UNIVERSIDADE D
COIMBRA

Jason Pimenta Wrisez

BIOGRAPHIES OF THINGS USING BLOCKCHAIN

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Vasco Pereira and João Barata and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

January of 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Jason Pimenta Wrisez

Biographies of things using blockchain

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Vasco Pereira and João Barata and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

January 2023

Acknowledgements

First, I would like to thank Prof. Vasco Pereira and Prof. João Barata for their continuous advice and support of my work throughout both semesters. A special thanks to Prof. João Barata for the initial theme proposal and to Prof. Vasco Pereira for the late night revisions.

Many thanks to my jurors, Prof. Karima Velasquez and Prof. Bruno Cabral for showing interest in my work by reading, evaluating and suggesting improvements.

A personal thanks to my colleagues and friends João Paiva and Tiago Ribeiro, for their companionship during the whole graduation journey, support, and for being good partners.

I am grateful for my girlfriend Charline Monges, for bringing the emotional support, affection and purpose a man needs to accomplish his highest goals.

I would like to thank my family, especially mom, dad, grandma and my sister Line, for the usual support and their presence that also make this effort worthwhile.

Lastly, I would like to show appreciation to my roommates, for bringing me a fun and pleasant environment to live and work in.

Abstract

This thesis addresses the concept of product biographies hosted in blockchain-based systems. A product biography is a log that defines the characteristics of a product from the beginning to the end of its life, along with potential events and transformations originating new lifecycles of that product. A Design Science Research (DSR) methodology was followed during the course of this project, starting with the elaboration of a literature review about product tracing and different implementations of blockchain-based networks. An architecture for a generic system capable to host product biographies was represented, picking a solution based on literature review findings around enterprise blockchain solutions. This architecture gives attention to quality attribute scenarios and requirements regarding stakeholders' roles, product data, and the blockchain network and infrastructure. After this design phase, the objective was implementing a prototype of this platform, by setting up a network, a smart contract with different functions suited for operating with product biography information, and an external client application for end users, composed of a web application backend and frontend. With this prototype, simulations and testing were done to validate the previously defined scenarios of the architecture, in order to support different quality attributes and demonstrate the usefulness of a blockchain solution for this use case. Finally, the thesis wraps up with a conclusion, guidelines for potential future work about this theme.

Keywords

Product biography, Distributed Ledger, Enterprise Blockchain, Product Tracing, Smart Contract

Resumo

Esta tese aborda o conceito de biografias de produtos hospedadas em sistemas baseados em blockchain. Uma biografia de produto é um registo que define as características de um produto desde o início até ao fim de sua vida, juntamente com possíveis eventos e transformações que originam novos ciclos de vida desse produto. Uma metodologia de *Design Science Research (DSR)* foi seguida no decorrer deste projeto, começando com a elaboração de uma revisão de literatura sobre rastreamento de produto e diferentes implementações de redes baseadas em blockchain. Foi representada uma arquitetura de um sistema genérico capaz de hospedar biografias de produtos, com base em descobertas da revisão de literatura à volta de soluções de blockchain empresarial. Esta aquitetura dá atenção a cenários de atributos de qualidade, e a requisitos sobre as funções de partes interessadas, dados de produto e rede e infraestrutura de blockchain. Após esta fase de design, o objetivo foi implementar um protótipo desta plataforma, através da criação de uma rede, um *smart contract* com diferentes funções adequadas para operar com informação de biografias de produto, e uma aplicação externa para utilizadores finais, composto por um frontend e backend de uma aplicação web. Com este protótipo foram feitas simulações e testes para validar os cenários previamente definidos da arquitetura, de forma a suportar atributos de qualidade diferentes e demonstrar a utilidade de uma solução blockchain. Por fim, a tese é concluída com orientações para possíveis trabalhos futuros neste tema.

Palavras-Chave

Biografia de Produto, Blockchain Empresarial, Rastreamento de Produto, Registo Distribuido, *Smart Contract*

Contents

Acronyms

ATAM Architecture Tradeoff Analysis Method.

AWS Amazon Web Services.

BOL Beginning Of Life.

CA Certificate Authority.

CFT Crash Fault Tolerance.

CLI Command-Line Interface.

CPU Central Processing Unit.

DAO Decentralized Autonomous Organization.

Dapp Deecentralized Application.

DLT Distributed Ledger Technology.

DSR Design Science Research.

EAN European Article Number.

ECC Elliptic-Curve Cryptography.

EIP Ethereum Improvement Proposal.

EOL End Of Life.

ERC Ethereum Request for Comments.

ETH Ether.

EVM Ethereum Virtual Machine.

GLN Global Location Number.

GPS Global Positioning System.

GRAI Global Returnable Asset Identifier.

GUI Graphical User Interface.

IoT Internet of Things.

JS JavaScript.

MOL Middle Of Life.

MSP Membership Service Provider.

NFT Non-Fungible Token.

OU Organizational Unit.

P2P Peer-to-Peer.

PBFT Practical Byzantine Fault Tolerance.

PLM Product Lifecycle Management.

PoET Proof of Elapsed Time.

PoS Proof of Stake.

PoW Proof of Work.

QA Quality Assurance.

QAS Quality Attribute Scenario.

RFID Radio-Frequency Identification.

RIPEMD RIPE Message Digest.

RPC Remote Procedure Call.

RSA Rivest–Shamir–Adleman.

SBC Single-Board Computer.

SGX Software Guard Extensions.

SHA Secure Hash Algorithm.

TLS Transport Layer Security.

UTXO Unspent Transaction Output.

VPS Virtual Private Server.

WWAN Wireless Wide Area Network.

List of Figures

List of Tables

Chapter 1

Introduction

This document is the final thesis "Biographies of things using Blockchain". This first chapter is an introduction to the theme of this work, clarifying what the context, motivation, objectives, and expected outcomes are, followed by a concise structure for the rest of the document.

1.1 Context

In our modern economy, the design, production, usage, and dismissal of all kinds of products are being done by separate organizations and people from all around the world. Keeping track of materials and processes used for each component of a complex product is not an easy task, requiring the supervision of different parts of an organization's value chain. Some cases of design, supply, and service chain disruption cannot be prevented, mostly due to malfunction, scarcity, or accidental reasons, but a lot can still be done regarding product waste, loss, and storage management, as shown in [Rezaei and Liu, 2017].

Product Lifecycle Management (PLM), a concept well defined in [Terzi et al., 2010], can be seen as a business model point of view and as a strategy for managing and linking the three phases of the lifecycle of a product:

- Design and manufactoring periods stand for the Beginning Of Life (BOL).
- The following distribution, sale and use of these products corresponds to the Middle Of Life (MOL).
- Finally, the End Of Life (EOL) stage of a product happens when it is disposed of or collected for recycling and reuse of materials or components.

From a technology standpoint, PLM also represents a set of integrated tools that create a flow of information about the product in every phase of its lifecycle. The goal of these tools is to have the best data possible at the right time and with the correct context to mitigate preventable problems. Organizations are taking the

initiative to prevent future losses by developing a better tracing system for their products through the use of new technologies.

Since the appearance of the internet, new opportunities for improved lifecycle management have emerged for performance enhancement and cost effectiveness, such as the availability of shipment data, instantaneous communication with vendors and customers, or simply online purchasing, customer support, and product reviews [Lancioni et al., 2000], and things have not stopped. The adoption of blockchain networks takes PLM to a new level by allowing more tracing capabilities and data transparency between all stakeholders of a product. This is possible, for example, with the tokenization of assets, a concept used for the digital representation of products in a blockchain.

Due to this digital transformation and the growing demand for environmentally friendly practices, it is becoming increasingly appealing for users and organizations to learn more about products by keeping a complete log of their lives, from conception to disposal. Information such as the origin of a product, the duration of its storage before sale, and the number of owners it previously had are all important pieces of knowledge, and being able to store and verify this data, while respecting the privacy of the end consumer, is a challenging task. From now on, we will refer to this product log as a product biography, a term already used in previous papers, such as [Barata et al., 2020] and [Spring and Araujo, 2017].

1.2 Motivation

It is stated in [Ameri and Dutta, 2005] that around 60% of most businesses' operational time gets wasted due to disseminated, incorrect, and repeated data used during design and production. The development of a generic product biography aims to solve these issues by gathering the correct amount of data from every stage of a product's life and linking it together.

Note that representing a product from the beginning to the end of life is only a linear interpretation and simplistic point of view. In real life, a product can go through different stages of EOL and go through a "servitization" process to repair, for instance, a product by swapping a few components. These different cycles of a product are also named, from a business point of view, as a circular economy composed of maintenance, reuse, redistribution, refurbishing, remanufacture, and recycling phases. These steps are thoroughly described in [Spring and Araujo, 2017]. A well defined product biography has to contain all these transformations, where each component can be linked to each other to form a complex product.

Other interesting concepts to include in these biographies are user experience reviews and manual insertion of data about specific models of products. The distinction between autobiography, where the product itself is the source of data and third-party biography, where external user experience memorization occurs, is mentioned in [Barata et al., 2020]. These can be very useful for fault detection, tracking the desirability of a product, the rate of failure, and legal regulations.

Another interesting example of product biography applications is the new sustainability initiatives emerging from the European Union, stated in different documents and proposals, such as [Commission, 2022], that aim to create a "Digital Product Passport for textiles" in 2024, with mandatory requirements about environmental sustainability. An opportunity emerged in our contacts with a major textile fibre producer to understand the possible applications of a biography to their business. This represents an opportunity to create product biographies around textile products, a big part of these being clothing pieces, linking all data from raw material harvesting to disposal of worn-out remains.

The quality attributes of data in a blockchain, described in more detail later in the document, serve as the driving forces for using this new technology to host these biographies and, in result, give more reliable data to organizations and more confidence to consumers.

1.3 Objectives and expected results

Considering the problem stated above, a list of objectives is defined for the duration of this thesis. The order of these objectives also align with the structure of the rest of the document:

1. Establish a work methodology suited for this theme, with a progressive risk management and a showcase of tasks completed split between the two semesters;
2. Perform research in current product tracing implementations and do a literature review of blockchain concepts and popular cryptocurrency and enterprise solutions;
3. Design a generic architecture with quality attributes scenarios and requirements, suited for a platform hosting product biographies, and focusing on blockchain technology as a backend foundation;
4. Develop a prototype for product biographies, describing every step taken and applying the defined architecture;
5. Validate the architecture scenarios with tests on the implemented prototype, as well as showcasing testing techniques on the developed artifacts;
6. Conclude the document, stating the accomplishments of the thesis and provide guidelines for future work, with possible improvements.

Finishing this first chapter, an introduction to the theme of the thesis was concluded, along with the motivation of the work being done and a list of structured goals. As stated in the objectives, the methodology of the work and the risk management are reviewed in the next chapter.

Chapter 2

Methodology and work plan

This chapter describes the methodology used for this thesis and the work planned for both semesters. A brief and adaptative risk analysis was also performed to plan courses of action in case of failure in completing important tasks, in order to avoid potential problems. Gantt diagrams were used for each semester to show the sequence of the finished task, with a description for more context.

2.1 Methodology

A Design Science Research (DSR) approach was used for this thesis, consisting of several iterative steps for the construction of a solution [Peffers et al., 2007], while following DSR guidelines from [Hevner et al., 2004]. To better understand the process, Figure 2.1 shows the expected flow of the project, indicating each phase of the process, the outcomes, and what iterations may happen. Note that the last iteration, from the conclusion back to a new problem identification, stands for the possibility of picking up this thesis work and developing it even further.

DSR has been previously used by similar research, such as for development of blockchains for off-site production management [Wu et al., 2022] and supply chain management frameworks [Liu et al., 2022]. This methodology is helpful to get reliable knowledge into a new domain, enumerate a problem to solve, and draw a proposal for a solution, which corresponds to the first three steps and what was done in the first semester; and develop, evaluate the result and conclude with guidelines for future work, which is what was done in the second semester.

2.2 Work distribution

The thesis work started in the first semester with a literature review, presented in Chapter 3, on product tracing, mainly about how different industries track and version their products, how failures can be detected and what solutions ex-

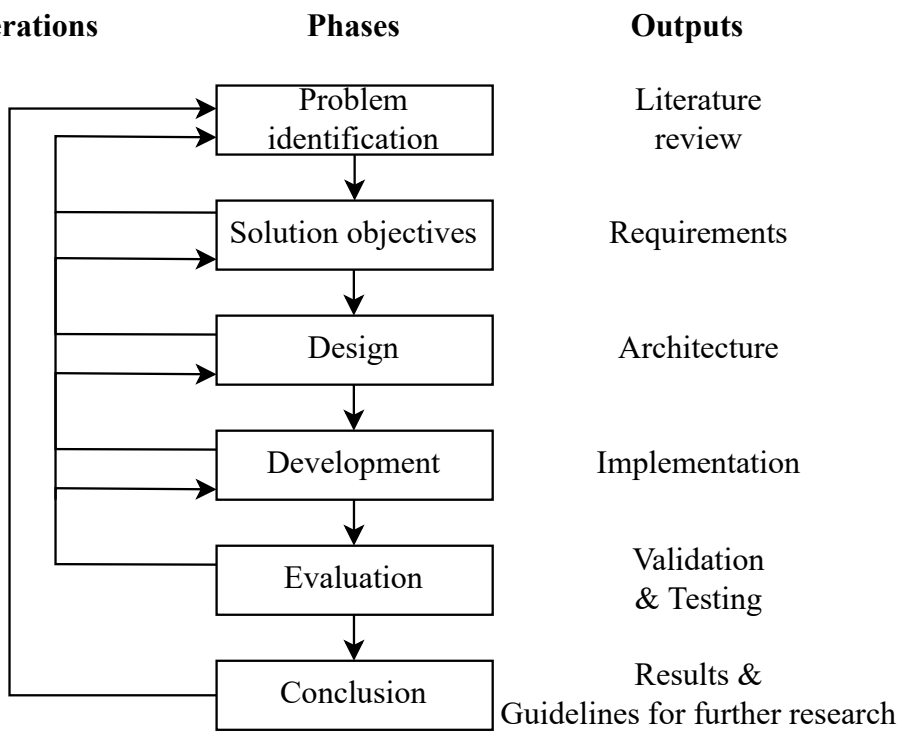


Figure 2.1: DSR flow for this thesis.

ist. Having chosen blockchain as our core technology for developing a future solution, extensive research was done about this concept to understand its real advantages. Current popular blockchain projects, around cryptocurrencies and enterprise use cases were also reviewed, in order to analyse their implementation decisions and perceive the reasoning behind them.

Based on the literature findings and contacts with industrial experts interested in our work, high-level requirements of a generic platform and the first version of the architecture were defined in Chapter 4.

A summary of the work of the first semester is shown in Figure 2.2.

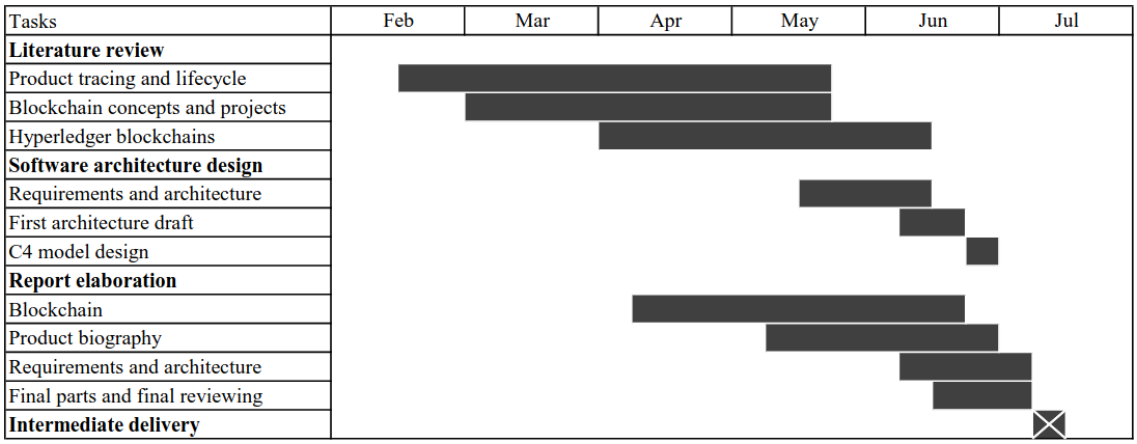


Figure 2.2: Gantt work chart for the first semester.

In the second semester, the first step was starting the development of a platform hosting product biographies, based around the projects reviewed in the previous semester. At the same time, the literature review was improved, adding corrections and more information to the document, as well as the definition of quality attribute scenarios, used to validate the architecture. While achieving this prototype, tests and evaluation of the implementation were completed to assure the correct function of the project. Finally, the final report was elaborated, documenting the whole implementation process. The work summary of the second semester is displayed in the Gantt chart 2.3.

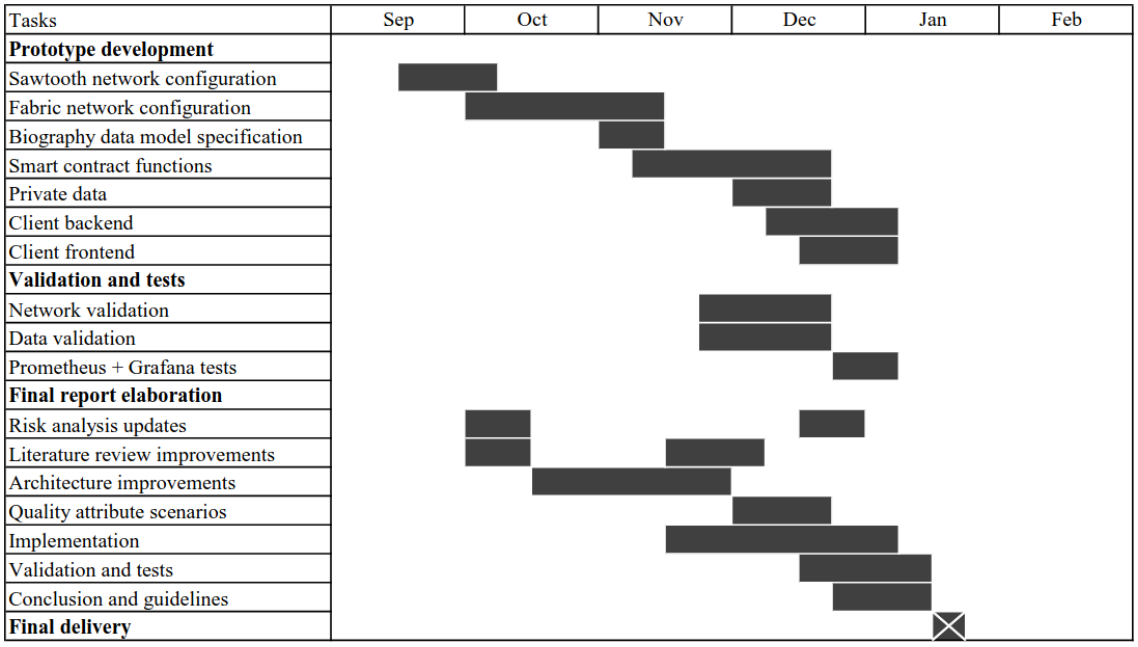


Figure 2.3: Gantt work chart for the second semester.

In the following section, the risk analysis taken during the thesis is described, showing the preventive steps taken for a better execution of the work.

2.3 Risk analysis

Due to the emerging nature of blockchain and its applications, some inherent risks exist in the elaboration of this thesis. Risk analysis was done throughout the duration of the project, in order to have a solid grasp of current conditions and to better react to potential consequences of the decisions taken. For each risk, probabilities, impacts and mitigation plans were defined.

A risk probability can be **low**, when the risk is not likely to happen during thesis, **medium**, when it is somewhat likely to occur, and **high**, which is very likely to happen during the project. As for the impact, the same three levels of likelihood were used. A **low** impact represents no significant changes or delays to the course of the project, a **medium** impact forces some changes to occur during the development, and finally, a **high** impact drives considerable changes and

potential delays, shifting the objectives of the thesis.

The following Tables 2.1, 2.2, 2.3 and 2.4 identify the risks taken into account in the beginning of the work:

Table 2.1: Risk 1

Risk	The research involves new technologies around blockchain; having a risk that these technologies may get rapidly outdated or inaccurate with new version releases and changes in a specific technology's architecture.
Probability	High.
Impact	Medium.
Mitigation plan	Engage through further research, for instance by analysing up-to-date documentation.

Table 2.2: Risk 2

Risk	The implementation process requiring using new and more recent tools, such as using new programming languages and containerize nodes to run a blockchain network; has a risk of needing more adaptation time than expected, leading to delays in the elaboration of this thesis.
Probability	Medium.
Impact	Low.
Mitigation plan	Doing early experiments with the technologies to be used.

Table 2.3: Risk 3

Risk	The technologies chosen for the architecture and implementation were only initially researched theoretically, having a risk of not working as expected or not satisfy the needs of this thesis in an applied implementation.
Probability	Medium.
Impact	High.
Mitigation plan	Opt for a fallback option that is also compatible with the requirements defined in Section 4.2.

Table 2.4: Risk 4

Risk	The applicability of the architecture details in a textile industry use case, where the possibility of all planned development is uncertain.
Probability	Low.
Impact	Low.
Mitigation plan	Follow through with an implementation for generic products.

As the thesis advanced in the second semester, a few risks materialized the following ways:

- The risk in 2.3 required the project to have a completely new architecture model and a different implementation part, switching from a Hyperledger Sawtooth backend to a Hyperledger Fabric one, which now included smart contract development. These technologies and their usefulness are thoroughly evaluated in the literature review;
- The risk in 2.4 happened, redefining a few points of the document and only using textile products as examples during the implementation process.

Risk 3 delayed the implementation and architecture validation process, it made risk 2, of Table 2.2, less likely to happen due to the larger choice of programming languages of the fallback option. Following this iteration, new risks, in Table 2.5 and 2.6, were taken into consideration, throughout the new phases of development:

Table 2.5: Risk 5

Risk	As smart contract functions are ran in a blockchain network, they are harder to test and debug in a development environment due to potential network related issues, risking a longer implementation timeframe.
Probability	High.
Impact	Medium.
Mitigation plan	Finding appropriate solutions for testing smart contracts, taking into account the type of network and programming language used.

Table 2.6: Risk 6

Risk	The development of an external client application interacting with a blockchain network is more complex when compared to other traditional components of a development stack, such as databases or microservices; risking an unfamiliar debugging process to this part of the implementation.
Probability	Low.
Impact	Medium.
Mitigation plan	Researching into more implementation details of the blockchain network.

And during the rest of the work, the following risks were addressed, helping in a achieving a better result:

- The risk in 2.5 was likely to happen, but it was correctly prevented by researching debugging and testing solutions, such as the ones used in Chapter 6;
- The risk in 2.6 was prevented, doing additional research around Hyperledger Fabric in the beginning of the second semester.

The following chapter will present a literature review of this project in order to understand the concepts required for the solution.

Chapter 3

Literature review

In order to understand the issue that needs to be solved, this chapter's literature review begins with a synopsis of the most recent developments in the field of product lifecycle management and tracing, with special attention to what infrastructure and technology is used. Subsequently, the distinctive features of popular Distributed Ledger Technologies (DLTs) are mentioned, followed by a more detailed review of blockchain concepts to emphasize what properties are the most relevant for our problem and what challenges are associated with the technology.

After covering the main points relevant to a blockchain data and network structures, an introduction is shown for two of the most popular cryptocurrency projects, Bitcoin and Ethereum,. These cryptocurrencies are based on blockchain technology and the intention of this review is to understand the rationale between the decisions taken, what their work in progress is, their impact on the financial system as a cryptocurrency and why it happened. Also, difficulties during their current open development are described in more detail, with the mention of relevant solutions aiming to solve the known problems of Bitcoin and Ethereum.

Following the cryptocurrencies review, an analysis of enterprise blockchains is provided, maintained by the Hyperledger foundation. These projects follow a philosophy much more suited for industrial applications and handling those specific types of use cases. A more detailed review is suited for the Fabric and Sawtooth blockchains, due to their applicability to supply chain solutions. The main purpose of this literature review is to highlight and compare different blockchain implementations in order to evaluate the applicability and possibilities in an architecture and for solution of a product biography.

3.1 Product tracing

As already mentioned in Section 1.1, different organizations have started to find solutions to better track their products in every stage of their lives. In this first section, we will examine what the latest and avaliable solutions are and what technologies are appliable in a platform hosting biographies of things.

To start this literature review, we reviewed three different implementations related to the theme of this thesis. Following this, a brief summary about the GS1 is made, an organization maintaining global standards about products in a wide range of industries. Since the goal of this research is to discover a suitable technology for our solution, and in part, define a generic architecture of a blockchain network, the results of each implementation do not have to exactly align with our goals.

An interesting implementation of a blockchain applied to the textile industry comes from [Agrawal et al., 2021]. The paper structures a Hyperledger Fabric blockchain architecture, covered in more detail and with the latest version in consideration, in Section 3.5, and a representation of a textile and clothing supply chain, which corresponds to six phases. If we were to use this supply chain to execute our product biography, there would be even more steps before and after to provide a thorough record of all the product's life.

Although this paper doesn't deal with Internet of Things (IoT) nor represents a generic way to insert different parameters for textile products, it provides insight as to how to track states of organic cotton transactions, generated from a smart contract, and account values in a blockchain, as well as performance simulations for a specific consesus algorithm, explained in the sections below.

