MTChain: Identity and Confidentiality for Blockchain based Systems

Jorge Guilherme Vieira Alves

Thesis to obtain the Master of Science Degree in Engineering Systems and Computer Engineering

Supervisor(s): Professor Miguel Nuno Dias Alves Pupo Correia
Eng. Diogo Tomás Ferreira

Examination Committee
Chairperson: Prof. João António Madeiras Pereira
Supervisor: Professor Miguel Nuno Dias Alves Pupo Correia
Member of the Committee: Prof. Carlos Serrão

November 2018
Acknowledgments

I would like to thank first my supervisors, professor Miguel Correia, Diogo Ferreira and other Multicert staff who supported, guided and kept the environment light, although it being a big project. Orientation was key to this work and the people who stood by my side 8h/day were crucial.

I would like to thank my family who, throughout the years have been supporting me and encouraging me to never stop and reach the end. For all the patience, support and dedication I thank you all.

I would like to thank my dearest friend Sofia Passadouro, who saw this project from the beginning and the way it consumed my time. For all the patience and the caring, I will be forever thankful.

I would like to thank all my close friends who, in moments of stress and pressure always stood by my side, giving me great moments that made me a happier worker.

I would like to acknowledge Multicert. Thank you for the opportunity that was to work with a real team and the overall environment that I was inserted to. Thank you to everyone in the company for the good and the hard times.

Finally, I would like to thank my university and its services, for giving me a home, good means to learn and for being so open and organized about the thesis context.


Resumo

Nos dias de hoje, a tecnologia da blockchain torna-se cada vez mais popular e adaptável a problemas que antes requeriam uma entidade central. Esta tecnologia, implementada pela primeira vez para suportar a famosa criptomoeda Bitcoin, tem vindo a evoluir para diversos tipos com maior consideração pela necessidade dos utilizadores por privacidade e identidade.

No trabalho exposto neste documento é descrito o processo de desenvolvimento de um sistema que possui como base uma rede gestora de blockchain, que irá interagir com a plataforma desenvolvida - MTChain. A gestão de identidade dos utilizadores do sistema será tratada por esta plataforma - sempre com consideração pela privacidade e confidencialidade da informação que é transmitida entre cliente, plataforma e rede de blockchain.

De modo a demonstrar o funcionamento do sistema completo, foi desenvolvida uma aplicação do sistema, Notário, descentralizando um serviço dispendioso e muito requisitado. Esta aplicação funciona num contexto web, tirando proveito do facto da sua base ser blockchain.

O desenvolvimento e pesquisa de todo o trabalho exposto neste documento foi feito sob a alçada da empresa portuguesa Multicert, onde esta sugeriu o tema, aplicando assim um tema bastante discutido atualmente no seu contexto empresarial.

O propósito, por parte da Multicert, para implementação e exploração deste trabalho foi a aplicação do seu negócio principal numa rede de blockchain. Assim, desenvolvendo uma aplicação com o uso destes mecanismos, a empresa poderia aplicar os recursos para reconhecimento de identidade, nomeadamente a certificação de indivíduos e identidades.

Palavras-chave: Hyperledger, Multicert, Blockchain, Notário
Abstract

Blockchains are becoming more popular and diverse, adapting to problems that used to require central entities. This technology, first implemented as the base for the famous cryptocurrency Bitcoin, has been evolving in the past years, taking into account aspects such as privacy, execution context and identity.

This work describes the development of a system that holds as its base a blockchain network, which interacts with the developed platform - *MTChain*. User identity management is done by this platform, which also provides the necessary means for privacy and confidentiality of the information that is transmitted through the client, the platform and the blockchain.

To present the work with more detail, an application serving as prototype was developed, Notary, decentralizing a service once assured by a third party entity. This application leverages the use of a blockchain on its base, establishing a trust level with the customers by the security insurances taken from it.

The development and research of the work in this document was done under an internship context in the portuguese certification company Multicert, suggesting this document matter, applying to its context a technology that is nowadays widely discussed.

For Multicert, the overall goal of this project’s implementation and investigation was to apply its business to it. The development of an application using blockchains would provide to the company an insight to what were the opportunities in terms of certification, where identity management for individuals and entities could be provided by the company.

**Keywords:** Hyperledger, Multicert, Blockchain, Notary
# Contents

Acknowledgments .......................................................... iii  
Resumo ................................................................. v  
Abstract ................................................................. vii  
List of Tables ............................................................... xi  
List of Figures ............................................................... xiii  
Nomenclature ............................................................... 1  
Glossary ................................................................. 1

1 Introduction ................................................................. 1
  1.1 Motivation ............................................................... 2  
  1.2 Topic Overview ........................................................... 2  
  1.3 Objectives ............................................................... 3  
  1.4 Thesis Outline ........................................................... 3

2 Background ................................................................. 5
  2.1 Blockchain and Smart Contracts ........................................ 5  
    2.1.1 Permissionless Blockchains .......................................... 6  
    2.1.2 Permissioned Blockchains ........................................... 9  
  2.2 Identity Management ................................................... 12  
    2.2.1 Shibboleth System ................................................... 12  
    2.2.2 OpenID Authentication ................................................ 13  
    2.2.3 PKI-based Authentication ............................................. 15  
  2.3 Confidentiality ......................................................... 20  
    2.3.1 Enigma ............................................................... 21  
    2.3.2 Hawk ................................................................. 22  
  2.4 Summary ................................................................. 23

3 The Infrastructure and the Application .................................... 25
  3.1 The MTChain Infrastructure ............................................ 26  
  3.2 The components ......................................................... 28  
    3.2.1 Hyperledger Fabric .................................................. 28  
    3.2.2 MTChain ............................................................. 30
List of Tables

4.1 Table of installing chaincode times .............................................. 52
4.2 Table of upgrading chaincode times .............................................. 52
4.3 Time measures of page loading and operations execution ............... 53
4.4 Number of requests per minute .................................................... 54
4.5 Media of time in throughput testing ............................................. 54
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Blockchain logic</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Hyperledger Fabric Protocol</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>OpenID’s Layered Architecture</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Blockstack Layered Architecture</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>MTChain Architecture Diagram</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>MTChain Layered View</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>Notary: The architecture of the prototype</td>
<td>36</td>
</tr>
<tr>
<td>3.4</td>
<td>Notary: Document signature process flow chart</td>
<td>37</td>
</tr>
<tr>
<td>3.5</td>
<td>Notary: Document after its signing process is finished</td>
<td>38</td>
</tr>
<tr>
<td>3.6</td>
<td>Start new signature Menu, for document description and upload</td>
<td>38</td>
</tr>
<tr>
<td>3.7</td>
<td>Menu &quot;Documents To Sign&quot; that contains a list of documents for the user to sign</td>
<td>40</td>
</tr>
<tr>
<td>3.8</td>
<td>Menu “History”, containing document that the user has or will be interating with</td>
<td>40</td>
</tr>
<tr>
<td>3.9</td>
<td>Flow chart of a new document initialization</td>
<td>42</td>
</tr>
<tr>
<td>3.10</td>
<td>Flow chart of Mark’s document signature</td>
<td>43</td>
</tr>
<tr>
<td>3.11</td>
<td>Flow chart of Elton’s document signature</td>
<td>45</td>
</tr>
<tr>
<td>3.12</td>
<td>Flow chart of the finalizing operation on the document</td>
<td>46</td>
</tr>
<tr>
<td>3.13</td>
<td>The transactions sent to the blockchain after the signature process is finished</td>
<td>47</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Blockchains date from 2008, when an anonymous group known as Satoshi Nakamoto developed a blockchain to be the backbone of the Bitcoin [1] cryptocurrency. Its implementation was done as a part of the Bitcoin infrastructure, consisting of blocks of data connected through hashes with security as the main goal. Coming to 2009, blockchain became open source and its potential was recognized by many, who believed that this technology had the power to revolutionize the industry and commerce fields, due to being immutable and transparent, while being secure, fast and trustworthy [2][3].

Early developments on the blockchain allowed for technological advances. An example of this is the Ethereum's blockchain [4] which explored the functions smart contracts could have: an automated, autonomous way of making transactions safely [5].

With the evolution of blockchains and the popularity of Bitcoin raising, some 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 restricted and controlled, called permissioned. In these platforms, unlike the previous, permissionless, not everyone is allowed to join the network and the working nodes are trusted in advance [6]. They usually are used in a closed environment such as company systems, and they allow for monitoring and control [7][8][9]. To promote honest work among the network, public blockchains implemented a consensus mechanism called proof of work. The first to calculate a valid value for the proof of work is rewarded, for it represents proof of the legitimacy on the work of the node. The mechanism also works as a counter measure against malicious activity. On private blockchains, the nodes lack the need of compensation. These systems rely on the consensus protocols for correctness verification, where the main concern are Byzantine faults that must be controlled [10][11].

Another interesting topic in blockchains systems is confidentiality and privacy. Public systems preserve the anonymity of their users, but leave available to the public the data that is processed. This happens for verifiability purposes, just like proofs of work on blocks from Bitcoin, that everyone can see and easily check. However, some blockchains are interested in other privacy and confidentiality features. Systems like Namecoin [12] and Blockstack [13] had a different approach on user anonymity, providing names and name resolving protocols (one name belongs to only one user), whereas systems

Although providing great security solutions and safely removing the middleman, blockchains still lack the necessary development in some areas. Some of those areas are explained in this document, being the first identity management.

### 1.1 Motivation

When working with blockchains, one can almost get the feeling that they were created with anonymity as a requirement, due to its public nature. Maybe because they were developed as the backbone to Bitcoin, the technology that involves all the mechanisms seems perfectly adequate to keep identity of the users anonymous. Some approaches try to provide identity to the users, but not without finding major difficulties in storage management or name assignment.

Another problem is the complex nature of the chain itself, which tends to keep all records visible and deny confidentiality to the users, while solving this by not connecting the records to any person/entity in specific, but to keys which, in the case of Bitcoin, are changed after every transaction to prevent tracing. Furthermore, the whole concept of security behind the blockchain sits on keeping a history and on being able to verify the validity of the performed operations, leading to confidentiality issues, for how can the system verify something that it can not read?

Reasoning about these two problems seems to lead to logical conclusions that are interconnected. When having strong identity, there is a need for user privacy, resorting to confidentiality upon the data that it stores. The goal of this work is to create a solution that is blockchain-based, with no third party verifiers, that also incorporates strong identity proofs, confidentiality and privacy on the system where only permissioned users are able to read the data.

### 1.2 Topic Overview

This document describes the development of a platform **MTChain** that will be communicating information records to a blockchain network. Although the stored information is not public (as it is in public systems), it is auditable for integrity and repudiation, while still confidential. The consideration for these topics create a trust factor between the user and the system.

Individual users are managed by a system administrator, providing identity to the participants through the means of PKI, assigning certification to its users. The administrator also has the purpose of adding and removing user from the system, where it sees fit in such actions.

Identity will be provided in terms of a public key infrastructure, for each participating part will have its own certificate that is known widely.

Furthermore, the values (i.e., data) of user actions (such as signing a document) will be stored in the system, but will be going through a set of cryptographic mechanisms for privacy purposes. After, it will
be transmitted to the blockchain where it is kept, but without disclosing the data, where an auditor can later check on the veracity of the data, but not the data itself.

For a better understanding on the system and its mechanisms, every implementation decision is explained in detail further in the document, being exemplified with resort to a prototype application, clarifying certain needs and utilities of the system.

This solution was developed during an internship in the company Multicert that specializes in the issuing of certificates, being the first to be recognized by the Portuguese National Security Cabinet as a certification authority [16][17]. Multicert co-oriented the work in professional environment, where the final goal was to develop a prototype of an application for demonstration purposes of the system. The final work should resolve into an extension to Multicert’s business model, creating means for the development of a new tool to apply their certification means, where the use of the resulting work would be exclusive to individuals or entities that would purchase/possess their certification artifacts.

