospitals to get knowledge of a patient's medical history. This has been done to convert a person, usually with access to sensitive military data, to a spy. The hackers will then pay the medical bill in exchange for sensitive documents provided

by their new spy [24].

Even though blockchain is useful in many cases, the amount of energy required to control and verify each transaction can be enormous. Each year Bitcoin consumes an estimated 3.6 billion dollars in energy[25]. However, it is impossible to know the true cost because a decentralized network does not have the same energy costs per node.

A blockchain transaction can also sometimes be slow and expensive. Bitcoin can only handle 7 transactions per second [12]. Any surplus will be put in a confirmation queue, known as *Mempool* [26]. This can cause a lot of waiting time compared to a bank which will simply scale up their capacity. Blockchain transaction fees can vary based on the size of the transaction and current demand for transactions. This transaction fee variation can make cryptocurrency a less attractive transaction platform, which causes some to use banks instead. Visa can process over 24 000 transactions per second. To compete with the rates of *VISA* faster blockchain technology is being developed, some of which can process up to 30 000 transactions per second [12].

2.6 Standardization

Since the first launch of blockchain, the amount of companies employing the technology has increased rapidly. *DLT*s are being used to store and manage business transactions, deals, documents, transportation of products etc. The main goal of *Hyperledger* project is to cover all types of community sectors. The project is founded by the biggest corporations within internet-related services and products. Forbes claims that 50 percent of the largest corporations which deploy blockchain are using *Hyperledger*. As a result, the *Hyperledger* project likely to have a huge impact in how other companies implement blockchain technology [27].

As of 2021, there is a lack of consensus on blockchain related standards and definitions. For instance, the requirements for handling sensitive information of a person has yet to be determined. The European Union, EU, wants to create regulations regarding sensitive information stored in a blockchain. The regulations are meant to decide how a subject's data should be used and who has access to it [28]. The Institute of Electrical and Electronics Engineers, also known as IEEE, have made a committee for development of standards for blockchain. The name of the committee is IEEE Computer Society Blockchain and Distributed Ledger Standards Committee. The goal is

to create standards regarding how blockchains are made, which organization is allowed to create them, what their purpose is and what type of security should be required in a blockchain [29]. The International Organization for Standardization, ISO, is trying to bring organizations such as IEEE, EU and members of *HyperLedger* together to agree on which standards to use [28].

Standardization could help companies to get a better understanding of the blockchain technology. It would let firms know about the security measures in place to store data, and which measures are put in place to maintain the privacy of each member in a secure fashion.

Standardization is needed to guide new innovation in blockchain technology to implement blockchains which follow standards for safe data handling, storing and managing. This could also encourage new entities to implement *DLT* to store their data and/or transfer digital assets like documents in intra-industry cases. A robust standardization can help prevent poor implementation decisions and help firms improve their data management.[30].

Chapter 3

Hyperledger Fabric

3.1 Background

Hyperledger is an open-source project, created in 2015 by the Linux Foundation. Their goal is to advance blockchain technologies designed for enterprises. The project is comprised of over 250 members and has several active projects [31]. One of their projects is IBMs *Hyperledger Fabric*, an open-source permissioned distributed ledger technology platform, designed for enterprises. It uses channels to provide a secure and private way for transferring digital assets. The blockchain is based on permissions, making it possible for companies to control access across the network. *Fabric* uses smart contracts, composed of one or multiple *chaincodes*, to make transactions on the network. The *Fabric* API currently supports *chaincodes* written in Go, Node.js or Java [32].

3.2 Networks and Channels

Fabric uses a combination of channels and networks to share digital assets, either between members of the same company or between multiple firms [10]. A network provides a way for applications to be able to use ledgers and smart contracts. The ledger can be used by businesses to store digital assets, while the smart contracts can be used to transfer the assets between members of the network. When a network is created, the organizations involved sets policies to define the permissions members should have. When documents

are shared in the network, all members can see it. If the documents are meant to be shared privately, because of their confidentiality, separate channels can be used. For instance, a node can use a channel to transfer a document to a specific person within the same corporation, without other nodes in the network being able to see or change it [33].

Channels are created by a consortium of organizations. It lets the organizations share private data, providing privacy from the network by separating the network ledger and the channel ledger. Each channel has one ledger per *chaincode* on the channel. It is comprised of a world state and transaction log. The world state is the current state of the channel ledger, while the transaction log is the log of all the transactions that has led to the current world state. In *Fabric* the transaction log is the blockchain. The members of a channel can transfer assets if they follow the policies of the channel [33].

3.3 Identity

To provide security to a type of *PKI* [34] has been used in *Fabric*. It stands for *Public Key Infrastructure* and makes sure the communication between the nodes in a network are secured and authenticated. *PKI* in *Fabric* has four main elements. It uses digital certificates to prove the identity of a user, by having a document describing the holder of the certificate using attributes.

This document is a *X.509* certificate containing the private and public keys of the user, as well as information about the user, that are used to create a unique identity on the network. The identity created by a *Certificate Authority (CA)*[34] is used by *Membership Service Provider*s to verify their membership. It cannot be tampered with as a small change will give a completely different identity and the *CA* will not authorize the user.

The *Certificate Authority* have public certificates so users can verify its authority. Since *CA*s provide the verification of the digital identity of members in a network, it is important that users check the public certificates to make sure the *CA*s are secure [34].

To prevent the exposure of a root *CA* in each network, *Fabric* uses intermediate *CA*s who work together in creating an establishment called chain of trust. This helps in a multitude of areas; First it can distribute and decentralize the authentication process, second it will reduce the risk of a *CA*'s private

key being exposed, and thirdly we can distribute separated certificates for any number of organizations on the network.

A private root *CA* known as *Fabric CA* has been developed to allow developers to create and customize *CA*s for use in their blockchain network. It provides a way to manage digital identities and the x.509 certificates of the users.

An important final element is *Fabric*s *Certificate Revocation List* [34]. It is the first list the *MSP* will check as it contains all certificates which have been revoked. Whenever a certificate is renewed, the old identity is added to the *CRL* [34].

3.4 Membership Service Providers

To become a member of a *Fabric* network, one must be authorized. This is done by using a *Membership Service Provider* [35] that manages the *ID*s and authentication of participants. The *MSP* makes sure only selected nodes have access to the assets being shared in the network.

The *MSP* uses the identities created by *CAs* to determine if a user has a valid membership and what permissions they have. This function forces all users to a part of at least of organization transact on the network, but also safeguards the private keys of users. Each *MSP* have their own *CAs* for issuing certificates that are used to register a user.

Figure 3.1 shows the interaction between local and channel *MSP*s on a blockchain network. The nodes, Peer and Orderer, has a local *Membership Service Provider* which decides their role and permissions. A channel has a global *MSP* which is shared between all members of a channel. The permissions and roles of each member are decided by the channel *Membership Service Provider*. In figure 3.1, the peer node is controlled by ORG2 while the orderer node is administrated by ORG1. The Root *CA* provides authenticated identities. The ORG1 trusts the identities provided by the root CA1, while the ORG2 trusts the identities from root CA2 [35].

