POLITECNICO MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

Application of the Blockchain to Supply Chain Traceability Systems

Tesi di Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: **Alessandro Sommaruga**

Student ID: 944890
Advisor: Prof. Francesco Bruschi
Co-advisors: Prof. Vincenzo Rana
Academic Year: 2020-21

Alla zia Lella

Abstract

The large number of actors involved in a supply chain network results in a lack of transparency and accountability. Nowadays, consumers are becoming more ecological and environmentally aware when shopping or selecting services, and paper-based certificates on products packaging are no longer sufficient to prove their quality and provenance.

Furthermore, standard methods for supply chain management do not keep up with the fast development of technologies and growing requirements on an increasingly globalised horizon. A possible solution to such problems, widespread in different chains, is the use of the emerging technology of blockchain.

As recent research highlights, blockchain-based services have huge potential to improve supply chains. The advantages of this approach are several. For example, the presence of a shared ledger, publicly available in almost no time, greatly reduces downtime (e.g., for quality controls), thus enabling a faster and more cost-efficient product delivery.

Within this context, the aim of the work is to design and develop a blockchain-based platform to manage supply chain traceability, improving efficiency. A model is proposed, flexible enough to be applied to multiple chains. A main requirement is to provide a user-friendly tool that integrates into existing solutions and can be used by non-experts in blockchain. Such a system implementation poses many challenges, addressed here.

To prove that out, it is introduced a real-world project, whose realisation includes different phases, such as the development of a blockchain infrastructure, involving the author's contribution. Specifically, some smart contracts have been developed on the Ethereum ecosystem, as well as their integration with already existing services.

The complete project implementation brings many benefits to the supply chain, mainly in terms of overall efficiency. But the real innovative aspects consist of *flexibility* and *versatility*, that allow for tracing goods throughout the whole chain, keeping a high blockchain *scalability*, obtained through a particular architectural solution. In addition, an innovative transaction *fee management* is proposed.

Keywords: Blockchain, Supply chain, Smart contracts, Supply chain traceability

Abstract in lingua italiana

L'elevato numero di attori coinvolti nel contesto di una filiera di approvvigionamento comporta una mancanza di trasparenza e responsabilizzazione. Al giorno d'oggi, i consumatori stanno diventando più ecologici e attenti all'ambiente, quando acquistano o selezionano servizi e i certificati sulle confezioni dei prodotti non sono più sufficienti a dimostrarne la qualità e provenienza.

Inoltre, i tradizionali metodi per la gestione di filiera non stanno al passo con il rapido sviluppo delle tecnologie e l'aumento delle esigenze in un contesto sempre più globalizzato. Una possibile soluzione a tali problemi, comuni a filiere di diversi settori, è l'utilizzo dell'emergente tecnologia blockchain.

Come evidenziato da recenti ricerche, i servizi basati su blockchain hanno un enorme potenziale per migliorare la filiera. I vantaggi di questo approccio sono molteplici. Per esempio, la presenza di un libro mastro condiviso, disponibile immediatamente, riduce notevolmente i tempi di inattività (si pensi al controllo qualità), consentendo così una consegna dei prodotti più rapida ed efficiente in termini di costi.

In questo contesto, lo scopo della tesi è quello di progettare e sviluppare una piattaforma basata su blockchain per gestire la tracciabilità di filiera, migliorandone l'efficienza. Viene proposto un modello, abbastanza flessibile da essere applicato a più filiere. Un requisito principale è fornire uno strumento di facile utilizzo, che si integri all'interno delle soluzioni esistenti e possa essere utilizzato dai non esperti di blockchain. La realizzazione di un tale progetto pone un gran numero di sfide, affrontate in questo documento.

A riprova di ciò, viene presentato un progetto reale, la cui realizzazione include lo sviluppo di un'infrastruttura blockchain, che ha visto il contributo dell'autore. In particolare, è mostrato lo sviluppo di alcuni contratti intelligenti su ecosistema Ethereum e la loro integrazione con servizi già esistenti.

Il modello implementato porta molti benefici alla catena di approvvigionamento, principalmente in termini di efficienza complessiva. Ma i veri aspetti innovativi sono *flessibilità* e *versatilità*, che permettono di tracciare le merci lungo l'intera filiera, pur mantenendo

un'alta *scalabilità* nella blockchain. Inoltre, è proposta una gestione innovativa del *costo* delle transazioni.

Parole chiave: Blockchain, Filiera, Contratti intellligenti, Tracciabilità di filiera

Contents

Abstract iii

Abstract in lingua italiana v

Contents vii

Introduction 1

1 Blockchain 5

1.1 History . . . 6
1.1.1 Origin . . . 6
1.1.2 Double-spending problem . . . 6
1.2 Definitions . . . 7
1.2.1 Blockchain key elements . . . 8
1.2.2 Transactions . . . 9
1.2.3 Blocks . . . 10
1.3 Consensus . . . 11
1.3.1 Proof-of-Work . . . 12
1.3.2 Proof-of-Stake . . . 14
1.4 Types . . . 15
1.4.1 Public and private . . . 15
1.4.2 Permissionless and permissioned . . . 16
1.5 Evolution . . . 18
1.5.1 Smart contracts . . . 18
1.5.2 DApps . . . 19
1.5.3 Fungible and Non-Fungible Tokens . . . 20
1.6 Useful concepts . . . 21
1.6.1 Notarization . . . 21
1.6.2 Relayer and meta-transactions . . . 22

1.6.3 Token economy 23
1.6.4 Contract factory 23
1.7 Applications 24

2 Blockchain for the Supply Chain: State of the Art 27
2.1 Supply chain 27
2.1.1 Definition of supply chain 28
2.1.2 Supply chain management 28
2.1.3 Digital supply chain 29
2.2 Blockchain for the supply chain 30
2.2.1 Introductory examples 30
2.2.2 State of the art 30

3 The Model: Blockchain for Supply Chain Control Systems 35
3.1 System description 36
3.1.1 Component architecture 37
3.1.2 Blockchain and smart contracts 41
3.2 Usability concerns 46
3.2.1 Non-expert users 47
3.2.2 Transaction fees 48
3.2.3 Centralisation 50
3.3 Features and innovative aspects 51
3.3.1 Versatility 52
3.3.2 Traceability 52
3.3.3 Scalability 53
3.3.4 Efficiency 53

4 Use case: Deply project 55
4.1 Contribution 57
4.2 Project description 59
4.2.1 Task and phases 59
4.2.2 Architecture 60
4.3 Implementation 63
4.3.1 Implementation of the application 64
4.3.2 Smart contracts 66

5 Evaluation 79
5.1 Features of the model 80

5.2 Smart contracts 82
5.2.1 User management testing 84
5.2.2 Notarization testing 97
5.3 Usability and functional aspects 102
5.3.1 Integration to legacy systems 102
5.3.2 Management of cryptographic keys 104
5.3.3 Notarization of incorrect information 104
5.3.4 Transaction fees 105
5.3.5 Centralisation 105

6 Conclusions and future developments 107
6.1 Future developments 108

Bibliography 111

A Appendix A 115

B Appendix B 123

C Appendix C 125
C.1 TestUserManagement 125
C.2 TestNotarization 136

List of Figures 143

List of Tables 145

List of Listings 147

Acknowledgements 149

Introduction

The large number of actors involved in a supply chain network results in a lack of transparency and accountability. Nowadays, consumers are becoming more ecological and environmentally aware when shopping or selecting services. Thus, trust in a brand is fundamental since paper-based certificates applied on products packaging are no longer sufficient to prove their quality and provenance.

Furthermore, another problem of the current supply chain is that the standard methods for supply chain management do not keep up with the fast development of technologies and greater requirements of competitive supply chains in an increasingly globalised horizon.

The problems just presented are not related to a specific supply chain, but affect multiple sectors, different from each other. For instance, the aerospace, agrifood, automotive, cosmetics, pharmaceutical sectors are only some of them.

A possible solution to the problem just presented is the use of blockchain. The blockchain is a Distributed Ledger Technology (DLT), a kind of technology which gives read and write access to a ledger shared among some users. Such a mechanism provides great benefits in terms of decentralisation, transparency, traceability, and immutability. It has application in many contexts, where the most popular is certainly the Bitcoin cryptocurrency in the finance domain.

Among the large number of the applications of the blockchain, an emerging one is that on the supply chain. Because of its properties, this technology allows to fix data in an immutable way on a distributed ledger: such feature can be exploited in order to trace products and processes in the supply chain context.

As recent research highlighted [1], blockchain-based services have a huge potential to improve supply chains. The advantages of this approach are several: enabling faster and more cost-efficient delivery of products, enhancing product traceability, avoiding product fraud and counterfeiting, improving interoperability and coordination between all supply chain members.

For example, the presence of a shared ledger, publicly available in almost no time, greatly

reduces downtime during different supply steps, thus enabling a faster and more cost-efficient product delivery.

Thanks to its transparency, blockchain also allows consumers to award those companies that decide to change for good, towards more sustainable, ethical and environmentally friendly production models. Indeed, consumers could view information about production processes. If they see a change in a company methods or increased quality controls, they could decide to purchase their products.

Motivating example

Transparency is a property that enhances production processes and products. Some supermarket brands allow customers to view the origin of the goods from their label, scanning a QR code. The same can be applied more generally to multiple sectors.

All consumers can scan a QR code and have access to information related to the collected production data. For instance, the information could be about: location where a product has been manufactured, transportation documents or results of laboratory analyses executed on foods, drugs or cosmetics, as well as, where relevant, the temperature detected during their transport.

This can be a discriminator in the choices of consumers, both for the origin (e.g., Italian people may prefer local goods) and for the production methods (handmade product/more thorough controls).

Objectives

The rationale behind this thesis is to provide innovative tools to improve the efficiency of the entire supply chain. More specifically, this idea results in the following goal.

The aim of this work is to design and develop a blockchain-based platform to manage supply chain traceability, also improving chain efficiency. The proposed model is meant to be general and applicable to multiple supply chains. The latter is one the innovative aspects that this thesis wants to bring: the solution must be comprehensive and flexible to different chains, while - currently - only sector-specific solutions exist.

For achieving that, innovative solutions are brought to the blockchain part as well. Among them, there is high blockchain scalability and a careful handling of transaction fees, obtained by means of ground-breaking architectural techniques.

A main requirement is to provide a user-friendly tool that integrates within the currently existing systems in supply chains and can also be used by non-experts in blockchain. The realisation of a system with all the mentioned features poses a large number of challenges, addressed in this document.

Methodology

The methodology adopted in this work is to first present all the necessary notions from a theoretical point of view about blockchain, supply chain and the application of the first to the second. After building up the proper context, a model is described that constitutes a potential solution to the introduced problem.

Then, the target is moved to a possible implementation of the model: a real-world use case is introduced, and the implementation is explored. Such project also involved the contribution of the author, during different phases, like the design of the system architecture, the design and development of the blockchain part, the user interface implementation and the testing.

As it will be clearer after the theoretical chapter about blockchain, the methodology for the blockchain part involves the development of some smart contracts on the Ethereum ecosystem, and the integration with already existing services (e.g., relayers), to improve the performance. In addition, all the code - before concretely using it - goes through extensive testing.

Outline

The following section contains the outline of the current thesis, including the organisation in chapters and a brief overview of each chapter content.

- Chapter 1. *Blockchain* - History, definitions, types, notable concepts, evolution and applications of the blockchain technologies.
- Chapter 2. *Blockchain for the Supply Chain: State of the Art* - Supply chain context and definitions, state of the art about the application of the blockchain to the supply chain.
- Chapter 3. *The Model: Blockchain for Supply Chain Control Systems* - Core part of the thesis, contains the (theoretical) solution to the problem posed, also introducing usability concerns.

- Chapter 4. *Use case: Deply project* - Description of the concrete project, implementing the proposed model, mention to the author's contribution.
- Chapter 5. *Evaluation* - Considerations about the implemented system with respect to the theoretical one, completeness analysis of the code, practical difficulties encountered, also about the usability aspect.
- Chapter 6. *Conclusions and future developments* - Summary of the project outcome and possible enhancements.

1 | Blockchain

While until a few years ago blockchain technology was perfectly unknown to the vast majority of people, nowadays it is becoming more and more popular and blockchain topics are talked-about all over the world. The sector that is subject to the largest wide spreading is finance: the trading advertisements and the discussions on social media are now common, with the cryptocurrencies, headed by Bitcoin, constituting the main theme. Without mentioning the emerging technology of the NFTs (Non-Fungible Tokens), which, after becoming popular in 2018, in the recent months has been affected by a really huge increase in interest.

However, that is just the tip of the iceberg. Under the surface of the debated - and sometimes too much speculated - topics, the blockchain has enormous potential from a research point of view. Actually, its applications range from governance, to the IoT, to digital identity verification.

The first chapter of this thesis aims at introducing the blockchain technology from a theoretical point of view, in order to provide the tools to fully understand the contents of this work, as well as having a greater understanding on the more and more popular blockchain-related innovations.

A particular attention is put on those concepts that, though they are not considered among the most common ones, are likewise important and relevant for this thesis purposes.

As far as the contents are concerned, the chapter can be logically divided into two parts. The first part is more theoretical and includes some basic definitions constituting all the small pieces that concur at the realisation of the complete picture (such as the definition of transaction and block). A brief history, some information about the consensus mechanisms, the typologies, and more advanced topics (such as smart contracts and DApps) are provided as well. This part covers multiple distinct sections.

Opposite to that, the second logical part of the chapter is fully contained in the last section (Section 1.7). The second part is a bit less theoretical and is devoted to the applications of blockchain technology. It better contextualises the tools introduced in the first part,

by describing some of the most common applications of this technology, such as finance, Industry 4.0 and supply chain. It is meant to be a general overview, done from a general perspective, without going into the details of any specific application.

1.1. History

This section clarifies the origin of the blockchain concept, both from a historical point of view and from a rationale perspective, by presenting a motivating example, related to solving the double spending problem.

1.1.1. Origin

Sherman et al. [2] give an overview of blockchain origin, starting from the very beginning, when the foundation ideas around blockchain were expressed.

The blockchain concept became popular in 2008 with the publication of the well-known paper titled *Bitcoin: A Peer-to-peer Electronic Cash System*, by the mysterious author Satoshi Nakamoto [3]. Actually, the name Satoshi Nakamoto is just a pseudonym for an unknown person. The major contribution of this paper, called the *white paper*, was the definition of an algorithm that enables the implementation of a distributed digital currency.

However, as stated by Sherman et al., most of the blockchain underlying principles had arisen many years earlier. For instance, the idea of a blockchain as a type of distributed database goes back to at least the 1970s (Wong [4]), while the idea of immutably chaining blocks of information with a cryptographic hash function appears in 1979 Merkle's dissertation [5]. The just-mentioned concept of hash, that may not be clear now, is explained in the next section (Section 1.2).

In 1990, two cryptographers, Haber and Stornetta [6] applied these ideas to time-stamp documents, i.e., associating each document with a precise date and time. These prior works laid the basis for modern blockchain, even if they do not include all the elements and techniques of blockchain.