1.3 Objectives

The work described in this document has several objectives to be taken into consideration:

- To provide a system that is available, reliable and permissioned, using a blockchain infrastructure, allowing only users that are authenticated by the use of certificates.
- The certificates used for authentication and private keys used for data signing should be provided by Multicert’s PKI, which conform to the eIDAS standard.
- The system should provide means of privacy to its users, being able to keep sensitive data undisclosed.
- The system should be manageable through an easy, intuitive interface. This interface should be divided two separated parts: the manager and the client(s).
- To provide a system that allows the implementation of different applications, promoting modularity.
- The final result of this work should be a prototype that demonstrates the functionalities of the system.

1.4 Thesis Outline

The remainder of the document is structured as follows. Section 2 provides an overview of the works related the area, what is the state-of-the-art and the technologies and mechanisms that are used to implement the qualities desired, proving single solutions to the aforementioned problems. Section 3 refers to the implementation of the solution which achieves the goals described previously. In the chapter, the reader will find a close look at the characteristics of the system, both as an aggregate of modules and as individual parts of the system. Finalizing the section, the document takes an approach to the prototype
developed, explaining the logic and mechanisms behind it in detail. Section 4 presents the results of the tests done to the prototype. These demonstrate several approaches for performance testing and a view over security of the system. Finally, Section 5 concludes the document with a summary of the topics and the work, describing the achieved goals and suggesting future work to improve the platform.
Chapter 2

Background

When talking about blockchain, the first thoughts that come to mind are cryptocurrencies, anonymity and safety, for blockchains are the basis to the famous Bitcoin. However, blockchains prove to be an efficient way of removing the middleman from multiples types of agreements, such as money transactions or paper signing. The main goal of this solution is to explore these approaches, keeping in mind the privacy and the security behind them.

This section addresses work related to the development of each one of the above mentioned examples and how they are going to be applied in this solution. In Section 3.1, the theme will be blockchains: a brief explanation of the mechanisms behind it and the presentation of permissioned and permissionless systems. Section 3.2 brings a perspective on existing identity management systems, mechanisms and infrastructures; and Section 3.3 addresses confidentiality and privacy blockchain based systems, and how it could be implemented in the solution.

2.1 Blockchain and Smart Contracts

Blockchains are infinitely growing lists of records that are known for their security properties: integrity, availability and verifiability [18]. Figure 2.1 demonstrates the structure of a ledger that is created with a blockchain logic. All of its components are explained in detail in the following sections.

Figure 2.1: Blockchain logic, where each block connects to the previous through the use of its hash.

The document starts by talking about this technology, going over some of the most popular systems that use this it, explaining how it is used, stating from the operations performed on it and the qualities it brings to systems, being these private or public, directed for cryptocurrencies or other matters.
Along with the blockchain specification, there will be a description of smart contracts, their purpose, evolution over time, and the mechanisms they provide for users.

2.1.1 Permissionless Blockchains

The blockchain comes from the first implementation of the Bitcoin system, where Satoshi Nakamoto [1], an anonymous group or individual, used the technology to support the system and provide the security measures needed to create a reliable infrastructure. This first ledger is public, which means that there are no needed permissions to join or use the system. The same goes with the following topic, Ethereum, that demonstrated great developments in the area of smart contracts. The use of blockchains in these systems was motivated by the decentralization and security features that built within the technology, which allow the use of certain mechanisms to protect user experience: integrity and verifiability to deny attackers leveraging the system, as well as incentives to motivate the correct use; availability to create an infrastructure that is both usable at all times and reliable information wise.

2.1.1.1 Bitcoin

Upon the release of the paper [1], the author explains the mechanisms behind blockchains, and what makes it reliable:

1. **Transactions:** The blockchain is a chain of blocks (as the name states) which contain transactions that are maintained secure by its connection to the next and the previous block. This connection is done in a simple yet efficient way, where each transaction has an hash containing the owner’s public key and the previous transaction, which is signed and verified by the previous owner. This creates a secure chain where, if one were to change a transaction, it would have to change all the transactions that happened before, which is impracticable.

2. **Timestamp Server:** When users interact with a system with such nature, synchronization is required. The lack of mechanisms can lead to a known problem, the double-spend problem, where the same monetary value is spent twice. This might happen because some nodes are not aware of the usage of that coin. To solve this difficulty and to start introducing a node majority agreement, Nakamoto introduced a timestamping server. This takes a block of items, creates a hash of them and publishes it widely. By logic, if there is a hash, then that means that a transaction happened in that moment. The timestamps’ hash also include the previous timestamp, forming a chain.

3. **Proof-of-Work:** The need for a distributed solution for the timestamp server mentioned in the previous topic is obvious. From this, comes the proof-of-work mechanism, which consists in searching for a value such that the block’s SHA-256 hash begins with a desired number of zero bits. This is accomplished by incrementing a nonce in the block until the number of zeros required is achieved in the hash. Once the proof-of-work hash is reached, nodes send it to the network, where verifications of validity are necessary for the addiction of the block to the chain. Once this process is
complete, changing the block itself involves the calculation of the value all over again, an exhaustive computational process. Integrity wise, this leads to unfeasible computations to temper with data: changing information would require a new hash for the block to be accepted, and that would require changes in the previous block that hold that hash. This would go on through the whole chain.

4. **Network**: The network consists of each node collecting new transactions, then working on their Proof-of-work until some node finds it. If all transactions on the block are valid and not spent already, the nodes accept the block and start working on inserting it to the next block on the chain, using the accepted block’s hash.

5. **Disk Space**: In order to save disk space, Bitcoin’s blockchain does not required every node in the network to know the exact transactions in every block. This is possible by using Merkle trees [19] to compute a root hash from the hashes of the transactions in a block and keeping it in the block’s header. By the calculations done in the paper, the predictions are an average of 4.2MB storage required per year for block headers, which represent a light and usable value, even if the headers need to be in the RAM of the computer.

6. **Simplified Payment Verification**: To verify payments on a chain, thanks to the Merkle trees from the previous topic, there is no need to keep more information than the block headers of the longest proof-of-work chain. Using the timestamp of the desired payment block header, he can get the transaction. Although it is not possible to check it, it is possible to see that the network accepted it.

### 2.1.1.2 Ethereum

From Bitcoin, other solutions came. One of the most notable is what could be considered to be the second biggest cryptocurrency flowing in the market, Ethereum. In its whitepaper [4], the authors explain in detail the system and how it diverged from Bitcoin.

The first main topic mentioned is the redefinition of the concept of transactions, which are called "Messages" in Ethereum. The biggest revolution is the fact that, in this system, messages can be created either by an external entity (an user), or by a smart contract.

The idea behind the smart contracts that Ethereum introduced was a concept recreated from the Bitcoin’s protocol, where a less complex version of them is implemented, but with resource to a scripting language, where transactions are owned not only by public keys (users), but also by a script expressed in a simple stack-based programming language, requiring transactions to provide data which would satisfy the script.

These scripts also incorporate other complex functionalities, like "multisig", where, for example, a script requires two signatures out of a given three private keys; or decentralized cross-cryptocurrency exchange. However, the language implemented in Bitcoin has several limitations:

- **Lack of Turing-completeness**: In order to avoid infinite loops during transaction verification, Bitcoin removed the support to them at all, turning the scripts very space inefficient, where code is
repeated over and over again.

- **Value-blindness**: The script language has no way of controlling the amount that can withdrawn which is a fall-back in trust when compared to centralized existing solutions.

- **Lack of State**: The fact that transactions in Bitcoin do not have a state makes the currency a strict object that can either be spent or unspent, leaving behind opportunities for multi-staging with contracts. Losing this comes with the cost of not having features such as decentralized exchange offers or two-stage cryptographic commitment protocols. Besides, it restricts the contracts to simple and hardly reusable. Combining this with value-blindness takes withdrawal limits off the table.

To overcome these weak points, Ethereum provided an elegant framework, compromising to achieve a solution that can build new blockchains efficiently, is able to use scripting with a Turing-complete programming language, and implements a scalable meta-protocol. With this premise in mind, Ethereum developed a more sophisticated version of a smart contract by using what is called contract accounts, which contain the contract's code and it's own data. This contract account, when executing a transaction, runs the contracts code, which can read the code and quantity of ether (Ethereum currency), while it also has the ability to write to its own storage and send transactions to other contract accounts. It can even create new contract accounts, such mechanisms that can lead to powerful tools, as mentioned in Blockchain 2.0 [5].

To provide this technology, Ethereum created a specific low-level, stack-based bytecode language, EVM code, that can be written using Serpent, Ethereum's high level language. EVM code basically consists of a series of bytes (each being an operation code) running in an infinite loop, until an ending/stopping point or error is reached. Then, after ending its code execution, the contract has the ability to return a value, if desired. The execution of the EVM code mode is simple as well. It uses a tuple containing some message information, the code of the contract and the block state, which contains a global state of all accounts, including balances and storage. Each round of the loop takes the pc-th byte of the code.

Approaching the first weak point of the Bitcoin system's smart contracts, and to make them more reliable, Ethereum introduces a concept called GAS, where every message contains GASPRICE and STARTGAS information:

- **STARTGAS**: The limit of how many computational steps of code execution can be spawned by a transaction.

- **GASPRICE**: The fee to pay to the miner per computational step.

Regarding the monetary property of a cryptocurrency, GASPRICE is very relevant. However, for this context, the focused matter will be STARTGAS. This limit was implemented to complete the lack of Turing-completeness weak spot. Although implementing an infinite loop computation style would bring completeness, nothing assures that a malicious contract would not be devastating to the whole network.
Therefore, in order to prevent such event, Ethereum provides a limit to the number of computational steps that can be executed. An attacker would need a continuous supply of gas to run the malicious contract.

Unlike Bitcoin, Ethereum’s framework implements state transition functions. This means that every transaction has a state, which changes according to the success of the transaction, reverting in case of failure and moving to the next state if successful. This covers the lack of state weakness.

The remaining problem, value-blindness, is addressed by granting the code the access to the contract’s information: value, sender and data of the incoming message.

Long-story short, Bitcoin’s implementation gives a general insight on the mechanisms that the system uses to achieve its uses. However, Ethereum’s system turns to a new level, by leveraging smart-contracts and fortifying the concepts behind them. Interesting enough, in the end of the Blockchain 2.0 document, there’s a proposition for an idea of a solution that can leverage the most from the use of both blockchains and smart contracts, but in a permissioned environment, divided into three layers, where the bottom one is the blockchain, the middle one is the smart contract’s implementation and the top layer would be similar to a web front end, and interface for the usage of this system. These systems will be the discussion theme for the next sub section, permissioned blockchains.

### 2.1.2 Permissioned Blockchains

It is possible to separate the technologies that incorporate blockchains into two distinct groups: the permissioned systems, and the permissionless systems. The latest were mentioned in the previous section, where the environment shows a public chain where anyone can enter and see the transactions, while maintaining the anonymity. But permissioned blockchains show a restrict and controlled environment, meeting the needs of corporations for instance, taking part of the burden from the expensive developing of proofs-of-work by setting the core of the network on trusted nodes, replacing the mechanism with a traditional Byzantine consensus protocol, which decides the order by which the blocks are inserted into the chain.

#### 2.1.2.1 Hyperledger Fabric

Hyperledger fabric [20][9][21] is a modular permissioned blockchain platform designed to support pluggable implementations of different components, that uses smart contracts, endorsed peers, ordering services and CA nodes to enable clients to manage transactions. It deploys the code of the smart contracts into the network, where it is executed and validated by the approved peers, which also maintain the blockchain and the state of the database, while respecting the endorsement policies. In the other hand, the byzantine ordering service is responsible for the creation and addition of blocks to the ledger.

Hyperledger Fabric’s transaction protocol, as seen in Figure 2.2, is somewhat simple.
The protocol works by:

1. The client starts by communicating a transaction proposal to the endorsing peers using the hyperledger API. It broadcasts the proposal to every peer that

2. will evaluate and test the transaction against the chaincode that is running. Depending on the result of such testing, the peers return a positive or negative message. If the message is positive, that means that the proposal satisfies the endorsing conditions and is stored in what is called an "envelope" with the signed answers for authenticity and traceability. The same happens when the message returns negative.

3. This envelope is sent to the client, which later sends it to the ordering nodes, the nodes responsible for the synchronization of the system.

