TÉCNICO LISBOA

Universal Consent Management Platform

André Santos Martins Nunes

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor(s): Prof. Miguel Nuno Dias Alves Pupo Correia

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. Miguel Nuno Dias Alves Pupo Correia
Member of the Committee: Prof. Carlos José Corredoura Serrão

November 2020

Dedicated to my family and friends

Acknowledgments

I would like to start by giving my thanks to my supervisor, professor Miguel Correia, for his continuous support, guidance and dedication throughout this journey. For the encouragement and much advice, I am grateful.

I would also like to thank my parents and family who, throughout the years, have always supported me during my studies and encourage me to never stop.

I also wish to thank Blockbird, specially José Figueiredo and Carlos Faria, for being supportive, contributing with good ideas and giving feedback during the elaboration of this thesis.

Lastly, I would like to thank my closest friends who were very supportive during the elaboration of this project and whose feedback was always welcome.

Resumo

Os sistemas baseados em Blockchain têm vindo a receber cada vez mais interesse tanto na área de investigação com na indústria. Uma Blockchain é um registo distribuído que consiste numa estrutura de dados, onde só é possível anexar outros dados, que armazena uma lista ordenada de transações, replicada por vários nós que por sua vez estão conectados entre si através da Internet. Neste documento vamos explorar as vantagens desta tecnologia quando aplicada à gestão de consentimentos na área da saúde digital. Os dados médicos devem ser propriedade dos pacientes e por conseguinte controlados pelos mesmos apesar de estarem espalhados por diferentes sistemas de saúde. Devido ao recente RGPD, terceiros, como é o caso das instituições de saúde e laboratórios de pesquisa médica, que armazenam dados médicos, não podem proceder à partilha destes mesmos dados médicos sem ter o consentimento adequado do proprietário (paciente). Propomos uma solução que tira proveito desta tecnologia em ascensão para implementar uma plataforma de gestão de consentimentos, a qual permite aos pacientes ter um maior grau de controlo sobre os seus dados médicos enquanto providencia às instituições médicas um meio de permanecer em conformidade com a legislação da UE.

Palavras-chave: blockchain, Ethereum, gestão de consentimentos, saúde

Abstract

Blockchain systems have received an outburst of interest in both research and industry. A blockchain, or distributed ledger, consists of an append-only data structure that stores an ordered list of transactions, replicated in several nodes that are connected by the Internet. In this document we explore the advantages of this technology when applied to consent management in the eHealth area. Healthcare data should be owned and controlled by patients despite being scattered in different healthcare systems. Due to the most recent GDPR, third parties such as healthcare institutions and medical research related institutions, who store medical records, can not share those same records with another institution unless given proper consent by the owner (patient). We propose a solution that takes advantage of this rising technology to implement a consent management platform that allows patients to have a better degree of control over their medical records while providing medical institutions a means to stay compliant with EU's law.

Keywords: blockchain, Ethereum, consent management, healthcare

Contents

List of Figures

Chapter 1

Introduction

Over the past few years *blockchain* technology has been obtaining increasing interest due to the vast possible variety of applications this new technology has shown [CDE+16, GHM+17]. It all started back in 2008 when an individual or group of individuals named Satoshi Nakamoto first presented Bitcoin [N+08], a peer-to-peer version of electronic cash with the objective of eliminating the need for a trusted third party in online transactions.

1.1 Topic Overview

With the evolution of blockchains and the raising popularity of Bitcoin, people started looking at the blockchain from a different perspective, in which, blockchains, although public, could have other uses and purposes. This has lead to the development of new blockchains that are more restrict and controlled, called *permissioned* or *private blockchains*. This also enabled the blockchain application scope to become wider and start affecting a much more ample variety of different areas, such as the Healthcare and real estate business areas. Today standalone applications are being transformed into distributed applications – DApp. This happens mainly because of the growing trend of deploying applications in cloud platforms.

Regarding healthcare business area, *Electronic Health Records* (EHR) [And17] have been gaining more and more importance over the past few years due to the advantages they bring in comparison to the traditional storing methods. An EHR is a digital version of a patient's paper chart. EHRs are real-time, patient records that make information available instantly and securely to authorized users. One of the key features of an EHR is that health information can be created and managed by authorized providers in a digital format capable of being shared with other providers across more than one health care organization. The benefits of EHRs can be categorized in three main dimensions:

- Clinical – decrease in errors and improvements in healthcare quality;
- Organizational – increase in operational performance and financial gains;
- Social – improvements on the overall population health (individual or aggregate).

Concerning the social dimension, individuals with long term diseases or conditions benefit the most from EHRs since these provide better monitoring capabilities for both their health and their treatments. On the other hand, at the aggregate population-level, the data provided by EHRs can be used to analyze and detect health patterns and trends on a given community. Healthcare organizations, who implement and deploy EHR as their main storage method for patients related medical data, can expect these systems overall benefits to outweigh their software and hardware implementation costs, as well as personnel training and system maintenance.

Companies and organizations inside the European Union must stay compliant with the General Data Protection Regulation (GDPR) [Eur16] in order to practice their business activities. With the GDPR, companies and organizations can not share any user personal data with third parties unless a proper consent has been given by the owner of that data, even if a sharing solution already exists [MPA+18, SC17]. In the healthcare business area, *consent management* can be defined as the set of policies a given medical institution applies in order to give their patients the ability to decide which user medical related information they are willing to share, with whom and under what circumstances. Having a consent management platform is a key mechanism to companies that rely on their users' personal data to practice their business activities.

The present document proposes Universal Consent Management Platform (UCMP) – a blockchain based consent management platform that allows healthcare institutions to share their EHR records with other institutions and organizations in accordance with their patients consents. The objective of the project is to design and implement a consent management platform that allows patients to monitor and control, up to a certain degree, their medical records. The following approach provides patients a means to manage their data allowing them to choose what to share and with whom. The proposed solution also enables healthcare institutions and organizations to stay compliant with the most recent GDPR policy by providing a tamper proof system that stores their patients consent preferences.

The implementation of UCMP uses the Google Firebase [1] service as the backbone for its core components. Firebase was used in order to provide our system with an easily manageable database, a backend server through the use of cloud functions, authentication mechanisms and a hosting service. Firebase is a Backend-as-a-Service (BaaS) that facilitates building apps without

[1] https://firebase.google.com/docs

the need to manage a server. On the other hand, while implementing UCMP's prototype we used Truffle [2] as the Ethereum client in order to connect and interact with the blockchain. Although we used Truffle due to its simplicity, Hyperledger Besu could have also been used. Besu provides additional features when compared to Truffle and therefore, an increase on the solution's complexity. This complexity was not desired when implementing a prototype, but should UCMP be deployed on the Ethereum main net, it is highly advisable to use Besu instead of Truffle.

1.2 Goals

This document addresses the lack of a system or platform that provides patients with a consent management platform while still providing strong privacy assurances. With this in mind, UCMP should assure the following requirements:

1. **Blockchain** – The solution shall incorporate a blockchain as a base for the whole system. This blockchain will operate under a public environment.
2. **User identity** – The identity notion in the system must be strong and reliable: an account can be traced by our system to one user and one user only.
3. **GDPR compliance** – The solution must be in compliance with the GDPR.
4. **Privacy** – Although user identity is required, no user should be able to identify another inside the network. Only authorized auditors shall be able to access user identities.
5. **Prototyping** – The system shall be deployed and later presented using a fully working prototype that enables testing and execution of all provided functionalities.
6. **Management interface** – The system should include an easy-to-use, versatile and intuitive management interface that allows patients to unequivocally set their sharing preferences.

1.3 Thesis Outline

This dissertation begins by giving, on Chapter 2 core concepts regarding blockchain technology as well as give some insights on the topics of GDPR and consent management. In this same Chapter we also present and briefly discuss current implementations of already existing systems that attempt at solving the consent management problem regarding the healthcare business

[2] https://www.trufflesuite.com/docs

area. UCMP, the proposed solution, and its core operations are then presented and described on Chapter 3. Chapter 4 presents and discuss an evaluation methodology to assess UCMP as well as the results from the experimental evaluation. Finally, Chapter 5 concludes this dissertation.

Chapter 2

Background and Related work

This section provides an understanding of how blockchains work. Specifically, several existing blockchain implementations will be introduced. Moreover, core concepts regarding blockchain's confidentiality and privacy will be addressed as well as existing implementations of consent management systems.

This section is organized as follows. Section 2.1 defines the core concepts of a blockchain as well as key notions of technologies that support blockchains. Section 2.2 briefly explains how Ethereum blockchain work first by addressing its functionality and goal and then by explaining the concepts of smart contracts and their interaction on the main chain. Section 2.3 gives a brief introduction to the fundamental requirements of enterprise blockchains and how they emerged. Section 2.4 explores Hyperledger Besu implementation and consensus protocols as well as their usage under different environments. Section 2.5 gives insights on existing systems that provide consent management platforms and key techniques that help achieve confidentially and privacy on blockchain networks. Finally, Sections 2.6 and 2.7 briefly introduces to the most important European Union legislation and policies regarding consent management and discuss current implementations of consent management architectures, respectively.

2.1 Blockchain

Blockchain is a cryptographically secured distributed ledger [N$^+$08, Bec18, DLZ$^+$18, Und16, Pec17, Her19, XPZ$^+$16] that uses a consensus mechanism to keep consistency, whenever a new transaction needs to be validated, and is maintained by all the nodes within its network. On the other hand, the name Blockchain can also be used as a tamper-resistant database that is consistent across a large number of nodes. One of the objectives of the blockchain technology is to remove the middle-man that is present in all kinds of transactions. Therefore this network

does not rely on any central trusted authority to keep on running.

There are two main types of blockchain in regard of their access scope: public and private [DLZ+18, Pec17]. The main difference between a public and private blockchain is the level of access granted to its participants [Her19]. *Public* or *permissionless blockchain* can be accessed by anyone with an internet connection and are behind most of today's digital currencies. This means that anyone can create a personal address, begin interacting with the network and leave the network at anytime, if they wish to. In order to add new entries to the ledger there are consensus protocols [Cor19] that are responsible for choosing the next block for the blockchain.

A consensus protocol is distributed algorithm used by a set of computers (nodes) to reach agreement on a value [Cor19]. Since these networks are open to the public, having a combination of economic incentives and game theory are a vital factor to help ensuring its participants behave properly and do not attempt to cause any harm to the network or its participants. Game theory is behind the execution of most consensus protocols with the most common ones being Proof of Work [Pec17] and Proof of Stake [KN12].

On the other hand private or permissioned blockchains have restrictions on who can join and participate in the blockchain. In these cases the owner of the blockchain must act like a gatekeeper in order to enforce some kind of access control policy, enabling them to control who has access to write operations on the blockchain and who has permissions to read the information contained in the blockchain. Private blockchains, on the opposite side, do not need any kind of incentive or rules in order to prevent their participants misbehaviour. Given the fact that its participants are known *a priory* (we know the mapping between user's and their blockchain account), if some users were to behave poorly they could be easily identifiable and would, eventually, suffer the consequences of their misbehaviour.

A consensus algorithm is a procedure through which all the participants of the blockchain network reach a common agreement about which block should be added next to the blockchain [DLZ+18]. In a way, consensus algorithms achieve reliability for the blockchain network and establish trust between unknown participants in a distributed computing environment. Essentially, the consensus protocol makes sure that every new block that is added to the blockchain is the one and only version of truth that is agreed upon by all the nodes in the blockchain.

There are two main data structures in the blockchain: *transactions* and *blocks*. The former is a piece of data representing an atomic event that is allowed by the underlying protocol of the blockchain. This event could represent a variety of actions, if we think of the blockchain as a distributed ledger that keeps a record of who owes who how much then a transaction is a payment but if we think of the blockchain as a data structure then a transaction is just an event

that updates the data store. The latter is simply a collection of multiple transactions that are aggregated together by the miners.

As mentioned above, miners are responsible for collecting transactions that were previously sign by a user's wallet and construct their own block of transactions. Miners can select which transactions to include on their block, from a pool of unconfirmed transactions on the network, that are still waiting to be processed. Afterwards, in order to add this block of transactions to the current blockchain a miner has to find the solution to a cryptographic puzzle (this is the proof of work). It is important to notice that every miner will have a different problem to solve based on the block they built. These problems are all equally hard to solve and consist of finding a hash that starts with an X given number of consecutive zero's. This hash is calculated based on the hash of the previous block, plus the transaction data that is present on the block itself, plus a random nonce that has to be generated by the miner. Since the hash of the previous block and the transaction data is constant, miners have to generate multiple random nonces in order to find one that, together with the rest of the block data, solves this problem.

This is the process referred to as *mining*. Once a miner has solved this cryptographic puzzle it broadcasts to the entire network its block of transactions along side the solution it found. Finally, a consensus protocol is executed and once there is an agreement between nodes, this new block is added to the current blockchain giving an incentive, usually in the form of a crypto currency, to the miner who added the new block to the blockchain. Proof of work as described above require that a miner invests both time and computational resources, which represent an electricity cost in order to have a chance at winning the race *versus* other miners. For this reason, proof of work is seen as a countermeasure method that prevents attackers from disturbing the normal flow of the blockchain network.

Given the fact that miners are competing against each other to see who can first solve the cryptographic puzzle and add a new block to the blockchain network, sometimes there can be a "draw", where two or more miners find a solution at nearly the same time. In these cases, the entire network might not agree on the same choice of the new block. When such event happens we are in the presence of a fork. Usually this situation arises because it takes some finite time for the information to propagate in the entire blockchain network and hence conflicted opinions can exist regarding the chronological order of events. These forks resolve themselves when one of the chain dies because, eventually, the majority of the full nodes will choose the other chain to add new blocks to and sync with.

The most relevant threat to a blockchain network is the 51% attack [ABC17], where more than half the network power is concentrated in a single entity. This entity can be either a single

person (node) or a collaboration between users such as mining pools, where a set of users come together in order to solve the cryptographic puzzle faster by applying a divide and conquer strategy and thus, splitting the rewards once they have successfully mined a new block. Having 51%+ of the network power allows this entity to alter the consensus rules as it sees fit, which could lead to a monopoly of the network, therefor defeating the main purpose of the blockchain technology – decentralize services.

There are two main types of nodes in any given blockchain: full nodes that consist of a full copy of the entire blockchain since the genesis block and lightweight nodes that only maintain a copy of a couple blocks of interest to them. When making decisions for the future of the network, full nodes are the ones that vote on proposals. Miners can be either a full node themselves or be a lightweight node and receive data from other full nodes on the network, in order to know the current status of the blockchain and the required parameters for the next block. Since they must verify that a given transaction is valid before appending it to the block they are assembling this requires miners to access the entire blockchains from the time of the genesis block in order to guarantee that a given transaction is valid (that a given person A has indeed the amount that they are trying to transfer to a person B)

In summary, a blockchain can be seen as a series of consecutive blocks that are linked to one another via a cryptographic hash. In other words every block has a hash based on the hash of the previous block plus the information that is contained on itself, this way should a block that is already within the chain be altered all subsequent blocks would become invalid, since this altered block would have a different hash therefore making every subsequent block also have a different hash. It is also important to mention that a copy of the same blockchain is spread among a vast number of nodes. This two properties together are behind every blockchain's strong tamper proof resistance characteristic. Considering that, in order to alter the information present in a single block, which has already been incorporated deep in the blockchain, not only does an attacker have to calculate every subsequent block's hash but also convince the other nodes that they possess the wrong copy of the blockchain. This means that transactions contained on a block that is deep within the blockchain can not be modified or deleted. If a user wants to update the information of a given transaction they must create a new one and append it to the end of the blockchain.

2.2 Ethereum

Ethereum is an open software platform based on blockchain technology that enables developers to build and deploy decentralized applications with a built-in Turing-complete programming

language [LCO+16, DAK+16, BP17, Sza97]. On the Ethereum blockchain, Ether is the cryptocurrency that fuels the network. Apart from being a currency, Ether is also used by people on the network to pay for code execution (transaction fees).

In other words, Ethereum is a distributed public blockchain network [B+14] that focus on running code for any decentralized application that is deployed on its network [Dan17]. In order to do this, users have to write *smart contracts* [Pec17] – agreements between mutually distrusting participants, which can also be seen as computer programs that are deployed on the Ethereum blockchain and have their correct execution enforced by the consensus protocol of the blockchain. In Ethereum, smart contracts [DLZ+18] are identified by an address therefore, when a given user wants to invoke a certain smart contract they have to send the transaction to that contract's address. Apart from having a unique address each smart contract also holds some amount of virtual coins (Ether) and, each single one of them, has its own private storage. Guaranteeing the correct execution of smart contracts is a necessity for achieving their effectiveness. Avoiding to do so would mean that an attacker could tamper with the contracts execution and therefore, redirect money from a legitimate user's transaction to oneself [ABC17].The code behind Ethereum smart contracts is a low-level bytecode language usually called Ethereum virtual machine (EVM) code. Users, on the other hand, can write smart contracts using a high-level programming language called Solidity which is compiled into EVM code before executing the smart contract [DAK+16].