1.1.2. Double-spending problem

More than analysing in detail the historical evolution of blockchain, to understand the reason why the blockchain technology has reached such a large diffusion, it is important to clarify the rationale behind its development, through an example.

Blockchain applications allowed to solve the so-called *double-spending problem*. The double-spending problem refers to the possibility that a valuable token, such as, for example, a car-wash token or a cryptocurrency, can be used twice or more. This issue concerns in particular the digital assets and not the physical ones: for instance, after inserting a coin into the car-wash machine, it is impossible to use it again.

In a decentralised system, it may occur if there is no way to check that the amount of tokens spent for a transaction are not simultaneously spent in another one. Considering the cryptocurrency context, it is a problem because a person who spends an amount of a cryptocurrency more than once, creates a disparity between the spending record and the amount of available cryptocurrency, as well as its distribution, basically breaking the system.

Bitcoin has been the first major digital currency whose mechanism solved the double spending issue. It achieved that by its innovative consensus mechanism, called Proof-of-Work (PoW), which is a mechanism for validating the transactions (more details about it are provided in Section 1.3). Before Bitcoin, it was not possible to do so while maintaining a universal ledger shared among the users. In this way, the Bitcoin blockchain keeps records of time-stamped transactions, going back to the first operations performed in 2009.

1.2. Definitions

This section contains the definition of blockchain considered in this work, as well as some related concepts such as transactions, hash, blocks, and others. The key elements that must be present in a blockchain are pointed out. Mining is also introduced, even if it is a feature of specific blockchains.

Before starting the description, an important remark needs to be made. Blockchain is part of a larger family, called Distributed Ledger Technologies (DLT), consisting of a logic of decentralisation and distribution of data and information, where the ledger is shared among all participants of the network. The same blockchain is not a single technology. Rather, it consists of an umbrella of multiple technologies and principles that combined together realise the function of a distributed ledger technology.

Such an idea of the umbrella consists of three fundamental categories, as pointed out by Swan [7]. The first, Blockchain 1.0, refers to the finance domain. The cryptocurrencies like Bitcoin, whose core features are linked to cash and monetary transactions, fall within it. The second category, Blockchain 2.0, includes the set of applications going beyond

simple cash movements, like smart contracts, smart property and loans. Blockchain 3.0, instead, goes beyond finance and cryptocurrency, but is about decentralised applications (DApps), for instance in the areas of art, health and governments.

As for this section, the focus is posed to Blockchain 1.0 and basic blockchain concepts; the evolution of such technology is discussed in Section 1.5 below.

1.2.1. Blockchain key elements

Antonopoulos presents the key elements of a **blockchain** in his book [8]. It must be said that the description is about an *open*, *public* blockchain: we specify the meaning of such attributes later on in Section 1.4. Nevertheless, such a definition is still suitable for our purposes.

The components of an open, public blockchain are the following:

- A *peer-to-peer (P2P) network*, which connects participants and allows for the propagation of transactions and blocks of verified transactions throughout the whole network, based on a standard propagation protocol. Each participant to the network is called a *node*.
- Messages, in the form of *transactions*, representing state transitions. The easiest possible transaction consists of currency transmission, in which a certain amount of value is transferred from a sender to a receiver.
- A set of *consensus rules*, governing what constitutes a transaction and determining which state of the blockchain is valid, resolving disputes about conflicting situations. Such rules take the form of a consensus algorithm, that decentralises control over the blockchain, by forcing participants to cooperate in order to comply with the consensus rules.
- A *state machine*, that processes transactions following the consensus rules. This could be simpler (as for Bitcoin), or more complex (as for Ethereum), according to how much complexity it is able to handle.
- A chain of cryptographically secured *blocks*, consisting of information records, that acts as a log of all the verified and accepted transactions.
- One or more *clients*, (open source) software implementations of the above components. The open source characteristic is only needed for public blockchains: private platforms are likely not to disclose their implementation.

Furthermore, Antonopoulos [8] also cites a reward scheme to economically secure the state

machine. Providing an incentive is particularly needed in a public environment to make the validation process more robust. More about this topic and consensus is reported in Section 1.3.

1.2.2. Transactions

Let's now move to the **transaction** topic. We said that a blockchain allows users to perform transactions in order to change the blockchain state. Each transaction has a sender and a receiver. In the most simple case, it consists of the payment of some valuable tokens (e.g., a cryptocurrency) from a sender user to a receiver one.

So, how to unambiguously distinguish users from each others? Each actor is identified in the blockchain by an address, usually expressed in hexadecimal notation. Blockchain actors are the users, but they can also be smart contracts, as it will be pointed out in the Section 1.5. An example of address is the following.

`0x1322d57171d5a3008f9e630b2e8518e3`

The blockchain technology relies on asymmetric encryption. This paradigm requires that each participant owns a pair of keys: a public and a private key. The address of an actor just introduced is nothing but its public key. Such a key is like a person's home address. By knowing that, anyone can send letters to it. Similarly, the address in the blockchain context is public and who knows it can send transactions to the owner. The private key, instead, is only known by its owner, and allows to enforce good communication properties, for example in the digital signature mechanism.

Actually, communication in the blockchain exploits the digital signature principles. These consist of employing the private and public key alternatively for the encryption and decryption of messages (i.e., transactions). Another ingredient used within this process are hash functions, which are worth a little digression.

A **hash function** takes some data as input and produces a sequence of bits or a string, the so-called *digest*, strictly related to the input data. Intuitively, the digest can be seen as a fingerprint of the input representing it in a compact way. In the IT (Information Technology) language, the hash is a non-invertible function that maps a string of arbitrary length into a string of fixed length. In particular, for cryptography purposes, an ideal hash function should have some fundamental properties:

- It must uniquely identify the message. It should never happen that two similar

messages have the same hash value;

- It must be deterministic, so that the same message is always translated in the same hash;
- The hash value must be fast and easy to compute from any kind of data;
- It must be a one-way function, i.e., a function *easy* to compute, but *hard* to invert, from a computational point of view. In practice, it must be very difficult or almost impossible to retrieve the original message from its hash value if not trying all possible messages.

A cryptographically secure hash should not allow to find a text that can generate it, in a time comparable with the use of the hash itself.

In a digital signature scheme, first the hash of the transaction is computed; then it is encrypted with the sender's private key, to prove integrity, authenticity and non-repudiation properties. Combining digital signature with asymmetric encryption techniques (i.e., encrypting the transaction with the receiver's public key) also enables confidentiality.

Let's also make a terminology specification about the difference between two terms: on-chain and off-chain. *On-chain* data include information about all transactions on a particular blockchain platform, while *off-chain* denotes any process or transaction performed outside the blockchain, which therefore does not benefit from the immutability and decentralisation inherent in the blockchain.

1.2.3. Blocks

Before proceeding with consensus mechanisms, the concept of **block** must be investigated more. A block can be seen as a set of data about transactions. Anders Brownworth [9] provides a schematic representation in its blockchain demo, represented in Figure 1.1.

Each block has a growing number, a nonce, some data, a reference to the hash of the previous block and the hash of the current block. The data field contains information about transactions included in the block: the number of transactions per block must be lower than a fixed maximum. The hash of the previous and current blocks are needed to link blocks in a chain, defining the proper order of transactions. The nonce is a number, put in the block header: its use is explained in the next section, as well as that of the *Mine* button at the bottom of the block. Notice that if the nonce changes, also the hash changes.

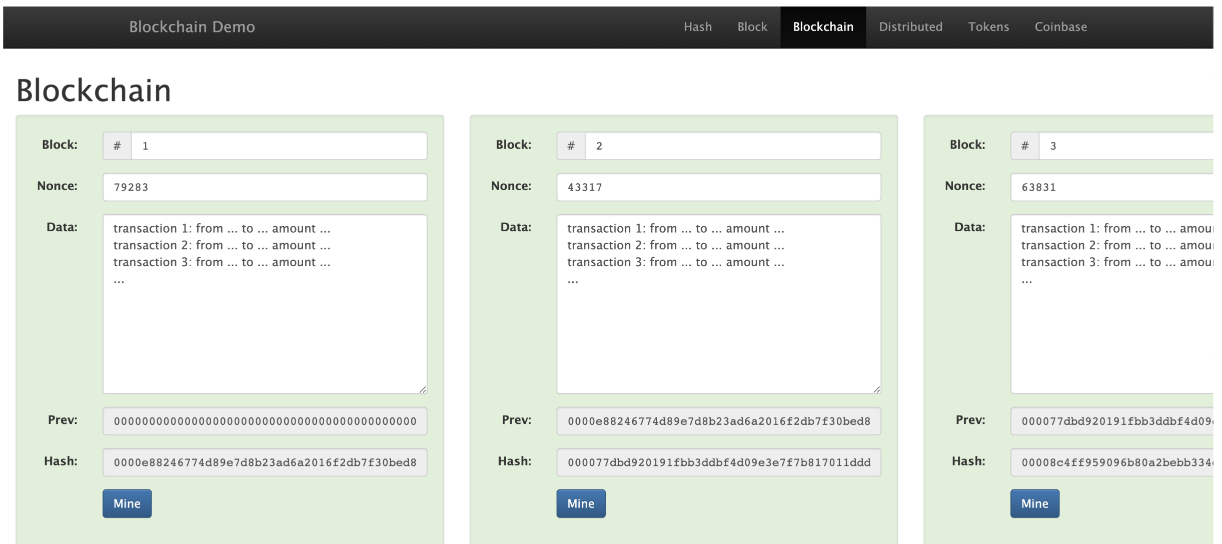

Figure 1.1: Blockchain demo from Anders Brownworth [9].

Another important concept is that of **fees**. In order to reach consensus, someone in the network must verify transactions and upload the blockchain distributed ledger with the needed changes. This operation is expensive, therefore it must be somehow payed. For this reason, transaction fees are introduced. The network participants recognise a portion of the cryptocurrencies they own as a reward for transaction execution and their organisation in blocks. This concept is better clarified when considered in relation to the consensus mechanism.

1.3. Consensus

The current section presents the consensus topic. As anticipated in the previous section, reaching a deal about agreement strategies is a fundamental point. In this regard, consensus mechanisms and their actuation into consensus algorithms are introduced. The main ones, the Proof-of-Work (PoW) and Proof-of-Stake (PoS) methods are explained.

Consensus among the network participants is one of the most important and revolutionary aspects of blockchain technology. A **consensus mechanism** allows continued and secure updates of the register. These updates contain the new information to be added onto the ledger, and the process of checking and verifying them is called validation. Its aim is to protect the network from attacks and counterfeiting.

Achieving consensus in a distributed network means to ensure that the nodes of the network eventually converge to a common state and append the agreed value to the blockchain state. The challenging part here is that the context is distributed, and thus

the tasks of control and validation do not belong to a centralised authority, but they are assigned to the multiple nodes of the network. Validator nodes, in particular for Proof-of-Work contexts, are referred to as miners. According to the consensus mechanism, the rules for becoming a validator are defined.

Following, the main consensus algorithms actually employed are presented.

1.3.1. Proof-of-Work

This section is about the consensus mechanism which is currently mostly adopted and corroborated. Proof-of-Work (PoW) is the mechanism proposed by Satoshi Nakamoto in his white paper [3], and it is the one presently exploited by Bitcoin and Ethereum, the two biggest cryptocurrency platforms, as well as a lot of others.

In PoW, the process of collecting and validating transactions is called **mining**. The term comes from the metaphor of a miner who struggles to extract precious minerals. Similarly, Bitcoin miners, to be able to introduce new currencies, must spend time on expensive and computationally complex operations, then providing a proof of their work. Hence, the name Proof-of-Work, in reference to the validation system mechanism.

PoW is described by Reyna et al. [10] as *a computationally intensive task that is necessary for the generation of blocks*. It is like a really complex puzzle, requiring a high computational power to be solved. In order to solve that puzzle, a miner should find the right *nonce* (i.e., a random number) to be added in the header of the block. The effect of the nonce is to modify the hash of the block.

The nonce should be done so that the hash of the block (specifically, the hash of the header) starts with an amount of zeroes equal to the one expected by the network. PoW nodes are called miners, and they are incentivised to consume energy - thus money - to produce the next block, since who generates it is rewarded. Specifically, a miner should insert in his block a special transaction, in which he transfers, to an address of his choice, a certain amount of newly-created Bitcoin (or, in general, cryptocurrencies). Thus, such a reward is made up of newly-forged cryptocurrencies, plus the transaction fees paid by users.

Precisely, mining means to find out such nonce that makes the block pointed by that hash meet the requirements defined by the consensus policy of the system. In the case of PoW, for instance, the requirement imposed is that the hash output must have a specific number of zeroes at the beginning. For instance, the demo implemented by Anders Brownworth [9] is featured by 4 leading zeroes. This can be also seen in Figure 1.1, looking at the

Hash and *Prev* (previous hash) fields.

The number of prepended zeroes to the hash is not fixed, but it is a parameter of the system, called **difficulty**. By increasing the amount of starting zeroes, the difficulty increases, and so the computational power required to find the right nonce; vice versa, by decreasing that amount, the required power decreases. The difficulty of the problem varies exponentially: to find a hash starting with 3 zeroes is way much easier than finding one that starts with 10 or more zeroes.

In Bitcoin, the difficulty parameter is tuned so that a miner, before being able to find the solution, takes an average of 10 minutes. Since the network is open and everyone can join it and become a miner, the overall computing power of the miners is variable: it can grow or reduce. In the event that it increases, and thus the time to produce a block reduces, the target of 10 minutes is restored by increasing the difficulty. According to Bitcoin policy, this happens every 2016 block, corresponding to about two weeks.

Specifically, the difficulty is measured in TeraHash per second and over time it significantly increased. In public networks like Bitcoin, anyone can participate in the mining, although the odds of winning the competition and adding the next block are directly proportional to the computing capacity available. In its very first years, it was possible to extract blocks even with a simple personal computer. Today, instead, - in order to concretely have a chance to make it - is necessary to use specialised and expensive machines.

To summarise, the whole mining process is meant to check the previous transactions, to fulfil two operations. First, to verify whether a network user has the right to perform that transaction, for instance, if the sender owns at least the cryptocurrency amount to be transferred. Second, to resolve a computational-intensive mathematical problem before adding a new block to the chain.

Consensus mechanisms such as PoW - and Proof-of-Stake described below - are built to avoid forks and data manipulation. A **fork** occurs when the simultaneous generation of blocks in the network leads to temporarily different branches coming from different miners. Since eventually convergence must be reached on the whole distributed ledger, forks need to be prevented. A policy is required to choose between a pair of diverging branches. In particular, the most pursued policy is the *pick the longest blockchain* one.