4. After receiving the envelopes (one for each endorsing peer), these are ordered, grouped into envelope blocks and broadcast to the endorsing peers that once again

5. verify these envelopes for validity and satisfiability of the endorsing conditions. If the block of envelopes is valid, it will be valid in every working node, where every one will add it to the blockchain.

6. Finally, after this process is completed, the client is notified by the peers by an event that is triggered on the peers upon completion.

The transactions to the endorsing peers are synchronous (step 1), forcing the client to wait for an answer (the envelopes), for logical reasons, since the client must later deliver them to the ordering nodes afterwards. However, this communication (step 4) is asynchronous, allowing the client to carry on from this point.

In the 2018 work [20], it is described some techniques to improve this platform. The goal was to expend less resources when comparing, giving better transaction latency and throughput by improving
on the protocols for ordering messages. This was done by reusing already existing BFT system: BFT-SMaRt [10] and WHEAT [11].

The first comes from the desire of creating a Byzantine Fault-Tolerant State Machine Replication that is both robust and practical for use in real deployments, resulting of a Java-based library that implements the protocol. BFT-SMaRt has the goal of having a highperformance in fault-free executions and correcting in faulty systems, while still bringing advantages such as support to replica sets reconfiguration, and efficient and transparent support to durable services.

The second, WHEAT, implements on top of BFT-SMaRt's library a subset of optimizations for decreasing the latency of strongly-consistent geo-replicated systems. WHEAT is meant to reduce latency by applying several techniques in wide networks, such as using a fixed leader for the consensus protocol, or enlarging the replica pool, but maintaining the quorum size. More practically speaking, WHEAT employs what is called tentative executions, where the clients deliver requests after the WRITE phase, letting the ACCEPT phase being executed asynchronously. Also interesting, this protocol's leader election mechanism relies primarily on the fastest replica present in the system, while preserving safety and liveness properties present in the original protocol, improving latency.

2.1.2.2 Tendermint

Many possible solutions for the consensus problem are provided besides BFT-SMaRt, for example, Tendermint [8][7], a tool for state machine replication that works with tolerance of up to (but not including) 1/3 arbitrary faults.

The protocol uses asynchrony in communication between the consensus nodes, but without discarding synchrony from the execution. Therefore, this protocol is what it is called a weakly synchronized protocol, which is composed by three distinct "components": Proposal, Vote and Lock. The beginning of the protocol happens when a proposer broadcasts a signed ProposalMsg with a composed block, containing transactions recently acquired. This proposer is ordered with others via a simple, deterministic round robin, allowing only one proposer to be valid per round, but known by every validator. The next step in the protocol is voting, which consists of a validator receiving a complete proposal, signing a pre-vote of it and broadcasting it, but broadcasting nil if no correct proposal arrives within ProposalTimeout, ensuring safety. If more than 2/3 pre-votes are received, then the validator is ready to sign a pre-commit vote, stating that it is ready to commit that block to the network. In case that more than 2/3 nil votes are received, then the node can broadcast a pre-vote for nil that will move the protocol to next round. After a validator receives more than 2/3 pre-commits, it commits the block, computes the resulting state and moves on to round 0 of the next block. The innovation on the protocol from Tendermint comes in the last phase of the protocol, where the lock component is added to ensure safety around rounds. Essentially, every validator pre-commit must be justified by a demonstration of more than 2/3 pre-votes on a block, locking after doing a pre-commit. This prevents the insertion of two different blocks in the same place on the chain, while still ensuring liveness and safety through two rules for the component.

Tendermint implements also a property that is useful in this area: Accountability, meaning that, after a violation of safety, the protocol will continue running on a read mode only, and will be able to detect the
malicious nodes contained in the network. However, and logically, this property can only be eventually achieved in the system, or, in other words, after a post-crisis broadcast protocol is detected. Nevertheless, the authors explain that this drawback could be improved by introducing the pre-votes in the pre-commits, which allows the nodes to identify malicious nodes in real time.

Tendermint and BFT-SMaRt represent viable solutions for a consensus problem. When comparing both systems, the deployment of Tendermint is easier to achieve, as well as the performance of this, where BFT-SMaRt combined with WHEAT is a more scalable solution due to the modifications it performs on geographically distributed system.

2.2 Identity Management

As seen in the previous section of the related work, blockchains started out as a support to cryptocurrency systems, and were later used in private environments with resource to mechanisms to prevent malicious activity, while maintaining total functionality and efficiency. However, analyzing the mentioned systems, it is noticeable that either they do not contain any type of user identity specification, or they comply with the anonymity of the user, addressing this problem with only a set of keys that would identify the owners of the exchanging tokens. One of the main goals of this work is to provide reliable and non-repudiable identity to the users and participants in each of the blockchain’s transaction. To do so, several works in the area provide a helpful insight on the possible mechanisms to achieve these conditions.

2.2.1 Shibboleth System

Starting with an approach regarding decentralized providers comes the Shibboleth’s Approach [23] to federated security. Shibboleth is an open-source system that extends webbased applications for secure access to resources, but also enables independent organizations to federate to extend the capabilities of their identity-management services. It also supports multi-organizational federations for scalability purposes, but without leaving aside the privacy of the users.

Shibboleth works by including two major components, the identity provider and the service provide, where the first works along side with the organizations and with the second to protect the private resources. A demonstration of the system through these components would be an example where an user tries to access data that requires identification, communicating to the identity provider the user’s organization, where this would verify the privilege of the user by sending it to the organization’s web site running the identity provider software, redirecting it to the login page that should be accessed with the user’s credentials. After a successful authentication, the browser is sent back to the page containing the resource trying to be accessed with proof that the user signed in. This proof is validated by the service provider that retrieves also additional information on the user from the organization website. The service provider passes these attributes to the resource provider, where it decides upon them if the user has the privilege to access them, displaying either the resource or the negative answer on the user’s web browser. For usability and performance mechanisms, some of these steps can be skipped using
cookies.

This procedure is nowadays a common practice, visible in many attribute/privilege based access to resources, where one must login to an institution or company to obtain access them. Several mechanisms are used to achieve this environment in a secure way.

Shibboleth uses the SAML (Security Assertion Markup Language) standard, the leading standards body for technologies using XML. This standard is based on federated identity. This is a crucial concept, which allows the web applications to offer users cross-domain single sign-on, that lets the user sign in once and gain access to authentication protected resources on other sites [24], coming as a huge interoperability capability between organizations.

The next improvement from Shibboleth would be the use of attributes to provide a more reliable authentication protocol when compared to regular authentication systems. To clarify, what the system does is, after a successful login, the service provider receives the confirmation of such event, goes to the source of the authentication and requests additional information about the user, the attributes, where other systems would only use the user identifier. This is especially useful when dealing with multi-organizational situations, where applications can not access the information about the user, since it would be in a different, restricted domain. However, these attributes provide possible vulnerabilities in the privacy of the users. To prevent this, Shibboleth was designed to provide these attributes with flexibility, extensibility, security and privacy in order to accomplish the demands of the federated scenarios. Another useful addiction is that attributes can themselves become authorizations to resources, like “entitlements”, where one could be “user X is authorized to access resource collection Y from organization Z”, for example.

2.2.2 OpenID Authentication

Focusing on the other existing standards (besides SAML), we move on to the OpenID authentication framework [25]. As seen in figure 2.3, the architecture of the framework can be split into layers. The top layer sits on the URL and XRI layer, that provide the base as the end user’s identifier; those who are below the Yadis layer, which provides a simple service discovery using XRDS document format from OASIS. This is followed by the single-sign-on authentication layer, which is bellow the Data Transport Protocol layer, that provides a level of abstraction for higher level services that depend on trusted data exchange. On top of these layers other identity-based services can exist, depending on the interest of the implementation.

Breaking the first four layers into parts, the bottom most would be the use of identifiers. For this first component, the authors explain how they achieved a user-centric identity management infrastructure by using two different architectural approaches to identifying clients: address-based and card-based identities. The first one uses a unique digital address to identify the user, which is later dereferenced to discover and invoke various associated services. The last uses a digital token that contains or references a collection of attributes or claims that identify the user, while providing the information necessary to accomplish an identity-based transaction. Both of these approaches have their value and can be
used collectively or separately. However, to support service discovery and bi-directional data exchange, OpenID’s platform is address-based, where the address may vary between two forms: an URL or an XRI i-name.

The second layer covers the discovery property of the platform. For this, it is used Yadis discovery protocol for the URLs, and XRI RES protocol for XRI resolution, but, at the time of the writing or the paper, Yadis was being improved to incorporate also XRI resolution for a single discovery protocol. The process is quite simple: retrieval via http or https of an XRDS document describing the services available for a particular URL or XRI. This document can also contain metadata.

The third layer is the motivation and result of this work, the OpenID authentication protocol, where users have the chance to prove that own they own a URL or i-name (XRI). To do so, the protocol involves three parties: The user, the Relying Party (the website which the user is trying to log into), and an Identity Provider, which will provide the service. There are 7 steps to complete before an user is logged in to the website:

1. User enters the URL of the website to initiate the login;
2. The relying party fetches Yadis for a document from the Claimed Identifier;
3. The relying party now creates a shared secret with the Identity Provider, which it finds through discovery;
4. After gaining the Identity Provider URL, the relying party redirects the user to it for login;
5. The User logs in and completes a trust request;
6. The identity provider redirects the user to the resource website with cryptographic proof of the log in and some additional data that the user allowed;
7. Finally, the user is logged into the resource website.

It is also possible to use a private digital address by entering the URL or i-name of their Identity Provider. In this scenario, the Relying Party still uses Yadis discovery to determine the OpenID service endpoint, and uses the same protocol flow, while the Identity Provider recognizes its own digital address.
The last layer, Data transport, provides an abstraction to the third layer, supporting both “pushing” and “pulling” between the Replying party and the Identity provider. This allows OpenID to be an extensible framework capable of supporting several services that require trusted data sharing relationships.

In summary, OpenID’s four layers provide the base for a scalable, decentralized user-centric identity framework. From here comes the flexibility of the platform for the user, where each one can choose its digital address, its identity providers and its service providers.

2.2.3 PKI-based Authentication

User-centric authentication mechanisms are practical and robust, as seen in the previous sections. However, these require extra measures to achieve the conditions necessary for identity management.

Another possible approach to user authentication and identity management are Public Key Infrastructures: light, fast and widely used. These convey authentic and unique user information, traceable to the owner. Besides, PKI related artifacts, such as client certificates, have, alone, the means necessary to authenticate users, discarding the use of passwords.

In the following sub-sections, a deeper notion of public key infrastructures will be discussed, followed by a specification of certificates and the Portuguese company that Multicert that issues them; and finally an implementation of a naming system using PKI on a blockchain.

2.2.3.1 Public Key Infrastructures

Identity management amongst distributed systems was not always as consistent as we see today with certificate based Public Key Infrastructures [26]. One of the main goal of these was to provide strong authentication for non-centralized systems, but also being scalable. These infrastructures have been noticeable for the advances they allowed on security solutions over the internet, resulting in protocols such as SSL/TLS which are nowadays widely used. They provide a solution to confidentiality and integrity for instance, but the main focus on this section will be the authentication and authorization that comes with the use of PKI, where the aim is to assure the identity of the participating parties.

Public key infrastructures (PKI) are combinations of software, asymmetric key cryptography and services for providing security over communications and transactions over public and private networks [26]. The idea behind these is that each party has a pair of keys: a public key, which is public and therefore known by everyone, and a corresponding private key, which should only be known by its owning party. These keys are usually created using public key cryptographic algorithms that are suitable for both digital signing and encryption by using the private and the public key, respectively. For authentication, it is ensured the authenticity of the public key by issuing digital certificates, which have the purpose of binding the public key to a subject in a way that the integrity and validity of it is verifiable. This binding is established by a third party called Certification Authority, such as Multicert [17]. CA’s are trusted parties and their function is to generate, publish, revoke and archive certificates. Certificates, upon creation, are signed with the private key of the owner, and issued along side its root certificate, consisting of a certificate issued by the CA to proof the legitimacy of the creation. A certificate, typically, consists of:
names of the subject and issuer, public key associated, the period before revoking, an identifier for the cryptographic algorithms used by the CA and another for the public key. Whoever wants to use this certificate needs a valid copy of the public key of the CA and must have a copy of the root certificate.