As in any other public blockchain Ethereum also needs to have a method of rewarding miners for they computational effort when mining new blocks [BP17, Dan17]. In order to achieve this, Ethereum uses the concept of Ethereum gas. Ethereum gas is a unit that measures the amount of computational effort that it will take to execute certain operations. Every single operation that takes part in Ethereum requires some amount of gas, be it a transaction or a smart contract execution. Miners get paid, as a reward, a certain amount of Ether based on the gas it took for them to execute a given operation.

It is important to understand that gas is simply a unit for measuring the cost of a transaction. Gas is the amount of computational power required while Ether is the currency used to pay for that gas. There is no fixed price for conversion between gas and Ether, it is up to the sender of a transaction to specify the gas price that they are willing to pay for the execution of their transaction. On the other hand, it is up to the miner to verify and include on their block any transactions they want. With this being said, miners usually try to include first transactions that specify a higher gas price when compared to the average gas price of the network. Since gas price is not fixed, it can also increase and decrease according to the network traffic on the moment a transaction is created. Gas price goes up during times of high network traffic, since

there are more transactions competing among each other to be included in the next block and equivalently, gas price goes down during times of low network traffic. Inside this concept of Ethereum gas, there is also the concept of gas limit and a refund mechanism that enables users to get their transaction fees back, in case something goes wrong during the execution of a smart contract or the transaction itself. Gas limit refers to the maximum amount of gas the sender is willing to pay to get his transaction included in a block. Miners as a counter-measure against resource exhaustion attacks will stop executing a given transaction the moment it run out of gas. This means that every smart contract execution is finite thus, even if there is an infinite loop inside a certain smart contract the operation will eventually run out of gas and it will be aborted. In these cases, the operation that ran out of gas is reverted back to the original state, before it has started and the operation fee is not refunded to the operation generator, since the latter must sill pay the miner the fee for their computational costs. On the other hand, if there is any gas left over, it will be refunded to the operation generator.

2.3 Enterprise blockchain

The enterprise blockchain concept has been developing over the past few years [Pec17]. When blockchains first appeared, there were only public blockchains and these networks would be deployed in unsafe environments where any user could attempt an attack to exploit both these networks and other users. At this point of time, it was unthinkable that this technology could have any possible applications other than cryptocurrencies.

Over the years and with the increasing popularity of this new technology, companies started applying the concept of public blockchain on their own internal communities creating the first private blockchain networks [Pec17]. These blockchain networks, also known as enterprise blockchains [RBBM19] had a new feature – the need to have a strong consensus protocol and economic incentives were not a must but rather an option. The first permissioned blockchain to appear was *R3* [BCGH16] which consisted of a consortium formed by a group of financial institutions who aimed at improving the efficiency of payments between banks. Due to this approach, the belief that blockchain usage was only to create cryptocurrencies started fading away giving opportunity to the rise of new concepts where blockchains could be applied on. The two most popular permissioned blockchains are *Hyperledger Fabric* (HLF) [ABB$^+$18, Cac16] and R3.

Hyperledger Fabric is a modular permissioned blockchain platform that allows several well-known implementations of different components (such as ordering and membership services), to be easily plugged in a blockchain system. The ordering service is responsible for creating blocks

for the distributed ledger, as well as the order by which blocks are appended to the ledger. It uses a pluggable consensus component which in the current release supports three different implementations for the ordering service:

- Solo – In this implementation only a single node is used for ordering. As a result, this implementation is not fault tolerant. For that reason, Solo implementations should only be considered for testing purposes.

- Raft – This implementation is a *crash fault tolerant* (CFT) ordering service based on an implementation of Raft protocol [OO14]. It follows a "leader and follower" model, where a leader node is elected and its decisions are replicated by the followers.

- Kafka – This implementation is similar to Raft as it is also a CFT implementation that uses a "leader and follower" node configuration. Kafka utilizes a ZooKeeper ensemble for management purposes rather than the Raft protocol approach.

Once a block is appended, the decision is final, so the block cannot be replaced or modified. HLF also supports the execution of non-deterministic smart contracts (called chaincode), that are executed and validated by the endorsing peers, who maintain a ledger, a state database and follow the endorsement policies.

The increasing popularity that enterprise blockchains have gained over the years has resulted in companies who already had a certain business area of operating to expand their services in order to also include blockchain related products. On the other hand there are also some entities that have been founded to exclusively serve the enterprise blockchain market. These entities are usually small start-ups whereas the former tend to be larger-scale corporations originated from other industries. It should also be noted that the most significant part of enterprise blockchains are related to the finance and insurance department which could mean that this technology is not yet as versatile as one could imagine. Surprisingly as it might seem, the vast majority of enterprise networks are designed for shared use between different, non-affiliated entities, which leads to a greater and simpler service exchange method that enables these organizations to thrive on their business's market, by giving a greater access to user related information for instance.

2.4 Hyperledger Besu

Hyperledger Besu is an open source Ethereum client implementation built for enterprise use. Besu can operate on the Ethereum public network or on a private permissioned network and it

includes several consensus algorithms such as proof of work (PoW), proof of authority (PoA) and Istanbul Byzantine fault tolerance (IBFT) [SHW19].

Depending on the type of blockchain Besu is being used with, a different consensus protocol from the previous list may be applied to better serve the requirements of that given blockchain environment. For instance, PoW consensus is usually used on the main Ethereum blockchain when mining is required but, on a private permissioned blockchain, where participants are known to each other and a certain level of trust exists between them, a PoA consensus protocol is preferred. On the latter, IBFT is an example of a PoA consensus protocol that uses approved accounts as validators. These validators are responsible for creating appending blocks, one at a time, and for voting to add or remove validators from the approved accounts list. Blocks in IBFT protocol are final, which means that there are no forks and any valid block must be somewhere in the main chain. In order to avoid a faulty node from generating and appending an invalid block, each validator can only append new blocks when it has received 2F+1 signatures from other validators confirming that the block is indeed valid and can be appended to the main chain. These signatures are also written on the block's header before inserting it into the current chain. As it is, this approach could cause problems on block hash calculations since the same block from different validators could have a different set of signatures. This would cause the generation of different block hashes for the exact same block given that [signature one] + [signature two] + [signature three] + [block transaction] has a completely different hash than [signature two] + [signature three] + [signature one] + [block transactions].

Being an Ethereum client, Besu contains:

- An execution environment for processing transactions in the Ethereum blockchain
- Storage for persisting data related to transaction execution
- Peer-to-peer networking for communicating with the other Ethereum nodes on the network in order to synchronize state
- APIs that allows application developers to interact with the blockchain

2.5 Blockchain security

When it comes to blockchain security there are a couple of terms that are important to understand such as confidentiality, privacy, integrity and availability. These properties play a key role when we want to meet the requirements that enterprise blockchains should have, in order to satisfy their users needs. There are also a couple of frameworks that attempt to produce

more attractive features on top of the current blockchain technology, in order to make it more appealing for companies to move a centralized service to a decentralized blockchain.

Given the four concepts mentioned above, it is important to understand the differences between them in order to have a better grasp of the reasons behind the development of frameworks that are used to deploy applications on top of existing blockchain networks [ZNP15]. *Confidentiality* can be seen, in the context of protecting data, as the process and methods behind transforming a given piece of data so that it is possible to limit the access by third parties [ZNP15]. This could involve hiding the transaction details, account and wallet balances, or any relevant information that is important to keep secret from the general public and is present within the blocks of a public or private blockchain network. Similarly, privacy also involves protecting a piece of information that is involved within the blockchain network but in this case the goal is to protect the identity of its participants rather than the information referring to the transaction that is contained inside the blocks. As mentioned in Section 2.1, *integrity* refers to the ability that a blockchain has to store its information in an immutable (extremely difficult to tamper with) state. This is, usually, achieved through the consensus protocol used by a given blockchain. Equivalently availability is also achieved due to the characteristics of the blockchain network. *Availability* in the scope of a blockchain refers to its ability to withstand outages and attacks. If a given network were to go down, then every single participant in that same network would have to be offline by either having their machine not connected to the network or due to a fault on their internet connection. Even though this might be possible to happen on small blockchain networks this is nearly impossible to happen to a well established network that has users all over the globe. Even if there were no power outages a given network could also have a low availability if it is not able to handle a high amount of traffic and transactions while still remaining functional and responsive to its users.

Despite existing several frameworks [KMS+16] that tackle these problems, the high requirements of confidentiality and privacy that are asked for, along side the high amount of transaction per second which results in low availability are still the biggest drawbacks that prevent large companies, such as banks, to move their services to a decentralized network.

There are a couple of companies that attempt to ease the burden of learning a new specific programming language for a certain framework by developing simple and intuitive internal frameworks. Not only do some of these frameworks facilitate our everyday work but they also are equipped with additional features that more experts users could exploit in order to produce better and more sophisticated applications and programs, that meet higher expectations while still complying with the base requirements. As an example of such frameworks we have

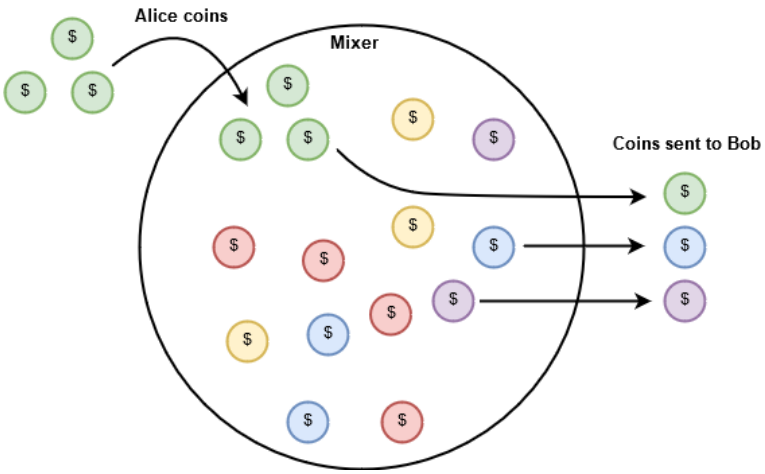

Figure 2.1: Example of how mixers work. Alice wants to send Bob 3 coins

both Hawk [KMS+16] and Coco Framework [Mic17]. For the sake of achieving key enterprise requirements, these frameworks take advantage of existing mechanisms and techniques that mainly attempt to restrict access to blockchain networks while still maintaining a decentralized approach. Some of most common techniques include the usage of mixing and zero knowledge proofs [ZNP15, YGWO16].

Mixing can be seen as a black box service that requires a trusted third party where users (senders) send their funds to [YGWO16], instead of the final owner (receivers). Afterwards, these funds are partitioned into different random values mixed together with other funds from another senders and sent at different times to different addresses all belonging to the same rightful owner. This method helps obfuscate the path that a single transaction takes, making it extremely hard for an observer to find the path between sender and receiver.

Given the extra confidentiality and privacy that mixers offer, they are usually used by ill intended users who want to either cause any type of damage to a network and its users or participate in illegal matters. Nonetheless there are several users who are only concerned with their transaction's privacy and therefore often end up resorting to mixers in order to achieve their goal. This, however, may have negative side effects, for instance, a legitimate user may use a mixer to send his funds to a friend and once the funds are inside of a mixer they may get mixed with other funds from other users that might be using that very same mixer to perform an illicit activity. This results in his friend receiving tainted funds. Figure 2.1 gives a visual example of how mixers operate.

One other concept that is used to improve a users privacy inside a blockchain is *zero-knowledge proofs* (ZKP). By applying this concept, a given party can prove to another that a statement is true without revealing any information to the other participant. In practice, by using a zero-knowledge protocol, users could send a transaction to a miner without revealing

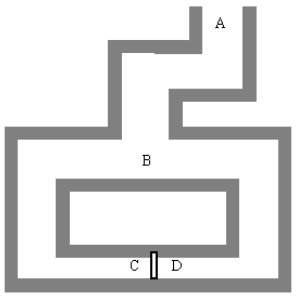

Figure 2.2: Example of the cave's layout

their account balance and prove to the miner that they in fact do hold enough funds to perform the transaction they are requesting. With this method users can participate in blockchain networks without the disclosure of their funds to other participants. Hyperledger Fabric provides anonymous client authentication with Identity Mixer by leveraging ZKP to offer anonymous authentication for clients in their transactions.

The best analogy to explain zero-knowledge proofs is to resort to the cave example. Lets suppose we have a cave that has 2 separate tunnels and they are connected through a magic door that only opens if you now the secret password, Figure 2.2 shows an example of this cave. This magic door is placed between points C and D. Alice wants to prove to Bob that she knows the secret password, but she does not wish to reveal this secret to Bob. In this example, both Alice and Bob are outside the cave on point A. Alice goes first inside the cave until she is by the magic door (position C or D), and wait for further instructions from Bob. Bob then goes inside the cave as well and stops by the intersection of both tunnels (position B) shouting which direction he wishes to see Alice appear from.

After this step there are two possible outcomes, Alice is either on the right side of the magic door and just walks to meet Bob or Alice is on the opposite side of the door and has to use the magic password that only she knows to cross to the other side. Performing this process once ends up proving nothing, since there is a 50/50 chance that Alice got lucky and was already on the right side of the door, but if we repeat this experiment a couple times, eventually Bob can be sure that Alice does indeed know the secret password and that it was not luck what got her to appear from the correct side of the cave. This process is repeat indefinitely until Bob is satisfied with the result, if this is done 10 times then the probability that Alice got lucky is merely 0.098%. The higher the amount of times this protocol is executed the lower the probability that Alice is lying. It is important to note that by applying this process there will be always a probability, small as it might be, that the prover (Alice) got lucky on every single attempt, this is known as the soundness error.

Zcash [HBHW16] takes advantage of the base concept of ZKP by implementing Zero-Knowledge

Succinct Non-Interactive Argument of Knowledge (zk-SNARK). The main improvement of zk-SNARK when compared to the normal method of ZKP is its non interactive property. With this approach, the verifier does not need to communicate with the prover in order to validate a certain zero-knowledge proof.

HLF has a *membership infrastructure* that enables participants of the network to strongly authenticate themselves in transactions. As a result, no unauthorized nodes can store the ledger or issue transactions. HLF implements a *channel architecture* that can be used to offer privacy for its participants. A channel, in this case, can be thought of as a virtual overlay blockchain network, that sits on top of a physical blockchain network. Channels in Hyperledger Fabric are configured with access policies that govern access to the channel's resources, restricting access to information exclusively within participants in the channel. Furthermore, HLF also offers the ability to create *private data collections*, that allow a certain subset of organizations on a channel, the ability to endorse, commit, or query private data without having to create a separate channel. In other words, there is some data inside a channel that is accessible only by a subset of entities within the channel. Data collections can also be used when transaction data must be kept confidential from the ordering service nodes.

2.6 GDPR

The General Data Protection Regulation (GDPR) [LCT] is a regulation in EU law which states how organizations should handle any type of personal data that they store or process by giving rights to data subjects and duties to data holders. Its main objective is to give individuals a greater control over their personal data that is held by third-parties. This regulation is followed by every country, member of the European Union (EU), and protects personal data regardless of the technology used for processing that same data. Countries may also have their own set of data protection laws while still complying to the GDPR.

A piece of data is considered to be personal data if it has any type of information that relates to an identified or identifiable living individual. Personal data also includes several pieces of information which collected together can lead to the identification of a particular person. Encrypting personal data or applying pseudonymization techniques in a way that it is still possible to restore the original state of a piece of data, remains personal data and falls within the scoped of the GDPR. On the other hand, anonymized data which can never be restored to its original state stops being considered personal data and no longer falls under the scope of the GDPR.

Among the several rules imposed by the GDPR, the right to access and right to erasure

(most commonly known as right to be forgotten) [Fin18] should be considered the ones with most interest to both companies and clients that have their data held by a third party. On one hand, these entities must fully comprehend the implications of this rights so that, in the event that a given client evokes them, they are prepared to take the necessary measures to ensure that everything is handle according to the law. On the other hand, clients should be completely aware of their rights in order to prevent companies to take advantage of their data.

The right to access states that every subject has the right to inquire the controller whether or not their personal data is being used, the purpose of this usage as well as other controllers that might have access to this same data. Applying this article of the GDPR to blockchain technology would raise serious questions to how would the conditions that are stated within the article be imposed and verified by external parties. Supposing that a data subject can contact a node, it would be impossible for that same node to verify the usage of a subject's data.

According to the right to be forgotten controllers are obliged to delete any and every type of personal data they own from a given data subject, should this request to do so. The same article also states that controllers may refuse the deletion of data under certain circumstances such as: deletion of personal records that would indulge in a loss (either monetary or information wise) to the company, or result in a obstruction to justice or public interest. One of the strongest properties of blockchain technology is its immutability meaning that, once written on the blockchain, data can never be forgotten. This results in a failure of a straightforward application of this right.