The effect of this policy is explored following. After concurrent blocks have been produced, miners have to choose which block to consider the right one, because they have to include its hash in their block, and then devote work on it. Now, a miner is interested to go for the block that will be considered, in the future, the right one, which is the same chosen by

all the others. If that does not happen, the miner's block becomes part of a dead branch of history, and his reward is lost. As a consequence, forks are intrinsically *unstable.* As soon as one of the two branches exceeds the other in length, that one is considered the right one, and all the miners concentrate their work on it, boosting its development even more, and creating a gap with the other branches.

However, PoW has a big drawback. The computational power required to extract a block is constantly increasing, due to the continuous growth in the difficulty. Its consequences concern the energy consumption of the whole network. Although it is impossible to determine the exact consumption related to mining, it is yet possible to estimate it. According to data from the University of Cambridge Bitcoin Electricity Consumption Index (CBECI), the present consumption due to mining Bitcoin is about 134 TeraWatt-hour, comparable to the consumption of a medium-sized nation.

From these data, it is clear that the PoW algorithm is extremely expensive and involves huge power consumption, having a non-negligible environmental impact.

1.3.2. Proof-of-Stake

For the reasons mentioned above, a different consensus mechanism is sought. A less consuming alternative to the PoW mechanism is Proof-of-Stake (PoS), proposed for the first time in 2011. It is more ecological and user-friendly with respect to PoW.

The core idea of PoS is that the chances of the nodes to validate the next block are proportional to the node balances in terms of cryptocurrencies. Therefore, the voting mechanism of the blocks changes. While in PoW the miners express their vote through their computational power, in PoS, instead, the validators vote by locking (*staking*) part of their currencies as a warranty. It is no longer a matter of computing capacity, but it is about the amount of money owned and put into play. Specifically, the money put at stake are frozen and cannot be spent, so that the so-called *staker* may lose them if he exhibits a malicious behaviour. For instance, a malicious behaviour could be to not validate a block after being chosen as a staker, or to intentionally induce a fork. The outcome is the same as PoW: the chosen validator checks the transactions and add a block to the blockchain.

One main difference from the PoW mechanism is that in PoS, there is no a block reward (i.e., a special transaction introducing new currency to the network), but only the reward given by the fees.

The process of choice of the validator, as said before, is based on the amount of currency owned, But not only, otherwise, only the richest users would participate in the validation

process and earn fees. Among the various methods have been proposed to face this problem, we present the *coin age* one, described in [11]. The principle is not only taking into account the amount of money, but also its *seniority*, i.e., the period of time in which the currency was kept without being spent.

The coin age is calculated by multiplying the number of coins by the average time during which they have been possessed. For example, 3 EXP (where EXP is a toy name) that have been possessed for 9 days would have a coin age of 27 EXP-days.

The chances to be selected as the next validator are proportional to the coin age of the cryptocurrencies that a participant puts at stake. In this way, anyone can participate in the consensus process, even those who only offer a small amount, providing a major degree of decentralisation than PoW.

Nevertheless, also PoS has drawbacks. It is argued that PoS is not a complete algorithm, because it does not allow to manage the initial distribution of cryptocurrencies. Indeed, this algorithm only distributes as a reward the transaction fees and there is no block reward. The networks that use this algorithm have not adopted it from the beginning, but they had to change the consensus algorithm on the run. For example, the Ethereum network (i.e., the second greatest one in terms of market capitalisation) is planning to move from PoW to PoS by July 2022.

1.4. Types

The current section is devoted to an analysis of the existing typologies of blockchains, from different perspectives. According to the taken point of view, blockchains can be classified as public or private, or - in a transversal manner - as permissionless or permissioned. These two distinctions can be composed and generate more specific kinds of blockchains. Furthermore, hybrid types of blockchains could exist in a sense, which is presented and discussed below.

1.4.1. Public and private

The first presented distinction is between public and private blockchains. This distinction is made according to the access policy of the network. The access policy determines who may read the information kept on the blockchain. A blockchain is then called:

- **Public**, if everyone has read access to the network. In this case, the list of blocks and transactions is publicly disclosed and updated in real-time. All the major public blockchains have a dedicated explorer, which takes care of this process (e.g.,

`https://blockstream.info/` and `https://etherscan.io/`).

- **Private**, if it is managed by some authorities, who have the right to give or take away the access to the participants. Private blockchains are typical of the proprietary networks of those companies which internally use a blockchain to store information.

1.4.2. Permissionless and permissioned

A different distinction is possible, different from the first one, into permissionless or permissioned blockchains. This second classification is made according to the control policy of the network. The control policy determines who may participate in the evolution of the blockchain and how new blocks may potentially be appended to the blockchain. This classification is also valid for the broader class of DLT.

As highlighted in a previous section, blockchain is part of a larger family called Distributed Ledger Technologies (DLT). There are various types of DLT with different architectures and functionalities. As said, the considered classification is based on the network type. Networks in which, in order to access, a user must be registered, identified, and then authorised by some admins or the network itself are called *permissioned*, On the other hand, networks where anyone can access without permission are defined as *permissionless*.

Going back to the blockchain, the choice of the best solution is a cornerstone for the definition of the high-level architecture. Hereafter the strengths and weaknesses of each possibility.

- The use of **permissionless** blockchain is by far the most robust choice in terms of protocol reliability and decentralisation, offering a high degree of transparency and legal validity of its content. In this context, the most popular and recognised blockchain platforms are Bitcoin and Ethereum, which however, present the problem of the high cost of transactions; this makes their use very difficult in production processes that make use of a large number of transactions. Moreover, Bitcoin does not support complex scripts, making it even harder to use. Other permissionless blockchains are emerging in the last period, such as Solana, which offers high transaction execution speeds and very low costs. However, being a new and still little known blockchain, it presents significant difficulties related to the implementation and maintenance of the tools required for the proper functioning of the architecture. These difficulties are directly reflected in the high cost of implementing and maintaining the infrastructure.

- The use of **permissioned** blockchain has the advantage to be more efficient. Since

access to the network is restricted, there are fewer nodes on the blockchain, resulting in less processing time per transaction. In addition, it is possible to manage both the cost of single transactions and the speed of generating new blocks. Examples of well-known permissioned blockchains are Hyperledger and Corda The main drawback of this solution is that the content is not legally recognised and it has a low reliability for public opinion.

In summary, permissionless blockchains are more robust, reliable and transparent than permissioned blockchains; however, they also tend to have long transaction processing times due to the large number of nodes and the large size of the transactions. On the other hand, permissioned blockchains tend to be more efficient, requiring less processing time per transaction.

Moreover, also some hybrid models exist, which are placed somewhere in the middle between permissionless and permissioned.

- The solution adopted by Ripple cryptocurrency, referred as **semi-permissioned**, supports permission-based roles for participants. It allows anyone to participate in the network, but only some to take care of the validation of transactions.

- Another hybrid solution is to use a **sidechain anchored to permissionless**. This solution consists of making use of a reactive and economical sidechain, capable of satisfying production needs without negative effects on transaction costs. The working principle of these types of sidechain is based on the use of specific cryptographic functions, applied to the blocks of the network, which allow the sidechain to rely on a permissionless blockchain (also called mainchain), by certifying entire sequences of blocks on it on a regular basis. This operation, called anchoring, allows these sidechains to significantly reduce time and costs, without renouncing the properties of transparency and legal validity of purely permissionless blockchains. Such a solution is graphically represented by Figure 1.2 below. One of the most interesting sidechains is Polygon. It is an Ethereum-like network that anchors itself directly to the Ethereum permissionless blockchain. This solution allows the development of tools in all respects identical to those that would be created for the Ethereum network, also benefiting from all the properties of the permissionless network to which it is anchored, such as the validity of the content for legal purposes, transparency and the reputation of Ethereum.

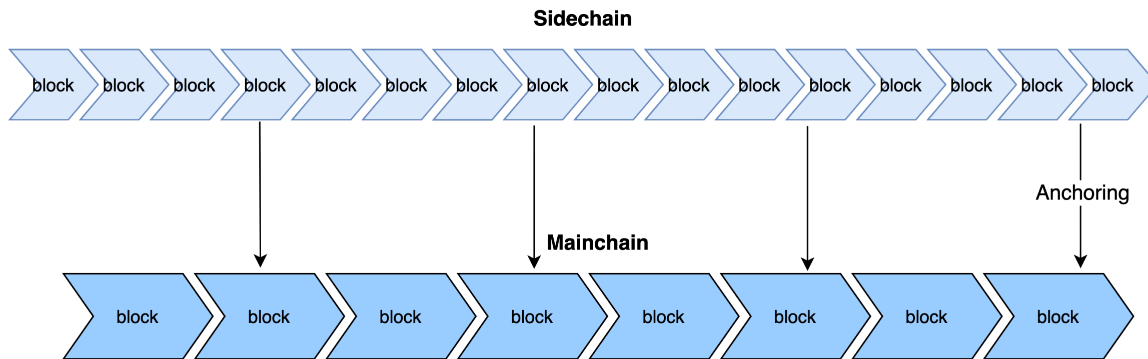

Figure 1.2: Sidechain anchored to mainchain.

1.5. Evolution

This section is about an overview of the evolution of the blockchain. From the ability to only perform mere money transfers, this technology has changed and keeps doing so. Two main milestones included in this section are smart contracts and DApps (Decentralised Applications). Finally, a section is devoted to the ownership certificates represented by NFTs.

1.5.1. Smart contracts

This section presents the smart contract concept, one of the core tools for the purposes of this thesis. If nowadays it is impossible to talk about smart contracts without talking about the blockchain, the origin of the smart contract concept is disconnected and independent from the blockchain.

The term **smart contract** was introduced for the first time by Nick Szabo in 1994. According to Szabo [12], smart contracts are *computerized transaction protocols that execute the terms of a contract*; automatic execution was meant to reduce the dependence on intermediaries in traditional transactions.

However, smart contracts reached a large diffusion with the advent of the blockchain technology. Indeed, features such as security enforced by cryptography and verifiability of the network enabled to code programmable contracts. Smart contracts can be considered executable code snippets of contractual terms, with a public cryptographic key. They operate as autonomous actors: the execution of the code is automatically and independently activated when the contractual terms are satisfied. Their behaviour is predictable, verifiable, and deterministic.

A useful feature of smart contracts is their ability to emit *events*. Events can be used as an asynchronous trigger including data. As a publish/subscribe mechanism, upon the emission of an event, an observer (or listener) is notified and can start the execution of some methods, e.g., on a user interface. Furthermore, events are also useful for logging data in the blockchain, as a form of storage, cheaper than keeping all the state history on smart contract storage.

Smart contracts are triggered by users' transactions or messages that are cryptographically signed and verifiable. Some external entities, called *oracles* are required to record data from the external world and translate them in a digital form. Oracles can be categorised into two main groups, according to the source of information they use: software oracles, which use the web, and hardware oracles, which rely on sensors. Another classification is based on the information flow; in this case the distinction is between inbound oracles, which bring information to the blockchain, and outbound oracles, which take the information from the blockchain.

Smart contracts are supported by different blockchain platforms. Ethereum is the main one and the most known. On Ethereum, smart contracts are typically written in Solidity, a Turing-complete programming language, and compiled into low-level byte-code to be executed by the Ethereum Virtual Machine (EVM).

However, Ethereum is not the only platform supporting smart contracts. Solana, Cardano, Polkadot, Hyperledger Fabric and Tezos are examples of other platforms adopting this technology.

1.5.2. DApps

It is difficult to foresee how smart contracts will evolve in the future. There are already more general forms of applications, relying on smart contracts features, such as DApps (Decentralised Applications) and DAOs (Decentralised Autonomous Organisations).

A **DApp** can be defined as an application running on blockchain, where information is securely protected, and the execution of the operations is spread across all the nodes of the network. Their main advantage is the independence from a centralised server. Typical sectors where DApps are developed include: finance, games, exchange platforms.

The relation between DApps and smart contracts is the following. Usually, the transactions that call smart contracts are generated by DApps and presented to users for further verification before actually being sent to the blockchain. Compared to a conventional application, a DApp is hosted on decentralised storage services like IPFS.

An example of a DApp is *OpenBazaar*, a decentralised peer-to-peer trading network.

According to [13], A **DAO** is *a blockchain-based system that enables people to coordinate and govern themselves mediated by a set of self-executing rules deployed on a public blockchain, and whose governance is decentralised (i.e., independent from central control)*.

In practice, a DAO is an organisation whose activity and executive power are obtained and managed through codified rules, such as automated computer programs like the smart contracts.

Storj is an example of a DAO: it is a decentralised platform enabling peers to buy and sell cloud storage space in return for tokens, with the objective of leveraging a market monopolised by players such as Google and Dropbox.

One of the first examples of a DAO, with an appropriate name, was *The DAO*. It consisted of complex smart contracts running on the Ethereum blockchain that were supposed to act as an autonomous venture fund.

DAO tokens were sold in an Initial Coin Offering and provided a holding fee and voting rights in this decentralized fund. However, shortly after the launch, about a third of the funds was stolen in one of the largest hacks in the history of cryptocurrencies.

The result of this event was the division of Ethereum into two chains following a hard fork. In one, fraudulent transactions were actually reversed, as if the hack had never happened. This chain is what we now call the Ethereum blockchain. The other chain, respecting the "code is law" principle, left the fraudulent transactions untouched and kept the immutability. This blockchain is known today as *Ethereum Classic*.

1.5.3. Fungible and Non-Fungible Tokens

This section clarifies the difference between Fungible Tokens (FT) and Non-Fungible Tokens (NFT).

In general, a fungible asset is one that can readily be exchanged. The most explanatory example is money: two €5 notes are worth the same as one €10 note, since all euros are equal. An asset with equal value is a fungible token. For instance, the native tokens on Bitcoin or Ethereum are FTs, because each BTC or ETH is worth the same amount as the other.

Instead, if something is non-fungible, its value cannot be compared to something else. A non-fungible asset has its own unique properties, that can never be interchanged with other things. A cat or a dog are two examples, or a painting like the Monna Lisa, which

has to be seen to be appreciated. You can take a picture of the cat or the painting or buy another cat or a copy, but they will also differ from the original one.

An NFT on the blockchain is a digital asset that is unique, and it can be purchased and sold like any other digital property or resource, but it has no matching physical form. Examples of digital tokens are certificates of ownership for virtual or physical assets.

Among the advantages provided, NFTs allow people and companies to store real-world information on the blockchain. Information stored can be of any kind, including identity documents, certificates, and real properties. For instance, a diploma issued on the blockchain as a digital document in future could be widely recognised, without the need for it to be translated, notarized, or verified.

1.6. Useful concepts

This section is meant as an overview of some concepts needed in the model description of Chapter 3. The concepts contained in this section do not share any particular characteristics, except from being part of the blockchain technology. The concepts of notarization, relayer and meta-transactions, token economy, contract factory are discussed below.

1.6.1. Notarization

To notarize a document to the blockchain means to guarantee its immutability from a certain date on, i.e., who receives the document can verify that it has not been modified.