The document [26] presents a model for certificate-based PKI systems, using already existing frameworks for state and event-based systems. This PKI system is encapsulated in a closed environment and does not take into account all the functions that are provided by typical PKI systems, but only authentication related operations, identity management and administrative functions such as introduction of new users and certificate authorities into the system, as well as removal. The closed environment has characteristics for easily performing control over every user, helping maintaining certain properties: that every user can be clearly identified, every user holds one certificate and its corresponding private key, and only the owner of the private key knows it, being this last one an assumption of PKI systems.

Besides the use of certificates and certification authorities, there is also a specification of a model for user credentials essential for this system. These credentials include two segments, a public and a private one. The public one contains the certificate of the user, which was signed by a trusted CA, and the list of the cryptographic algorithms that are supported for key generation, ciphering and signing of data. The private part comprises the private key and a list of trusted CA’s along with their public keys.

The authentication part of the system occurs in an authentication server which is assumed to operate under a bank or university environment, being therefore closed. The administrator of the server has the function of keeping an updated set of registered users in the system. With this list, the server keeps the certificates of the users, which are then subjected to a verification function when an user tries to authenticate that resolves the knowledge of the user to the its private key. The server also has a list of trusted CA’s and their public keys, a list of revoked certificates, the current date and a list of supported cryptographic algorithms. With these parameters in hand, the server performs operations for evaluating validity of the components of the system. For the digital certificates, it verifies that it is valid if:

1. The certificate was issued by a trusted CA;

2. The certificate has not been revoked yet and therefore does not belong in the revocation list;

3. The certificate has not passed its validity period, i.e., the current date is within the dates of the certificate;

4. The certificate is not modified since its creation date, verifiable by doing integrity checks, comparing with the signature of the certificate.

After these verifications, a certificate can be considered valid. Follows the verification of the user knowledge of its private key. The simple process to validate the knowledge of an user can be described in the following steps:

1. The server sends a nonce to the user;

2. The user receives the nonce and signs it with its private key. Proceeds by sending it the signed nonce to the server;
3. The server receives the signed nonce and checks the validity of the signature with the nonce and the public key of the user, retrieved from the copy of the certificate owned by the server;

4. This result of this evaluation will be the answer to the verification of the knowledge of the secret key by the user.

2.2.3.2 Certificates

In previous mentioned work, the specification of certificates is general, it doesn’t consider specific cases. So, to clarify the flexibility of those, this section aims for explaining in a more precise way some of the certificate types that can be issued and their purpose. To do so, this document will be focusing on the certification authority Multicert [17][16], the first enterprise to be accredited by the Portuguese National Security Cabinet as a Certification Authority for Qualified Digital Certificates, Applications and Services Certificates and Advanced Certificates.

Qualified Digital Certificates are meant for legal representation of entities or the owner itself in an online context. These can be divided into three different types:

- **Collective Person Representing an Entity** - This type of certificate is used for legal representation of a collective person or entity, where the issued person holds total or partial representation powers of the entity. In it there are contained the purposes of its use.

- **Certificate with Quality Mentioning** - Certificates that have the quality of the owner mentioned, for example, Engineer, Architect, etc. This is meant for authenticating or for digitally signing documents or transactions, stating in the signature the quality of the signer in the context of a particular collective person. If issued for an individual, the certificate contains the quality in which the owner will be authenticating to third parties and signing documents. It is also evidence of the role or position that the owner holds.

- **Individual Certificate** - A certificate to be issued to individuals, not linked to any entity but without the desire of having a quality appearing in the certificate’s information or when using it. It is meant to be used when authenticating or digitally signing transactions or documents.

Besides these qualities in the certificates, they all have some related aspects, where they all can also be used for digital signing mails for instance and all include a secure signature creation device, either this being an cryptographic card or a USB token device, for securely signing under physical presence. These devices, as well as the certificates, comply with the Portuguese and the European Union legislation and are issued according to the best known security practices. Also, in case of the Collective Person Representing an Entity type of certificate, if the owner for some reason is no longer fitted to use it, there’s a service for revoking it.

Besides these types of certificates, Multicert also issues applications and services certification, where one can: get TLS/SSL certificates for server authenticity; get certification upon its applications, for signing documents or identify VPNs, for instance; and can sign electronic invoices.
Finally, the last type of certificates are Advanced certificates. These are issued as stated by the ITU X.509 international standard, certificating applications for performing digital signatures of documents and e-mails, encrypt data, authenticate its owner and establish secure channels for communication with servers. The qualities that these certificates bring are for instance an unambiguous electronic identification of someone, or secure authentication. These are issued in USB Tokens, Smartcards and by download.

2.2.3.2 Blockstack

Naming systems have the function of providing human-readable names to objects or users online, identifying and linking them to entities or persons. Blockstack [13] is an example of a blockchain-based naming system implementation, which relates deeply to the goals of this work. It was conceived firstly on top of Namecoin's blockchain, where the authors deployed a decentralized PKI service that would allow users to choose human-readable names with a strong sense of ownership.

The idea of using blockchains as base for naming systems introduces several and important security benefits. Namecoin [12][27] was one of the pioneers on the subject of naming in blockchains, introducing decentralized namespaces and identities. Based on the Bitcoin's software, this system was created to prevent censorship, providing a decentralized infrastructure with no central authority. This would help prevent attacks and manipulations on web accesses, such as DNS blockage or DNS cache poisoning.

On top of Namecoin's blockchain, Blockstack creators built a PKI and an identity system, which was called Blockstack ID. For the publishing of public keys and profile data they used PGP, Pretty Good Privacy. Although Namecoin already supported human-readable names and name-value pairs, it provided a limited storage for each one, which the authors expanded by using linked lists.

Blockstack ID functioned for a year, where it was studied, collecting enough information for building their own blockchain-based naming system, the Blockstack. The collected information demonstrated necessary improvements on network reliability, in how to secure blockchains, and on how to deploy consensus. The Blockstack implementation was done on top of the Bitcoin's blockchain, for "no other blockchain even comes close to Bitcoin in terms of these security requirements", they state.

The design of the Blockstack system is done to implement functions that make a naming system on top of the Bitcoin's blockchain. To make this work, they had to deal with natural blockchain challenges: Limits on data Storage, Slow Writes, Limited Bandwidth and Endless Ledger. To outcome these difficulties, the implementation of the system is divided in two parts: the control plane and the data plane. The control plane is responsible for registering the names, creating the bindings between them and a hash, which belong in the data plane, and for creating bindings to owning cryptographic key pairs. On the other hand, the data plane is the part of the system that controls data storage and availability. It consists on two elements:

1. **Zone files** that contain the hash or the URL of the data;

2. **External storage systems** for storing the data values.
These data values are signed by the owners’ public keys. Clients can access and verify the authenticity of this data from the data plane by finding its hash on the zone files or by checking the public key signature of data on the data itself. This solves the first challenge, for it separates the data plane from the blockchain.

Another property imposed in the system is that the underlying blockchain is agnostic, meaning that there are no limitation upon choosing a blockchain, any could be used. It also allows data migration from one chain of blocks to another, which is very handy when deploying new versions of the software, for instance. The system also contains a state machine and a track of its states. This state machine is implemented on a layer on top of the blockchain called virtualchain, which will be later discussed. The system can also introduce new types of state machines without making any changes on the underlying blockchain, which prevents the creation of forks upon upgrade.

These divisions and mechanisms are represented in a layered architecture presented in Figure 2.4, where the blockchain is in the bottom layer, followed by the Virtualchain layer, the Routing layer and the Storage layer. These are explained throughout the section.

![Layered Architecture of the Blockstack system.](image)

The bottom layer would be the blockchain. This provides the consensus operations to assure order on the write operations.

Above, the virtualchain, which was already briefly discussed above. It lays on top of the blockchain and is the layer responsible for the Blockstack operations, where they are encoded to blockchain transactions as additional metadata. Blockchain nodes can only see raw transactions, for the logic of the operation is always processed in this layer. Only Blockstack nodes are aware of it. This layer is also responsible for accepting of declining operations, where accepted ones construct a database that keeps the state of the system along with state changes on blocks.

On top of the virtualchain comes the first component from the data control segment, the routing Layer. For performance reasons, this layer works separately from the storage of data, which allows the coexistence of multiple storage providers. As mentioned previously, the system uses zone files for storing the files paths, these files that have the format of DNS zone files. These files are stored in this layer that does not require a trust commitment, for the virtualchain layer stores binds between the names and their hashes, providing a verification mechanism in the control plane. Another quality present in this layer is that it only stores the zone files if its hash was announced already in the blockchain, white-listing data.

The final, top-most layer, is the storage layer. Here is where the data values of name-value pairs is
kept, which is always signed by the key of the owner. This is an improvement on the storage problems of the blockchain, for it saves the values outside of the chain. This allows values of any size to be persisted and a wide range of storage back-ends. As the routing layer, the storage layer does not need to be trusted, for the data integrity can be verified in the control plane.

There are two modes of operating in this layer, and their difference is the way the integrity of the data is verified. The first, *Mutable Storage*, is the default. The user has a zone file that contains a route to the data, which is constructed containing a signature with the user's private key. To read this data, the system fetches the zone file and the data, verifying both against the hash kept in the Blockstack and the user's public, respectively, guaranteeing integrity and authenticity of the data. This allows for fast writes and fast reads.

The second mode, *Immutable Storage*, is similar to the previous, except that it places a TXT record in the zone file. This record contains the hash of the data and is used to verify the integrity of the data. It is necessary to also verify the data's signature and the zone file authenticity. This mode is suited for data that does not change very often, improving the reading capabilities of the system. However, updating data stored in this mode requires a new transaction on the blockchain, for it modifies the file's hash, modifying the TXT record and, therefore, the zone file. This makes data updates much slower than while using the mutable storage mode.

The naming system in Blockstack works like a normal transaction in the Bitcoin system: first to write it, wins. However, there's a second step here, to prevent race attacks. The first step consists of preordering and registering a name. After it does both successfully, the user can update the name-value pair by performing a transaction and uploading the new value in the storage layer. There are other possible operations, the *transfer* operation, which simply changes the address that is allowed to perform transactions and the *revoke* operation, which disable any operations for names.

This system is implemented by defining the previously mentioned state machine and rules for the transition of states. The creation of names and namespaces has a cost in the system, thus preventing the creation of objects that are not used. The PKI also has a role in this part of the system, where each PKI has an .id namespace with a pricing function associated. This solves the low bandwidth problem.

However, there is not yet a solution to the endless ledger problem, where Blockstack nodes need to process more and more blocks to boot. This is solved by the use of consensus hashes. These can be calculated independently and help the nodes to locate themselves. By "using an untrusted database of state information combined with a trusted consensus hash", a node can bootstrap. From that point on, the new node can reconstruct the virtualchain using the database, doing hashes along the way. In the end, if the recalculated hash matches the consensus hash, the database can be trusted. This technique is much faster than fetching all transactions.

### 2.3 Confidentiality

The previously mentioned blockchain systems and identity management mechanisms provide notions for efficient and secure systems. However, a key aspect is missing: confidentiality. Although the first
blockchain systems did not require it, confidentiality is a requirement in other contexts and its assurance is crucial for clients to use applications feeling safe. Several contributions have been made to this area. In this work, there will be depicted two systems that are strongly related to this work.

2.3.1 Enigma

Enigma [15] is a decentralized platform that incorporates an off-chain network on top of a blockchain infrastructure, created for the development of applications meant to be private by nature. Enigma tries to solve some of the main difficulties related to blockchains: scalability and privacy.

The combination of the platform and the blockchain results in a symbiotic relation, where the work is split between them. The blockchain executes the public and less computational exhaustive part of the code, and Enigma executes the private and exhaustive computationally part in its off-chain network. The proofs of correct execution are later stored in the blockchain and remain public for auditing. By working together, these two achieve both privacy and correctness.

Enigma provides a scripting language for usage in contracts, where the contracts are capable of handling private information. These are called private contracts, and show great improvements in scalability: the code is not executed on every node. Instead, it is distributed across the network, where an interpreter splits the contract’s and assigns the public and private parts.