[Barata et al., 2020] introduces the concept of "biography of things", corresponding to product biography, mentioned in Section 1.1, and a complete gathering and storage system of sociomaterial data about products, behaving as an extension of regular Product Lifecycle Management (PLM) and using blockchain as well as offchain data. The approach taken by this paper is to follow the production of paper pulp by using digital twins, IoT devices to track this data, located in the same area. Some limitations to this approach are revealed and addressed:

- It is impossible to have only one digital twin in a product with different manufacturing stages and components. To solve this, a digital twin is used per phase, tracking each part individually.
- Having a digital twin for as little as each tree, in the case of paper pulp production, is unfeasable. In this case, materials should be tracked in batches, for instance, with one digital twin per production lot.

In the case of [Barata et al., 2020], the composition of a product made from paper pulp could be represented in three different stages, by (1) raw materials, (2) factory, where paper pulp is produced, and (3) the final product, ready for sale or use. To gather more details about an implementation using IoT devices, [Salah et al., 2020] depicts a use case about a cold supply chain of pharmaceutical products, where real-time attention to temperature and humidity levels are crucial.

As for the IoT devices, [Barata et al., 2020] and [Salah et al., 2020] implementations use similar technology, such as different sensors to track temperature, humidity, vibration, light and other movements; Global Positioning System (GPS) and Wireless Wide Area Network (WWAN) modules; smoke detectors and Radio-Frequency Identification (RFID), all linked to Single-Board Computers (SBCs) or

microcontrollers. These devices enable a real-time logging of data during manufacturing processes, transportation and usage, which is invaluable for an organization, making them able to evaluate losses, find the causes of an anomaly more easily and give it more time to react and mitigate the economic or logistic impact. It is also vital to prevent and detect errors from these devices, especially during automatic data writing. A way of prevention is to use more than one of each device in a place at a time.

To finish, GS1, as mentioned in the beginning of this section, is a global organization that maintains a number of standards for products. [GS1, 2022a] shows a detailed document about every standard and their specification, that These standards help identifying products in a global commercial network, such as:

- European Article Number (EAN) is a 13-digit number, used as a barcode, to identify products in the industry and to be scanned during transactions and inventory checks;
- Examples of unique identifiers are the Global Returnable Asset Identifier (GRAI) and the Global Location Number (GLN), used to track reusable products, for example containers used in transportation, and locations, respectively. These numbers are all structured in a similar fashion, using the first one, two or three digits as country codes and the rest as manufacturer and product or the specific location.

Figure 3.1: EAN-13 barcode from the GS1 standard, taken from [GS1, 2022a].

These identifiers can be employed in a potential solution suggested in this thesis since they can have a significant impact on the process of tracking products along a supply chain.

These solutions are all distinct from one another since the objectives are different, but they all collect vital information that we should be aware of while building our own solution. By implementing a solution based on blockchain, one still has to deal properly with privacy-sensitive data by using encryption and possibly with permissioning systems. The industry could benefit from a honest system with biographies of things, using DLT technologies which are addressed next.

3.2 Distributed ledgers and blockchain key concepts

After a brief overview about product tracing implementations, key concepts and notions are identified in this section in order to better understand the subject and the potential of blockchain based solutions.

In the following sections, *nodes* and *peers* are mentioned interchangeably, as synonymous for entities participating in the validation of a blockchain or distributed network as well as *data* and *transactions* for information stored and/or verified in the blockchain. These concepts apply to the majority of existing blockchain implementations.

The concept of DLTs was established as a consensus of digital data that is shared and synchronized among other peers which can be personal computers, servers or any other computing device with enough storage and network capabilities. Each node can store a local copy of the ledger and use one of several consensus protocols to agree on which data is stored. The data is timestamped and usually appended in chronological order in the ledger, making it impossible to edit or delete the content after it is inserted.

This means, by default, that one of the quality attributes a DLT has to offer is its immutability and the concept of a product biography may benefit from that. Other quality attributes can be assured, mentioned in [Kannengießer et al., 2020] such as avaliability of data and confidentiality, integrity and security of the ledger depending on design decisions and trade-offs.

Comparisons between blockchain, Tangle and hashgraphs technology are continuously made, for example in [Schueffel, 2017] done in 2017, but for the sake of simplicity and to opt for a more popular and patent-free solution, only blockchains are covered in this thesis as a backend solution for a product biography platform.

A blockchain, shown in 3.2, is a data structure based on a group of blocks of data usually used in distributed and potentially decentralized and public networks. Distributed networks are networks that divide tasks between different devices in order to fulfill a certain goal, whereas decentralized networks distribute power between different entities, with the goal of having no central authority.

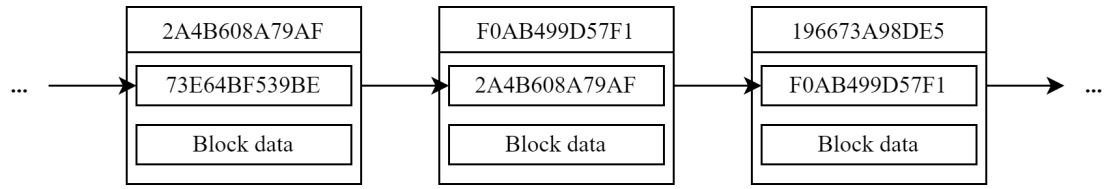


Figure 3.2: Generic representation of a blockchain, adapted from [Nakamoto, 2008].

These blocks are linked together by a mathematical one-way and deterministic function called a cryptographic hash, explained below. Each block of data contains the hash value of the block preceding it, except for the first block, also known as the Genesis Block or Block 0.

3.2.1 Hashing

Hashing is used to verify integrity of data. Hash functions can take any data of any length, called a message, and convert it into a fixed size hash value or digest, which is impossible to revert in a practical fashion if the algorithm used

is safe, like the Secure Hash Algorithm (SHA) 2 family of functions, KECCAK used in Ethereum, standardized as the SHA-3 family, or the now deprecated MD5 function. The size of this digest depends on the algorithm used.

These functions also have a desired avalanche effect, meaning for the slightest difference in the message, the result will be completely different. This is shown in Figure 3.3 and we can also notice that the messages are indistinguishable from the hash values.

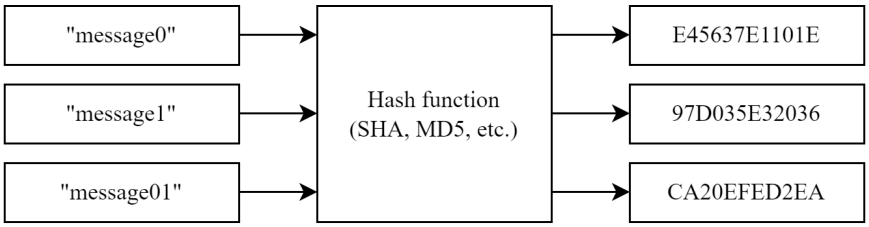


Figure 3.3: Hashing function process and its avalanche effect at work, adapted from [Nakamoto, 2008].

Hashing cryptographic procedures are widely used and essential parts of very important projects like OpenSSL and its security is continuously tested [Gilbert and Handschuh, 2003].

In a blockchain, each block has its own hash value that depends on the previous block's hash. Any change in a previous block will also change the resulting value of the hash function, which in turn will change the hash value of that block and every block succeeding it. This makes the whole structure immutable, which is essential for a blockchain and its applications.

3.2.2 Asymmetric cryptography

Another key concept in a blockchain is the use of private/public key pair encryption, or assymetric encryption. Transactions occuring in a blockchain are signed by someone using its own private key and verified by others using its public key to prove its origin.

The private or secret key is usually a string, a seed phrase or a number generated or picked from a big range of numbers. The public key is derived from the private key and used to verify these signatures, giving proof of who submitted the transaction. To generate the public key, an asymmetric cryptography function is used such as the Rivest–Shamir–Adleman (RSA) algorithm. This function should also be one-way, deterministic and takes longer to run than other types of encryption, so it should not be used for actual encryption of data.

Cryptocurrencies use *wallets*, that manage private/public key pairs and allows users to own and send coins to other users through their public keys, called *addresses*. Every public key may be explored in the blockchain to see the chain of transactions of circulating coins, for example. Implementations of this are described in the following Sections 3.3 and 3.4.

For our solution, it is essential to identify who is writing data into the blockchain, despite only using key pairs for identification being a pseudonymous process.

3.2.3 Data storage and Merkle Trees

Scalable data storage is tricky to do in a blockchain. In theory, all kinds of data may be stored into a blockchain but in real use cases, it is not feasible to distribute any files or heavier data into a public and decentralized ledger. This is due to storage limitations of every node and network limitations of distributing that data to every peer avaliable, even more so if the blockchain is public.

The block size in a blockchain can also vary, some projects use a fixed size while others use a variable size. A bigger block size requires more computational power from the nodes, which can limit the number of nodes and in turn centralizes more the project, but increases the transaction throughput, as seen in [Göbel and Krzesinski, 2017].

A common solution for data storage is to not store all the data but instead use the hash values of it, grouped in a Merkle Tree, represented in Figure 3.4.

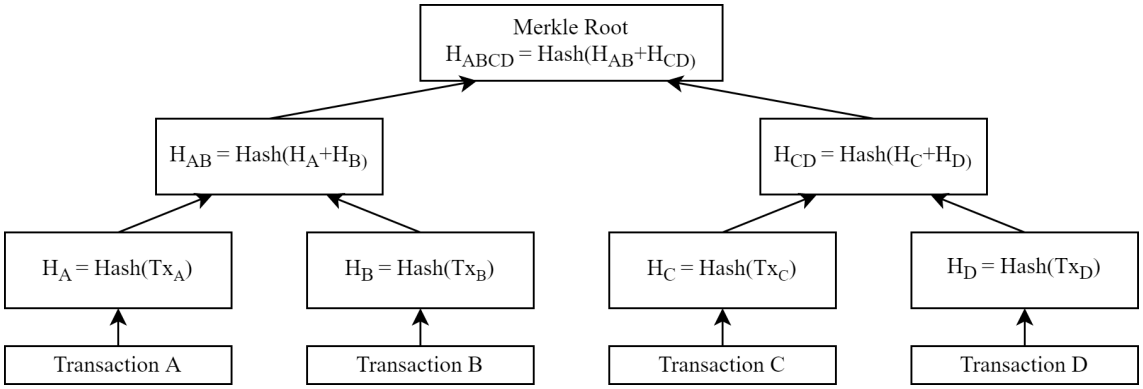


Figure 3.4: Merkle Tree representation, aggregating four transactions, adapted from [Nakamoto, 2008].

Transactions are grouped in a tree by hashing them together, giving the verifiability needed for the transaction to be immutable, or in other words, tamper-proof.

Because a blockchain is not the best way to store raw data, other ways of storage are used and verified with the hash signatures, for instance by using an external database or other. This approach can centralize the data source to a unique entity and therefore making the blockchain lose its purpose as a decentralized tool.

3.2.4 Network permissioning

Blockchain networks, just like any other distributed environment, can have its own sets of rules for who is allowed to participate in the validation or access of the blockchain.

We refer blockchain networks as permissionless when they don't require the permission of any entity to participate and validate new blocks, as opposed to permissioned blockchains which require permission and only a set of entities are allowed in. As for submitting transactions, a public blockchain allows anyone to transact in, whereas private blockchains only allow certain entities to interact. Semi-private solutions also exist, which can have roles implemented for different nodes with different types of access, for instance.

Different attributes of permissionless and permissioned blockchains are compared in Table 3.1. These attributes are related to transactions and it is important to mention that these are just broad generalizations and no attribute has a fixed result for every scenario. For instance, a permissionless blockchain can be private, although it is an unusual tradeoff to take.

Table 3.1: Comparison between permissioned and permissionless blockchain networks.

	Permissionless	**Permissioned**
Transaction speed	Slower	Faster
Decentralization	Total	Parcial or none
Data security	Stronger	Weaker
Cost	Expensive	Cheaper

Usually, in a public and decentralized blockchain, anyone can run a node with its own copy of the blockchain without the a central authority coordinating the process, as long as the node follows the expected rules. Nodes can find each other in several ways, for instance by maintaining a list of nodes they already know, consulting servers that respond with a list of avaliable nodes or just run an adequate gossip protocol, disseminating addresses and data.

This raises a big challenge: how can people agree on which blockchain is the real one, if anyone can validate blocks? To solve this, different consensus protocols were put in place for each use case as systems of agreement for all the participant nodes in a blockchain. Since everyone can submit transactions or other data into a blockchain, other nodes need to verify that data.

3.2.5 Consensus protocols

When working with distributed systems where data replication exists, it is important to agree on what data is being maintained and eventually protect the network from attackers that wish to exploit it [Wahab and Mehmood, 2018]. In a blockchain, consensus protocols are used to decide who has the capacity of writing the next block.

In permissionless networks, to ensure the authenticity of every transaction and data in the blockchain, consensus mechanisms are used in every participating node to appoint a winner of a lottery to write the next block, repeating the process for each one. Whereas in permissioned networks, where the participants are already managed separately, consensus protocols are usually used to prevent

byzantine [Pease et al., 1980] or crash failures instead. In summary, two kinds of consensus protocols exist:

- Probabilistic algorithms, when the writer of the block is chosen by a lottery between the nodes;
- Deterministic algorithms, when a predefined node is in charge of submitting blocks for an arbitrary period of time.

The most well-known and used consensus protocols used in blockchain presently are two probabilistic algorithms: *Proof of Work (PoW)* and *Proof of Stake (PoS)*, and two deterministic ones: *Practical Byzantine Fault Tolerance (PBFT)* and *RAFT*.

To understand the intricacies of different consensus mechanisms, a full context of a blockchain structure and network is needed. Every consensus protocol has its own trade-offs and that is the reason why there is not a singular solution for every use case, whether it be a permissioned or permissionless blockchain and what kind of data it holds.

In the following sections covering blockchain implementations, we will cover in more detail how these concepts are applied and how consensus algorithms incentivize usage, in the case of cryptocurrencies and how well they can scale for our solution, as well as the other concepts exposed here that are crucial to understand the need of a blockchain and why it can be useful for our solution.

3.3 Bitcoin

The literature review will now focus more on examining projects with blockchain implementations relevant for our problem, starting with Bitcoin. Having this approach after reviewing important aspects from the last section will help us understand better the decision making of each implementation.

In 2008, Satoshi Nakamoto started designing and developing the Bitcoin protocol [Nakamoto, 2008], a solution for a decentralized and Peer-to-Peer (P2P) digital money system that discards the need of a mediator or a central entity to confirm transactions. This is achieved with a public and permissionless network, where anyone with a powerful enough computing device can be a node and help support the network, validating the blockchain and confirming the incoming transactions. This process of supporting the network is called mining, in the case of Bitcoin and other blockchains that use a PoW consensus mechanism.

Its development was based on previous cryptocurrency work, such as [Szabo, 2022], and with the use of several design decisions mentioned below and extensive and open development, Bitcoin managed to be widely adopted as an alternative to fiat money and speculative asset, with a market capitalization reaching 1 Trillion USD in 2021, recorded in [CoinMarketCap, 2022] while its creator remains anonymous.

In the following subsection, we will introduce some key decisions taken by Bitcoin that can, in a way, justify its adoption as well as its main challenges as a cryptocurrency currently.

3.3.1 Key design choices in Bitcoin deployment

As mentioned in the previous chapter, Bitcoin went through an extensive development period where critical decisions were made that wouldn't be easy to revert now, with the cryptocurrency actively in use and widely adopted. Any future change to the blockchain structure would require an update of the Bitcoin Core software, which is the tool needed to run a node.

In case of disagreement between the nodes in which node version to use or which chain of blocks to maintain, a fork of the blockchain may occur at a certain block number where the original blockchain is maintained and a new one starts to be used and mined [Webb, 2018]. Forks may also happen momentarily, in case two nodes verify a block nearly at the same time, coexisting for a short moment. Bitcoin had several forks from the original protocol, the most widely used today being Bitcoin Cash.

It is important to review these design choices to achieve and justify an architecture of a blockchain suited to store our product biographies.

Transaction storage

The model Bitcoin uses to store balances in the blockchain is called Unspent Transaction Output (UTXO), where the amount avaliable to a wallet equals to the amount received and not yet spent. [Nakamoto, 2008] defines a coin as a chain of digital signatures and allows transactions to have several inputs and outputs, therefore splitting and merging different amounts for a new UTXO, as a more efficient way to move coins. UTXO-based systems are fundamentally different from other account-based systems, where a state is kept for each wallet instead and enables a more modular approach with blockchain data [Chan, 2021].

This also allows Bitcoin to "prune" the Merkle Tree, previously described in 3.2.3, by removing transactions with the totality of a coin spent, saving disk space.

Double hashing

Bitcoin kept its hashing implementation relatively simple, using just the SHA-256 algorithm and RIPE Message Digest (RIPEMD)-160 in its code [Chan et al., 2020], for several things:

- Maintaining data integrity of every transaction;
- Hashing a Merkle Tree;

- Linking transaction inputs and outputs;
- Creating public addresses with better privacy;
- Mining new blocks, with block header, PoW and previous block hashes.

In every case, every hashing is computed twice either by using the SHA-256 function twice, where $hash = SHA256(SHA256(message))$ or, using the RIPEMD function to output a smaller hash: $hash = RIPEMD160(SHA256(message))$, used for public addresses mainly. Both double hashes are done as a preventive security measure.

Ownership of coins and digital signatures

The Bitcoin protocol enables anyone owning a private/public key pair to make transactions by signing with the owner's private key a hash of the previous transaction plus the new owner's public key [Nakamoto, 2008]. This makes it impossible to spend someone's coins unless the private key is compromised.

The owner of a private key can remain a secret since no identifier is needed to own one, making Bitcoin a pseudonymous network. It is not truly anonymous because public keys can still be queried to analyse where the coins were transferred to. So the coins are effectively based on a chain of digital signatures and in Bitcoin, the public key is derived from the private using Elliptic-Curve Cryptography (ECC) functions.

Elliptic-Curve Cryptography

ECC is a more recent approach from traditional asymmetric encryption systems, such as the RSA, using a different mathematical approach. In short, the public key generation uses arithmetic operations around an elliptic curve, as opposed to around a modulo number. Nowadays, ECC is considered to be better compared to RSA and other conventional systems since a smaller key size in ECC can provide the same level of security and is faster calculating the public key [Rebahi et al., 2008] due to the lower key size but slower verifying it, although these differences might be neglegible in our context.

Elliptic-curve public-key cryptography is still not as used by common systems nowadays, mainly due to uncertainty about patent-related issues and its security due to the algorithm not being as mature and understood as RSA for example.

An elliptic curve's mathematical expression is $y^2 = x^3 + ax + b$ with a and b as parameters that should be carefully chosen. The curve chosen for Bitcoin's public-key generation is called secp256k1, which is a name derived from the curve's parameters and these parameters are also the properties that limit the number of possible private keys to a little less than 2^256, since not all private keys are secure.

Mining incentives

A way to secure the network normal and protect it from attackers, Bitcoin found a way to incentivize the validation of new and honest blocks. To do so, Bitcoin does two things:

1. Issuing a block reward for each mined block;
2. Collecting transaction fees.

To validate blocks, the protocol uses an up to date version of Hashcash [Back et al., 2002], which is a PoW mechanism. The function runs on each participating node, which then brute-forces a adjusted value, called "nonce", until one of nodes reaches a necessary hash value and receives the block rewards and the corresponding fees from the last block. This transaction is the first one of the new block.

Blocks are generated approximately every ten minutes and this is achieved by increasing the complexity of the hash to find if blocks are mined below that time threshold and vice versa. In the original Hashcash specification, the complexity of the hash to find could only be doubled or halved by changing the number of leading zeros of the hash, whereas in Bitcoin, the complexity can be changed more gradually, by considering the hash as a very big integer and the hash to find being one below that integer. By taking into account the total processing power of the network, the function tunes its complexity to always stay close to the ten minutes [Nakamoto, 2008].

The incentivization works also to prevent Sybil attacks in Bitcoin. A Sybil attack is a type of attack which can occur in decentralized networks, where power is seized by fake entities, controlled by a unique person or organization. In the case of Bitcoin, a Sybil attack may happen if 51% of the hashing power belongs to the same entity, ending up controlling the output of the network.

So effectively, the consensus mechanism is probabilistic and works as a lottery between the nodes, biased towards the nodes with more processing power. As a mention, Bitcoin goes also through "halvings", where the block rewards go down by half every 210000 mined blocks, so with time the network relies more and more on transaction fees and the actual value of the coins for large scale mining to be profitable.

3.3.2 Challenges

There are a few challenges associated with Bitcoin's usability and traceability of transactions. First, as the network usage grows with more and more transactions submitted and constrained by a small (1MB) and fixed block size, network congestion will eventually occur. As a consequence, transactions aren't immediately submitted to the blockchain on the next block but instead go through a memory

pool that acts like a priority queue, where a transaction's rank depends on a paid fee.

The higher the fee, the faster a transaction will get confirmed and as usage grows, the fee threshold for a transaction to be accepted will rise, making the blockchain very expensive to transact directly at one point. The main answer for this nowadays is to use what we call *layer 2* solutions, which deal with transactions themselves through other means, making Bitcoin act like a settlement layer. The most well known layer 2 for Bitcoin is the Lightning Network [Antonopoulos et al., 2021].

Another challenge is the lack of privacy of users using the blockchain. Since blockchain has an open ledger, everyone can see where coins flow and even though public addresses have no associated name, a single identified transaction reveals all the movements from the identified user. In Lightning Network again, private channels exist, and are mentioned in [Antonopoulos et al., 2021], where users can secretly exchange coins, although the privacy of the users depends on the trust in others to not reveal the information, making it not an ideal solution.

Finally, the PoW consensus mechanism, while being the most secure one known, is very intensive on energy costs, due to the existant competition for block rewards.

Reviewing these challenges is important for a platform hosting biography of things, since a lot of data will be flowing for each product and scalability has to be assured and the privacy of some data is also a concern. The most relevant aspects of Bitcoin for our supply chain solution have been reviewed. Other projects under review, starting by Ethereum have some of these implementation choices reflected in them, further emphasizing how important Bitcoin is for the blockchain technology.

3.4 Ethereum

To better understand the progress in blockchain technology, it is important to also review the Ethereum project. This blockchain is heavily influenced by Bitcoin yet was a driving force for innovation. The innovative elements will be the main subject of the review, such as the popular smart contract and tokenization functionalities.

The Ethereum blockchain started its development in 2014 and its intent was not to replicate the work done in Bitcoin or deny its intrinsic value, but to allow developers to build programs in a Turing-complete programming language in a blockchain. One of the original founders, Vitalik Buterin, stated in [Buterin, 2015a] that the project's goal was to merge the idea of decentralized money with programming capabilities, using programs called smart contracts by making a distributed state machine, rather than a simpler distributed ledger.

In [Wood, 2022], transaction data are called "messages" outputed by "accounts", that can either be external entities that own a private key, or internally by the

smart contracts. As opposed to Bitcoin, Ethereum's data is stored by state, instead of UTXO data, but accounts work similarly to Bitcoin wallets in practice, besides a few more fields containing the following:

- An Ether (ETH) balance, not to be confused with Ethereum, which is the cryptocurrency hosted by the blockchain;
- A counter for processing transactions only once, called nonce;
- A storage that can contain variables and Ethereum Request for Comments (ERC) tokens, explained further below.

Smart contract accounts also contain their code stored inside in addition to what was listed and the global state of these smart contracts and all accounts is what's maintained by the blockchain, as opposed to Bitcoin's UTXO model. In the following section, we will cover in detail how contracts are deployed and how they can be used.

3.4.1 Smart contracts and decentralized applications

Smart contracts are a term used for programs stored on a blockchain. This idea is not new [Szabo, 1997] but Ethereum's implementation is what made the term popular. They can only be executed after receiving a transaction from external accounts and these programs can have a multitude of purposes, such as:

- Create financial applications or services and automate transactions, based on on-chain conditions;
- Define "tokens", which are sub-currencies that exist in Ethereum;
- Decentralized exchanges, between ETH and other tokens;
- Decentralized Autonomous Organizations (DAOs).

Network fees

As already mentioned, the smart contract has to be deployed in order to be used by external accounts. Both its deployment and usage costs a certain amount of ETH, called gas and the gas price depends on the usage of the network and how congested it is, making use of a similar memory pool as Bitcoin. This gas fee has to be paid to cover for incentive and energy costs of the nodes participating in the node, since the code is also executed by them.

These gas fees are also applied to generic transactions of ETH from one account to the other and the amounts are most commonly represented by gwei which equals 10^9 ETH and is awarded to the node participants, in a similar fashion as Bitcoin. Other units are pre-labelled in [Buterin, 2015a].

Solidity and Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) is used to run the smart contracts from EVM bytecode when they receive a transaction, in order to make arbitrary calculations and output further transactions. This is described in [Wood, 2022].

To generate this bytecode, high-level languages are used instead, where Solidity is the most prominent one for Ethereum. It is influenced from C++, Python and JavaScript and shares a lot of design choises from these languages, including object-oriented features and static typing. A prior study [Lopes Barata and Rupino da Cunha, 2019] was also conducted about the understandability of Solidity smart contracts from people with background in these languages. To maintain and run consistent bytecode generation with improvements to the Solidity language, it is required to specify a version or a range of versions of the language in the beginning of every contract file. Otherwise, legacy smart contracts would not run properly anymore.

The use of smart contracts is very important for our potential solution, in order to give more trust into the data around every product biography. This is achieved because smart contract transactions do not need verification from intermediate entities. Other advantages of smart contracts include:

- Speed of usage, as opposed to traditional business processes;
- Safety, since every smart contract is tamper-proof;
- Accuracy of results, due to the automated process.

One of the use cases of smart contracts is to represent "things", called tokens, that needed standardized interfaces for regular use between other smart contracts.