3.5 Access Control Lists

To implement even more permission layers, *Fabric* supports *Access Control List*s [36]. It is often used to specify which nodes have access to which network operations. For instance, an *ID* can have access to execute a *chaincode*,

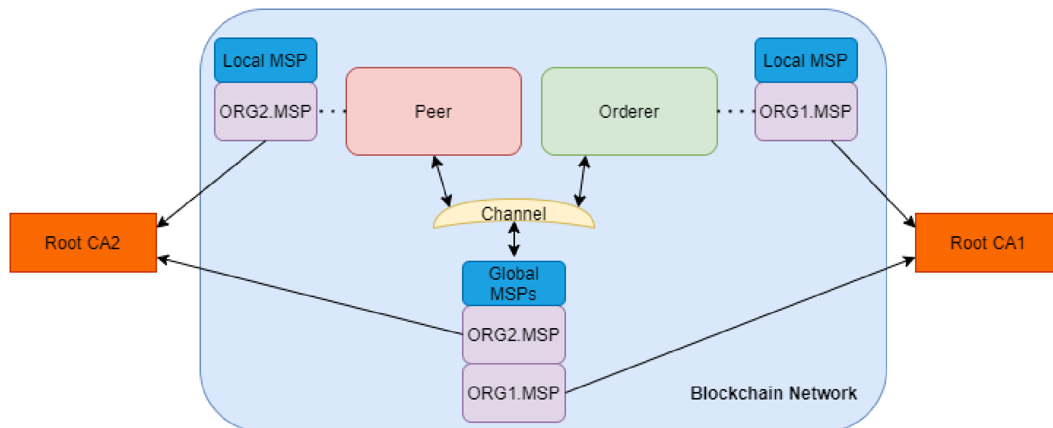

Figure 3.1: Local and global MSPs [35]

but not be able to create a new one. The lists provide a way to assign roles to node types like peer and orderer nodes. The ACLs combines resources with policies to control who has access to them. These policies defines whether a user is permitted to do a requested operation. ACLs uses two types of policies, signature and implicitMeta. The signature policy decides how many or which person who must sign before the policy is satisfied. The implicitMeta policy checks if other policies on a lower level has been satisfied. For instance, check if an orderer node has sent the request. Since the orderer node is defined in a policy at a lower level, the implicit meta policy must check if the orderer node policy is satisfied before itself can be satisfied. ACLs are created using key-value pair. The key is the function name, while the value is which of the two types of policies it is, followed by a set of rules regarding actions that must be done for the policy to be satisfied [36].

3.6 Smart Contracts

Smart contracts in *Fabric* has its origin from Ethereum. It is used to transfer digital assets within public networks and private channels by executing transactions following a base set of conditions. The conditions are logical statements that determine whether a transaction should be carried out. For instance, if both parts have signed an agreement, the documents are transferred. If none or only one part has signed, then the documents shall not be transferred [11].

Even though the terms smart contracts and *chaincodes* are interchangeable, they have different usages in *Fabric*. Smart contracts provide the transactional logic for interacting with the world state, a database which holds the current state of every assets. A smart contract can perform the operations; get, put, and delete. When information about the current state of an object is needed, the get statement is used. The put operation is used to create or/and modify objects on the world state. The delete statement is used to remove objects. The objects are only deleted from the current state of the ledger [11].

A *chaincode* decides how smart contracts are packaged before it is deployed to the network. For this reason, *chaincode*s are often used to group smart contracts that are related. Multiple contracts can be defined within a *chaincode*. When it is deployed to a network, every contract defined by the *chaincode* is made available to the participating organizations. For this reason, only administrators have to worry about *chaincodes* [11].

Smart contracts can be used for transferring other types of assets than documents. Suppose a car dealer is doing a trade. The car keys can be locked inside a safe, which do not open before the buyer has paid the price and both parts have signed an agreement. A smart contracts can be created in this instance, to unlock the keys when certain conditions have been met [37].

3.7 Peers and Orderers

In *Fabric*, firms can define their own assets and values using key-value pairs. A *chaincode* sets the rules for who can read or/and alter the assets and values [38]. There are two types of nodes: peers and ordering. They work together to make sure only legitimate transactions are committed to the ledger. A *Fabric* network consists mostly of peers whose main purpose is

to host smart contracts and ledgers. Peers can have copies of ledgers and smart contracts, making them able to execute and verify transactions. Each peer can host multiple ledgers and chain codes, making it easier to develop a flexible system. If an administrator wants to change a *chaincode* they must interact with a peer, for that reason peers is considered a key role in the *Fabric* network. Anchor peer nodes are used by applications to execute commands. It makes it possible for the applications to either update or query a ledger. A peer can only be part of one organization. The permissions and ownership of a peer are determined by the *Membership Service Provider* [39].

A peer can have different roles. For instance, some peers act as endorsing peers whose purpose is to sign the proposal before it is sent to the orderer. The main role of orderer nodes is to order the transactions and keeping track of the history of events on the network. When multiple nodes collaborate, they form an ordering service. The service is mainly used to receive transactions and create blocks on the blockchain during the process of committing a transaction to the ledger. The process of a transaction is illustrated in figure 3.2 [40].

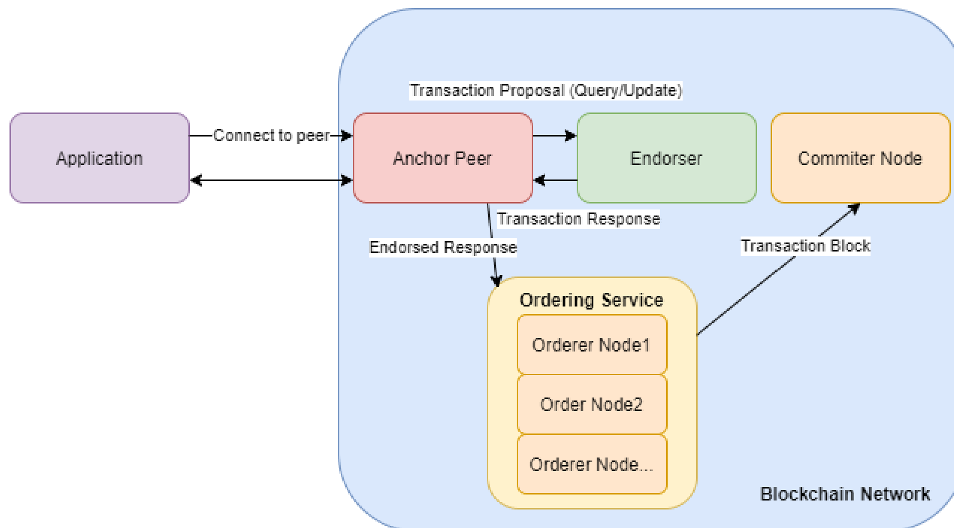

Figure 3.2: The basic transaction flow in *Fabric* [41]

3.8 Raft

When creating an ordering service, it is recommended to use the Raft ordering service implementation [40]. It is a *Crash Fault Tolerant* ordering service. This means that the system can achieve consensus even if some components fail [42]. The Raft ordering service uses a protocol known as the Raft protocol. It forces the service to use a system which practices the leader and follower model. The leader is an orderer node which has been elected by other nodes to become the leading orderer node of a channel. The main purpose of the leader is to receive new log entries, replicate and send it to its followers. When an entry is committed to a ledger, the leader is the one who manages it [40].