Scalability in Enigma is applied in more than just performance. The system provides an off-chain storage constituted by off-chain nodes that build a distributed database, where each one of those nodes have a different view of the shares and encrypted data. This data is encrypted on the client-side and can be found by looking for it’s reference on the distributed hash-table present in the blockchain, where access-control to that data is also programmed. This distributed database can also store public data.

Enigma’s privacy stands on top of secure multi-party computation based on secret sharing, distributing the computations, where no single node ever get access to complete data, but a meaningless part of it. For secret sharing, the platform uses a threshold cryptosystem that requires a minimum of parties to decrypt it. This secret is divided among all the participating nodes and when that minimum is achieved, then the secret can be reconstructed. This mechanism requires an honest majority to achieve privacy and correctness. However, if along side there’s a bound on the attacker’s computational power, there is no need for a majority, but fairness and the decision upon the output need it.

Private contracts also deploy new privacy qualities. The scripting language introduced is written using similar syntax to well-known programming languages, for usability purposes. This programming language also implements a new feature for privacy management: private data types, which are to be used to specify objects that are private, ensuring that when computed they remain undisclosed.

Data access in Enigma is accomplished by the use of three different decentralized databases that the system possesses, each on accessible through a dictionary. The first would be the public ledger, the blockchain, which is public and append-only, keeping the entire history stored. This can be accessed and changed, and is where the keys to the values can publicly be found. The second would be the DHT, or distributed hash table, which is where the actual data is saved. This data is encrypted locally
before transmission and has data access constraints related to privilege, where access can be limited
to a set of public keys or just the owners public key. If there is no encryption requirement for the data,
then the data is public but distributed. The final data base would be the MPC, or multi-party computing.
This database is used for sharing the secret. It includes mechanisms for sharing the secret to potential
computing parties and to reference the data for computation using both the key to the value and the
secret without revealing it.

2.3.2 Hawk

Since the start of blockchains, the potential was obvious and waiting to be applied. However, although
many good uses have been presented with this technology, some others remain in a more complicated
state, due to the fact that blockchain applications require a public part.

Running on blockchains we have smart contracts, programs that compute the transactions informa-
tion. These are the main concern related to the privacy of the systems’ users, for when running in a
peer-to-peer network, everything requires a certain level of disclosure, assuring the intervenients of
the safety and correctness of both the transactions and the code from the smart contracts.

Following these concerns, Hawk [14] was created: a programming framework for private smart con-
tracts, with the goal of providing an intuitive, familiar programming language, that is meant for not-so-
experienced programmers.

Hawk provides a compiler from the already existing compiler PINOCCHIO (PGHR13), that compiles
the users program into an arithmetic circuit that will later compute a zero knowledge proof. This concert
is the key to Hawk, that works by hiding the sensitive information until a certain point in time, making
sure that no one on the network is able to see the information.

The system is better explained through an example: an auction where the winner would pay the
price of the second bidder. In a system such as bitcoin, this application would not be possible, for as
soon as any user bids on the object, the action becomes a transaction and it is made public in the
blockchain, where any user would know the winner’s paid value (financial disclosure). In Hawk the
process is different, keeping the value secret.

In an auction program written using the Hawk framework, the program would take as parameters
zero knowledge proofs, which would come from each user by taking an encoded part of the function
they are going to run. The bids are then sent to the smart contract as commitments (and not plaintext)
by every user, alongside with the zero knowledge proof to assure the money is spendable and has not
been spent yet (the double-spend problem). After a certain period of time, the auction would terminate
and an evaluation upon the bids is made.

The evaluation of the values takes part in a new logical component, "the manager", a minimaly trusted
party. The commitment is encrypted with the manager’s key and sent to the manager to decrypt, as well
as it is sent to the blockchain. Then, it compares the values and sends the refunds to the losing parties,
and the change (what money there was left) to the winner. This is sent to the blockchain along with a
zero knowledge proof to ensure the correct program execution.
The weak point of this model is the fact that there is a need for a trusted third party, which is what every blockchain is trying to avoid. However, this third party does not need to be trusted for:

- **Correctness** - the manager needs to send a zero knowledge proof, which ensures the correct execution of the code;

- **Input independence** - the manager could be a bidder as well, where if he would see the values before they were evaluated he could leverage that. However, since all the inputs are only sent to the manager after the bid is closed, this is not a concern;

- **Security of the currency** - the Hawk framework is built on top of the Zero Cash platform, which ensures that no e-money can be affected by the manager, in the eventuality that this get corrupted;

- **Fairness** - if the manager failed to complete the task, then this would be publicly known. By requiring collateral deposits from all users, the contract would then refund every user in such a case.

The third party does need to be trusted to prevent disclosure of the information posterior to the bid. Since, during the evaluation stage, the manager has access to the bids information, this could leak the bids data, if it became corrupt. However, in the eventuality of this situation, if there are several instances of the same program running, it does not mean that every execution is corrupted, only one is, while the others remain secure. To prevent this situation, the writers suggest implementing the manager with multi-party computation or trusted hardware.

### 2.4 Summary

This chapter presented several systems and approaches that could serve as either parts of the system or base for development.

The first subject to be presented was the blockchain, starting with permissionless blockchains, moving to permissioned ledgers.

This was followed by identity management systems and techniques for authorization. The first systems mentioned are Single-Sign-On systems, meant for authorization between different entities with the same credentials. The next subject was public key infrastructures, which was finalized by the mentioning of a blockchain system that incorporates blockchains.

Finally, this last sections mentions confidentiality in blockchains, mentioning two systems that implement confidentiality mechanisms in blockchain systems and smart-contracts.
Chapter 3

The Infrastructure and the Application

On the present chapter, we will start to take a deeper look at what the document is about. In Section 3.1, we present the architecture behind the system and then, in Section 3.3, we describe the prototype developed, the application Notary.

The considerations explained in the previous chapters led to the development a system that would provide:

- **Persistency of records** - Records of interactions with the system should be persisted and secured: the use of a blockchain;

- **Network management** - Although using a blockchain, this is a permissioned network, where the number of nodes present should be the one that the administrator decides best suits the needs of the system;

- **User management and identification** - The users’ information should be able to be modified and added, giving special attention to identification, which is to be updated as needed. Lack of valid identification due to being expired or revoked means the system can not be used;

- **Practical, intuitive interface** - Both the user and the manager should not have to deal with an interface that requires experience. The interface should be graphic and simple, but complete. Also, the ideal would be an application that would not require a lot from the users platform, such as a web application;

- **Platform flexibility** - The advantages of having a one-use-only platform are meager. The platform should be flexible enough to allow different types of applications and, if necessary, to be extended easily.

- **Privacy** - The overall system will be dealing with information related to user interaction with the application artifacts, where some of the data transmitted might be sensible and private. The means to prevent information leakage are a big concern, both from an internal and external point of view.

From these needs, we developed MTChain (Multicert Chain), an infrastructure formed by four logical components, which merge from a lower level – the blockchain, to two higher levels – admin and user
clients, being connected by the platform itself, where this acts as server for the clients and as a client to the blockchain. Combined, these components achieve the desired goals, providing persistence to information in a decentralized and secure way. By communicating through an interface, the users and the admin interact with the platform (MTChain) that will be the gateway to the blockchain, either for alterations on the logic of the chaincode (admin), or for persist the records in it (users).

To demonstrate the system’s functionalities, and to satisfy one of the objectives - the development of an application prototype that is described in the final chapter of this section: the Notary.

In the following sections, the implementation of these components is explained in detail, both individually and as a whole. The final section approaches the prototype - The Notary.

3.1 The MTChain Infrastructure

This section addresses the complete architecture for MTChain, which is represented in Figure 3.1. A more specific, simplified architecture is also demonstrated in Section 3.3, in Figure 3.3. This scenario corresponds to our deployment of the Notary application with MTChain.

The system can be decomposed into four distinct components: Hyperledger Fabric Blockchain, MTChain, the applications side and the administrator application. Working together, these components create the overall infrastructure: a platform that communicates with a blockchain for managing and recording purposes.

The Hyperledger Fabric Blockchain is the component responsible for the persistent storage of the records necessary for the validation of operations. Inside the abstraction, there is one or more groups of nodes - organizations. Each organization works individually or along with other organizations to achieve consistency in records and execution correctness. Smart contracts – chaincode in the hyperledger fabric context, run in the nodes inside the organizations, indicating the functions to perform on the data received. These nodes save the records in the blockchain, assuring the blocks are inserted in the chain correctly, checking on the execution of the contract and the order of the transaction. This assures safety and correctness. The records stored change according to the application, where in the Notary application we write timestamps, document hashes, signers, and other information.

The records to be saved in the blockchain come from the platform, MTChain. This works as a server side to the applications, receiving requests from both the administrator and the users’ clients. To do so, it implements an interface to be accessed through the browser (users’ computers). Issuing requests may result in three different actions:

- **Simple html page load** - As any web application controller, the server has the function of loading certain parameters to web pages.

- **Database interactions** - When a request changes or requires server information, such as adding a new user or updating the chaincode, this leads to an interaction with a database.

- **Blockchain interactions** - The overall goal of the platform is to persist records in the blockchain, where some server requests lead to that. These interactions can be of different types, depending
on the issuer of the request and on the request itself: the administrator operates on the blockchain, updating the logistics behind its functioning; the applications operate functionally on the blockchain, querying it or saving transactions on it.

Besides these functions, MTChain also comprises the means for verification of identity and validity: it is responsible for the authentication of the users and the administrators, as well as the verification upon cryptographic operations. However, some information is better kept away on the clients side (application users’ private keys, for instance), forcing some of the cryptographic work to be done on the browser, such as file hashing or signature of artifacts.

Separately, MTChain provides interfaces for applications and an administrator, as previously mentioned. Starting with the administrator side, it has the function of providing the system with the information for identity management - the users certificates and/or credentials; the function of managing the blockchain services - the chaincode versions and the applications that run for each chaincode; the function of verifying the blockchain nodes, connecting or disconnecting them if so is desired.
On the other side, the applications use the functions of the chaincode. Acting as the front-end, the application is the service that interacts with the platform for chaincode calls and identity checks on users. These are authenticated using information from the data base. The nature of the platform allows for multiple different applications to run simultaneously. It is also possible to run the same application with a different chain of blocks. The resulting interactions with the blockchain will be the ones allowed by the programmer.

3.2 The components

Describing the infrastructure using a layered view, as seen in Figure 3.2, it is possible to verify that the base of the system is the blockchain network, Hyperledger Fabric. The platform was built on top of it in order to provide the required features of the system, leading to both the administrator application and the users applications.

The following subsections approach each of the components and subcomponents individually, explaining the decisions made behind them and their functionalities.

3.2.1 Hyperledger Fabric

The first component to mention would be the bottom one: the Hyperledger Fabric blockchain. Many options were taken into consideration for the picking of a system that would provide a blockchain. However, due to the evolution seen in the past by Hyperledger Fabric, as well as its documentation, IBM endorsement and the fact that it is a permissioned blockchain network, we decided to take this system to provide us with blockchains.

Hyperledger Fabric works by grouping different sets of nodes into organizations. The nodes, as stated in section 2.1.2.1, are of three different types:

- Endorsing peer nodes (or just peers), responsible for running the chaincode and hosting the ledger, executing the calls from the client (MTChain, in this case) and store the records in the blockchain;
• Ordering nodes, responsible for running the consensus protocol, synchronizing the peers’ chains;

• CA nodes, providers of identification, creating certificates and personal keys for every peer in the network, and the clients - admin and several users, if desired. CA peers enable a trust notion on the network, as well as the possibility for the usage of the TLS protocol. In future studies over this component of the network, an alternative to the Hyperledger Fabric certificates would be the generation of these by Multicert.

Chaincode runs on the peers, a program that receives several parameters - one of which is the function to invoke inside the program, and tells the peers what to do:

• **Query** - Resquesting Hyperledger for a record or a set of records, querying by an identifier defined by the programmer of the chaincode. Any peer authorized can respond, making this operation fast.

• **Update** - Resquest to persist some record in the blockchain. This is a spending process, due to the necessity of running the consensus protocol. Updating the blockchain means adding a new transaction (or set of transactions) to a block. This is, once again, relative to the implementation of the chaincode proposed by the programmer.