3.4.2 Token standards

Tokens in Ethereum are used to represent virtually any object in the blockchain and standards exist to improve the usability with other smart contracts. These standards are named ERC and can also represent application-level methods used by the tokens themselves.

For these standards to exist, Ethereum uses a decentralized approach for development with Ethereum Improvement Proposals (EIPs) where these new standards are first proposed, go through a review from the community that can choose to favour, oppose or ask for further revisions of the standard and only gets implemented after its acceptance. A review of three of the most popular standards is done down below.

ERC-20

It is the most popular standard used, stated by [Fröwis et al., 2019], and is very commonly used to represent, *subcurrencies*, the name Ethereum uses for alterna-

tive currencies that exist inside the blockchain. Complying the smart contract code to this standard to build a custom currency makes it trivial to use with already existing wallets, without extra steps for integration. These tokens are also fungible, which means every token issued has the same value as all the others.

ERC-20 specifies six different functions, described in [Fabian Vogelsteller, 2015] all necessary to query the token's balances, total supply and making transfers between external accounts or through a smart contract, with approval.

ERC-721

Also known as Non-Fungible Token (NFT), as the name states it is a standard to implement tokens, in the same manner as in ERC-20, but without being fungible. This means each token is unique and identified with a *tokenId* number, used to link the token to the signature of data or even as the documentation states, encoded images or real data, up to 256 bits of data [Fröwis et al., 2019].

ERC-1155

This standard can represent both fungible and non fungible tokens and is an improvement of both ERC-20 and ERC-721 standards, both in efficiency and correcting implementation errors. Its functions are similar to the mentioned standards, except that they allow for multiple requests in a single call.

Other standards are still in EIP phase to fix known bugs or unintentional behaviour and give additional functionalities such as to allow users to reject incoming tokens from blacklisted addresses, to make the miner of the block, instead of the sender, pay for the gas fees while transfering tokens (ERC-865) or have a new token type that can only be transferred once, sticking to the account afterwards [Buterin, 2015b].

Conforming to these standards can be very useful for replication in a future solution to facilitate law regulations and interoperability with other blockchains, especially those compatible with the EVM.

3.4.3 Decentralized Autonomous Organizations

A DAO is an organization based on the blockchain and structured with smart contracts. Usually, the participation requires the ownership of some kind of token and smart contracts functions that enable proposals for voting, adding or changing features in the DAO. Organizations governed this way have one inherent advantage which is a completely decentralized and fairer rule by obeying the code and the voted proposals. Voting power for proposals is usually reflected in the amount of tokens owned and delegated to be part of the DAO, although other governance methods are available. This can lead to some problems for the organization if unregulated and not audited properly, like vulnerability to Sybil

attacks, mentioned in Section 3.3.1. The World Economic Forum recently published a paper about this concept [Aiden Slavin, 2022].

Although this type of governance is less used in enterprise blockchains, since the configuration of a permissioned blockchain has already a type of governance, it remains an interesting idea for the solution.

3.4.4 Scalability issues

Similarly to Bitcoin, Ethereum has had scalability issues hindering its popular adoption by having exorbitant and volatile gas fees for even the simplest transactions, this led to a lot of research for a solution.

Recently, Ethereum upgraded to the version 2.0 of Ethereum, which aimed to make the network more sustainable and cheaper to use. The network switched from a PoW to PoS consensus mechanism, merging with an already live version of a new blockchain, called Beacon chain. This upgrade also enabled the process of implementing sharding, which is a scaling method consisting of dividing the network in smaller group of nodes, called shards, to validate transactions in parallel and reduce congestion of the network.

PoS is also based on a lottery system to decide who gets the block reward, like PoW, but instead of deciding the winning node by doing intensive computing to find a hash value and upping the odds with more processing power, the probability is decided by the amount of currency a node "stakes" to validate the network.

Other proposals with new features, such as the EIP-1559, have been improving Ethereum's base code to reduce this volatility [Roughgarden, 2020], but no measures have significantly upgraded the overall scalability issues of the main blockchain layer so far. These issues are represented by the blockchain trilemma shown in Figure 3.5.

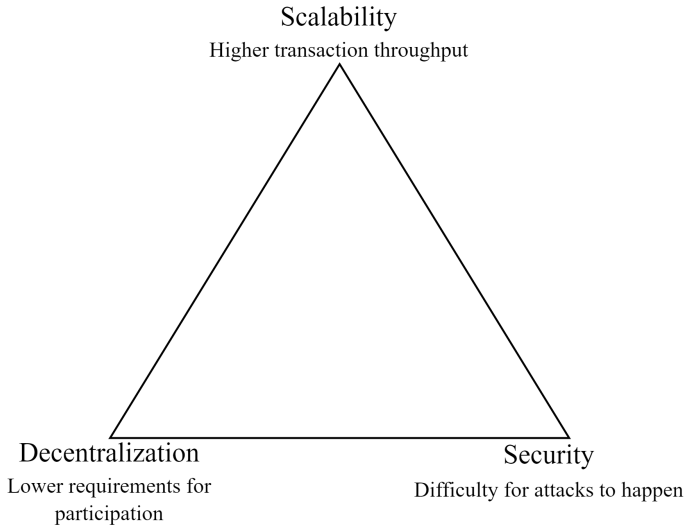


Figure 3.5: Scalability trilemma, adapted from [Karaarslan and Konacaklı, 2020].

[Buterin, 2015a] states that every blockchain network has to decide in a trade-

off between scalability, decentralization and security. Both Bitcoin and Ethereum have a vision opting for a combination of decentralization and security, leaving the scalability issues to be mitigated with layer 2 solutions.

Taking advantage of the programmability of Ethereum, other processing layers have been developed, alike to the Lightning Network for Bitcoin. They can be separated in two groups:

1. *Sidechains*, which are different EVM compatible blockchains that have different parameters, run independently and are connected to the Ethereum blockchain through two-way bridges. This makes it possible to "connect", through a smart contract, tokens from Ethereum's main network to these sidechains. Unlike real layer 2 solutions, sidechains usually don't update states or post transactions to the main blockchain;

2. What Ethereum effectively calls layer 2 solutions, are blockchains that inherit the security properties of the main network. These solutions always post the new states and bundles of transaction data.

Other blockchains with EVM compatibility and different implementation decisions, such as different consensus protocols or block parameters, also emerged with the main goal to provide a platform with the same capabilities but without the gas fee tradeoff.

As Ethereum got more popular with the usage of smart contracts, they became the foundation of what are called Deecentralized Applications (Dapps) and the decentralized web or Web 3.0. Dapps use smart contracts as a backend foundation and a web interface to make applications hosted with the help of decentralized blockchain technology, as opposed to hosting on centralized servers [Buterin, 2015b]. With the usage of Ethereum external accounts and events emitted by smart contracts made to be read by any external program, this internet is much more personal and in control of the users.

In the following section, we will evaluate how Hyperledger open-source projects differ from cryptocurrency platforms. Narrowing down this research into enterprise platforms gets closer to the specific use case related to the thesis.

3.5 Hyperledger

A review of Hyperledger enterprise blockchains is presented in this section. Hyperledger is a community hosting and responsible for a number of frameworks, libraries and tools suited for enterprise-grade DLT deployment. It was founded in 2016 and is part of the Linux Foundation, along with some other well known projects such as Kubernetes, NodeJS or the Linux kernel [Labs, 2022b]. These projects use a lot of concepts from the previously reviewed blockchains, but with different purposes from cryptocurrencies.

Unlike blockchains used in cryptocurrencies, enterprise blockchains can be fundamentally different in function. The inherent characteristics of using a solution based on a blockchain can be applied to many different industries and Hyperledger offers back-end technologies for a wide range of practical applications.

Currently, the Hyperledger foundation hosts more than ten projects with backing from more than a hundred industry members, with names such as IBM, American Express and Oracle. As for the project themselves, they are divided in different DLT and network implementations, libraries and tools to support them and are all centered around blockchain technology, as shown in [Hyperledger, 2021].

These projects, including Fabric and Sawtooth which are reviewed next, are all open-source, i.e. their code is available publicly and has permissive licensing for modification and distribution. Their components will be reviewed in more detail, since the network implementation of this thesis is formed around them.

3.5.1 Fabric

Fabric started to be developed by IBM, being integrated later on in Hyperledger. It is a back-end allowing organizations to form blockchain networks divided by channels and composed by two different kinds of nodes, *peers* and *orderers*, managing the network. Its main goal is to be a modular architecture for permissioned blockchains, allowing for the use of various pluggable components like databases, smart contracts, different consensus protocols, and membership management.

As Fabric is currently in development, this thesis is describing the structure of the latest version released, `v2.4.6`. Since this version is relatively recent, most of the review is done around the official documentation, [Hyperledger, 2022b]. A simple Fabric network is shown in Figure 3.6, with the following key components:

- *Organizations*, representing different entities in the network;
- *Membership Service Providers (MSPs)*, acting as identity providers of these organizations;
- *Channels*, a representation of a network with different organizations;
- The *ledger*, which is composed by the blockchain and the current state of data;
- *Chaincode*, a term used interchangeably with smart contract, are functions that define the transaction logic on the channel;
- *Peers*, responsible for endorsing and executing transactions from the chaincode;
- *Orderers*, a node that does transaction ordering and writes data to new blocks;
- *Applications*, interacting externally with the network.

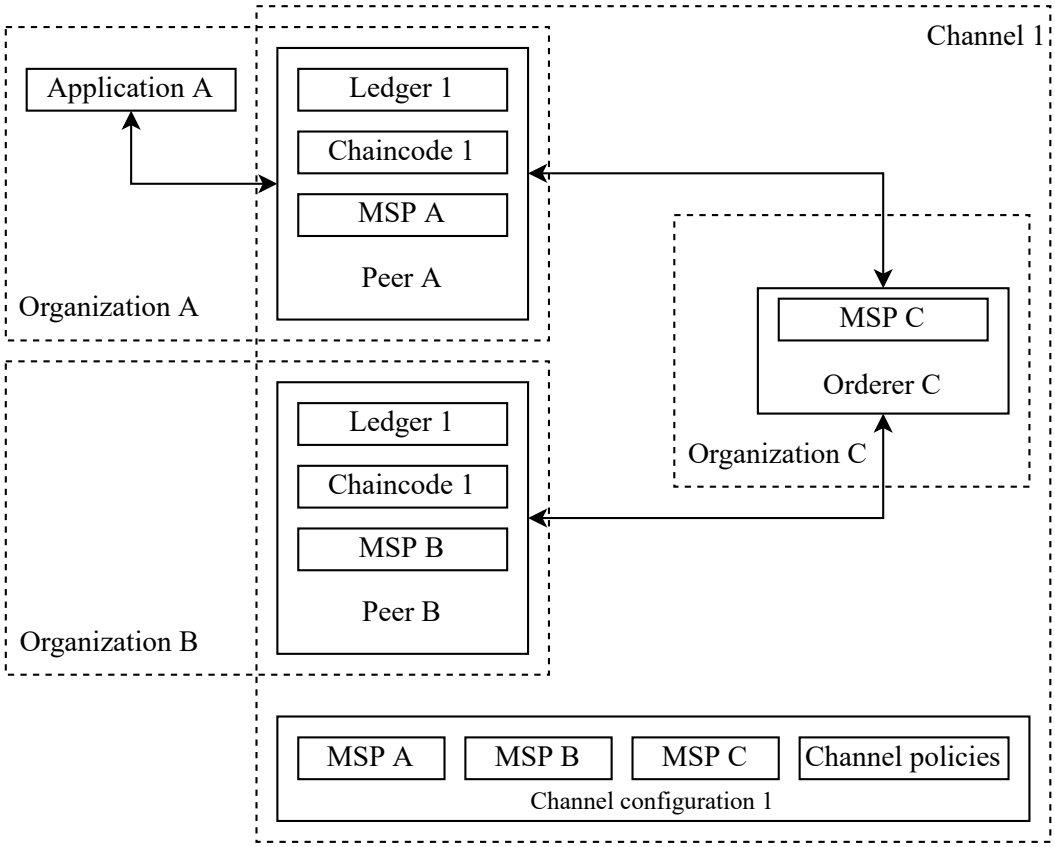


Figure 3.6: Components of a simple Fabric test network, adapted from [Hyperledger, 2022b].

Identity

Organizations represent a singular or group of entities interacting in the Fabric network. These entities are identified in the organization through a MSP, composed of identities issued by a Fabric Certificate Authority (CA), trusted by its own organization and assigned a role. More details about MSP's structure are documented in [Hyperledger, 2022h]. It is only possible to participate in the network using this identity and being registered in the network.

Each organization may be divided in different Organizational Units (OUs) and each entity can have different roles in the network, through a special OU called *NodeOU*. The available roles in NodeOU are *client*, *peer*, *admin* and *orderer*, and each one also has an associated certificate. As an example, with NodeOU enabled in an organization, only an entity with a certificate issued by the peer role may perform functions related to the peer node, described below. Finally, a Fabric CA server may be used to register new organizations, along with generating certificates for identities and other certificates used for encrypted communication through TLS. Specific documentation about the Fabric CA is also available in [Hyperledger, 2022a]

Channel

A blockchain network in Hyperledger Fabric is commonly called a channel. Referring again Figure 3.6, it is composed by two different nodes participate in it and a channel configuration.

The channel configuration has the identity of every organization participating in it, also called the *Channel MSP* along with the policies of the network, defining which entities from which organizations can do certain operations. If a new organization has to be added to a channel, the configuration has to be changed and agreed between the administrators to support new nodes with redefined roles. This is different to permissionless networks reviewed above, where the capabilities and policies of the network are public and the same for every participating entity and where each configuration change requires a software update from all nodes.

Each channel also has its own ledger, composed by a blockchain and a *world state*, representing the current value of objects on the ledger. As mentioned in [Hyperledger, 2022g], having an additional world state database is faster to obtain the current values, instead of performing queries in a blockchain. Fabric offers two options for the database, *LevelDB*, which is a simple key-value database, and *CouchDB*, a more complete option for ledger states structured in objects, such as in JSON format, yet any type of database can be plugged in, therefore the options for this database are not restricted to these two.

Chaincode is also a part of the channel definition that must be approved by the channel administrators in order to be invoked. Each chaincode may be upgraded into new versions and have their own *endorsement policies*, and define who, in the network, has to verify transaction outputs before they get validated. Chapter 5 takes a closer look into the chaincode deployment and transaction lifecycles, as part of the implementation of a Fabric network. Depending on how the chaincode is implemented, compatibility with token standards is possible, such as Ethereum's ERC reviewed in Section 3.4.2, in order to provide interoperability possibilities.

Nodes

As mentioned before, there are two kinds of nodes in Hyperledger Fabric: peers and orderers.

Starting with peers, these are the participants responsible for executing and endorsing transactions, through the *Fabric Gateway service*, a new concept from the latest version of Hyperledger Fabric in [Hyperledger, 2022j]. Transactions can have two types:

- Read: This type only needs the proper authorization through the certificate to read the ledger from a single peer;
- Write: Depending on the endorsement policy of the chaincode, this type

requires additional steps to validate the transaction, such as having other peers *endorsing* it, by repeating the execution and signing their response, using a cryptographic function, and submit the transaction to the *ordering service*. The complete steps are mentioned in [Hyperledger, 2022j] and will be taken during the implementation process in Chapter 5.

An orderer, on the order hand, doesn't store any chaincode data but instead, orders the results onto a new block and sends it back to the peers for validation. The validation process of each peer, referred in some parts of the documentation, such as in [Hyperledger, 2022i], follows the subsequent steps:

1. Ensuring the transaction have been endorsed by the required organizations;
2. Verifies that every endorsement's results match;
3. Makes sure the recent committed transactions are not invalidated by other recently appended transactions.

If any of these verifications fail, the block is not added to the blockchain and the ledger is not updated.

As of the latest Hyperledger Fabric version, every orderer uses a Crash Fault Tolerance (CFT) algorithm to replicate the ordering decision to different orderers, if they exist. This algorithm is called *RAFT* and is introduced in [Ongaro and Ousterhout, 2014]. In the RAFT protocol, every orderer in a cluster may have a *follower*, *candidate* or *leader* states:

RAFT uses terms, starting when no leaders are available and one has to be chosen through an election, or when an arbitrary time value elapses. The next points describe, in order, the election procedure:

1. An election starts by giving each follower a random election timeout, usually between 150 and 300 milliseconds, and when the first follower's timeout elapses, its state changes to *candidate* and starts sending vote requests to the remaining followers;
2. A follower, upon receiving a vote request and in case it has not voted this term, increments its term number by one and votes for that candidate;
3. Finally, after every candidate have sent and received their votes, the candidate with the majority of votes becomes the leader.

After the election, the leader's duty is to send requests to the followers in order to reset the term time and to notify that he is still alive. Only the leading orderer receives the validation requests from the peers, but still requests every follower for their blocks. This

External applications can also interact with the fabric network and request transactions with the required authorization, by talking to a peer's gateway service

from a suited client API available in different languages. In Chapter 5, a client application is implemented using this exact API.

Finally, Hyperledger Fabric also implements a gossip protocol, documented in [Hyperledger, 2022f], for better data integrity and consistency. It is used to anchor available peers from different organizations, detect peers recently gone offline and disseminate ledger data to out of sync peers and new peers in the network. The pings sent to detect alive and dead peers, also called heartbeats, have a programmable frequency and use signed data from their certificates, so malicious peers can not impersonate others

Going back to the scalability trilemma in Figure 3.5, Fabric and other permissioned blockchains use the security plus scalability combination, since they don't face the trade-off of having a public network. Finally, the Fabric architecture makes it the most popular project on Hyperledger, since it covers a wide range of industry applications, our use case included.

3.5.2 Sawtooth

Hyperledger Sawtooth is another enterprise blockchain platform. Intel started its development and the overall structure presents some differences when compared to Fabric. Figure 3.7 shows how a Sawtooth network is structured, with the internal look of a node.

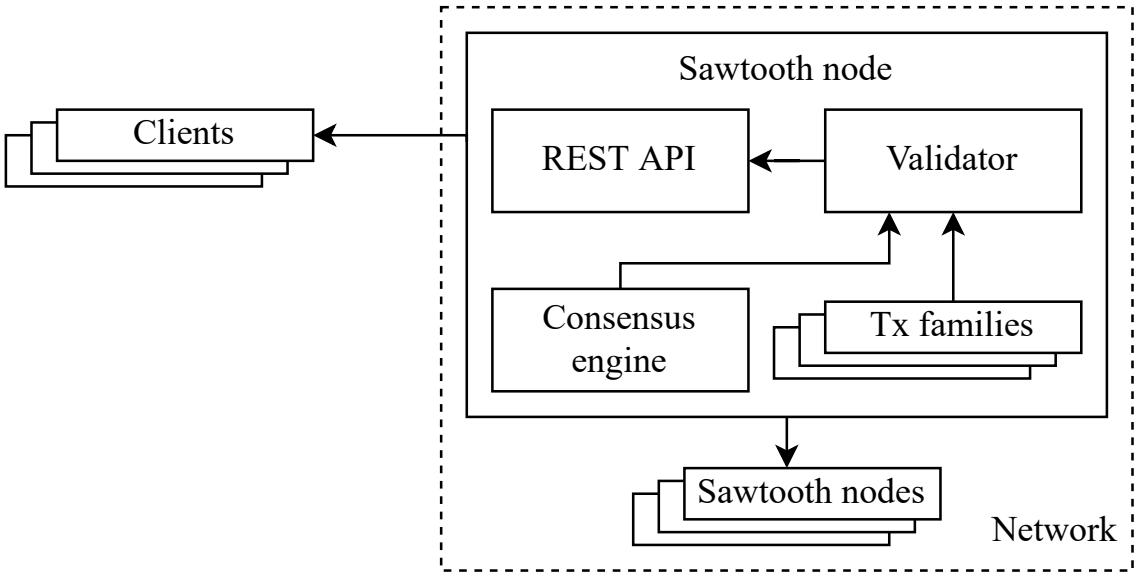


Figure 3.7: Example of the Sawtooth network, with different components and transaction families, adapted from [Hyperledger, 2022l].

While Fabric uses different types of nodes, such as orderers and normal peers, Sawtooth uses a unique type of node, shown in Figure 3.7. Each internal component of a Sawtooth node is detailed in the following subsections.

Validator

The validator is the main component integrating the entirety of the node's functions, being responsible for maintaining consensus, validating batches and creating blocks, and coordinating communication with the other nodes and external clients [Hyperledger, 2022o]. It is also responsible for receiving proposed transactions and blocks from other nodes, in order to validate them with their result and a set of rules for verify their integrity and authorization.

Batches are used to reduce possible congestion from hundreds or thousands of transactions in a short period of time, which are atomic units of changes with ordered transactions, executed in parallel. The structure of batches, transactions and their headers, as well as detailed information about these data objects is documented in [Hyperledger, 2022n]. The transaction family processors get delegated by the validator, after receiving requests from other nodes or from the REST API to perform and validate these transactions.

Transaction families

A transaction family is composed of written in different programming languages and implementing the business logic of an application. They can be used in the same manner a smart contract is used in other blockchain networks, or as an interface or compatibility layer for easier development.

Another relevant feature of Sawtooth is the ability to use both on-chain and installed smart contracts, as opposed to only installed in Fabric or only on-chain in Ethereum. By default, Sawtooth already provides samples in Sawtooth's core repository for essential operations, such as the following, taken from [Hyperledger, 2018]:

- Collect information about the ledger's blocks;
- Identity and policies management;
- Configure on-chain network settings;
- Compare blockchain system performance;
- Test the network with transactions;
- Compatibility layers, such as the *Sabre* transaction family, which implements on-chain smart contracts executed in a WebAssembly virtual machine; and the *Seth* transaction family, used to run EVM compatible smart contracts.

Transaction families have transaction processors, written in different programming languages and implementing the business logic, and client interfaces, usually exposing REST API endpoints or providing a Command-Line Interface (CLI).

Network communication

Clients can interact with these transaction families through a optional and customizable REST API available, with the necessary permissions. An event system is also put in place to broadcast changes, where applications can subscribe to them to gather data and relay information back. As for the rest of the communication between nodes, also represented in Figure 3.7, Protobuf serialization is used for better efficiency and speed.

Through Seth, using Solidity smart contracts is possible, also opens new doors for compatibility and interoperability options between these networks, using the same contract code.

Consensus engine

In order to build an agreement between nodes, Sawtooth offers the possibility to use consensus protocols suited for different permissioning levels of the network. RAFT and PBFT are used for permissioned consensus. Sawtooth's PBFT implementation uses an improved version of the original mechanism with dynamic network membership, regular view changes, and a block catch-up procedure, detailed in [Hyperledger, 2022l].

PBFT is a leader-based algorithm always assumes that messages going through different nodes can not be always trusted, because it's assuming the channel is insecure [Castro and Liskov, 2002]. It starts by electing a leader, which will, in the case of a blockchain, set the current caller for agreement by everyone. If one or more nodes are malicious, including the leader, they can only control the final state by being above 1/3 of the total nodes. Otherwise, no matter what states the malicious nodes emit, the legitimate nodes go by the majority of votes received and agree upon this result. An example of this is shown in Figure 3.8, where we represent this mechanism and state X as the legitimate blockchain state and Y as the malicious one.

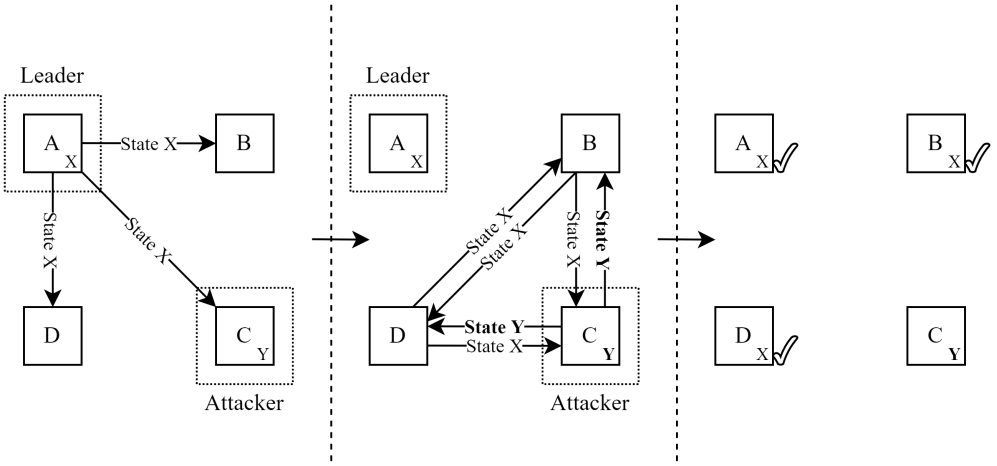


Figure 3.8: PBFT mechanism in three steps, with leader A and attacker C, adapted from [Castro and Liskov, 2002].

Proof of Elapsed Time (PoET) is another option for consensus mechanisms available in Sawtooth. It is a lottery-style consensus protocol, similar to PoW and PoS, where each node has a probability of being the winner of a new block. Instead of using computational power or currency stake, PoET assigns a wake up time to every validator, which in turn validators sleep using a function in a trusted execution environment. Whoever gets the best time and wakes up first is the one writing the block.

To avoid malicious manipulation, such as changing the given time to a shorter period, Intel created a trusted environment for these kind of code executions, called Software Guard Extensions (SGX). PoET, in practice, has two versions. One is only crash-tolerant, much like RAFT, but enables the use of this protocol with any Central Processing Unit (CPU). Whereas only modern Intel CPU can take advantage of the Byzantine tolerance part of the protocol, which in turn also enables Sawtooth to become a permissionless network.

For testing purposes, Sawtooth also has a developer mode consensus protocol available, which just picks a random leader to be in charge of validating the network.