An orderer node do not have the leader role forever. The access to the abilities may vary depending on the circumstances. Since the service is *CFT* it can withstand the loss of either the leader or some of the followers. The leader will send heartbeat messages to the followers to let them know it is still "alive". If followers do not receive these messages within a configured time-period, the nodes will mark the leader as dead and elect a new one. The new node is chosen from a set of candidates. To become a candidate a follower must promote itself as the best node for the role. The candidate nodes will request votes from the followers, and if it gets enough votes it will become the new leader [40].

Quorum is defined as the minimum amount of orderer nodes, clustered together in a channel, which has to agree on a proposal for a transaction and/or a log entry to be added to the *Finite-State Machine* [40]. Each channel has one cluster of orderer nodes. The clustered nodes forms the ordering service. The cluster must contain at least three nodes to agree on a proposal. To accept a proposal, the number of nodes who consent must be larger than the nodes who do not. The *FSM* is used to make sure the logs are written in the same order. Since every node has a *FSM*, it is also used to make sure every node has the exact same ordered log [40].

3.9 Update Transaction

For applications to be able to get access to the ledgers and *chaincodes* they must connect to peer nodes. By connecting to the peers, the applications can either execute *chaincodes* or/and update the ledger. To update the ledger all peers in the network must agree to the change. The process is called *consensus* and are meant to prevent malicious nodes from tampering with the ledger without other peers know about it. When all peers have approved the transaction, causing an update to the ledger, the connected applications will get notified by the peers about the changes to the ledger. The process is separated in 5 parts and makes sure every peer has the same copy of the ledger [39]. A figure illustrating the five steps of the process is shown in Figure 3.3.

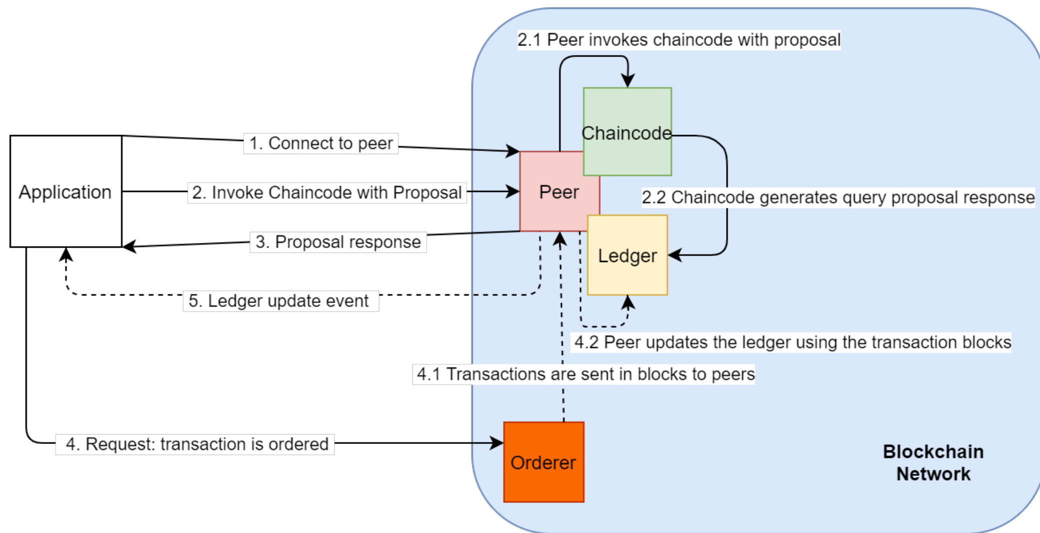

Figure 3.3: The five steps of an update transaction [39]

To update the ledger a client application must generate a transaction proposal. The proposal is then sent to endorsing peers of the organizations involved for the application to get permission to do the changes. The peers execute a *chaincode* based on the proposal to create a response to the proposed transaction. If they agree they will sign the response. The peers will at this time, not add the change to their ledger, in case the change does not get permission from the other peers. If the application gets enough signed proposal responses the next phase will begin [39].

In the second phase the ordering service node will check if enough endorsing peers has signed the proposal responses. If it is the correct amount, the orderer service will create blocks of transactions and distribute it to the peers who is members of the channel. This is done to make a final validation before the change is committed. Since the service receive so many transactions, they arrange them sequentially before they put it in containers known as blocks. It is from these blocks blockchain gets half its name. In the last phase each peer will inspect the blocks given by the orderer. This is done to make sure everyone received the same result. If the validation of the transaction update is successful, the peers will commit the block to the blockchain and update the ledger[40].

3.10 Query Transaction

A ledger query transaction requires a three-step interaction between an application and a peer. A figure illustrating the three steps is shown in Figure 3.4. For an application to execute a query it must connect to a peer. When a connection has been successful, the application will invoke a *chaincode* with the proposed query. Each peer has a copy of the ledger. As a result, a peer does not need to consult with other peers, to accept and execute the *chaincode*. This causes a query proposal response to the ledger. The peer then sends a query response to the application, which contains the query result. The query process is completed when the application has received the proposal response [39].

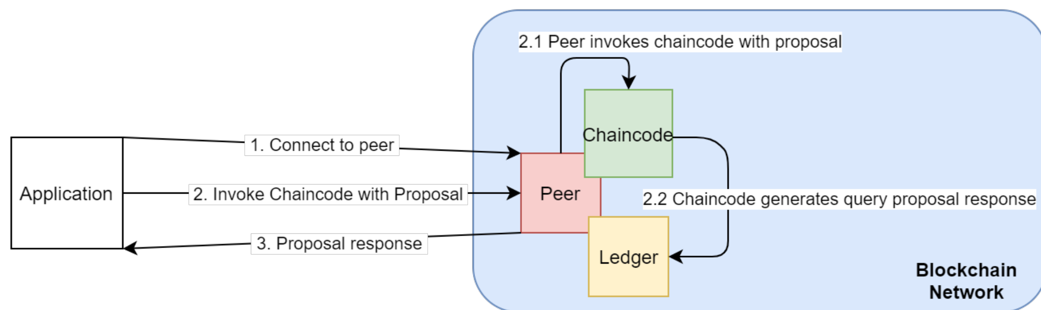

Figure 3.4: Three steps of a query transaction [39]

3.11 Private Data Collection

Private data collection [43] was introduced in *Hyperledger Fabric* version 1.2. It makes it possible to keep data private, without the need of separating channels. In earlier versions of *Fabric*, organizations had to create new channels if they wanted to keep the data private from other channel members. This caused administrative overhead, since it required the administrators to maintain a lot of unnecessary channels. Every authorized peer stores the private data in a separate database from the channel ledger. The private database is often referred to as *SideDB*. The data can get accessed only by selected organizations, through *chaincode* of an authorized peer. To still benefit from the immutability of the blockchain storage, a hash representation of the data is added to the world state. This gives extra security without exposing the data. Whenever private data is accessed it is checked against the stored hash to validate it not been changed[43].