As mention on Section 2.1 Blockchain can be used as a decentralized database where its participants can freely access the information contained there. Given the concepts of GDPR and blockchains its plausible to think that these two concept are highly incompatible (especially public blockchains) since data that is stored on the blockchain is exposed to anyone within its network. This means that if there is any kind of personal data kept inside a blockchain this is easily accessed and leaked, therefore going against the goals of GDPR leading us to conclude that personal data should not be stored inside a blockchain.

2.7 Consent management

Consent management refers to a set of processes and policies a given company applies in order to give their customers the ability to decide which user related information their are willing to share and with whom. Since the GDPR implementation back in 2018, companies started giving more attention to how they collect, store and use user related data [RDD+18]. Some enterprises quickly started adopting their own consent management platforms that enable an autonomous

process to take place. These platforms are specially useful for large sites that have high volumes of traffic since they allow its users to understand what data is being collected and for which purposed they might be used for.

The vast majority of these platforms are centralized or, in other words, each company holds, on their own, the power to manage both user information data and their consent [GZL+17]. As it is, users have to trust that this third party will both follow what has been agreed on, and it will not exploit in any way the information they possess. This may raise some serious questions on whether or not some sites are trustworthy or not. One possible solution to this problem would be to apply DLT (distributed ledger technology or blockchains) to enforce a consent management platform where users are in total control of their consent management. The two main features that make blockchain based solutions attractive are:

- **Trust improvement** – users no longer have to rely on a single point of trust, as mention on Section 2.1, trust is now distributed among every single participant on the blockchain network;
- **Data immutability** – one of the main features that blockchain technology brings is data immutability since the consensus protocols that are used, ensure that once a given piece of data inside a block is attached to the existing blockchain it is extremely hard to modify that block. Data is distributed among several peers so, in order to change a block on the blockchain, every single peer that holds a copy of the blockchain would also need to have their copies modified.

There are a couple of systems that already attempt to solve this problem by applying blockchain technology to develop a decentralized consent management platform [DRFM18, ZNP15, YWJ+16].

2.7.1 Zyskind et al.

[ZNP15] presented a solution, specifically focused on mobile applications, that enable users to manage their applications personal data sharing consents, using blockchain technology while the data itself is stored off chain, on data silos. This solution was designed to be compatible with any public blockchain without revealing their client's personal information. Furthermore, applications who implement this solution on their systems, are responsible for safely storing their clients personal data. This framework, in order to ensure that users own and control their personal data, recognizes the users as the owners of the data and the services as guests with delegated permissions.

In order to achieve the aforementioned privacy, Zyskind et al. use *compound identities* – shared identity for two or more parties, where some parties (at least one) own the identity (owners), and the rest have restricted access to it (guests). Compound identities can be seen as a set of public keys (from the owners and guests) as well as a symmetric key used to encrypt (and decrypt) the data, guaranteeing that only the allowed parties have access to it. Since the data is stored off chain, only a hash to that same data is written on the blockchain alongside the permissions of the guest parties who can access that data.

Zyskind et al.'s solution accepts two new types of transactions: T_{access}, used for access control management; and T_{data}, for data storage and retrieval. T_{access} is used to configure the system access policies whenever a new users joins the network. T_{data} is used to send data collected by the user's phone to the blockchain. This data is then stored off-chain while a pointer to that same data is written on the public ledger (the pointer is a hash of the data). This hash allows accessing parties to verify that the data has not been tampered with.

Even though this solution tackles some of the main problems regarding data storage and users consent management, by enabling users to be fully aware of which data has been collected about them and by which service providers as well as how it is being used, this approach lacks when it comes to processing data. In order to process large volumes of data, there is a need to access a vast variety of permissions from a single client, this makes processing data from the the data silos very impractical. Another drawback in this solution is related to the GDPR right to be forgotten, as it is, once a service queries a piece of data, it could store that data and use it for future analysis without the user's consent. This means that untrustworthy parties that might, hold user information on their own storage can still profit from that data without the user's consent.

2.7.2 Healthcare Data Gateway

Yue et al.'s solution for health care consent management [YWJ+16], enables patients to own their personal data as well as control who has access to their data, without compromising the patients privacy. In order to tackle the problem of having data from the same patient spread among several institutions, this solution stores every EHR on the blockchain. Given the fact that data is stored on the blockchain and, in order to ensure the desired privacy to its users, all data that is written on the private blockchain cloud must be first encrypted.

The proposed solution, HDG – Healthcare Data Gateway, consist of a smartphone App which works as a gateway for its users to access the private blockchain cloud. This App is also responsible for evaluating all data accesses, both incoming and outgoing ones, and determining

whether or not a certain user is allowed to access the requested resource.

This solution is divided into three layers. First we have the storage layer that is responsible for providing a scalable, secure, highly available and independent storage service for healthcare data. Then there is a data management layer which is compromised of a set of individual HDGs that are connected and work independently from one another. The HDGs that are present in this layer work as a firewall, by executing access control policies, and as a database manager, by performing queries to facilitate the cloud accesses. Finally there is the data usage layer that represent all entities that use patients healthcare data stored on the private cloud.

HDG also tackles the problem of having different kinds of data (records, text, images, etc.) stored on the same system by designing a schema where each patient has a single table to store all their medical records. When querying for data, participants who fall under the data usage area, have to go directly to the patients table before specifying which type of data they which to retrieve.

This approach as it is, may lead to problematic situation on two different matters. Firstly, if the medical data records are stored on the blockchain then the information contained on those records could be accessed by anyone with a copy of that blockchain. Should this information be compromised by an adversary and its contents made public, other users could run supervised algorithms on that same data in order to profile users and related those same profiles which could eventually pin point to a certain patient medical records removing the provided privacy. Secondly, in order to store the medical records, and given the current increase in the amount of information about e-Health that is being registered every year, there could be incompatibilities between the block sizes and the amount of information that has to be store for each patient, which may lead to a point where each block can only hold information for a single patient, making it impractical to store data for a large community of users.

2.8 Summary

In this chapter, we started by explaining the base concepts of blockchain technology. We then continued, by introducing Ethereum – a global, open-source platform for decentralized applications. Afterwards, we introduced the concepts of enterprise blockchains and discussed an Ethereum client – Hyperledger Besu.

Later, we discussed confidentiality and privacy in blockchains and gave a brief introduction to the GDPR, which states how organizations should behave when dealing with their client's personal data. Finally, we conclude this chapter by explaining the concept of consent management and the reasons why it is important that companies and clients should be aware of its

existence. We also briefly discuss and analyse existing systems that already attempt to solve the consent management issue by applying blockchain technology.

Chapter 3

Universal Consent Management Platform

The Universal Consent Management Platform (UCMP) is a system that allows patients to decide which medical personal information they wish to share and with whom. In order to achieve this, each patient has to give his consent before any information is shared with a third party. This consent happens in the form of permissions which is then written in the Ethereum blockchain due to the strong immutability property that blockchains have to offer. UCMP implements two main approaches that allow patients to easily manage their sharing consents: à priori consent and consent on request. In order to ensure that no third party tampers with the patients' consents, each patient writes his sharing preferences directly on the blockchain on a personal smart contract.

3.1 Participants and Roles

Within the scope of this system there are several different participating actors where each is assigned a role based on the service they provide. UCMP has two main active roles and another passive role that does not directly interact with the platform. Each role represents a given group of participants. The roles are the following:

- **Patients or Data Owners** – individuals that attend to a medical facility due to a disease or clinical condition.
- **Healthcare Institutions or Data Holders** – hospitals, private clinics, medical centers (any institution that directly interact and attends to patients healthcare needs).
- **Data Consumers** – organizations and companies that might need patient related data to

improve a patient's treatment quality, to perform medical related research or simply use that data to exercise their business activities.

Patients and healthcare institutions are the two active roles while data consumers, even though being important for the use case, do not interact with UCMP directly. According to the task each participant has, a different role with a given set of permissions is applied. Firstly, there is the patients' role where they are only allowed to perform read and write operations on the blockchain as well as interact with the server in order to update their personal information stored on the database. Secondly, we have the healthcare institutions' role. This role has the same capabilities of the previous one but, in addition, any actor associated with the role may also query the server's database to retrieve patient related information. This is used to enable institutions to access specific medical records as long as the given patient has given proper authorization. Furthermore, any actor assigned to this role may request access to a given patient's medical records regardless of their sharing consent, based on the circumstances.

This last role gives doctors the ability to access medical records when their patients are in need of an urgent medical intervention and are unable to either communicate or give proper consent. When doing so, an event is written down on the blockchain and an alert is sent to the patient. If the patient is indeed in need of treatment and unable to alter their consent, doctors can simply access it. On the other hand, if the patient is not in need of treatment and they believe it was a mistake, and in order to prevent a privileges exploit, patients can simply revoke the access and the doctor who made the request is unable to access their medical records. It is important to note that every doctor will have their own set of private/public keys in order to sign the access requests to patients data. This way, the transactions written on the blockchain, on behalf of medical institutions, will have both the institution's signature as well as the signature of the requesting doctor.

3.2 UCMP operations

This Section describes and gives insights on UCMP's functionalities based on the supported system operations.

3.2.1 Login and authentication

All users have to create an account before using this platform. The account is used to authenticate both patients and healthcare institutions. While healthcare institutions have to login using a traditional user and password method, patients can take advantage of the Portuguese

citizen card features. This citizen card possesses both a *public private key pair*, that can be used to digitally sign messages or documents, as well as a digital certificate (which contains a corresponding public key) that can be used by patients to prove they are who they say they are. UCMP also takes advantage of another technology called *Chave móvel digital* [1] which can be used instead of the Portuguese citizen card to authenticate a given user remotely. Firstly, when creating an account, each patient has to decide which type of account they wish to create - Custodial or Non-Custodial:

- **Custodial** – the patient trusts our platform to generate, store and manage their Ethereum account keys.
- **Non-Custodial** – the patient is the sole responsible for managing their Ethereum account keys.

These two account types exist to facilitate the introduction of blockchain concepts to patients that wish to use UCMP to manage their consents but, due to lack of knowledge, are not familiar with how blockchains work or simply do not wish to safe keep their Ethereum key pair. Should a patient choose a custodial account, UCMP's server will generate a private/public key pair and directly store it on the database. Before storing the key pair on the database, the private key is encrypted with a secret known only to UCMP.

On the other hand, if a non-custodial account was chosen, patients have to make sure that they have an Ethereum account (EOA or Externally Owned Account that represents a normal Ethereum address). Afterwards, patients have to authenticate themselves before the Portuguese government using *Chave móvel digital* or their citizen card, in this process, after a successful authentication, each patient will receive/generate a token that can be used by third parties to verify their identity. To complete their registry in the management server, patients will be asked to create an account with an username and password. Finally, a message will be sent over to the server, this message contains three attachments:

- Firstly, some information contained within their citizen card (name, age, gender, id, NIF - *número de identificação fiscal*);
- Secondly, the token that they obtained during their authentication with Portuguese government which will be used by the server to verify their identity;
- Finally, if the account type was Non-Custodial, the public key that they are going to use in order to access the blockchain. On the other hand, if the account type was Custodial,

[1] https://www.autenticacao.gov.pt/a-chave-movel-digital

there is no need to send this information since the public key is already stored on the database.

The information contained within the citizen card is sent in order to enable healthcare organizations to query the database when they wish to access/request access to a given set of medical records. Similarly, the Ethereum public key of each patient has to be sent to the database in order to create a mapping between a given patient's public key and the information contained inside their citizen card. This enables healthcare organizations and companies to query for specific medical data, belonging to one of their customers. For instance, an insurance company could ask for medical records of one of their patients to verify that they indeed have a certain medical condition.

Should a patient with a non-custodial account type lose his private key, they also lose access to their smart contract, therefore making it impossible to further set consent permissions and alter previous consent decisions that have been made. This happens because the patient's smart contract that is deployed on the blockchain, when a new patient registers in UCMP, has a contract owner associated with it. Only the contract owner may alter the consent permissions within that smart contract. To tackle this problem, UCMP offers an option of setting a new private key as the new contract owner. In order to do so, a patient has to authenticate themselves on UCMP using their Portuguese citizen card or their chave móvel digital. Afterwards, and only if the authentication is successful, the patient is requested to provide a new Ethereum address which will be associated as the new contract owner of their smart contract. Finally UCMP sends a transaction to the blockchain requesting that the provided address becomes the new contract owner and removing the old address from the smart contract. After this point, only the new key pair is able to modify the contract permissions.

Figure 3.1 illustrates the data flow process during an access made by a patient, which can either be from a first time accessing patient or from an already registered patient.

3.2.2 Access Request to Personal Information

In order to obtain patients' medical records, a given data consumer has to perform a request to the data holder that is holding them. Afterwards, the data holder has to use UCMP in order to write an access request on each relevant data owner's smart contract. After the request is written on the blockchain, the next time a data owner signs in on UCMP, a new consent request will pop up. The data owner now has to decided whether or not to accept the sharing request that is being made. These requests shown to the data holder with the following parameters:

- **Data of Request** – the date at which the request has been written on the blockchain

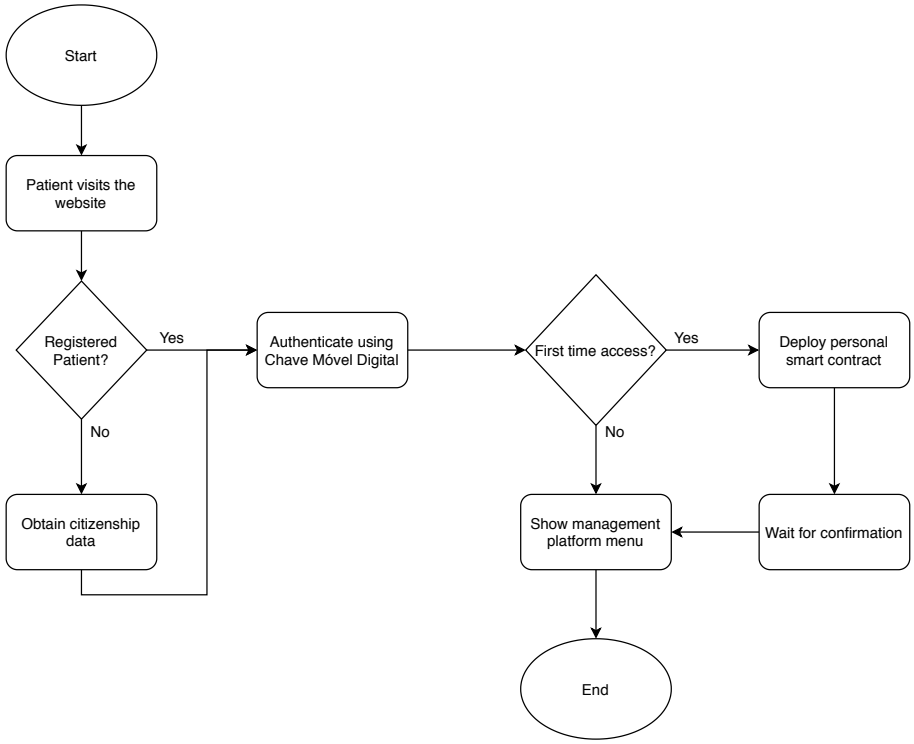

Figure 3.1: Flowchart representing the patient's client behaviour when attempting to interact with the system

- **Data Consumer** – the institution name with whom the data holder wishes to share data
- **Data Holder** – the Institution name that is holding medical personal data
- **Permissions** – a list of attributes that will be shared if the data holder gives consent
- **Description** – which purpose will those attributes be used for

Once a data owner makes his decision, they have to sign a transaction using their Ethereum account private key and send it to their clients' smart contract. The transaction contains meta data representing that decision.

Data holders may check, at anytime, the status of their request, where they can get information about how many data owners have already gave their consent as well as how many are still awaiting a response or have been declined. The data holder may also export a CSV file containing the data owners' ids that have given their sharing consent. With this information they can send the requested medical documents to the data consumer(s).

Since each data owner has their own smart contract and every request that is written on any smart contract has the data holder's and data consumer's name, should any data owner decide to revoke their sharing consent of a given request, it is possible to look up on the blockchain for who is holding what information about whom. This was designed with the purpose of notifying

data consumers, who hold a copy of a certain medical record from data owners who do not wish to share their personal information any longer. This enables data consumers to be notified right away and allow them to take the appropriate measures in order to delete those same records.

Figures 3.2 and 3.3 illustrates how the involved parties should behave when they receive a request to share medical records. The former represents the access protocol of a data holder while the latter represent the access protocol of a data owner.