Despite not being as popular as Fabric, Sawtooth might be a powerful option as a backend to host biographies of things, especially with the use of Grid, reviewed in the next subsection.

Grid

An useful tool to be used with Sawtooth is Hyperledger Grid. It is composed by a number of smart contracts to be ran on the Sabre engine, mentioned above. These smart contracts are particularly useful for a use case of supply chain management, dealing with schemas for product creation compliant with GS1 standards [GS1, 2022b] already implemented. These standards were referred in Section 3.1. In summary, the smart contracts available from Grid enables:

- Definition of schemas for the creation of products;
- Insertion and updates of product data;
- Location definition and updates for products;
- Access control of the previous points.

Grid can be installed in the same machine as a Sawtooth validator, and provides a web interface, a CLI tool and a REST API for user interaction, making it possible to have different options for integration of clients and actors, such as IoT devices.

With a proper initial configuration of the Sawtooth inner layer, it is possible to interact only with Grid interfaces for data manipulation.

3.6 Final comparisons

After reviewing Bitcoin, Ethereum, Hyperledger Fabric and Hyperledger Sawtooth, final comparisons are due to summarise important aspects of each of these projects, and for the decision making needed in the design of the architecture and implementation phases. Since each blockchain project was reviewed differently to highlight their notable attributes, this section aims to succintly compare each of these qualities, to give a better basis for our subsequent work phases.

Node operations vary highly in each implementation, because each project has different objectives and priorities. Table 3.2 summarises the difference in different domains for these nodes.

Table 3.2: Comparison between node functionalities in blockchain networks.

	Bitcoin	**Ethereum**	**Fabric**	**Sawtooth**
Permissioning	Open	Open	Closed	Closed
Storage model	UTXO	State	State	State
Smart contracts	No	Yes	Yes	Yes
Decentralization level	High	Medium-high	Not guaranteed	Not guaranteed
Tx submission	Always	Always	Policy and node type dependent	Policy dependent
Code execution	Non existant	Optional	Policy and node type dependent	Policy dependent
Block writing	Always	Always	Policy and node type dependent	Policy dependent

The mentioned cryptocurrencies show a different permissioning level from Hyperledger, giving the opportunity for everyone to participating in the network, as long as they fulfill the device or staking requirements of their consensus protocols. Fabric and Sawtooth backends, despite having a lower level of decentralization, are more suited for our solution, as they fit a better role scaling and storing larger amounts of data. These tradeoffs were mentioned in 3.4.4.

Out of the projects reviewed, only Bitcoin does not support programmability in its main network. Not every node in an Ethereum, Fabric or Sawtooth network must be able to execute smart contract code, either. Besides that, only nodes with the required policies may execute and validate transactions in the Hyperledger blockchains. In the special case of Fabric, two types of nodes exist, separating the block writing logic.

As mentioned in 3.2.5, each distributed blockchain network needs some form of consensus mechanism, no matter the permissioning level of the network. However, it's the type consensus that defines if a blockchain may be permissioned or permissionless. The following Table 3.3 addresses each algorithm mentioned so

far, with some attributes.

Table 3.3: Comparison between consensus protocols in blockchain networks.

	PoW	**PoS**	**PBFT**	**PoET**	**RAFT**
Probabilistic finality	Yes	Yes	No	No	No
Permissionless networks	Yes	Yes	No	Both	No
Byzantine failure tolerance	Yes	Yes	Yes	SGX only	No

Sybil attacks, mentioned in Section 3.3.1, may happen in any distributed network with a degree of decentralization. In the case of cryptocurrencies, it is avoided by incentivization of the network, paying truthful block miners coins for their contribution, in the case of PoW and PoS. For more centralized approaches, these attacks are avoided by identity verification mechanisms.

PoS and PoET algorithms have lower energy consumption than PoW, but are more exposed to centralization issues in a permissionless network, due to PoS requiring value staking and giving more power to richer entities and PoET's SGX version only working with Intel CPUs, restricting the number of potential participants.

RAFT, despite not showing any major strengths in the comparisons

3.7 Summary

With this review, we understood the necessary concepts for a blockchain network to function, going through important aspects of its architecture, such as:

- Permissionless and permissioned blockchains divergences.
- Data security measures, such as hashing functions and public key encryption.
- The need for consensus protocols and incentivization of public cryptocurrencies.

The design decisions made in Bitcoin and Ethereum were examined because they are shared by the majority of active blockchain projects. Despite some of their features not being directly used for our use case, some concepts remain relevant for analysis. With the introduction of smart contracts, programmability in a blockchain is a game-changer and essentially what enables us to build a future solution. To better understand the impact of smart contracts, the following topics were reviewed:

- Differences between UTXO and account-based models.
- What triggers smart contract execution and its cost.

- Some developed programs and standards based on smart contract code.

By analysing these features on widely adopted cryptocurrencies, we can argue that blockchain technology is truly disruptive in the financial sector, and with the introduction of Hyperledger DLTs, this positive change is also available to other industries.

Both Fabric and Sawtooth projects are part of the Hyperledger family of DLTs, which have more of a focus on industry data and applications, with improvements for security and scalability available in a permissioned network. The key differences between enterprise blockchains and cryptocurrencies were also extensively reviewed in this report.

Chapter 4 introduces the first drafts of the proposed architecture, integrating the insights collected during the literature review phase for a potential implementation of a product biography platform, while also briefly mentioning the value of having one.

Chapter 4

Requirements definition and architecture

Following the literature review about blockchain concepts and implementations, this chapter will delve into the creation of a software architecture for an implementation of a blockchain platform to host product biographies.

For that, key practices of the book [Bass et al., 2003] are used, such as meeting non-functional requirements described as quality attributes in this chapter, understanding the requirements of the system and managing trade-offs of the chosen attributes. After this decision-making phase, we will represent our architecture in a C4 model diagram, defined in [Brown, 2022], using three layers of detail to describe the system with its different components and containers.

Before going further, it is important to highlight what the real purpose of a product biography platform is. The usefulness of a product biography, in a real world scenario, can be illustated with the following points:

- Having an immutable history of usage and registry of various events of a product gives a possibility for manufacturers to optimize their supply chains;
- Besides the supply chain improvements, Quality Assurance (QA) is also easier to provide, by identifying potential issues in produced batches;
- By having transparency of data, and ideally by also storing third party data, end users may have more trust in their products and more loyalty to a brand that offers clarity in their processes;
- Regulators can take advantage in this transparency as well, making it easier to verify compliance.

Overall, a product biography must help organizations, end consumers and regulators to meet their expectations, although a proper implementation of a product biography depends on the quality of information that is provided.

4.1 Quality attributes

To measure the usefulness of a product biography platform based on blockchain technology, quality attributes, also referred to as "ilities", are defined in this section, also in a way to delineate which aspects the architecture and the implementation of a prototype have to respect. Following this, Quality Attribute Scenario (QAS) are described telling how the architecture and implementation should respond.

For blockchain networks in general, the following quality attributes can be defined, some of them already being discussed in the literature review:

- Security;
- Decentralization;
- Privacy;
- Immutability.

These quality attributes are taken into account with scenarios for the implementation in the next chapter and validated in Chapter 6. The following Tables 4.1, 4.2, 4.3 and 4.4 illustrate the elaborated scenarios for the architecture and implementation to follow:

Table 4.1: Security QAS

ID	S1
Source of stimulus	End user
Stimulus	The user performs an action, initiating communication between the network.
Artifact	Messages sent during the communication.
Response	The message is encrypted, being secure against external entities with malicious intents trying to perform a man-in-the-middle attack.
Response measures	Network analysis of the messages.

Table 4.2: Decentralization QAS

ID	S2
Source of stimulus	Participating organizations
Stimulus	Actions with different permission levels.
Artifact	Blockchain network.
Response	The control and validation of the actions must have a different permission levels, in order to distribute power to different participating organizations.
Response measures	Implementation of a correct configuration of network, organization and user permissions and definitions.

Table 4.3: Privacy QAS

ID	S3
Source of stimulus	End user
Stimulus	The user stores or queries private product data.
Artifact	Peers of the network.
Response	The end user's privacy is respected, by only allowed the authorized peer to access and change the data.
Response measures	Blockchain or database analysis of encrypted private data storage in the correct organizations.

Table 4.4: Privacy QAS

ID	S4
Source of stimulus	Block writer
Stimulus	An additional block gets appended to the blockchain.
Artifact	Blockchain network.
Response	The responsible nodes must validate and append the new block, making it unable to edit or revert it back.
Response measures	Blockchain analysis of snapshots of previous blocks, verifiability of block hash values.

Achieving these quality scenarios is a continuous process during implementation and runtime and the development of the prototype should not compromise these scenarios.

The projects reviewed in the previous chapter already have a lot of open development and discussion about dealing with the trade-offs of using a blockchain network, but due to the sheer difficulty of approaching these trade-offs, this thesis will not attempt to deal with them. Despite that, the impacted attributes must still be acknowledged:

- Performance, scalability: By choosing a consensus protocol that favorises security and decentralization, the speed of transactions, overall throughput of a blockchain network and the potential size and user base of the network is constrainted;
- Interoperability: For distinct blockchain networks to be interoperable with other networks or systems, additional requirements might need to be met, which could compromise the modularity of data and the level of privacy of the data replicated on these other networks.

It is worth noting that these scenarios and the architecture as a whole can evolve, emphasizing the impportance of documenting the process and adapt to changes, for instance through methods like the Architecture Tradeoff Analysis Method (ATAM), detailed in [Kazman et al., 1998].

Overall, these attributes along with their scenarios, will help guiding the design of the architecture defined below. But before that, high-level requirements for the architecture are also defined in the following section.

4.2 High-level requirements

This section covers a number of high-level requirements lists that a platform of product biographies should respect, after defining the key quality attributes that the architecture must use. These lists have some requirements of more importance than others, and as a method of prioritization, the MoSCoW prioritization technique is used, introduced in [Clegg and Barker, 1994]. MoSCoW is an acronym standing for four different levels of priority, from most to least important:

- M - **Must have**, are mandatory requirements to be implemented;
- S - **Should have**, are important requirements, since they bring much more value to the prototype;
- C - **Could have**, are nice-to-have requirements that do not have a large impact for this development phase;
- W - **Won't have**, are requirements that have no importance for now.

This prioritization is useful to decide what to implement first in the prototype of the next section. The requirements are divided in four sections: *data*, *stakeholder*, *network* and *infrastructure*.

4.2.1 Data requirements

To build a platform for biographies of all kinds of products, data modularity is key in order to support the most diverse number of attributes, needed in a platform composed of generic products. Setting a stricter data model for our products is easier and more predictable to develop and interact with client applications, but restrictive for an architecture with generic products.

In an effort to achieve the best of both worlds, the following requirements define how data should be stored in an immutable ledger and interacted with in the blockchain network:

1.1 The platform **must be** able to gather product data, with a name and its events and transformations;

1.2 The platform **must be** able to represent singular products, as well as batches of products;

1.3 The product data **must have** a name attribute and an unique identifier ID;

1.4 A batch of products **must have** quantity and unit attributes;

1.5 The product data **must be** immutable;

1.6 Each different product **should have** a modular data structure, depending on its needs;

1.7 The product data **could be** compatible with GS1 standards, an industry standard used worldwide for identification and tracking;

1.8 Public product data **could be** used externally, by following established Non-Fungible Token (NFT) standards, such as the Ethereum Request for Comments (ERC)-721 or ERC-1155 discussed in Section 3.4.2;

1.9 The platform **could be** able to gather customer and reusage data, among other types of data from a circular economy point of view;

1.10 Product data **could be** used for a digital textile passport, such as for the digital passport use case, mentioned in Section 1.2;

1.11 Additional data, such as images, **could be** stored outside of a ledger and referenced there.

These requirements, with only a few necessary attributes, help in building a simple data model for a generic product biography.

4.2.2 Stakeholder requirements

A product is generally owned, made and regulated by different but generally predefined entities. In reality, these entities should be replaceable, like changing the owner of a product in a second hand sale. Also, trusting a single entity of doing keeping the records right is not possible due to the likely corruption of the system, so Distributed Ledger Technology (DLT) systems with shared trust and distributed operations are better suited so every participant can approve which entities to trust, whether this system is public or not.

In the case of permissioned blockchains, these requirements can only be applied with policy settings and clear identification of the participants whereas in permissionless networks, smart contracts with whitelists for who is allowed to do certain calls could be used, for instance, but generally every entity can participate in the validation of the network.

Stakeholders must have explicit permissions regarding what they can and cannot do in a network, regardless of whether they are businesses, individual end users, governmental bodies, or automated devices. For this reason, we define a list of requirements for their roles with their permissions:

2.1 Network administrators **must be** allowed to configure the network and define participants and policies;

2.2 Organization users and applications, such as automated Internet of Things (IoT) devices, **must have** different policies to query and input new product data in a ledger;

2.3 Other clients, organizations and applications **must be** able to make personal or transactional data private;

2.4 Any entity **must not be** allowed to edit or delete previously inserted product data;

2.5 Other clients, organizations and applications **must be** allowed to query unrestricted and authorized product data;

2.6 Clients **could be** associated to product data, publicly or privately, with attributes such as ownership.

4.2.3 Network requirements

The requirements for a blockchain network are discussed in this section, after having a better understanding of the type of data required for product biographies and the roles that each stakeholder should play in the network.

Implementing smart contracts in a permissionless network, like Ethereum, would be a possible solution, but the gas costs are very discouraging, as mentioned in Section 3.4.4 and the throughput of transactions is also restrictive, because of the large consensus needed in such a big network. Therefore, in this architecture, the options are limited to permissioned networks. Two enterprise blockchains capable of this task, Fabric and Sawtooth, were reviewed in Sections 3.5.1 and 3.5.2, respectively, with comparisons made between the two. For a better user interaction, it is also necessary to have a adequate interface for an end user, in order to transact with product data.

Succinctly, our ledger has to follow the following requirements to make the most use of DLT technology:

3.1 The network **must have** a shared blockchain ledger of product data;

3.2 The blockchain **must be** distributed by, at least, more than one node from a different organization;

3.3 Different entities **should have** an interface to interact with the network and ledger;

3.4 Manual and automatic transactions to the network **should be** supported by the network;

3.5 Client front-end applications **could be** implemented to query product data.

4.2.4 Infrastructure requirements

The solution also needs to fulfill some physical requirements to better ensure the availability quality attribute of the network and data. To host our ledger nodes, physical hosting as well as cloud hosting solutions, such as Virtual Private Servers (VPSs), Amazon Web Services (AWS) or Azure can be used, so a variety of solutions exist here. IoT devices used for automatic data harvesting, for example, should be capable of submitting data the most precise way possible, with an accurate response time and also in a way the data is relevant and doesn't overload the network. This is hard to enforce, therefore the following requirements will only meet infrastructure requirements for the internal network:

4.1 Each ledger **must be** distributed to every participating node;

4.2 There **should be** at least one node maintaining the ledger for each participating organization;

4.3 Each participating organization **should have** at least one node verifying ledger transactions and smart contract executions impacting them.

These requirements, despite not being absolutely necessary for a working network, are important to bring the wanted value to product biographies and respect the quality attributes mentioned above. In the appendix, requirements are also listed, for further consultation, from Tables A1 to A4. In the following section the architecture is described thoroughly, along with an explanation of the decisions made.

4.3 Architecture

After describing the necessary quality attributes and requirements needed to fulfill our goal, this section will cover the planned architecture for a generic implementation of a blockchain network capable of hosting product biographies. This architecture was designed based on the Hyperledger Fabric backend but can be replicated in other systems, respecting certain conditions which will be described in this section.

The C4 model is a widely used graphical notation to represent software system architectures, created by Simon Brown [Brown, 2022]. C4 is based on four layers, each layer giving more detail and less abstraction than the previous one. In this architecture, three of the four layers are represented:

- The *system context layer*, representing the entities acting on the system and the software systems themselves;
- The *containers layer* adds more detail to these systems by expanding into several containers, which can represent different interfaces, applications and data stores;

- The *components layer* represent parts of each container shown above, displaying what happens close to the actual code.

For legibility purposes, containers and components have been divided into relevant parts as they are being described in this section. A full view of the layer 2 and 3 of the architecture model, in Figures A1 and A2 respectively, are depicted in the appendix of this document.

4.3.1 System diagram

The first layer, represented in Figure 4.1, represents three organizations: *Organization 1*, *Organization 2* and *Orderer Organization*. Each organization has a responsible administrator, having the most permissions within the organization; and a regular user, except for the Orderer Organization, with different permissions suited to different functions. Each organization is registered in a channel, called *Channel 1*.

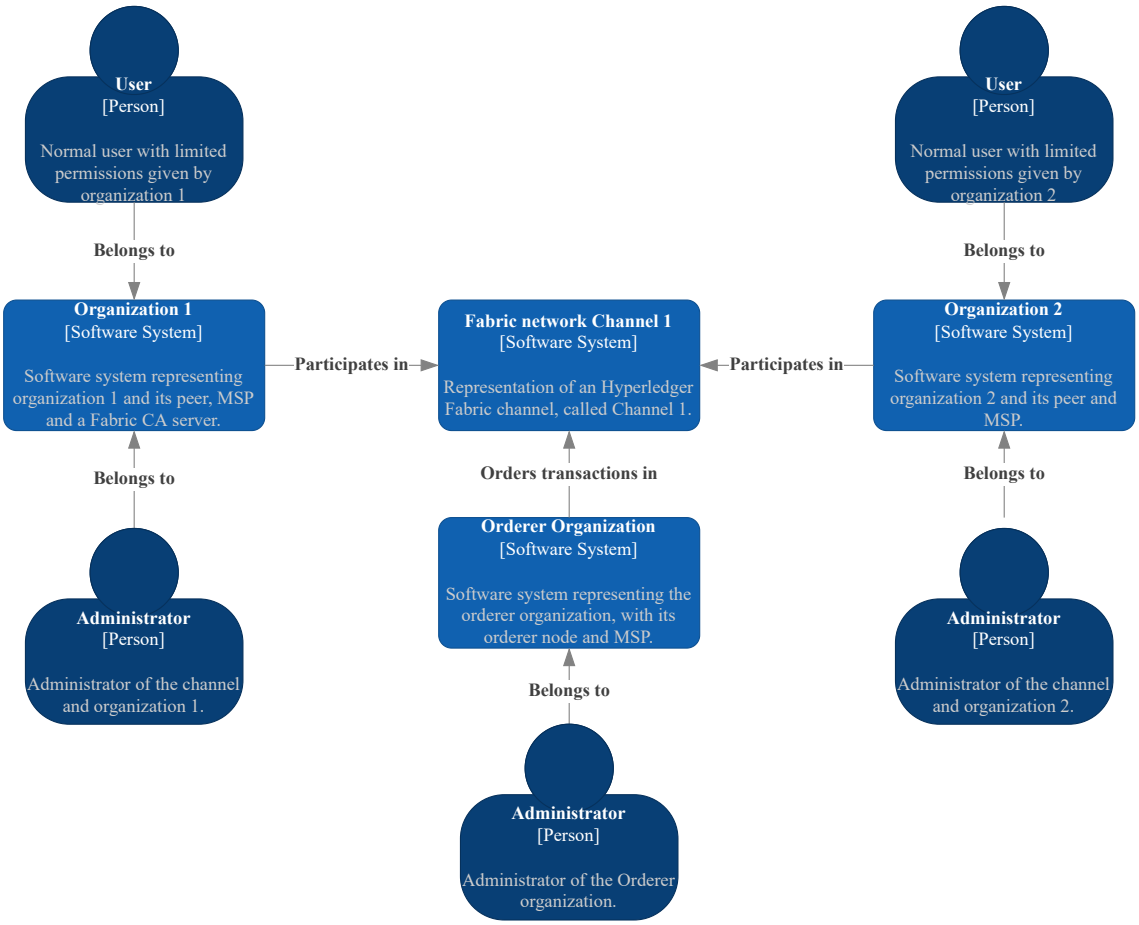


Figure 4.1: Layer 1 diagram of the C4 architecture model.

The functions and capabilities of the members of each organization in a channel is described in the lower layers. In reality, each organization can have more participating members and belong to more channels, while a channel can have

any number of organizations. A more detailed look at the structure of one of the organizations and the channel is provided in the following section.

4.3.2 Container layer

Going to the second layer, high level technical blocks of each organization and the channel systems is shown, starting with the organization diagram. Only the Organization 1 structure is displayed, that is because the Organization 2 structure should be the similar, only with the exception of the Fabric CA server.

Organization system

Figure 4.2 represents how both the administrator and user of an organization can interact with the network. Two main functions are represented in this diagram:

- The enrollment, registration and identification of members, through a *Fabric CA client*, a *Fabric CA server* and the *Local Membership Service Provider (MSP)*;
- The interaction in the Channel 1, either directly through a peer called *Peer 1* or a *Client Application*.

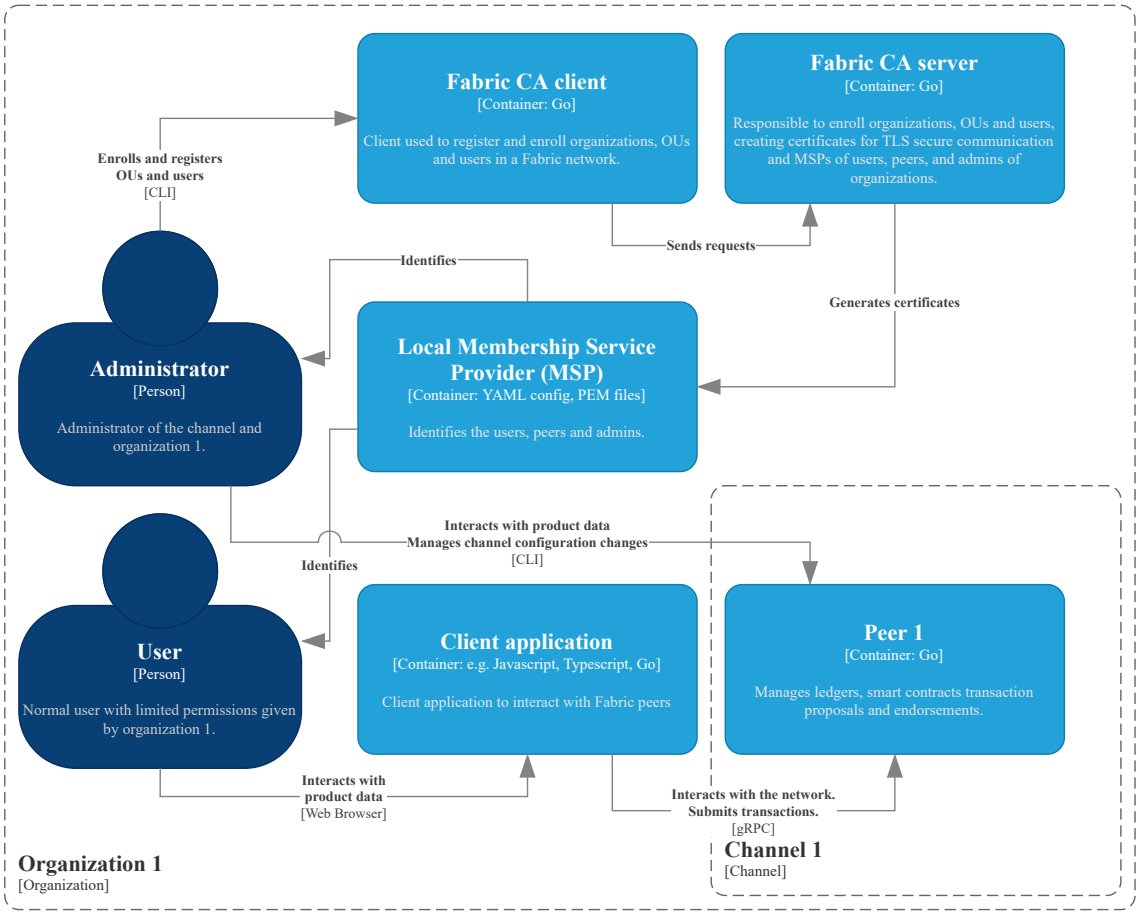


Figure 4.2: Organization system diagram of the C4 architecture model.

In this case, the administrator is responsible to register and enroll organization members into a Fabric Certificate Authority (CA) server, which is the source of trust of CA certificates in a Fabric network, in this example. This means that other organizations, such as Organization 2, have to get their identities from the same root of trust, in this example to the same CA server. Note that in a real network, it is possible and even recommended to use replicated CA servers in other organizations, while connecting to a high availability proxy demonstrated in [Hyperledger, 2022a] and sharing the same certificates database, in order to share the same registered and, if needed, revoked identities.

In a production environment, more fabric CA servers can be implemented to issue certificates, having to be balanced by a HA proxy and having to share and/or replicate their identity databases. PostgreSQL or MySQL are currently the supported databases to run in a cluster, whereas SQLite is the default one for a single fabric CA server.

Along with the certificates, MSPs have configuration files composed of each identity, optionally structured in Organizational Units (OUs). These units can divide different members with different permissions, both locally and in a channel, as described in 3.5.1. Members of an organization with provided identities can interact with the channel in different ways:

- The Organization 1 user accesses the network through a client application, using a library that interacts to the peer through gRPC, a Remote Procedure Call (RPC) framework using Protobuf messages developed by Google;

- The administrator can access the peer directly, through a Command-Line Interface (CLI) shown in the next section, to interact with data directly by invoking smart contracts to read or input data. The CLI interface is not limited to an administrator, but it's impractical for regular users to access a device that way.

In this architecture, it is assumed that the administrator of Organization 1 also has administrative privileges in the channel, defined in the following Figure 4.3, meaning that he is also able to manage channel policies. An organization can run multiple peers and orderers, also depending on the authorization given by the channel policy, with its structure shown in the following section.

