Blockchain in Healthcare

Dharita Queshil Chotalal

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Prof. José Manuel da Costa Alves Marques
Eng. Marco Paulo Travassos de Freitas

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. José Manuel da Costa Alves Marques
Member of the Committee: Prof. Paolo Romano

October 2018

For my parents and my best friend...

Acknowledgments

Firstly, I thank my advisors, Professor José Alves Marques, Eng. Marco Freitas and Eng. Tiago Santos, for welcoming me in Link Consulting and for their enthusiastic guidance throughout this work. I thank my family and João Pedro Guita de Almeida for the endless support and encouragement throughout my studies. Lastly, I thank all my friends who have turned the last few years into a delightful journey.

Resumo

Os registos médicos eletrónicos (RME) representam informação altamente sensível que necessita de ser partilhada de forma segura entre diferentes prestadores de serviços de saúde. O nosso objetivo com esta dissertação foi criar um sistema para, de forma segura, armazenar e partilhar RME usando a *Blockchain*. A *Blockchain* é uma tecnologia descentralizada, imutável e fornece garantias de não-repudiação dos dados. Estas propriedades criam uma oportunidade para desenvolver um sistema para armazenar RME utilizando *Blockchain*.

Neste relatório foi estudado o trabalho relacionado relevante acerca da *Blockchain* e os diferentes protocolos de consenso que podem ser aplicados à *Blockchain*. Os tipos diferentes de *Blockchain*, bem como as plataformas que permitem o desenvolvimento de aplicações descentralizadas foram meticulosamente detalhados na secção de trabalho relacionado. Aplicações da área da Saúde que exploram tópicos relacionados com este trabalho, tais como a utilização da *Blockchain* para resolver problemas nos atuais serviços de saúde, também foram estudados e detalhados.

Também é explicado o modelo do sistema e todas as considerações relacionadas com a implementação de forma a permitir que os registos de saúde de um utilizador sejam armazenados de forma a que esse utilizador possa aceder aos seus dados e fornecê-los a diferentes provedores de serviços de saúde a qualquer altura. A arquitetura proposta, à qual chamamos *HyperMedical*, permite que os dados armazenados sejam visualizados, inseridos, alterados, partilhados ou eliminados pelas entidades participantes no sistema dependendo das políticas de acesso definidas. O *HyperMedical* melhora a forma como os registos médicos são partilhados ao, simultaneamente, garantir a privacidade dos dados e do dono destes dados, disponibilidade de serviço, imutabilidade dos dados e gestão de controlo de acesso.

Palavras-chave: *Blockchain*, *Hyperledger Fabric*, Serviço de Saúde, Descentralizado

Abstract

Electronic health records (EHRs) are highly sensitive information that needs to be shared in a secure way with different healthcare providers. Our goal with this dissertation was to create a system to securely store and share EHRs using blockchain technology. The blockchain technology is decentralized, immutable and guarantees non-repudiation of the data. These properties create an opportunity to develop an EHRs system using blockchain.

In this report we have surveyed the relevant related work regarding blockchain technology and the different consensus protocols that can be applied to the blockchain. The different existing blockchains, as well as platforms that enable the development of decentralized applications are meticously detailed in the related work section. Health applications that explore topics affiliated to this work, such as using the blockchain to solve issues with the current healthcare services, were also studied and detailed.

In this report is also explained the system model and all the implementation considerations to enable the electronic health records of a user to be stored so that the user can access his data and provide it to the different healthcare providers at any time. The proposed architecture, which we named HyperMedical, allows stored data to be viewed, inserted, altered, shared or deleted by certain entities depending on defined access policies. HyperMedical improves the way that medical records are shared by simultaneously ensuring data and owner privacy, service availability, data immutability and access control management.

Keywords: Blockchain, Hyperledger Fabric, Healthcare, Decentralized

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays, the world is led by information. People, as a way of dealing with this huge amount of information, give up their own personal data and preferences to get a personalized view of what exists in the market. All collected data about the population lifestyle lead brands to adapt their products and people have become precious targets whose profiles are sold and converted into marketing strategies worldwide.

From this volume of data, arise several concerns as the information being spread by several systems, service availability and storage security can affect user's privacy.

The concern of data dispersion also affects the health area, which will be the explored area. In Figure 1.1, it is possible to observe the multiple healthcare providers. The concern is our medical data being spread through all these entities.

It is crucial to understand what is a health record and why it is important. The health record of a patient is a set of documents that contains information about the health status, personal history, examinations performed, contacts with health services, surgeries, therapeutics among other data. This set of documents aims to improve healthcare efficiency and effectiveness, so it is necessary that the health record of a user is available when necessary. There are also other purposes that need this information, such as clinical research and health services management [1].

The Electronic Health Record (EHR) which is the subset of Health Record or Medical Record that is digitally stored information, for simplifying shared care, training and research. The EHR is a tool to help the continuity of the quality, access and efficiency of healthcare delivery [2].

Figure 1.1: Healthcare Providers

1.1 Motivation

The EHRs are stored in each healthcare provider in centralized databases, the Governance, consequently exists a central entity that could be consider a single point of failure. Another problem in the healthcare services is the fragmentation of health records [3]. Electronic Health Records are fragmented across hospitals, health centers, doctors, insurance companies and wearable devices. The fact that patients can't precisely describe what was said by one doctor and the lack of information sharing mechanisms between medical entities, can lead to inaccurate diagnosis in different appointments. To increase the probability of an accurate diagnosis and more effective treatments it is important to have a system that allows all the participants to view, insert, modify, share and delete EHRs.

However, data is a sensitive matter. This way, the storage mechanisms used should not only have precautions related to security, integrity and availability of the data, but also different access control levels to manage what entities get to consult what type of data. Another issue to be solved is service availability and data immutability.

To solve these problems, it was decided to study and create a user application that uses Blockchain technology as Back-end service. This way, creating a unified solution for all the entities related to the Healthcare system and discarding the necessity of relying trust on a central authority. Because Blockchain technology is immutable, the data will be preserved and it will be also stored in a decentralized way.

It was made an attempt in Portugal to try to solve the problem of data fragmentation by allowing a user to access and share his EHRs. The Portuguese National Health Service [4] has a portal that allows access to the data collected by the public service and private services. Private entities have their own platforms, as it happens in *Hospital da Luz* [5] and in *Hospital Lusiadas* [6].

To solve the system's challenges could be used a centralized database, but these type of solution store all the data on a single database and are highly dependent on the network connectivity. Other relevant issues are related to availability, scalability and business model. A distributed database was not chosen to solve this problem because in this type of database it is assumed that all the data should be distributed among several computers and all nodes trust each other and will not misbehave or leak data [7].

Given the system's challenges, such as access control, information preservation, data authentication and non-repudiation mechanisms, the Blockchain is a possible approach that not only has these features but also solves the main problem of data fragmentation.

Blockchain was introduced in the paper *Bitcoin: A Peer-to-Peer Electronic Cash System* [8]. This technology was created to solve the problem of double spending, so transactions could be carried out without having a central system that dealt with such transfer as happens in banking institutions. By definition, according to [9], a blockchain is a decentralized transparent ledger for transaction records — an infrastructure that is shared by all network nodes, updated by miners, monitored by everyone, and owned and controlled by no one. To support the solution it will be used a permissioned blockchain, to be possible know who the entities involved in the network are.

1.2 Objectives

The main goal of this project was to enable the electronic health records created by different healthcare providers to be stored so the user can access his data and provide it to the different healthcare providers at any time. It is allowed to stored data to be viewed, inserted, altered, shared or deleted by certain entities depending on defined access policies.

The blockchain technology solves the data fragmentation problem and also helps to preserve all the data collected about the patient in such way that the information stored in the chain cannot be altered nor deleted, but can be modified from a user's perspective by adding new information on the chain. Provides an efficient exchange of health records between entities without compromising data privacy and respecting owner's privacy.

By law, the user is the owner of his personal information. This way, the user has the right

to access his clinical process, to define a set of policies that state who can access his data and even if his medical history has to be preserved.

1.3 Thesis Outline

The remainder of the document is structured as follows. Chapter 2 explains Blockchain technology and Healthcare systems that used Blockchain as technology. Chapter 3 explains the system model and data strutures created. Chapter 4 describes the tecnical design and implementation issues. Chapter 5 describes how the architecture was evaluated. Chapter 6 presents conclusions regarding this dissertation.

Chapter 2

Background

In this section, the Blockchain technology will be detailed, consensus protocols and the main applications that use Blockchain technology, Bitcoin, Ethereum and Smart Contracts. It will be described technologies that can help to accomplish the project's goals and solutions that solve a parcel of the problems in EHRs. In the end of this section, it is detailed the legal framework regarding sensitive data.

2.1 Blockchain

The technology known as the Blockchain was introduced by Satoshi Nakamoto [8] in 2008 to solve the problem of Double Spending in digital currency. The author/s introduced the concept of Bitcoin, which is the digital currency that uses Blockchain as technology.

The blockchain is a distributed peer-to-peer (P2P) system protected by cryptographic methods to allow only those who have permission to get access to the data. Blockchain uses a peer-to-peer (P2P) protocol, as in BitTorrent [10]. Blockchain combines a peer-to-peer networks and uses distributed consensus algorithm to solve traditional distributed database synchronize problem [11]. According to *Blockchain & Alternative Payment Models* [12], blockchain can be considered a protocol to solve the Byzantine Generals' Problem [13], a famous computer science problem that aims to reach consensus in a decentralized system in presence of faults.

This technology is composed by six key attributes [11]:

- Decentralized: Blockchain is a P2P system so it does not need a central server, all machines are equal and each machine provides a piece of its resources. In a P2P system potentially, the higher the number of machines, the faster communications will be and consequently the service performance.

- Transparent: The stored data in the blockchain can be seen by all stakeholders.

- Open Source: It is possible to freely create application using blockchain technology.
- Autonomous: Due to the consensus protocol, any node can safely transfer or update data by trusting in the system and not in a single person.
- Immutable: Any information stored in the blockchain cannot be deleted and modified unless an entity is able control 51% of the network nodes.
- Anonymity: To execute a transaction, it is only necessary to know the person's blockchain address. This address can not be linked to the owner's real IP address.

It is considered the existence of three types of blockchains [11] [14] [15]:

1. Public blockchain: Anyone is able to read or perform transactions and expect them to be verified. All the nodes can participate in the consensus process. A consensus process will decide which blocks are inserted and what is the current state. Public blockchain is generally considered completely decentralized. Bitcoin and Ethereum are examples of public blockchains.
2. Consortium blockchain: Permission to read blocks can be public or restricted to certain participants. The consensus process is only obtained by the nodes defined in the network configuration. Consortium blockchains are, generally, considered partially decentralized. Hyperledger is a consortium blockchain.
3. Fully Private blockchain: Read permissions can be public or restricted, but write permissions are unique to a single entity.

Public blockchains are also known as permissionless and the consortium and private blockchains are known as permissioned blockchains. Permissioned blockchains are faster, easier to implement and energy-efficient [16]. These types of blockchains are capable of supporting consensus processes other than Proof-of-Work [10] or Proof-of-Stake [17] which consume a high quantity of energy. Also, the throughput of these types of consensus processes could be lower than the consensus adopted in permissioned blockchain implementations, such as the implementation of a Byzantine Fault Tolerance consensus algorithm [18].

The blockchain is a chain of blocks, as can be seen in Figure 2.1. A block is always dependent on the previous block. A block is created by the nodes that participate in the consensus protocol. Each block is composed of two parts, according to [19], the block header and the block body as illustrated in Figure 2.2.

The block header usually is composed by:

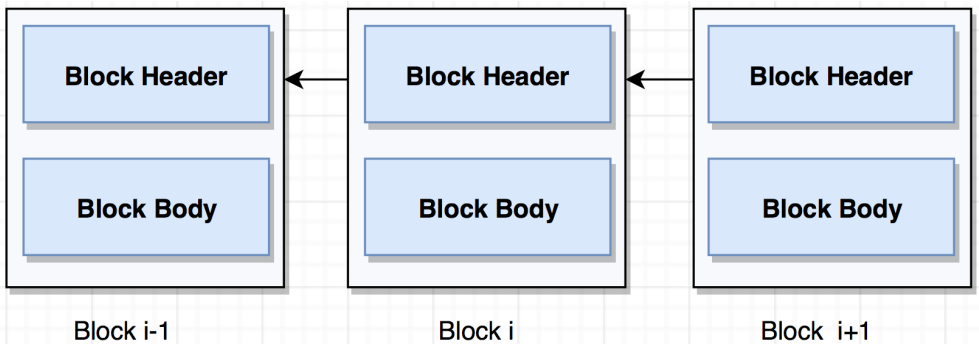

Figure 2.1: Blockchain

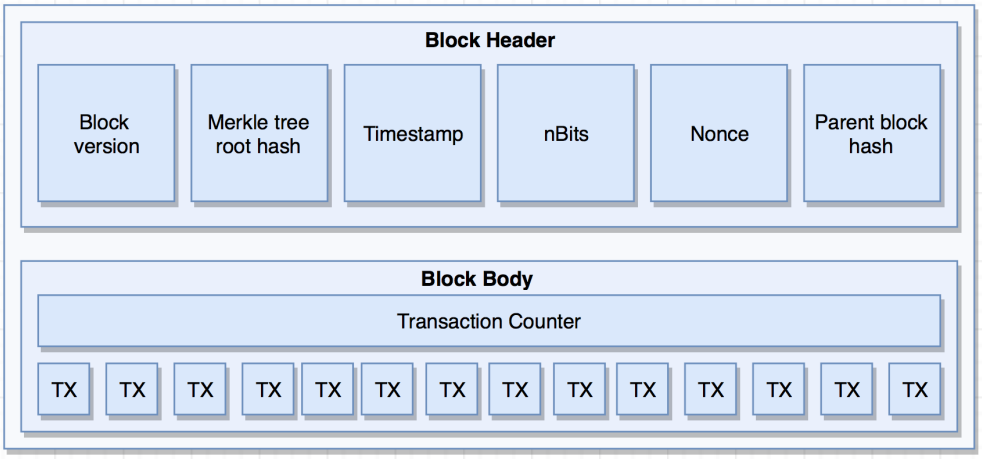

Figure 2.2: Block Structure

- *Block version* indicates which set of block validation rules to follow;
- *Merkle tree root hash* is the hash value of all the transactions in the block;
- *Timestamp* is the current time in seconds in the universal time since January 1, 1970;
- *nBits* is the target threshold of a valid block hash;
- *Nonce* is a 4-byte field, which usually starts with 0 and increases for every hash calculation;
- *Parent block hash* is a 32-byte hash value that corresponds to the previous block.

The block body contains the number of transactions of that block and the transactions itselves, as represented in the Figure 2.2.

In general, the blockchain protocol works as follows [11]:

1. The node creates new information and broadcasts it to all nodes of the network.
2. A node creates a new block when it receives correct information and has enough information stored to create a block with a minimum number of transactions.
3. The nodes execute the consensus algorithm on the block. If consensus is successful, the block will be added to the chain.

Due to the security requeriments a Consortium Blockchain is the type of blockchain that could help achieve the goal of the project. The consortium blockchain allows the definition of access permission types that restrict the access to the medical records of each user.

We further conclude that transactions are digitally signed, leading to a non-repudiation of the data. All entities keep a copy of the transactions, so it is verified ensuring that it has not been modified [20]. It will also be possible to solve the problem of data preservation because the blockchain is immutable.

The two most well-known applications of the blockchain data structure are Bitcoin and Ethereum, which are explained in section 2.3 and 2.4, respectively.

2.2 Consensus

Consensus algorithms aim to make a uniform decision [16]. A consensus protocol ensures *agreement*, this is, all honest nodes will agree on which transaction are selected, *termination* in which all honest nodes eventually learn the chosen transaction, and *validity* which guarantees that the selected transactions were actually proposed by some node [21].

There are many consensus protocols that can be applied to the blockchain, such as Proof of Work (PoW), Proof of Stake (PoS), Proof of Existence (PoE), Proof of Activity (PoW and PoS hybrid), Proof of Elapsed Time (PoET), Proof of Importance, Delegated Proof of Stake (DPoS), Practical Byzantine Fault Tolerance, Ripple, Tendermint, among many others [16]. Some of these consensus protocols will be explained in the next subsections.

2.2.1 Proof of Work

The consensus protocol used in Bitcoin and Ethereum is Proof of Work (PoW). According to the creator of Bitcoin [8], the goal of PoW is to find a value such that when the block header is hashed with SHA-256, it begins with a certain number of zero bits. To define this protocol, the author based on a similar system, Adam Back's Hashcash [22].

To understand Proof of Work it is important to understand what is the *target*. The "*Target*" is a 256-bit number shared by all Bitcoin clients and is set so the mining time limit is 10 minutes. According to *Mastering Bitcoin* [10], the target value will be modified every 2016 blocks [11] using the following equation:

$$NewTarget = OldTarget * (ActualTimeofLast2016Blocks/20160minutes)$$

The nodes that execute this consensus protocol are called miners. Miners will construct a block that contains transactions. Then, the miner will calculate the hash of the block's header and verify if it is lower than the current target. If the hash is not lower than the target, the miner will increment the nonce by one and repeat the process [10]. When a miner is able to get the desired value, it will broadcast the nonce so that other miners in the network can verify it. If the number of mining nodes, defined in the consensus rules by the developers, validates the nonce, consensus is achieved and a new block will be inserted into the chain [19].

If it is possible to find a nonce such that the header's hash is smaller than the target in less than 10 minutes, the difficulty[1] will increase, otherwise, it decreases.

PoW is essentially a one-CPU-one-vote protocol, because SHA-256 is a completely unpredictable pseudo-random function and the only way to create a valid block is by brute-force testing different values for the nonce [23].

This consensus mechanism will prevent Sybil attacks [2] because the creation of a block requires considerable processing which would result in a waste of resources. Proof of work can be performed by any node in the network, as long as it has enough computational power [23].