How is notarization accomplished? Through hashing procedures. As explained in Section 1.2, a hash function provides a digital footprint for the given input, in the form of an alphanumeric string, which uniquely identify the document. The following is an example of a 256-bit hash.

```
61a496af5f302c678f1324735b568eaf7c45a8effe42f350dd15dc83019d51da
```

So when we notarize a document on the blockchain, we do not actually send the physical document (e.g., the whole PDF document), but a reference to it. That could be, for instance, the hash above.

Why don't we upload the whole document? Because executing transactions has a cost, that varies also according to the size of the transaction data: if the entire document is uploaded, or even an image or a video, then the transaction fees explode. Furthermore,

it is also possible that the file size exceeds that maximum allowed size: in this case, it is concretely not possible to upload that file. In all those cases, by exploiting notarization, the immutability property can be guaranteed.

1.6.2. Relayer and meta-transactions

This section is about a service acting on blockchain platforms, which is a boost to blockchain usability.

Let's consider this scenario. A company wants to put in place a business based on a blockchain platform, to let their customers autonomously trade the company's products. The biggest concern for the company manager would be to remove as much as possible the technical difficulties deriving from the use of a newly-adopted blockchain technology. In particular, the main problem is that of transaction fees.

As also pointed out in the previous sections, the fees deriving from a transaction are highly variable according to multiple factors, and thus unpredictable. It would be desirable for the company to prevent the customer from paying further fees, by including the trading feature into its system.

A service accomplishing this function exists and it is called a relayer. Specifically, a relayer allows users to execute transactions by taking care of the fees. Such a service simplifies the management of the interaction with the blockchain: it deals with fee variability, maintenance and updates of the functional component based on blockchain, lightening the complexity of the blockchain management company-side.

In this way, the variable fees problem is solved: the administrator pays a fixed cost for the service and the latter executes the transactions on his behalf.

From a technical point of view, the relayer exploits a tool provided by the blockchain, the **meta-transactions**. The mechanism is the following:

1. The company sends information about the transaction to be executed to the relayer (sender, receiver, if it is a call to smart contract the name of the function and the parameters);

2. The relayer, based on the received information, sets up the transaction, and wraps it into a bigger transaction, called meta-transaction;

3. The relayer executes the meta-transaction, by taking charge of the corresponding fees;

4. The network recognises the special type of transaction and, reading the payload of

the meta-transaction, retrieves the data about the original one;

5. The network applies the status changes required by the transaction.

1.6.3. Token economy

This section explores the concept of token economy. A token economy is a blockchain solution that exploits smart contracts technology for the emission of fungible tokens. Some smart contracts are developed to create a new token, in which some parameters are specified, like the token name, the maximum supply and usage rules. The rules defined in the smart contracts also determine the minimum and maximum amounts of the eventual punishments or rewards.

In this way, all the parties involved in the blockchain (of any kind, private or public...) can possess and exchange such tokens, which assume a value. The usages for the token are multiple. One possibility is to provide them as a reward for good behaviour inside the chain, or imposing punishments by taking them away upon bad conduct. The tokens can be exchanged for some bonuses given off-chain, as for example, money or a voucher.

1.6.4. Contract factory

Another possible feature of a system based on the blockchain is the development of a smart contract factory. A smart contract factory (or simply contract factory) allows the network managers to define customised rules for whatever specific domain the system is about. These rules are enabled by the deployment of some simplified smart contracts, that make their application automatic.

For example, in an industry context, the rules could be related to the management and integration of agreements between different actors. Or, thinking about the education domain, a system of automatic release of certification upon the completion of a training course.

The idea is that contract development by users of the platform is eased by a simple user interface, which triggers the smart contracts deployed at the beginning. In practice, a platform participant defines rules about specific agreements and these rules are automatically translated and used to deploy a new ad hoc smart contract under the control of the participant, on which he can operate and develop his business.

1.7. Applications

The current section consists of the second logical part of the chapter and includes an overview of the main blockchain applications. Its aim is not to go deep into the details of each application, but only to present them and underline the key theoretical concepts related to each application.

While at the beginning blockchain was mainly applied to the financial sector, nowadays there is a big and continuously increasing number of domains where this technology is exploited. A *systematic literature review of blockchain-based applications* across multiple domains is provided by Casino et al. [14]. The main application sectors identified by this study include: finance, business and industry, privacy and security, education, healthcare, Internet of Things, governance, integrity verification, and data management.

Figure 1.3 below shows such domains of application, also specifying the sub-domains.

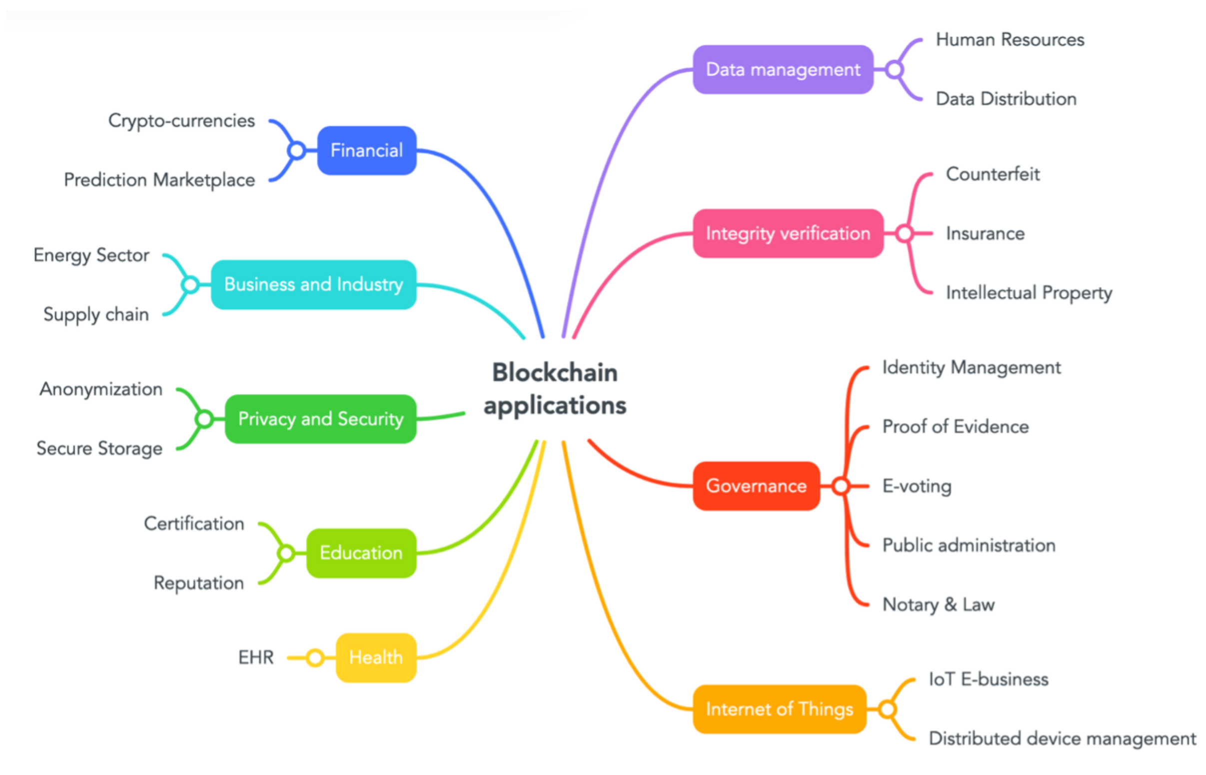

Figure 1.3: Blockchain application domains.

Hereafter, we briefly discuss each of the presented domains.

- **Financial**. This is the domain where blockchain has had historically a very strong impact (Casey et al. [15]). This technology is known for its crucial role in cryp-

tocurrency systems, but is also applied to a large variety of other fields, such as prediction markets, settlement of financial assets, and economic transactions. Even if research in the other domains is running fast, finance is still the most prominent domain of application for the blockchain.

- **Business and Industry**. This is a wide domain, that encompasses all industrial areas. Blockchain has the potential to improve, optimise, and automate business processes. Supply chain management and the energy sector are surely two areas where the blockchain technology can provide a strong impact.
- **Privacy and security**. Blockchain represents an opportunity for enhancing the security aspects of personal and sensitive data. Thanks to encryption techniques data can be stored in a confidential and immutable way. This is a really transversal domain: various domains of application rely on privacy and security properties guarantees given by the blockchain.
- **Education**. Blockchain can be used for educational record, reputation and educational certificate management, improving data security and trust. The release of open certificates through the blockchain, in particular, is currently happening and is a really promising application.
- **Healthcare**. Blockchain technology could play a crucial role in the healthcare domain. In particular, the management of patients' Electronic Healthcare Records (EHRs) can be performed, guaranteeing security and privacy.
- **Internet of Things (IoT)**. This domain is actually a sub-domain of the bigger Industry 4.0 domain, which is not underlined in the previous image. The use of blockchain in the IoT allows decentralised IoT platforms to be implemented, providing secure and auditable data exchange, higher scalability and efficient management.
- **Governance**. The use of Blockchain in this domain aims at providing the same services that are offered by public authorities in a decentralised and efficient way, while maintaining the same validity. Examples of these services include identification management, marriage contracts, taxes and voting. Basic and advanced voting mechanisms can be easily implemented by means of smart contracts.
- **Integrity verification**. Blockchain applications for integrity verification store information and transactions related to the creation and lifetime of products or services. Their main use is about counterfeiting, insurance, and intellectual property (IP) management. The privacy and security domain discussed above is strongly

related to integrity verification.

- **Data management**. The use of blockchain in this application domain facilitates by default auditability, since all of the operations could be verified. Areas such as human resources can benefit from the blockchain technology, which can enhance data storage and selection processes. As the privacy and security field, also this one is horizontal with respect to the others.

For the purposes of this thesis, the most interesting fields of application of blockchain are supply chain, IoT, and - to a certain degree - data management for the verifiability. Let's say something more about the supply chain, the only one not commented upon among those denoted as *interesting*.

In **supply chain** management blockchain technology has the potential to increase transparency and accountability, bringing benefits on visibility and optimisation. Blockchain can be used in logistics to identify counterfeit products, reduce paper load processing, facilitate origin tracking, and enable buyers and sellers to transact directly without intermediaries. In addition, the usage of blockchain in supply chain networks can safeguard security and offer better customer service through advanced data analytics. This application domain is the central focus of this thesis and will be further analysed in the next chapter (Chapter 2).

2 | Blockchain for the Supply Chain: State of the Art

As Section 1.7 pointed out, blockchain applications are countless and spread across a large number of domains. These applications have already been presented from a generic perspective through a brief overview, including the supply chain one.

The aim of this chapter is to focus on the latter domain, by contextualising it and providing an analysis of the state of the art in the supply chain.

The choice to consider this specific sector is strictly related to the topics of the thesis, and it is useful to better understand the characteristics of the model proposed in Chapter 3. Instead, doing the same for all the blockchain applications - given the extent of this technology - would constitute a too extensive work, resulting in a costly and useless deviation from the actual target of the document.

The outline of the current chapter is presented hereafter. First, Section 2.1 is intended to provide some context about the supply chain. It gives the main definitions about supply chain, supply chain management, and digital supply chain. After that, Section 2.2 discusses the state of the art about blockchain apply to the supply chain. Recent research works will be mentioned and a picture is depicted about the current progress on that domain.

2.1. Supply chain

This section provides an overview of what a supply chain is, and touches the notions of supply chain management, and digital supply chain. It is structured as a series of definitions, related to each other. Since in the literature there is no full agreement about a standard definition, we report the most common, considered the most explanatory ones.

2.1.1. Definition of supply chain

The concept of Supply Chain has evolved over the years from an initial phase, characterised by a flow of products to a more mature phase where also services and information are exchanged. This is reflected by the definition provided by the Council of Supply Chain Management Professionals (CSCMP) in August 2013: supply chain is defined as *the material and informational interchanges in the logistical process, stretching from acquisition of raw materials to delivery of finished products to the end user. All vendors, service providers and customers are links in the supply chain.*

Another widely recognised and more complete definition, is given by Coyle et al. [16]: a supply chain is seen as *a series of integrated enterprises that must share information and coordinate physical execution to ensure a smooth, integrated flow of goods, services, information, and cash through the pipeline.* This description not only emphasises the variety of exchanged entities, but also stresses the need to operate in a cooperative and integrated way. The concept of *integrated* network is therefore a fundamental characteristic of a supply chain.

2.1.2. Supply chain management

Since the supply chain is a complex and multifaceted entity, its management is also a challenging task. According to Gibson et al. [17], *Supply chain management is a vital, yet often underappreciated facilitator of trade that fosters customer convenience, business success, and societal development.* Consumers greatly benefit from supply chain management, but they do not pay attention to the processes and activities required in providing them with products and services, although their quality of life depends on productive, efficient supply chains.

In a complex global economy like today's, supply chain management has become a priority for companies and organisations: it is impossible to compete effectively in isolation when suppliers and customers may be on different continents. Supply chain management goes beyond the scope and capabilities of a single organisation and may involve a number of different and heterogeneous players across large geographical areas. Therefore, companies need to develop efficient and agile supply chain capabilities to respond to these dynamic market requirements. In particular, a central aspect - which is debated and could improve with the application of the blockchain - is the traceability of products and processes.

Although this is a recognised need, there is not a consensus among industries and organisations about the concept of supply change management. In the previous paragraph, a

background has been given about such concept; hereafter we try to give a definition.

The more representative definition of this concept is reported by Gibson et al. [17]: *The planning and management of all activities involved in sourcing and procurement, conversion, and all logistics management activities. More important, it also includes coordination and collaboration with channel partners, which can be suppliers, intermediaries, third-party service providers, and customers. In essence, supply chain management integrates supply and demand management within and across companies.*

Organisations must plan and coordinate activities among their network of suppliers and customers to ensure that the end product is available in a timely, safe, and cost-efficient manner. Supply chain management involves managerial efforts, that can be both strategic or operational, by the organisations within the supply chain for the fulfilment of customer demand.

2.1.3. Digital supply chain

In the last years, we have observed a transition from the traditional supply chain to a digital supply chain. Two driving forces are responsible for this evolution: on the one hand, the considerable advances in ICT (e.g., Big Data, Cloud Computing, Blockchain, Internet of Things, Augmented Reality technologies); on the other hand, the increased customer expectations. As highlighted by Alicke et al. [18], this concerns additional service expectations, a much stronger fragmentation of orders, and further individualisation. In addition, the online-enabled transparency and the availability of a multitude of shopping and goods options influence competition in supply chains.

A comprehensive definition of the concept of digital supply chain is provided in [19] by Büyüközkan & Göçer: *A Digital Supply Chain (DSC) is a smart, value-driven, efficient process to generate new forms of revenue and business value for organisations and to leverage new approaches with novel technological and analytical methods. DSC is not about whether goods and services are digital or physical, it is about the way how supply chain processes are managed with a wide variety of innovative technologies, e.g., unmanned aerial vehicles, cloud computing, and internet of things, among others.*

The digitisation of the supply chain enables companies to address the new requirements of the customers and brings a number of benefits to industries [18]: reduced delivery time, increased flexibility (thanks to ad hoc and real-time planning), increased granularity, increased accuracy, increased transparency, increased efficiency.