A client application more adequate for a regular user is also shown, interacting with the network in an easier way, shown later in Figure 4.4. Despite not being shown directly, it is assumed an administrator has every permission a regular user has, so it is also possible for him to interact with this application. This administrator, by having access to the device where the peer is running, can also interact directly with it, through a CLI, displayed in Figure 4.5, in the next architecture layer.

Channel system

The view pictured in Figure 4.3 represents the minimum parts of a channel necessary for a proper blockchain network. Every channel must have a group of policies, associating identities generated by a CA with capabilities in the channel, such as:

- The ability to read and write from the ledger;
- Who is able to maintain peers and orderers in the channel;
- Who are the administrators of the channel.

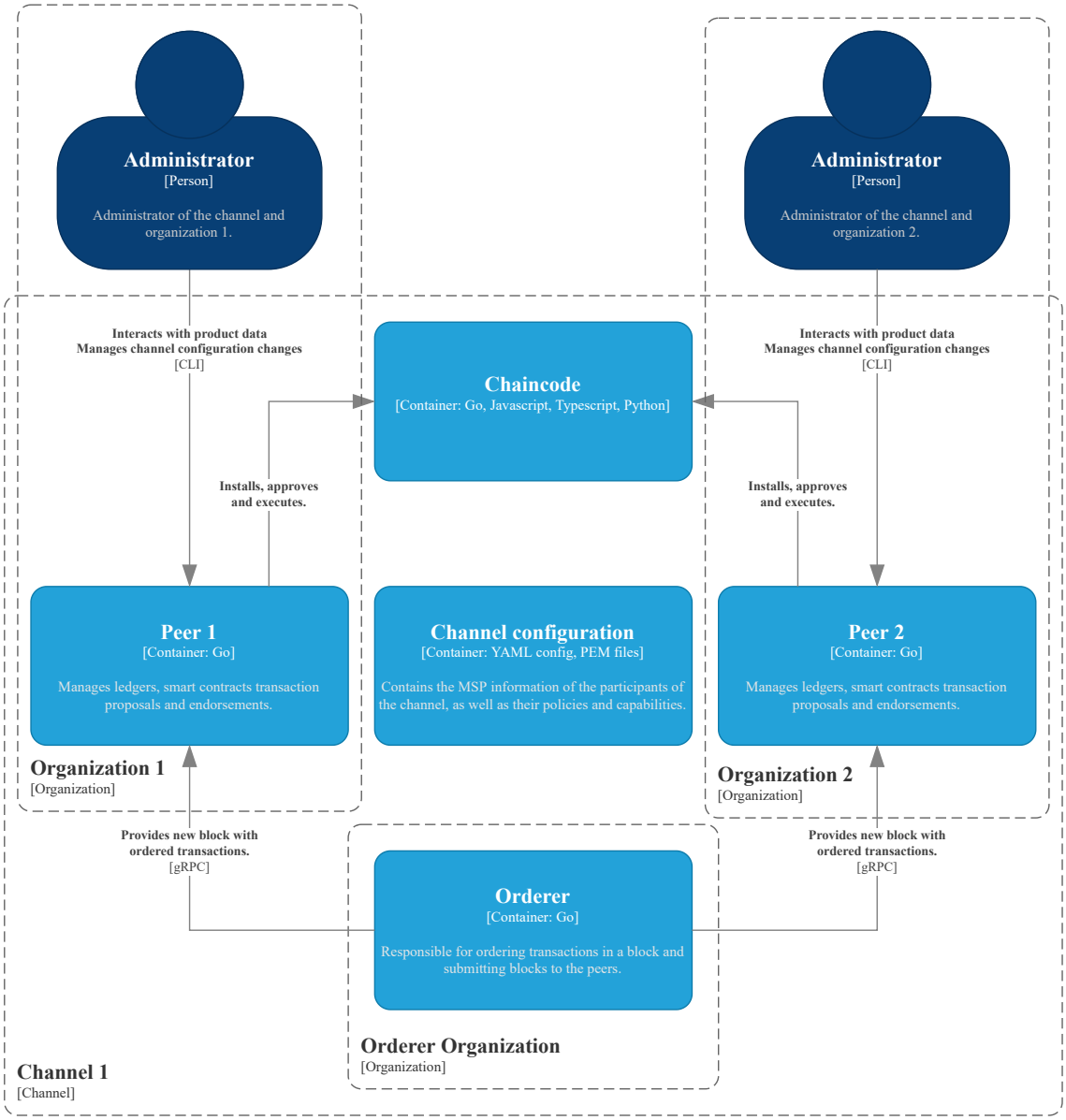


Figure 4.3: Channel system diagram of the C4 architecture model.

In order for a blockchain network to have a real purpose, its ledger has to be distributed and validated between different entities. This is why, in this example,

two peers are responsible for executing and endorsing transactions, while an extra orderer organization orders the result of those transactions into new blocks, fed back to the peers. This process was thoroughly described in Section 3.5.1.

For simplicity, only an orderer is represented here, although in a real network, it is recommended in [Hyperledger, 2022k] to run more than a single orderer, where in this case a consensus algorithm, reviewed in Section 3.5.1, is used to get additional agreement among different organizations. No users are represented in the orderer organization, because orderers' main functions do not have any direct user actions.

This distinction between peers, orderers and their functions is an intrinsic part of the Fabric structure. The main idea of this separation can be replicated in different technologies, by giving the power of executing different smart contracts, access different types of data and validate the network to separate entities.

4.3.3 Component layer

After reviewing the second layer, components of the client application and peer of Organization 1 are presented with additional detail, using the third architecture layer of the C4 model.

Client application container

The application, shown in Figure 4.4, is structured in a similar way to any modern web application and only the Fabric gateway API is part of the Fabric project. Starting off with the *frontend application*, this component can be composed by any technology able to display pages to the end user with data from the network, either by a web browser used in this case, or alternatively with a native application from any operating system.

As the user executes actions in the frontend, HTTP(S) requests go to another component, an *backend application with REST API*, which is responsible from parsing the data correctly and use a *Fabric gateway API* to make the proper requests in gRPC to the network. Note that these calls require infrastructural logic, which the gateway API uses, letting the backend only worry with business logic.

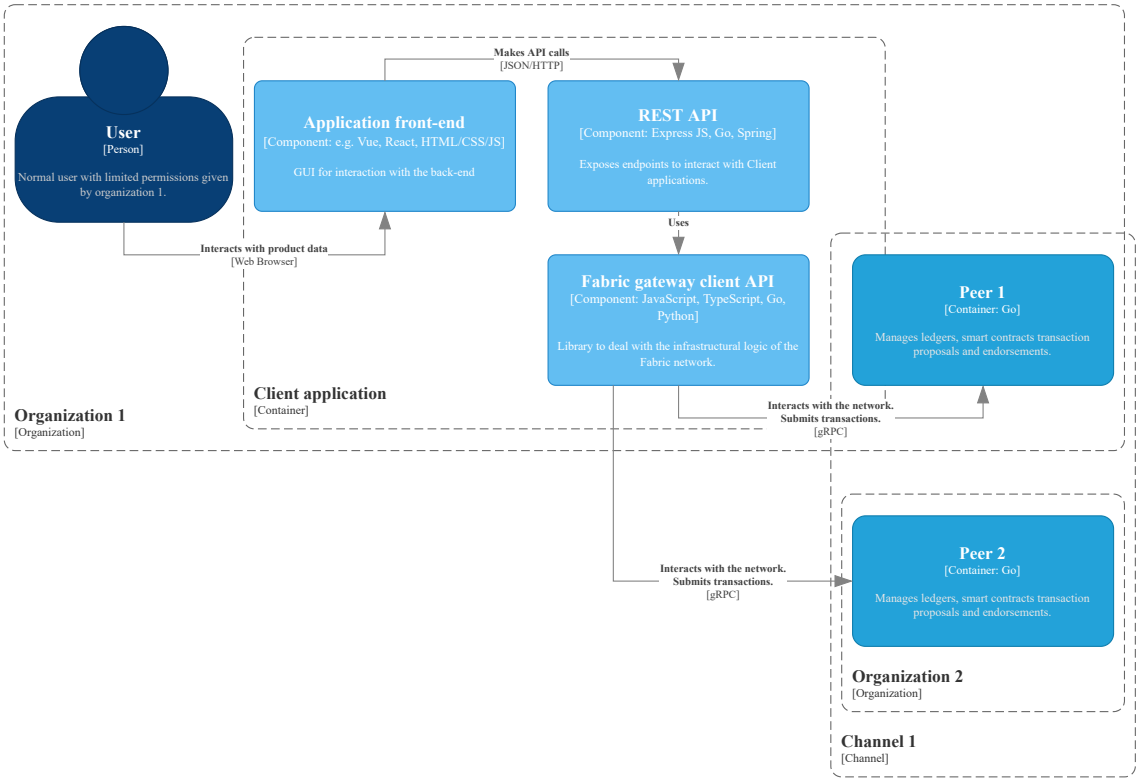


Figure 4.4: Client components diagram of the C4 architecture model.

After the network processes the calls, the response is dealt with the application backend, which responds with a HTTP status code and body, in the case of a web application as stated above.

Peer container

Figure 4.5 shows components that make up a peer in the Hyperledger Fabric network. To simplify, some components used in the peer are not illustated independently, such as the internal gRPC interface and the gossip protocol used to discover peers.

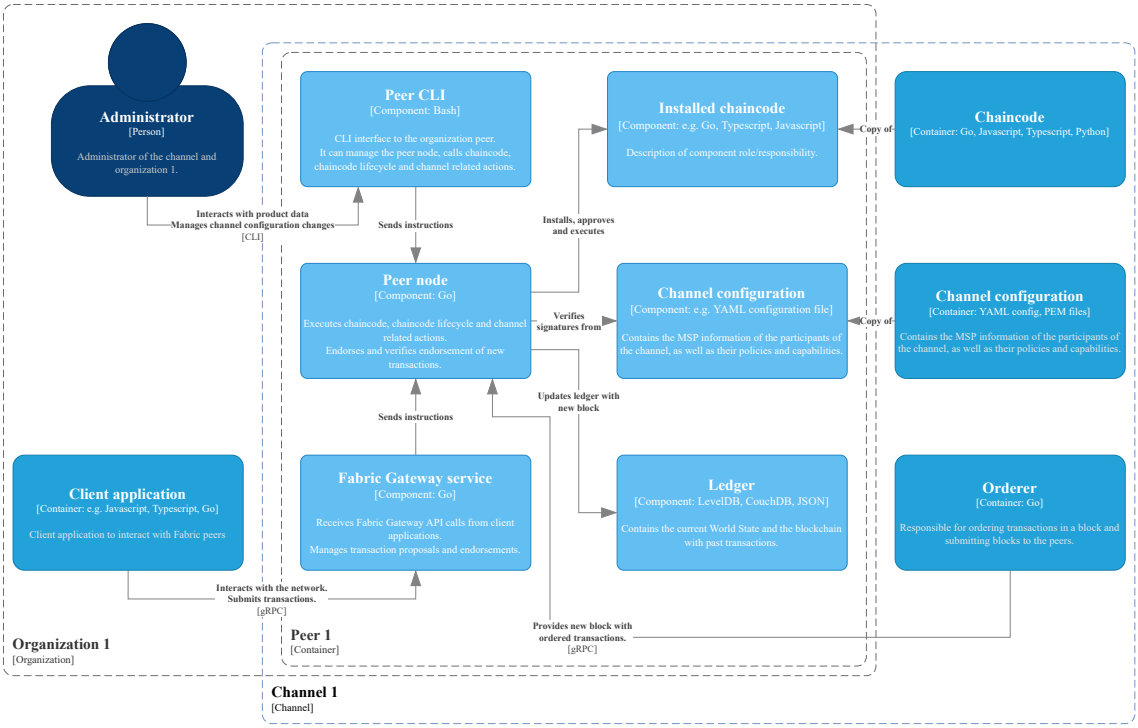


Figure 4.5: Peer components diagram of the C4 architecture model.

Using peer executables, an administrator can interact with the network through a CLI, as mentioned already. These interactions require the *peer node* to be running, which executes the necessary functions, which can be anything from managing the peer, channel or chaincode functions the peer is participating in. Every peer participating in a channel must have a copy of:

- The *channel configuration* which includes the channel MSP and a configuration file stating the capabilities of the rest of the members of the network;
- The chaincode used in every channel, whenever it needs to execute and endorse the transactions he is in.

While the peer node daemon is running, an orderer can also send him messages of new blocks, through gRPC as it is done for all internal communication in the network, and update his ledger and world state, described in Section 3.5.1.

4.4 Summary

This chapter described the process of designing an architecture, based on the C4 model, starting with the listing of relevant quality attributes for a blockchain network, giving relevant scenarios for these attributes and trade-offs, followed by an enumeration of different high level requirements, separated in different domains.

With reference to the projects discussed in the preceding chapter that address some quality attributes, it was determined that these were outside the scope of

this work. Some high level requirements were also set with a lower priority and will not be approached in the next chapter, during the development of a prototype.

By finishing this chapter exploring a possible architecture, it is expected to fulfill its role as a blueprint of a generic platform of product biographies. The objective for the next chapter is to implement a prototype based on this architecture, describing every relevant step taken.

Chapter 5

Implementation of a product biography platform

This chapter shows the process of implementing a prototype of a platform for product biographies, exemplifying the architecture defined in the last chapter. The implementation phase started off by identifying the needed software and dependencies to run, followed by a summarization of the initial attempt of setting up a Sawtooth network, and upon trial and error, follow through with a Fabric implementation instead. Some key artifacts were developed along with the network, such as:

- A base data model to define a product;
- The smart contract around this product definition to be used in the Fabric network;
- A REST API to connect to the Fabric network and simplify the infrastructure logic of the implementation, defined as the backend of a client application;
- A simple web page to interact with the REST API and show results to the end user, defined as the frontend of a client application.

The goal of this implementation is to achieve the *Must have* and *Should have* points defined in Section 4.2, mentioning every high level requirement fulfilled throughout the development, and show off the possibilities of this network.

5.1 Prerequisites

Before starting with the implementation of the project, some software dependencies must be installed. In this section, every dependency for every developed artifact is listed.

To test the Hyperledger Sawtooth test validator, only Docker, docker-compose are required, as the inner dependencies of Sawtooth are not documented but instead just added to a docker-compose YAML configuration file. Ubuntu 18 is also required to run a Sawtooth node natively.

The final implementation of the **Hyperledger Fabric network**, has the following requirements, without any specific operating system required:

- Fabric binaries and docker images of the test network, using the latest version `v2.4.6`;
- `docker` and `docker-compose`;
- `jq`, a Command-Line Interface (CLI) JSON processor;
- `logspout`, a log router for Docker.

Fabric provides four different language options for the Chaincode implementation. **JavaScript (JS)** was chosen for this thesis because of the familiarity with the language and past experience. To use this language, the following requirements are needed:

- `node`, a JS runtime environment, using the latest version, `v19.0.1`;
- `npm`, a package manager that comes bundled with Node, using the latest version, `8.19.2`.

The dependencies of the **Node chaincode** project itself are:

- `fabric-contract-api`, providing a contract interface for the JS code, using the latest version, `v2.2.3`;
- `json-stringify-deterministic` and `sort-keys-recursive`, to make sure the JSON data of products is already read and written in a deterministic way, using the latest versions `v1.0.8` and `v2.1.8`, respectively.

Despite the `fabric-contract-api` dependency requiring a `v12.16.1` version of `node` and `npm` with version `6.4.1`, the smart contract showed no problem running with the versions stated above. For the client application and as mentioned in the introduction, two projects were created. The **backend application, involving the REST API**, uses the following dependencies:

- `express`, a JS web application framework. using the latest stable version `v4.18.2`;
- `fabric-gateway`, providing the API to the Fabric network, using the latest version `v1.1.2`;

- `uuid`, to generate unique product IDs, using the latest version `9.0.0`;
- Other dependencies used for the API documentation and HTTP request handling: `swagger-jsdoc` in `v6.2.7`, `swagger-ui-express` in `v4.6.0`, `cors` in `v2.8.5` and `body-parser` in `1.20.1`.

And finally, for the **frontend of the client application**, the following dependencies were used:

- `vue`, a JS framework for user interfaces;
- `vuetify`, a `vue` framework for UI components.

Other dependencies were used during development, such as a linter for the JS projects and a build tool used with the frontend application, but were not listed as they have no impact to the result of the implementation. The choices were made around JS frameworks, Express and Vue, again, due to the familiarity of it with past projects. The client application's sole purposes is to demonstrate the network and chaincode's capabilities with external requests, so the technology selection here has little impact on the project as a whole.

Following the prerequisites summary, the next chapter shows the attempt taken in implementing a Hyperledger Sawtooth network.

5.2 Setting up a Sawtooth network

An initial implementation using Hyperledger Sawtooth was attempted, since it had more desirable features that could help for a product biography platform and the overall architecture would be simpler to apply, for example by using only one type of validator, opposed to what happens in Hyperledger Fabric.

Despite using the latest version `v1.2`, the project showed a lack of maintenance and stability of dependencies during the installation and execution of a Sawtooth node, documented in [Hyperledger, 2022m]. Windows and Linux-based operating systems were used.

The first attempt was running a test validator, available from a `docker-compose` file in the Sawtooth documentation. The docker-compose worked as expected, creating the necessary images and running containers for the REST API and the validator itself. Problems started occuring when, in order to run the necessary software to perform transactions, unmaintained packages had to be used and without it, not even the test validator would work. After further trial to use this software, more dependencies had to be downgraded to match the packages' versions and, after a cost-benefit and risk analysis, the decision to opt for the implementation of a blockchain network using more popular options, such as the Hyperledger Fabric project, was taken.

As a preemptive measure, it was previously stated in Section 2.3, that in case a Sawtooth implement would become unsuccessful, a fallback solution would be to implement a new solution using the Fabric framework, reviewed in Section 3.5.1, followed by a new architecture from the last semester as well.

The following Section will depict the development taken, in a successful way, using Hyperledger Fabric as a backend for a blockchain network.

5.3 Setting up a Fabric network

After the trial and error of setting up of a Sawtooth network, an implementation for a Hyperledger Fabric network was attempted. The Fabric project has different public repositories available for the different components in open development, including one called `fabric-samples`, which has implementation samples of several Fabric artifacts, such as a test network, a Fabric CA server and different types of chaincode and organization examples.

These samples were pulled from the repository, using the latest version of Fabric, `v2.4.6` at the time of the elaboration of this thesis, along with binaries used to run the peer related functions. A group of scripts are present in the samples to simplify different actions, such as:

- `network.sh`, by setting up a sample network, with options to create a channel which acts as our blockchain network, to set up a CA server ran by different organizations, and to select a different world state database, such as CouchDB;
- `monitordocker.sh`, monitors every actor of the test network, more detail in the next chapter;
- Other scripts called by `network.sh` also exist, for example `deployCC.sh` to deploy a chaincode in the test network or `createChannel.sh` to create a channel and anchor it to the available peers.

To begin, after the prerequisites were installed, the command

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

was run in the *test-network* folder of the fabric-samples, to generate the following running docker containers:

- `peer0.org1.example.com` and `ca_org1`, representing a peer node and a CA server owned by Organization 1, matching the *Peer node* component in "Peer container" of Section 4.3.3 and the *Fabric CA server* container in "Organization system" of Section 4.3.2;

- `peer0.org2.example.com` and `ca_org2`, representing a peer node and a CA server owned by Organization 2;
- `orderer.example.com` and `ca_orderer`, representing an orderer node and a CA server owned by the Orderer Organization;
- `cli`, corresponding to the *Fabric CLI* container in the architecture layer 2 in Section 4.3.2, has the scripts above exposed to interact with the Fabric network.

From here onwards, the `fabric-samples` scripts may be ran locally from the pulled repository folder or in the `cli` container.

To interact with a specific peer, since both are running on the same machine, environment variables are used to refer to the peer with want, along with their Membership Service Provider (MSP) data. To do this, these variables can be set manually or with the help of another script present in the Fabric samples, `setOrgEnv.sh`. The variables used by peer functions are, but not limited to:

- `PEER_TLS_ENABLED`, may be true or false to enable Transport Layer Security (TLS) communication;
- `PEER_LOCALMSPID`, referring the MSP organization ID of the peer;
- `PEER_TLS_ROOTCERT_FILE`, is the path to the certificate used for TLS communication;
- `PEER_MSPCONFIGPATH`, pointing to the MSP folder structure;
- `PEER_ADDRESS`, is the address and port where the peer is connected.

At this point, a basic network is ready to be interacted with, but before starting the implementation of the smart contract and client application, we will take a look into how both organizations and channel are configured.

5.3.1 Organizations configuration

As mentioned in the literature review, in Section 3.5.1, organizations are identified through MSPs and their identities composed by CA certificates. To have a working organization, the following structured must be followed by itself and every member:

```
msp
├── cacerts
│   └── cert.pem
├── intermediatecerts
├── keystore
│   └── signer.key
└── signcerts
```

```
    └── signer.pem
├── tlscerts
    └── tls-ca-cert.pem
├── tlsintermediatecacerts
└── config.yaml
```

Each members has access to a private and public key pair, `keystore/signer.key` and `signcerts/signer.pem` respectively, to sign and verify signatures of data, such as the result of transactions. As for the certificates, the `cacerts` folder contains the certificates of the root Certificate Authority (CA) trusted by the organization. Every member of the organization must have a certificate derived from this one, directly or through an intermediate certificate, used for example to separate different Organizational Units (OUs). In this case, these certificates belong in the `intermediatecerts` folder. For secure communication, it is the `tlscerts` certificates that are used for TLS, and the `tlsintermediatecacerts` if needed, by the same logic of `intermediatecerts`. The list of MSP folders goes on, mentioned in [Hyperledger, 2022h], but they are unnecessary for the implementation of this network.

In the case of the test network, there is a MSP for each one of the organizations and for both members: users and administrators. The next section shows how these MSPs are referred to and given permissions in a channel.

5.3.2 Channel configuration

A channel MSP, in contrast to a local MSP shown above in detail, is not a set of folders but part of a YAML configuration file, as mentioned in the Channel part of the Hyperledger Fabric review in Section 3.5.1.

This configuration file has essential attributes and capabilities, defining both policies between organizations and version requirements:

- `Organizations` has a list of all organizations included in the channel and all the information needed about them, which includes their MSP path, ID, name, IP addresses of endpoints and `Policies`;
- `Capabilities` define the mininum version requirements for orderers, peers and both, in order to run in a compatible manner;
- `Application`, `Orderer` and `Channel` attributes are linked to the policies and other functional configuration settings for peers, orderers and channels respectively.

A full list of a sample configuration file is available in [Hyperledger, 2022c]. Additionally, this means that the channel itself lacks a public/private key pair and its own certificates.

This configuration, in addition to the blockchain, must be kept in consensus in every node and, if needed, updated accordingly.

5.4 Product biography smart contract

The goal in this section is to develop a working smart contract to give fundamental requirements for a product biography, mentioned in Section 4.2, To start, the data model representing a product must be defined, in order to properly store its data. Then, the functions interacting with this data model are described, in order to properly interact with the product biography.

5.4.1 Data model definition

As mentioned in the 4.2.1 Section, to have the least restrictions on the data model, we consider only three data objects in the smart contract: *product*, *event* and *transformation*. These objects are the basic foundation of a biography, portraying what happens to a product and how it is used. The following JSON structure represents a product with a single event and transformation, based on a textile product, mentioned in Section 1.2:

```
{
    "id": "1234-5678-90ab-cdef",
    "name": "Cotton_1",
    "events": [
        {
            "description": "Creation",
            "location": {
                "name": "Coimbra",
                "latitude": 40.20564,
                "longitude": -8.41955
            },
            "changes": {
                "color": "White",
                "quantity": 100,
                "unit": "m3"
            }
        }
    ],
    "transformations": [
        {
            "description": "Cloth production",
            "location": {
                "name": "Porto",
                "latitude": 41.14961,
                "longitude": -8.61099
            },
            productsId: [
                "cdef-5678-1234-90ab"
            ]
```

```
            }
        ],
        "color": "White",
        "quantity": 100,
        "unit": "m3"
    }
```

A **product** is composed of a string `id`, a string `name`, `events` and `transformations` arrays and, optionally, `quantity` and `unit` to represent a group or a batch of products, and any other attribute to represent a product's inherent characteristics, such as the `color` in this example.

An **event** is used to describe any procedural change in the product, starting from the very beginning with a harvesting process, for instance. It is composed of a `description` string of the event and a `location`, composed of a name string and its decimal coordinates. Any attribute can be changed during an event, except for the id of the product and the name, if the event is the first one of the product. The changes of each event get stored in a `changes` object in the event, with the attributes to change and their new values.

Finally, a **transformation** represents an instance where a product originated new ones, through a manufactoring process for example. Each transformation object contains a `description`, a `productsId` array of the IDs of the new created products and a `location` structured the same way as in the event.

In the next section, the functions and deployment of the chaincode will be shown, making use of this data model.

5.4.2 Smart contract functions and deployment

Hyperledger Fabric offers four complete language options for chaincode development: Go, Java, JavaScript and TypeScript. As mentioned in the prerequisites section, JS is used and in this language, the smart contract is a subclass of the `Contract` class, given by the `fabric-contract-api`, called `ProductBiography`. Every function exposed in this class may be invoked upon deployment. The functions exposed to implement a product biography are the following:

- `CreateEmptyProduct` creates an empty product, with only an `id` and `name` and no genesis event;
- `CreateProduct` creates an product, with an `id`, `name` and a genesis `event` that may contain information about the creation of this product, along with different attributes;
- `UpdateProduct` updates a product, taking an `event` and changing its attributes if needed;
- `TransformProduct` updates a product with a `transformation`, creating new products with an initial `event`, also given as a parameter;

- `DeleteProduct` deletes a product, given its `id`;
- `GetProduct` returns a product from a given `id`;
- `ProductExists` receives an `id` and checks if a product exists or not;
- `GetAllProducts` returns all existing products. This function uses an internal `getStateByRange` method to query all products, although in a production environment, it may be advised to set an adequate range to not overload the network with long queries through the blockchain.