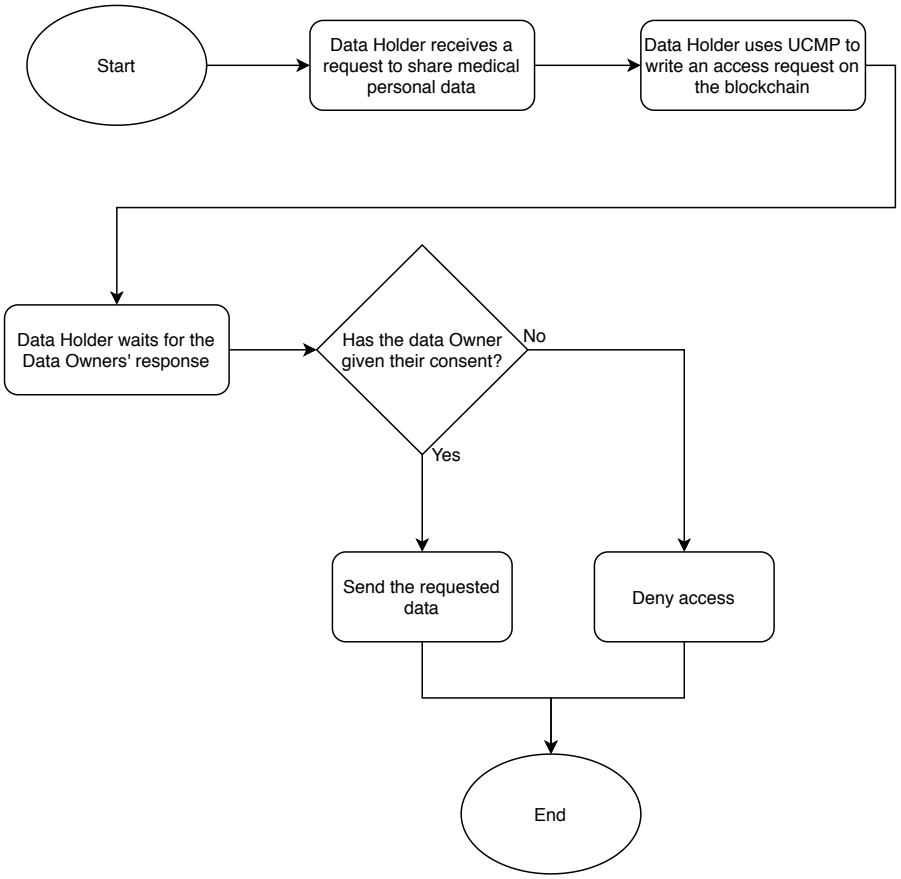

Figure 3.2: Flowchart representing the process behind the access protocol from a data holder perspective

3.3 Architecture

UCMP can be deployed on any blockchain that runs the EVM (this could be Ethereum main net, Ethereum Classic or Ropsten for instance) as well as on private blockchains based on clients that run EVM (such as Hyperledger Besu or Ganache for example). UCMP is composed of four main components:

- **Blockchain** – responsible for storing the patients' consents as well as keep a log of all changes that are performed.

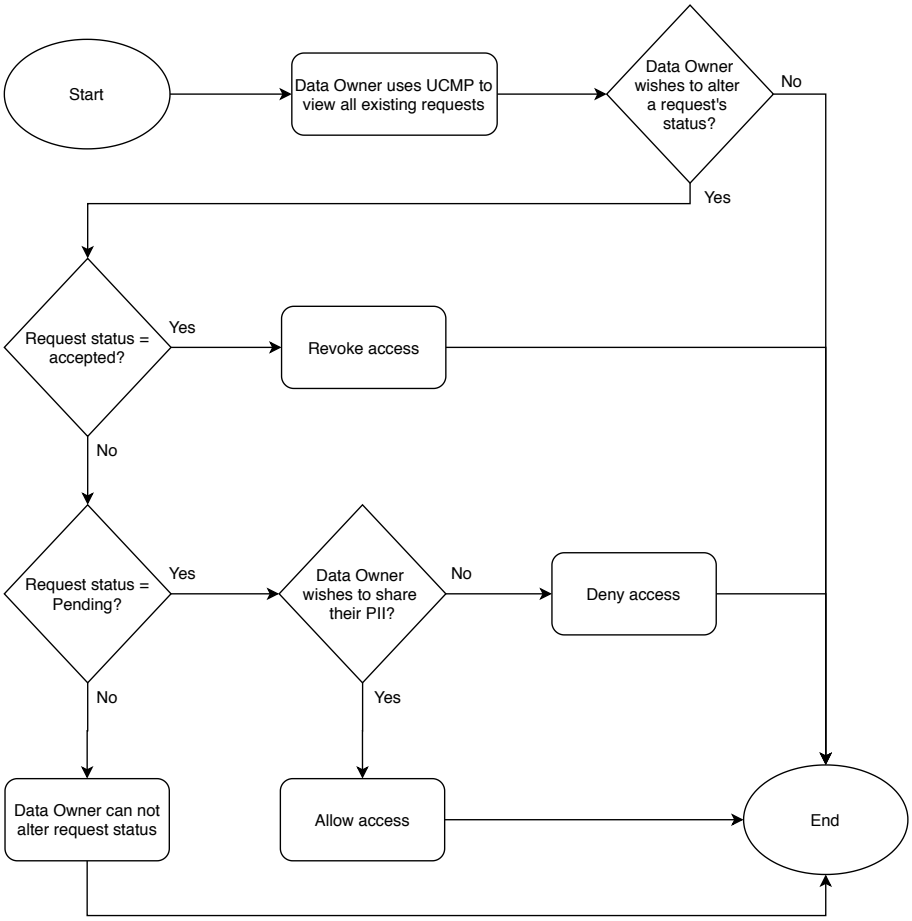

Figure 3.3: Flowchart representing the process behind the access protocol from a data owner perspective

- **Web App** – responsible for allowing each participant to interact with the blockchain, database and web server.
- **Database** – responsible for storing patient related personal information as well as a mapping that identifies each patient on the blockchain.
- **Web Server** – responsible for reading and writing information on the database. The Web Server acts as a bridge, connecting the Web App and the Database.

Even though UCMP helps institutions stay GDPR compliant by giving data holders a means to save their patients consents in an immutable form, each institution is still responsible for storing their patients medical data records as well as ensuring that they successfully reach the other party once an access is requested. Figure 3.4 illustrates UCMP's overall architecture.

The web app serves as the interface between the end users and the blockchain by reading the information written on the blockchain and then by presenting it in a simple manner. Through the web app, data holders may request and verify which patients have given their sharing consents while data owners are able to manage their consents.

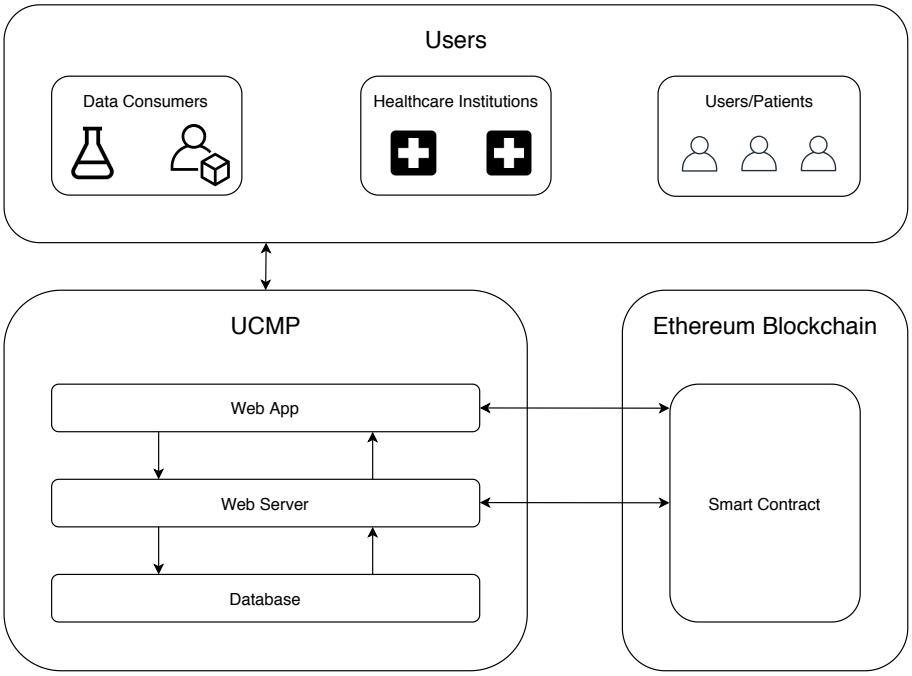

Figure 3.4: UCMP's Architecture

The web server facilitates institution to patient and institution to institution communications. Since institutions do not know the mapping between the account address used to sign the sharing consent and the patient's ID that a given consent belongs to, the web server is responsible for reading and sending that information from the database to the web app. With this approach the web server and the database act as a directory service which stores non medical patient related information.

For instance, if a data consumer wishes to request access to a single patient medical records, they only need to know that patient's ID. Afterwards the web server will retrieve that patient's account address from the existing mapping that is written on the database. Finally, the consent request written on the blockchain will be made using that patient's account address. It is important to note that before answering the request, the server is also responsible for retrieving the necessary information from the blockchain in order to stay compliant with the patients consent preferences. Every participating party can communicate with each other using the blockchain.

In order to correctly identify the same patient between different institutions, UCMP uses a global ID to reference each patient. The global ID used by our platform is the patient's NIF (*número de identificação fiscal*). The NIF is an ID that every Portuguese citizen has in order to be identified by the country's tax authority. Furthermore, each citizen can only have one NIF, making NIF an unique ID.

UCMP's architecture can be divided into three separate layers. The first layer is related

to the network and how each node exchanges messages with each other. The components of this layer are also responsible for following Ethereum's consensus protocol. The second layer is responsible for the correct execution of the defined smart contracts, where all transactions are executed, and for storing client related data. This layer consists of the blockchain itself. The access policy will also be implemented in this layer and it is compromised by the public address of each participant with permission to interact with the blockchain and by the addresses of existing nodes on the network. Participants on the network will contact these nodes whenever they wish to write a new transaction on the blockchain. Finally the third layer is composed by UCMP as well as the blockchain's interacting parties such as patients, healthcare institutions and data consumers.

UCMP takes advantage of Google's Backend-as-a-Service Firebase in order to implement several of its components. Backend-as-a-Service is a cloud service model in which developers outsource all the behind the scenes aspects of a web or mobile application so that they only have to write and maintain the frontend. Backend-as-a-Service vendors provide pre-written software for activities that take place on servers, such as user authentication, database management, remote updating, and push notifications (for mobile apps), as well as cloud storage and hosting.

3.3.1 Blockchain

There are a total of three main smart contracts on UCMP that are responsible for implementing the desired features:

- **Identity Relayer** – smart contract responsible for receiving, validating and rerouting a relayed transaction to the appropriate user's smart contract function.
- **Smart Consent Management** – smart contract responsible for storing all the information regarding data owners. This smart contract also acts as a template when deploying a new user's smart contract.
- **Smart Consent Management Factory** – smart contract responsible for deploying a new instance of the *Smart Consent Management* smart contract. When deploying a new smart contract the patient's address is passed down in order to set the owner of the smart contract. With this approach each patient is the sole responsible for altering the consent permissions of their smart contract.

Note that, upon first accessing our platform and given the fact that each patient is responsible for managing their own smart contract, every time a new patient registers in UCMP a new Smart Consent Management contract is deployed. This user's smart contract is an exact copy of the

Smart Consent Management. Furthermore, the Smart Consent Management smart contract inherits all the functions written on the Identity Relayer smart contract. Since the Identity relayer is responsible for validating and rerouting the relayed transactions (transactions that are sent by UCMP in the name of custodial users), this enables each smart contract to act independently from a central smart contract.

Each data owner smart contract keeps a record of all existing permissions of that data owner, as well as their current state. A permission can have four different states (Pending, Accepted, Denied, Revoked) and only the smart contract owner can alter the permission status. New permissions can be added to the smart contract by data holders. Data holders have to use UCMP in order to submit a new sharing request that is saved on the blockchain as a permission.

The identity relayer exists in order to allow Custodial users to freely interact with the blockchain without the need of having to pay ETH for their transactions. Since this type of user is supposed to have little to no knowledge about blockchains, it is safe to assume that they are also not familiar with crypto currencies and the need to pay for transactions. To tackle this issue we use a meta transaction in order to write the Custodial users sharing preferences on the blockchain.

A *meta transaction* is a type of transaction that is signed by a given key pair (in this case the user's key pair), but it is funded by a relayer. The relayer submits the transaction to the network as if they were the original sender and pay for the gas fee. The destination contract of the transaction, the Identity Relayer smart contract, can determine the original user, their intent and can process the contract call accordingly. In order for this to work, when a Custodial client wishes to write some data on the blockchain, they have to use UCMP to encode the appropriate function call, retrieve a valid nonce by performing a contract call to the client's contract, generate a hash representing the transaction and finally sign the generated hash with the client's private key.

Afterwards this signed transaction is sent to UCMP's backend server and submitted to the identity relayer. Once the identity relayer receives the transaction, it verifies the client's signature along side the encoded function call and if everything is successfully validated the encoded function call is executed by the client's smart contract.

Figures 3.5 and 3.6 illustrate the process behind assembling and digesting a meta transaction.

3.3.2 Web App

UCMP's web app can divided in three major components in accordance to their use:

- **Login and Registration menu** – menu responsible for retrieving personal data, such

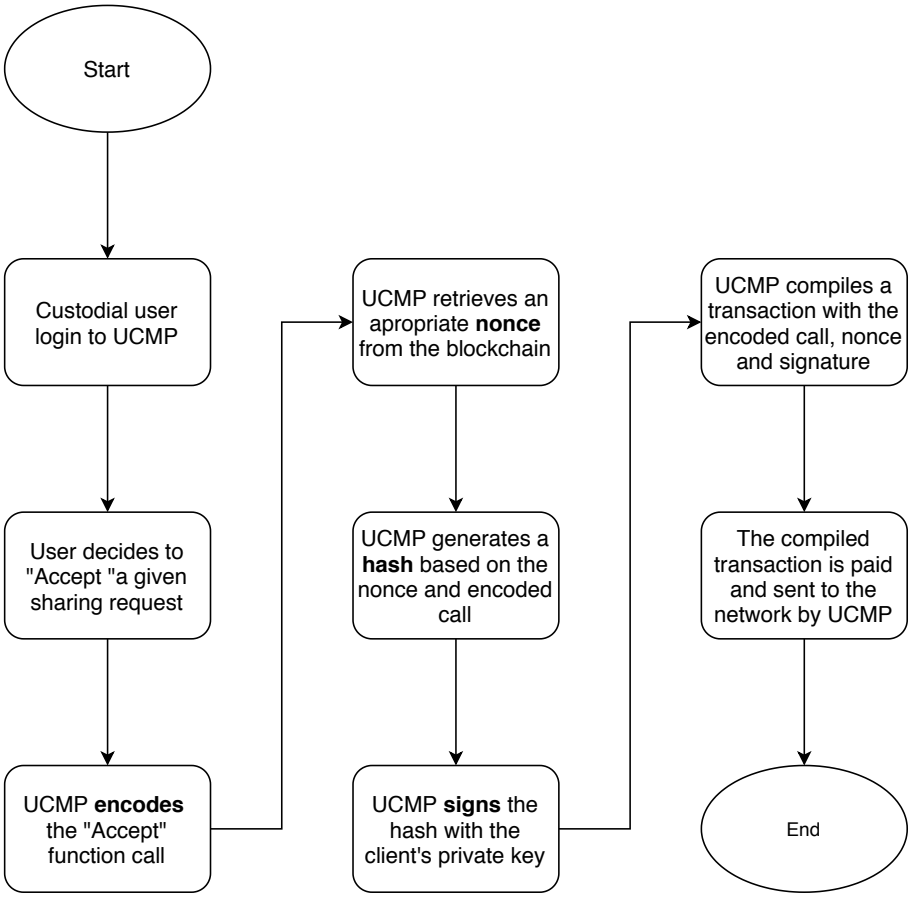

Figure 3.5: Flowchart representing the the process behind assembling a meta transaction

as NIF and name, from each client upon a registration on our platform. This personal data is then used to create a record on the database that allows UCMP to unequivocally identify its clients. Furthermore, this menu is also responsible authenticating the clients using firebase's authentication feature.

- **Data Owner menu** – menu that is exclusively accessed by the data owners. The menu allows its users to check the status of their permissions as well as modify any permission that they wish to.

- **Data Holder menu** – menu that is exclusively accessed by the data holders. On this menu data holders can perform new sharing requests to their patients as well as monitor already existing requests.