2.2. Blockchain for the supply chain

This section contains an in-depth analysis of the state of the art regarding the application of blockchain to the supply chain. First, a non-technical overview is provided of some solutions present on the market. Then, the discussion takes a more specific approach: starting from the limits of standard supply chain models to the recent solutions based on the blockchain, supported by some research papers.

2.2.1. Introductory examples

Blockchain solutions are already being used in supply chain management. However, this application is still in its early stages: some projects are working, but some of them do not leverage the features of blockchain as they could. Following, we provide two examples, just to give a little view of how this technology can be concretely employed in the supply chain domain.

In the agrifood industry, for instance, the brand of supermarkets Carrefour employs a blockchain-based solution to trace the food distribution chain, and to give consumers information about its products [20]. However, the solution is still in an experimental phase: only some kinds of food are traced, and - even when it will be completed (the expected deadline is by 2022) - only the food products of the specific FQC *(Filière Qualité Carrefour)* chain will be traced.

The pharmaceutical supply chain is another example: a permissioned blockchain is used by two hospitals in the UK (in Warwick and Stratford-upon-Avon) to monitor the deployment of the COVID-19 vaccines. In particular, it allows to monitor the storage temperature of the vaccines batches, by sending alerts if it goes beyond a certain threshold [21]. Nevertheless, the application is not really distributed, since it is used internally by a single organisation.

It is possible to employ the blockchain technology also in other ways, closer to its decentralisation paradigms, as explained in the next section.

2.2.2. State of the art

This section presents the state of the art about the blockchain applications to the supply chain. To do so, it starts with the limits of legacy methods for supply chain management, and gives some examples of blockchain-based solutions. The benefits provided will be highlighted, as well as the remaining limitations.

As specified by Gaur and Gaiha [1], the problem of the current supply chain is that the standard methods for supply chain management do not keep up with the fast development of technologies and increase in the requirements.

The ERP (Enterprise Resource Planning) systems, that in the past brought innovation so far, are no longer adequate to fulfil the increasing needs of globally competitive supply chains. The reason is that there is no unified network or platform to connect different firms' ERP, regardless of their location and domain.

What prevents such a network to be implemented are economic matters. The ERP systems, indeed, are from various companies in competition with each other. Each ERP is customised for a specific sector and customer, in order to provide the best functionalities. Moreover, these solutions are proprietary and, being their source code not publicly disclosed, it is hard to reach a communication standard. This restrains a standardised integration of business processes, keeping consistent discrepancies along the chain.

In the new supply chain scenario, collaborative and frequent transactions are ordinary. A tool is needed for integrating the different supply processes, going beyond the limits of the rigid interactions among participants and opening up to brand-new supply chains. In such a scenario, blockchain technology is likely to pay an enormous contribution.

Among the innovations that blockchain could bring, there is minimising third party intermediation and related costs. The decentralisation characteristic of blockchain allows for big data integration and supports sharing among different participants. Such an approach reflects several benefits: enabling faster and more cost-efficient delivery of products, enhancing product traceability, avoiding product fraud and counterfeiting, improving interoperability and coordination between all supply chain members.

According to Gaur and Gaiha [1], the use of blockchain in the supply chain context provides a complete, transparent, trustworthy, tamper-proof history of the information flows, inventory flows, and financial flows in transactions. For instance, unique identifiers are assigned to assets such units of inventory, loans and orders, and serve as tokens that are transferred from one participant to another during a transaction; participants are also assigned unique identifiers, which allow for a sort of digital signatures used to sign the blocks they add to the blockchain.

The blockchain provides added value since it encompasses a chronological string of blocks that integrate the three types of flows in the transaction, capturing details that are not recorded by standard financial-ledger systems. In addition, each block is encrypted and shared among all participants, who keep their own individual copies of the blockchain,

making it possible for all parties to review the status of a transaction and identify errors. In this way, the blockchain solves part of the issues emerged in the traditional digital supply chain, e.g., traceability and coordination problems.

The use of the blockchain can also facilitate financial assessments: for instance, based on the transactions encoded in the blockchain, a bank can take lending decisions without having to conduct physical audits and financial reviews. The blockchain can include lending data together with data about invoicing, payments, and the physical movement of goods, making the transactions more cost-effective, easier to audit, and less risky.

Smart contracts can play a critical role in order to automate some of these functions, because they can be programmed to automatically take actions (e.g., releasing a payment) when specific conditions are met. It is worth noting that a blockchain would not replace the broad range of functions performed by ERP systems (e.g., invoicing, payment, and reporting). Its main role is to facilitate the integration of various flows of transactions across firms, by interfacing with legacy systems across participating firms.

To show the benefits the blockchain can bring, some application contexts are presented ([1]). In the pharmaceutical sector, the blockchain can be used to identify and trace prescription drugs. As required by USA legislation, pharmaceutical companies have to identify and trace prescription drugs to protect consumers from counterfeit, stolen, or harmful products.

A large pharmaceutical company decided to use the blockchain for this purpose. Specific codes are used to tag units in the drug inventory. When an item flows in the supply chain from its source to the consumer, at each step, its tag is scanned and recorded on the blockchain. This technology is successfully tested by the company in the USA and then extended to other parts of the world. The information shared is minimal and does not include data about purchase orders, invoices, and payments; this entices companies that are cautious about sharing competitive data to participate.

Other companies follow a similar approach in other sectors: for example, IBM in collaboration with Walmart, creates a safer food supply chain (IBM Food Trust), by using the blockchain for tracing fresh food and other food products [22]. In case of a damaged product, the blockchain enables the company to trace the product, identify all suppliers involved with it, and efficiently recover it. The blockchain can also be used to help identify counterfeit goods, because these kinds of products would lack a verification history on the blockchain.

Another example, in the manufacturing sector, is represented by Emerson, which uses

the blockchain to increase supply chain efficiency and visibility. Emerson, a multinational manufacturing and engineering company, is featured by a complex provision system across many suppliers, customers, and locations. A small disruption in any part of the supply chain could have consequences on the others, leading to extra stock out in other parts.

Blockchain represents a practical solution to these challenges: it could be used to share the inventory flows of the participating companies, allowing each company to make its own decisions, using shared and complete information.

The use of the blockchain to increase the efficiency of the supply chain also is envisaged by other companies, such as Hayward, a multinational manufacturer of swimming pool equipment, and Walmart Canada that began using blockchain with its inventory trucking companies.

Furthermore, some solutions exist in the context of tracing products devoted to sustainable marketing technology, aiming to reduce the environmental impact, or give assurances about the working conditions of the manufacturers. For instance, Provenance is an organisation operating in such a context, sustaining the use of blockchain to enforce supply chain traceability [23]. Anyway, because of the high energy consumption derived from public blockchains, Provenance does not broadly employ them. The use of blockchain is very conservative: it is limited to recording third-party verification of proof points, where a decentralised approach adds unique value to the brand and shopper.

However, all the examples seen have some limits. Some of them are applied to specific sectors or even products; others are based on permissioned or private blockchains and therefore not decentralised; others are still in the experimental phase. In other words, the currently present solutions are only partial or do not fit the blockchain paradigms.

As Jabbar et al. sustain [24], there is still a large number of challenges before getting to an implementation of the blockchain completely reflecting its academic definition. Among them, there are scalability and interoperability. Scalability is the ability of a system to keep working after increasing the size of the input. Traditional public blockchains do not scale well in terms of speed (transactions per time interval), cost of the fees and time for producing a block. Interoperability refers to the communication capabilities among different blockchains. It can bring to a more connected world, in which it is easier to exchange information.

The limitations to blockchain application to the supply chain are following summarised. There is no general framework that can adapt to different sectors and constitute a standard, but there are only specific ones (e.g., the IBM solution is only for food chains). The

usability of the blockchain is still a great obstacle for non-experts in the industry; in fact, several solutions realised lie on permissioned platforms, which can be better controlled, and do not fulfil the blockchain decentralisation.

The information stored on-chain regards, for example, the tracking of products, or the temperature of conservation. But it is only about one specific aspect: there is not a platform that is transversal to the supply chain and allows to completely trace raw materials, processes, products and how the first two contribute to the realisation of the last.

It is a task of this work to propose a model that solves some of the above-mentioned limitations, by involving innovative solutions. The description of such a model will be the subject of the next chapter (Chapter 3).

3 | The Model: Blockchain for Supply Chain Control Systems

This chapter is the **core** of the thesis. It contains the main contribution made, namely, the description of a platform, based on the blockchain technology, that can be used to manage supply chains across different sectors.

After having analysed of the state of the art, presented in Chapter 2, it is clear that a unifying standard solution does not exist in this field. However, it is possible to exploit the benefits of blockchain technology to implement a platform that could handle different supply chains. One the one hand, the use of blockchain brings improvements (e.g., the tracking process becomes faster and more efficient); but on the other hand it poses a large number of challenges (e.g., how an ordinary person, inexperienced in the sector, can interact with the blockchain).

The aim of this chapter is to give a precise definition of the system from an engineering point of view. The innovative aspects will be pointed out, as well as the critical points. The former will be addressed in a specific section. The latter will be discussed and complemented with alternative ways to deal with them. In particular, one feature that deserves a remark is usability, understood as the whole user experience - in terms of ease of use and satisfaction - towards the system.

The current paragraph gives an outline of the chapter. The first section (3.1) provides a description of the above-mentioned system as a whole and its components. The description starts with a brief overview of what we want to accomplish with the system, then follows a top-down approach, from a general scheme to a more detailed one, including a dedicated part for the blockchain. For each component, its role inside the system is clarified. The second section (3.2) presents the usability aspect and some concerns related to this specific blockchain scenario. An overview is provided of the many challenges involved in the design process. Finally, Section 3.3 points out the innovative aspects that emerge from the model description.

3.1. System description

The current section describes the system in general, from a technical point of view. The required components are introduced, as well as their interaction. After a general architectural view, the focus moves to the specific components. The blockchain part, because of its relevance, is addressed in detail into a dedicated section (3.1.2). At the end, some possible enhancements are discussed, for example related to the Internet of Things.

The main **idea** behind the system is to employ the blockchain in the supply chain domain to implement a modular traceability service, that should also be user-friendly. Following, a model is reported that fulfils such an idea and tackles at best the many challenges posed.

The solution should be able to integrate into multiple supply chains, also from different sectors, and improve their efficiency. In particular, it should be possible to write to the blockchain data about processes occurring in the supply chain and products handled. Afterwards, it should be straightforward to read such data with the guarantee that there was no change. The use of blockchain boosts the tracking capabilities of the system and enforce security against counterfeiting.

The system realising all the requirements mentioned above can assume different forms. The one chosen and proposed in this thesis takes the form of an application, accessible from browsers and also mobile devices, on which the supply chain participants can write information about that. The application must allow to do it in an intuitive way, without or with a minimum training process. The information to be stored should range from raw materials, to transformation and processes, to the finished products.

On the other hand, another application (or another view of the same application) should be accessible by the product end users. The latter, after having bought a product, may want to consult data about the processing phase of that, in a fully transparent fashion. Also this side must guarantee a high ease of use, not to prevent the users from accessing the information. For example, if she is interested in the origin of the raw materials used to prepare the medicine prescribed to her son, she must be addressed to a screen displaying the specific information required, with the guarantees that they are correct.

A main innovative aspect of the system with respect to the present ones is its *flexibility* and *versatility*. While some blockchain solutions exist for some products in the agrifood chain, it does not exist a single platform that could handle both of them together. The proposed system is flexible enough to handle chain from both chains, but also more, as for instance, the automotive or cosmetics sectors. Furthermore, it is meant to trace the whole supply process, from the origin up to the delivery to the retailer, including the

processes carried out in the middle, as the transport.

The role of the blockchain in such a system is to improve the speed of the tracking process, also decreasing its costs. The speed is increased, for example, by making the retrieval of data immediate. Indeed, data shared on the blockchain distributed ledger can be directly read by the participants. In turn, this involves a lower time in stock, cutting down costs. However, the property that participants can read data on the Ledger is easily implementable using any database and it is not peculiar of the blockchain.

Another important function played by this technology and distinctive of the blockchain is to enforce transparency and security. The data stored on-chain are immutable and their correctness can be verified by anyone. Also, blockchain certifies data authenticity. The term authenticity in the communication between two actors expresses the fact that the receiver has sufficient evidence to make him believe that the message actually originates from the indicated sender. The public key mechanism - on which the blockchain relies - provides such a property.

In particular, blockchain is exploited for notarization. As explained in Section 1.6, to notarize - e.g., a document - means to guarantee that the document is immutable from the notarization date and time on. We remark that, with this technique, only a hash is loaded on the blockchain and not the full document. In this way, users can exploit the blockchain to enforce immutability and transparency about supply processes and products. Finally, notarizing on the blockchain is also a way to overcome the heterogeneity of the old supply chains, based on multiple different customised solutions.

A major limitation of the usual blockchain platforms is scalability. The transaction cost is high and greatly variable and a widespread user base implies a non-affordable amount of fees. This model proposes an innovative solution to this problem, both for decreasing the fee price and for managing their variability. By means of an ad hoc blockchain architecture, high *scalability* and a *fixed cost* of transactions is achieved, as specified in Section 3.1.2 and 3.2.2.

3.1.1. Component architecture

This section focuses on the architectural description of the system and its components: starting from a general view and decomposing it in a progressive way. The highest-level scheme of the architecture, which is as generic as possible, is shown in Figure 3.1.

Before the actual description of the architecture, we provide a notation remark about the distinction between system and platform. By *system*, we mean the presented solution as a

whole including the blockchain part, which is detached from the platform. The *platform*, instead, is the subset of the system including the front-end and the back-end, which allows for the supply chain management. It can be thought of as a web application, accessible from the user and communicating with the blockchain through API calls. This distinction is made in order to ease the model description; it is not meant to be applied outside this work and is better clarified by Figure 3.1, presented in the next section.

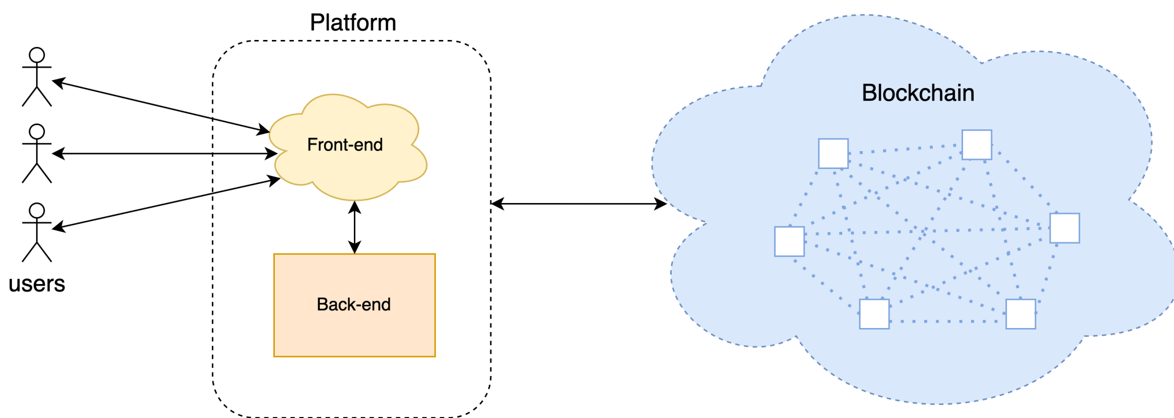

Figure 3.1: Most general architectural view of the system.