[1] Difficulty is the amount of processing power required to find a hash below a given target.
[2] In Sybil attacks an attacker creates and controls more than one identity in a single device [24].

2.2.2 Proof of Stake

The algorithm Proof of Stake (PoS) compared to Proof of Work is an energy saving solution [25]. PoS replaces PoW's competition by offering the authority to update the blockchain to a randomly selected stakeholder [26]. PoS offers an explicit monetary reward to update the blockchain but does not impose any explicit cost upon agents to gain the authority to update the blockchain [26]. For the PoS algorithm there are different types of implementations like Nxt [27], BlockCoin [28] or PPCoin [17] that is explained in more detail.

The PPCoin algorithm was introduced in 2012 in the article *PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake* [17] is a hybrid solution between PoS and PoW. In this article, it was introduced the concept of coin age, which is defined by the authors as: "currency amount times holding period"[17]. For example, if Bob received 10 coins from Alice and did not spend them for 90 days, that means Bob has 900 coin-days of coin age.

PoS blocks contain a new type of transactions called coinstake. In this type of transaction the owner will consume his coin age, and gain privileges to generate a new block and mint for proof of stake.

The coinstake transaction is composed by the Kernel input and the Stake input. The Stake input is the reward added by the nodes that will mine this block. The Kernel input has to follow a certain hash target protocol to create the block in a similar way to proof of work blocks.

The hash target in PoS is a coin-day consumed, whereas in Bitcoin it is a fixed target applied to every node. Therefore, in PoS finding the hash is dependant on the coin age consumed, if users have more coin age, the easier it is to discover the hash target.

In this new type of consensus, it is possible to reduce the possibility of malicious attacks. This is given the fact that to carry out an attack is expensive and it will reduce the incentives for the attacker since he would have to own the majority bitcoin to mint the block and consuming his own coin age [11].

The consensus protocol used by Ethereum is PoW, but the creator of Ethereum declared that it will be modified to a hybrid version of PoW and PoS, which will be called Casper [29].

2.2.3 Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (PBFT) is a replication algorithm that supports Byzantine faults. This protocol was created to offer a solution to Byzantine Generals Problems [30]. PBFT is only used in a permissioned blockchain where all network stakeholders are known.

The components of this protocol are the primary node and replicas. The total number of replicas will be equal to $n = 3f + 1$, where n is the total number of replicas and f is the amount

of failures the system can handle. To ensure safety and liveness proprieties, it is assumed that is not possible to have more than $(n-1)/3$ faulty replicas.

In Figure 2.3 are illustrated the three phases of this algorithm, pre-prepare, prepare and commit. At each stage, a node votes for a value until it reaches a byzantine quorum, it is only moved to the next phase if each node has received the answer of (2/3) of the nodes [19].

Cryptographic methods such as digital signatures, message authentication codes and hash functions prevent this algorithm from spoofing attacks, replay attacks and detect corrupted messages. An access control mechanism was created for customers to authenticate themselves and which prohibits access in case they are trying to request a resource that they don't have permissions to access.

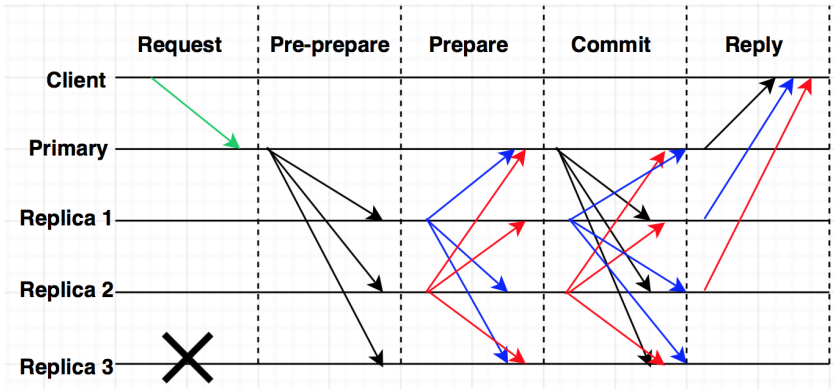

Figure 2.3: Practical Byzantine Fault Tolerance

In general, the algorithm works as follows [30]:

- Primary node receives digitally signed requests containing a timestamp from the client and will multi-cast to the replicas;
- Replicas execute the request and send the digitally signed response to the client;
- Client waits for $f+1$ replies from different replicas. If the digital signatures are valid, timestamps and the results of the request are equal it becomes the result of the request made.

If the replies do not arrive in time, the client broadcasts the request to all replicas. If the request was already processed, the replicas re-send the reply to the client and send the request to the primary node; replicas save the last reply sent to each client. If the primary node does not multicast the request to the replicas, it will eventually become suspected and may occur a view change, this means that the primary node was modified.

PBFT could be adapted to be used with a blockchain solution, as examples of this adaptions are: BFT-SMaRt [31] and Tendermint (section 2.2.5)

2.2.4 Ripple

In the Ripple consensus protocol [32], nodes are divided into two types, the server that participates in the consensus process and the client that will transfer funds.

This consensus algorithm is composed by six components [32] [19]:

- *Server* executes the Ripple Server software and will take part in the consensus process;
- *Ledger* keeps the amount of money in each account and is updated when a transaction successfully completes the consensus;
- *Last-Closed Ledger* is the ledger where the consensus process was successfully applied;
- *Open ledger* store the transactions while the consensus process is taking place;
- *Unique Node List (UNL)* is a list of the servers to communicate to complete the consensus process;
- *Proposers* are the only ones that can propose a value during the consensus process.

This consensus protocol works in rounds and in each round [32] [19]:

1. Each server forms a public list with all valid transactions that have not gone through the consensus process, corresponding to the "candidate set";
2. Each server will join all the "candidate sets" in the UNL and vote to send their opinion about the transactions;
3. Transactions that earn more than the minimum percentage of positive votes can go to the next round. If this does not happen they will be discarded or included in the candidate set again;
4. In the last round, it is mandatory to have at least 80% of the servers from UNL to agree on the transaction, it means that at least 80% of validating servers from the UNL should have the same set of transactions;
5. If all transactions meet these requirements the ledger will be closed and become the last-closed ledger.

2.2.5 Tendermint

Terdermint [33] is a Byzantine consensus algorithm. The nodes are connected using a peer-to-peer architecture and the information is transmitted through gossip. The nodes are designated

as validators because these are the ones who propose the blocks and vote. This algorithm is divided into four steps [19][33]:

1. Propose: The chosen proposer for that round will broadcast a block to the neighbours by gossip.

2. Prevote: Each validator must decide on the validity of the block and broadcast the decision. In case the block is invalid or does not arrive on time, its prevote will be nil.

3. Precommit: If a node receives more than 2/3 of prevotes, it does a broadcast of the precommit of the block, otherwise it sends nil. If this node receives 2/3 of precommit it commits.

4. Commit: If the node validates the block and broadcast the commit, and receives 2/3 of commits it accepts the block.

2.2.6 Summary of Consensus Algorithms

Table 2.1 demonstrates that in a permissioned blockchain the main consensus algorithms used are the Practical Byzantine Fault Tolerance and the Tendermint algorithms. It is illustrated that only the PBFT, Ripple and Tendermint are consensus algorithms that are energy savers. In the third column of these table is represented the percentage of faults tolerated by the algorithms to successfully finalize consensus.

Table 2.1: Consensus Algorithms [19]

	Node identity management	Energy Saving	Tolerated power of adversary	Example
PoW	Open	No	<51% computing power	Bitcoin / Etheruem
PoS	Open	Partial	<51% stake	Peercoin
PBFT	Permissioned	Yes	<33.3% faulty replicas	—
Ripple	Open	Yes	<20% faulty nodes in UNL	Ripple
Tendermint	Permissioned	Yes	<33.3% byzantine voting power	Tendermint

2.3 Bitcoin

Bitcoin was proposed by Satoshi Nakamoto [8] as a peer-to-peer (P2P) distributed timestamp server to generate computational proof of the chronological order of transactions. According

to [34], a blockchain is a cryptographically secure decentralized peer-to-peer (P2P) electronic payment system and it enables transactions involving virtual currency in the form of digital tokens.

Bitcoin properties that enable validity in a decentralized system [10] are composed by:

1. Public ledger, which is a blockchain;
2. Set of rules to confirm transactions and currency issuance, which are the consensus rules;
3. Proof of Work mechanism to obtain consensus for the blockchain.

The wallet is the data structure to store the pairs of private/public keys. These pairs are used in the digital signature of the data which will be inserted into the blockchain. To create the digital signature is used the Elliptic Curve Digital Signature Algorithm (ECDSA). The public key is used to create a Bitcoin address used to identify the owner of the bitcoin. This address is a double hash of the public key, $RIPEMD160(SHA256(Kpub))$.

Digital signatures have three goals on the Bitcoin protocol:

- Proves that the owner of the private key is who implicitly owns the amount that will be spent, authorized this debit.
- Guarantees non-repudiation since it works as proof of authorization.
- Proof that the transaction or a specific signed part of the transaction, has not been and cannot be modified by anyone.

It is considered the existence of two types of wallets, deterministic or non-deterministic. The difference between these two types of wallets is whether the stored keys are related to each other or not. In deterministic wallets, all keys are generated from a master key, known as the seed, wherein non-deterministic wallets keys are generated independently from a random value.

In the Bitcoin protocol, the concept of a transaction, functions as a double-entry bookkeeping ledger. Two types of transactions are:

1. *Output transaction* composed by:
    - Amount of bitcoin in satoshis (0.00000001 Bitcoin);
    - Cryptographic puzzle that determines the necessary conditions to spend that value.
2. *Input transaction* identifies, by reference, the *unspent transaction output* (UTXO) that was spent and the proof of ownership from an unlocking script.

To build a transaction, the wallet selects from the UTXO which has enough value to make the payment that the user requires. Transactions propagate to the network using the flooding method, so they propagate more quickly through the P2P network. The transaction also has proof of ownership for each amount of bitcoin that is spend in the form of a digital signature. The digital signature can be validated by any node in the network.

When a block is created, it has a timestamp, a nonce, the previous block's hash and the data structure's root hash labelled Merkle tree that stores all transactions in the block. Merkle tree is a type of binary tree that allows a node to get information about a certain block from different sources, but having the guarantee that the data will be correct. The hash of the transaction is propagated upward in the Merkle tree, if a malicious user tries to change a transaction, it will end up generating a change in the root hash causing an error in the protocol.

Nodes that compose the bitcoin network can be full nodes or simplified payment verification (SPV) nodes. The full nodes keep a complete and updated copy of the blockchain, being able to verify any transaction without needing an external reference. SPV nodes keep part of the blockchain, to verify transactions will be used the SPV method. SPV nodes only contain the headers of the blocks, establishing the link between the transaction and the block using the Merkle path. To guarantee the privacy of SPV nodes were created bloom filters which are filtering mechanism than fixed patterns. The bloom filters allow nodes to get access to the transactions without revealing which are addresses they are interested. Mobile phones or tablets are examples of SPV nodes.

According to [8], the steps of the protocol are the following:

1. New transactions are broadcast to all nodes;
2. Each node collects new transactions into a block;
3. Each node works on finding a difficult proof-of-work for its block;
4. When a node finds a proof-of-work, it broadcasts the block to all nodes;
5. Nodes accept the block only if all transactions in it are valid and not already spent;
6. Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

2.4 Ethereum and Smart Contracts

The goal of Ethereum was to develop an alternative protocol to create decentralized applications. To develop an alternative protocol was created a blockchain with a built-in Turing-complete pro-

gramming language that allows anyone to write a smart contract and decentralized applications where they can define their own rules of ownership and formats of transactions.

The concept of smart contract first appeared in 1994, introduced by Nick Szabo. The author of the article [25] defined smart contract as a computerized transaction protocol that executes the terms of a contract. In the context of blockchain, the creator of Ethereum [23] defined smart contracts as cryptographic "boxes" that contain value and only unlock it if certain conditions are met. Smart contracts have vastly more power than that offered by Bitcoin scripting because it was added powers of Turing-completeness, value-awareness, blockchain-awareness and state.

Ethereum Virtual Machine (EVM) is the biggest innovation in Ethereum. EVM is a large decentralized computer that has millions of objects, called "accounts". It also introduced the concept of message and the notion of transaction was reformulated.

Accounts have four fields:

- *Nonce* represents the counter to ensure that a transaction is handled only once;
- *Ether balance* (digital currency of Ethereum) of the account;
- *Account contract code*, if required;
- *Account storage* (empty by default).

The accounts maintain internal databases, execute the code and communicate with each other. According to the creator of Ethereum, there are two types of accounts, the externally owned accounts and the contract accounts. Externally owned accounts are controlled by the private key, the contract field is empty and it is possible to send a message by creating and digitally signing a transaction. Contract accounts are controlled by the contract code, each time this type of account receives a message the contract code will be executed allowing reads or writes on the internal storage and sending other messages or creating contracts.

The smart contract is executed at the EVM [35], written using the Solidity programming language that is a contract-oriented, high-level language whose syntax is similar to JavaScript [36]. These contracts are autonomous agents of the Ethereum execution environment. Smart contracts are executed when triggered by a message or a transaction. Smart contract have access to the ether balance and the saved data in the account to guarantee the compliance with the persistent variables [23].

A transaction in Ethereum is a signed data packet that holds the message to be sent to an externally owned account.

The transaction is composed by [23]:

- Recipient of the message;
- Identification of the sender through the digital signature;
- Quantity of ether to be transferred;
- Optional data field that by default has no defined function, but each EVM has an opcode with which the contract code can access this field;
- STARTGAS represents the largest number of computational steps the transaction is allowed to perform;
- GASPRICE represents the rate that the sender has to pay per computational step;

The last two fields set the cost of the transaction and are crucial for securing an anti-denial-of-service model. To prevent infinite loops or computational wastage, each transaction is required to set a limit of computational steps of code execution.

When a transaction is added to the block or when the block is validated, the contract will be executed. Each contract has a unique address that identifies it. All nodes execute the contract using their EVMs and will agree on the output of the contract and the new state is obtained in the consensus [23] [37].

Messages are like transactions but they are created by other contracts and not by an external actor. The message is created when a contract executes the system call $CALL$ opcode, which will produce and execute the message.

The components of a message are [23]:

- Message Sender (implicitly);
- To whom the message will be sent;
- Amount of ether transferred with the message;
- Optional data field;
- STARTGAS value;

According to [23], the block validation algorithm in Ethereum has to:

1. Verify if the previous block exists and is valid;
2. The timestamp of the block is greater than the previous block and less than 15 min into the future;

3. Verify if the block number, difficulty, transaction root, uncle root and gas limit are valid;

4. Verify if the proof of work has been done with success;

5. Save the most recent state in the block;

6. Check if the Merkle tree root of the most recent state if equal to state provided in the block header.

The biggest difference between Ethereum and Bitcoin is the smart contract and the block components. In Ethereum each block has a copy of the list of transactions, the most recent state, the block number and the difficulty used in the consensus protocol.

2.5 Platforms

There are several platforms that have already explored blockchain as technology. The platforms that will be studied in more detailed are Quorum, Chain Core, Corda and Hyperledger platforms. Microsoft Azure supports Chain Core, Hyperledger fabric, Corda and among others as Blockchain as a Service (BaaS).

2.5.1 Quorum

Quorum [38] is a private/permissioned blockchain based on the Go implementation of the Ethereum protocol. Quorum is ideal for applications that require high-speed, high-throughput processing in transactions between known entities, such as applications in the financial services industry. The main feature of this tool is the protection of sensitive data so it cannot be viewed by those who are not authorized.

To protect sensitive data was created a private transaction. To create this new type of transaction the base code of go-ethereum was adapted because the proposal for a new block and the validation process were different. As already explained in the 2.4 section, if transactions contain code this will be executed by all nodes in the network. In Quorum, this process occurs in a similar way, but the private transaction will only be verified by whom has permissions, for the others this transaction will be ignored. Data privacy in Quorum is achieved through cryptography applied to the data in transactions and segmentation that is applied to each node's local state database which contains the contract storage and is only accessible to the node. To ensure privacy, Quorum includes several sharding properties. As defined in [38], sharding is a segmentation of the transactions therefore only public transactions are processed by all nodes.

Each network participant will save the blockchain, a private and a public database. Not all the nodes have access to private transactions, so there will only be consensus in the public state. The consensus algorithms available in Quorum are Raft-based and Istanbul BFT. The creators are implementing the QuorumChain protocol to obtain consensus using a majority voting protocol. This protocol will be implemented through smart contracts because they control who and how the network will be organized. In the majority voting protocol, nodes have the power to vote in the blocks, will be inserted the valid block with the highest number of votes in the blockchain.

2.5.2 Chain Core

Chain Core [39] is a software that was built to be able to manage issuance, ownership and control of digital assets, in the financial area using permissioned blockchain.

Each asset has an ID associated, which is unique among all blockchains. Each asset ID corresponds to an issuance program, which aims to define the rules for issuing units of that asset. From the moment these units were issued, the rules to be consumed are determined by control programs.

Transactions transfer values from the input to the output. An input specifies the source of the value, which can be the output of a previous transaction or a new value. An output stipulates the destination of the value, which is the control programs. Each transaction will have one more field that will be the witness. According to the authors of the article [39] this field is not included in the transaction ID so it is not protected by the digital signature. When a block is created it must satisfy the consensus algorithm defined in the header of the previous block. In this way, it guarantees that unauthorized participants cannot create new blocks.

The consensus protocol used is the federated consensus protocol. In this protocol, the block is only accepted if it has been signed by a quorum of block signers. The rule used, defines that it must exist *M-of-N multi-signatures*, where N is the number of block signers and M the number of signatures required to the block be accepted. It is considered that the network is secure through a federation of block signers. When a new block is created, it can use the same consensus protocol or change it if it has been approved by the majority of the block signers.