Data Sharing Request

UCMP only allows data holders and data owners to interact with the system. With this being said, data consumers may ask data holders to perform a sharing request using UCMP. This sharing request may contain either one or multiple attributes related with medical personal

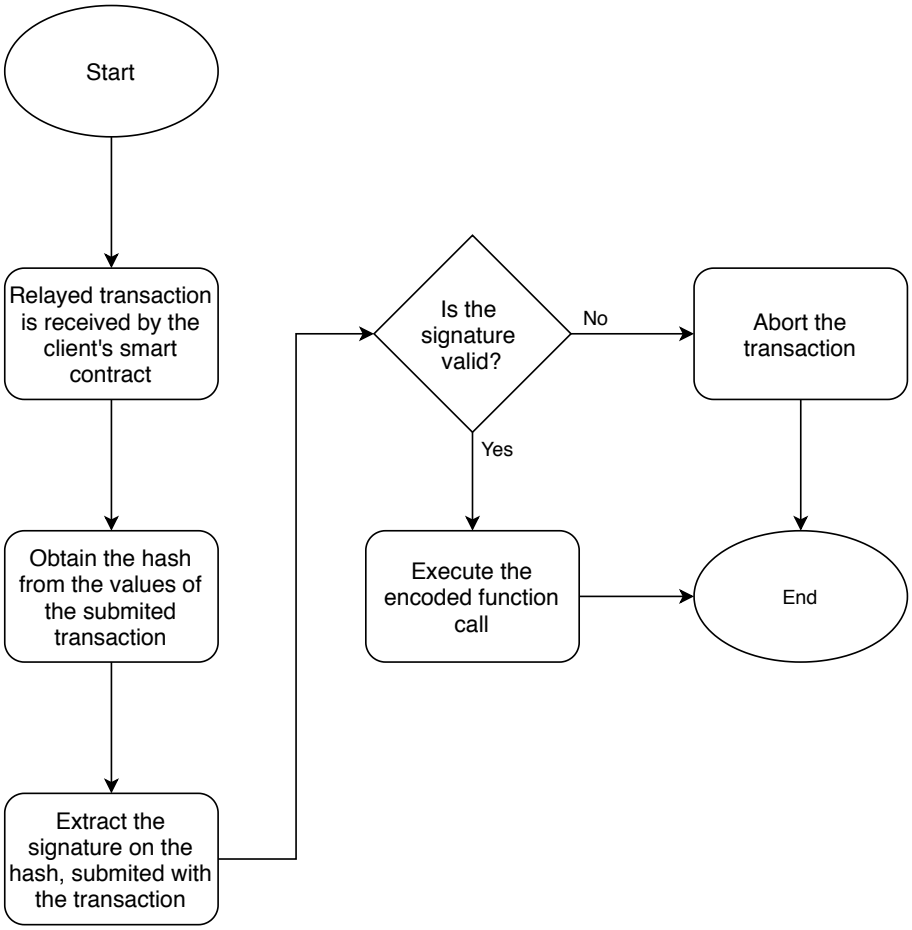

Figure 3.6: Flowchart representing the workflow of the identity relayer

information as well as being performed to multiple data owners simultaneously. On a sharing request data holders have to specify the following fields:

- **Data Consumer** – the institution(s)' name(s) with whom the data holder wishes to share data
- **Attributes** – a field representing all the attributes that are being shared (for example: "Age", "Name", "Exams", "Medical Prescriptions", etc.)
- **Purpose of Use** – the reason behind requesting access to those attributes (for example: "Research", "Write a Report" etc.)
- **Additional Comments** – Any additional information that the data holder wishes to share with the patient
- **Data Subjects** – the global id (patient's NIF) of all data subjects that the data holder wishes to request access (this field has to be submitted via a CSV file where each line contains a single patient's NIF).

Upon submitting a request, the information above is written on each data owner's smart contract. Furthermore, a general overview of the request is written on the database in order to create a record of all requests that have been performed using UCMP. Data holders can then check these permissions in order to know the overall status of the request.

When updating their set of permissions and, therefore, performing a write to the blockchain, data owners will also request UCMP to update its own database with the status change that has been performed. This way we can keep on the server's database an overview of the total number of permissions as well as their updated status. Data holders, by checking the overview of a request, can visualise how many permissions have been accepted, denied and which ones are still waiting for a clients' response. At any time they wish to, data holders may also export a CSV file containing a list of NIFs from all patients who have already accepted to share their personal information. On the request overview, data holders may also access the patients' NIF that have revoked the sharing consent. This way they are able to take the appropriate measures in order to guarantee that a client's right to be forgotten is successfully met.

Managing permissions

Although UCMP is a consent management platform, when we wish to refer to any given consent we often use the term permission. When a data holder proceeds to use UCMP in order to ask for a sharing consent this is done by writing a transaction to the blockchain requesting the patient's permission to share a given piece of personal data. Therefore the term permission is often used in place of the term consent.

Regarding permission management, UCMP offers its clients two different approaches. Clients can choose, from a given set of permissions, certain fields that they are willing to share. Furthermore, patients can also choose with which data consumers' they are willing to share their data based on the area of expertise. For instance, one client may decide that he wishes to share his name, age and his medical records with any data consumer that is related with medical research. In this particular case, when a data holder asks for the client's sharing consent, and if the data consumer is a company that performs medical research, the data consumer will get immediate access to the patients personal data. In this process, two writes are performed to the blockchain. One write is performed when the client decides his sharing preferences, while the other happens when the data consumer gets access to the shared medical personal information. In the latter, an event is issued on the blockchain, these events can then be checked at any time, working as a history of past states of the blockchain.

The second permission management approach allows patients to receive sharing requests.

These requests are sent from data holders who wish to share their patients personal information. When receiving one of these requests, patients can choose to either accept or deny the sharing request. Either option will lead to a write on the blockchain, altering the state of a permission, as mentioned on section 3.3.1. With this approach patients can decide, on the spot, whether or not they wish to share their personal information with that given third party. Furthermore, if a patient decided to accept the sharing request, he can always revoke the request at a later date. At the moment of making a choice, UCMP gives patients information regarding:

- **Date of Request** – the date at which the request was made/written on the blockchain
- **Data Consumer** – the institution's name with whom the data holder wishes to share data
- **Data Holder** – the institution's name who holds the given personal data
- **Attributes** – a field representing all the attributes that are being shared
- **Purpose of Use** – the reason behind requesting access to those attributes
- **Additional Comments** – Any additional information that the data holder wishes to share with the patient

When data holders perform a sharing request, UCMP will first check if the involved patients have already given permission to share the requested personal information with that given data consumer. If this is not the case, UCMP will then give data holders the opportunity to write a sharing request on each of the relevant patients' smart contract.

3.3.3 Database

UCMP takes advantage of Google's Cloud Firestore, that provides the app's database. Cloud Firestore is an non-relational database (NoSQL) that stores data in documents that contain fields mapping to values. Those documents are stored in collections, which can be seen as containers for those same documents. UCMP's database has three collections:

- **Users** – This collection contains all the necessary information regarding the UCMP's users.
- **Requests** – This collection contains information regarding every sharing request that is performed by the Data Holders.
- **Vaults** – This collection contains the public and encrypted private key of custodial users. This key pair is used to access the blockchain.

Users Collection

The user's collection has a record for every client that uses UCMP. Each record is represented by an unique identifier that is automatically generated by Firebase when a client first registers to UCMP. In addition to the unique identifier each record also stores the following fields:

- **contractAddress** – the contract address is saved in order to be provided to data holders when they wish to perform a sharing request to a given client.
- **email** – the email is saved in order to send notifications to the client. One possible notification could be when the user first registers to UCMP. Another notification example could be when a client needs to change ownership of their smart contract, should they lose their key pair.
- **isCustodial** – this field represents a bool and is used by UCMP to know whether or not a given client is custodial or non custodial. This information is then used by the system to determined which method to use during transaction signing.
- **name** – the name which is stored in this field is the one provided by the client during the registration process. The name is then used when sending emails. After a user logs in to UCMP the name is also displayed in a sidebar so that clients know which user is logged in.
- **nif** – In this field there are two values stored. One is the NIF of a given client and the other is a hash of that same NIF. The hash is stored in order to reduce the number of function calls to Firebase's cloud functions (these cloud functions will be presented and discussed further ahead on Section 3.3.4). This hash is generated by converting the *nif* (number) into a string, add salt and then interact the final string multiple times with a hash algorithm.
- **type** – this field stores which type of user has been registered in UCMP – either a data holder or a data owner.
- **createdAt** – this field is a timestamp that marks the moment a new user is created. It can be used for audit purposes and to perform checks on the system consistency.
- **updatedAt** – this field is a timestamp that represents the last time a user record received an update. One possible update that would trigger this field could be a change performed on a user's email address. This field can also be used for audit and consistency purposes.

Requests Collection

Request's collection is composed of every sharing request performed by the data consumers. In this collection each request gets attributed a random ID which is generated by Firebase, similarly to the clients' ID. Each record on this collection has to contain all the information regarding a request that is performed by a data holder. In order to achieve this purpose the following fields were created on the Cloud Firestore database:

- **attributes** – this field contains a list of all attributes that were written on each patient's smart contract.
- **dataConsumer** – this field refers to the data consumer that requested access to the attributes mentioned on the field above.
- **dataHolder** – this field stores data regarding the data holder that placed the request. It stores the name of the data holder as well as the unique ID that is generated by Firebase, when the data holder registers to UCMP.
- **purposeOfUse** – this field stores the reason why a given data holder performed a sharing request.
- **accepted** – this field stores a list of NIF hashes from all clients who have already given permission to share their medical personal information, which was mentioned on the attributes field.
- **denied** – similarly to the previous field, a list of NIF hashes from clients who have denied the sharing request is stored on this field.
- **pending** – this field follows the same approach as the two previous ones but in this case it stores information regarding clients who have a pending request.
- **revoked** – identically to the last three fields, revoked stores the hash of clients who have revoked their sharing consent.
- **createdAt** – this field stores the timestamp at which the request was submitted by the data holder. It also has the same purpose as similar fields on different collections – to server as consistency checks.
- **updatedAt** – this field also stores a timestamp, except in this case, the timestamp is updated every time a write is performed on the request. In other words, when the database information is update, this field also receives an update on the timestamp. When a client

alters the state of their own permission request, for instance, they accept the sharing request their NIF hash will go from the field *pending* to the field *accepted*, at this time a write is performed on the database and the *updatedAt* field is filled with the timestamp of that write operation.

The request collection was created in order to store relevant information, that can be used to show data holders an overview of the status of each request they perform. Although most of this information is also written on the blockchain, and given the fact that each request may have thousands of patients involved, reading the data from several smart contracts at the same time, would simply take too much time to become practical. To tackle this issue, not only is the request information written on the database (such as attributes, purpose of use, data consumer and data holder) but also four fields were created to store the different possible status of a request. These fields are automatically updated by UCMP when a client accepts, denies or revokes a request.

Since UCMP uses the Ethereum main net, a client with enough knowledge about blockchains, transactions and meta data may also send transactions directly to their smart contract without using UCMP's web app. To prevent the inconsistencies that this action may result in, specially on the requests collection at the database, UCMP performs daily checks on the information written on the database. This happens during the times with less traffic so that the server resources do not get consumed all at the same time, possibly leaving the platform with a higher response time to its clients requests. With this being said, UCMP has a cloud function that executes everyday during night time (where it is estimated to have fewer traffic). This cloud function checks each request individually and verifies if the information saved on the database matches the information written on the network. If there is a client that has indeed sent a transaction to the smart contract without using UCMP then the cloud function updates the database with the change that was performed on the blockchain.

A similar problem may also occur when a data holder wishes to export the CSV file containing all the patient's NIFs who have already accepted the sharing request. In this situation, if a client has sent a transaction without using UCMP and the cloud function has not yet been executed, there could be a missing NIF on the file (if the patient has decided to accept the sharing request) or more problematic, there could be an extra NIF on the file (if the patient revokes an already accepted request). In order to address this issue, when a data holder decides to export the CSV file, the previously mentioned cloud function is called so that the file containing the patients NIFs is up to date with the information written on the blockchain.

Vaults Collection

The vaults collection is used to store the key pair of UCMP's custodial users. Since this type of user is assumed to have little to no knowledge about blockchains and how they work, we decided to store the key pair on UCMP's side. With this approach we hope to reduce the barriers between blockchain technology and end users.

In order to reduce the information saved on each record, this collection associates the ID of the custodial user with their own record. In other words, each record is being identified on the database by the user's ID that they belong to, avoiding the need to write any additional information on the record itself other than the key pair that is being stored. With this being said there are only two fields on each record:

- **accountAddress** – this field contains the client's public key that also serves as the account address.
- **encryptedPrivateKey** – this field contains the client's private key. This private key has been encrypted before is was sent to the database.

The following section 3.3.2 gives more insights on how this key pair is generated as well as the encryption performed on the private key.

Database Rules

Firebase Security Rules for Cloud Storage leverages a superset of the Common Expression Language (CEL) that relies on match and allow statements that set a condition for access at a defined path. These rules allow the specification of path based permissions that restrict Google Cloud Storage requests to a certain user or type of users. Firebase Security Rules for Cloud Storage ties in to Firebase Authentication for user based security. Furthermore, Firebase Security Rules can also read the fields used in Cloud Storage and use it to grant role-based access, for example.

Each database collection in UCMP has its own set of rules in order to provide strict access control to the information that is written there. The rules specify which user or type of user can read and write on any given record of each collection. Although every database has its own set of rules, there is a global rule that applies to all collections - only authenticated users can access the database. This means that UCMP's web app can only retrieve information from the database if the user who is performing the request has logged in beforehand.

When a user signs in UCMP, Firebase Authentication generates a token that poses as the users identity within Firebase products. After a successful sign in, it is possible to access the

user's basic profile information, as well as control the user's access to data stored in other Firebase products. We can also use the provided authentication token to verify the identity of users in our own backend services.

The code bellow represents the security rules developed for cloud storage.

```
service cloud.firestore {
    match /databases/{database}/documents {
        match /vaults/{uid} {
          allow read, write: if false
        }
        match /users/{uid} {
          allow read, write: if request.auth != null
                             && request.auth.uid == uid
        }
        match /requests/{uid} {
          allow read, write: if request.auth != null
                             && request.auth.uid == uid
        }
    }
}
```

The collection with the most sensitive information and therefore the most restrictive rules is the vaults collection. Given the fact that it contains the key pair for every custodial user, only UCMP's cloud functions can read or write information on this collection, no user, regardless of being a data owner or a data holder, can access the data on this collection.

Considering that, the requests collection stores information regarding data holders' request, the rules developed for this collection only allow a user which type is *data holder* to access the collection. Additionally, in order to guarantee that the patients privacy is ensured only the data holder who performed the request has read and write permissions over that same request. Similarly, the users collection can only be access by the the own user and Firebase's cloud functions. By doing so, it is possible to restrict database access only to the necessary parties.

3.3.4 Web server

UCMP uses Firebase's cloud functions in order to implement all the web server backend logic needed to write and access data on both the database and the blockchain. These functions

are executed on Google's servers when a function call is performed or when a certain event occurs. One possible event that triggers a cloud function execution could be, for example, when a document is created or updated in the database. Similarly, a cloud function could also be trigger in response to the creation and deletion of Firebase user accounts. Furthermore, if there are no events that suit our needs, Firebase also allows cloud functions to be scheduled and then executed at specified times. Finally, the method that is more commonly used to trigger cloud functions is calling the function directly from our app using the provided Firebase API. The vast majority of UCMP cloud functions are trigger by a function call leaving only a few to be trigger by events. Among UCMP's cloud functions the most important are:

- **createEthereumAccount** – function responsible for generating an Ethereum key pair for custodial users. The newly generated key pair is then sent to a collection in the database where it is stored. Before storing the key pair in the Cloud Firestore database, the private key is encrypted with a secret based on the users' unique ID generated by UCMP plus a given salt. This way we can ensure that two different private keys are not encrypted using the same password.

- **deployContract** – function responsible for deploying a new Smart Consent Management contract (user's smart contract) on the blockchain. In order to deploy a new smart contract that belongs to a single user, the web app has to pass down the user's public key when calling this function. Then, the public key is used to call the Smart Consent Management Factory smart contract, which is responsible for deploying a new contract on the network. Afterwards, the function writes on the user's collection in the database, the public address of the newly deployed smart contract.

- **signEthereumMessage** – function used to sign a transaction on regard of a custodial user. The function execution is triggered with a call request, when a custodial user wants to modify their sharing consent on a given permission. For that purpose, the user has to submit a transaction to the blockchain which requires the cloud function to access the database, so that an appropriate key pair can be retrieved. Afterwards it decrypts the private key in order to sign the transaction that is being sent on behalf of the user and finally returns the signed message to the user.

- **relayTransaction** – function responsible for sending a signed transaction to the database on behalf of a user. Before submitting a relayed transaction, the cloud function has to perform a contract call in order to obtain a valid nonce. The nonce is used to prevent replay attacks, where the same message is sent more than one time to the network.

- **checkExistingClients** – function called when a data holder wishes to submit a new consent request to the data owners. This function receives an array of user's NIFs and then checks the database for matches. The function returns to the data holder a list of Ethereum public addresses belonging to the matches found. In other words, the function checks which NIFs belong to existing UCMP's users and returns their public addresses so that a request can be written on each clients' smart contract.