And to exemplify the possibility of having private data, equivalent functions of `ProductExists`, `CreateProduct`, `GetProduct` and `GetAllProducts` were used. Finally, interal functions also exist to verify if the objects follow the constraints mentioned in the previous section.

Every JSON structure being outputted by the smart contract have its attributes alphabetically ordered and deterministically structured, to avoid conflicts in results between different peers and ensure more consistency of data. The ID string must be generated outside of the Fabric Network or in a deterministic way to ensure agreement between different peers.

In order to install a chaincode into the network, the sequence of steps below must be followed. The Fabric binary `peer lifecycle` contains the necessary function to follow these steps. Note that in the test work, a `./network.sh deployCC` script may also be used to automatically install it.

1. First, the smart contract has to get **packaged** properly, to be installed in the peers;
2. The chaincode package emust be **installed** in every organization that is required by the endorsement policy of the chaincode, in order to properly execute and endorse transactions. Since the endorsement policy is a default one, the majority of the peers must validate the transaction, therefore it must be installed on both peers;
3. Upon installing the chaincode, the peers must **approve** the definition of the installed package. In this case, the approval is not based on the validating peers required by the chaincode, but the `LifecycleEndorsement` rule in the channel configuration file. The package ID is the combination of the chaincode label and a hash of the chaincode binaries;
4. When the necessary conditions of the `LifecycleEndorsement` are met, or in the case of this network, upon approval of the majority of organizations, one of them can **commit** the chaincode definition into the channel.

Finally, the developed chaincode is ready to be invoked, and if required, to be properly initalized. A special `Contract` function `Init` may be required to run for a proper chaincode initialization, as well as to specify an endorsement policy,

that is, who must endorse the transactions for its results to become valid. Both of these settings are applied during the approval phase of the chaincode.

In order to upgrade the chaincode to newer versions, the process is similar, except upon approving a chaincode definition and committing it to the channel, the name should remain the same and the sequence number properly incremented, using the `--name` and `--sequence` flags.

Two different chaincode transactions will happen while running the prototype: ledger *queries*, where the ledger is only read by a chaincode function. This chaincode function is executed by a handler for JS code and only endorsed by a single peer, since no transaction is happening. Ledger *updates*, as opposed to queries, need the execution of the handler and the endorsement of the majority of the peers, as well as the validation of the result.

To interact with the Fabric network and this smart contract outside of the network, the implementation of a client application was described in the following section.

5.5 Client application example

This section covers the implementation process of the client application. The objective of this application, which is external to the network, is for it to interact with the network and carry out actions related to a product biography via the smart contract created in the previous section. An end user will be able to insert data in a familiar way, using the frontend page, whereas any developer may use the web application interface, that is, the REST API, to perform functions through HTTP requests.

5.5.1 Backend application

The backend application is one of two parts of the client web application and is responsible to expose a REST API, and in the same time, interact with Fabric peers through a Fabric Gateway API. The goal is to create a frontend interface that is simpler to interact with and to show how any other application can communicate with the network from the outside.

Two ways of authentication could be implemented here, the first is to manage its own authentication service, and use a specific role of an authorized organization to interact with the network, while the second could just allow the end users use their own certificates. In a real world scenario, the first option makes more sense in order to also manage other attributes, such as the bandwidth usage of every user in an exposed API. For this implementation, the two peers have their certificates in the backend folder structure, to be able to show the data difference in private insertions of data.

As for the REST API, the following endpoints with request methods and response

codes are exposed, respecting the guidelines of [Mulloy, 2013] and with `/api` as the root:

- `GET /info`, shows a page with the API information generated using Swagger. Returns `200`;
- `GET /params`, shows the current parameters used to access the network, which may be from Organization 1 or 2. Returns `200`;
- `PUT /login/org1` and `/login/org2`, to simulate a login action and change organizations. Returns `200`;
- `GET /products`, returns a list of all registered products in the network, using the . Note that in a real case scenario, this request should use additional parameters to not overload the network by pulling *every* historical product; Returns `200`;
- `POST /product`, creates a product from an `id`, `name` and, optionally, an `event` from the request body. Returns `201`, `400` or `500`;
- `GET /product/:id`, reads the product with the given `id`. Returns `404` if the product is not found or `200`;
- `PUT /product/:id`, updates the product with an event in the request body. Returns `200`, `400` or `500`;
- `POST /product/transform`, updates a product with a transformation, from a given `id`, and with an `event` and a `transformation` in the request body. Returns `201`, `400` or `500`.

As shown by the response codes, the API properly sanitizes the network output, catching potential client and server-side errors. All responses related to the product biography are structured in JSON objects, such as the structure described in Section 5.4.1.

Despite the REST API being easily accessed through services like Postman or programs such as `curl`, a frontend application was implemented in the next section for easier visualization, interaction for the end user, and as a way to confirm this possibility.

5.5.2 Frontend interface

The frontend part of this implementation is a single web page, composed of reactive components of `Vuetify` to display the contents of a product and its events and transformations.

Five actions are possible in this web page:

- Select the organization to view the page from. This is done to exemplify how private product data is implemented;

- Select a product from a list of existing products, empty products and private products. This shows a timeline with all the events and transformations in the product. Empty products are products that do not contain any event or attributes and private products are products that are only available within the current organization;
- Create a product from scratch;
- Add an event to a selected product;
- Add a transformation to a selected product, by also giving the names of new products.

Each action involves a HTTP request to the backend, which will respond with the desirable product data; parsing the data and properly render the page with the updated data.

The end result is shown in the next section, along with the interaction with all the other implemented components.

5.6 Results

After describing the implementation process of each developed artifact, this section displays the end result of the project, as well as how every component interact with each other. To start up every component, a `start.sh` script was developed, doing the following:

- Initializes the network, writing certficates from the CA root certificate, creating the network and its peers and orderer;
- Deploys the `productbiography` chaincode to the network, by following the steps of Section 5.4.2;
- Updates the MSPs, along with the cryptographic material generated during the network initialization, in the backend project and gives user permissions to use the private/public key pairs;
- Starts the two parts of the client application;
- Runs a grafana + prometheus container, to inspect the Fabric network activity. Its use is covered in the next chapter.

By accessing the frontend link through a browser, exposed locally in port `8080`, the home page shown in Figure 5.1 appears, showing four buttons for different actions and an empty product list.

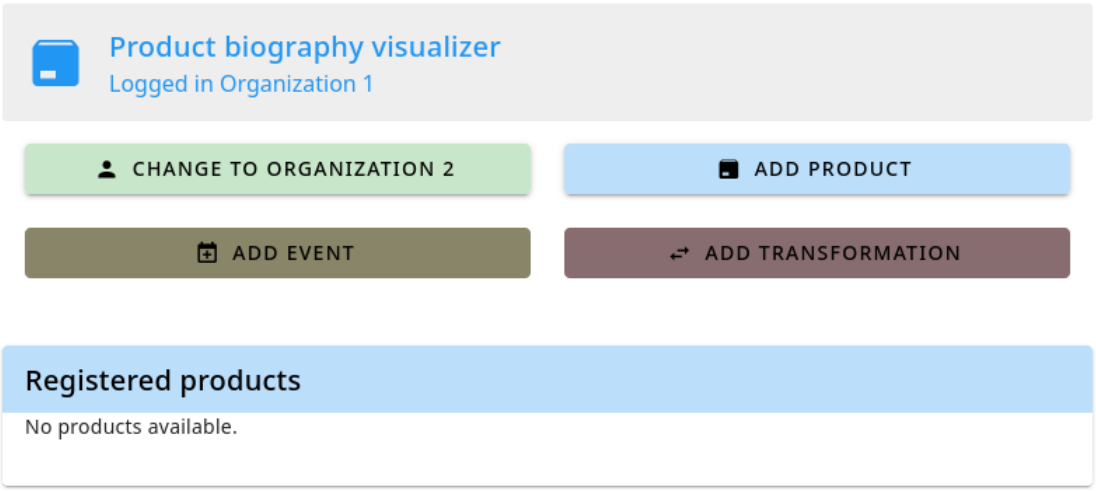


Figure 5.1: Home page of the frontend application.

Every time the page is loaded or a product updated, HTTP requests to the backend are made, which can be seen by inspecting the browser's *Network Monitor*, or by the logs from the backend itself:

```
[api] GET /api/products called.
[api] GET /api/private/products called.
[services/fabric] Calling the GetAllProducts function of
productbiography.
[services/fabric] Calling the GetAllPrivateProducts function of
productbiography.
[services/fabric] GetAllProducts success.
[services/fabric] GetAllPrivateProducts success.
```

The "Change to Organization x" button changes the running organization in the backend, from Organization 1 to Organization 2 and vice versa. In a real world scenario, this change would be made via a login page to ensure proper authorization of the end user. By clicking the button, the `PUT /api/login/org1` or `PUT /api/login/org2` requests are made, and in success, a notification is shown for the recent change made.

Initially, only two action buttons are enabled, since to add events or transformation, at least one product must exist to be selected. To proceed with the product creation, after clicking the "Add product" button, the form in Figure 5.2 appears, in order to create a product with the following data, also displayed there. Note that a quantity checkbox is also included, if a batch of products has to be described, instead of a single one. The frontend page also allows the user to create empty products, without events and to make these products private, meaning it may only be seen by the current organization.

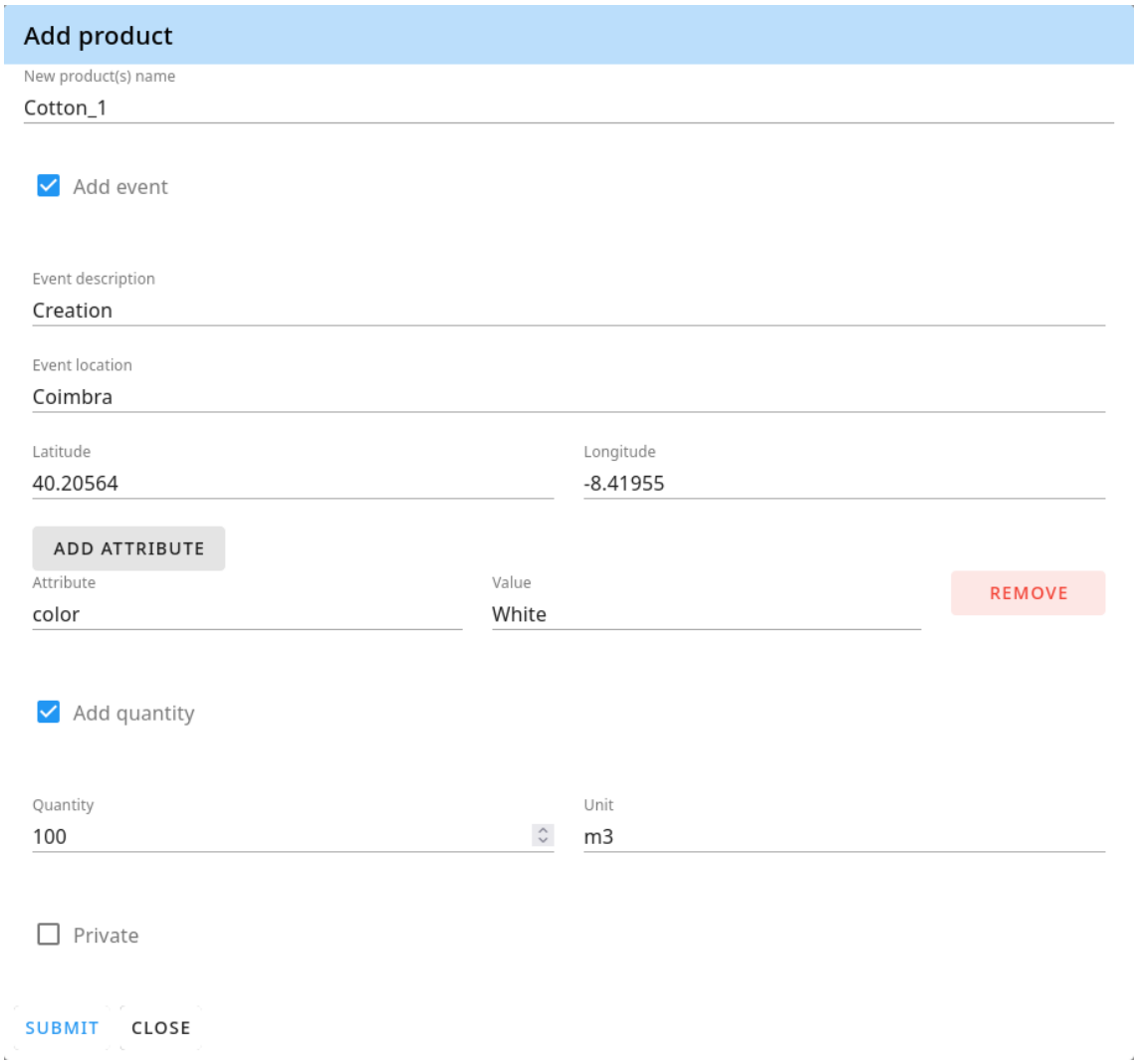


Figure 5.2: Product creation form, with event data.

By submitting this form, the endpoint `POST /api/product` is called, which the backend responds by submitting the transaction to the Fabric network, following the process described in the previous Section 5.4.2.

Upon success, the list of products is queried again and the new product will show up. In Figure 5.3, the product is selected, showing a timeline of events and transformations, containing just a single genesis event for now. This event has a timestamp, which contains the date of creation of the product and an array of changes, which are the attributes added at the time.

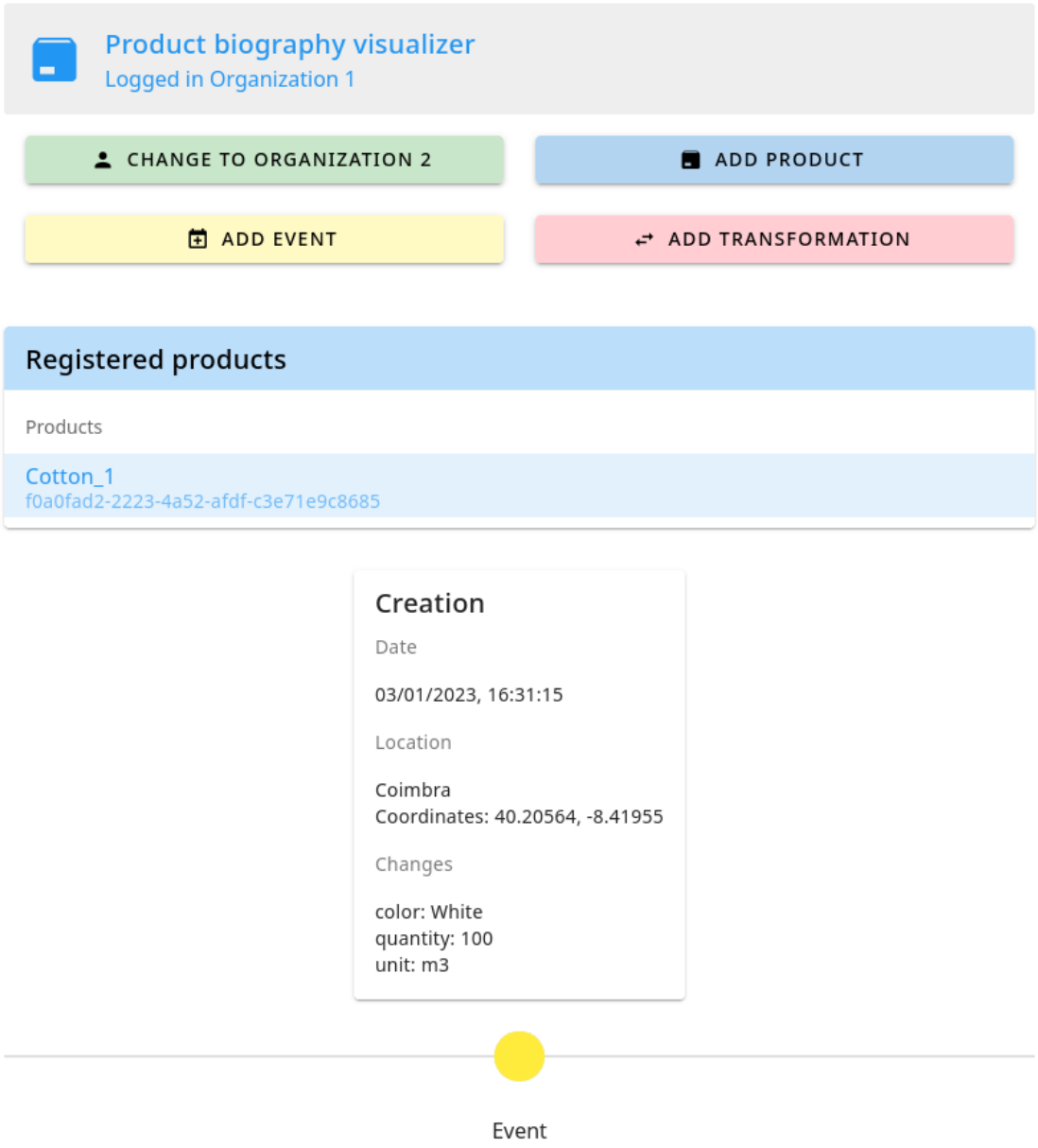


Figure 5.3: Selected product, with a unique event.

To add an event and update the product, the "Add event" button will show another form, similar to the form used to create the genesis event in 5.2. Submitting an event will make a call to the `POST /api/product/:id` endpoint, and in success, append a new event to the `events` array of the product.

Finally, the "Add transformation" shows a different form, shown in Figure 5.4, to submit a transformation of a product to new ones. This action will not delete the selected product, but update it with the transformation, while automatically creating new products with attributes from a genesis event, derived of this transformation.

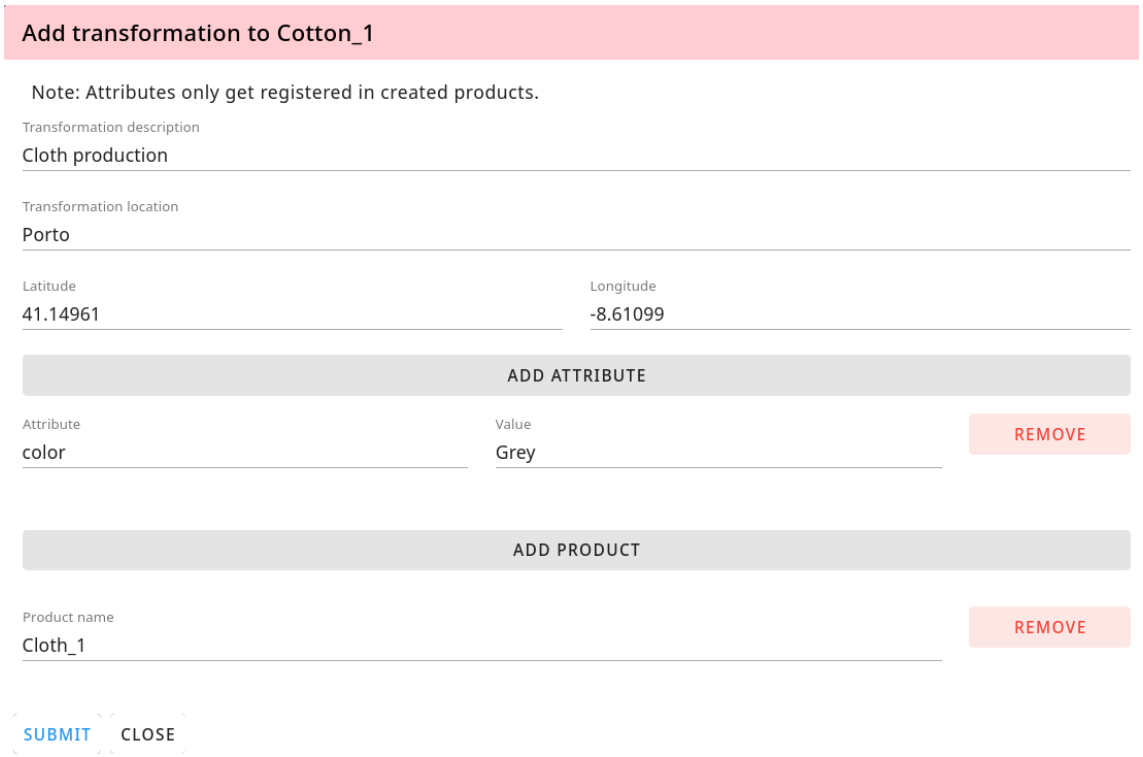


Figure 5.4: Product transformation form, with event data and a new product.

Submitting this form makes a request to `POST /get/product/transformation`, appending the transformation to the current product and creating new products, with a genesis event composed of the same data. This final result can be seen in Figure 5.5, with the selected product having two events and a transformation, in chronological order. For optimization, the transformation only contains the ID of the new product that appears in the updated list.

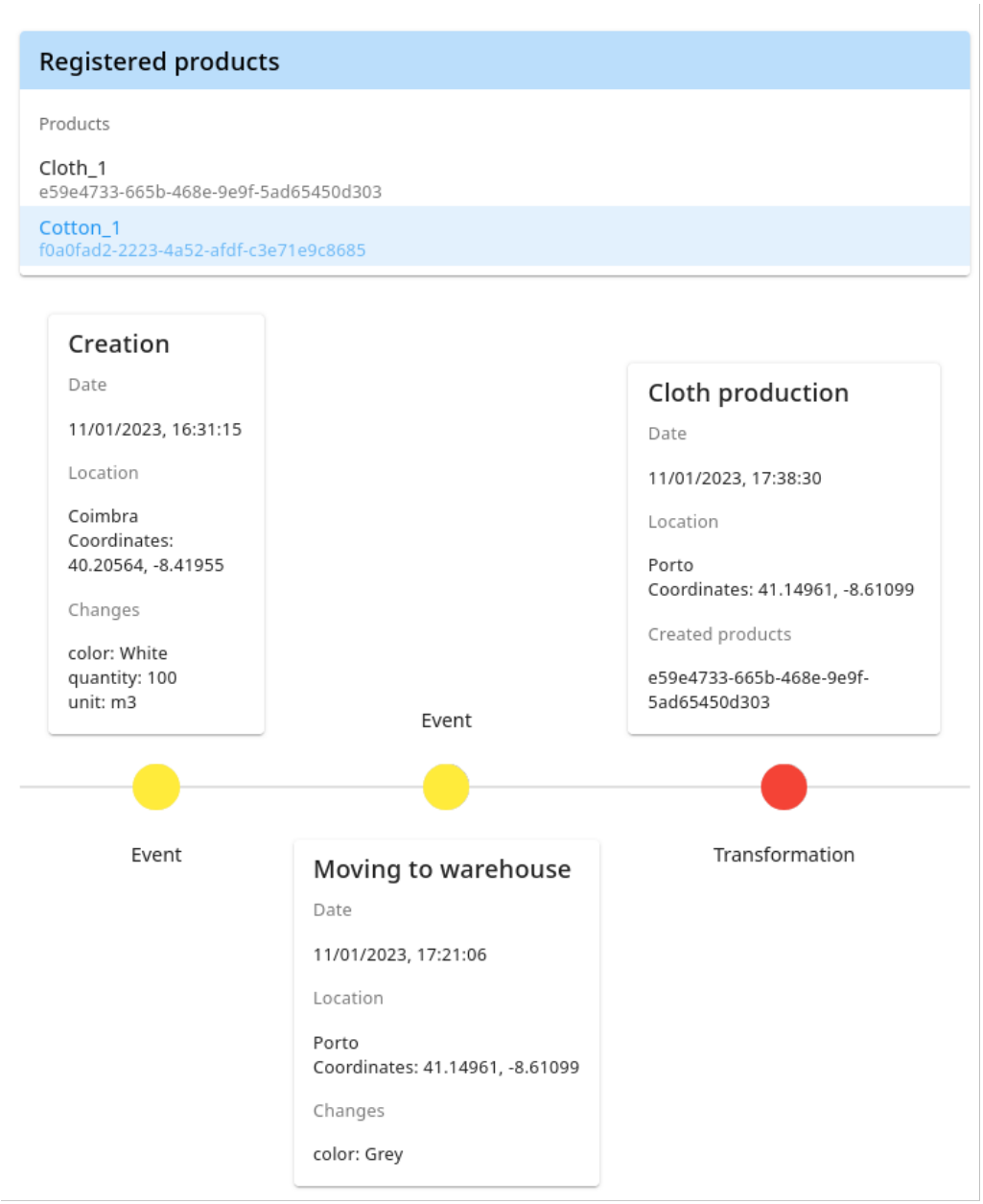
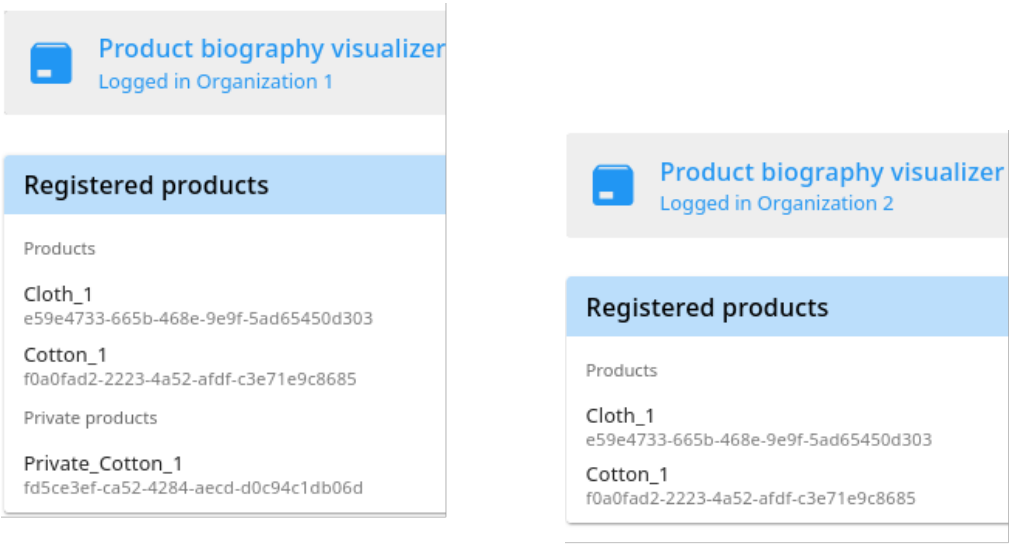


Figure 5.5: Product final timeline, showing two events and a transformation with the ID of a new product.