From left to right, we can identify three main entities:

- Users,
- Platform,
- Blockchain.

Following, we give a brief overview of each of them and of the links between each others.

The **platform** can be in the form of a web app, a desktop app or a mobile app. It is internally divided into a back-end and a front-end part, where the latter allows access to the users. The **users** can be insiders, but also common people interested in consulting information about the supply chain. The **blockchain** is one of the technologies used by the platform: it allows to store certified data about the supply chain processes and products, and to consult them subsequently. The link between the platform and the blockchain takes place through library or API calls.

It is desirable for the platform to reside in the cloud, not only for the front-end but also for the back-end. This is not strictly required for the system to work, but it is necessary in many real-world cases. A cloud platform ensures, among the others, complete scalability

and continuity of service. The allocated resources, in terms of computing power and storage, may vary according to the number of users, allowing to handle an arbitrarily large amount of users.

Now, the high-level architecture presented above is enough to introduce a first problem. In Chapter 2, we explained that a problem is the presence of actors controlling access to the system. The architecture proposed involves that the blockchain is accessed through a platform, which constitutes a centralised application.

Given that, a question arises: is it not true that this centralization affects the remarkable properties of the blockchain? The answer is that it is true, the presence of a centralised platform could be a limitation for the model decentralisation. This is a relevant concern and is specifically addressed in Section 3.2.3.

However, the degree of detail provided is not sufficient to properly depict the system characteristics. In the next paragraph, we further investigate each of the components about their features and roles.

The architecture of Figure 3.1 is useful to give an insight of the system, but it is way too general to adequately describe it. The objective now becomes to go into details on the various components and address questions like what are the tasks reserved to the back-end, or would a link to the blockchain as simple as the one represented work in a real-world scenario.

In order to do that, we need a more exhaustive view of the architecture, as the one provided in Figure 3.2 below.

The new scheme breaks down into smaller components the two main parts presented at the beginning: the platform and the blockchain (users are not decomposed since they are already atomic). Also, the link between the two parts is better specified.

The platform, partially decomposed already before, is made up of:

- Front-end,
- Back-end,
- Database (DB).

The blockchain part, instead, actually consists of two distinct blockchains:

- Sidechain,
- Mainchain.

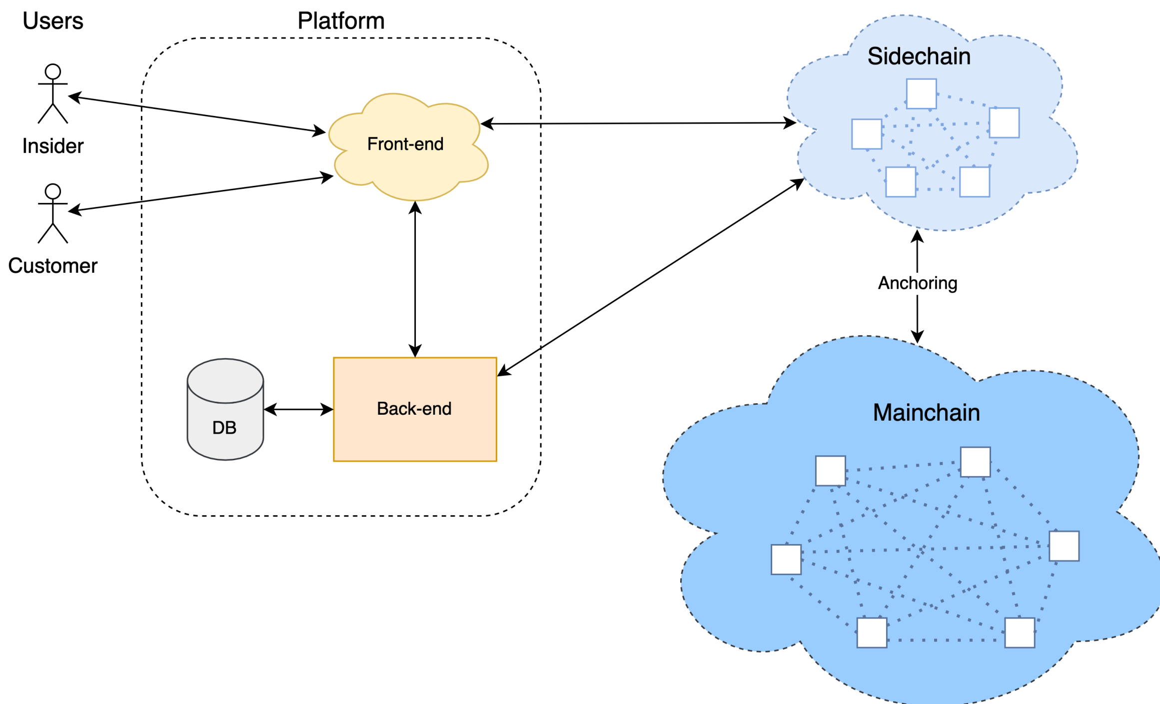

Figure 3.2: Component view of the system architecture.

Let's now analyse the components one by one, better exploring the usage of each.

The **front-end** provides the application interface: it includes all the elements that are visible to the users. The interface can be different according to the user types. For instance, the storekeeper of a supermarket will not visualise the same interface as the customer: the former will probably see a form to notarize the delivery of the goods; the latter a QR code reader and then a page with information about the product he just bought.

The front-end allows to keep data client-side, without publicly sharing them: as long as they are not explicitly submitted, it is possible to safely manage private and sensitive data. For example, MetaMask - a well-known browser plugin that act as a cryptocurrency wallet - stores the user's private key in the browser in a reliable manner. Similarly, no private information from the back-end passes through the front-end if not meant to.

From an implementation point of view, the main languages employed are HTML (for the content), CSS (for the design) and JavaScript (for the interactivity).

Strongly-related to the front-end is the **back-end** of the platform. It is essential for the proper functioning of the system and includes everything that operates behind the

scenes. It takes care of carrying out the actions performed by the user through the front-end. It is a task of the back-end to manage users' authentication, exchange data with the database, trigger the execution of a code snippet upon the occurrence of an event. The snippet can be a function accomplishing a task, or a call to the blockchain (done through a known library or an external third-party API). The implementation does not have a standard language, it should be done with an object-oriented multi-platform language (e.g., JavaScript, or Python are two frequent choices).

A component first introduced in Figure 3.2 is the **database**, an organised collection of data that can be stored on a single location or on multiple physical locations. It is worth to highlight the distinction between the two solutions, i.e., between a centralised and a distributed database.

A centralised database stores information in one single location. Authenticated users from any location can easily access it, and no other user can without authorisation. This solution guarantees that the data stored within the database are secured. Contrary to a centralised solution, in distributed databases data are distributed among multiple locations, interconnected through communication links. Given that, the best solution for the system is to adopt a distributed solution, that guarantees a higher scalability in terms of data access speed.

Moving to the blockchain part, the theory behind this architectural choice is explained at Section 1.4, here we remind the main idea. There are two blockchains: the **mainchain**, a public permissionless blockchain, and the **sidechain**, not restricted to a specific type. This use case requires the sidechain to be public and permissionless as well. The data are not directly stored on the mainchain, but on the sidechain, which is *anchored* to the former. More details are provided in the next section (Section 3.1.2).

As far as the **link** between the platform and the blockchain is concerned, Figure 3.2 specifies it. As shown, both the back-end and the front-end of the platform are connected to the sidechain component. In particular, the connection with the front-end requires a remark. It is true that the front-end calls modify the blockchain (precisely the sidechain) state, but actually, such the link is a shortcut. The communication goes through an intermediate step: the operations made by the users on the front-end trigger a back-end function, which in turn involves a change in the blockchain.

3.1.2. Blockchain and smart contracts

This section is devoted to deepen the blockchain component of the architecture, by adding further details with respect to what already stated in the previous ones. Particular atten-

tion is dedicated to the smart contracts. Figure 3.3 below gives the necessary additions.

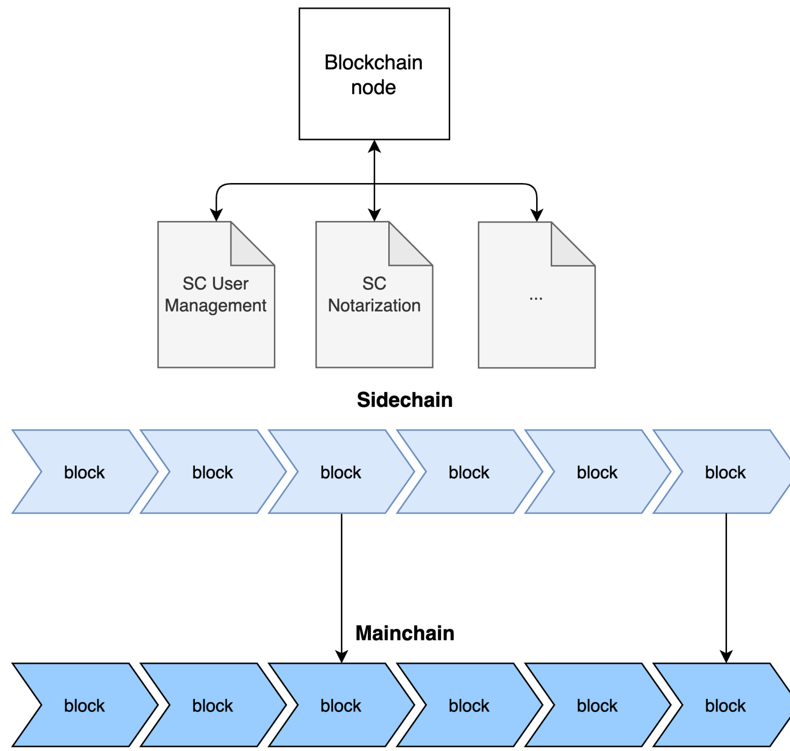

Figure 3.3: Detailed architecture of the blockchain part of the system.

Besides the sidechain and the mainchain (i.e., a public permissionless blockchain), the scheme shows a generic blockchain node and the smart contracts (SC). The blockchain node acts as an intermediary between the platform and the **smart contracts** on the sidechain. The latter are stored in each node and contain the code executed by the blockchain validators in order to change its state through a transaction. For the purposes of the current model, two core functions must be accomplished by smart contracts: user management and notarization (i.e., certification that the supply chain data are authentic, as specified before and also in Section 1.6).

Such functions could be implemented in various ways. The most linear solution is to allocate each function to a smart contract, thus obtaining two of them: the user management contract and the notarization one. Thus, the latter would be related (i.e., linked) to the former, in order to regulate access control and let only the authorised users make certain operations. This solution is the preferred one and it is represented in Figure 3.3.

An alternative solution could be to implement both user management and notarization in the same contract. This is feasible, but from a code readability perspective it is preferable

to split the functions into separate contracts. Moreover, if a smart contract is too big and exceeds the contract size limit, it cannot be deployed to the blockchain. Later on in the description, other smart contracts will be introduced. This also applies to those contracts: each new feature is assigned to one or more distinct contracts.

Regarding the user management contract, it must handle different types of users. Beyond the customers, not relevant because they only perform reading operations, another distinction is needed. The role and capabilities of the platform manager should be different from those of the supply chain users, able to notarize.

We just said that data are notarized to the blockchain. An matter that could be unclear is the following: what is the point of having a separate DB if data are saved on the blockchain? This is a *crucial aspect* and deserves an extensive explanation. As mentioned in Chapter 1, transactions have a cost. Such a cost varies according to the type of transaction and the execution time and storage space required. For example, a simple token transfer requires a cost being as low as possible, while running a smart contract function requires a larger one. In particular, the cost is as high as the data structure involved is complex and as the computation time is longer.

For this reason, it is very expensive if not impossible to store the entire data pool of each supply chain directly on the blockchain. The established solution in this field is to minimise the data loaded on blockchain, while not renouncing the properties guaranteed by this technology. The idea is to **notarize**. Such a term refers to the certification that the data are correct and not counterfeited. The latter is checked through the hash, which are the only information - in addition to those related to the addresses of users - uploaded directly to the blockchain. The hash then is used as a key to a specific database entry, where the information are stored in full.

For what concerns the employed blockchain platform, it could be the Ethereum ecosystem, currently the most famous for developing smart contracts and decentralised applications. In particular, the model is the *sidechain anchored to permissionless* one, described in Section 1.4, and the **mainchain** is the Ethereum blockchain. Among the different possibilities, the **sidechain** of Polygon, is an efficient one, since it offers low transaction fees, extremely high transactions per second and scalability. Furthermore, we recall that the anchoring to a well-established mainchain as Ethereum rises the security level and has a legal value to certify data authenticity.

In this way, several positive aspects can be appreciated.

- The development of smart contracts and means communicating with the blockchain

can take place using tools and languages of the blockchain Ethereum, given the nature of this sidechain. The development environment is more consolidated than competing blockchains, thus reducing both programming and maintenance costs.

- The fees that must be incurred for the correct execution of a transaction are lower compared to Ethereum and other similar solutions, which, however, are less known and more difficult to maintain.
- The intrinsic structure of Polygon involves anchoring to the permissionless network of Ethereum. Thanks to a series of cryptographic functions, the entire content of the sidechain in question is reported on Ethereum and is fully recoverable and demonstrable, despite a full transposition of all transactions is not made. In this way, the total operating costs of the network are further reduced, while still being able to benefit from the important properties of transparency, popularity and legal value enjoyed by the Ethereum network.

Considering the lower part of the scheme, only a snapshot of the two chains is represented. Its main value is to show the anchoring "in action": the sidechain, after producing a certain amount of blocks, notarizes a prove on the mainchain, thus achieving the good properties of the latter.

Up to now, we presented the minimum features in order to obtain a working model. However, multiple features can be added in order to complement this core part and enhance the model functionalities.

A feature that widens the application perspectives of the system is the implementation of a **token economy**. In order to do it, it is necessary to exploit smart contracts technology for the emission of fungible tokens. Some smart contracts are needed to create a new token. Parameters like the token name, the maximum supply, and usage rules are specified inside the contracts. Thus, a token economy for the different supply chains is generated. The aim is to encourage desirable behaviour, by offering rewards or imposing punishments in terms of tokens. A good behaviour is, for example, the delivery on time or in advance, while a bad one could be the missing or late delivery, or the transport damage. The tokens can be exchanged for some bonuses in the supply chains, as a discount. The rules defined in the smart contracts also determine the minimum and maximum amounts of the punishment or the reward.