However, in order to use the Hyperledger Fabric system, chaincode needs to be in a certain point of functioning. To reach this point, several steps need to be performed by the administrator:

1. Start an **install** operation. This will upload the chaincode in the peers;

2. **Instantiate** the chaincode. This will run the program inside the peers, initializing the chaincode. This is a heavy operation, for it requires consensus.

3. (Optional) **Upgrade** the chaincode. If a new version is needed, the chaincode must always be upgraded, for Hyperledger does not support a **stop** nor a **delete** operation.

4. (Optional) **Instantiate** the upgraded chaincode, upgrading definitely the version.

Although not possible to remove installed or instantiated chaincode, it is possible to use an old version, since functional operations require a chaincode version input, giving the administrator of a fallback option in case of bugs or other software development related problems.

Chaincode installations are able to generate dependencies amongst nodes and even organizations. This is what is called the endorsing peers, where peers must endorse transactions: they must validate that the transactions. In these cases, it is possible to choose a set of peers or give priority to some over others. The objective of this is to have a set of peers decide that a transaction is valid, instead of having a policy that allows only one peer to verify the transaction execution to accept it.

Hyperledger Fabric supports a dynamic number of nodes, with no limit to the organization size or the number of organizations. Therefore, there is no restriction to the implementation of the work described in this document.

Running a client on top of Hyperledger requires the creation of channels, which link applications to one or more organizations. This is the connection element, where applications can only communicate
with the peers and orderers defined in that channel, acting as a high level socket between Hyperledger Fabric's network and the client – MTChain, in this case.

The creation of a channel consists on running a set of binary tools from Hyperledger that, from xml input files, defines the channel characteristics:

- The channel name;
- The organization characteristics;
- The first block of the channels blockchain;
- To which organizations, peers, orderers and CA peers it will connect;
- The information regarding the organizations identification;
- The information regarding to which each peer will be working with.

Overall, the implementation of Hyperledger is not static and can be set into a protocol:

1. The definition of a network, by running the binary tools from Hyperledger;
2. Starting the peers, the ordering nodes and the CA peer(s);
3. Uploading the necessary files to the running nodes;
4. Order them to start according to the configuration file resulting from the binary tools;
5. Definition of the channel that will connect a client (or more) to the network;
6. Installation of the chaincode;
7. If no error messages, the network is ready for usage.

### 3.2.2 MTChain

The platform is the middle layer and the connector between the users, the administrators and Hyperledger's network. It is divided into several subcomponents that form MTChain itself, along with other tools.

MTChain is the provider of the web service and the communicator of transactions to the blockchain. Hyperledger Fabric SDK (in Figure 3.2, HLF SDK) is used when communicating with the peers from Hyperledger Fabric. To perform its cryptographic functions, MTChain contains a cryptographic module. To persist information that is relevant for the use of the Hyperledger SDK, for identity management and for inner logistics, there is a connection to a database.

In the following sections, these complements are described and justified.
Hyperledger Fabric SDK

The implementation of the Hyperledger Fabric SDK on the platform consists of using a set of methods and objects to create a communication link between the two components.

It provides a set of utilities to define what is, for example, a chaincode or a channel. By defining these concepts in Java, the SDK allows the platform to create a connection, from where it will send requests and receive answers. It also allows the definition of transaction parameters, creating the means to manipulate the chaincode and update/query the blockchain.

The process of using HLF’s SDK starts by informing the platform of the nodes active in the network. This is done by proving the system with a YAML file that contains not only the channel and peers names and addresses, but the location of the identity certificates (for the Hyperledger nodes and for the platform itself, allowing it to authenticate) and the configurations of the connection - for example, if it uses TLS or not. This file is read using a specific method from the HLF SDK, that converts the information to JSON and, optionally, afterwards, to an object of type `NetworkConfig`, an implementation also from the HLF SDK. An alternative implementation allows for the definition of the network programmatically, by defining a channel manually, setting the nodes that will participate in it.

From this object, it is possible to create a channel between Hyperledger and the platform. The file contains the channel name, that is added to an object that, upon initialization with the channel name, it searches the nodes and starts a connection with them.

Resorting to the created channel, operations on the blockchain and chaincode are executed. The channel associates to a `ChaincodeID` object, which contains the chaincode information so that the peers know what to run the call against. This object will be added as a parameter to a `TransactionProposal` object, along with the necessary parameters to be read on the chaincode.

The `TransactionProposal` object returns with the answers. Depending on the type of operation called, this proposal might be sent to the ordering nodes as an envelope (blockchain update or chaincode instantiation) or be read for information (blockchain query or chaincode install/upgrade).

Hyperledger Fabric SDK is now available in two different programming languages: Java and JavaScript. The one used was the Hyperledger Fabric Java SDK, for we had more experience in such language. Also, being this a company associated project, the staff favors from the use of Java, which is also a more commonly used language. However, the Java SDK is still in development and presents some still unavailable features. It also presents lack of documentation to this moment, both for the overall SDK and for the configuration files.

Cryptographic Tools

When privacy and security are concerns for a system, the use of cryptographic tools is essential. These allow information to be kept secret throughout communications, to verify the authenticity, integrity and freshness of that information, among other features. It is nowadays crucial to use such tools, protecting the end-user and providing a reliable, trust worthy system.

In MTChain, cryptographic tools are used in more specific moments of the execution flow. These are
provided with a higher focus on information integrity and validity than on keeping it secret, that being a concern on the client-side.

The platform contains within its functions the ability to perform and check signatures, where at the start of its execution it loads a pair of RSA asymmetric keys (the same used for the TLS communication protocol, discussed in the interface section). These are loaded from a Java keystore (JKS) and separated into a Certificate and a PrivateKey java objects, using the latter to sign messages.

To verify user’s signatures, it retrieves the user session, associating a request to a precise user. By doing so, it will be able to query the database for the user’s certificate. Due to restrictions on the message and signature format on the client side, the server provides verification and signature on both byte array and hexadecimal types. To be able to use any of these types and enforce interoperability, this cryptographic module provides conversion methods.

The last feature available with the implemented cryptographic tools is the computation of the hash of a document.

All these methods made available by the cryptographic module of the platform represent the needs of the business logic so far, where there was no need to implement other functions (encryption for example). Other methods could be implemented later, upon necessity.

The database

As any system, some information need to be persisted to create consistency among nodes, a connection between components and a fallback in case of failure.

To provide this persistence, MTChain connects to a database that keeps information regarding the parts of the system:

- Users information;
- Admin information;
- Chaincode information;
- Applications information;
- Others, relative to the information the applications need to store.

The database service in the current implementation is provided by PostgreSQL, an object-relational database management system. A database server is deployed outside our system, to which the MTChain connects to in the beginning of its execution.

The accesses to the database done by using Spring Data’s JPA repositories, an abstraction layer upon the data accesses to minimize written code. Through the use of entities and the repository, queries and database updates are facilitated. Entities are what correspond to the tables in the database, which are created in Java using Springboot annotations. These contain the fields to persist, along with methods to use its variables and configurations to provide to the database.
To access these entities, repositories divide into two parts: the repository and the repository service. The repository has direct access to the database, which queries it, allowing for native queries, using annotations and *SQL strings*, and *JPQL queries* (Java Persistence Query Language), a query language based on *Hibernate Query Language*. The repository service, on the other hand, is an abstraction of the repository, corresponding to an implementation of the necessary logic regarding the database, reducing some of the boilerplate code on MTChain’s database operations.

The creation of the tables is done when they are first accessed. To clarify, when MTChain tries to retrieve information from any entity, if there is no corresponding table, a new one is created.

Each entity has its own purpose:

- **Users information** - the database persists the identification of the users. Information on this table is updated by the administrator. The most relevant information contained in it is the user’s name, application and the certificate. These are used for user identification, authentication and cryptographic verification;

- **Admin information** - admin information is added in the database to identify admins. It is the same principle as the user, but with restricted accesses.

- **Chaincode information** - to keep MTChain information up-to-date regarding chaincode versions, the database saves information regarding each installed chaincode. Since communication between Hyperledger Fabric network and MTChain require that information about the chaincode is provided, the records on the database are kept consistent and updated, so that several nodes can synchronize and recover from failures.

- **Applications information** - A simple association of a chaincode to an application, where an application connects with a specific chaincode.

The database is not static. New tables can be added to it, according to the needs of the applications. However, at the moment, it consists of a centralized part of the system. It is implement so that only one server is used. The decision behind this started by defining the type of database to be used, which began by suggesting a peer-to-peer system with other nodes running MTChain that would synchronize their file system, creating persistency. However, this would increase drastically the complexity of the platform. Based on this, a PostgreSQL was chosen. This can also be replicated, removing the centralization. Besides that, and being this a company project, it is also a regularly used system on Multicert.

Nevertheless, the database represent a single point of failure in the system that requires an extra level of trust. To prevent disasters, replication methods should be applied and other solutions considered. In future implementations, the best approach would be the implementation of the previously mentioned peer-to-peer network that would contain the necessary data stored among the peers. Until that point, a replicated PostgreSQL database located in a secure, trustable environment is the best solution to prevent data loss, where this data would be encoded to prevent discloser of information.
3.2.3 Application and Admin Side

The top layer of the system is the application layer. In other words, it is the layer that is accessible to the users and the administrators, the clients for the server MTChain. This layer provides the GUIs and it is divided between two front-end components: the applications and the admin application. These two are described in the following sections.

Web Service and Authentication

Applications work as servers for the web services, generating a point of interaction between browsers and the MTChain, as seen in Figure 3.1, in the Application components.

To provide service, applications use Spring Boot, a Spring project that produces stand-alone applications, aiming to abstract the configurations of web services, reducing the code and creating an intuitive construction process resorting to code injection.

These applications run Spring Boot using Apache’s Tomcat application server, which is responsible for the usage of the TLS protocol to communicate with the browser. It also hosts the files necessary for running the front-end and back office.

Spring Boot provides a set of features that manage server's identity management and authorization: Spring Security. From a configuration file, the programmer defines what are the limit zones for users and/or administrators to be in while authenticated. This protects files, pages and services from unauthenticated personnel. It also allows for the definition of public zones: pages that can be accessed without any need for authentication.

In applications, authentication can be performed in three ways:

- **Username and Password**, where the server keeps the username and the password's hash. The password would be hashed along side salt, which would be a value stored in the database. If desired, it would be provided a stronger method of hashing the password that would be hashing the password with the salt for a N number of times, N being stored in a configuration file;

- **Certificate**, where the server keeps a certificate of a user. When entering a page, the browser automatically requests a valid .p12 file, from a tomcat request, which is configured to authenticate the client too for the TLS protocol. Tomcat provides the certificate to Spring Security that compares it to the certificates stored in the database. If valid, it authenticates.

Tomcat has its own key pair as well, in order to use TLS. This key pair is also loaded into the server and decomposed into a private key and a certificate. It is later used for signing messages to be kept in the blockchain.

The interface provides the services necessary for the front-end. Both the backoffice and applications are described in the following sections.
Backoffice

Accessible only to the managers of the network (or administrators), this application runs on top of MTChain. It is used through the browser and accessible inside Multicert’s network, in order to mitigate attempts of unauthorized access.

By performing interface calls, the administrators manage the network, installing and upgrading chain-code, assigning applications, editing and adding application users, and checking the clockwork of the system.

This application is the only one that is static: only one backoffice application can exist. It can be altered, programmatically, and replicated. Through it, administrators can find a GUI to manage the system. This interface is constructed using HTML templates, JavaScript and CSS. These templates were based on already existing templates that use Bootstrap, a framework based on HTML, CSS and JavaScript for creating responsive, mobile web sites and web applications.

Applications

Available for users registered in the system and already added by the administrator, applications represent a more flexible side to the application layer.

Many applications can fit in this layer. The idea is that it suits the necessities of the developers and the clients by developing only a small part of the back-end and the front-end. To allow so, the platform might require some extensions to its modules.

It is up to the business logic of the application to define what a developer will make out of the application. Many types of applications can fit the platform, as long as it requires a blockchain. The stronger example at the moment is *The Notary* application. The following and final section of the implementation describes this in detail.