Private products may also be submitted with the smart contract. To do this, a "Private" checkbox can be checked in the form of Figure 5.2, submitting the product privately. In this case, the state of the product in only maintained by the organizations allowed to read the product data. As an example, a new product, `Private_Cotton_1` is inserted privately in Organization 1, showing in the list of Figure 5.6a, but absent of Figure 5.6b, showing the Organization 2 list of products.

Figure 5.6: Different listing of private products from both organizations.



(a) Organization 1.

(b) Organization 2.

After this implementation review, a few final points must be made. The frontend part of the client application currently does not use some defined endpoints, such as `GET /api/product/:id`, because all the information about the product is already given by previously called endpoints. Also, some smart contract functions ended up not being used at all by the client, such as `DeleteProduct`, because it does not currently have a use in the client application, since blockchain data is immutable, but may be used to clear up space in the world state of the ledger.

5.7 Summary

In conclusion, this chapter was able to display the implementation of a platform of product biographies, with main functions such as the insertion, updates and relating new products with existing ones. Starting off with a failed attempt running a Hyperledger Sawtooth network, four artifacts were developed, each with a unique and essential function to a usable platform, using Hyperledger Fabric. Also, the *Must have* and *Should have* high level requirements were thoroughly respected, and with further development, the rest of the optional requirements may be implemented in a future iteration of this project.

This shows the possibilities of describing real world data into an immutable log, which may be controlled by different entities, and with some effort, achieve a respectable level of decentralization. With this available prototype, the following chapter will review additional implementation details, in order to validate the architecture of this thesis, by simulating certain experiments; and also display how the artifacts may be tested for debugging and further development iterations.

Chapter 6

Validation and testing

After describing the implementation process and results, this chapter's purpose is to use the developed prototype to validate the Quality Attribute Scenario (QAS) of the architecture elaborated in Chapter 4 and show how further testing can be done, reviewing what existing tools may be applied to aid smart contract development and tools such as Prometheus and the different `peer` binaries for blockchain and network monitoring.

6.1 Validation of the architecture

The validation of the architecture was done by simulation of some parts of the developed prototype, considering the defined scenarios of Section 4.1, and using different tools, described further, to test the implementation and prove that these scenarios are effectively applied. From here on, each quality attribute scenario is going to be referred by its ID.

6.1.1 Network security

In order to respect the quality attribute scenario **S1**, the Hyperledger Fabric network uses Transport Layer Security (TLS) to encrypt data between the nodes. To verify this, the network may be inspected using tools such as `Wireshark` [Foundation, 2022], an open-source network and packet analyser, used to examine and dissect packets sent in an active network. In this case, we want to verify that TLS communication is effectively applied during a ledger query and ledger update, using the `GetProduct` and `CreateProduct` functions of the chaincode, mentioned in Section 5.4.2.

To identify what packets must be analysed, it is essential to know which IP addresses are involved. Since the test network is running in `docker` containers, we need to identify the subnet used by our peers, as well as their IP addresses, using the following commands:

```
# Find the fabric_test network ID, used in the docker-compose file
$ docker network ls | grep fabric_test

# Obtain the subnet used by this docker network
$ docker network inspect -f \
'{{range .IPAM.Config}}{{.Subnet}}{{end}}' \
<docker_network_id>
172.20.0.0/16

# List the containers to get the peers' container ID
$ docker ps

# Retrieve the IP address used
$ docker network inspect -f \
    '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' \
    <container_id>
172.20.0.8
```

The identified subnet is `172.20.0.0/16` and, by repeating the last command for each container, the obtained list of IP addresses is:

- `172.20.0.8` for the Organization 1 peer;
- `172.20.0.7` for the Organization 2 peer;
- `172.20.0.9` for the orderer.

After obtaining this information, it is now possible to properly detect what packages are sent between these addresses. The next step is analyse what packets are sent, by reading any interface available in the device, applying a filter to only read messages from the subnet and invoking a ledger query with the `GetProduct` function. Part of the result is showed in Figure 6.1, where after a successful TCP connection is made, a TLS handshake occurs, initiating the encrypted communication.

Source	Destination	Protocol	Length	Info
172.20.0.8	172.20.0.7	TLSv1.3	347	Client Hello
172.20.0.7	172.20.0.8	TLSv1.3	14…	Server Hello, Change Cipher Spec, App
172.20.0.8	172.20.0.7	TLSv1.3	998	Change Cipher Spec, Application Data,
172.20.0.8	172.20.0.7	TLSv1.3	114	Application Data
172.20.0.8	172.20.0.7	TLSv1.3	99	Application Data
172.20.0.7	172.20.0.8	TLSv1.3	105	Application Data
172.20.0.7	172.20.0.8	TLSv1.3	99	Application Data
172.20.0.8	172.20.0.7	TLSv1.3	99	Application Data
172.20.0.8	172.20.0.7	TLSv1.3	107	Application Data
172.20.0.7	172.20.0.8	TLSv1.3	107	Application Data
172.20.0.8	172.20.0.7	TLSv1.3	107	Application Data
172.20.0.7	172.20.0.8	TLSv1.3	107	Application Data
172.20.0.8	172.20.0.7	TLSv1.3	107	Application Data
172.20.0.7	172.20.0.8	TLSv1.3	107	Application Data
172.20.0.8	172.20.0.7	TLSv1.3	107	Application Data

Figure 6.1: Snapshot of network communication during a ledger query.

In the case of a ledger query, this exchange only occurs between the two different peers, because the orderer is not involved in reading the blockchain, as opposed to a ledger update, showed in Figure 6.2, where the Organization 1 peer does a TLS handshake with the orderer, before sending data.

Source	Destination	Protocol	Length	Info
172.20.0.8	172.20.0.9	TLSv1.3	344	Client Hello
172.20.0.9	172.20.0.8	TLSv1.3	15…	Server Hello, Change Cipher Spec, App
172.20.0.8	172.20.0.9	TLSv1.3	999	Change Cipher Spec, Application Data,
172.20.0.8	172.20.0.9	TLSv1.3	114	Application Data
172.20.0.8	172.20.0.9	TLSv1.3	99	Application Data
172.20.0.9	172.20.0.8	TLSv1.3	105	Application Data
172.20.0.8	172.20.0.9	TLSv1.3	99	Application Data
172.20.0.9	172.20.0.8	TLSv1.3	99	Application Data
172.20.0.8	172.20.0.9	TLSv1.3	48…	Application Data
172.20.0.8	172.20.0.9	TLSv1.3	389	Application Data
172.20.0.9	172.20.0.8	TLSv1.3	120	Application Data
172.20.0.8	172.20.0.9	TLSv1.3	107	Application Data
172.20.0.9	172.20.0.8	TLSv1.3	130	Application Data
172.20.0.8	172.20.0.9	TLSv1.3	120	Application Data
172.20.0.9	172.20.0.8	TLSv1.3	107	Application Data

Figure 6.2: Snapshot of network communication during a ledger update.

By observing the data sent in packets, for example in Figure 6.3, Wireshark identifies the payload as over 6KB of encrypted data sent by the orderer with the IP address `172.20.0.9`, and the encrypted gRPC message is unreadable. We can deduce that this bigger packet is a new block of data to be appended in the blockchain by the peers in `172.20.0.7` and `172.20.0.8`, but there is no way to tell what is inside.

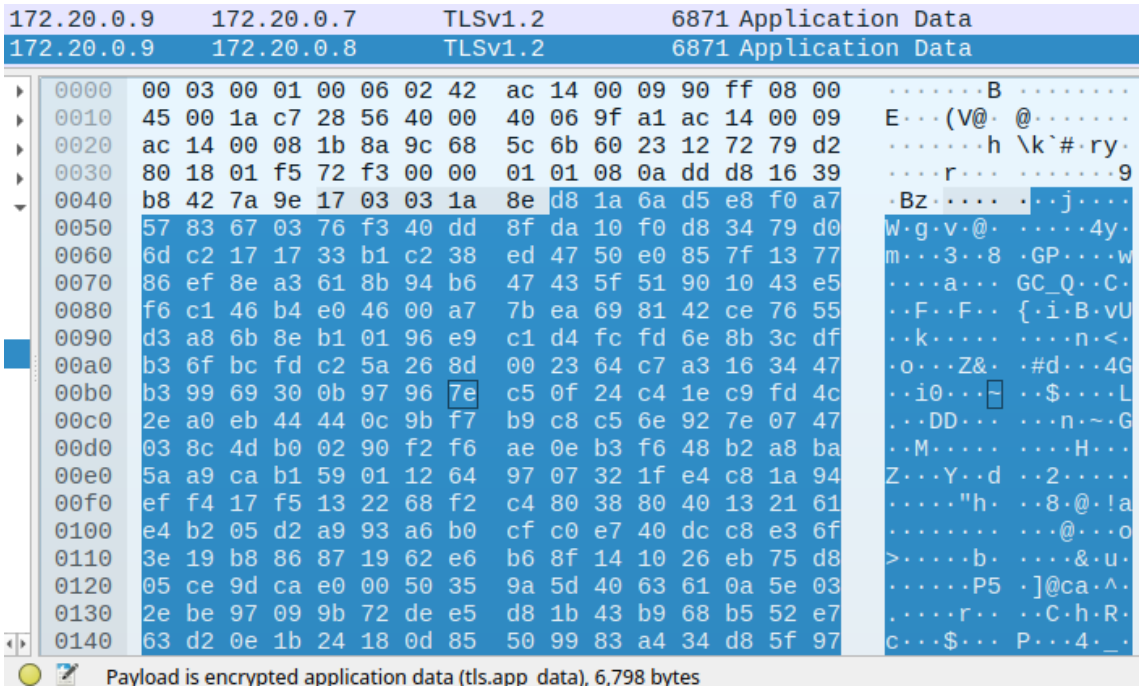


Figure 6.3: Encrypted message of an orderer.

By this network analysis, we can validate that the network is secure from man-

in-the-middle attacks, although the ultimate security of the network is always dependent on the security of the secret keys used by the nodes.

6.1.2 Network decentralization

Although being especially hard to achieve in a permissioned network, decentralization is one of most valued attributes of a blockchain network.

To fulfill the quality attribute scenario **S2**, the Hyperledger Fabric network enables each channel and smart contract to be controlled by different entities, forcing a consensus between different organizations. This is done in the channel configuration, documented in [Hyperledger, 2022d], which explains every point of the configuration as well as the process of updating it.

In summary, three groupings of configuration exist in a channel:

- `Channel`, includes the overall channel configuration and the policies on that level;
- `Application`, which has the configuration and policies of peers;
- `Orderer`, includes the configuration and policies needed for orderers to operate.

Each one of these groupings have the following `policies` subgroups: `Readers`, `Writers`, `Admins`, and exceptionnally, `BlockValidation` for the `Orderer group`. In this configuration, every key aspect of the network management is set with rules for each participating member. For example, to approve the definition of a chaincode implementation to the channel, or to update the channel configuration, it requires, by default, a majority of administrators to approve a chaincode definition.

But ultimately, unlike permissionless networks, the disadvantage of this implementation is that it requires the trust of whoever controls the channel and endorses the product biography's smart contract, in order to gain in overall performance of the network to host a product biography.

6.1.3 Data privacy

As seen in the implementation results in Section 5.6, The private data of a user may only be queried if it is part of the organization holding it. To respect the scenario **S3**, the organization that owns private data has the only peer holding it and to prove it, this section will take an internal look into the state database each organization's peers.

LevelDB, the default database used in the Hyperledger Fabric test network, is too primitive for external database analysis, since it does not provide a server or Command-Line Interface (CLI), and to be able to view the data, a third party

viewer must be used in the filepath `/var/hyperledger/production` in each peer container.

Instead, the Fabric network was initialized with the `-s couchdb` option. This creates a separate CouchDB container for each peer, accessible with the ports `5984` for Organization 1 and `7984` for organization 2. CouchDB is a NoSQL database, offering a Graphical User Interface (GUI) interface and more complex query options. In the `/_utils` endpoint, using a browser, a login page is shown, displayed in Figure 6.4.

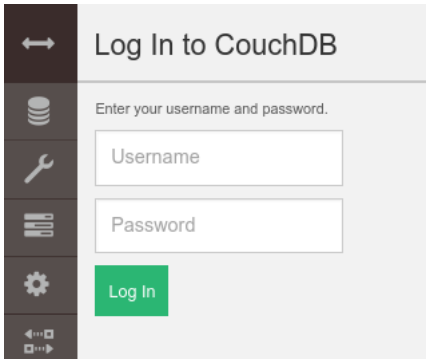


Figure 6.4: Login page of CouchDB.

The login credentials are available in the `docker-compose` file with the CouchDB instances, but may also be set in the peer's configuration file.

After logging in, a menu with different options to manipulate the database is available, and in the `/_utils/#/_all_dbs` page, every database used by the peer is displayed. The databases related to our product biography smart contract start with `mychannel_productbiography`. To visualize what databases are stored in each peer, a public and private product are inserted through Organization 1. The end result is presented in Figures 6.5 and 6.6, where each organization contains a `h_(...)_orgX` and `p_(...)_orgX` database, with its size in the middle and number of documents, or JSON objects, to the right.

mychannel_productbiography	0.6 KB	1
mychannel_productbiography$$h_implicit_org_$org1$msp	0.5 KB	1
mychannel_productbiography$$p_implicit_org_$org1$msp	0.6 KB	1

Figure 6.5: CouchDB databases of Organization 1.

mychannel_productbiography	0.6 KB	1
mychannel_productbiography$$h_implicit_org_$org1$msp	0.5 KB	1
mychannel_productbiography$$p_implicit_org_$org2$msp	0 bytes	0

Figure 6.6: CouchDB databases of Organization 2.

The `h` database only contains a reference to the private product, present in both databases, but with no actual data at all, which is instead stored in the `p` database. This difference in data is shown in Figures 6.7 and 6.8, respectively displaying the actual private data that only exists in Organization 1, and the reference that exists in both peers.

mychannel_productbiography$$p_implicit_org_$org1$msp

Save Changes Cancel


```
{
  "_id": "2dacef23-0990-416c-b214-52d7fe9f2e43",
  "_rev": "1-ae65ad7b1bbb2d80cfa7fa1520befd79",
  "color": "White",
  "events": [
    {
      "changes": {
        "color": "White"
      },
      "description": "Creation",
      "location": {
        "lat": "10",
        "lng": "20",
        "name": "Somewhere"
      },
      "timestamp": "13/01/2023, 11:40:24"
    }
  ],
  "id": "2dacef23-0990-416c-b214-52d7fe9f2e43",
  "name": "Private_1",
  "transformations": [],
  "~version": "CgMBBgA="
}
```


Figure 6.7: Private product document of the "p" CouchDB database.

As a note, the `version` attribute in both JSON objects is the same.

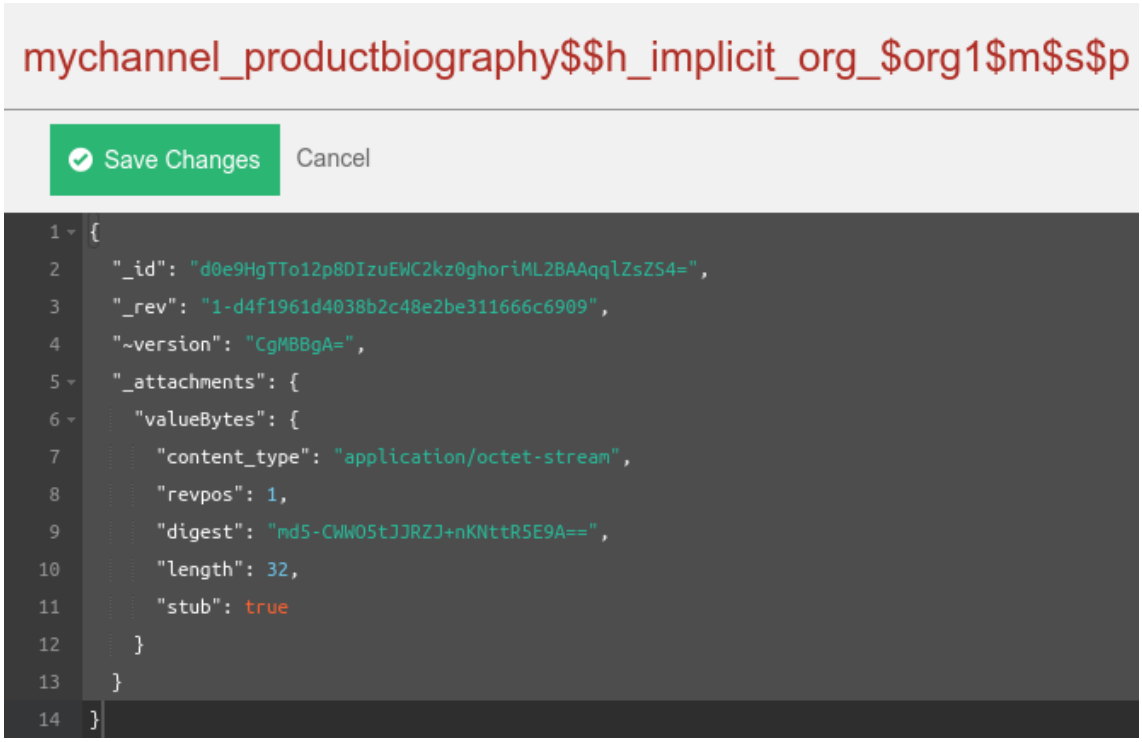


Figure 6.8: Private product document of the "h" CouchDB database.

Through this analysis, we can validate that the system is able to maintain the confidentiality of sensitive data, where only unauthorized access to the peer may leak this private data.

6.1.4 Data immutability

Finally, the last quality attribute scenario to validate is around data immutability, defined as **S4**: The blockchain is not reverted back by a single or a group of malicious entities. The use of a blockchain is especially useful to make sure appended data does not get modified.

Hyperledger Fabric provides a way, through the `peer` and `configtxlator` binaries, to get information about the blockchain ledger. In order to verify this scenario, it is important to verify the state of the blockchain while performing changes to a specific product, and compare the current world state with past blocks. The following commands are used to save the latest block into a file and decode it to JSON data:

```
# Get information about the current block
$ peer channel getinfo -c mychannel
InitCmdFactory -> Endorser and orderer connections initialized
Blockchain info: {"height":8,
"currentBlockHash":"cW22jR8QJLbqAREBOyn5JBEf6d0krxLv2wkhya76iPQ=",
"previousBlockHash":"s+mf+Z5oqXFIxEYDeBZiNCsnjeY2ILDKMZDvTjbMlmU="}
```

```
# Fetch latest block
$ peer channel fetch newest original_product.pb -c mychannel -o \
    <orderer_address> --tls --cafile <orderer_tls_certificate>
InitCmdFactory -> Endorser and orderer connections initialized
readBlock -> Received block: 7

# Displays the latest block information in JSON
$ configtxlator proto_decode --input latestBlock.pb --type \
    common.Block | jq
(...)
```

The `getinfo` command outputs the current and previous block hashes, as well as the number of blocks currently in use. After fetching the block and converting it, the JSON output still has encoded information, stored in base64. As an example, some base64 information available in `original_product.pb`, which is the extracted block's file name, is about a product with a single event, making the product's `color` equal to `White`. After making a change to this product, by generating an event which turns the `color` attribute to `Black` a new block is generated by the orderer and the blockchain updated by the peers. The final result is illustrated in figure data is effectively updated, which can be seen in the new latest block, but block 7 remains intact, showing the appended information did not get replaced.

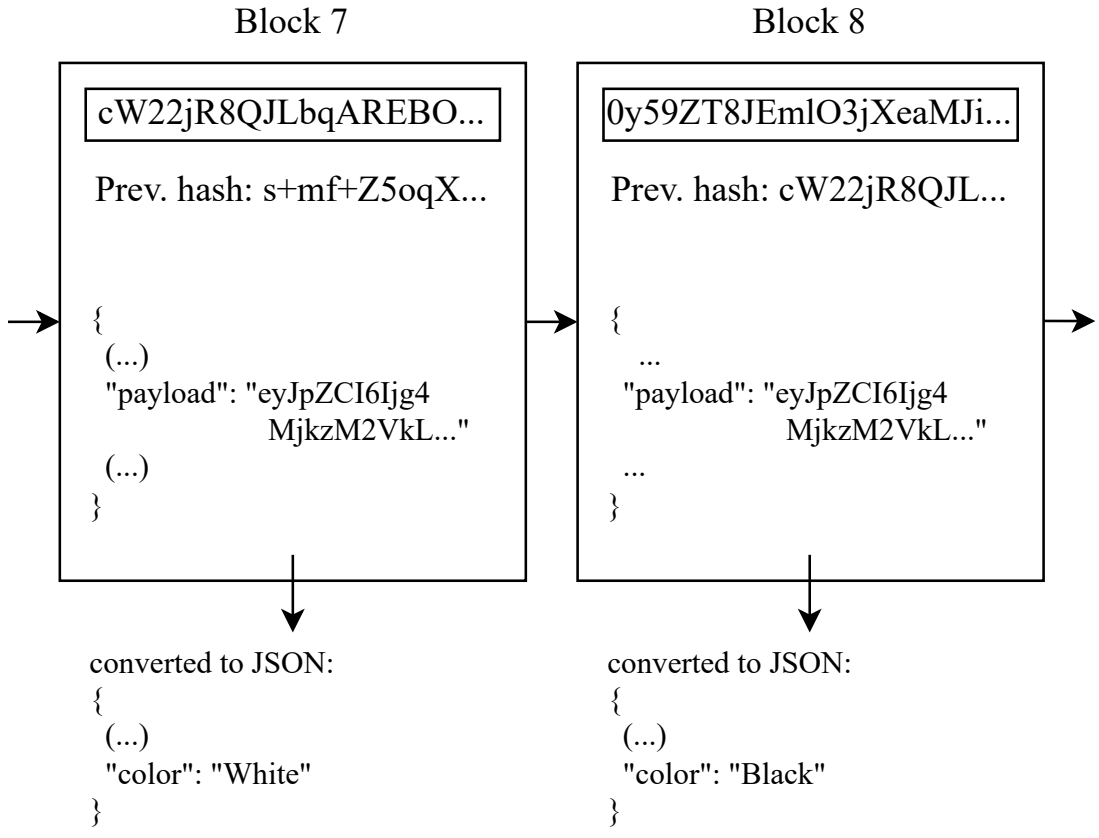


Figure 6.9: Brief representation of the Hyperledger Fabric blockchain.

Finally, we can validate the scenario, by confirming that the blockchain actually exists, has only appended blocks and any change occuring to the past block changes the hashes of the subsequent blocks.

This section has successfully validated the architecture around the QAS, through the implemented prototype, by simulating different actions in the blockchain and analysing what happens using different tools. The following section will show further options to assist the development of a platform for product biographies, applied to the technologies already used in the implementation process.

6.2 Further testing of the implementation

During the validation of the architecture, some tests on the blockchain itself and the network were already executed. This section will explore further possibilities of testing, around the development of smart contracts and the network itself.

6.2.1 Smart contract testing

While developing chaincode for the Hyperledger Fabric network, it is important to be able to test the code locally, instead of deploying the chaincode to a network everytime a change is made. Two options are available for local testing of the smart contract code:

1. Running a local peer in development mode;
2. Unit testing smart contract functions.

The first option is documented in [Hyperledger, 2022e], which requires setting up a network in a similar manner to the test network. Instead of using a channel configuration with rules for endorsement and proper policies divided between organizations, a configuration profile called `SampleDevModeSolo` is used, to simply run a single peer and orderer to run the chaincode with. In this case, the chaincode must still follow the deployment steps of Section 5.4.2, but without the installation part, to be effectively used.

Another option for debugging the smart contract is elaborating unit testing cases, for instance, for each exposed function in the contract. In JavaScript (JS), the language used during the development of this thesis, peer actions to replicate reading or writing data during transactions may be faked, using mock implementation of the chaincode interface, to make these operations in memory instead. Dependencies such as `sinon` [Sinon.JS, 2022] may be used, which offers the mentioned mock implementation, along with `chai` [Chai, 2022] as an assertion library.

Finally, to cover all nuances of blockchain development, it is still recommended to run a test network simulating the end network as closely as possible. Even with passing unit tests and proper trials in a development mode, issues related to the endorsement of the developed transactions may occur, for example.