2.5.3 Corda

The creators of Corda [40] have defined this platform as a distributed ledger platform for recording and processing financial agreements, with pluggable consensus. This platform was built to be used in regulated financial institutions [16]. The network of Corda is semi-private, with the

access controlled by a doorman [41]. To enter the network, a node has to contact the doorman, when the node is accepted it can communicate with other nodes using point-to-point. The identities of the nodes are attested by X.509 certificate signed by the doorman. The most important definition of this platform is the state object which, according to the authors of Corda [40], is a digital document that records the existence, content and current state of an agreement between two or more parties. This object will be shared with those who can view them to ensure system's consistency. It is possible to guarantee this consistency based on the secure cryptographic hash to identify entities and data. Corda has three main characteristics to help achieve distributed consensus globally:

- Smart contracts to ensure the validity of the transactions;
- Timestamping and singularity to guarantee the order of the transactions and eliminate conflicts;
- Framework to generate a template of the protocol between entities.

To run and validate the smart contracts was used the Java Virtual Machine with a custom sandbox to enforce the security requirements and the deterministic execution. The transaction in this platform is a proposal to update the ledger and it will be committed if the transaction achieves the consensus when these two conditions are met:

- Transaction validity
- Transaction uniqueness

To acquire validity consensus the contract must be valid and signed by all the required entities. The uniqueness consensus will prevent the double spending problem.

2.5.4 Hyperledger

Hyperledger is an open source project hosted by Linux Foundation. The goal of Hyperledger is developing an open source tool that allows the creation of distributed ledgers to support business transactions.

Hyperledger is composed by four frameworks:

1. Hyperledger Burrow [42];
2. Hyperledger Iroha [43];
3. Hyperledger Sawtooth [44];

4. Hyperledger Fabric [45].

Hyperledger Burrow [42] was created by Monax [46] and co-sponsored by Intel. Burrow's aim is to create a secure framework and ensure data privacy. This framework executes Ethereum smart contracts using a permissioned virtual machine, an adaptation of the EVM. To develop the smart contract, Monax provides the *monax chains*. The consensus protocol used in this framework is Tendermint (section 2.2.5).

Hyperledger Iroha [43] authors are developing a blockchain with C++ emphasizing on mobile applications and have created chain-based Byzantine Fault Tolerant consensus algorithm, called Yet Another Consensus. Iroha is designed to create and manage custom complex assets, user accounts, access control policies for transactions execution and queries and validation of business rules for transactions and queries. The blockchain-powered applications could be written in Python, Java, JavaScript and C++ as well as for Android and iOS mobile platforms.

Hyperledger Sawtooth [44] is an enterprise blockchain project. In this framework was created the concept of *Transaction families* that correspond to the smart contracts in Ethereum. The transaction families are used to define the business logic. Sawtooth has two implementations that can be used as consensus algorithms. The first one is Dev_mode, this algorithm is explained in the Sawtooth documentation [44] as "a simplified random leader algorithm that is useful for developers and test networks that require only crash fault tolerance". The second one is Proof of Elapsed Time (PoET) [47], this is one of the solutions for the Byzantine Generals Problem. PoET uses a trusted execution environment to improve the efficiency compared to Proof of Work. The transaction families and the consensus algorithm will help decide the state of the data in the blockchain.

Hyperledger Fabric [45] was initially contributed by IBM and Digital Asset. This framework is a permissioned blockchain. Hyperledger Fabric were designed to be modular and allow pluggable and custom components such as consensus and membership services. The *Membership Service Provider* (MSP) abstracts the cryptographic mechanisms and protocols behind issuing and validating certificates, and authentication for participants within the blockchain network. Only participants who are part of the same MSP are able to communicate. Hyperledger Fabric created the concept of *Chaincode*, which correspond to the smart contracts. The chaincode will be responsible for the business logic and solely this code can read or manipulate the data on the chain. The four main components of this framework are:

1. *Fabric CA* generates certificates for the users;

2. *Peer* stores the blockchain;

3. *Endorsing Peer* receives and simulates the transaction and executes the chaincode;

4. *Ordering Service* guarantees the transactions order, creates the block and sends it to the peer.

Consensus is achieved when the order and results of the blocks' transactions are verified. The ordering mechanisms used are Solo Order and Apache Kafka [48].

In this framework exists the concept of *Channels*. A Channel is a private blockchain overlay which allows for data isolation and confidentiality [49]. Peers can be part of one or more channels. Each peer has its own identity when they join the channel, this identity is given by the MSP and will authenticate each peer to its channel peers and services. Each channel has associated at least one chaincode which is responsible for that channel's ledger.

Besides these frameworks, Hyperledger has tools to make the access to and development of blockchain easier and more effective. Currently, three tools are available:

1. Hyperledger Cello aims to reduce the effort needed to create, manage and terminate the blockchain.

2. Hyperledger Composer is a set of tools that creates blockchain business network, development of smart contracts and the deployment beyond the distributed ledger.

3. Hyperledger Explorer creates a user-friendly Web application to view, invoke, deploy and query blocks, transactions and other information stored in the ledger.

2.5.5 Summary of the studied Platforms

Table 2.2 present a summary of the different platforms studied. All the platform are permissioned blockchains. The second column explains how transactions are processed and last column the consensus mechanism used.

Table 2.2: Platforms

	Transaction Flow	Consensus Mechanism
Quorum	Private transactions propagated in a subset of nodes, while public transactions are network-wide	Raft-based or Istanbul BFT
Chain Core	Bitcoin-like	Federated consensus protocol
Corda	Bitcoin-like	Validity and Uniqueness consensus
Hyperledger Burrow	Transactions are sent to the consensus mechanism, blocks are then propagated in the network.	Tendermint
Hyperledger Iroha	Transactions are proposed to the peers, then is applied the consensus mechanism, and finally the block is sent to the peers.	Yet Another Consensus
Hyperledger Sawtooth	Bitcoin-like	PoET or Dev_mode
Hyperledger Fabric	Transactions are proposed to the endorsing service, then is applied the consensus mechanism, and finally the block is sent to the other peers.	Apache Kafka or Solo Order

2.6 Health Applications

Many authors and companies created applications to apply blockchain as the technology used in healthcare systems. In this section, it will be presented four examples of architectures that used blockchain to create a healthcare system.

2.6.1 MedRec

MedRec [50] tried to solve the following problems of a users' Electronic Medical Records (EMR):

- Fragmented and slow access to medical data;
- System interoperability;
- Patients are not fully informed of their medical history and any modifications to it.
- Improved data quality and quantity for medical research.

The authors chose a public blockchain, specifically the Ethereum blockchain. The authors chose Ethereum blockchain to use smart contracts to contain metadata on record ownership, permission and data integrity. EMRs are stored in external databases, the blockchain only contain pointers to these databases. The blockchain contains a hash of the saved data to ensure the data integrity. When an entity adds data about a patient it sends a notification to the patient and the patient can verify it before accepting or rejecting. He/She will be able to share his/her data with other entities. To verify the participant's entity is used the public key and

a DNS-like mapping between the Ethereum address and the client's social security number or name.

In order to define the relations between the entities and users, three types of contracts were created:

- *Registrar Contract (RC)* is used to map the participant identity with the Ethereum address, which is equivalent to the public key, and have the Summary Contract;
- *Patient-Provider Relationship Contract (PPR)* has the permissions to access data;
- *Summary Contract (SC)* has references to the PPRs, representing all interactions made with the system nodes.

In Figure 2.4 it can be seen that the architecture of this system has four components:

1. *Backend Library* aims to abstract communications with the blockchain;
2. *Ethereum Client* will create peer-to-peer connections on the network, encrypt and send transactions. Ethereum Client also stores a local copy of the blockchain;
3. *Database Gatekeeper* implements an off-chain, it has a pointer to a local SQLite database and it is possible to get access to it depending on the permissions stored in the blockchain;
4. *EMR Manager* is composed by the above components. This component will be used to EMR management and for user interface application. Users will receive notifications when medical data is updated and can share and retrieve data.

The authors of this architecture used Ethereum's consensus protocol, Proof of Work.

Figure 2.4 illustrates how the system will operate if a provider inserts medical records for a new patient.

2.6.2 Oncology specific data management

The framework proposed in [51] was developed to manage oncology data. This platform was created to allow users to define access control policies of their data. Another important feature of this platform is to improve the sharing of this data among the various entities. To create this framework, was chosen a permissioned blockchain for the following reasons:

- Importance of knowing the user's identity;
- Health records are sensitive data, so monitoring the communications between the patient and the entities can expose sensitive data, violating the privacy of the user;

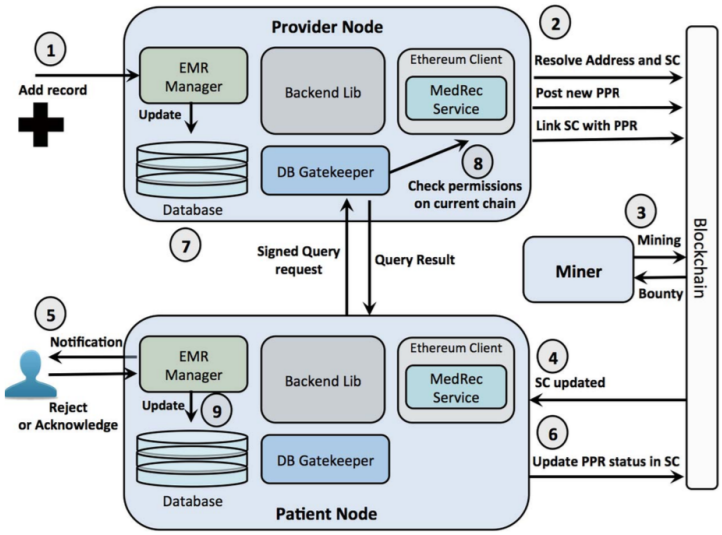

Figure 2.4: MedRec's use case: add record for new patient

- A quick response of the system when doing any procedure is crucial for the user;
- If an entity has to pay to make changes to the user's health records, it could limit the system's usability.

This framework is composed by four components:

1. Membership service records the roles of system actors. The roles define the blockchain functionalities that are available.
2. Databases store health records off-chain.
3. Nodes address the consensus protocol.
4. APIs for different types of users.

User data is stored in two different locations. A local database placed in the hospital and a cloud where data is organized according to sensitivity degree and encrypted with the user's symmetric key. An entity can access data in the cloud depending on the access control policies defined by the client.

To implement the blockchain technology it is used Hyperledger platform. All nodes will be validating peers and will receive all transactions performed by users through role-based APIs. The consensus protocol used was Practical Byzantine Fault Tolerance explained in section 2.2.3.

2.6.3 Healthchain

To design Healthchain [52] it were followed the rules set out in the Health Insurance Portability and Accountability Act of 1996 (HIPAA), used in the USA. HIPPA was used to define the Protected Health Information (PHI). PHI are rules for privacy, individually identifiable health information including demographic and genetic information. HIPAA privacy rule sets standards for keeping the privacy of the individuals' PHI under control and provides patient's rights over the information.

This solution ensures that only authorized people can access, analyze and update an agreed record. The patient will create the initial version of PHI and this will be inserted in the blockchain. Smart contracts provide the guarantee that the user system only inserts an initial version in the blockchain.

Healthchain uses three components in the architecture:

1. Private Blokchain on IBM Blockchain;
2. Deployed on Bluemix, IBM cloud [53];
3. Web page, where the NodeJs server is responsible for the communication with Hyperledger fabric and the chaincode.

Hyperledger Fabric will guarantee data confidentiality, scalability and security. Smart contracts define the privileges of the network nodes. The consensus protocol used by Healthchain is the Practical Byzantine Fault Tolerance, explained in section 2.2.3. Transactions of this system are not considered complete until consensus is achieved. All updates to PHI are only visible to the nodes who have the corresponding permission.

2.6.4 MedicalChain

MedicalChain [3] is a platform that uses blockchain as a technology to safely store and transfer electronic health records. The blockchain stores health records and maintains a single version of users' data. Privacy is ensured with Hyperledger Fabric because this framework implements a private blockchain and allows to define data access control. The platform collects all the medical documents. The physical documents will be scanned, ciphered and uploaded to the cloud. The EHR will be standardized, ciphered and uploaded to the cloud.

To accomplish the goals, the creators of MedicalChain follow these steps:

1. Data is produced by a wearable device, doctor notes, scan or medicine dispensation;

2. Data is encrypted and sent to a cloud storage. The patient blockchain will store an ID of the data stored in the cloud;

3. When data is requested, the ID saved on the blockchain is used to retrieve the encrypted data;

4. The data is decrypted and displayed on the device or application.

Some of MedicalChain's features are:

- User has access to and control over his/her data;

- Provides information about the data, the time and the type of data that has been accessed;

- Insurance companies may have access to data and verify it. This verification will ensure that the treatment of the patient is undergoing is consistent with their expectations.

2.6.5 Summary of the Health Applications

Table 2.3 present a summary of the comparative study between different health applications. To try to solve the problems existing in healthcare services the type of blockchain used by the majority of the applications studied is permissioned. To create this type of blockchain most of the applications chose Hyperledger Fabric.

Table 2.3: Summary of the Health Applications

	Type of Blockchain	Consensus	Data	Platform
MedRec	Permissionless	PoW	Database	Ethereum
Oncology	Permissioned	PBFT	Database and Cloud	Hyperledger Fabric
HealthChain	Permissioned	PBFT	Cloud	Hyperledger Fabric
MedicalChain	Permissioned	—	Cloud	Hyperledger Fabric

MedRec's architecture uses Ethereum to implement the Blockchain, as mentioned in the section 2.4 it uses PoW as consensus algorithm so it will consume many resources. Oncology's architecture was designed to only store Oncological data. Finally, HealthChain's architecture is similar to the architecture from Oncology but it stores all types of medical data but, like MedicalChain, it also stores the date in the Cloud, this way decreasing system's availability.

2.7 Legal Framework

This section will explain some of the legal requirements that will be taken into account to use personal data.

After May 2018 it is mandatory to apply the General Data Protection Regulation (GDPR) [54], defining the rules related to the processing and free movement of personal data.

This regulation, replaced the Directive 95/46/EC, but the conditions defined in the Directive 95/46/EC are considered as valid.

According to this Regulation, personal data cannot be kept in a form which can identify the subject for longer than required for processing. Data can be stored for long periods if it will be used for public interest, scientific or historical research or statistical purposes.

The personal data has to be processed in a way that guarantees proper security.

The data subject has right to request to erase her/his personal data. If any personal data has been rectified or erased, these actions must be communicated. The data subject has the right to access, erase, correct and identify who has accessed their data. To ensure a level of security:

- Pseudonymization and encryption of personal data should be used;
- Processing systems and services should ensure confidentiality, integrity and availability.

Chapter 3

System Model

The goal of this project is to create a secure system to store and share electronic healthcare records (EHRs) using blockchain technology. The healthcare records are sensitive data, so it is important to assure the privacy of the user's identity and data privacy. It is also important to guarantee preservation of the data to be possible to access all the medical history of a patient. In this type of data, it is essential to define access control policies and also be able to identify who has accessed the data. The last requirement is to assure system availability.

3.1 System Components

In this chapter, it will be introduced the necessary components to create an application that uses Blockchain technology for a Healthcare system. The name choosen for this application is *HyperMedical*.

The most important decision of this project was to choose a platform that implements a permissioned blockchain. Taking into account the Blockchain platforms and Health applications studied in section 2 we chose Hyperledger Fabric.

Figure 3.1, demonstrates the main three high-level components

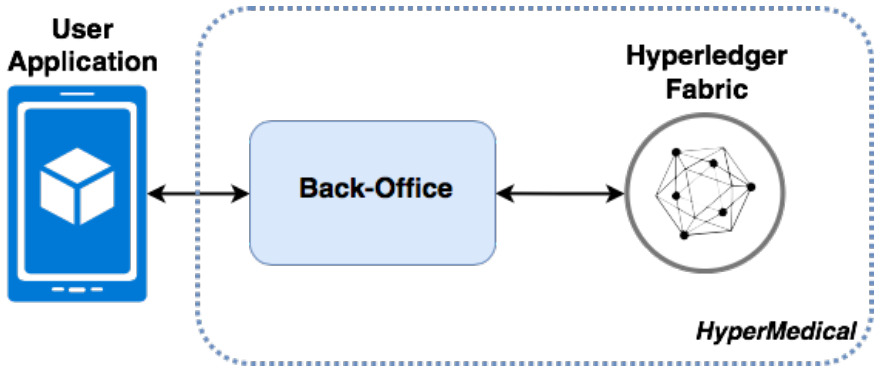

Figure 3.1: HyperMedical High-Level Architecture

The user application is not the main objective of this project but was created as a proof-of-concept of the proposed architecture. The goal was to create an application that could be used by real users to prove that it is possible to create a healthcare system using blockchain technology.

The Back-office receives requests from the user application and sends it to the Hyperledger Fabric network.

According to the functionalities and use-cases were defined specific requirements. In the Table 3.1 were created different types of roles and each type of user has predefined access control policies.

The patient owns the data and is able to modify or delete data that he/she created. Patients may withdraw access permissions at any time. Doctors have access to the same functionalities that a patient has, except the functionality of modifying and deleting a medical record. Health Aides, like nurses, can only view and insert new medical records. Insurance Companies are only able to visualize medical records that were shared with them by the patients. All the healthcare providers that are working in the emergency service can only share data with a healthcare provider if the patient authorizes. The authorization for emergency services to share data with a healthcare provider has to be defined by the patient when he/she registers on the system and it can be changed anytime.

All users also have one more functionality which is the right to oblivion, they are able, at anytime, to delete their account and all the information that was created would not be accessible.

Table 3.1: Roles and Access Controls

	View	Insert	Modify	Share	Delete
Patient	Yes	Yes	Yes[1]	Yes	Yes[1]
Doctor	Yes	Yes	—	Yes[2]	—
Health Aides	Yes	Yes	—	—	—
Insurance Companies	Yes	—	—	—	—
Emergency Services	Yes	Yes	—	Yes[2]	—

[1] Medical record inserted by the patient.
[2] With patient approval and only with healthcare providers.