3.4 Implementation

UCMP's components and functionalities were implemented in a prototype. For the smart contracts, Solidity was used as the programming language while the web app was developed using Vue.js. Solidity is an object-oriented, high-level language for implementing smart contracts and is designed to target the Ethereum Virtual Machine (EVM). On the other hand, Vue.js is a progressive framework for building user interfaces. Vue is designed from the ground up to be incrementally adoptable and has its core library focused on the view layer only.

UCMP's smart contracts were not deployed directly on the Ethereum main net given the fact that Ether was required to send transactions to the blockchain. Instead, we decided to use Ropsten test network considering that Ropsten, from all the available test networks, is the most similar to Ethereum due to the fact that both use a proof of work consensus mechanism. In addition to deploying on Ropsten network we also deployed on Rinkeby test network for testing purposes. With these two deployments we were able to compare network availability as well as latency between requests.

During the practical implementation of the proposed solution, we also took advantage of a few libraries and tools to ease both the buildup of the smart contracts as well as the buildup of the web app and cloud functions. The main libraries and tools used were the following:

- **Ganache** – Ganache is a personal blockchain for rapid Ethereum and Corda distributed application development. Ganache was used across the entire development cycle, enabling us to develop, deploy, and test our dApps in a safe and deterministic environment. Although Ganache provides its users with a desktop application – Ganache UI, we decided to use the command line tool – Ganache-cli in order to simplify the development stage of our proposed solution. The main reason behind choosing ganache-cli over Ganhache UI was due to the fact that we just needed a local blockchain to test and deploy smart contracts, the extra features that Ganache UI had to offer were not mandatory and would only add complexity levels to the developed prototype.

- **Truffle** – Truffle provides a development environment, testing framework and asset pipeline for blockchains using the Ethereum Virtual Machine. Truffle was responsible for compiling, deploying and testing the developed smart contracts on both ganache and Ropsten test network.

- **OpenZeppelin** – For the smart contracts, we used OpenZeppelin, a library for secure smart contract development. This library provided our system with cryptographic functions that were used to verify signatures, decode function calls and produce and verify hashes. This library was most used when developing the Identity Relayer smart contract, given the fact that this smart contract was responsible for validating transactions to ensure that each client was the sole responsible for managing their own smart contract. From the eligible hash algorithms accepted by National Institute of Standarts and Technology, we decided to use SHA3-256.

- **Firebase** – As mentioned on the previous sections, Firebase was used to provide Authentication, the Cloud Firestore database, and cloud functions, that supply UCMP with a simple user-password authentication, storage and backend services respectively.

- **Infura** – In order to connect UCMP to the blockchain, the developed cloud functions use Infura as a service provider. Infura is a hosted Ethereum node cluster that allows its users to run their applications without the need to set up and manage their own Ethereum node or wallet. For that end, Infura provides its users with the necessary tools and infrastructure to perform simple connections to the Ethereum blockchain.

- **MetaMask** – Metamask is a wallet which is available as a browser extension and as a mobile app. MetaMask provides a key vault, secure login, token wallet, and token exchange and was responsible storing an Ethereum key pair and for signing and sending transaction to both Ganache generated blockchain and Ropsten test network.

- **Web3** – web3.js is a collection of libraries that allow developers to interact with a local or remote Ethereum node using HTTP, IPC or WebSocket. Web3 was used during the development of UCMP's web app, being responsible for connecting UCMP's users and cloud functions to a service provider such as Infura and Metamask as well as connecting UCMP to ganache and Ropsten test network. Furthermore, web3 was also used by a cloud function to generate the Ethereum key pairs of custodial users as well as generate the hashes that would be attached to the transactions sent by clients. The hash algorithm used by web3 was SHA3-256 since it had to match the algorithm used by OpenZeppelin to enable validation. If a different algorithm was used, the hashes produced over the

same piece of data would not match and no transaction would be accepted by the smart contracts.

- **Vuetify** – In addition to Vue, Vuetify was also used to facilitate the interfaces' building process. Vuetify is a Vue UI library that provides reusable components for web and mobile applications. Vuetify material design framework is built on top of Vue.js and was developed exactly according to material design specification.

- **CryptoJS** – CryptoJS is a growing collection of standard and secure cryptographic algorithms implemented in JavaScript using best practices and patterns. CryptoJS was used to encrypt the web3 generated private keys. The encryption algorithm of choice was AES-256, given that we needed a symmetric encryption algorithm approved by National Institute of Standarts and Technology (NIST) and AES was one of those. The 256 bit version was chosen due to CryptoJS functionalities where it generates a 256-bit key from a given passphrase. This passphrase is created by UCMP by performing a hash to the string resultant from merging a client's UID with a given salt. The goal of this process is to ensure that UCMP can generate unique passphrases for each user.

When implementing the cloud functions, there was a need to provide some critical information to those same functions. This critical information is: (1) the private key used by UCMP to write on the blockchain on behalf of custodial users, (2) the Infura provider key used to access Infura's services and (3) the salt used in conjunction with the user's ID when creating the passphrase used when encrypting and decrypting custodial users' private key. There were two possible solutions for this problem, the first solution was to simply save this critical information within the cloud functions themselves as global variables, the other solution was to use Firebase to store this critical information as environment variables. We decided to follow the second approach, based on the security that was offered by firebase services.

By simply storing critical information as global variables within cloud functions, we could restrict client access, since only an administrator can interact with cloud functions. Nonetheless, if the developed code was stolen by an attacker, he then would have total access to critical information. With this being said, using Firebase to store the critical information was the best approach given the fact that we are able to store information, at anytime, on a configuration file and then, the developed cloud functions can access this same file during runtime. Bellow there is an example of the code produced in order to store and access critical information using Firebase services. In order to set the configuration file, we had to use Firebase API and the following command line instructions:

```
firebase functions:config:set secretsauce="MYSECRETSAUCE"
firebase functions:config:set infura.key="RANDOMAPIKEY"
```

Then after setting the configuration file all we had to do was access that same file inside a cloud function. The following snippet represents the code used to access the critical information.

```
const functions = require('firebase-functions')

...

const provider =
      `https://rinkeby.infura.io/v3/${functions.config().infura.key}`

const secretSauce = functions.config().vault.secretsauce
```

For security reasons, Firebase saves this configuration file separately from the main database files, this means that even if a database breach should happen the critical information would still remain confidential. This second approach also solves another problem, given the fact that the critical information is no longer written directly on the code, should an attacker manage to get access to UCMP's code, they would still have no access to the critical information.

3.5 Summary

This chapter presented the Universal Consent Management Platform architecture and implementation. In Section 3.1 we started by describing and giving insight on the platform's participants as well as the roles they had within UCMP. Afterwards, in Section 3.2 we defined the base operations that UCMP had to perform in order to provide its clients with appropriate confidentiality measures. Additionally, we also described the protocols behind the base operations performed by UCMP. Section 3.3 starts by giving a general overview of UCMP's architecture, then we give a more detailed description of the smart contracts used, the web app architecture as well as the database and web server developed. Finally, in Section 3.4, we finish this chapter by presenting tools and libraries that were used to implement UCMP, along with a brief discussion of the problems we faced during its implementation.

Chapter 4

Results

This chapter presents the evaluation of Universal Consent Management Platform. We start by describing the evaluation methodology as well as give insights on several decisions that were made during UCMP implementation and testing. We further describe the approach performed in order to benchmark our platform as well as the issues that arise during the testing phase. We then continue to present and discuss the experimental results obtained during the tests.

4.1 Evaluation Methodology

In this section we will measure the performance of UCMP implementation. We focus the evaluation on the cost of submitting transactions to the blockchain as well as the latency of both the platform (web app and database) and the blockchain.

UCMP is prepared to have its smart contracts deployed on Ethereum, however, given the fact that Ethereum charges Ether to submit transactions and execute smart contracts, the performed tests and evaluation were made using test networks or simply testnets. As mentioned on Section 2.2, these transaction fees exist in order to prevent ill-intentioned users from consuming all of the network resources.

Ethereum test networks allow developers to design and test new decentralized applications without the need to pay Ether to execute smart contracts. Instead, developers can received free Ether, that is only valid within a given test network, and use that same Ether to execute their smart contracts. This Ether can be received via *faucets* - websites that will give crypto currencies in exchange for viewing adds of performing simple tasks. There are four main test networks that can be used to deploy decentralized applications:

- **Ropsten** - Ropsten has a proof-of-work consensus algorithm. Given the fact that it has the same consensus algorithm as Ethereum, it best reproduces the current production

environment of the live mainnet. Ropsten can also be used by all Ethereum clients which makes it ideal for test purposes. In order to receive Ether for this test network one can either place a request on a faucet or mined the Ether directly from the test network. On Ropsten, a new block is usually mined around every 30 seconds.

- **Rinkeby** - Rinkeby is a proof-of-authority (PoA) consensus algorithm. Ether on this test network can not be mined, meaning that Ether supply is controlled by trusted parties. This approach produces a blockchain that is immune to spam attacks. Since the Ether supply is controlled by third parties, each user only gets a small amount of Ether to perform their deploys and tests. Should a user decide to spam the network with random requests, this Ether will quickly run out avoiding a spam attack. Afterwards, the trusted third parties simply have to refuse all requests from that user asking to receive more Ether. Since this is a PoA network, Rinkeby does not fully reproduce the current production environment as Ropsten does. On this test network, a new block is usually mined around every 15 seconds.

- **Kovan** - Kovan is very similar to Rinkeby test network. Like Rinkeby, Kovan also has a proof-of-authority consensus algorithm, users can only receive Ether by using a faucet and for the same reason this test network is also immune to spam attacks. On the other hand, a new block is mined around every 4 seconds on the Kovan test network.

- **Görli** - Similarly to Rinkeby and Kovan, Görli also uses a proof-of-authority consensus algorithm. On this test network Ether can by acquired by using a one-way throttled bridge from any of the other three test networks to Görli. Given the fact that Ether can be obtained from the other three test network by performing a transaction, Görli is not immune to spam attacks because a user can mine Ether on Ropsten and then send it over to Görli. The main advantage of Görli is its stable implementation that is supported but all Ethereum clients. On this test network, a new block is usually mined around every 15 seconds.

Given the available test networks we decided to use Ropsten as our main testnet during the implementation phase, due to the fact to its strong resemblance to the current production environment Ethereum. Nonetheless, when performing tests and measuring the appropriate metrics we decided to use both Ropsten and Rinkeby to have a means of comparison between two different consensus protocols.

UCMP was evaluated by benchmarking the performance of its requests. These requests include transactions to the above mentioned blockchains and requests to our database and

web server. With this in mind, we decided to use two indicators to evaluate the blockchain deployment and one indicators to evaluate our database and web server requests. The indicators used were the following:

- **Latency** - This metric was used for both the blockchain and the database requests. On the former, we tested the response time of a request to a client's smart contract, while on the latter we tested the response time of a read operation on the database. The time elapsed for the database request also includes a cloud function call, this cloud function is responsible for receiving client requests and then access the database in order to retrieve the necessary information.
- **Transaction Cost** - This metric was only applied to the blockchain requests, where we measure the cost to send various transaction to a client's smart contract. The cost refers to the computational cost needed to perform each request. By doing so, we can achieve the average cost of a transaction for each client, enabling us to predict and inform the clients with an estimate cost for each transaction they are about to perform.

4.2 Experimental Results

In this section we cover the results obtained while performing several tests to our system. In order to achieve more accurate results we performed 40 transaction submissions for every test that was made. Notice that we use the term "direct submission" many times throughout this chapter, with this term we are referring to the action of simply sending an already signed transaction to the blockchain without any further computations.

4.2.1 Latency

During our testing phase, we decided to measure and compare the time it takes for our system to perform several core actions. Among these actions we have direct transaction submission from a client's browser, UCMP transaction submission on behalf of a client and UCMP custodial versus non-custodial users submission times.

Direct transaction submission

Initially, and in order to obtain some base line results that could be use as a baseline throughout the testing phase, we decided to measure the time it took to submit a transaction to the blockchain directly from a client's wallet provider (without using UCMP). During this test, we measured the submission time for two blockchains: (1) Ropsten test network and (2) Rinkeby

test network. The average time to mine a new block for each of these blockchains is 30 seconds and 15 second respectively.

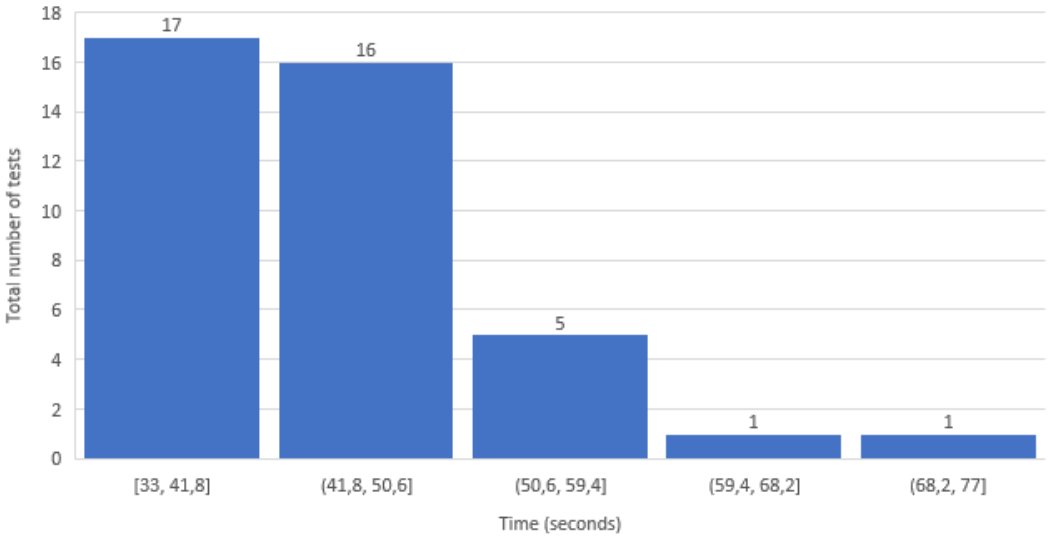

Figure 4.1: Graph representing the obtained values for latency when submitting transactions to the Ropsten test network

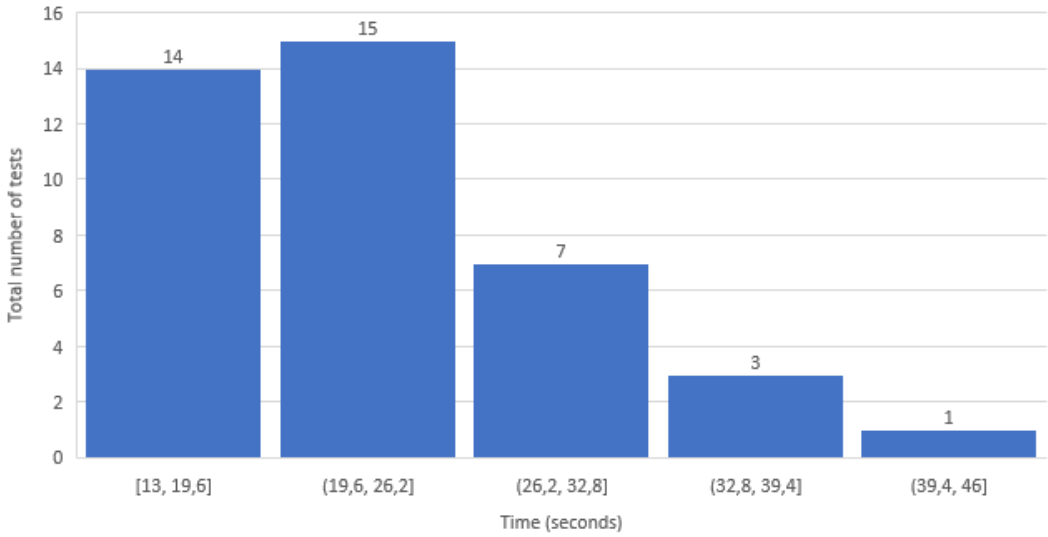

Figure 4.2: Graph representing the obtained values for latency when submitting transactions to the Rinkeby test network

Figure 4.1 and Figure 4.2 illustrate the test results regarding Ropsten and Rinkeby test networks. In the results, we have grouped the transaction performed into time periods based on the time it took for a transaction to complete. By observing the value distribution of both figures, we can deduce that the average time it took for a transaction to complete is higher than the average time it takes to mine a block (it takes around 15 seconds for Rinkeby and 30 seconds for Ropsten to mine a new block). The average time, to submit a transaction, that we obtained in our tests is around 22,94 seconds for Rinkeby test network and around 43,77 seconds

for Ropsten test network.

This transaction time is higher than the average time to mine a block due to the fact that when a transaction is submited it takes the time for finishing mining the current block plus the average time to mine the block that our transaction is going to be in. Therefore, the obtained average time to submit a transaction is within expectations (the expected value should be around the average time to mine a block plus half the average time to mine a block).