6.2.2 Network analysis

Analysing network metrics is essential in a distributed network and hard to achieve the more a network scales. Hyperledger Fabric's test network contains images ready for network monitoring, running `prometheus` [Prometheus, 2022] and `grafana` [Labs, 2022a], which will be tested.

Prometheus is also an open-source tool, a monitoring and alerts system. Its goal is to collect metrics from a distributed network and store it in a time series database, which may be later used or just queried. Grafana is one of the tools that can use these metrics, in order to display it interactively in a dashboard, for example.

Upon starting the mentioned containers, Grafana exposes a port `3000` to access an available frontend, already plotting graphs for all sorts of metrics. An example is shown in Figure 6.10, displaying some content from an already existing dashboard, named "HLF Performance Review", already tailored for our needs.

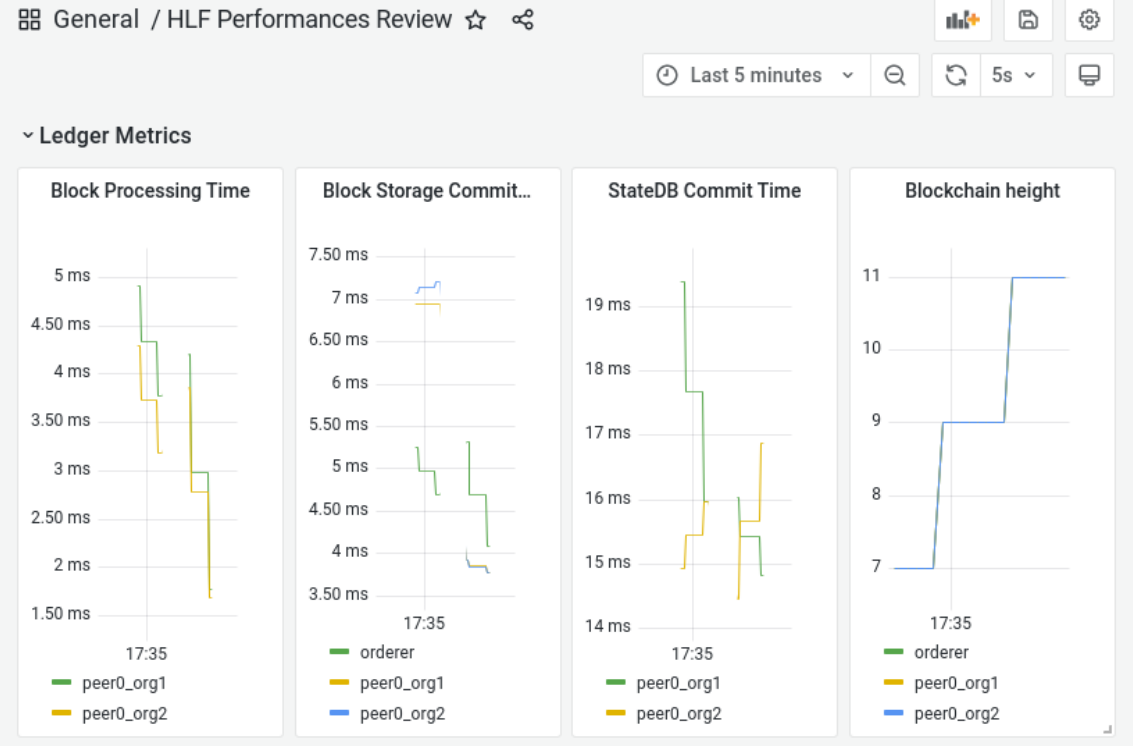


Figure 6.10: Ledger related metrics, from the Grafana dashboard.

After executing some transactions, these ledger metric show the time needed for certain activities, as well as the blockchain height going up for each new product, event or transformation inserted. More metrics are provided in this dashboard, such as:

- Central Processing Unit (CPU), memory and disk usage;
- Chaincode metrics, such as the number of requests received, completed and their duration;

- Endorsement proposals also received, completed and their duration.

Prometheus's metric gathering may also be expanded to cover the implemented client's REST API. Despite not being validated in the architecture of this thesis, these monitoring tools improve network *observability* and are useful to properly analyse the *performance* of the network.

To conclude, besides the dashboard analysis, a `monitordocker.sh` script is also provided by the `fabric-samples` repository, mentioned in Section 5.3. This script uses a program called `logspout`, used to route the different logs of the containers used for the network, into a single screen. It is especially useful to analyse the functioning flow of the network when a transaction occurs, for instance, by observing all nodes' processes at once.

6.3 Summary

In this chapter, the main goal of validating the previously designed architecture was achieved, respecting the formalized QAS of Section 4.1. During this validation process, iterating through each scenario, simulation of some parts of the prototype was required, along with using additional tools to gather more detailed information about blockchain data and the network itself. Finally, to uncover additional information about the network's operation, a monitoring and logging system were covered.

Chapter 7

Conclusion

This thesis has shown the possibilities of a platform of product biographies, using a blockchain network. Due to blockchain's applicability and benefits, the work started by analyzing and comparing the key design decisions made by various projects using it: two cryptocurrencies and two enterprise projects.

Going forward with a Hyperledger Fabric implementation for the blockchain network, high level requirements and quality attributes were identified for this platform, and a software architecture was designed for a first iteration of a blockchain network with the desired traits.

The implementation process proved that it is possible to apply the designed architecture in a functional way, due to the successful creation of a test network, a smart contract designed to control basic operations of a product biography, and a client application to operate with these products. It also suggests that Hyperledger Fabric is a possible solution, enabling a wide range of choices for future work.

The architecture was also validated and tested, using the current implementation as a prototype for simulating different outputs, to be assessed with proper tools.

In the following and sections, as predicted in the methodology used for this thesis in Chapter 2, we identify limitations of the output of this thesis and guidelines for a next possible iteration of the work.

7.1 Limitations

This section covers three main limitations of this thesis. These limitations describe what was not possible to complete during the thesis timeframe and may also be reconsidered in future work.

The implemented artifacts lack with real world data in a practical use case. Because of this, more specific requirements could not be defined, such as specific organizational structures and different modeling of data.

Scenarios about other relevant quality attributes, such as performance, availability and scalability, were not elaborated. These attributes are particularly challenging to verify realistically in a test network, as the response measures must be more objective and linked to a real life scenario, which is harder to achieve.

Finally, not much information by regulators is available to properly define architectural constraints. Despite these limitations, the thesis results provide insightful information on how to create a platform of product biographies. The last section outlines the future guidelines to expand the work in possible future iterations.

7.2 Future work guidelines

This final section will provide future work possibilities in different domains, addressing the work carried on by this thesis. These suggestions were divided by different domains, taking into account the steps of the methodology used for this thesis.

Starting with the solution objectives, future work may extend the goal of a generic product biography into more specific domains, such as covering the life of textile products, from raw material gathering to disposal. The objective would be extracting more value of the representation of specific products, in a well defined environment. The network may be implemented to adapt to this real environment, in order to gather real-time data of physical products.

For the architecture, the lacking quality attributes mentioned in the previous section may be applied and validated. Also, specific organizations may be referred and planned for in a production environment, such as different companies and regulators.

Finally, in the implementation part, the work done can be extended in a few ways. Adding interoperability options to the products is an option, such as structural compatibility with Ethereum Request for Comments (ERC)-721 and ERC-1155 or even an interface to enable cross-communication to Ethereum Virtual Machine (EVM) compatible networks, which are very popular nowadays. In relation to the data, as products, events and transformations grow in the peer's local storage, the database could be extended to a regular relational database, outside of the peers, to manage larger quantities of data in a more scalable fashion. In addition to these three data objects, others could be put in practice, depending on the domain and on the storage method used.

Hopefully, with this final analysis, the path to develop further iterations around this project becomes clearer, since the use of a platform of product biographies is unquestionably attractive for all entities participating in the life of a product.

References

Tarun Kumar Agrawal, Vijay Kumar, Rudrajeet Pal, Lichuan Wang, and Yan Chen. Blockchain-based framework for supply chain traceability: A case example of textile and clothing industry. *Computers & industrial engineering*, 154: 107130, 2021.

Kevin Werbach Aiden Slavin. Decentralized autonomous organizations: Beyond the hype, 2022.

Farhad Ameri and Deba Dutta. Product lifecycle management: closing the knowledge loops. *Computer-Aided Design and Applications*, 2(5):577–590, 2005.

Andreas M Antonopoulos, Olaoluwa Osuntokun, and René Pickhardt. *Mastering the Lightning Network*. " O'Reilly Media, Inc.", 2021.

Adam Back et al. Hashcash-a denial of service counter-measure, 2002.

João Barata, Vasco Pereira, and Miguel Coelho. Product biography information system: a lifecycle approach to digital twins. In *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 899–904. IEEE, 2020.

Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

Simon Brown. The c4 model for visualising software architecture, 2022. URL `https://c4model.com/`.

Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform., 2015a.

Vitalik Buterin. Decentralized society: Finding web3's soul, 2015b.

Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

Chai. Assertion library, 2022. URL `https://chaijs.com/`.

Aldar C-F. Chan. Utxo in digital currencies: Account-based or token-based? or both?, 2021.

Wai Kok Chan, Ji-Jian Chin, and Vik Tor Goh. Evolution of bitcoin addresses from security perspectives. In *2020 15th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 1–6. IEEE, 2020.

Dai Clegg and Richard Barker. *Case method fast-track: a RAD approach*. Addison-Wesley Longman Publishing Co., Inc., 1994.

CoinMarketCap. Coinmarketcap historical snapshot, 2022. URL https://coinmarketcap.com/historical/20210221/.

European Commission. Eu strategy for sustainable and circular textiles, 2022.

Vitalik Buterin Fabian Vogelsteller. Eip-20: Token standard, 2015. URL https://eips.ethereum.org/EIPS/eip-20.

The Wireshark Foundation. Wireshark, 2022. URL https://www.wireshark.org/.

Michael Fröwis, Andreas Fuchs, and Rainer Böhme. Detecting token systems on ethereum. In *International conference on financial cryptography and data security*, pages 93–112. Springer, 2019.

Henri Gilbert and Helena Handschuh. Security analysis of sha-256 and sisters. In *International workshop on selected areas in cryptography*. Springer, 2003.

Johannes Göbel and Anthony E Krzesinski. Increased block size and bitcoin blockchain dynamics. In *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 1–6. IEEE, 2017.

GS1. Gs1 general specifications, 2022a.

GS1. Gs1 system architecture, 2022b.

Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, pages 75–105, 2004.

Hyperledger. Sawtooth: An introduction, 2018.

Hyperledger. An overview of hyperledger foundation, 2021.

Hyperledger. Hyperledger fabric ca documentation, 2022a. URL https://hyperledger-fabric-ca.readthedocs.io/en/latest/.

Hyperledger. A blockchain platform for the enterprise, 2022b. URL https://hyperledger-fabric.readthedocs.io/.

Hyperledger. Channel capabilities - hyperledger fabric main documentation, 2022c. URL https://hyperledger-fabric.readthedocs.io/en/release-2.5/capabilities_concept.html.

Hyperledger. Updating a channel configuration - hyperledger fabric main documentation, 2022d. URL https://hyperledger-fabric.readthedocs.io/en/release-2.5/config_update.html.

Hyperledger. Running chaincode in development mode - hyperledger fabric main documentation, 2022e. URL https://hyperledger-fabric.readthedocs.io/en/release-2.5/peer-chaincode-devmode.html.

Hyperledger. Gossip data dissemination - hyperledger fabric main documentation, 2022f. URL https://hyperledger-fabric.readthedocs.io/en/release-2.5/gossip.html.

Hyperledger. Ledger - hyperledger fabric main documentation, 2022g. URL https://hyperledger-fabric.readthedocs.io/en/release-2.5/ledger/ledger.html.

Hyperledger. Membership service provider (msp) - hyperledger fabric main documentation, 2022h. URL https://hyperledger-fabric.readthedocs.io/en/release-2.5/membership/membership.html.

Hyperledger. The ordering service - hyperledger fabric main documentation, 2022i. URL https://hyperledger-fabric.readthedocs.io/en/release-2.5/orderer/ordering_service.html.

Hyperledger. Peers - hyperledger fabric main documentation, 2022j. URL https://hyperledger-fabric.readthedocs.io/en/release-2.5/peers/peers.html.

Hyperledger. Using the fabric test network - hyperledger fabric main documentation, 2022k. URL https://hyperledger-fabric.readthedocs.io/en/release-2.5/test_network.html.

Hyperledger. Hyperledger sawtooth documentation, 2022l. URL https://sawtooth.hyperledger.org/docs/.

Hyperledger. Setting up a sawtooth node for testing, 2022m. URL https://sawtooth.hyperledger.org/docs/1.2/app_developers_guide/installing_sawtooth.html.

Hyperledger. Transactions and batches - hyperledger sawtooth documentation, 2022n. URL https://sawtooth.hyperledger.org/docs/1.2/architecture/transactions_and_batches.html.

Hyperledger. sawtooth-validator - hyperledger sawtooth documentation, 2022o. URL https://sawtooth.hyperledger.org/docs/1.2/cli/sawtooth-validator.html.

Niclas Kannengießer, Sebastian Lins, Tobias Dehling, and Ali Sunyaev. Trade-offs between distributed ledger technology characteristics. *ACM Computing Surveys (CSUR)*, 53(2):1–37, 2020.

Enis Karaarslan and Enis Konacaklı. Data storage in the decentralized world: Blockchain and derivatives. *arXiv preprint arXiv:2012.10253*, 2020.

Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson, and Jeromy Carriere. The architecture tradeoff analysis method. In *Proceedings. fourth ieee international conference on engineering of complex computer systems (cat. no. 98ex193)*, pages 68–78. IEEE, 1998.

Grafana Labs. Grafana: The open observability platform., 2022a. URL https://grafana.com/.

Grafana Labs. Grafana: The open observability platform., 2022b. URL https://www.linuxfoundation.org/projects.

Richard A Lancioni, Michael F Smith, and Terence A Oliva. The role of the internet in supply chain management. *Industrial Marketing Management*, 29(1):1–12, 2000.

Jiongbin Liu, Gingsun Yeoh, Longxiang Gao, Shang Gao, and Ojelanki Ngwenyama. Designing a secure blockchain-based supply chain management framework. *Journal of Computer Information Systems*, page 5, 2022.

Sofia Lopes Barata and Paulo Rupino da Cunha. Legal and smart! an exploratory case study on understandability of smart contracts, 2019.

Brian Mulloy. Web api design, 2013.

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system", 2008.

Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pages 305–319, 2014.

Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

Prometheus. Monitoring system & time series database, 2022. URL https://prometheus.io/.

Yacine Rebahi, Jordi Jaen Pallares, Nguyen Tuan Minh, Sven Ehlert, Gergely Kovacs, and Dorgham Sisalem. Performance analysis of identity management in the session initiation protocol (sip). In *2008 IEEE/ACS International Conference on Computer Systems and Applications*, pages 711–717. IEEE, 2008.

Maryam Rezaei and Bin Liu. Food loss and waste in the food supply chain. *International Nut and Dried Fruit Council: Reus, Spain*, pages 1–2, 2017.

Tim Roughgarden. Transaction fee mechanism design for the ethereum blockchain: An economic analysis of eip-1559. *arXiv preprint arXiv:2012.00854*, page 3, 2020.

Khaled Salah, A Alfalasi, M Alfalasi, M Alharmoudi, M Alzaabi, A Alzyeodi, and Raja Wasim Ahmad. Iot-enabled shipping container with environmental monitoring and location tracking. In *2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6. IEEE, 2020.

Patrick Schueffel. Alternative distributed ledger technologies blockchain vs. tangle vs. hashgraph-a high-level overview and comparison. *Tangle vs. Hashgraph-A High-Level Overview and Comparison (December 15, 2017)*, 2017.

Sinon.JS. Standalone test spies, stubs and mocks for javascript. works with any unit testing framework, 2022. URL `https://sinonjs.org/`.

Martin Spring and Luis Araujo. Product biographies in servitization and the circular economy. *Industrial Marketing Management*, 60:126–137, 2017.

Nick Szabo. Formalizing and securing relationships on public networks. *First monday*, 1997.

Nick Szabo. Bit gold, 2022. URL `https://unenumerated.blogspot.com/2005/12/bit-gold.html`.

Sergio Terzi, Abdelaziz Bouras, Debashi Dutta, Marco Garetti, Dimitris Kiritsis, et al. Product lifecycle management-from its history to its new role. *International Journal of Product Lifecycle Management*, 4(4):360, 2010.

Abdul Wahab and Waqas Mehmood. Survey of consensus protocols. *arXiv preprint arXiv:1810.03357*, 2018.

Nick Webb. A fork in the blockchain: income tax and the bitcoin/bitcoin cash hard fork. *North Carolina Journal of Law & Technology*, 19(4):283, 2018.

Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger., 2022.

Liupengfei Wu, Weisheng Lu, Fan Xue, Xiao Li, Rui Zhao, and Maohong Tang. Linking permissioned blockchain to internet of things (iot)-bim platform for off-site production management in modular construction. *Computers in Industry*, page 3, 2022.

Appendices

Appendix A

Table A1: Data requirements.

ID	Description
1.1	The platform **must be** able to gather product data, with a name and its events and transformations.
1.2	The platform **must be** able to represent singular products, as well as batches of products.
1.3	The product data **must have** a name attribute and an unique identifier ID.
1.4	A batch of products **must have** quantity and unit attributes.
1.5	The product data **must be** immutable.
1.6	Each different product **should have** a modular data structure, depending on its needs.
1.7	The product data **could be** compatible with GS1 standards, an industry standard used worldwide for identification and tracking.
1.8	Public product data **could be** used externally, by following established NFT standards, such as the ERC-721 or ERC-1155.
1.9	The platform **could be** able to gather customer and reusage data, among other types of data from a circular economy point of view.
1.10	Product data **could be** used for a digital textile passport.
1.11	Additional data, such as images, **could be** stored outside of a ledger and referenced there.

Table A2: Stakeholder requirements.

ID	Description
2.1	Network administrators **must be** allowed to configure the network and define participants and policies.
2.2	Organization users and applications, such as automated IoT devices, **must have** different policies to query and input new product data in a ledger.
2.3	Other clients, organizations and applications **must be** able to make personal or transactional data private.
2.4	Any entity **must not be** allowed to edit or delete previously inserted product data.
2.5	Other clients, organizations and applications **must be** allowed to query unrestricted and authorized product data.
2.6	Clients **could be** associated to product data, publicly or privately, with attributes such as ownership.

Table A3: Network requirements.

ID	Description
3.1	The network **must have** a shared blockchain ledger of product data.
3.2	The blockchain **must be** distributed by, at least, more than one node from a different organization.
3.3	Different entities **should have** an interface to interact with the network and ledger.
3.4	Manual and automatic transactions to the network **should be** supported by the network.
3.5	Client front-end applications **could be** implemented to query product data.

Table A4: Infrastructure requirements.

ID	Description
4.1	Each ledger **must be** distributed to every participating node.
4.2	There **should be** at least one node maintaining the ledger for each participating organization.
4.3	Each participating organization **should have** at least one node verifying ledger transactions and smart contract executions impacting them.

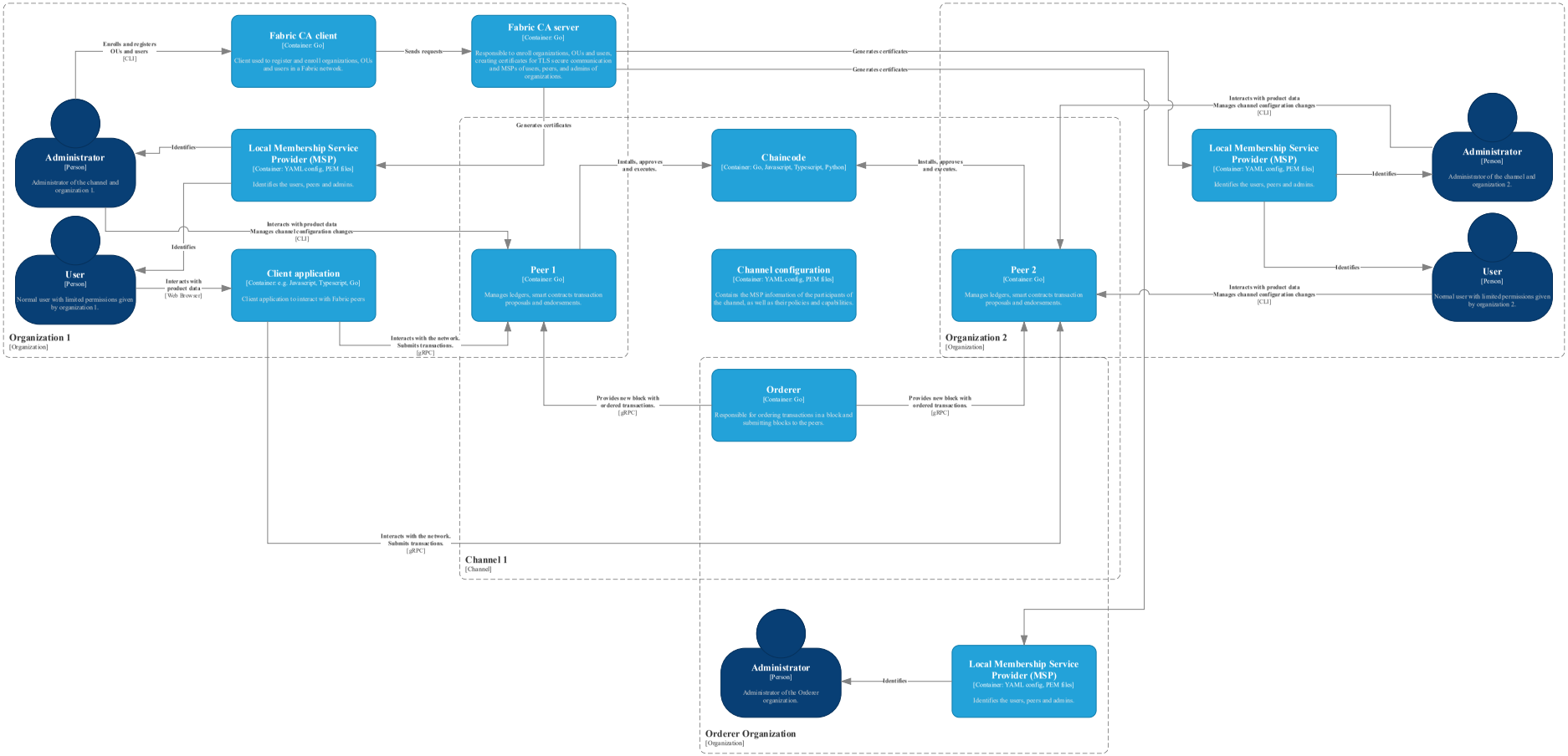


Figure A1: Complete layer 2 diagram of the C4 architecture model.

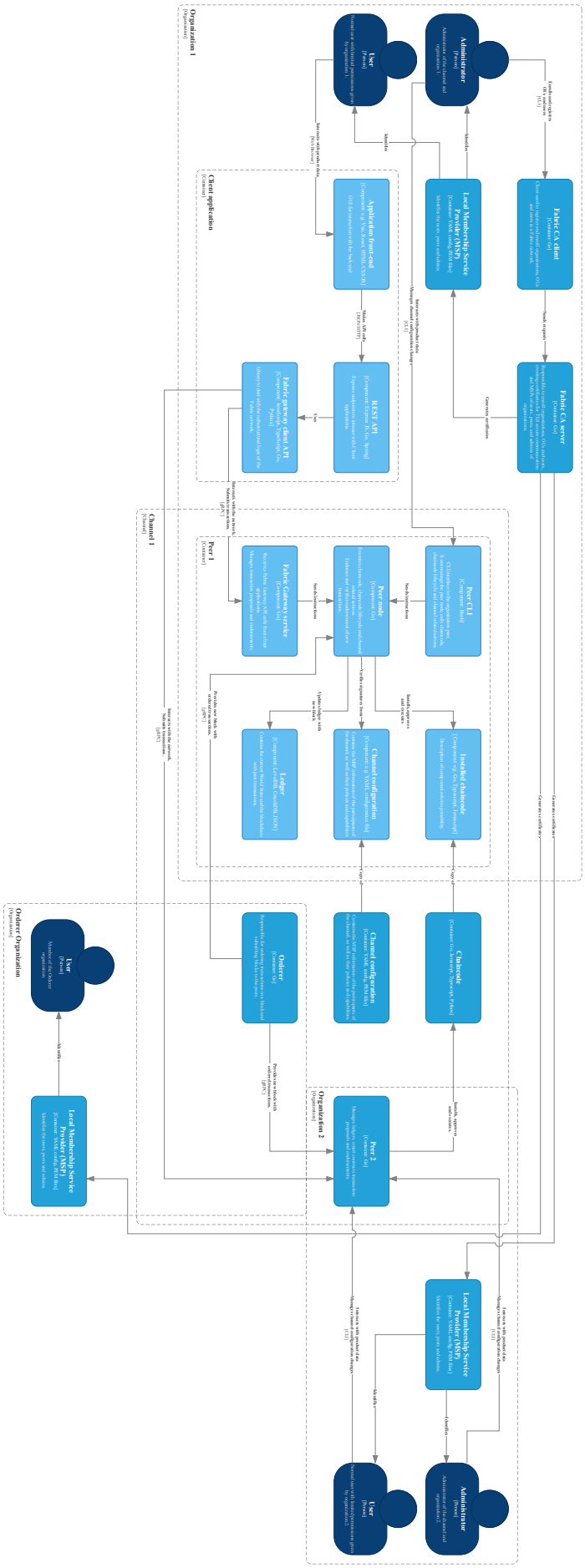


Figure A2: Complete layer 3 diagram of the C4 architecture model.