In order to enable the definition of user types, the Back-office can register and authenticate users. To authenticate a user it is necessary the username and password. The username is used as the unique identifier and the password is generated in the registration phase. For patients, the unique identifier is the patients' number. For doctors, health aids and emergency services the unique identifier is the medical license number. For insurance companies, it will be considered the insurances' code.

The unique identifier in a real world scenario should be verified by a specialized authority.

3.2 Functionality

To accomplish the system goals it were defined the following functionalities:

1. Administrator:
    - User registration;
    - User Login;
    - System monitoring.
2. Patient:
    - Insert / View patient's personal information;
    - Insert/ View/ Share/ Medical Record;
    - Modify patient's personal information;
    - Delete Medical Record created by the patient;
    - Share Medical Records on Emergency State;
    - Withdraw access privileges;
    - Delete account;
    - Notify the user when an operation is executed on his/her Medical Record.
3. Medical Staff:
    - Insert/View Medical Record;
    - Share Medical Record with other medical staff;
4. Emergency Service:
    - Insert/ View Emergency Medical Record;
    - Share Emergency Medical Record with other medical staff;
    - View Patients' personal information;
5. Insurance Companies:
    - View Medical Records.

3.3 Data Model

All the user's information and medical records will be stored in the blockchain. For that purpose, it were created four chaincodes to define different business assets:

1. User information;
2. Patient Medical Record;
3. Healthcare Entity Medical Record;
4. Emergency Medical Record.

The first asset, represented in Table 3.2, defines the *User information*, representing all the personal information about a user. The last two variables represented in Table 3.2 are only required for the Patients' initial info and background diseases. Table 3.3 represents the Patient's personal information. The background variable details the diseases that a Patient had.

The Personal and Background information was decided using the forms [55] [56] [57] from different Insurance companies as a model.

Table 3.2: User Information

Variable Type	Variable	Description
String	Name	User's full name
String	Type	User's type (Patient, Doctor, Health Aid, Insurance Company)
Int	ID	Unique ID, like patient number, medical unique identifier and insurance code
String	Birthdate	User's Birth Date
Into	Contact	Number for contact
Personal	Info	Patient's initial info
Background	Background	Patient's background diseases

Table 3.3: Personal Information

Variable Type	Variable	Description
String	Gender	Patient's gender
String	Height	Patient's height
String	Weight	Patient's weight
String	Blood Pressure	Patient's blood pressure
String	Alcohol	Alcohol consumption
String	Smoke	If the patient is a smoker
String	Blood Type	Patient's blood type

As represented in Table 3.4, it was created a new type of asset that represent the *Patient medical record* that will be stored in the blockchain. To be possible to share a medical record, it

was created an Access Control List (ACL), this is, the list of users with which a specific medical record was shared with.

Table 3.4: Patient Medical Record Information

Variable Type	Variable	Description
Int	Record ID	Unique random generated number
String	Medical ID	ID of the medical staff that is inserting that record
String	Patient ID	Patient's ID
String	Date	Record's insertion date and time
String	Specialty	Medical specialty
String	Local	Where the patient was treated
String	Notes	Medical annotations
String	Document	Extra medical documents encoded in base 64
String	Document Name	Extra medical document's name
AccessControl	Access Control	Access control list with the users' ID which this record was shared with

The *Healthcare Entity medical record* has the same attributes as the patient medical record, except the ACL. The ACL does not exist in the healthcare entity medical record, because this type of data can only be accessed by doctors, health aides, insurance companies or emergency service.

The *Emergency records* contain the same data model as the medical record and the initial information of the patient. In the case of the emergency service, it is important to have an ACL to register the medical staff with which that information was shared.

3.4 Use Cases

To validate the architecture of this system, four specific use cases were defined to exemplify the main functionalities of the project's features:

1. User Registration and authentication;
2. Insert and View EHRs;
3. Share EHRs;
4. Emergency status.

In Figure 3.2 it is defined the *User Register and Authentication* use case. In the user registration phase, all the information demonstrated in Table 3.2, is inserted by the user and HyperMedical will register the user in the HLF network and create a block with the users' initial information. In the end of this process, the password generated by the HLF network will be retrieved. Internally, during this process, it is created the cryptographic material, to be

possible for the user to propose transactions. In the authentication phase, the user will insert his/her unique ID and his/her password which are then verified by HyperMedical. The verification done by HyperMedical will verify if the user is registered in the application and also if his cryptographic material is valid.

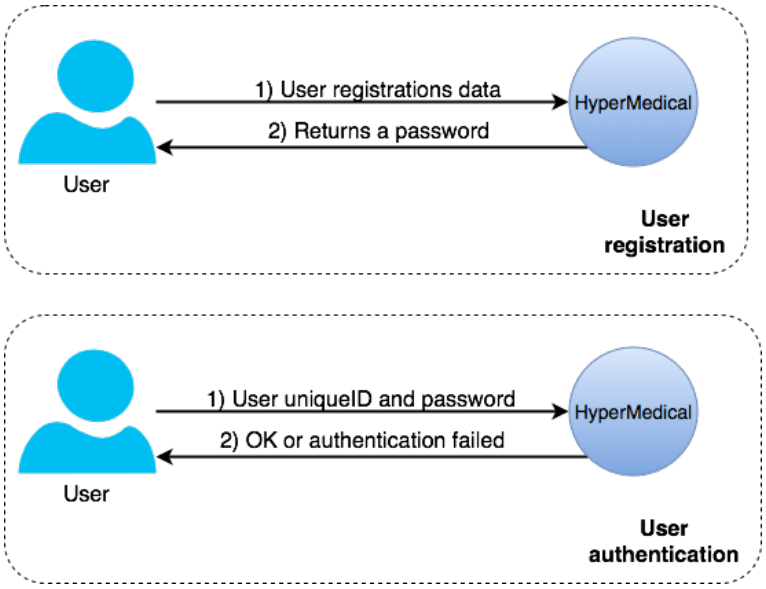

Figure 3.2: User Register and Authentication

In Figure 3.3, it is defined the *Insert and view of medical records* use case. In this use case, the patient will share his unique ID with the Doctor. The Doctor will insert the information demonstrated in Table 3.4, and internally it will be created two different types of data structures. The first transaction will be of the type *Patient Medical Record*, which is only accessible by the Patient. The second type will be a *Healthcare Entity Medical Record* that is only accessible by the Doctor that created it. The Patient Medical Record and the Healthcare Entity Medical Record are ciphered using the Patient and Doctor key, respectively. When the transaction is committed the Patient will receive a notification in order to view his medical records.

The use case, illustrated in Figure 3.4, demonstrates how to *Share medical records*. In this use case the Patient requests to share his medical record with Doctor 2. Internally, the patient medical record is queried, it is added the Doctor 2's ID to the access control list and a new transaction is created using the data structure *healthcare entity medical record* ciphered using the Doctor 2's key. After the transaction is committed, the doctor is able to access it.

The last use case defined is the *Emergency status* functionality, as illustrated in Figure 3.5. When a user registers in the application, he/she will state whether he/she accepts or not to share his/her information with the emergency services. In this case, it is important to share his/her medical information with the medical staff so the treatment is more effective.

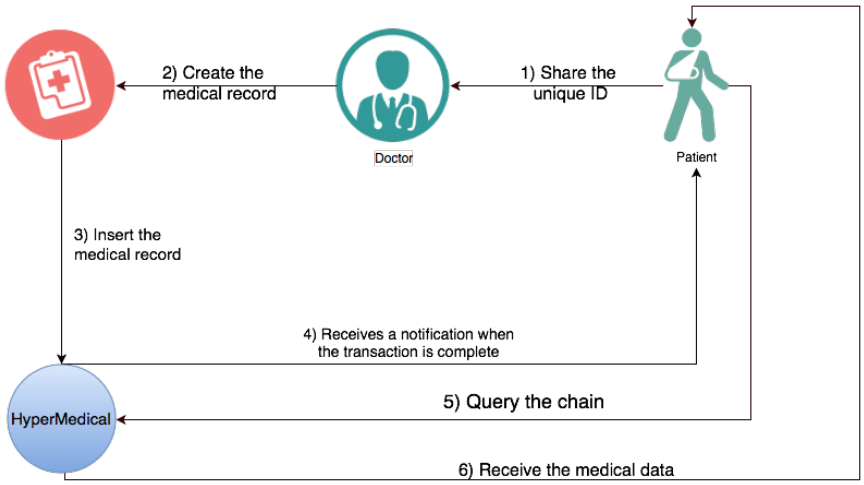

Figure 3.3: Insert and View Medical Record

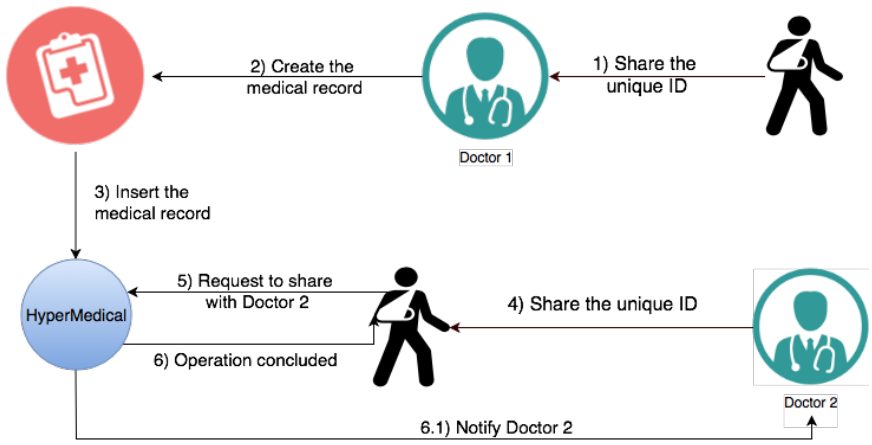

Figure 3.4: Share Medical Record

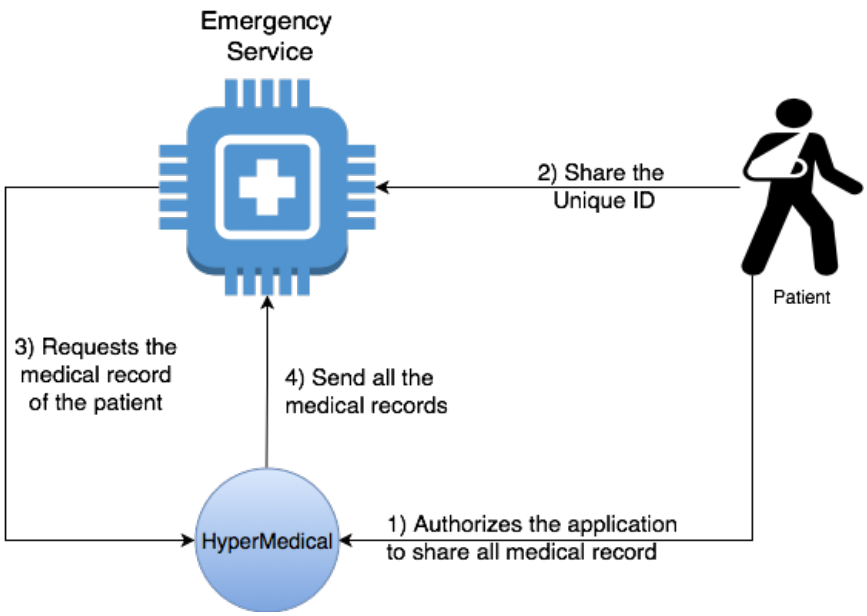

Figure 3.5: Emergency Status

Chapter 4

Implementation

This chapter addresses the implementation details of HyperMedical. Section 4.1 presents an overview of all the components of HyperMedical. In the following sections each component and operation procedures will be explained in greater detail.

4.1 Implementation Overview

HyperMedical's four main technical components are demonstrated in Figure 4.1.

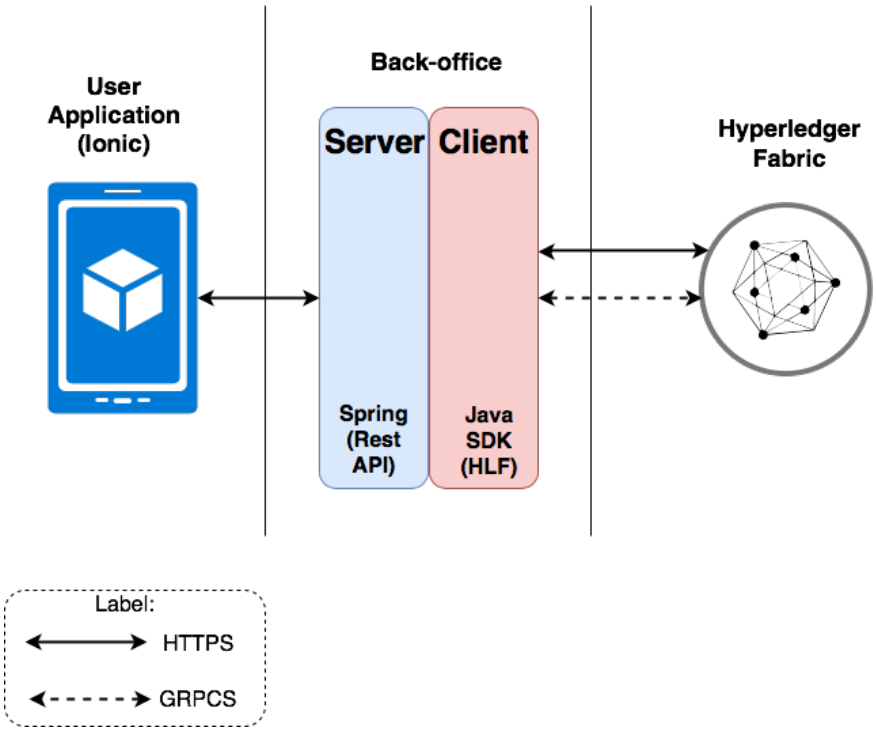

Figure 4.1: HyperMedical Components

The HyperMedical Hyperledger Fabric network is the technical infrastructure that provides ledger services to the created Back-office. The definition of the network components is in YAML

files, which is a data serialization standard [58]. The network components are simulated using Docker containers. The data structure stored in the blockchain and all the business requirements are defined using Go Programming language [59].

The second component is the Back-office which receives requests from the user application and sends it to the Hyperledger Fabric network. This application is divided in two components as demonstrated in Figure 4.1. The REST API, which is invoked by the user application, was implemented using Spring Boot framework [60]. It was also created a user interface using SwaggerUI framework [61]. The second component of the Back-office was created using the Java SDK for Hyperledger Fabric version 1.1 [62]. The Back-office was implemented to communicate with Hyperledger Fabric's network.

HyperMedical's last component is the user application which was created as a proof-of-concept of the proposed architecture. The user application provides all the functionalities to support the defined use cases.

The user application was implemented using Ionic framework [63] which allows the creation of cross-platform applications. This way, it is possible to simulate the defined use-cases in different types of platforms. To create this application Ionic3 framework, Typescript and HTML was used.

4.2 Hyperledger Fabric Network

The Hyperledger Fabric network is composed by five main elements, as demonstrated in the Figure 4.2.

The Certificate Authority used was Hyperledger Fabric CA [64], which issues the PKI-based [65] certificates to network members and their users. All the certificates issued by the Fabric CA are X.509 certificates which use Elliptic Curve Digital Signature Algorithm (ECDSA) to digitally sign the messages.

The Hyperledger Fabric CA is used for [66]:

- Register users;
- Issue Enrollment Certificates (ECerts);
- Renew and revoke certificates.

When a user is registered, the Back-office comunicates with Fabric CA to register the user and retrieve the cryptographic material, including the password used for the login functionality. The Hyperledger Fabric CA uses a database to save users' identities and certificates. The default

database is SQLite. To avoid a single point of failure was defined two Fabric CAs. For that purpose, Fabric CA uses PostgreSQL to store the identity and certificates of users.

It was decided to use PostgreSQL because SQLite only allow one write operation at any given time and does not support managed connections with set access privileges to the database and tables [67].

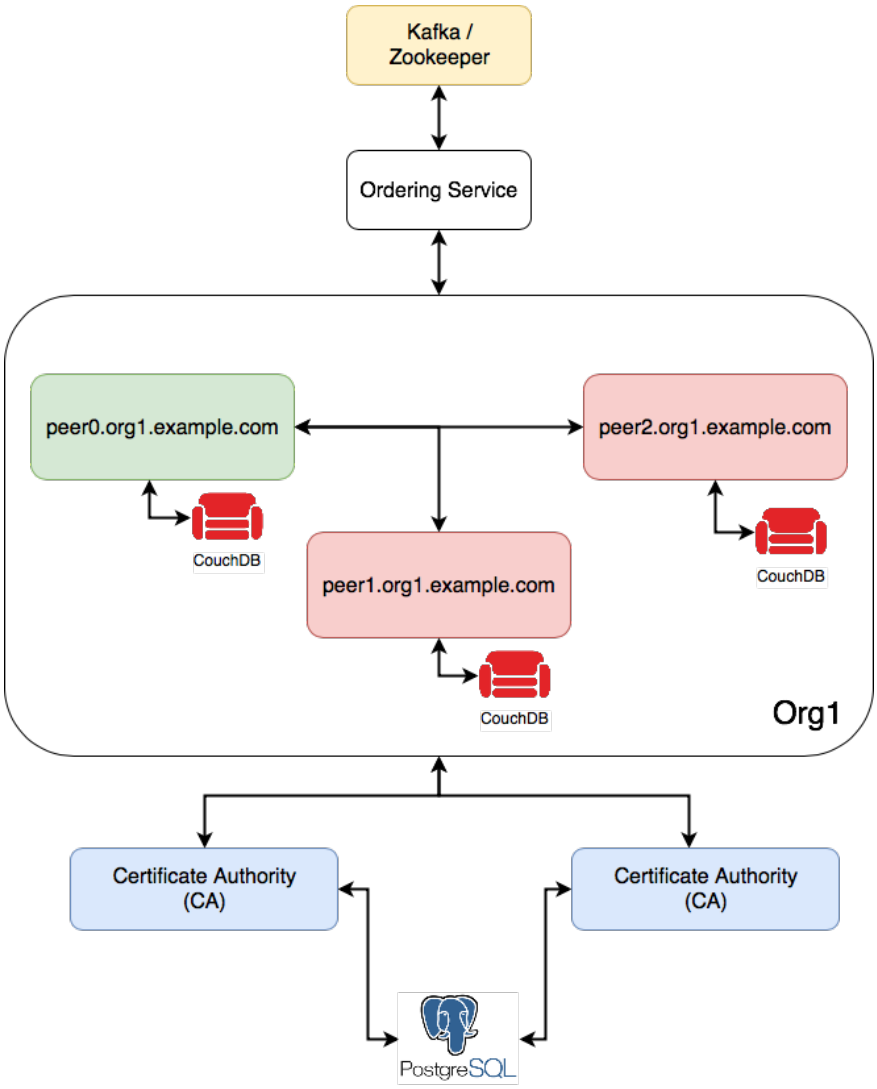

Figure 4.2: Network Components

The communication between PostgreSQL and Fabric CA is done through TLS. The certificate was generated using openssl [68].