3.12 Docker

Docker [44] is an open-source tool used to create, deploy, and run sandbox applications. *Docker* uses containers [45] to run different parts or entities of an application. This provides a way to make an application run on any operating system or system configurations. It shares similarities with a virtual machine, however the containers require less memory usage. This is because the containers can use the same operating system as the host, instead of having a separate one. *Docker* is comprised of a *Dockerfile* [46], *Docker* image [47], and *Docker container.*

The *Dockerfile* contains all requirements and specifications needed by a sandbox application. These are used to build a *Docker* image which contains a set of layers which each represent inscructions from the *Dockerfile.*

The *Dockerfile* contains all the requirements needed by the sandbox applications. The text document should contain a specification of which operating system to use, file location, language etc. The requirements defined in the *Dockerfile* is used to build a *Docker* image. The image contains a set of layers which each represent instructions from the *Dockerfile.*

A *Docker* container is created every time an image is running. When it is created, a container layer [48] is added to the top of the image. The image layers before and after the container is created is shown in figure 3.5. Every layer in the image is read-only except the container. All changes and

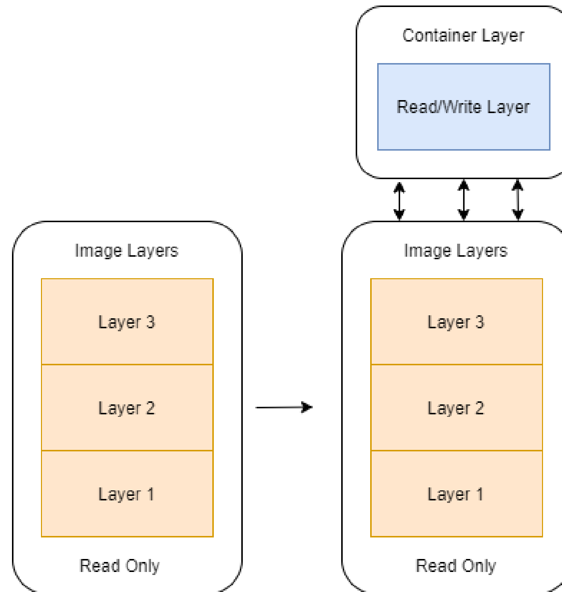

Figure 3.5: Illustration of the image layer structure, before and after a container has been generated [48]

modifications done in a container is stored in the container layer. Since a container do not influence an image, several containers can use the same image even if they have had different state changes.

3.12.1 Docker Compose

If an application requires many containers, it is recommended to use *Docker Compose* [49]. The Compose tool makes it easier to run complex applications which requires a lot of containers. It lets the user define all the containers in one file. Instead of running each container, one could just run a single file. This makes it easier to do the required processes to make an application run [44]. *Docker* and *Docker Compose* were used to generate the peer nodes used in our project network.

Chapter 4

Project: Using Fabric for hospital data collection

This chapter covers the project part of this paper. The requirements, entities and architecture of the network. Finally we'll discuss some different solutions to the same problem.

4.1 Requirements

Integrity

- It is vital that data in a hospital is not tampered or manipulated.

Availability

- The data must always be available to users and the network. It is important not only for the users using it, but also for the network to operate properly.

Confidentiality

- The information stored is extremely sensitive and personal and must therefore be fully confidential.

All these three points can be achieved using a decentralized permissioned blockchain.

4.2 Proposed Architecture

Using *HyperLedger Fabric* we construct the following network.

4.2.1 Network entities

Patient Platform - An application portal for patients to connect to the network. The patient is the person whose data is being stored. A patient would want the ability to read their journal and decide who can access their data. This is done through the Patient portal, a *JavaScript* application. Using a wallet to store the identity of patients, a patient can query the blockchain with limited access.

Doctor Platform - An application portal for doctors to connect to the network. A doctor is a user who wishes to be able to both read and write to journals they have access to. They also need the ability to request access to journals they are unauthorized for. A doctor should be able to have access to multiple journals.

Endorsers - Peers that validate and endorse transactions on the network. Running the same *chaincode*, they expect the same output as every other peer. They ensure that the transaction is executed as expected and no data has been manipulated without the endorsement of the network.

4.2.2 Confidentiality and authentication

We tried using private data collections for storage, but it proved difficult to find a way to transfer the data to another peer's unknown collection. This is because only the peer itself can access their own private storage. It is saved locally and only a hash of the private data is stored on the network. So even if the data is successfully requested and we know the names of both private collections, they are stored on separated peers. The only way to transfer something from one private collection to another is to either commit it to the world state for all to see, or transfer between two collections on the same peer.

A possibility is separating each user into their own organization on a single peer. This way we can share data by simply adding new users to the organization *MSP*, however this does not take advantage of *Fabric*'s transaction

endorsement policies and would be completely centralized.

Instead, we use a wallet [50] to store identities of users to be used on the network. The identities are *X.509* certificates stored in a local key-value pair storage. The certificate contains the public and private key used to validate the users' identity on the network. It also has a marker for which organization this user belongs, as well as an option to add other attributes or other fields for identification. *Fabric* includes a *GetID* function, seen in listing 4.4, which returns a *base64* value encoded from the users certificate. This is guaranteed to be unique within the *MSP* and is unchanged with certificate renewals. The wallet is stored in the file system of the peer. An option would be to use instead CouchDB for easier data recovery [50].

We add a basic read/write *ACL* using the client ID of the patient for each journal. This guarantees no duplicate key values for journals as all patients are part of the same organization and *MSP*.

This solution relies on the peers of the network to be secured as the payloads propagated across the channel are not encrypted. A possible solution to this would be to encrypt any data put on the blockchain and keep the decryption key off the network. Off-chain or confidential services are still quite unexplored in blockchain technology.

4.2.3 Network structure

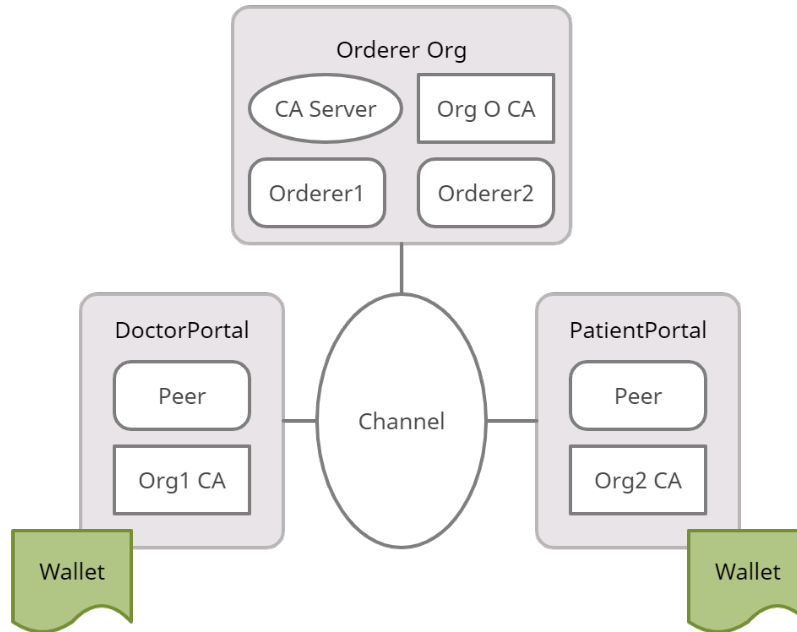

Figure 4.1: Network structure of peers, orderers, wallets and the channel.