Another useful blockchain feature of the system is the development of a **smart contract factory**. A smart contract factory (or simply contract factory) allows the supply chain administrators to define customised rules for the supply chain management. These rules

are enabled by the deployment of some simplified smart contracts, that make their application automatic. They are related to the management and integration of the agreements between different actors, in the context of the supply chain. An example is the automatic payment of fines.

Contract development by the supply chain administrator is eased by a simple and intuitive user interface. In practice, they define rules about the supply chain agreements and these rules are automatically translated and used to deploy the new smart contracts on which the platform will operate.

The following paragraphs cover the **Internet of Things** topic. Although it is not enabled by the blockchain technology, the IoT properties well combine with the blockchain, as described in Section 1.7. IoT sensors could be used in a supply chain, in order to record data about products and processes. The data that can be supplied are the most diverse: for instance the temperature of the goods, the completed shipping after arriving at the destination location, the sale of a product.

The implementation of specific smart contracts allows to manage IoT devices in the supply chains. Inside them, information is stored about gathered data, device model, serial number and read/write authorizations. Furthermore, the information could be encrypted in order to enforce privacy and confidentiality. The upload process to the blockchain can be made automatic: the application logic of the IoT device, albeit small, is sufficient to implement some API calls to perform the notarization. It is also possible to realise a trigger in the back-end to call the smart contract method with almost no effort.

To sum up, the complete model requires a richer smart contract infrastructure, represented in Figure 3.4.

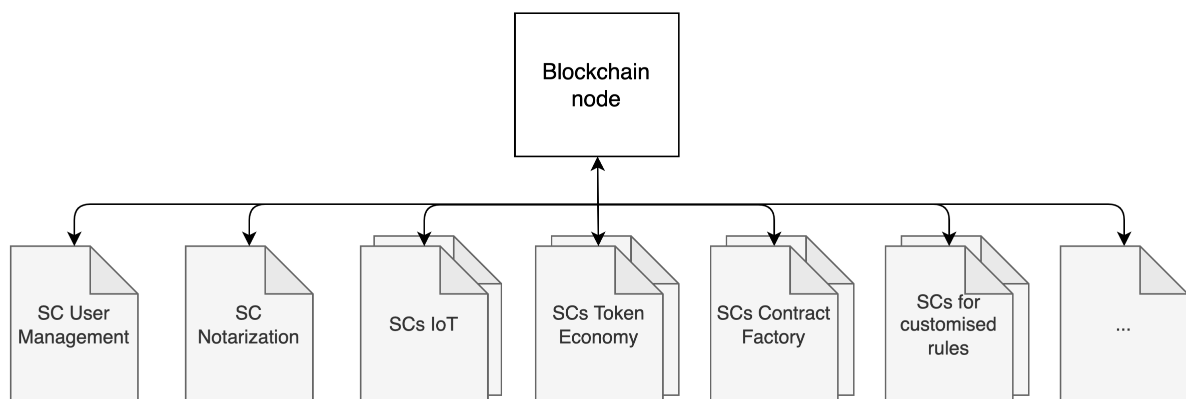

Figure 3.4: Complete smart contract infrastructure.

The infrastructure above requires at least the following smart contracts.

- A smart contract for user management,
- A smart contract for notarization,
- A smart contract for IoT devices integration,
- One or more smart contracts implementing the token economy,
- One or more smart contracts implementing the contract factory,
- An undefined number of smart contracts realising user-defined customised rules.

3.2. Usability concerns

A key aspect for massive blockchain adoption is usability. Nowadays, the low usability is the biggest obstacle that prevents common people from using such a technology. Because of its criticality, it is important to devote this section to discussing the problem and its implications for the presented model. Some possible solutions or ways around the issues are provided.

According to the ISO definition, usability is *the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use* [25]. Therefore, it also refers to methods for improving ease-of-use during the design process. So, in this context, usability is about developing an intuitive and user-friendly front-end interface, but also other crucial aspects such as cryptographic key management.

From this perspective, the development of the system poses many challenges. First of all, to be concretely adopted, it must not be a disruptive change with respect to legacy systems (i.e., the existing supply chains); otherwise no one could learn how to use it.

The latter problem is a standard one: it concerns all the newly-developed systems that must be integrated into the current ones. The proposed model consists of an application (web or mobile) that allows users to perform all the required operations inside it. An application is something that users know and can easily accept.

However, there are a lot of blockchain-related issues that have not been considered so far and, if not properly handled, can cause faults or even prevent the system from working at all. Following, we deal with them.

3.2.1. Non-expert users

A remarkable issue that must be addressed is the interaction between non-expert users and the blockchain. Indeed, most of the target users for the platform do not know anything about the blockchain.

This has implications on the handling of **cryptographic keys**. The private key of a user is at the basis of executing transactions, and allows to have access to the owned tokens. If the private key is lost, then the tokens cannot be employed anymore, since no transaction can be sent from the owner. To avoid the loss of the key, several advises are shown during the wallet generation phase, to stimulate the users to save the key (or the recovery seed phrase) offline or on a sheet of paper.

However, for many users it is not enough to guarantee the key preservation. For this reason, in some real-world cases the wallets are custodial, i.e., wallets in which the private key is held by a third party. In case of adoption of a custodial wallet for the system, the role of the third party is played by the administrator. Anyway, the choice between a custodial and a non-custodial wallet is anything but obvious, and must be well thought-out.

A useful feature of the blockchain technology is data immutability. Yet, it may be a double-edged sword. A distracted user, indeed, could perform the notarization of **incorrect information**. A simple solution to this problem is to make the notarization not immediate, by adding an intermediate step for confirmation. This can be directly managed by the front-end interface: before notarizing, the data switch to the *waiting for approval* state. In a second moment, the same user, some authorised users or only the administrator, after checking for correctness, trigger the notarization. This is an barrier to ease-of-use, but it is necessary to improve the correctness of data.

Another issue that slows down the massive blockchain adoption is the payment for transactions. How could a non-expert user, with no knowledge of wallets and transactions, possibly get the necessary tokens? This problem is not easy, and the simplest way to solve it is to take charge of the users' wallets and handle the blockchain part on their behalf. However, this is in contrast to the blockchain decentralisation idea. Therefore, another way must be found. A more elaborate solution is the one proposed in the next section (3.2.2) that, though thought for another problem, fits this one as well.

3.2.2. Transaction fees

From the system administrator point of view, a big concern comes from the blockchain **transaction fees**. As also pointed out in Chapter 1, the fees deriving from a transaction are highly variable according to multiple factors, therefore they are unpredictable. This is a problem because too high fees could cause the system owner to go over budget and he may not afford to do it.

A possible solution could be to take the transaction fees into account as a variable cost, for instance by fixing an expected and a limit cost for them. This solution is not always available since the fees fluctuate a lot. If the budget is not enough to cover costs during a peak, it is not feasible.

Let's consider, for example, the price of gasoline (the example would also work for the price of the properties, or all the assets with a variable price). The choice of purchasing or not those assets depends on how much their price varies. If the price of the gasoline last week was worth - let's say - 1.80, this week is 2.50, the next one 6 and the one after that 20, it is likely that the owner of a car finds an alternative way to reach his workplace. For instance, he could take transport means and make a subscription for them. The problem is that the variation would be too wide for him to be able to afford it. Thus, finding a way to smooth it out, like the transport subscription, could be a solution.

The reasoning for the blockchain fees is analogous: since the cost variability is too much, a way must be found to stabilise it. And the solution is exactly that: subscribing to an external service, and relying on it for the execution of the transactions and the payment of the fees.

Such a service simplifies all the management of the blockchain interaction: it deals with fee variability, maintenance and updates of the functional component based on blockchain, lightening the platform tasks. In this way, the variable cost problem is solved: the administrator pays a fixed cost to use the service and the latter executes the transactions on his behalf. This is the principle of meta-transactions, explained in Section 1.6, and the external service provider is called a **relayer**. This solution also solves the problem of provisioning the users with tokens for the transaction, since the fees are on the relayer.

We remark that the employment of a relayer service represents an innovation with respect to the state of the art, since it has not already been employed in this domain.

Since in the last sections we introduced some further architectural components, it is worth to put together all the partial views, provided during the discussion. To this end, a comprehensive scheme is provided in Figure 3.5 below.

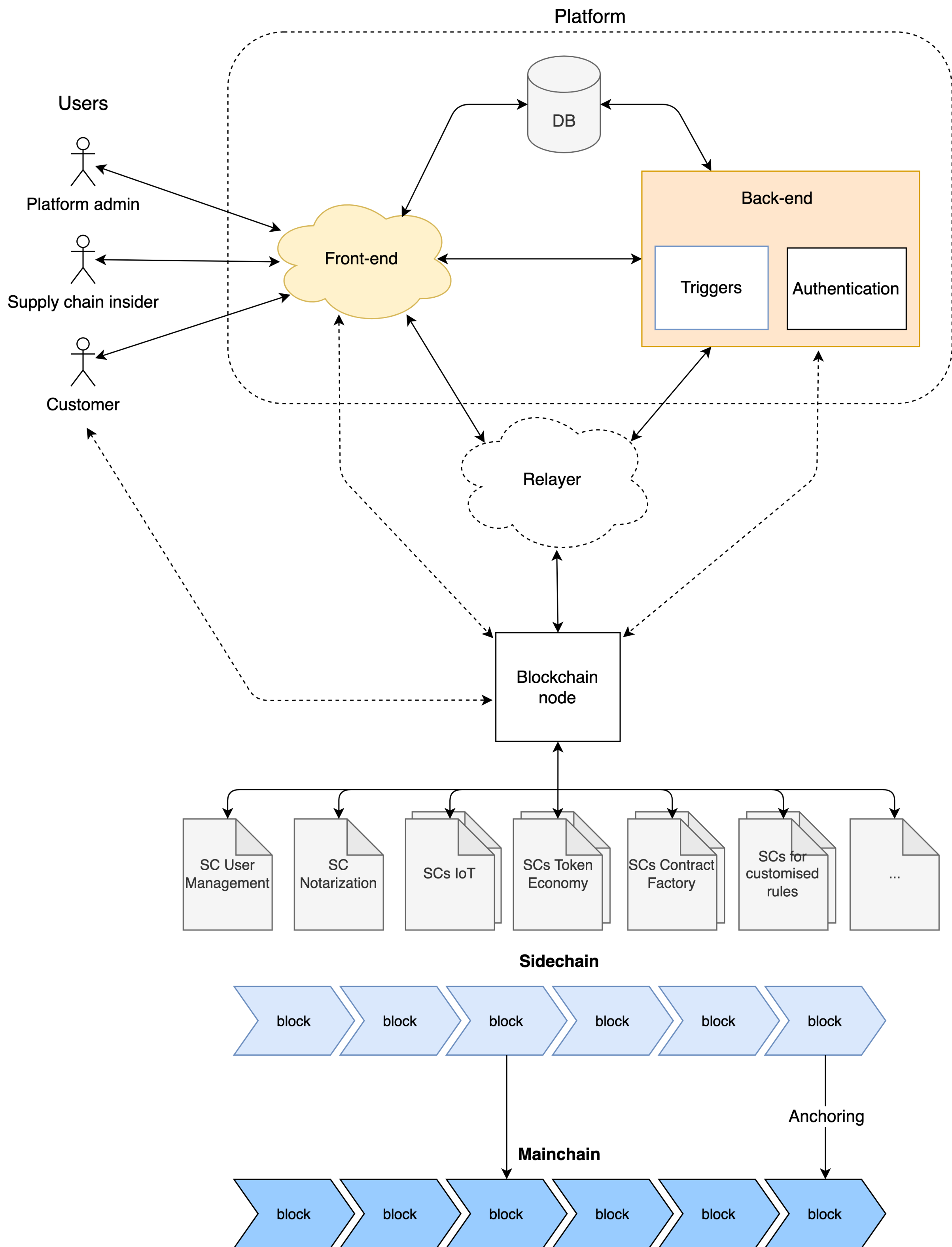

Figure 3.5: Complete system architecture of the model.

Such a representation includes: the back-end features presented before - triggers and authentication - as back-end sub-components; the three distinct user types; the complete smart contract infrastructure; and the relayer. Notice that the direct links between the platform and the generic node of the blockchain (though dashed) still stand. Furthermore, a dashed link between the users and the blockchain has been added. These choices have a precise meaning, explained in the next section.

3.2.3. Centralisation

In Section 3.1, we introduced an issue about the platform being centralised, which is discussed hereafter. The problem is that the platform just presented constitutes a single point of failure for the system. Consider, for example, what would happen if some components went offline.

If the platform as a web application goes offline, users cannot pass through it to read and write data on the blockchain. What they can do is to directly communicate with the blockchain to execute the required operations. Looking at the figure, the path is the dashed double-arrow between the users and the blockchain node. The reason for it to be dashed is that it is just a fallback link.

However, using this method has a strong impact on the usability side. As just explained, the user interface of the application is a powerful tool to enforce usability, and by skipping that step all the benefits are lost. In order to do that, the user must have access to his wallet from outside the application and know how to use it, he has to own enough funds if he wants to notarize and he actually has to execute the transactions and pay the fees. A possible mitigation for that is to design a platform as reliable as possible, which the highest availability to minimise downtime.

If, instead, it is the relayer that goes offline, the situation is similar. The application could still be used, but no notarization to the blockchain could be performed. Luckily, the problem is only related to the write operations, since the read operations do not have a cost and, therefore, they can be done easily from the platform without effects in terms of usability.

As far as write operations are concerned, it is still possible to do them in two ways: the first is the one suggested for the event of the platform being offline; the second is to exploit the fallback paths connecting the platform to the blockchain. Most of the disadvantages seen before show up again: the user must own some cryptocurrency tokens and he is in charge of paying the transaction fees.

The two examples discussed above show clearly the possible consequences of the initial problem, which is an actual limitation. The reasons for this choice are that, in a real-world case, the developer company may want to maintain a certain degree of control over its platform, limiting decentralization. For instance, given the features of the user management contract, the mechanism for users' registration requires the intervention of the contract owner.

However, it is also true that, once registered, users can notarize autonomously, potentially even bypassing the application. On the one hand, higher decentralisation can be enforced and the blockchain benefits are kept; but on the other hand, it causes a loss in terms of usability and the system does not fully work. This is a crucial concern, with no easy answer. A possible way to solve it could take the form of a DAO, a Decentralised Autonomous Organisation (see Section 1.5 for more details), whose members manage users accesses according to fixed rules, coded in some smart contracts.

3.3. Features and innovative aspects

This section is devoted to summarising the features of the system and underlining the benefits that each of them brings. Particular attention is put on the innovative aspects of the solution with respect to the existing ones.

The purpose of the system is to provide innovative tools to improve the efficiency of the entire supply chain and to trace the data contained in a certified manner, increasing the degree of protection against counterfeiting and at the same time introducing the paradigms of transparency and immutability.