Each peer of the network is associated to an organization. In Figure 4.2 is represented only one organization as "*Org1*". Only the peers that are associated to "*Org1*" are considered as trusted peers because they are associated to trusted organizations that are validated by the certificate authority.

Peers are the network entities responsible for maintaining a ledger and run chaincode con-

tainers to perform read/write operations to the ledger, creating the read/write sets. Each peer maintains the state data in a state database improved effieciency in reads and queries from chaincode [49].

The ledger is a blockchain to store the immutable and sequenced records in blocks, as well as a state database to maintain the current state. Each block can contain one or more transactions. Each transaction results in a set of asset key-value pairs that are committed to the ledger as creates, updates or deletes [69].

The ledger's current state data represents the latest values for all keys ever included in the chain transaction log. The state database supported by Hyperledger Fabric are LevelDB and CouchDB. CouchDB [70] was chosen because uses a client-server model, that is accessed through a REST API over secure HTTP and also provides addition queries support when chaincode data is modelled as a JSON [71], permitting rich queries of the JSON content [69]. CouchDB supports the definition of access rights for users per database [72] [73].

The ordering service exists independent of the peer processes and orders transactions on a first-come-first-served basis for all channels on the network [49]. An ordering service atomically broadcasts state updates to peers and establishes consensus on the order of transactions [74]. The ordering services available in Hyperledger Fabric are Solo Orderer and Apache Kafka-based [48].

Solo Orderer [74] is a centralized module, that represents a single point of failure and is not fault tolerant. This way, it was necessary to study a different ordering service.

The Apache Kafka-based ordering service is a distributed streaming platform with publish-subscribe interface, aimed at high throughput and low latency.

The Apache Kafka instance provides the functions of atomic broadcast, which offers scalable publish-subscribe messaging and strong consistency despite node crashes, based on Zookeeper [74].

The ordering service publishes transactions to Kafka topics and leverages the ordered and immutable nature of records in kafka topic to generate a unique ordered sequence of transactions in a block [73]. It will be verified if one of this three conditions to create a block is met:

1. The block contains the specified maximum number of transactions;
2. The block has reached a maximal size (in bytes);
3. An amount of time has elapsed since the first transactions of a new block was received.

Each channel maps to a separate single-partition topic in Kafka. When an ordering service receives a transaction through a Remote procedure call (RPC) broadcast, it checks if the client

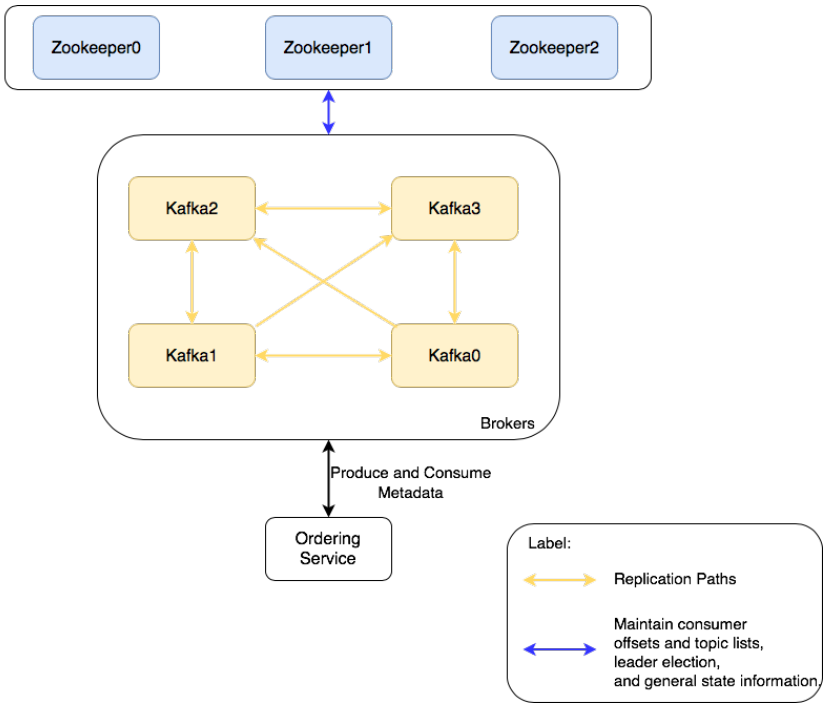

Figure 4.3: Kafka and Zookeeper

has permissions to write in that channel, then publishes the transaction to the appropriate partition in Kafka. This partition is also consumed by the ordering service which will group the received transactions into blocks locally, stores them in its ledger and serves them to receiving clients via the deliver RPC [48].

The orderer is a special node that ensures that every peer's ledger is kept consistent. These nodes establish the total order of all transactions in HLF, as demonstrated in the Figure 4.2 the orderer is like a proxy between the peers and Kafka. An orderer is entirely unaware of the application state and does not participate in the execution nor in the validation of transactions [75].

4.2.1 Network Configuration

In this section, all the network components that are necessary to achieve the goal of creating a permissioned blockchain network are presented. Figure 4.4 demonstrates all the existing entities and the corresponding ports. Each entity portrayed in Figure 4.4 represents a Docker container.

As Figure 4.4 depicts:

- Each peer depends on one CouchDB instance and orderer and it may have a chaincode instanstiated.
- Peer1 of each organization still depends on the endorsing peer.

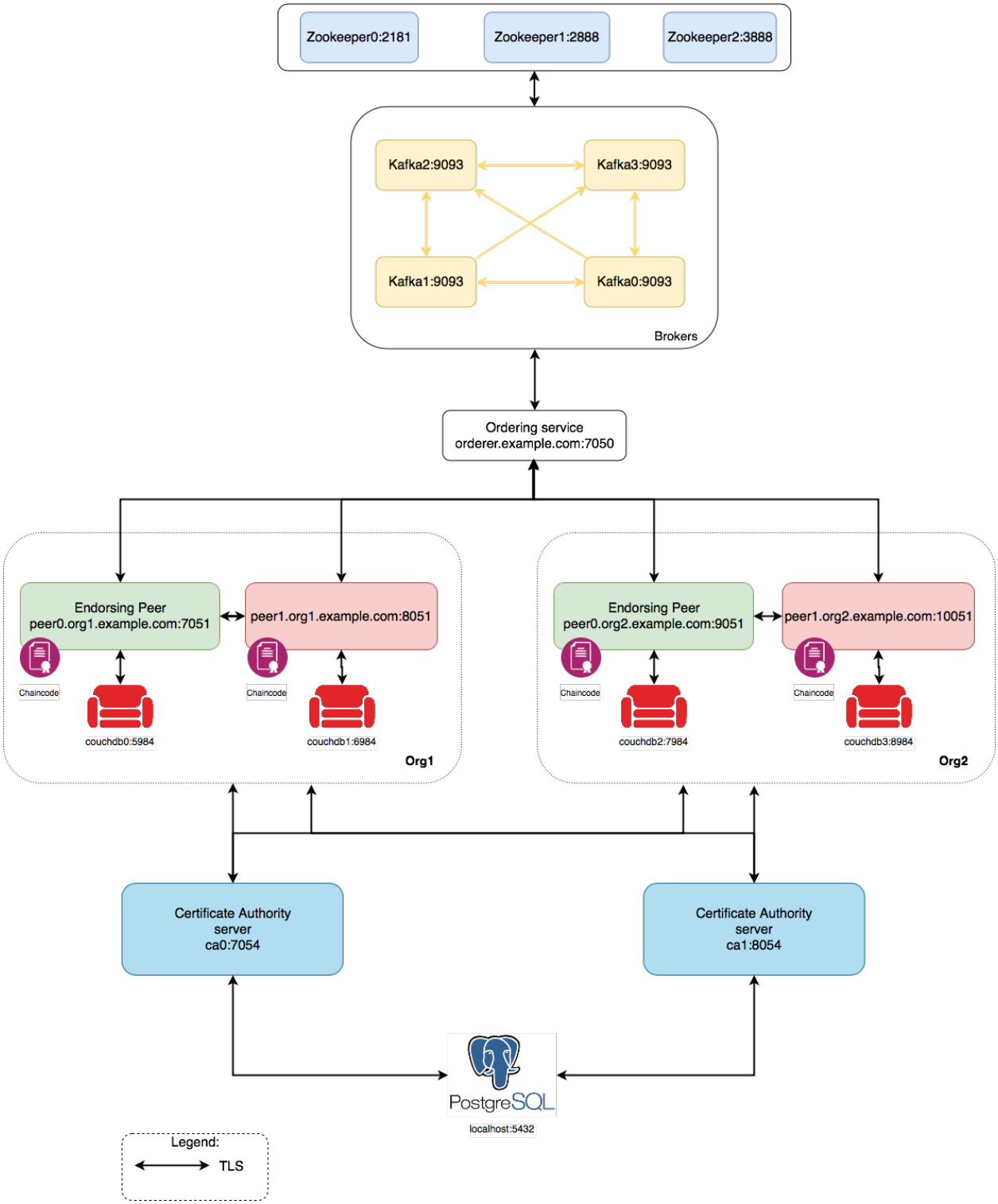

Figure 4.4: Network

- Each Kafka instance depends on all the zookeeper instances, to guarantee the leader election and topic configuration.

The number of Kafka brokers is four, because it is the minimum number of nodes necessary to tolerate crash fault tolerance and failstop failures but does not support arbitrary faults. Three Zookeeper servers were defined because it should be an odd number to avoid split-brain [1] scenarios and larger than 1 in order to avoid a single point of failure [48].

The organizations, represented as "*Org1*" and "*Org2*" are the different medical institutions. The goal of the membership service provider (MSP) is to abstract all the cryptographic mechanisms and protocols behind issuing and validating certificates and user authentication.

For the configuration of the HyperMedical network were created two configuration files to define the ordering service, the organizations, the respective membership service provider's ID, the anchor peers of each organization and the necessary information to generate the cryptographic material for the orderer and all the peers. These configuration files are consumed by two different tools, cryptogen [76] and configtxgen [77].

All the connections between the different HLF components were done using TLS. The certificates used for the connections between the components, shown in Figure 4.4, are generated by the cryptogen tool [76]. For the Kafka ordering service, it was created a script to generate the certificates using the Keytool to create X.509 certificates to add SSL to the communications.

The configuration files allows the creation and inspection of artifacts related to the configuration of channels. These artifacts are:

- Genesis block;
- Channel configuration transaction;
- Configuration update to add an anchor peer on the channel.

4.3 Back-office

4.3.1 Client (HLF Java SDK)

The Back-office was designed to communicate with the HLF network described in the section 4.2 using Java SDK created by Hyperledger Fabric [62].

To accomplish this goal, it was defined a Java network configuration file to set as the default properties used by the client application to communicate with the network entities. The

[1] Two or more Zookepeer servers make progress independently leading to inconsistent behavior.

communications with the Fabric CA are done using HTTPS, and with the peers and orderer are done through gRPC Remote Procedure Calls over TLS (gRPCS) [78].

As said before, the HLF network contains the Fabric CA servers. This way, in the Back-office it was necessary to create a Certificate Authority client to communicate with the Certificate Authority server running in the Docker container.

When a HLF network is created, it is also created an administrator user for each organization and all the cryptographic material necessary. In the Back-office, the administrator user have to be enrolled in each organization [62].

To create a private blockchain overlay, which allows data isolation and confidentiality, a channel is created [49]. A channel-specific ledger is shared across the peers in the channel, and transacting parties must be properly authenticated to a channel in order to interact with it [49].

The final step to have all the components ready to use the application functionalities is the installation and instantiation of the chaincode. The installation of the chaincode is the process of placing a chaincode on a peer's file system [49]. The instantiation is the process of starting and initializing a chaincode application on a specific channel [49]. After the instantiation process is complete, peers are ready to accept chaincode invocations.

There are two types of invocations that can be done on chaincode [79]:

- Transaction Proposal Request (TransactionProposalRequest) that will send the transaction to all the peers that have that chaincode installed and peers will execute the chaincode, retrieve the response and then that transaction will be sent to the orderer;

- Query by Chaincode Request (QueryByChaincodeRequest) will send the transaction to all the peers for value querying and the peers after simulating the chaincode will retrieve the response to the client.

4.3.2 Server (Spring REST API)

In the Back-office, it was also created a REST API using Spring Boot Framework [60] and it was also used Swagger UI [61] to create a visual and interactive API, represented in Figure 4.5. To differentiate the APIs for the different types of functionalities, three controllers were created. Each controller is explained in greater detail in the Appendix A.

The *AdminController* is used for personal information and administration purposes. Table 4.1 details this controller's functionalities.

The *EmergencyController* that is used only by healthcare providers that are working in the emergency service. Table 4.2 details each functionality that exists in this controller.

swagger default (/v2/api-docs) Explore

Api Documentation

Api Documentation

Apache 2.0

admin-controller : Admin Controller — Show/Hide | List Operations | Expand Operations

POST	/rest/background/{id}/{disease}	background
GET	/rest/chainqueries	chainqueries
POST	/rest/createUser/{name}/{type}/{id}/{birthDate}/{contact}	createUser
DELE	/rest/deleteUser/{id}/{password}	deleteUser
GET	/rest/getPlayerID/{id}	sendPlayerID
GET	/rest/getUserInfo/{id}	getUser
POST	/rest/initialInfo/{id}/{gender}/{height}/{weight}/{bloodPressure}/{alcohol}/{smoke}/{bloodType}	initialInfo
GET	/rest/login/{id}/{pass}	login
POST	/rest/sendPlayerID/{id}/{playerID}	sendPlayerID
POST	/rest/shareInitialInfo/{id}	shareInitialInfo
PUT	/rest/updateInitialInfo/{id}	updateInitialInfo

emergency-controller : Emergency Controller — Show/Hide | List Operations | Expand Operations

POST	/rest/initEmergencyRecord/{healthID}/{patientID}/{local}/{speciality}/{notes}/{file}	initEmergencyRecord
GET	/rest/queryEmergencyPatientID/{healthID}/{patientID}	queryEmergencyPatientID
GET	/rest/queryEmergencyPatientInitialInfo/{healthID}/{patientID}	queryEmergencyPatientInitialInfo
GET	/rest/readEmergencyMedicalRecord/{doctorID}/{recordID}/{patientID}	readEmergencyMedicalRecord
POST	/rest/shareEmergencyRecord/{recordID}/{doctorID}/{patientID}	shareEmergencyRecord

medical-record-controller : Medical Record Controller — Show/Hide | List Operations | Expand Operations

DELE	/rest/deleteRecord/{recordID}/{patientID}/{password}	deleteRecord
DELE	/rest/deleteShareRecord/{recordID}/{doctorID}/{patientID}	deleteShareRecord
POST	/rest/initMedicalRecord/{doctorID}/{patientID}/{local}/{speciality}/{notes}/{file}	initMedicalRecord
GET	/rest/queryRecordsByDoctorID/{doctorID}	queryRecordsByDoctorID
GET	/rest/queryRecordsByPatientID/{patientID}	queryRecordsByPatientID
GET	/rest/readMedicalRecord/{recordID}/{patientID}	readMedicalRecord
POST	/rest/shareRecord/{recordID}/{doctorID}/{patientID}	shareRecord
POST	/rest/shareRecordEmergency/{recordID}/{patientID}	shareRecordEmergency

Figure 4.5: Swagger UI

Table 4.1: Administrator Controller

Administrator Controller	
Create User	Register the user in the HLF network generating a certificate, private key and the password through Fabric CA. Creates the first transaction of this user.
Inser Initial Informations	Insert the initial information of the user created. This function is only called by Patients.
Update Initial Information	Updates the initial information. This function is only called by Patients.
Insert Background Information	Insert the diseases that the user has or had. This function is only called by Patients.
Login	Verifies if the user exists and if the cryptographic material is valid.
Get User Information	Retrieves the User information represented in the Table 3.2.
Share Initial Information	Share initial information with emergency service.
Delete User	Delete all the information about a user that exists in the system.

Table 4.2: Emergency Record Controller

Emergency Controller	
Init Emergency Medical Record	Create an emergency record to be accessed by the healthcare providers on emergency service and a medical record to be accessed by the Patient. The emergency record is ciphered using the AES key generated in the chaincode.
Read Emergency Medical Record	Read the medical record if it is a Patient or an emergency record if it is a Healthcare provider.
Query Emergency Medical Record by PatientID	Get all the emergency records of a Patient.
Query Patient Initial Information	Get the patient initial information
Share Emergency Medical Record	Share emergency medical record with a healthcare provider.

MedicalRecordController is used by all the other users of the system for the features related to the medical record. Table 4.3 details each functionality that exists in this controller.