Figure 4.1 shows the network structure of a simplified solution. The *DoctorPortal* and *PatientPortal* are separated into two organizations, each having one peer and a *Certificate Authority*. Using the identities stored in the Doctor- and PatientPortal wallets, users can connect, authenticate themselves and connect to the network. This simplified solution does not have any endorsing peers which would not be viable in a production network. But by using the *fabric-samples* we are only given the option of one peer per organization. The organization orderers are separated from their organizations as per *Fabric*'s security recommendations. This was an easy choice as the *fabric-samples* already have the orderer nodes separated.

Figure 4.2 and 4.3 show the flow of a doctor or user commiting a transaction to the network.

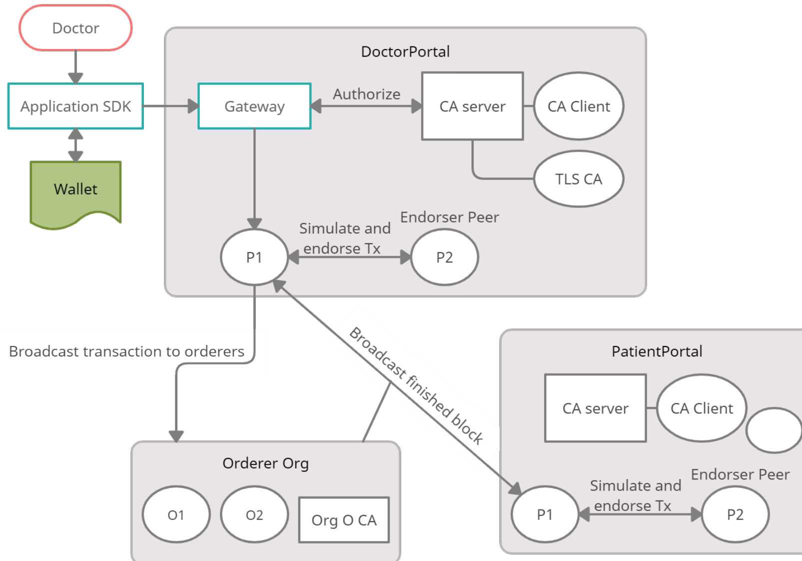

Figure 4.2: Network structure of the organisation peers and orderers.

This explains the steps of flow in figure 4.3 and 4.2:
(1): The application gets the user identity from the wallet and passes it to the gateway with the update transaction.
(2): The gateway authorizes the user with the CA server.
(3): If successful the gateway will issue the request to the anchor peer.
(4): The transaction is simulated and endorsed if valid by the endorser peer.
(5): The endorsed transaction is sent to the orderer node for ordering using TLS.
(6): The orderer CA server checks if the peer is authorized to submit a transaction.
(7): A finished block is broadcast using TLS to the network anchor peers.
(8): The endorser checks the endorsement of their transactions of the block.
(9): The block is committed to the blockchain and the world state is updated.

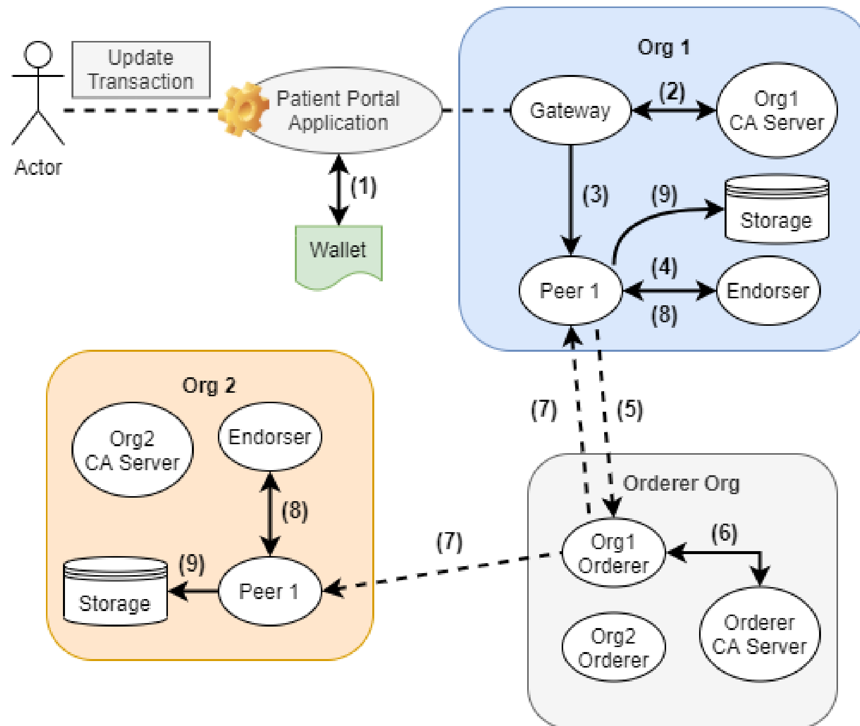

Figure 4.3: Flow of an update transaction.

4.2.4 Certificate Authorities

Our network includes three organisations; two for peers and endorsers, and the third for orderer nodes. The *Fabric* documentation recommends using two *CA*s per organization; One for generating organization and node identities, and the other for issuing *TLS* certificates. The *TLS CA* is generated automatically when the organization *CA* is created and is another security implementation by *Fabric* for securing communication between peers. This prevents the risk of *man in the middle* attacks.

In total we need six *CA* clients to achieve this.

4.2.5 Chaincode

The *chaincode* is written in *Go* and is quite simple. A demonstration of how these commands are used can be found in section 4.2.6.

There is one type of *struct* in the code, *Asset*. *Asset* is used as a journal which entries can be added to, a list of authorisations of a journal, and a list of access requests to a journal.

```
20 type Asset struct {
21   Owner   string              `json:"owner"`
22   Data    string              `json:"data"`
23   Entries map[string]string `json:"entries"`
24 }
```

Listing 4.1: The asset struct.

When creating the journal, the ID of the user invoking the call is given read access ("r") and added to the list of authorized users. On line 49 the *JSON* is added to the world state with *journalID* with the "auth" prefix as key value.

```
42   peers := Asset{
43     Entries: map[string]string{creator: "r"},
44   }
45   peerJSON, err := json.Marshal(peers)
46   if err != nil {
47     return err
48   }
49   err = ctx.GetStub().PutState("auth"+journalID, peerJSON)
```

Listing 4.2: Creating the list of authorized users.

The creator is added as owner of the journal and it is put into the world state with *journalID* as key value.

```
54   journal := Asset{
55     Owner:   creator,
56     Entries: make(map[string]string),
57   }
58   journal.Entries["default"] = "default"
59
60   journalJSON, err := json.Marshal(journal)
61   if err != nil {
62     return err
63   }
64   return ctx.GetStub().PutState(journalID, journalJSON)
```

Listing 4.3: Creating the journal object.