For instance, if we take Rinkeby as an example, the expected average time to submit a transaction to this blockchain would be around: 15 seconds to mine the block our transactions is going to be in, plus 7.5 seconds to mine the remaining of the current block that the miner is working at. Since the miner can receive our transaction at anytime when mining a block, on average, it will take half the expected time to finish mining the current block.

On the other hand, we can also observe in Figure 4.1 and Figure 4.2 that there were a few transactions that took more than double the average time to mine a block. These cases could have happen due to the fact that the transactions submited had the lowest possible gas fee, and, for that reason, were not chosen by the miner to be part of the next block. Another possible explanation for this event could be related to the network, if our transaction got delayed due to high traffic, the miner would have not received the transaction right away and therefore when mining the second block where our transaction is supposed to be in, it would be missing. This would cause our transaction to show up only on a third or higher block.

UCMP

Afterwards, we decided to measure the extra time needed to complete a transaction when submitting it using UCMP. This would be the case for a custodial user. On this test, we had to slightly modify UCMP's implementation in order to measure only the submission time of a transaction when using UCMP. To do so, the client had to sign the transaction before using UCMP as the submission medium. Recall that, the cloud function used to submit the transaction to the blockchain, has to first access the database to retrieve the appropriate private key, then sign the transaction on the client's behalf and finally send that same transaction to the blockchain. For this test, this overhead was undesirable, therefore, the cloud function was modified in order to simply receive an already signed transaction and submit it to the blockchain. We also decided to use Rinkeby test network for this test because it had a lower average time to mine a new block when compared to Ropsten test network. Section 4.3 gives more insights on the reason why Rinkeby was chosen over Ropsten.

Figure 4.3 illustrates the overhead generated by UCMP when submitting an already signed

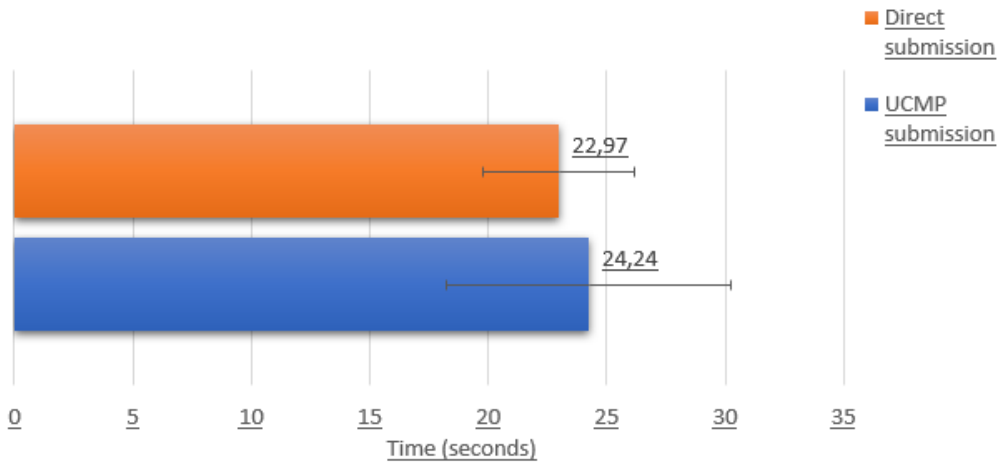

Figure 4.3: Graph representing the obtained values for UCMP overhead when submitting transactions to Rinkeby test network

transaction, this overhead includes only the cloud function call. The measured value indicates that it takes approximately 1,27 extra seconds to submit a transaction when using UCMP. This result meets our expectation by showing that UCMP adds only a small delay when submitting transactions on behalf of the clients. It is important to note that, this overhead takes into consideration the network delay when communicating with Google's Firebase service in addition to the computational time required to execute the cloud function. One further aspect that may also contribute for this delay is the fact that the provider used to submit the transaction is not exactly the same. When submitting the transaction using UCMP we have to perform requests to Infura service provider while when submitting directly to the blockchain we used MetaMask as a service provider. Although MetaMask at its core also uses Infura as its own service provider, we can not assume that both have the same level of efficiency when submitting transactions. For these two tests we also did not take into account the time required to build and sign the transactions as they were variables common to both tests.

Custodial and Non-Custodial users submission times

Given the fact that we already had the baseline average value to submit a transaction as well as the overhead caused by UCMP when simply submitting a transaction we decided to test and compare these values with the overhead caused by UCMP when requesting data from our database, assembling, signing and subtitling a transaction to the blockchain. In other words, we would be comparing the latency of a transaction that is submited by a non-custodial user with a transaction that is submited by a custodial user.

On this test, contrarily to the previous one, we also measured the time it takes for assembling and signing a transaction. The assembling process includes a Read operation to the blockchain, in order to retrieve the appropriate nonce, and performing a hash on the relevant parameters that will be sent to the blockchain. Our goal is to determined as precisely as possible the time it takes for both types of clients to submit a transaction.

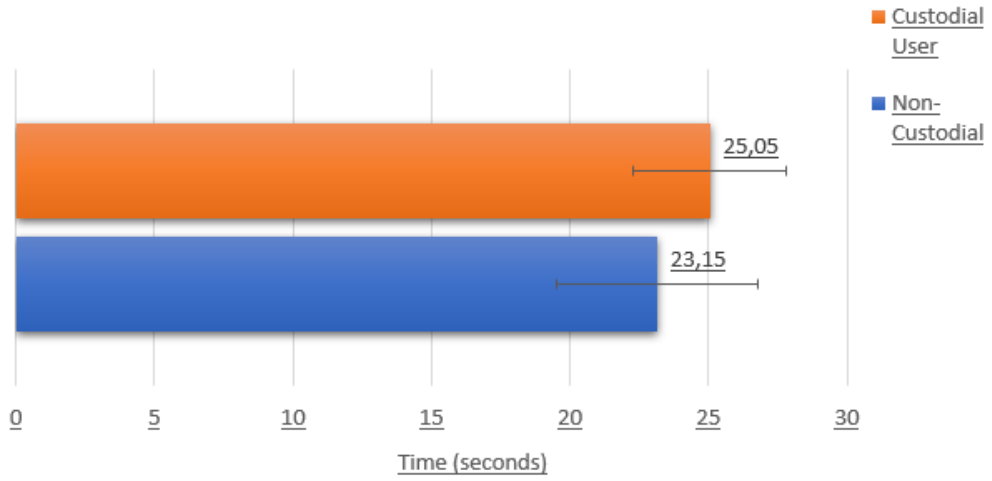

Figure 4.4: Graph representing the obtained values between a submission made by a custodial and by a non-custodial users

Figure 4.4 represents the obtained results when both user types submit transactions representing an accept action. In other words, we measure the time it took for each user type to accept a sharing request. For this test we also had to perform slight adjustments to UCMP smart contracts. On the developed smart contracts, it is only possible to accept a request with a *pending* status, therefore after every test round we had to return the request status to *pending*. Usually, this would not be possible since the *pending* status is only given as the default value when creating a new request but, in order to avoid the necessity of creating a new request on every test round, we decided to temporally remove this restriction and create a function that converts a request status from *accept* to *pending*. This function was automatically called at the end of every test round and had no influence on the obtained results.

As we can observe in Figure 4.4, the custodial users submission takes a small additional time when compared to a non-custodial user. This value is within expectation since a custodial user submission has to perform extra actions when compared to the almost direct submission process of a non-custodial user. We can further observe that the average custodial user submission takes 1.90 seconds longer than a non-custodial. As mentioned on the previous test, the main reasons behind this delay are the network connection and the extra computational time performed by

the cloud function. In addition to these reasons, we also have to take into consideration the time it takes UCMP cloud function to access our database and retrieve the client's private key. Although this access time should be very small when compared to the network delay and the computational time a cloud function takes to execute, it will still impact the total submission time and should be taken into account.

4.2.2 Transaction Cost

After getting the results for the latency tests of UCMP operations we decided to measure the transaction cost of the different actions that data holders and data owners could perform with UCMP. With these test we wish to be able to provide UCMP's clients with an estimate cost for each transaction they perform.

A transaction's cost can be influenced by two main factors: (1) the actions performed within the smart contract or smart contract efficiency and (2) the gas fee paid by the sender. There can be two smart contracts that perform the same action but with different implementations where one of these has double the gas consumption of the other. Reading and writing from storage has a gas cost associated with it, having needless read and write operations within a smart contract's code can lead to an unnecessary gas consumption.

In addition to those two main factors, the information sent within a transaction also influences the transaction cost. Lets take a request submission action as an example. In this transaction that is sent by a data holder when he wishes to ask a data owner for a consent request, he has to send to the blockchain several fields. Recall Section 3.2.2, where the data holder has to specify the name of the data consumer as well as which attributes they wish to access and the purpose of use.

Although the size of a string influences the gas consumption, from these three fields, the one that impacts the gas consumption the most is the attributes field. The size variation of the data consumer's name and the purpose of use only slightly influences the gas consumption given the fact that both fields only have a single string sent to the smart contract. Therefore, having a data consumer named "My Data Consumer" and another named "My Other Data Consumer" would have minimum influence on the transaction cost difference between two equal transactions when also taking into consideration the attributes field. On the other have, if we have two similar transactions with the only difference being on the number of attributes that are being requested, we can observe a steady increase on the transaction cost. During this test, we assume that each attribute has, on average, six characters, therefore the difference between a five attribute transaction and a ten attribute transaction is thirty characters.

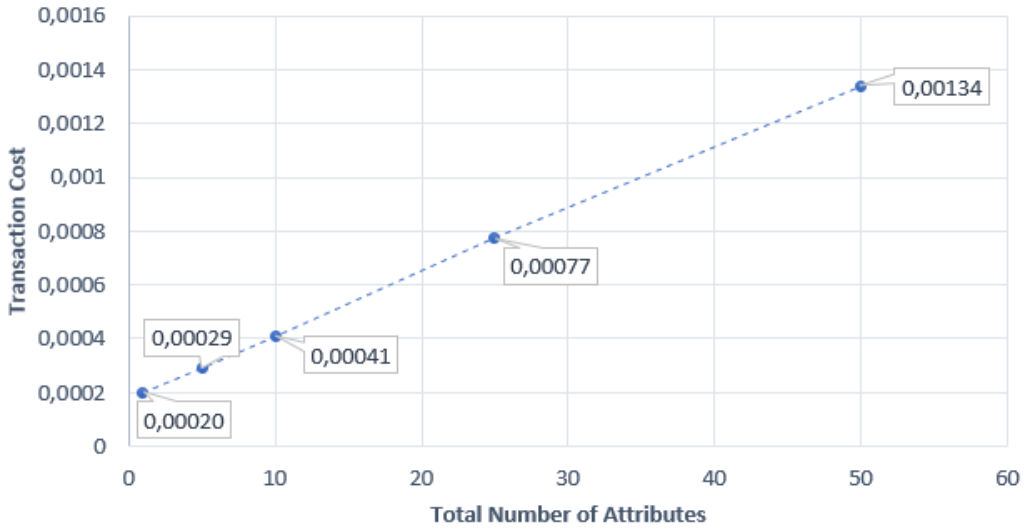

Figure 4.5: Graph representing the obtained values for the relation between transaction cost and number of attributes submited

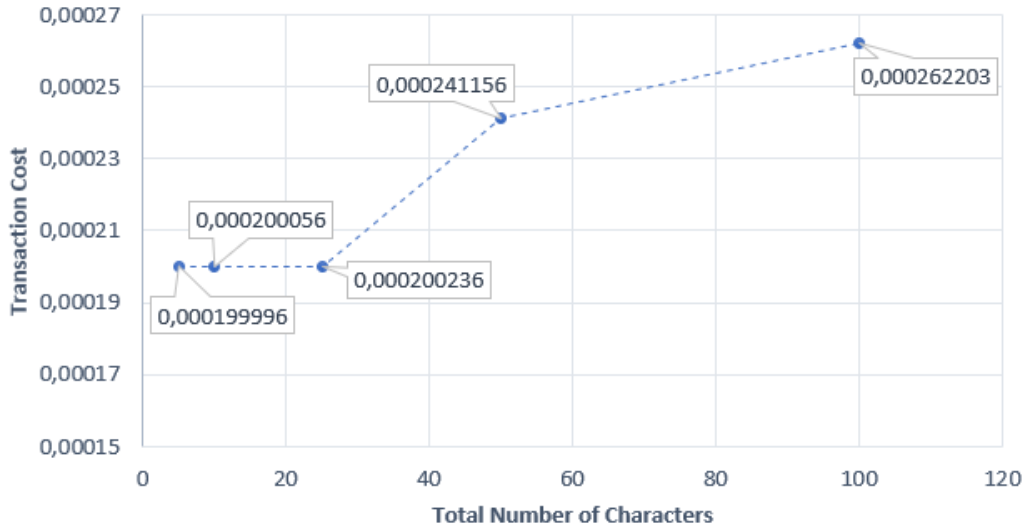

Figure 4.6: Graph representing the obtained values for the relation between transaction cost and number of characters used on a string

Observing Figure 4.5 and Figure 4.6 show us two comparisons. The former between the cost of a transaction with the number of attributes submited and the latter between the transaction cost and the number of characters of a single string. For both test we have a transaction that is sent by data holders when they wish to request sharing consent to their clients. This transaction has the following structure and parameters:

- **Data Consumer** - "CUF"
- **Data Holder** - "Lusiadas"
- **Attributes** - "["Name", "Age", "Blood Type"]"

- **Purpose of Use** - “Data needed to perform an exam”,
- **Additional Comments** - “” (empty string)

A practical example of a transaction like this would be the following:

```
await SmartConsentManagement.methods
      .requestPermission(
        "CUF",
        "Lusiadas",
        ["Name", "Age", "Blood Type"],
        "Data needed to perform exam",
        ""
      )
      .send({
        gas: 2000000,
        from: web3.givenProvider.selectedAddress,
      })
```

This transaction was used as a baseline for our test regarding the transaction cost. As mentioned before we decided to perform two variations of this transaction in order to determine the influence of adding or removing data from a transaction.

For test 4.5, we increase the number of fields that were sent over on the transaction, while maintaining the remaining fields as they were. On the other hand, for test 4.6 we decided to increase the length of an attribute’s string instead of altering the number of attributes.

As expected, by observing graph 4.5, increasing the number of attributes that are sent in a transaction, has a major influence on the transaction cost. On the other hand, we expected graph 4.6 to have the transaction cost remain almost the same. By observing the image, we can see that there is an disproportional increase when a string with fifty characters is sent as well as a string with one hundred characters. The reason behind this sudden increases is related with Solidity and the Ethereum Virtual Machine architecture. Recall that EVM works with a word size of 256 bits (or 32 bytes), and has memory addresses of the same size. One byte is capable of storing one character meaning that it can store strings as long as 31 bytes on a single memory address (the extra byte is used to store metadata and therefore can not be used to store any string character). Storing any string within this size corresponds to one single write operation

on the EVM storage. When a string longer than 31 characters is submited, additional memory addresses have to be used in order to store the remaining characters.

This is the case for the strings submited during our tests. By sending in the transaction a string with fifty characters, EVM has to use two memory addresses in order to store that string. This corresponds to two write operations instead of one, which dramatically increases the transaction cost (when compared to a single write operation). Similarly when submitting a transaction with a string size of one hundred characters, three write operations have to be performed in order to store this string which further increases the transaction cost.

Applying the same line of thought to the results obtained on the first graph 4.5, submitting multiple attributes within a single transaction corresponds to multiple write operations on the blockchain. For instance, by sending five attributes in a transaction the EVM has to perform five unique write operations, for ten attributes, ten write operations are needed and so on. With both test results we can clearly see that adding more attributes to a transaction has a much greater impact than adding several characters to a string, for example, adding a description on the *additional comments* field.

4.2.3 Gas price

The final test we performed was related with the gas price and the latency reduce that was caused by this increase. For this test we decided to send the same transaction with different gas prices. Recall that miners choose which transaction to add to the block their are mining based on the gas fee that the sender is willing to pay for that same transaction. By increasing the gas price, the sender also increases the chance that his transaction will be picked by a miner. In other words, the higher the gas price, the higher the transaction fee a sender has to pay is and the higher the chance that his transactions is chosen by the miners.

Gas prices may also vary according to the current network traffic, for instance, using 10 GWEI, as a gas price, at a time where the network is crowded may make the sender's transaction take longer to execute than paying 6 GWEI when the network is with low traffic. For our experiment we decided to use three different values for the gas price (10, 20 and 30 GWEI).