Table 4.3: Medical Record Controller

Medical Record Controller	
Init Medical Record	Create a Patient medical record (Table 3.4) and a Healthcare entity medical record. Each record is ciphered using the AES key of the user that should access the record.
Read Medical Record	Read the Patient medical record or a Healthcare entity medical record.
Query Medical Record by PatientID	Get all the medical records of a Patient.
Delete Medical Record Information	Delete the medical record. This function only can be called by a Patient and will only delete medical records created by himself.
Share Medical Record	Share medical record with a Healthcare provider.
Delete Share Medical Record	Delete the access permission to a Healthcare provider.
Share Medical Record to Emergency Service	Share medical record with emergency service.
Query Medical Record by HealthcareID	Get all the Healthcare entity records of a Healthcare provider.

The Back-office provides data preservation guarantees. All the data needed to reconstruct the application is serialized and saved in a MongoDB [80] database. The necessary information to reconstruct the application, is the information about the channel and users that are part of the system.

The database chosen to guarantee the persistency of the application was MongoDB because it is recommended by Fabric Java SDK's developers to choose a NoSQL database.

The communication between the Back-office and MongoDB is done using SSL. In order to guarantee a secure communication with the REST API, which was created using Spring Boot framework, it was created a self-signed certificate to assure HTTPS connections.

4.4 Chaincode

As described in Section 2, the chaincode is a program to encode assets and the transaction's instructions, this is, the business logic. Each chaincode runs in a secured Docker container isolated from the endorsing peer [81].

All the assets defined are described in the section 3.3. All the requests that are done to the chaincode are explained in the Section 4.3.2.

In the chaincode, it is also performed a validation about the type of user that is performing the request. This validation is done using the attribute inserted in the certificate of the user.

It was used the Client Identity Chaincode Library [82] which enables access control decisions based on the identity of the chaincode's invoker. As previously explained, the access control decisions are made using attributes.

All the stored information is ciphered in the chaincode using an AES 256-bit key. The key is generated with Java code and is stored with the user information in the MongoDB database, each user's has its own key and IV.

The chaincode uses a stream encryption with CBC-mode to encrypt deterministically. The IV should be sent as a transient parameter because the chaincode invocation needs to be endorsed by multiple peers. If the IV is not the same it would cause the endorsement of conflicting read/write sets [83].

The Key is also sent as transient fields to guarantee that the key is not stored in the ledger [83], the chaincode decrypts the message and puts it in the proposal response. An invocation would persist the result in the ledger for all channel readers to see whereas a query can be discarded and so the result remains confidential [83].

4.5 User application

The last component of this project, is an application that can be used by all types of users of this application. The application was developed using Ionic Framework to allow the creation of a cross-platform mobile application.

It was decided to implement the application using a framework capable of creating a cross-platform application because, in a real environment the medical staff will use this application in a web environment. Patients, doctor and insurance companies will, possibly, use mobile devices. This framework supports the emulation of the application for Web, iOS, Android and Windows Phone.

All users that want to use this application must register and the HLF will retrieve a password. After the registration, users have to use their unique ID and password to login in the application. It was defined that only patients are required to fill the biometric and background information.

The unique ID and the user type is stored in the Storage [84] functionality existing in the Ionic Framework. The Storage is an easy way to store key/value pairs. The storage engine will be chosen depending on the platform.

It was decided to store these two variables to smooth the usage of the application. As demonstrated, the REST API always receives the unique ID, this way, it was decided to store

this value, so the user does not have to insert it every time that he wants to do an action. The user type was stored so only users with permission to use certain functionalities will be able to do so.

In the application, it was not considered the security and design issues that should be taken into account when designing a cross-platform application, since it is out of scope of this project.

4.5.1 HyperMedical Transaction Flow

HyperMedical has many entities involved in the processing of a transaction, as illustrated in the Figure 4.6.

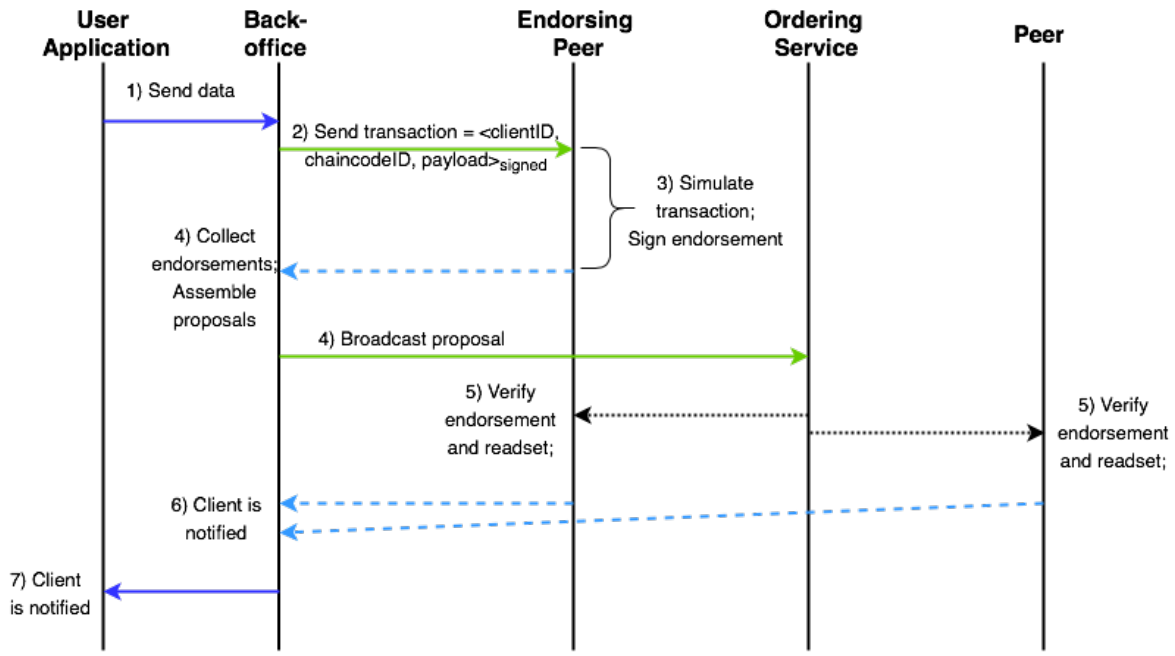

Figure 4.6: HyperMedical Transaction Flow

Figure 4.6 illustrates how a transactions is processed in HyperMedical [75] [31].

1. User fills the necessary information presented in the mobile application;

2. This informations is sent through a REST request to the Back-office;

3. The Back-office will create the transaction that will contain the User ID, Chaincode ID and the information received through the mobile application. The transaction will be digitally signed using the User certificate that was generated by the Hyperledger Fabric CA when the User registered in the application. Then, the transactions will be sent to the endorsing peers;

4. The endorsing peers simulate the transaction and produce an endorsement signature. They verify if the client is properly authorized to perform the transactions by evaluating the

access control policies of a chaincode. The transaction is then executed against the current state. Endorsing peers will transmit to the client the result of this execution (the read and write set associated with the current state) alongside with the endorsing peer's signature. In this point the ledger is not updated.

5. The Back-office will collect and assemble endorsements into a transaction, verifying whether the endorsing peers signatures are valid or not, and if the responses have matching read/write set and the endorsement policies has been fulfilled. If all the conditions are met, the transaction proposal is created, which will contain a signed envelope with the endorsing peers' read/write set, signatures and Channel ID.

6. After the transaction proposal is created, it is broadcast to the ordering service. The ordering service does not read the envelope's contents, it only gathers envelopes from all channels in the network, orders them using atomic broadcast and finally creates signed chain blocks containing these envelopes.

7. The blocks, after being created, are sent to all the peers, including endorsing peers. When the block is received by the peers, it is verified again whether the endorsing policies were fulfilled and it is checked if the read set were not changed since the transaction execution. If this validations are met, the transaction proposal contained in envelopes is marked as valid unless is marked as invalid.

8. Peers will append the received block to the channel's blockchain. An event is triggered to notify the Back-office that the transaction has been immutably appended to the channel's blockchain.

9. Finally, the User application is alerted that a new transaction was committed.

Chapter 5

Results

This chapter addresses the results of HyperMedical's performance, fault tolerance and functionality evaluation. Section 5.1 presents the testing environment used to test HyperMedical. Section 5.2 addresses the perfomance tests regarding different configuration and supported operations translated to throughput and latency values. Section 5.3 focuses on the impact in performance of tolerated faults by the adopted ordering service. Section 5.4 describes the completed tests related to the supported system's functionalities and also shows the performance benchmarks for user functionalities.

5.1 Testing environment and Evaluation criteria

For HyperMedical's evaluation it were considered two testing environments:

- The first is a dedicated server with a CPU Intel Xeon D-1520 – 4/8t – 2.2 GHz and 128GB DDR4 of memory running Debian 9. This dedicated server (HLF Server) was used to test different configurations of the Hyperledger Fabric network, emulating a higher number of entities in the network.

- The second environment is a Macintosh (Back-office Server) with 2,6 GHz Intel Core i5 and 8 GB 1600 MHz DDR3 of memory. The Back-office Server was used to deploy the Back-office and also to test HyperMedical's functionalities.

For the evaluation of this project, 3 types of tests were made:

1. *Performance tests* has as main goal find the optimal network configuration. In these tests the main objective was to test the variations of block size in bytes and number of transactions per block that would maximize the throughput and minimize the latency. The variations done in this test are related to the way blocks are constructed in Hyperledger

Fabric. Next, it was tested the impact of the variations of the number of peers and orderers. In this section, it was also tested the overhead of ciphering the data. The values of throughput and latency were extracted from proposing four types of transactions, a write, a read, an update and a delete operations. The user functionalities are also compared in terms of performance with running the Hyperledger Fabric network in the HLF Server and in the Back-office Server.

2. *Fault tolerance* tests relate to the impact in throughput and latency in increasing the number of Kafka brokers and, consequently, increasing the number of faults tolerated by the system.

3. *Functional tests* focus on testing every provided functionality, presented in the Chapter 3.

In all the executed tests, in which the network configuration is omitted, the default network configuration is 2 peers, where one of the peers is the endorsing peer, one ordering service, 4 kafka brokers and 3 zookeeper servers.

For the Hyperledger Fabric Network benchmark and performance tests were used Hyperledger Caliper[1] that is a blockchain benchmark tool that allows users to get performance indicators like Transactions per Second and Latency. This platform is in development so there are still some limitations, one of the main limitation is that this tool does not support more than one ordering service node in the network configuration.

5.2 Performance Tests

5.2.1 Hyperledger Fabric Network benchmark

The main goal of this section is to find the optimal network configuration. According to [75] the block size is a configuration parameter that impacts the throughput and the latency of the HLF network.

Initially, it was tested the throughput and latency for the default implementation which corresponds to 10 transactions per block and 512 KB of maximum block size. To assign the block size for the following tests and to evaluate the block size's impact in HyperMedical, it were tested different values, as illustrated in Figure 5.1 . The block sizes tested were 0.5, 1, 2, 4, 6 and 8MB.

As both graphics demonstrate (Figure 5.1), latency and throughput do not improve after 6MB, instead it worsens. Therefore, it was adopted 6MB as the block size for the following experiments to maximize the throughput.

[1] https://www.hyperledger.org/projects/caliper

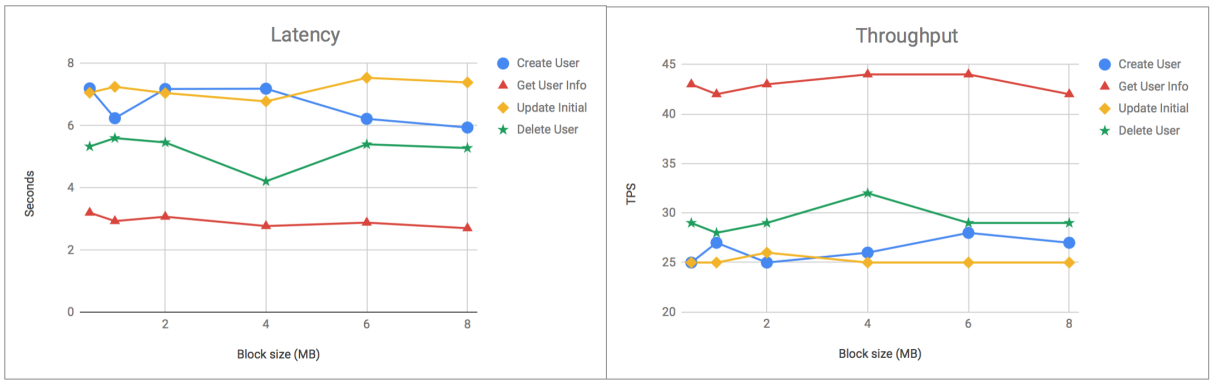

Figure 5.1: Impact of the block size

After assigning the block size, it was evaluated the impact that the number of messages has on the throughput and latency. As both graphics represented in Figure 5.2 demonstrate, the value which minimizes latency and maximizes throughput is 50 transactions per block, for a block with a preferred size of 6MB.

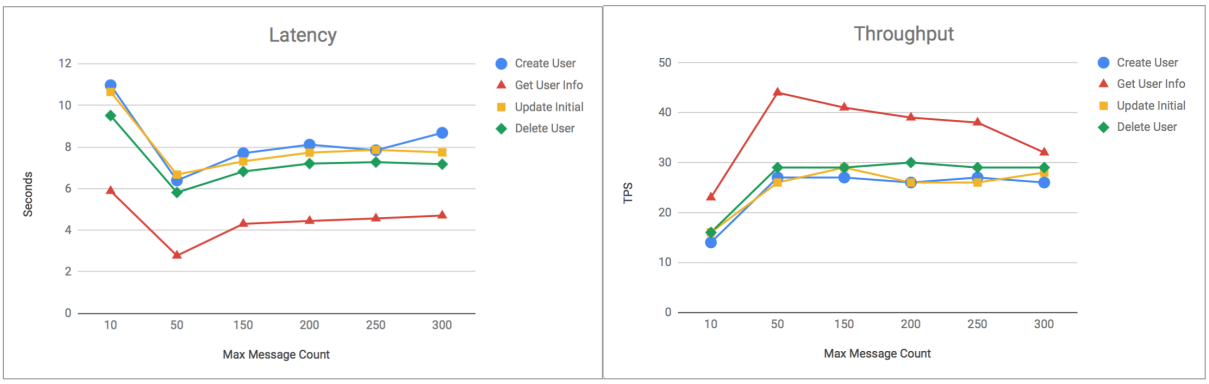

Figure 5.2: Impact of the number of messages on the block

After discovering the ideal block size and the number of transactions per block it was tested to send 1 transaction to define which should be values to be considered as baseline for the next tests as acceptable latency for the following operations are:

- Create User (write operation): 2,78s;
- Get User (read operation): 2,69s;
- Update Initial Information of a User (update operation): 2,73s;
- Delete User (delete operation): 2,79s;

In [75], the authors performed tests, in order to discover the block size that would maximize the throughput and minimize the latency. In these tests, it was obtained the best results for 2MB block size, performing transactions with 0.5s of latency. These results show an observable

difference to the obtained results for this dissertation. This difference can be justified by the fact that in the referenced article the network entities were running in different Virtual Machines and the used chaincode performs less complex operations, this way, taking less time to execute.

5.2.2 Impact of peers and ordering service nodes

After finding the optimal value for the blocksize and the number of messages on a block was tested the impact of increasing the HLF network size in the throughput and latency. Figure 5.3 and 5.4, represents the impact in throughput and latency when the number of peers increase. The network was deployed in the HLF Server and it was executed Hyperledger Caliper from the Back-office Server to simulate the communication latency.

The HLF Server is the reason why the throughput is decreasing, because this is the unique computing intance. Using htop[2], which is a real-time resource consumption monitoring tool, it was concluded that when the number of containers is bigger than 15 the CPU is always around 100%. Using the HLF Server, where all the components of the Hyperledger Fabric network are running in the same computing instance, independently of the provided resources is a considerable degradation factor. In this case, all the components are running in a single machine, the communication between the peers and the nodes involved in the ordering service will be faster.