Mentioned earlier, using the *GetID()* function in the *GetClientIdentity()* interface is used for authorization. Found on line 96 in *Cid.go* [52].

```
1  func (c *ClientID) GetID() (string, error) {
2    // When IdeMix, c.cert is nil for x509 type
3    // Here will return "", as there is no x509 type cert for
       generate id value with logic below.
4    if c.cert == nil {
5      return "", fmt.Errorf("cannot determine identity")
6    }
7    // The leading "x509::" distinguishes this as an X509
       certificate, and
8    // the subject and issuer DNs uniquely identify the X509
       certificate.
9    // The resulting ID will remain the same if the certificate
       is renewed.
10   id := fmt.Sprintf("x509::%s::%s", getDN(&c.cert.Subject),
       getDN(&c.cert.Issuer))
11   return base64.StdEncoding.EncodeToString([]byte(id)), nil
12 }
```

Listing 4.4: *GetID* returns the unique client ID of the invoking user.

To give authorization to a journal we add the user ID to the map saved with the key value *"auth" + journalID*.

```
174 func AddAuthentication(ctx contractapi.
      TransactionContextInterface, journalID string, clientID
      string, access string) error {
175   peers, err := GetAuthenticatedPeers(ctx, journalID)
176   if err != nil {
177     return err
178   }
179
180   peers[clientID] = access
181
182   peersAsset := Asset{
183     Entries: peers,
184   }
185
186   peerJSON, err := json.Marshal(peersAsset)
187   if err != nil {
188     return err
189   }
190   return ctx.GetStub().PutState("auth"+journalID, peerJSON)
191 }
```

Listing 4.5: Adding a user ID to the authorized users list.

In *request.go* you find all the access request related functions: *RequestAccess()*, *GetAccessRequests()* and *AnswerAccessRequest()*. *RequestAccess()* checks that the user is not the owner, then adds the client ID as key value to the access request list before updating the world state.

```
10 func (s *SmartContract) RequestAccess(ctx contractapi.
     TransactionContextInterface, journalID string, access
     string) error {
11   creator, err := GetClientID(ctx)
12   if err != nil {
13     return fmt.Errorf("Failed to get owner ID, %v", err)
14   }
15   owner, _ := s.IsOwner(ctx, journalID)
16   if owner {
17     return fmt.Errorf("Cannot request additional access to a
     asset you own")
18   }
19   requests, err := AccessRequests(ctx, journalID)
20   if err != nil {
21     return fmt.Errorf("Failed to get Access requests %v", err)
22   }
23   requests.Entries[creator] = access
24   requestsJSON, err := json.Marshal(requests)
25   if err != nil {
26     return fmt.Errorf("Failed json Marshal, %v", err)
27   }
28   return ctx.GetStub().PutState("request"+journalID,
     requestsJSON)
29 }
```

Listing 4.6: Adding an access request to the access request list.

Shown in figure 4.4,when approving an access request, the code calls *AddAuthentication()*, shown in listing 4.5, adding the requested access to the journal's *ACL*. Then it deletes the request before updating the world state.

```
80       err = AddAuthentication(ctx, journalID, peerID, request)
81       if err != nil {
82         return err
83       }
84       delete(requests.Entries, peerID)
85       requestJSON, err := json.Marshal(requests)
86       if err != nil {
87         return err
88       }
89       return ctx.GetStub().PutState("request"+journalID,
     requestJSON)
```

Listing 4.7: Approving an access request.

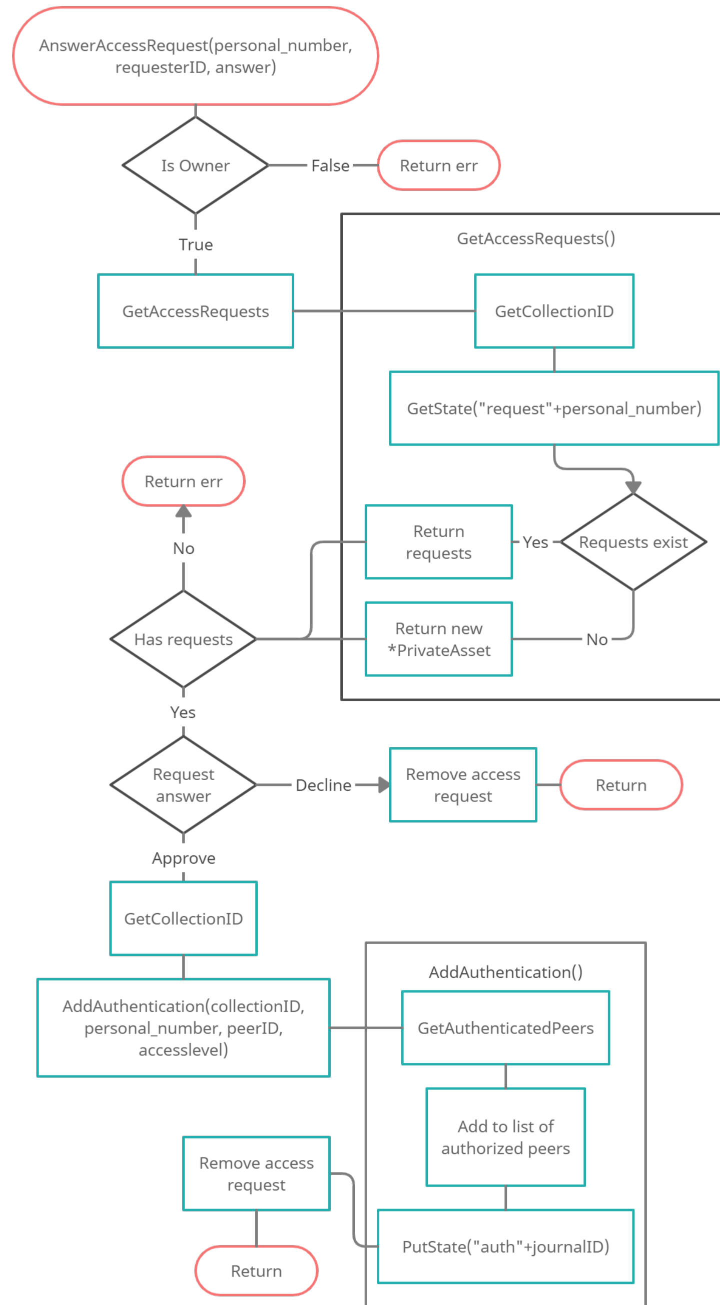

Figure 4.4: Flowchart showing the authentication of the answer access request function.

4.2.6 JavaScript API

The *JavaScript* API allows users to connect to the network using their identities stored in the application wallet. While there is limited restrictions on functions at the moment, splitting the API into two would allow for easy control over what functions a user can operate. Below are the current working commands.

JavaScript API Commands

enrollAdmin

Enrolls a CA administrator for an organisation and stores the identity in the peer's wallet.
There can only be one administrator per organization.
Usage: **node enrollAdmin.js org**

---

registerEnrollUser

Registers a new client with an organization CA and stores the created identity in the peer's wallet.
Usage: **node registerEnrollUser.js org userID**

---

createJournal

Used by the patient to create the initial journal object, add them as the owner and give them read rights for their journal.
Usage: **node createJournal.js org userID journalID**

---

getJournal

Returns the full journal object from the blockchain.
Requires read rights to the journal.
Usage: **node getJournal.js org userID journalID**

---

addEntry

Adds an entry to a journal with given *entryID* and given data string.
Requires write rights to the journal.
Usage: **node addEntry.js org userID journalID entryID data**

---

getEntry

Returns the entry of given *journalID* and *entryID*. Usage: **node getEntry.js org userID journalID entryID**

---

requestAccess

Requests access to a given *journalID*
Cannot be used by the owner to gain greater access.
Usage: **node requestAccess.js org userID journalID access**

---

getAccessRequests

Returns the list of users requesting access.
Owner only.
Usage: **node getAccessRequests.js org userID journalID**

---

answerRequest

Used by the owner to respond to access requests.
Owner only.
node answerRequest.js org userID journalID requesterID answer

---

4.3 Running the application

4.3.1 Prerequisites and binaries

To run *Fabric*, there are a few prerequisites listed below. We use *Linux Subsystem* in *Windows 10* to run an instance of *Ubuntu*. Docker and Docker-compose are used to generate and operate the containers running the network. *Golang* is used to implement the *chaincode* for the network. Finally we use *Node* to run the *JavaScripts* for generating identities and interacting with the network through their respective peers.