By observing Figure 4.7 we can see that increasing the gas price did not influenced the average time to submit a transaction. The expected results would be to see a time reduction when the gas price increases. One possible explanation for the obtained results could be related to the network. If there was low network traffic during the tests it would mean that the gas price that was paid was irrelevant to the transaction speed. In other words, if we assume that each block can contain a maximum of twenty transactions and there is only enough network

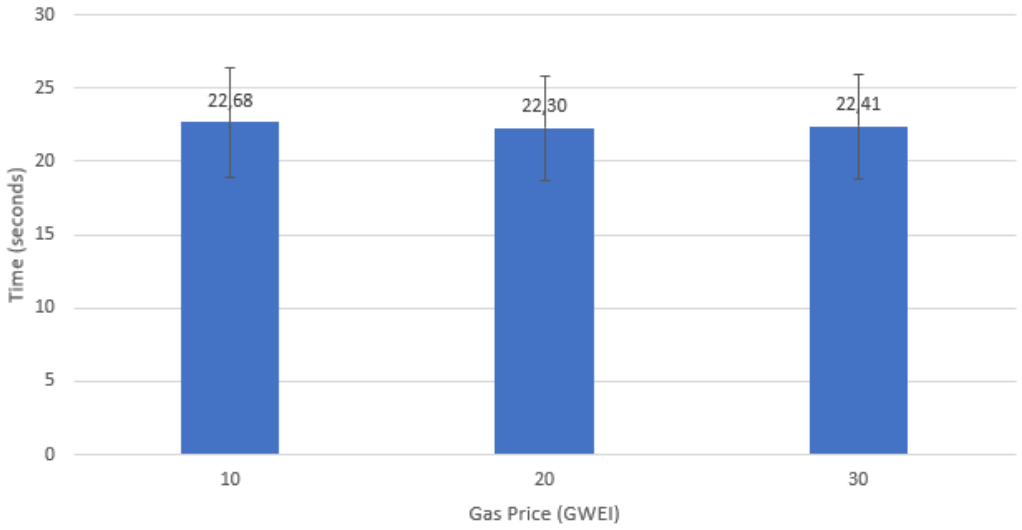

Figure 4.7: Graph representing the obtained values for the relation between gas price and transaction speed

traffic to submit an average of fifteen transactions, having a low gas price will yield the same results as having a high gas price.

4.3 Discussion

In this section we will briefly discuss the obtained results and give some insights on the reasons why the transactions used during our testing phase were chosen.

Recall that UCMP operations perform several transactions to the blockchain:

- **(1) Request Sharing Consent** - transaction sent by data holders when they wish to ask one of their clients for a sharing consent request.
- **(2) Deploy the client's smart contract** - transaction sent by data owners when they first use UCMP in order to deploy their personal smart contract on the blockchain.
- **(3) Accept Request** - transaction sent by data owners when they wish to give their sharing consent to a given request.
- **(4) Refuse Request** - transaction sent be data owners when they wish to refuse a sharing request from a given data holder
- **(5) Revoke Request** - transaction sent by data owners when they wish to revoke their sharing consent regarding a specific request

During our first test, reported on Section 4.2.1, we decided to use the transaction (1) due to the fact that no adaptations had to be made on the developed smart contracts in order to

perform the tests. Note that, since we are testing the network latency, any transaction could have been used as all of them would have yield similar results. From this first test, we retrieved a baseline result that could be used as "an ideal value to achieve" when measuring latencies using UCMP as a submission medium. Here, we obtained an average time value for submitting transactions to both Ropsten and Rinkeby test networks. We were able to attain an average submission time of 43,77 seconds for Ropsten test network and an average time of 22,94 seconds for Rinkeby test network. As already explained on Section 4.2.1, these values are expected to be higher than the average time to mine a new block, recall that the average time to mine a new block is around 30 seconds for Ropsten and 15 seconds for Rinkeby. With the theoretical results, for submitting a transaction, being around 45 seconds for Ropsten and 22,50 seconds for Rinkeby we can affirm that the obtained practical results, although having slightly lower time values for the Ropsten test network, are within expectations.

One possible explanation for the measured values, regarding Ropsten test network, being slightly lower than the theoretical expected results, could be related with the network traffic. Remember that, on Section 2.1 when we first introduced miners, we mentioned that they had to validate transactions before appending to the block they would mine, as well as, in case of a proof of work consensus algorithm network, solve a cryptographic puzzle that requires computational power to be invested in. Miners perform these two actions simultaneously by adjusting the amount of computational power they apply to solving each task. Assuming that there was a low network traffic during our first test, it would mean that there were fewer transactions to be validated. With this being said, miners could simply invest almost one hundred percent of their computational power on trying to solve the cryptographic puzzle, reducing the time it would take to mine a new block by a few seconds.

On our second test, Section 4.2.1, we decided to use the already obtained baseline results for the latency when submitting direct transactions to the network and compare it with the latency when submitting transactions using UCMP. In order to measure these value we used the (3) transaction, given the fact that only data owners may have the need to use UCMP as the submission medium. Note that, although transactions (4) and (5) are also executed by data owners, these perform very similar actions that would not impact the overall latency of a request. On this test, we decided to use the Rinkeby test network, due to the fact that this test network presented better results than Ropsten test network during our first test (the results obtained were closer to the theoretical values).

When using UCMP to submit transactions, we obtained an average time value of 24,24 seconds. By comparing this value with the results of our first test we can see that by using

UCMP a client can be expected to wait, on average, an extra 1,27 seconds. This additional time corresponds to only about 5% of the total time it takes to submit a transaction using Rinkeby test network. Considering that, this 1,27 extra seconds, takes into consideration the time it takes to perform the additional request to UCMP and the time it takes to call and execute the cloud function used to submit the transaction, we believe the overhead generated by UCMP to have little impact on the performed action. Assuming that this overhead absolute time (1,27 seconds) would be similar when using Ropsten testnet, it would mean an increase of only approximately 2,9% on the total submission time. The longer the average time to submit a transaction the lower this percent value will become.

Our third test was related to transaction (3), we wanted to compare the latency of our clients requests when submitting a transaction to the blockchain using our system. For this test we also accounted for the time it takes the client's browser and UCMP to assemble and sign a transaction. Our goal was to simulate as closely as possible, a client's routine when using UCMP to manage their sharing consents. As mentioned on Section 4.2.1 we had to perform slight modifications on the developed smart contracts in order to facilitate this test. The performed modifications had no impact on the results obtained and were made simply to speed up the testing process. Our results showed us that custodial users take a bit longer to submit their requests wen compared to non-custodial users. The difference between this two request was 1,9 seconds. Although both requests perform the same actions we have to take into consideration that during a custodial user's request submission UCMP has to access the database as well as call and execute a cloud function. Most of this overhead is generated by the cloud function execution due to the fact that when a function call is made by our system, Firebase has to launch the cloud function. Even though this Firebase approach prioritizes saving resources it brings a small delay, in the order of milliseconds, to the system normal execution.

After testing the latency of our solution, we decided to test the transaction cost of two variations of transaction (1). Transactions (2), (3), (4), and (5) had no user inputs, therefore their cost is almost constant when compared to transaction (1). The cost for this transactions was measured using a gas price of 1 GWEI. By using this gas price we can easily predict the total cost of a transaction to any other gas value, we simply have to multiply the shown results by the desired gas price.

Transaction (2) cost is around 0,0040 ETH which represents the cost needed to deploy a client's smart contract on the blockchain. Since transactions (3), (4), and (5) only perform a single write operation on the blockchain, their costs are around 0,00020 ETH per transaction. On the other hand, transaction (1) is assembled based on UCMP's data holders' inputs, therefore its

cost may vary according to the inputs that are given by the data holder. As shown on our results in Section 4.2.2, increasing the number of attributes highly influences the transaction cost. This is due to that fact that these attributes are store inside a mapping on the smart contract. For this reason, the EVM has to perform a write operation in order to store a single attribute. By having multiple attributes sent in a single transaction means a direct increase on the transaction cost. Since each attribute refers to a certain category of personal information, very rarely will a data holder need to perform a transaction that contains more than ten. Nonetheless, in our tests we decided to go as far as performing a transaction that requests access to fifty unique attributes. By doing so we could get a better grasp of the transaction's cost should we wish to expand the attributes list.

On our last test, we decided to measure the influence that the transaction gas price had on the overall transaction speed. We used three different gas values with equal intervals between them. With this approach we ought to find the most efficient gas/transaction time value possible. By observing the results we can determine that the gas price did not have any impact on the transaction speed. Although this does not meet the expected results, we can easily identify the main cause for this discrepancy on the final results - the network traffic. Due to the lack of network traffic, there was never a competition between senders in order to get their transactions chosen by the miners.

With these test results, UCMP meets our expectation of producing a low overhead regarding transaction submission as well as having an appropriate cost per transaction. Data holders who actually need access to certain personal information from their patients can easily ask for a sharing consent while data holders who may try to abuse our system by requesting additional attributes than the ones needed will have an extra value to pay in term of ETH.

4.4 Summary

In this section we started by presenting the methodology used to evaluate UCMP performance. We then proceeded to the testing phase where we submited our system to several performance tests. These tests aimed at finding and evaluating UCMP impact regarding the requests latency for both user types. We started by establishing some baseline results that were then used to measure UCMP performance.

Further ahead we discussed the transaction cost for two different variables - the length of a parameter versus the amount of parameters used in a single transaction where we concluded that the latter has the most influence on the final cost of a transaction.

Afterwards, we presented the influence that the gas price has over a transaction submission

speed. Finally, we end this section with a brief discussion regarding the obtained results and the insights that they give regarding UCMP performance.

Chapter 5

Conclusions

This document explores the core concepts of blockchain, namely, as a replicated append only data structure that stores an ordered list of transactions. Furthermore, we also briefly explain how to take advantage of this technology in order to develop DApps. A characteristic of DApps is that they benefit from a strong tamper proof resilience offered by the blockchains. A clear distinction was made between permissionless and permissioned blockchains.

Along the document and particularly in Section 2, an overview is presented about the several blockchain based systems related with our proposed solution. Those systems were briefly analysed based on their functionality, provided service and security capabilities.

Our main goal for this thesis, as previously mentioned, was to provide a consent management platform capable of enabling patients to have a better control over their personal medical data while providing data holders a means to stay compliant with GDPR. On Chapter 3 we proposed a consent management platform that allows patients to manage their sharing consents and control the accesses made to their medical records that are stored on different medical institutions.

We also explain and detail the core components as well as the architecture of the system. When describing the architecture of our system we also presented the data structures that were used as well as the backend logic responsible for managing both our clients and their consents.

Our work on design and implementation of a blockchain platform for consent management, made UCMP capable of satisfying the needs of patients by providing a decentralized platform that stores their sharing consents without the risk that central storage offers (the data stored could be easily altered without a patient knowledge) and by delivering a solution that data holders can use in order to collect their patients consents while staying compliant with the EU's law.

Lastly, on Chapter 4, we presented the practical results for the developed prototype. Users can expect UCMP to add a small overhead when submitting transactions to the blockchain. We

also demonstrated the actual cost of each transaction, allowing users to choose which price they are willing to pay (in Ether) for each transaction. As mentioned before, the transaction fee can vary based on the gas price each client decides to use.

5.1 Future Work

Even though this solution incorporates blockchain technology and the healthcare business area, similar solutions which aim at providing consent management for personal information that is being stored by third parties, could be developed by adapting the methodology of this platform to the desired business area. To adapt and enhance this work, there are some tasks that could be done as future work. As an example of it, we propose the following points:

- Expand UCMP in order to be used in additional areas outside Healthcare. In order to achieve this, the *attribute* field would have to be modified. This field allows the developer to adapt the platform to his needs.
- Incorporate both submission methods in a single one. It is possible to convert both the custodial and non-custodial user's submission methods in a single one, which would relief non-custodial users of the burden of having to pay or transactions. They would simply have to sign their own transactions and UCMP would be in charge of submitting those transactions (paying the transaction fee).
- Create a trade broker that would allow similar platforms to share or trade existing consents. One additional use for this trade broker would be to allow users to sell their personal data directly to the data consumers, creating a marketplace around personal information.

Bibliography

[ABB+18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018.

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, pages 164–186. Springer, 2017.

[And17] Ana Raquel Rocha Andrade. Adoption of electronic health records in the portuguese healthcare system in the presence of privacy concerns. Master's thesis, Universidade Católica Portuguesa, 2017.

[B+14] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3:37, 2014.

[BCGH16] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: an introduction. *R3 CEV, August*, 1:15, 2016.

[Bec18] Roman Beck. Beyond bitcoin: The rise of blockchain world. *Computer*, 51(2):54–58, 2018.

[BP17] Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *International conference on financial cryptography and data security*, pages 494–509. Springer, 2017.

[Cac16] Christian Cachin. Architecture of the hyperledger blockchain fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers*, volume 310, page 4, 2016.

[CDE+16] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On

scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 106–125. Springer, 2016.

[Cor19] Miguel Correia. From byzantine consensus to blockchain consensus. In *Essentials of Blockchain Technology*, chapter 3. CRC Press, 2019.

[DAK+16] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*, pages 79–94. Springer, 2016.

[Dan17] Chris Dannen. *Introducing Ethereum and Solidity*. Springer, 2017.

[DLZ+18] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Ji Wang. Untangling blockchain: A data processing view of blockchain systems. *IEEE Transactions on Knowledge and Data Engineering*, 30(7):1366–1385, 2018.

[DRFM18] João Pedro Dias, Luís Reis, Hugo Sereno Ferreira, and Ângelo Martins. Blockchain for access control in e-health scenarios. *arXiv preprint arXiv:1805.12267*, 2018.

[Eur16] European Parliament and European Council. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation), April 2016.

[Fin18] Michèle Finck. Blockchains and data protection in the european union. *Eur. Data Prot. L. Rev.*, 4:17, 2018.

[GHM+17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[GZL+17] Philippe Genestier, Sajida Zouarhi, Pascal Limeux, David Excoffier, Alain Prola, Stephane Sandon, and Jean-Marc Temerson. Blockchain for consent management in the ehealth environment: A nugget for privacy and security challenges. *Journal of the International Society for Telemedicine and eHealth*, 5:GKR–e24, 2017.

[HBHW16] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. *Tech. rep. 2016–1.10. Zerocoin Electric Coin Company, Tech. Rep.*, 2016.

[Her19] Maurice Herlihy. Blockchains from a distributed computing perspective. *Commun. ACM*, 62(2):78–85, 2019.

[KMS+16] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 839–858. IEEE, 2016.

[KN12] Sunny King and Scott Nadal. PPCoin: Peer-to-peer crypto-currency with proof-of-stake. self-published paper, 2012.

[LCO+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269. ACM, 2016.

[LCT] Tom Lyons, Ludovic Courcelas, and Ken Timsit. Blockchain and the GDPR.

[Mic17] Microsoft Corporation. The Coco framework - technical overview, 2017.

[MPA+18] David R Matos, Miguel L Pardal, Pedro Adao, António Rito Silva, and Miguel Correia. Securing electronic health records in the cloud. In *Proceedings of the 1st Workshop on Privacy by Design in Distributed Systems*, page 1. ACM, 2018.

[N+08] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference USENIX ATC*, pages 305–319, 2014.

[Pec17] Morgen E Peck. Blockchains: How they work and why they'll change the world. *IEEE Spectrum*, 54(10):26–35, 2017.

[RBBM19] Michel Rauchs, Apolline Blandin, Keith Bear, and Stephen B McKeon. 2nd global enterprise blockchain benchmarking study. 2019.

[RDD+18] Konstantinos Rantos, George Drosatos, Konstantinos Demertzis, Christos Ilioudis, and Alexandros Papanikolaou. Blockchain-based consents management for personal data processing in the iot ecosystem. In *ICETE (2)*, pages 738–743, 2018.

[SC17] Carlos Serrão and Elsa Cardoso. Handling confidentiality and privacy on cloud-based health information systems. *Journal of Information Privacy and Security*, 13(2):51–68, 2017.

[SHW19] Roberto Saltini and David Hyland-Wood. IBFT 2.0: A safe and live variation of the IBFT blockchain consensus protocol for eventually synchronous networks. *arXiv preprint arXiv:1909.10194*, 2019.

[Sza97] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[Und16] Sarah Underwood. Blockchain beyond Bitcoin. *Communications of the ACM*, 59(11):15–17, 2016.

[XPZ+16] Xiwei Xu, Cesare Pautasso, Liming Zhu, Vincent Gramoli, Alexander Ponomarev, An Binh Tran, and Shiping Chen. The blockchain as a software connector. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 182–191. IEEE, 2016.

[YGWO16] Danny Yang, Jack Gavigan, and Zooko Wilcox-O'Hearn. Survey of confidentiality and privacy preserving technologies for blockchains. *R3, Zcash Company, Res. Rep*, 2016.

[YWJ+16] Xiao Yue, Huiju Wang, Dawei Jin, Mingqiang Li, and Wei Jiang. Healthcare data gateways: found healthcare intelligence on blockchain with novel privacy risk control. *Journal of Medical Systems*, 40(10):218, 2016.

[ZNP15] Guy Zyskind, Oz Nathan, and Alex Pentland. Decentralizing privacy: Using blockchain to protect personal data. In *2015 IEEE Security and Privacy Workshops*, pages 180–184. IEEE, 2015.