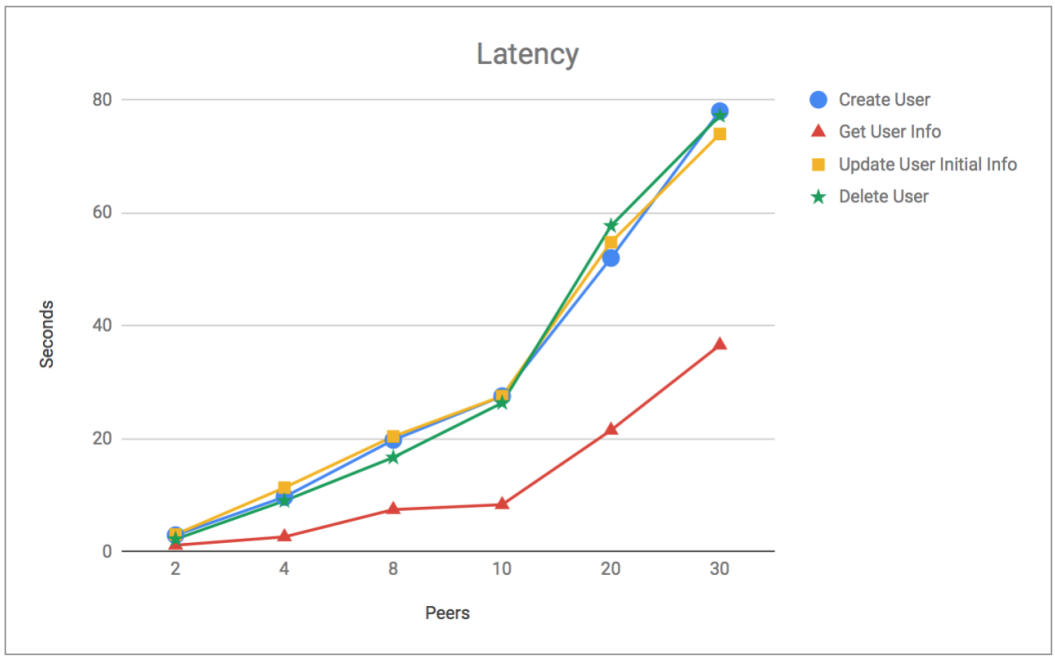

Figure 5.3: Latency Peers variation impact

Due to the limitations of Hyperledger Caliper, it was developed a Java application to calculate

[2]https://hisham.hm/htop/

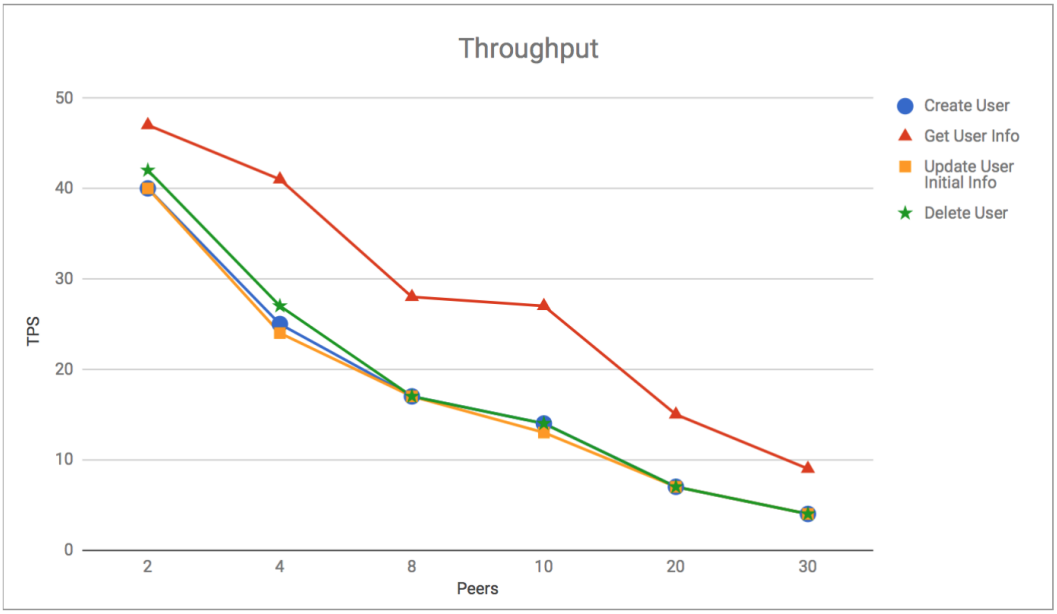

Figure 5.4: Throughput Peers variation impact

the throughput and latency when increasing the number of orderers in the network when using the create user functionality. After testing with a different number of peers (p) and orderers, it was concluded that the throughput is maximized if the number of orderers (n) is n = p + 1. Figure 5.5 and 5.6, illustrates the throughput and latency impact when varying the number of orderers in comparison when the same number of peers only has one orderer associated. This test has also proven that the ordering service is not the blottleneck of performance in the network.

This way, all the other tests performed using Hyperledger Caliper will not be impacted by the use of only one orderer.

In [73], the authors studied the performance of Hyperledger Fabric V1.0 and identified potential performance bottlenecks, with the purpose of understanding better the system and promote possible optimizations. The three main performance bottlenecks that the authors found are the cryptographic operations, the serial validation of transactions in a block and multiple REST API calls to CouchDB [73]. The same authors also found a significant impact when the endorsement process executes different endorsement policies, different number of channels, as well as a impact on the transaction arrival rates and block sizes.

5.2.3 Encryption Impact

In this section is tested the impact of ciphering the data and as expected the throughput is higher without ciphering the data. As demonstrated by Figure 5.7 the *Create User* operation and *Get User Information* operation the throughput is 11% higher when the data is stored without ciphering. As Figure 5.8 shows in *Update User* the operation the throughput 12%

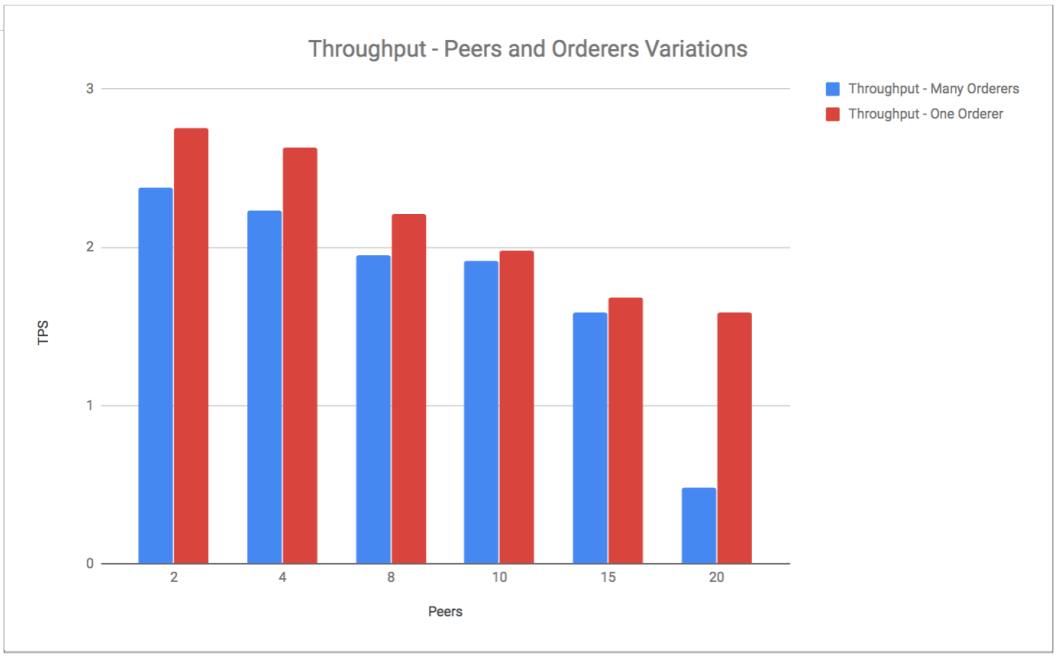

Figure 5.5: Throughput Peers and orderers variation

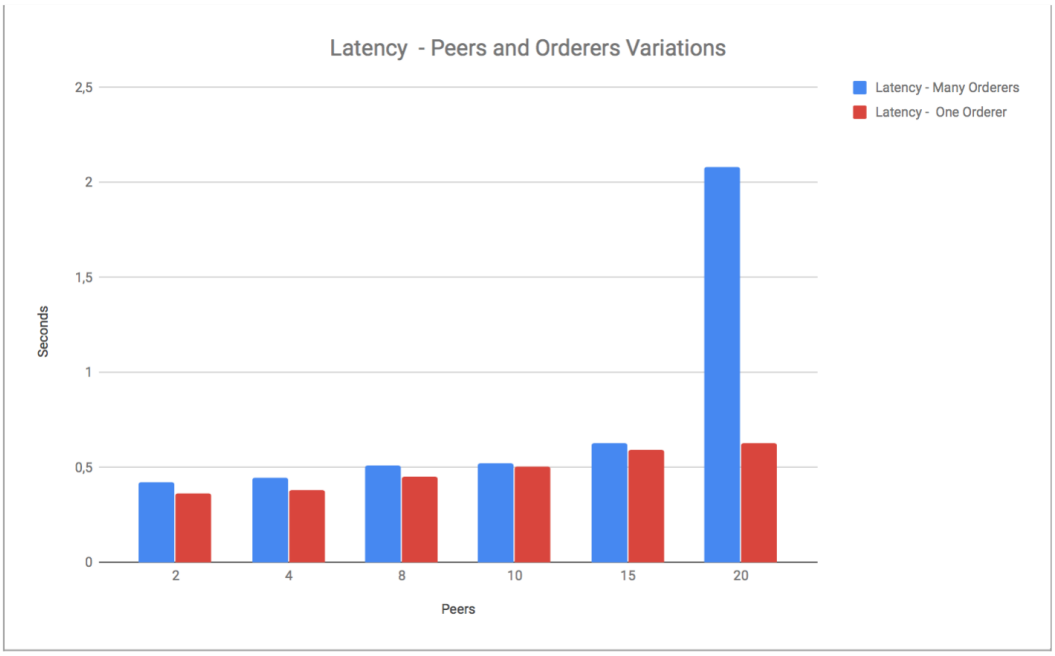

Figure 5.6: Latency Peers and orderers variation

higher and for *Delete User* is 10% higher.

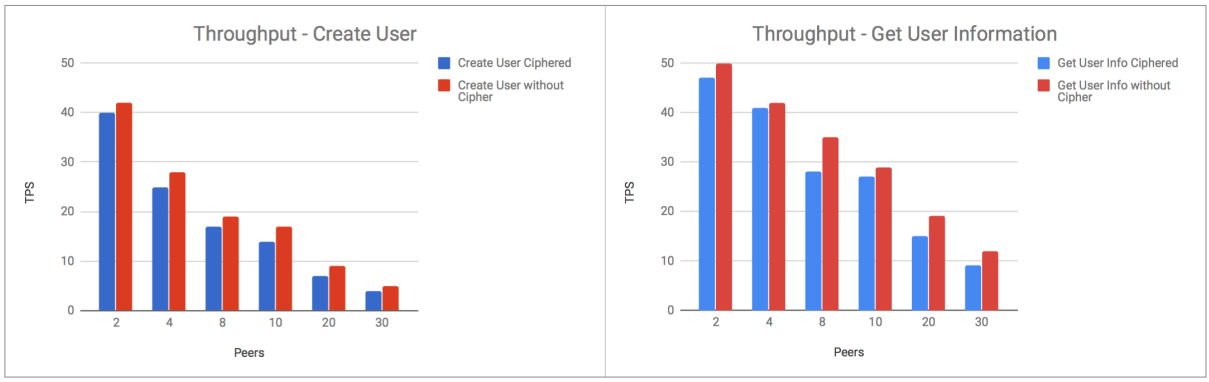

Figure 5.7: Create and Get User

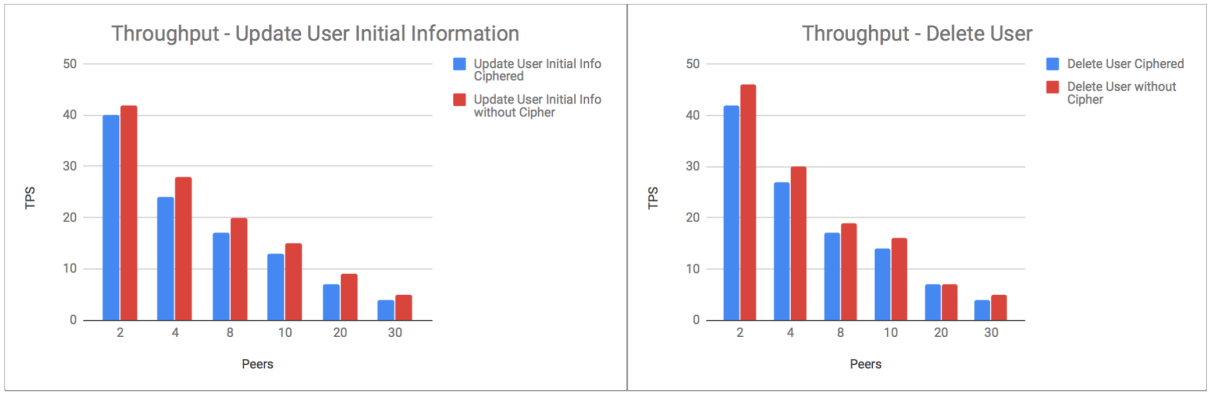

Figure 5.8: Update User Initial Information and Delete User

5.3 Fault Tolerance Tests

In this section it is shown the results of the performance impact when the system is exposed to failures in the beggining of the execution. For these tests, it was varied the number of Kafka brokers for supporting an increasing number of broker failures. The configuration used was 4, 7, 10 and 13 Kafka brokers, where 1, 2, 3 and 4 faults are tolerated, respectively. As illustrated in Figure 5.9 the impact in latency when we use a variable number of brokers in a failure free and fail stop environment is higher in a failure free envrionment. This is given to the fact that a less number of follower brokers have to be in-sync with the leader broker, consequentially, causing less internal messages to be exchanged, during the consensus processing functions.

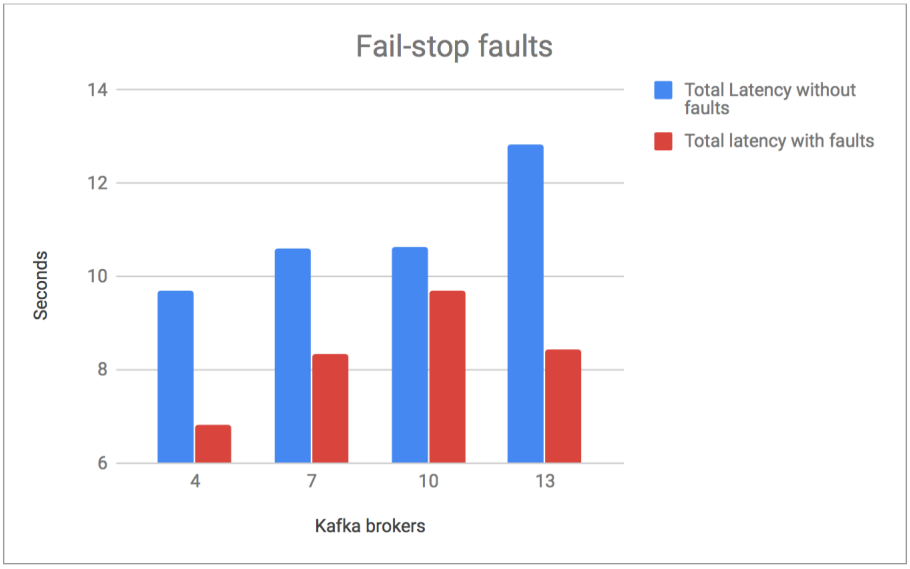

Figure 5.9: Fail-stop comparison

5.4 Functional Tests

To validate HyperMedical's requirements, it were defined nine use-cases to simulate all system's functionalities. These tests were executed using JUnit. Before the execution of JUnit tests, it were created four Doctors and one Patient. The following list details each use-case defined.

US1 - Create medical record:

- Doctor 1 creates a medical record of the Patient.
- Doctor 1 and the Patient can visualize the medical record.

US2 - Share medical record:

- Doctor 1 creates a medical record of the Patient.
- Patient shares with Doctor 2 the medical record created by Doctor 1.
- Doctor 2 can visualize the medical record.

US3 - Share Initial information with emergency service:

- Patient shares the initial information with the emergency service.
- Doctor 3 (working in the emergency service) can visualize the initial information related to the Patient.

US4 - Share medical record with Emergency service:

- Doctor 2 creates a medical record of the Patient.

- Patient share the medical record with the emergency service.
- Doctor 3 (working in the emergency service) can visualize the medical record.

US5 - Patient delete medical record:

- Patient creates a medical record.
- Patient delete the medical record created by him.

US6 - Patient try to delete medical record:

- Doctor 2 creates a medical record of the Patient.
- Patient try to delete EMR without sucess, because it is a not authorized operation.

US7 - Withdraw access privileges:

- Doctor 2 creates a medical record of the Patient.
- Patient share the medical record with Doctor 5.
- Doctor 5 can visualize the medical record.
- Patient delete the Doctor 5 ID from the access list.
- Doctor 5 cannot visualize the medical record.

US8 - Delete Patient Account:

- Register Patient 2.
- Doctor 2 creates medical record of the Patient 2.
- Delete Patient 2 account.
- Doctor 2 cannot visualize the medical record.

US9 - Create emergency medical record:

- Doctor 4 creates an emergency medical record of the Patient.
- Patient, Doctor 3 and 4 can visualize the emergency medical record.

It were compared the latency values recorded that a user takes to finish when the network is running in the Back-office Server, JUnit tests were also executed in the computer with the network running on the HLF Server.

As Figure 5.10 illustrates, the total time to execute each use case is lower when the HLF network is running in the dedicated server. This way, it is possible to conclude that by running the HLF network in the HLF Server it is possible to obtain a decrease in latency of about 8% when compared to running the network in a single machine.

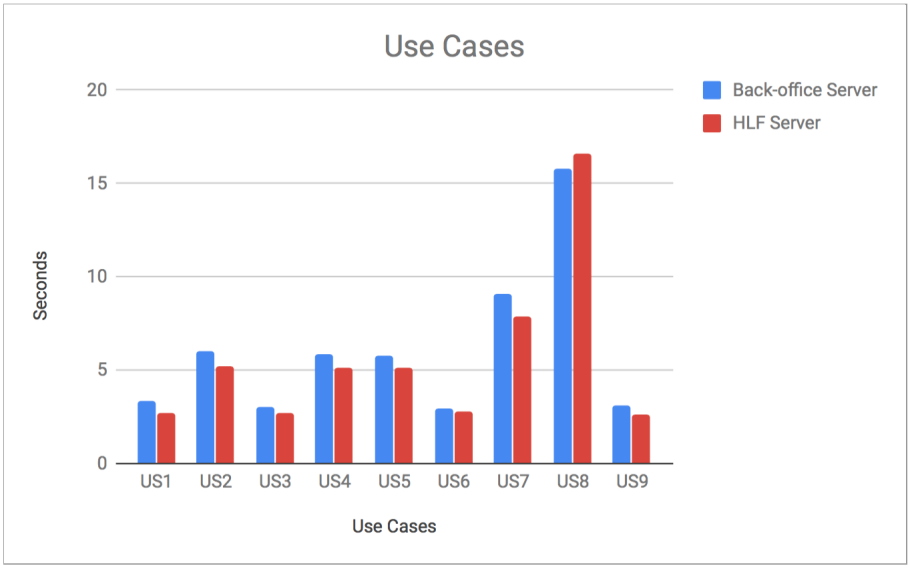

Figure 5.10: Back-office Server vs HLF Server

5.5 Summary

In this chapter it was addressed the results of HyperMedical's performance, fault tolerance and functionality evaluation. In the Performance tests, it was evaluated the Hyperledger Fabric network conditions to create a block, through the study of the throughput and latency results obtained by running benchmarking operations in Hyperledger Fabric. It was also studied the impact of increasing the number of entities in the network and also the overhead of ciphering the data. In the Fault Tolerance tests was tested the impact of fail-stop faults in HyperMedical. Finally, all the functionalities were tested using JUnit and it was also compared the difference between running the Hyperledger Fabric network in the HLF Server and in the Back-office Server.