### 3.3 The Notary - The MTChain prototype application

The implementation of MTChain was accompanied by the creation of a prototype. It implements an application called "Notary", which aims to emulate a notary service, decoupling a verification third party from the document signing process. In this section, we describe the application, how it was implemented, the features available and the decisions made upon development. The specific architecture of the prototype can be seen in figure 3.3.

#### 3.3.1 The Notary Application

A notary is an entity whose profession is to perform acts in legal affairs. One of it’s most common functions is to witness document signings, acting as a third party that verifies a set of conditions, such as hour or intention.

The notary application aims to provide an emulation of this entity. The clients, instead of dislocating themselves and paying for the service, have it available from their web browser. Besides, since the no-
Figure 3.3: Notary: The architecture of the prototype

Overall, the notary application is a web application that will have a certain set of users. These users are able to upload documents to be signed, choosing the other users that sign it. Sequentially, each of the chosen users sign the document. The notary application works not only by allowing users to sign the documents, but also signs the document itself, creating a cycle of signatures with points that stamp a document signature process: start operation, signatures of the clients and end operation.

In figure 3.4 we can observe a flow chart of the document signature process.

The whole process is divided into slices of activity:

1. **Start Operation**: The platform receives a hash of the document, which it signs to prove the oper-
ation initialization time of the signing process and to store the original hash of the document.

2. **Sign Operation:** Then, one user at a turn (these who were earlier selected by the document issuer, upon document upload) will sign the document. After every user who was assigned to sign is done, then;

3. **End Operation:** The platform finishes the process, by signing and closing the document, proving again the time of finalization.

In the end of the process, the document is signed and finished. The signature should be one over another, to prove the previous one, as seen in figure 3.5, so that each signer endorses the previous and the next signer (where they know who signed before and who will sign next).

The application front-end shows four main sections: **User information**, which presents information about the user of the application, such as name, certification expiration date and documents to sign; **Documents to sign**, which present a list of documents that the logged user has to sign; **Start new signature**, that allows for the upload of a document and selection of the users that will participate in the signing process; **History**, which presents the documents that the logged user has ever participated in and if they are concluded or not.

The following sections explain the last three points in detail.
Figure 3.5: Notary: Document after its signing process is finished. Each "{}" represents a signing over the inside information. The document is not the actual document, but its hash.

Start new signature

In the cycle of a document, the first step is to upload it to the system, so that operations can be performed on it. This is done in the first section of the web application described in this document: Start new signature.

In this section, an issuer of a document can upload it for signature. It is required that the issuer user defines a title and a description of the document, so that this becomes identifiable by other users upon signing time and by itself when searching the history. It is also possible to select other users to sign the same document, while this is not mandatory: an user might just want to register that it signed the document. The menu can be seen in Figure 3.6.

![Start new signature Menu](image)

Figure 3.6: Start new signature Menu, for document description and upload.

This information is inserted in an HTML Form that posts to the server. However, since privacy of this document is the main concern of the section, this will not be sent to the server, only its hash. The file is executed in JavaScript, that will compute its hash and insert it into the form.
On the server-side, this form is translated to a new Document object (which was introduced in the system to persist document information). The document hash is then signed by the platform, performing the start operation on the blockchain with the document hash and the signature.

After this initial process, the document is ready to be signed by the users selected.

**Documents to sign**

The second part of the process is the document signature by the issuer and the selected users. After the process has been started by the platform, one of the signers at a time will receive the permission to sign the document.

In this section, each application user can find the documents in which it is their turn to sign. This section can be found empty if no documents are to be signed at that moment. Each document will come in a different container in the page, where each will have the document information:

- **Title** - the title of the document;
- **Description** - the description of the document;
- **Already Signer by** - a list of users who have already signed the document;
- **Next Signer** - the signer that will take part on the next signature, after the current user signs;
- **Remaining Signers** - the signers left to sign;
- **Date** - the date of the upload of the document into the system.

The first signer will always be the issuer of the document, which can be the only signer or just the first signer. Following that, there is no criteria for the decision upon the following user to sign.

The signature of the document happens when the user presses the document name. This will open a modal box that will contain a form with two fields: a file input waiting for a .p12 file containing the signing key and the password to access that key. This will later be executed in JavaScript along with the signed hash of the document, producing a signature that is sent to the server. This process repeats for each of the signing users upon their turn.

On the server side, the signature is received and verified against the authenticated user’s certificate. If it verifies, the new last signature is updated on the created Document object and then the signature stored in the blockchain.

In Figure 3.7 we can see the menu where to be signed documents are shown.

In case the last signer signs the document, the platform will automatically sign the last signature and store it in the blockchain, finalizing the process of signature.

**History**

Every document information is always made available to their signers. The platform performs queries on the blockchain to provide the users with information on the documents signatures. Figure 3.8 demonstrates this menu.
The section History is much like the Documents to sign section. It displays the same information, with the exception of a Status field, that describes the current state of the document: if finalized (and at what date it was finalized) or not. Besides that, pressing the document name will move the page to the document's signature timeline, which will display in a timely order the signatures on the document.

The timeline of each document contains all of its current signatures, the original documents hash and the document name in a user-friendly view. Each signature contains:

1. Signer - the user who performed the demonstrated signature;
2. Signature ID - the identifier attributed to the signature;
3. Signature - the signature presented in hexadecimal;
4. Date - the date (day and hour) at which the signature was performed.
3.3.2 The Blockchain

On the blockchain side, the Hyperledger Fabric network will run a chaincode specific for this application, which has three main updating operations corresponding to the signing moments on the platform:

- **start_op** - the operation that stores the first transaction on the blockchain, indicating the initialization of a process;
- **sign_op** - the operation that stores each of the users signatures on the blockchain;
- **end_op** - the operation that stores the last transaction on the blockchain, indicating the closure of a process.

Apart from start_op, which also receives and stores the original document hash on the blockchain, each operation receives as parameters the signature id (for querying purposes), the signer’s name, the signature and finally the time of the operation.

There is another simple operation, **query**, which retrieves documents’ informations from the blockchain based on their signature’s id. This id is provided by the platform.

3.3.3 A flow demonstration

Let us now consider a situation where a new document is uploaded to the system by user “Mark”, which selects “Elton” for signature too.

Assuming both users are already added to the system, logged in; and the chaincode already installed and instantiated, it follows the flow of operations upon a document and the blockchain from the application.

**Start new signature**

Figure 3.9 represents the flow of the system when starting a new document signature. Here, the user Mark uploads a document to the web application. The browser hashes that document to a SHA1 hash and sends it to the server, along with the informations regarding the remaining signing users (Elton, in this case), title and description. Once on the server side, the hash is signed and stored in the blockchain along with the signature, a ID to the signature, the signer (in this case, the platform) and a time stamp.

**Documents to sign**

In the section “Documents to sign”, users find the documents they are assigned to complete at that moment. In figure 3.10, Mark signs the document. Since he was the one to upload it, he is the first to sign it. By providing his .p12 key file and his password, the browser performs the signature using SHA1withRSA algorithm upon the already signed hash (signed by the platform). Then, sends it to the server. There, the signature is verified against Mark’s certificate and, if valid, it is inserted in the blockchain.
After that, Elton is able to find the same document in its "Documents to sign" section. There, he signs the document by completing the same steps as Mark, as seen in figure 3.11. Once again, the browser computes the signature value from Mark's key and the already signed document hash, signed by Elton and the platform at this point. The platform also acts the same way, with the exception that it has now...
Figure 3.10: Flow chart of Mark signature on the document.
the responsibility of finishing the document, signing it once again.

In figure 3.12 there is a flow chart representing the final operation. When the last signer user signs the document, the platform needs to close it. To do so, it will sign the hash already signed by the platform and all the users and save it in the blockchain, storing in the persistent store information that will block any other attempt to sign the document.

The transactions succession and timeline

After all the signatures have been completed and the document is closed, the order and content of the transactions is the one depicted in figure 3.13. Start_op receives the original document hash and the same context as the other transactions: the operation that led to that transaction, the ID of the signature, the signature itself and a timestamp.
Figure 3.11: Flow chart of Elton signature on the document.
3.4 Summary

This chapter describes the development and architecture of the infrastructure.

The chapter starts by explaining the designed system, a platform that communicates with two different entities: the clients and the blockchain network, provided by Hyperledger Fabric. This chapter goes over two architectural designs, logical and layered, mentioning each one of its components and modules, and how they interact together.

The second and final part of the chapter describes the prototype developed to experiment with. This prototype *Notary* represents an application that performs the functions of a notary entity. In this section, the example is explained and mapped into the solution presented in the previous chapter.
Figure 3.13: The transactions sent to the blockchain after the signature process is finished.
Chapter 4

Experimental Evaluation

This chapter describes the evaluation done to the system, MTChain. To perform this evaluation, several tests are run using the implemented prototype, the Notary application.

The goals of MTChain are split into two main subjects: security and performance. This being, the evaluation was designed to answer the following questions:

1. Does the system allow the necessary conditions for user privacy, identification and overall security?
2. Since interoperability is a constant topic in this document, does it provide acceptable performance?

The following sections address these two questions, starting with the security and identity concerns, moving to both latency and throughput measures.

4.1 Security and Identity

When considering security measures to protect both the system and its users, the best approach to take is to use already defined/implemented methods that are assured by trusted entities.

This system is no exception: security tools were not custom made, but instead reused, where security is assured by the usage of already existing methods and objects.

Privacy

In this specific system, privacy is one of the most important feature in terms of security. Losing privacy might mean that a private, sensitive document is leaked to the outside or to the platform itself, which is undesired. To ensure this will not happen, both the platform and the applications' developers work together, where both apply logics to prevent this from happening.

On the server side, the platform is only ready to deal with documents' hashes and public certificates. This ensures that the documents signed and uploaded to the client never leave the client, not even to share with other users permitted to view them. This is allowed, for logically every user should have the document which they are going to upload/sign.
On the client side, to avoid the transmission of the document's sensitive information, when a user uploads a document and presses the upload button, it does not trigger the POST from the form. Instead, it triggers a JavaScript function that will retrieve the document from the form. This document will be encoded and hashed with the SHA256 algorithm, which is then sent to the server in a hexadecimal format.

With these two considerations, the privacy of the user's documents is not violated. Allied to that, the server keeps only the public certificate of the user, which does not disclose the user's private key, but used to verify the necessary information.

The communication over the network between the client and the platform is also done using the TLS protocol, encrypting messages, preventing evesdroppers to see their content.

Confidentiality

To ensure confidentiality, all the information transmitted from the client and to the client is sent over an TLS secured channel. The same happens with the communications between Hyperledger Fabric and MTChain, even though this information is auditable and would not disclose data.

The client side is also authenticated, where the server requires also the client's certificate in order to accept communications with it.

Authorization

Authorization over services requires authentication of the users. This mechanism is to be done as the application's programmer choose, but the standard is the use of the user's certificate, whose CA is located in the server keystore and the certificate itself generated by the browser from the .p12 file that contains the private key.

This is later compared with the system owned certificates. If it returns true, then the user can be authenticated.

To improve this mechanism, the use of a username/password is also possible, where this would preferably be kept in the server after being hashed with a salt value an $n$ number of times. $n$ which would be kept in a configuration file.

Non-Repudiation

All communications and interactions with data are done with users that are authorized in the system. This authorization requires the passage through an authentication process that is associated to their certificate. This allows the system to always be able to trace back to users for miss conduct and attempts against the integrity of the system or the system artifacts.

A practical use of this property is in the Notary application, where each signature is always traceable to its signer, where these are verified against the authenticated user's certificate for veracity.
Database data breaches

The database accesses and privileges are different depending on the type of user that accesses it. Also, username/password based authentication is required to access the data.

4.2 Performance

Performance over a system that comprises web applications is a fundamental concern. Therefore, the need to reach a certain acceptance level is crucial and must be tested.

The most regular metrics of performance are latency and throughput, which are exactly the tests that will be performed, with different variants. These tests were performed for both the Notary application and the Admin application.

The environment for these tests is done using:

- A single instantiation of the admin application;
- One organization containing one ordering node and two peers.
- A single instantiation of the Notary application;

All the values found in tables result from a media of thirty measurements. The tests were run on a localhost machine, a Toshiba laptop with 16Gb of RAM and an i5 Intel CPU. The whole system ran in this machine, disregarding the internet latency from the measures. Tests were automated, ran by calls and specific waiting times, discarding user’s delays in interactions.