Prerequisite	Version
Git	Latest
Curl	Latest
Ubuntu OS	18.04
Fabric	2.3.2
Docker	20.10.5
Docker-compose	1.29
Golang	1.15.xx
Node	10.15.3

Additionally we need *Fabric-samples*, a repository made by the *Fabric* team to showcase the functions of their framework. We use the auction sample to take use of the node imports in the application directory for simpler queries.

To download the correct version of *fabric-samples*[51], the following command can be used:

```
curl -sSL https://bit.ly/2ysbOFE | bash -s – 2.3.2 1.5.0
```

This will clone the *fabric-samples* repository, download the latest *Hyper-Ledger Fabric Docker* images and finally download the CLI tool binaries that help generate and interact with the network.

Last step is to download the journal project files from Fredrik's *GitHub* repository, /fwoie/fabric-journal. The folder contains the smart contracts and JavaScript's used and can simply be merged with the *fabric-samples* folder replacing any duplicate files.

4.3.2 Starting the network

From the */fabric-samples/test-network/* folder, run the following command:

```
./network.sh up createChannel -ca
```

This starts the *Fabric* network and creates the default channel *mychannel*. The -*ca* flag activates the use of CA to verify users on the network.

Next we run the following command to deploy our *chaincode* to the channel with the name *journal*.

```
./network.sh deployCC -ccn journal -ccp ../auction/chaincode-go/ -ccl go -ccep "OR('Org1MSP.peer','Org2MSP.peer')"
```

From the */fabric-samples/application-javascript* directory, we can now take use of the *JavaScript API*, section 4.2.6, to connect and interact with the network using it's commands.

To install the node modules needed to use the *JavaScript API*, the following command is used in the application directory:

```
npm install
```

Following the steps provided in the README in the project repository the application can be run and tested using the *JavaScript API*.

Chapter 5

Conclusion

5.1 Summary

In this thesis we have implemented a blockchain based hospital data management. We used the framework, *Hyperledger Fabric*, to create a distributed and immutable ledger within a permissioned blockchain.

Using *X.509* certificates stored in a key-value wallet to authorize users accessing the network, we created per-object *ACL*s to secure the journals of patients. With this a user could own or have access to as many assets as needed and control who can access their assets. Authorization is divided into read, write and read/write giving good control over the object.

This allows a patient to create a journal, give their doctor access to it and be able to read any future changes. They would gain full control over who can access their journal while not being able to read others journals. A doctor can then have access to all their patients journals once given permission.

Our solution to a blockchain based hospital data management can provide an easier sharing process for journals in medical fields. We hope that hospitals will take inspiration from our work and implement a similar solution to their data storage.

5.2 Further work

Several changes could be made to improve our hospital data management. Since our project only uses one peer per organization it does not accomplish the desired immutability for endorsing transactions. By adding endorsing peers for each organization, the transactions would be safer from potential malicious attacks.

In our proposed network it is crucial that the peers are secured as the peer's ledger is not encrypted. This represents an opportunity for a malicious actor to access confidential data if the application portal or peer is exposed. A possible solution for this would be encrypting the data as it is stored, but this adds new difficulties of how to store and share the decryption key safely. We could also separate each user into distinctive organizations to take use of *Fabric*'s private data storage and secure data transfer. However *Fabric* does not seem to support a feature like this at the moment so further development is needed.

A simple addition would be to store different types of data instead of only journals. For example using a prefix in the key value of the object stored, other data such as prescriptions and x-ray imaging could be stored together with journals.

Other applications

Although a key-value type database with per object authorization is not new, it is normally not backed by blockchain technology to ensure the same level of immutability, reliability, and security. This type of blockchain based data management could therefore be used in many fields of applications, like a platform for sharing and collaborating on patents between patent offices and inventors. It could be used as a online diary for which you could approve your friends to read or write entries.

List of Figures

2.1 Illustration of double spending 4
2.2 The five main elements of blockchain [14] 6

3.1 Local and global MSPs [35] 14
3.2 The basic transaction flow in *Fabric* [41] 16
3.3 The five steps of an update transaction [39] 18
3.4 Three steps of a query transaction [39] 19
3.5 Illustration of the image layer structure, before and after a container has been generated [48] 21

4.1 Network structure of peers, orderers, wallets and the channel. 25
4.2 Network structure of the organisation peers and orderers. . 26
4.3 Flow of an update transaction. 27
4.4 Flowchart showing the authentication of the answer access request function. 31

Listings

4.1 The asset struct. 28
4.2 Creating the list of authorized users. 28
4.3 Creating the journal object. 28
4.4 *GetID* returns the unique client ID of the invoking user. . . . 29
4.5 Adding a user ID to the authorized users list. 29
4.6 Adding an access request to the access request list. 30
4.7 Approving an access request. 30

Bibliography

[1] Wikipedia, *Blockchain*,
https://en.wikipedia.org/wiki/Blockchain, (accessed Jan. 18, 2021).

[2] J. Frankefield, *Smart Contracts*,
https://www.investopedia.com/terms/s/smart-contracts.asp, (accessed Jan. 18, 2021).

[3] Wikipedia, *Nick Szabo*,
https://en.wikipedia.org/wiki/Nick_Szabo, (accessed Mar. 8, 2021).

[4] L. Meholm, *Kryptovaluta, bitcoin, ICOer og Blockchain*,
Norway: Hegnar media, 2018.

[5] S. Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*,
https://bitcoin.org/bitcoin.pdf, (accessed Jan. 17, 2021).

[6] J. Frankenfield, *Double-spending*,
https://www.investopedia.com/terms/d/doublespending.asp, (accessed Feb. 10, 2021).

[7] J. Frankenfield, *Proof of Work*,
https://www.investopedia.com/terms/p/proof-work.asp, (accessed Feb. 10, 2021).

[8] Trade Finance Global, *History of Blockchain*,
https://www.tradefinanceglobal.com/blockchain/history-of-blockchain/, (accessed Jan. 12, 2021).

[9] Wikipedia, *HyperLedger*,
https://en.wikipedia.org/wiki/Hyperledger, (accessed Feb. 22, 2021).

[10] HyperLedger Fabric, *Open, Proven, Enterprise-grade DLT*, https://www.hyperledger.org/wp-content/uploads/2020/03/hyperledger_fabric_whitepaper.pdf, (accessed Feb. 22, 2021).