Chapter 6

Conclusions

In this chapter, it is described the main findings from this dissertation, as well as the existing open issues and introducing some future work ideas.

6.1 Achievements

Information is considered one of the most valuable assets and all this information is fragmented through different entities. This volume of data, arise several concerns as the service availability and storage security that can affect user's privacy.

Nowadays, the Electronic Health Records (EHRs) are fragmented through different healthcare services. The problem in the healthcare services is the fragmentation of health records [3]. Electronic Medical Records are fragmented across hospitals, health centers, doctors, insurance companies and wearable devices. The fact that patients can't precisely describe what was said by one doctor and the lack of information sharing mechanisms between medical entities, can lead to inaccurate diagnosis in different consultations. To increase the probability of an accurate diagnosis and more effective treatments, it is important to have a system that allows all the participants to view, insert, modify, share and delete EHRs.

The main goal of this project was to enable the electronic health records created by different healthcare providers to be stored so that the user can access his data and provide it to the different healthcare providers at any time. It is allowed to store data to be viewed, inserted, altered, shared or deleted by certain entities depending on defined access policies.

The blockchain technology solves the data fragmentation problem and also helps to preserve all the data collected about the patient in such way that the information stored in the chain cannot be altered nor deleted, but can be modified from a user's perspective by adding new information on the chain. The blockchain provides an efficient platform to exchange health

records between entities without compromising data privacy and respecting owner's privacy.

In the Related Work chapter of this dissertation, Blockchain technology and its properties were studied. It were also studied different consensus mechanism as well as different types of blockchain. Given the defined acess control requirements, it was concluded that a consortium blockchain would be the most appropriate type, since it enables the definition of access permission types. The main blockchain were meticulously detailed with the objective of providing a detailed look to different implementations of the same technology. It were studied different platforms that enable the development of decentralized applications with the objective of deciding on which platform would be the most appropriate for the development of the proposed application. With the objective of studying and aggregating the most positive aspects from different solutions, it was studied several Health applications. Following the analysis of the development platforms and Health applications, Hyperledger Fabric was considered as the most appropriate platform for the development of the proposed project.

HyperMedical, the developed application, is detailed in the System Model and Implementation chapters. The functionalities, data model and provided use cases are greatly detailed. HyperMedical's architecture was divided in three application layers to distinguish the main three components of the project, the HLF network, Back-office and the User application. The defined architecture and implementation reflects the main objectives of this dissertation which was to provide a unified solution which solves the problem of data fragmentation in Health applications while providing data and users' privacy. In the Implementation chapter, all the different technologies and system's entities are explained. Due to the privacy concerns of this project, the communications done between the different components of the system, as well, as the storage of data were taken in special attention.

For the evaluation of the proposed system, it were made 3 types of tests. Performance tests were done in order to find the optimal network configuration. Fault Tolerance tests relate to the impact in throughput and latency in increasing number of Kafka brokers and, consequently, increasing the number of faults tolerated by the system. And finally, Functional tests has as main goal test every functionality. Although, the proposed solution is lacking in terms of performance results when compared to other promising solutions, we found interesting results. These results indicate that using Blockchain technology in order to provide a solution for the data fragmentation problem in Healthcare is a viable option.

The immutability and logging characteristics of the blockchain provide a platform for users to know the operations done over their data and solves trustability concerns related to decentralized environments. Given the developed mobile application, it was provided the means for all entities

related to the Healthcare providers to use the system in such way that services, data and users are not fragmented across several platforms and environments.

6.2 Future Work

The evaluation of the proposed system should be done in a distributed environment with more resources. In this solution, the entities that compose the network would be disperse and distributed across different machines. This way, it is expected, that the results that were obtained in the Evaluation Chapter are higher, since the resource bottleneck would be removed.

The chosen ordering service, Kafka, only provides fail-stop tolerance to the implemented system. Given the modularity of Hyperledger Fabric, it would be interesting to test the performance impact in the implementation with a Byzantine Fault Tolerant algorithm such as BFT-SMaRt [31].

Given the limitations of the chosen library for the encryption of the data, encCC, it was only used the AES encryption algorithm. It would be interesting to test the encryption functionality using different algorithms to test the impact in performance. Another non-addressed issue related to encryption were encryption keys that are provided via a transient parameter and key vault should be used.

One issue that was not solved was, in an emergency situation, the patient should be able to share all the required information to the necessary medical staff at the same time.

Although it were observed interesting results in the implemented solution, there are improvements to the solution. Namely, the study by Delloite [85] which indicates that the solutions should provide two types of storage: an on-chain and an off-chain storage. Data that is bigger in size such as X-rays should be stored in an off-chain storage, due to the fact that the high size of the block will affect the processing and possibly the system's scalability. It would be interesting to implement this solution and provide an evaluation in order to test the impact of the off-chain storage solution when compared to an all on-chain storage solution.

Another matter that was not addressed was the authentication. Authentication could use the *Cartão de Cidadão* API for example or use a token based protocol such as OAuth or OpenID. The authenticity of doctors and other entities should be verified with their unique identifiers.

Bibliography

[1] "*Processo Clinico*." http://portalcodgdh.min-saude.pt/index.php/Processo_clinico.

[2] "Electronic health record." http://portalcodgdh.min-saude.pt/index.php/Electronic_Health_Record(EHR).

[3] "Medicalchain whitepaper." https://medicalchain.com/en/whitepaper/, 2017.

[4] "Sns." https://servicos.min-saude.pt/utente/.

[5] "*Hospital da Luz*." https://portalcliente.luzsaude.pt/pt.

[6] "*Lusíadas*." https://omeuportal.lusiadas.pt/pt/Paginas/Login.aspx.

[7] M. Hearn, "Corda: A distributed ledger," 2016.

[8] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009.

[9] M. Swan, *Blockchain - Blueprint for a new economy*. O'Reilly Media, Inc., 1st ed., 2015.

[10] A. M. Antonopoulos, *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. O'Reilly Media, Inc., 1st ed., 2014.

[11] I.-C. Lin and T.-C. Liao, "A survey of blockchain security issues and challenges," *I. J. Network Security*, vol. 19, pp. 653–659, 2017.

[12] K. Yip, "Blockchain and alternative payment models," 2017.

[13] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, pp. 382–401, 1982.

[14] M. Pilkington, "Blockchain technology: Principles and applications,"

[15] "On public and private blockchains." https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/.

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

[17] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake," 2012.

[18] M. Vukolic, "The quest for scalable blockchain fabric: Proof-of-work vs. bft replication," in *iNetSeC*, 2015.

[19] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An overview of blockchain technology: Architecture, consensus, and future trends," *2017 IEEE International Congress on Big Data (BigData Congress)*, pp. 557–564, 2017.

[20] D. P. Massimo, "What is the blockchain?," *Computing in Science & Engineering*, vol. 19, no. 5, pp. 92–95, 2017.

[21] M. Herlihy, "Blockchains from a distributed computing perspective," 2018.

[22] A. Back, "Hashcash - a denial of service counter-measure," 2002.

[23] V. Buterin, "Ethereum white paper," 2014.

[24] R. John, J. P. Cherian, and J. J. Kizhakkethottam, "A survey of techniques to prevent sybil attacks," *2015 International Conference on Soft-Computing and Networks Security (ICSNS)*, pp. 1–6, 2015.

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

[26] F. Saleh, "Blockchain without waste: Proof-of-stake," 2018.

[27] "Nxt whitepaper." `https://web.archive.org/web/20150203012031/http://wiki.nxtcrypto.org/wiki/Whitepaper:Nxt#Blocks`.

[28] P. Vasin, "Blackcoin's proof-of-stake protocol v2," 2014.

[29] "Casper." `https://github.com/ethereum/wiki/wiki/Casper-Proof-of-Stake-compendium`.

[30] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, pp. 398–461, 2002.

[31] J. Sousa, A. N. Bessani, and M. Vukolic, "A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform," *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 51–58, 2017.

[32] D. Schwartz, N. Youngs, and A. Britto, "The ripple protocol consensus algorithm," 2014.

[33] J. Kwon, "Tendermint: Consensus without mining," 2014.

[34] M. Conti, S. Kumar, C. Lal, and S. Ruj, "A survey on security and privacy issues of bitcoin," *CoRR*, vol. abs/1706.00916, 2017.

[35] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *SIGMOD Conference*, 2017.

[36] "Solidity documentation." `http://solidity.readthedocs.io/en/develop/`.

[37] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," *IACR Cryptology ePrint Archive*, vol. 2016, p. 633, 2016.

[38] "Quorum whitepaper." `https://github.com/jpmorganchase/quorum-docs/blob/master/Quorum%20Whitepaper%20v0.1.pdf`, 2016.

[39] "Chain protocol whitepaper." `https://chain.com/docs/1.2/protocol/papers/whitepaper`, 2018.

[40] R. G. Brown, J. R. Carlyle, I. Grigg, and M. Hearn, "Corda : An introduction," 2016.

[41] "Corda." `https://docs.corda.net/index.html`.

[42] "Hyperledger burrow." `https://github.com/hyperledger/burrow`.

[43] "Iroha whitepaper." `https://github.com/hyperledger/iroha/blob/master\/docs/iroha_whitepaper.md`.

[44] "Hyperledger swatooth." `https://sawtooth.hyperledger.org/docs/core/releases/latest/introduction.html`.

[45] "Hyperledger fabric." `https://hyperledger-fabric.readthedocs.io/en/latest/blockchain.html`.

[46] "Monax." `https://monax.io`.

[47] "Proof of elapse time." `https://sawtooth.hyperledger.org/docs/core/releases/latest/architecture/poet.html`.

[48] "Kafka and zookeeper." `https://hyperledger-fabric.readthedocs.io/en/release-1.1/kafka.html#kafka-caveat`.

[49] "Glossary." https://hyperledger-fabric.readthedocs.io/en/release-1.1/glossary.html.

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

[51] A. Dubovitskaya, Z. Xu, S. Ryu, M. Schumacher, and F. Wang, "Secure and trustable electronic medical records sharing using blockchain," *AMIA ... Annual Symposium proceedings. AMIA Symposium*, vol. 2017, pp. 650–659, 2017.

[52] T. Z. Ahram, A. Sargolzaei, S. Sargolzaei, J. Daniels, and B. Amaba, "Blockchain technology innovations," *2017 IEEE Technology & Engineering Management Conference (TEMSCON)*, pp. 137–141, 2017.

[53] "Ibm cloud." https://www.ibm.com/cloud-computing/bluemix/.

[54] "General data protection regulation (gdpr)." https://gdpr-info.eu.

[55] "Millennium." https://ind.millenniumbcp.pt/pt/Particulares/insurance/Documents/Questionario_MedisFIT.pdf.

[56] "Fidelidade." https://www.fidelidade.pt/PT/particulares/Familia/Seguros/Documents/SeguroVida-Proposta+IPC.pdf.

[57] "Real vida seguros." https://www.realvidaseguros.pt/images/stories/pdf/real_vida_previdencia_questionario_clinico.pdf.

[58] "Yaml." http://yaml.org.

[59] "Golang." https://golang.org.

[60] "Spring." https://spring.io.

[61] "Swagger." https://swagger.io/tools/swagger-ui/.

[62] "Java sdk for hyperledger fabric." https://github.com/hyperledger/fabric-sdk-java.

[63] "Ionic framework." https://ionicframework.com.

[64] "Fabric ca glossary." https://hyperledger-fabric.readthedocs.io/en/release-1.1/glossary.html#hyperledger-fabric-ca.

[65] "Public key infrascture." https://www.digicert.com/pki/.

[66] "Fabric ca." http://hyperledger-fabric-ca.readthedocs.io/en/latest/users-guide.html.

[67] "Sqlite vs mysql vs postgresql: A comparison of relational database management systems." https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems.

[68] "Openssl." https://www.openssl.org.

[69] "Ledger." https://hyperledger-fabric.readthedocs.io/en/release-1.1/ledger.html.

[70] "Couchdb." https://hyperledger-fabric.readthedocs.io/en/release-1.1/couchdb_as_state_database.html.

[71] "Json." http://json.org.

[72] "Couchdb vs leveldb." https://db-engines.com/en/system/CouchDB%3BLevelDB.

[73] P. Thakkar, S. Nathan, and B. Vishwanathan, "Performance benchmarking and optimizing hyperledger fabric blockchain platform," *CoRR*, vol. abs/1805.11390, 2018.

[74] J. Sousa, A. Bessani, and M. Vukolic, "A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform."

[75] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. 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. Vukolic, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *EuroSys*, 2018.

[76] "Cryptogen." http://hyperledger-fabric.readthedocs.io/en/release-1.1/commands/cryptogen-commands.html#.

[77] "Configtx." http://hyperledger-fabric.readthedocs.io/en/release-1.1/commands/configtxgen.html.

[78] "Grpc." https://grpc.io.

[79] "Peers." https://hyperledger-fabric.readthedocs.io/en/latest/peers/peers.html.

[80] "Mongodb." https://www.mongodb.com.

[81] "Chaincode tutorials." https://hyperledger-fabric.readthedocs.io/en/release-1.1/chaincode.html.

[82] "Cid." https://github.com/hyperledger/fabric/tree/release-1.1/core/chaincode/lib/cid.

[83] "Enccc." https://github.com/hyperledger/fabric/tree/master/examples/chaincode/go/enccc_example.

[84] "Ionic storage." https://ionicframework.com/docs/storage/.

[85] R. Krawiec, D. Housman, M. F. Mark White, F. Quarre, D. Barr, A. Nesbitt, K. Fedosova, J. Killmeyer, A. Israel, and L. Tsai, "Blockchain: Opportunities for health care," 2016.

Appendix A

Rest API

In all the methods that will be explained, it is verified that the user exists and that it is enrolled and also the type of user. In case of error, all the methods will throw a specific exception and will also return the string explaining the specific problem. In the following lists the "Error" represents the string containing the specific problem.

A.1 Administrator and User API

The *AdminController* has the following functionalities:

1. Create User – This method will register the user. All the users of the system will have to register in the system, so their first information is saved in the chain and registered in the Fabric CA. When a user performs the registration in the application it is added an attribute in the user certificate, detailing the type of the user that is enrolling.
    - Parameters: Name, user type, unique ID, date of birth and contact.
    - Result: Password, which is generated by the Hyperledger Fabric CA, or "Error".
2. Insert Initial Information – This method will insert the initial information about the user. Only the patients can call this method.
    - Parameters: Unique ID, gender, height, weight, blood pressure, alcohol consumption, smoker, blood type.
    - Result: String "OK" or "Error"
3. Update Initial Information – This method allows the patient to update his/her initial information. The parameters and the result are the same as the method to insert initial information.

4. Background – This method will insert the medical history about a user.
    - Parameters: Unique ID, diseases.
    - Return: String "OK" or "Error"
5. Login – This method validates if the user exists and if the password is correct.
    - Parameters: Unique ID, password.
    - Return: Type of user or "Error".
6. Get user initial information – This method retrieves the object user (data model represented in the section 3.3).
    - Parameters: Unique ID.
    - Return: User object or "Error".
7. Delete user – This method deletes all the information about the user that is stored in the blockchain.
    - Parameters: Unique ID, password
    - Return: "OK" or "Error".
8. Share Initial Information with Emergency Service – This method will share the initial information object of a patient with the emergency service.
    - Parameters: Unique ID.
    - Return: "OK" or "Error".

A.2 Emergency Service API

The *EmergencyController* has the following functionalities:

1. Init emergency record – This method is used when an emergency service wants to create a medical record about a patient to share it with a healthcare entity or when a user wants to share an existing medical record, which has the data type Patient Medical Record, as shown in the Table 3.4.
    - Parameters: Emergency Service ID, Patient ID, local, speciality, annotations, attachment (optional).
    - Return: Record ID or "Error".

2. Share emergency record – This method is used by the emergency service to share a medical record with a healthcare entity.
    - Parameters: Record ID, Healthcare ID and Patient ID.
    - Return: Share Record or "Error".
3. Query emergency record by patient ID – This method queries all the emergency records of a patient by ID.
    - Parameters: Emergency service ID, Patient ID.
    - Return: Emergency Record or "Error".
4. Get patient initial record – This method returns the initial information of a patient.
    - Parameters: Emergency service ID, Patient ID.
    - Return: Emergency Initial Record or "Error".

A.3 Medical Record API

The *MedicalRecordController* has the following functionalities:

1. Init medical record – This method is used to create a new medical record for a patient. It will be created a Patient medical record and a Healthcare entity medical record, so both of the intervenients could access the data.
    - Parameters: Healthcare ID, Patient ID, Local, Speciality, Annotations, Attachment (optional).
    - Return: Record ID or "Error".
2. Read medical record – This method retrieves a specific medical record depending on the role.
    - Parameters: Record ID, User ID.
    - Return: Medical Record or "Error".
3. Query medical records by patient ID – This method retrieves all the medical records owned by a patient.
    - Parameters: Patient ID.
    - Return: String of a JSON array or "Error".

4. Delete medical record – This method removes a record that was insert by the patient.
    - Parameters: Record ID, Patient ID, Password.
    - Return: String “OK” or “Error”.
5. Withdraw access privileges – This method revokes the access to a user.
    - Parameters: Record ID, Patient ID, Healthcare ID.
    - Return: String “OK” or “Error”.
6. Share medical record – This method shares a medical record with a healthcare entity.
    - Parameters: Record ID, Healthcare ID, Patient ID.
    - Return: Share record or “Error”.
7. Share medical record with emergency service – This method shares a medical record with the emergency service.
    - Parameters: Record ID, Patient ID.
    - Return: Share record or “Error”.
8. Query records by healthcare ID – Returns all the records that was shared with that healthcare entity.
    - Parameters: Healthcare ID.
    - Return: String of JSON Array or “Error”.