4.2.1 Admin application

Understanding the administrator application, the operations it performs are simpler but require more time to be finished than the ones in users’ applications. The processing is mostly located on the Hyperledger Fabric, creating a waiting period on the MTChain administrator, a bottleneck in Latency. This is explained in depth in the following section, which evaluates the administrator application in terms of latency.

Latency

On the administrator side, it would only make sense to test only latency, once throughput can not be affected by concurrency or other variants. So, in this section, we take a look over the operations that the system performs on the administrator side: the installing and instantiating of chaincode, and the upgrading and instantiating of the chaincode. These operations are tested along with the size of the files being installed, which provide the comparing factor amongst different times.

On table 4.1 we can observe the first tests performed upon the system: Latency resulting from the installations of chaincodes with different file sizes. Each one of the left names represent a different chaincode file, each with different complexities, increasing in an ascending order across the table. Installing chaincode involves an install and an instantiate operation transmitted to the network.
<table>
<thead>
<tr>
<th>Chaincode</th>
<th>Size</th>
<th>Time</th>
<th>StandardDeviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc</td>
<td>2.8kB</td>
<td>49s</td>
<td>3.56s</td>
</tr>
<tr>
<td>fabcar</td>
<td>6.4kB</td>
<td>47s</td>
<td>3.43s</td>
</tr>
<tr>
<td>marbles</td>
<td>24.9kB</td>
<td>47s</td>
<td>4.14s</td>
</tr>
</tbody>
</table>

Table 4.1: Time relation of chaincode installation related to size of file

From the resulting time, we can verify that the file size and the complexity of the chaincode are not affecting factors when installing. This is due to the nature of the network, where each peer compiles the code and communicates to the orderer that they did so.

Next, we evaluate the latency of upgrade operations with the same files as the previous tests. In table 4.2, we take the same approach as before:

<table>
<thead>
<tr>
<th>Chaincode</th>
<th>Size</th>
<th>Time</th>
<th>StandardDeviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc</td>
<td>2.8kB</td>
<td>47s</td>
<td>3.17s</td>
</tr>
<tr>
<td>fabcar</td>
<td>6.4kB</td>
<td>45s</td>
<td>3.27s</td>
</tr>
<tr>
<td>marbles</td>
<td>24.9kB</td>
<td>47s</td>
<td>3.14s</td>
</tr>
</tbody>
</table>

Table 4.2: Time relation of chaincode upgrades related to size of file

Once again, the times do not change a lot according to the size of the file, where we can deduct that the size of the file is not a constraint on the time of execution of chaincode installations and upgrades. Downgrades were no considered, since the operation consists on an immediate action on the variables on MTChain.

In resume, each operation took around 45 to 49 seconds to execute a change in the chaincode. This is a good time, if we consider that this means a downtime of around 1 minute per upgrade on applications. Choosing less used moments of the application to do so is an effective measure to upgrade without affecting user experience.

The following section addressing the testing done to the notary application.

### 4.2.2 Notary application

The admin application ends up being of simple processing when compared with the Notary application, which is more complex, both on the server side and on the client side, being here the light part on the blockchain.

First, an overlook of the times of single operations, testing the latency of the system. After, tests on the throughput of application.

**Latency**

Latency on this context would make sense to be measured in terms of the start of interactions with the server, at the client side, since some of the operations are executed on such part of the system.
In table 4.3 we see the time necessary to complete the operations on the system, discarding user interaction: after the POST requests, the timer starts, stopping when the page finishes reloading.

<table>
<thead>
<tr>
<th>Interaction</th>
<th>Documents in History</th>
<th>Type</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load index</td>
<td>0</td>
<td>GET</td>
<td>151ms</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>GET</td>
<td>162ms</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>GET</td>
<td>262ms</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>GET</td>
<td>646ms</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>GET</td>
<td>1897ms</td>
</tr>
<tr>
<td>Start signature</td>
<td>-</td>
<td>POST</td>
<td>582ms</td>
</tr>
<tr>
<td>Sign Document</td>
<td>-</td>
<td>POST</td>
<td>285ms</td>
</tr>
<tr>
<td>Complete Execution</td>
<td>-</td>
<td>-</td>
<td>1371ms</td>
</tr>
</tbody>
</table>

Table 4.3: Time measures of page loading and operations execution. Each time results from a media of 10 measures.

These are the most relevant and expensive operations on the application, which imply the loading of the timeline (Load index), hashing and platform signing of the document (Start signature), signing of the hash on the client side and signing by the platform on the server side. All these operations involve queries or updates on the blockchain.

All the times are reasonable, but as the number of documents increase, so does the time to load their information into the timeline. This is due to the fact that the filling of this section is done by querying the blockchain, which is a light process, but becomes heavier as the number of queries goes. The suggestion to improve this is to query documents individually and upon click.

To the reads consideration, all the times measured in Start signature and Sign document were done discarding the time of loading of the timeline, for a more precise approach.

The complete execution time represents the time elapsed in:

1. Loading the index page;
2. Starting a new signature (which involves server signing and blockchain update);
3. Loading the index page;
4. Signing the document (which involves client signing and blockchain update);
5. Finishing the document (on the server side, by signing the hash one last time);
6. Loading the index page;

The complete execution was done with a single user only, and it considers the times to reach the parts of the page. Each index loading includes the query of the document, except for the one in point 1, where the blockchain would be empty and had nothing to load.

**Throughput**

To test the throughput of the system, several requests were made over a period of time, where the frequency of these increase over each minute. The requests are done by one user only, representing
3 signatures, one over the browser, two on the server side. These are done by loading the index page, starting a new signature and signing the document: a complete process.

Over the course of 10 minutes, 175 requests were done with an increasing load over each minute, as seen in table 4.4. As seen in the previous section, each request takes around 1300 milliseconds to complete, where the expectations to these tests is that the system works regularly until minute 8 - 9. By minute 9 - 10, the system should start delaying requests, but without losing any.

<table>
<thead>
<tr>
<th>Minute</th>
<th>Requests/min</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 1</td>
<td>6</td>
</tr>
<tr>
<td>1 - 2</td>
<td>6</td>
</tr>
<tr>
<td>2 - 3</td>
<td>7</td>
</tr>
<tr>
<td>3 - 4</td>
<td>9</td>
</tr>
<tr>
<td>4 - 5</td>
<td>10</td>
</tr>
<tr>
<td>5 - 6</td>
<td>12</td>
</tr>
<tr>
<td>6 - 7</td>
<td>15</td>
</tr>
<tr>
<td>7 - 8</td>
<td>20</td>
</tr>
<tr>
<td>8 - 9</td>
<td>30</td>
</tr>
<tr>
<td>9 - 10</td>
<td>60</td>
</tr>
</tbody>
</table>

Table 4.4: Number of requests per each minute over 10 minutes, testing throughput.

In table 4.5, we can find a media of the time that took each request to finish in every minute. This tests throughput through both normal functioning and peak load moments.

<table>
<thead>
<tr>
<th>Minute</th>
<th>media time to process</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 1</td>
<td>1149ms</td>
</tr>
<tr>
<td>1 - 2</td>
<td>1244ms</td>
</tr>
<tr>
<td>2 - 3</td>
<td>1345ms</td>
</tr>
<tr>
<td>3 - 4</td>
<td>1176ms</td>
</tr>
<tr>
<td>4 - 5</td>
<td>1183ms</td>
</tr>
<tr>
<td>5 - 6</td>
<td>1143ms</td>
</tr>
<tr>
<td>6 - 7</td>
<td>1193ms</td>
</tr>
<tr>
<td>7 - 8</td>
<td>1545ms</td>
</tr>
<tr>
<td>8 - 9</td>
<td>4742ms</td>
</tr>
<tr>
<td>9 - 10</td>
<td>10104ms</td>
</tr>
</tbody>
</table>

Table 4.5: Time a request takes to be processed in both normal and peak load execution moments.

As expected, the values increase in the last two minutes. However, this increase is acceptable in the matter that either 4 or 10 seconds do not represent such a big impact in the system, specially in this nature, where moments like this would rarely happen. However, this could be improved by introducing parallelism into the network, distributing the computation. A front-end server would distribute the workload to several back-end servers, increasing the number of requests necessary to reach a peak load moment, scaling the system horizontally.
4.3 Summary

In the end of the tests, the results are as expected. Security wise, the assurances made by the used technologies are overall satisfactory. Performance, on the other hand, is always conditioned by the operations performed interoperability, where times can not be improved. With this constraint in mind, we think that the times are acceptable for the sake of the operations performed, leading to an user experience that is satisfactory and light for normal executions but not so much for peak load executions. However, the system's usage is not expected to reach this point, and, if it does, there are solutions to solve it.

In a real environment, there would be several machines running the service and there would be also delays coming from infrastructure logistics, such as internet delays or servers overload. In this environment, we would expect better throughput times, configuring a front-end load-balancer that would distribute to several back-end servers. However, it would be expect a higher latency due to load-balancing mechanics and the extra connections.
Chapter 5

Conclusions

In this document, it was discussed a solution for a system that incorporates a blockchain network, identity management, privacy and confidentiality. The goals presented are first analyzed in Section 2 and then approached in the Section 3, which depicts the solution implementation, its mechanisms and functions. On the previous Section, 4, this solution is evaluated according to security and performance.

The work described in the document, MTChain, is a platform that interacts with a Hyperledger Fabric network, a component that constructs and manages the blockchain. The platform goal is to provide the necessary means for applications to run on top of it, while not having to worry about method definition regarding communication with the blockchain.

The final result was a prototype: an application developed with the goal of demonstrating the system capabilities and functionalities. This application was named Notary and it works as a notary service, performing as a witness to a document signing.

5.1 Achievements

The system developed achieves a first approach to identity management, privacy and confidentiality on a service that is today done by an entity certified, providing even improvements to it, such as the fact that no one but the signers see the information of the document.

When evaluating the system, the results showed that it takes an average of 1.5 seconds to sign a document. Comparing to a physical dislocation to a notary office, this is a huge saving point. In technical terms, 1.5 seconds is also a good time for the user to perform it, taking into account the operations performed between that fraction of time. When comparing against other systems of the same type, the work described in this document presents security assurances that other centralized, reliant on third party systems can not comply with, due to the use of a blockchain network. In terms of veracity of operations and verifiability, there are several advantages in using MTChain, where there are some fall-backs as well, such as the computational power needed and the expensive cost of write operation to the blockchain. Everything comes at a cost and, from the point of view of security and reliability of data consistency, it is worth.
5.2 Future Work

Future work for this project seems endless at a first sight, when realizing the potential of the blockchain and the platform itself. However, some priorities should be considered when thinking about what would be done to improve the technology and the business logic.

Firstly, Hyperledger’s network uses an identification system based on X509 certificates, which authenticate both the nodes of the network and the clients of the service. These certificates are self-signed at the moment, generated upon network creation by a tool provided by Hyperledger. In the future, these certificates could be replaced by Multicert’s certificates, a step further in the certification of the system. This would be useful for the three parties involved: Hyperledger, which would benefit from a different implementation example of certification amongst nodes; Multicert, which would see another fit to its CA; the system described in this work, which would improve the authentication mechanisms between the client and the trust on the network.

Another feature to be implemented would be the discarding of the database server, making the platform a distributed, peer-to-peer network. This would increase the complexity of the network largely, but create a distributed hash-table and a storage facility distributed, improving qualities such as user usage and availability. On top of this improvement comes the possibility for the storage of the documents themselves with cryptography properties, allowing the uploaders to later view their documents, but restricting it from the peers. This would require strong synchronization mechanisms and a network of peers to be working, so it should be developed in an advanced state of the platform, where it is justified.

Finally, a last improvement would be the development of cryptographic tools for browser that would be simple and intuitive to use by the programmer. This would require further investigation on current algorithms and the creation of new technology, requiring certification. An interesting approach would be to develop tools that would already persist against quantum computation, where there seem to be no solutions yet. This would represent not only an improvement to this particular system, but for technology in general.
Bibliography