[11] HyperLedger-Fabric, *Smart contracts and chaincode*, https://hyperledger-fabric.readthedocs.io/en/release-2.2/smartcontract/smartcontract.htm, (accessed Apr. 11, 2021).

[12] L. Conway, *Blockchain*, https://www.investopedia.com/terms/b/blockchain.asp, (accessed Feb. 10, 2021).

[13] Built In, *Blockchain*, https://builtin.com/blockchain, (accessed Feb. 10, 2021).

[14] K. Panetta, *The CIO's guide to blockchain*, https://www.gartner.com/smarterwithgartner/the-cios-guide-to-blockchain/, (accessed Feb. 25, 2021).

[15] Westpoint Recruitment, *Blockchain*, https://www.westpointrecruitment.com/blog/blog/types-of-man-in-the-middle-cyber-attacks, (accessed Mar. 12, 2021).

[16] D. Floyd, *How Bitcoin Works*, https://www.investopedia.com/news/how-bitcoin-works/, (accessed Mar. 12, 2021).

[17] Blockchain Hub, *Token Security: Cryptography - Part 2*, https://blockchainhub.net/blog/blog/cryptography-blockchain-bitcoin/, (accessed Mar. 29, 2021).

[18] A. Khaliq, *The good, the bad and the ugly of bitcoin security*, https://www.hongkiat.com/blog/bitcoin-security/, (accessed Apr. 7, 2021).

[19] J. Frankenfield, *Proof of Stake*, https://www.investopedia.com/terms/p/proof-stake-pos.asp, (accessed Feb. 10, 2021).

[20] P. Wackerow, *Proof-of-Stake (PoS)*, https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/, (accessed Apr. 8, 2021).

[21] J. Frankenfield, *51 percent attack*, https://www.investopedia.com/terms/1/51-attack.asp, (accessed Apr. 11, 2021).

[22] P. Jayachandran, *The difference between public and private blockchain*, https://www.ibm.com/blogs/blockchain/2017/05/the-difference-between-public-and-private-blockchain/, (accessed Apr. 10, 2021).

[23] S. Seth, Public, Private, Permissioned Blockchains Compared, https://www.investopedia.com/news/public-private-permissioned-blockchains-compared, (accessed Apr. 11, 2021).

[24] Wikipedia, *Recruitment of spies*, https://en.wikipedia.org/wiki/Recruitment_of_spies, (accessed Apr. 28, 2021).

[25] Broctagon Fintech Group, *Proof of Work or Proof of Waste?*, https://medium.com/broctagongroup/proof-of-work-or-proof-of-waste-f9d54e989cff, (accessed Apr. 28, 2021).

[26] Blockchain, *Mempool Size (bytes)*, https://www.blockchain.com/charts/mempool-size, (accessed Apr. 28, 2021).

[27] HyperLedger, *Forbes Blockchain 50: Half of the biggest companies deploying blockchain use Hyperledger*, https://www.hyperledger.org/blog/2019/04/18/__trashed, (accessed Apr. 29, 2021).

[28] R. Oclarino, *Blockchain's Technology of Trust*, https://www.iso.org/news/isofocus_142-5.html, (accessed Apr. 29, 2021).

[29] IEEE, *Blockchain and Distributed Ledger Standards Committee*, https://sagroups.ieee.org/bdlsc/, (accessed Apr. 29, 2021).

[30] Rand, *The Potential Role of Standards in Supporting the Growth of Distributed Ledger Technologies/Blockchain*, https://www.rand.org/randeurope/research/projects/blockchain-standards.html, (accessed Apr. 29, 2021).

[31] HyperLedger, *Hyperledger Passes 250 Members with addition of 9 organizations*, https://www.hyperledger.org/announcements/2018/07/31/hyperledger-passes-250-members-with-addition-of-9-organizations, (accessed Apr. 15, 2021).

[32] HyperLedger-Fabric, *Introduction*, https://hyperledger-fabric.readthedocs.io/en/release-2.2/blockchain.html, (accessed Apr. 21, 2021).

[33] HyperLedger-Fabric, *Blockchain Network*, https://hyperledger-fabric.readthedocs.io/en/release-2.2/network/network.html, (accessed Apr. 21, 2021).

[34] HyperLedger-Fabric, *Identity*, https://hyperledger-fabric.readthedocs.io/en/release-2.2/identity/identity.html, (accessed Apr. 21, 2021).

[35] HyperLedger-Fabric, *Membership Service Provider (MSP)*, https://hyperledger-fabric.readthedocs.io/en/release-2.2/membership/membership.html, (accessed Apr. 19, 2021).

[36] HyperLedger-Fabric, *Access Control Lists (ACL)*, https://hyperledger-fabric.readthedocs.io/en/release-2.2/access_control.html, (accessed Apr. 19, 2021).

[37] M. Felder, *A real-world example of a smart contract*, https://www.linkedin.com/pulse/real-world-example-smart-contract-marvin-felder, (accessed Apr. 12, 2021).

[38] HyperLedger-Fabric, *Hyperledger Fabric Model*, https://hyperledger-fabric.readthedocs.io/en/release-2.2/fabric_model.html, (accessed Apr. 19, 2021).

[39] HyperLedger-Fabric, *Peers*, https://hyperledger-fabric.readthedocs.io/en/release-2.2/peers/peers.html, (accessed Apr. 19, 2021).

[40] HyperLedger-Fabric, *The Ordering Service*, https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html, (accessed Apr. 19, 2021).

[41] Sumit V, *Hyperledger Fabric - Part 1 - Components and Architecture*, https://blog.clairvoyantsoft.com/hyperledger-fabric-components-and-architecture-b874b36c4af5, (accessed May. 13, 2021).

[42] K. Rilee, *Understanding HyperLedger Fabric - Byzantine Fault Tolerance*, https://medium.com/kokster/understanding-hyperledger-fabric-byzantine-fault-tolerance-cf106146ef43, (accessed Apr. 30, 2021).

[43] HyperLedger, *Private Data Collections: A high-level overview*, https://www.hyperledger.org/blog/2018/10/23/private-data-collections-a-high-level-overview, (accessed May. 1, 2021).

[44] Docker, *Glossary*, https://docs.docker.com/glossary/, (accessed May. 1, 2021).

[45] Docker, *Container*, https://docs.docker.com/glossary/?term=container, (accessed May. 1, 2021).

[46] Docker, *File*, https://docs.docker.com/glossary/?term=Dockerfile, (accessed May. 1, 2021).

[47] Docker, *Image*, https://docs.docker.com/glossary/?term=image, (accessed May. 1, 2021).

[48] Docker, *About Storage Drivers*, https://docs.docker.com/storage/storagedriver/, (accessed May. 2, 2021).

[49] Docker, *Compose*, https://docs.docker.com/glossary/?term=compose, (accessed May. 2, 2021).

[50] HyperLedger, *Wallet*, https://hyperledger-fabric.readthedocs.io/en/release-2.2/developapps/wallet.html, (accessed May. 3, 2021).

[51] HyperLedger Fabric, *fabric-samples*,
`https://github.com/hyperledger/fabric-samples`, Version 2.3.2

[52] HyperLedger Fabric, *Cid.go - GetID()*,
`https://github.com/hyperledger/fabric-chaincode-go/blob/main/pkg/cid/cid.go`, Version 2.3.2