Before presenting the mentioned features, a remark is worth making about usability. Usability is concerned in every IT solution and it is certainly not an innovative feature of this one. However, in the context of blockchain, to provide a usable model is challenging and non-obvious. Usability aspects have already been addressed at Section 3.2: some innovative solutions have been provided, while others require further investigation.

From a technical point of view, the innovative aspects brought with respect to a non-blockchain solution (but also with respect to the state of the art) are the following.

- Versatility - enabled by modularity,
- Traceability - enabled by transparency and immutability,
- Scalability - enabled blockchain-side by the use of a sidechain,
- Efficiency - enabled by the previous factors.

Each of these aspects is discussed below, in a specific section.

3.3.1. Versatility

As mentioned, the main idea is to employ the blockchain in the supply chain domain to implement a modular traceability service, that should also be user-friendly. The first feature that stands out from this idea is the modularity: the system is made up of multiple modules, i.e., separate functional units that can be modified, tested and removed from the system, independently from each others, thus guaranteeing large flexibility and maintainability.

Among the requirements, the platform should be versatile to different use cases, by supporting supply chains from different contexts. The modularity property provides this function: because of its flexibility, a modular solution can easily integrate with and improve the already existing supply chains. It is sufficient to substitute the sector-specific modules to obtain the wanted behaviour, without changing the underlying logic.

Furthermore, adding new features is straightforward: it is enough to add a module, regardless of the current ones. For example, each company could easily define customised rules for its specific needs, which would be incorporated into the blockchain thanks to the usage of smart contract factories. In this way, different supply chains can be managed on the same infrastructure with customised smart contracts.

Such a high versatility and flexibility to multiple sectors constitutes a **main innovative aspect** of this work, with respect to the state of the art presented in Chapter 2.

3.3.2. Traceability

Goods and processes tracking is obviously already present in supply chain management; the innovation brought here is traceability improvement. In order to enforce it, the notarization of the entire production process of the chains employing the platform needs to be regulated. The origin of a product, all the steps and transformations it went through must be recorded on the blockchain.

To do this, by maintaining a decentralised management of the supply chains, the individual steps of the supply chain processing are delegated. This means that it is each company's responsibility to notarize its operations. In this way, all information related to production, transport and commercialisation of the goods, will be accessible and freely available to everyone, with the guarantees of transparency and legal validity offered by blockchain technology.

The choice of tracking the whole supply process in a decentralised way is quite innovative also with respect to those state-of-the-art solutions - currently adopted for the supply chain - which are based on the blockchain.

3.3.3. Scalability

Scalability is the property of a system to handle a growing amount of work by adding resources to the system. From the blockchain point of view, obtaining high scalability is challenging. Considering the mainchain of Ethereum, the low scalability, due to a slow transaction speed and high fees, is one of the main drawbacks. As explained in Section 3.1.2, the use of a cheap and reactive sidechain solves this problem and satisfies the production needs. Anchoring to a permissionless mainchain, then, allows to keep the other required features, transparency and legal value.

Considering the state of the art, the use of a sidechain to solve scalability problems has not already been applied in this domain: therefore it definitely represents an innovative aspect.

Looking at scalability from the platform side, it can be enforced by means of more conventional methods, like the employment of cloud solutions, that are quite widespread in the market. Therefore, it is not remarkable to discuss them for the purposes of the current section.

3.3.4. Efficiency

Finally, a note about efficiency. The innovation in this case refers to the improvement in supply chain efficiency. The latter is, namely, the ultimate objective of the proposed model. All of the above-mentioned aspects - to varying degrees - contribute to improve it.

In particular, the automated tracking of the products reduces the downtime between a step and the following one. A usable platform, well-integrated into the legacy systems, prevents from getting stuck during the recording of data. The versatility enables the capability of realising punctual updates to the system, to satisfy the ever-changing demands of specific sectors. Scalability avoids that the system slows down or gets congested, impacting supply chain processing.

4 | Use case: Deply project

If Chapter 3 was the description of the model *in theory*, Chapter 4 is the model *in practice*. Indeed, the author had the opportunity to see something really similar to the proposed model, applied to a concrete use case. In particular, it is a real-world project in the supply chain context, named Deply [26].

This author collaborated on the project by working together with the engineers of the developer company, specifically, but not only, on the blockchain design and smart contract coding.

The business case is the following. Deply objective is to bring innovative tools to the market, bringing several advantages: improving the efficiency of the supply chains, tracing in a certified way the data contained, increasing the protection of the authenticity of the products, introducing the paradigms of transparency and immutability of data notarized through blockchain, also for the end customer.

To achieve this goal, a cloud platform has been developed, based on the blockchain technology, able to integrate within six different production chains: aerospace, agrifood, automotive, cosmetics, pharmaceutical, textile. The platform comes in the form of an application, accessible from both the web and mobile devices.

In order to make the objective clearer, we give an example of how the system could be in its final state, with a mock-up about the pharmaceutical chain. This sector includes the vaccines and the drug supply chain. Two figures are provided to give an insight into the final result. First, Figure 4.1 shows a possible supply chain flow and all the steps it passes through. The interaction with the blockchain happens after each step: the notarization steps are specified as well.

The second figure - Figure 4.2 - focuses on the point of view of an employee of a transport company. The latter likely uses the mobile form of the application to add and view the received batches of products. The capture on the left refers to the delivery of vaccine batches notarized on the blockchain; the one on the right is the detailed view, reachable by tapping on the *Order details* option. It contains more information about the order,

including the date, hash and block ID of the notarization on the blockchain.

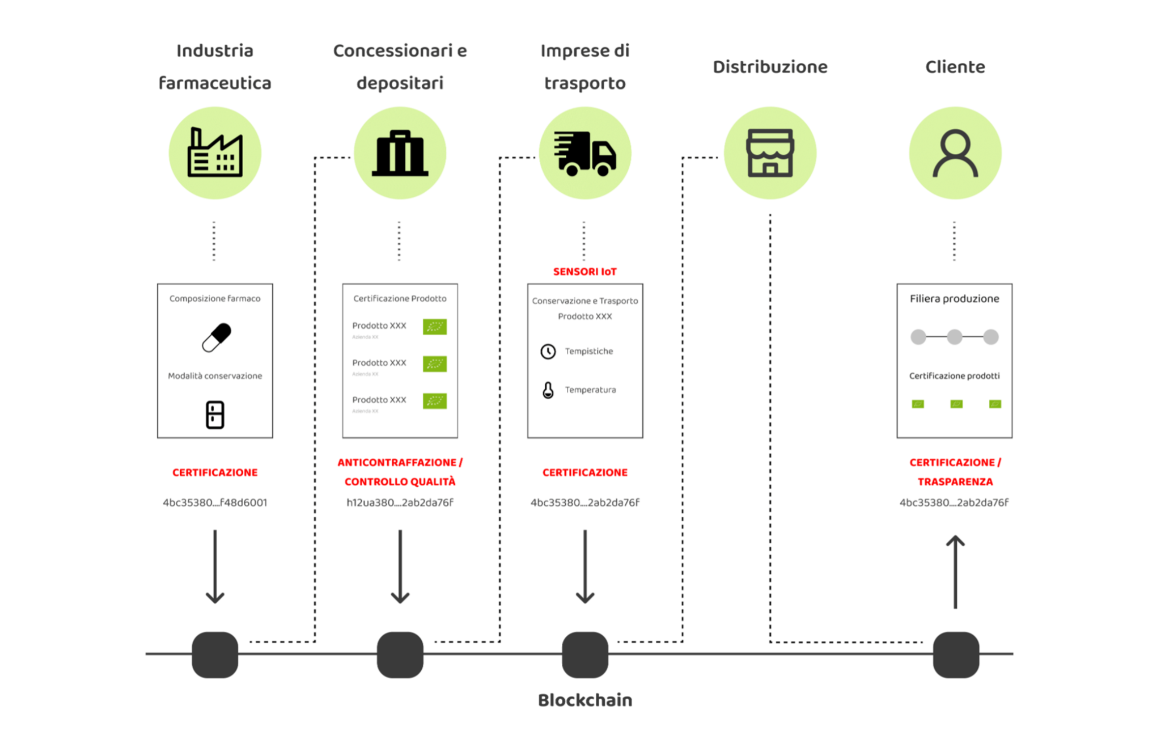

Figure 4.1: Mock-up scheme of the pharmaceutical chain process and interaction with the blockchain.

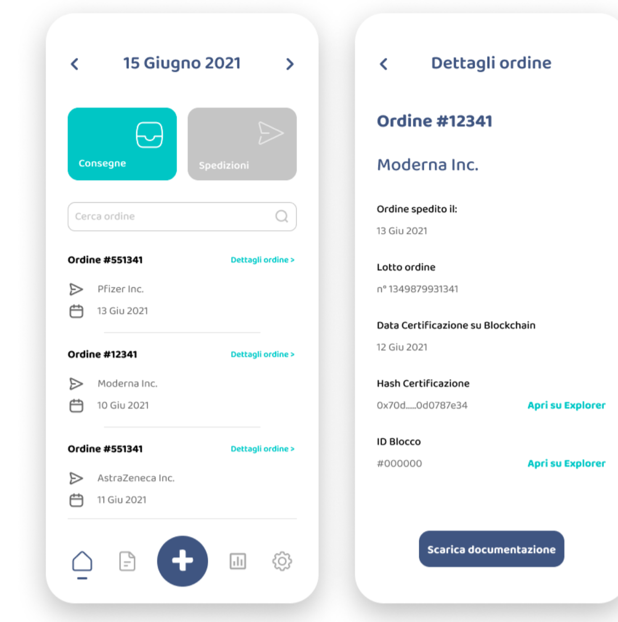

Figure 4.2: Mobile view of a specific step of the process: a transportation company views the deliveries (on the left) and the details of one product (on the right).

The outline of the chapter is the following. First, Section 4.1 clarifies the author's contribution about the different phases. Second, Section 4.2 gives a first general description of Deply project, as an occasion to see a concrete realisation of the model. Objectives and phases are briefly outlined, as well as the system components. For each of them, a punctual reference is made to the corresponding component of Chapter 3. Third, Section 4.3 is devoted to the use case implementation. It provides a more precise description of the implementation choices and the technologies actually adopted. Particular attention is put on the blockchain side and on the smart contracts.

4.1. Contribution

The following paragraph is devoted to presenting the author's contribution. It consists of an overview of the described model and project phases, with specifications about the author's work. Among the others, it will be mentioned: the design of the system and blockchain architecture (in particular smart contracts); the theoretical definition of the model; the implementation and testing of the smart contracts; the contribution about the user interface of the application.

The author joined Deply during its first phase, the project design phase (described in Section 4.2.1). At that point, a general idea about the system had already been defined, with some pending choices.

The author's started from this draft to write the theoretical model definition, subject of Chapter 3. Hence, the theoretical model is due to the author, who took inspiration from the on-going development of Deply.

Returning to the implementation, we said that there were some pending decisions. In particular, regarding the blockchain platform, two alternatives were considered: first, a public permissionless sidechain, anchored to a mainchain; and second, a public permissioned platform, developed on-premise. The author contributed to the architectural design decisions under the supervision of a blockchain engineer. The starting point was the choice of which solution to adopt for the system, and it fell on the first solution.

A second architectural choice was related to the notarization process. The problem of transaction fees, discussed in Section 3.2.2, had to be solved through meta-transaction mechanisms. The decision was between the employment of an existing relayer, to be integrated within the new platform, and the full development of a relayer service. Given the chance of using the already available Pablock service, the choice was in that direction, allowing us to dedicate resources to the other parts.

The architectural design also included the definition and refinement of the high-level and low-level architectural schemes presented following in Section 4.2.2. Concurrently with it, the author proposed some flowchart diagrams, in order to explore the interaction among the actors involved. This helped to clarify the task of each component, supporting the refinement process of the architecture.

After establishing the architectural scheme, it was possible to focus on the blockchain part, and, in particular, on the smart contracts. Again, the design of the smart contracts took place under the supervision of the same blockchain engineer as before. During this phase, all the decisions about the smart contracts emerging in Section 4.3.2 were taken. We mention, first of all, the separation of the logic into two contracts; then the required attributes and methods, and the choice about which data to keep on-chain (i.e., as little as possible).

The kind of problems that arose at this step are, for example, about the worker addition from an admin. We decided to prevent a non-subscribed admin from adding workers, and only keeping on the storage the workers associated to a subscribed admin. The flow expected by design was too detailed for the flowcharts, therefore the decision was taken here.

Then, the actual implementation of the smart contracts was undertaken. The author took care of this task entirely, including their testing. The parts involved in this step, beyond the coding of attributes and methods defined in the design phase, are the choice of the modifiers, the definition of the events, and the check for state consistency in general.

After the completion of the tests, performed inside a blockchain running on a local environment, the deployment of the smart contracts on a testnet was carried out, in order to test them in a context closer to the target one. A testnet is a copy of a real-world blockchain, devoted to developers' experiments and tests only. It consists of a really distributed environment, while, during the local tests, the decentralisation of the network was only simulated.

As for what concerns the front-end side, the author joined that after it was fully designed, during its implementation. He collaborated on the development of the user interface. In particular, the realised part is the backoffice, i.e., the web pages devoted to Deply platform admins. As said, it was accomplished using the React framework. First, the static views of the interface were implemented, then the dynamic parts were set up. The latter include the queries to the Firestore database and the interaction with the smart contracts, carried out by calling back-end cloud functions.

At the end, the author contributed to the integration between the different components and to the system testing phase, including the resolution of the problems. The kind of issues that arose at this point was about inconsistencies between some parts, as well as some initial settings causing errors.

4.2. Project description

The current section is a general overview of the project. After a brief summary of Deply objective and phases, the system description is provided. This section can be considered as the analogue of Section 3.1 in the previous chapter: each component presented above is mapped to the corresponding component of the concrete system.

4.2.1. Task and phases

As already mentioned, Deply **task** is to bring innovative tools to the market, to improve the supply chain efficiency and increase the protection of the authenticity of the products, by tracing in a certified way the data contained and introducing the paradigms of data transparency and immutability. The means to accomplish such a task is a cloud platform, based on the blockchain technology, able to integrate within different production chains.

As for the Deply **phases**, the main ones are three:

1. The *project design* of the functional and infrastructural solution - includes the analysis of the use-cases and design of the platform usage flows for each sector, design of the user interface.
2. *Platform realisation and testing* - biggest phase: includes the design of the architecture, of the blockchain components and smart contracts, of the web and mobile platform, their development, and the testing phase.
3. Complete platform implementation - product engineering and writing of the technical documentation.

Notice that, in this section, the term *platform* is used to reference also the whole system. At this point of the discussion, the information given about each component is enough to avoid confusion, therefore the distinction made in Chapter 3 is not needed anymore.

4.2.2. Architecture

After having introduced the project, let's now switch to the actual description of the platform. As before, we refer to an architectural scheme, shown in Figure 4.3.

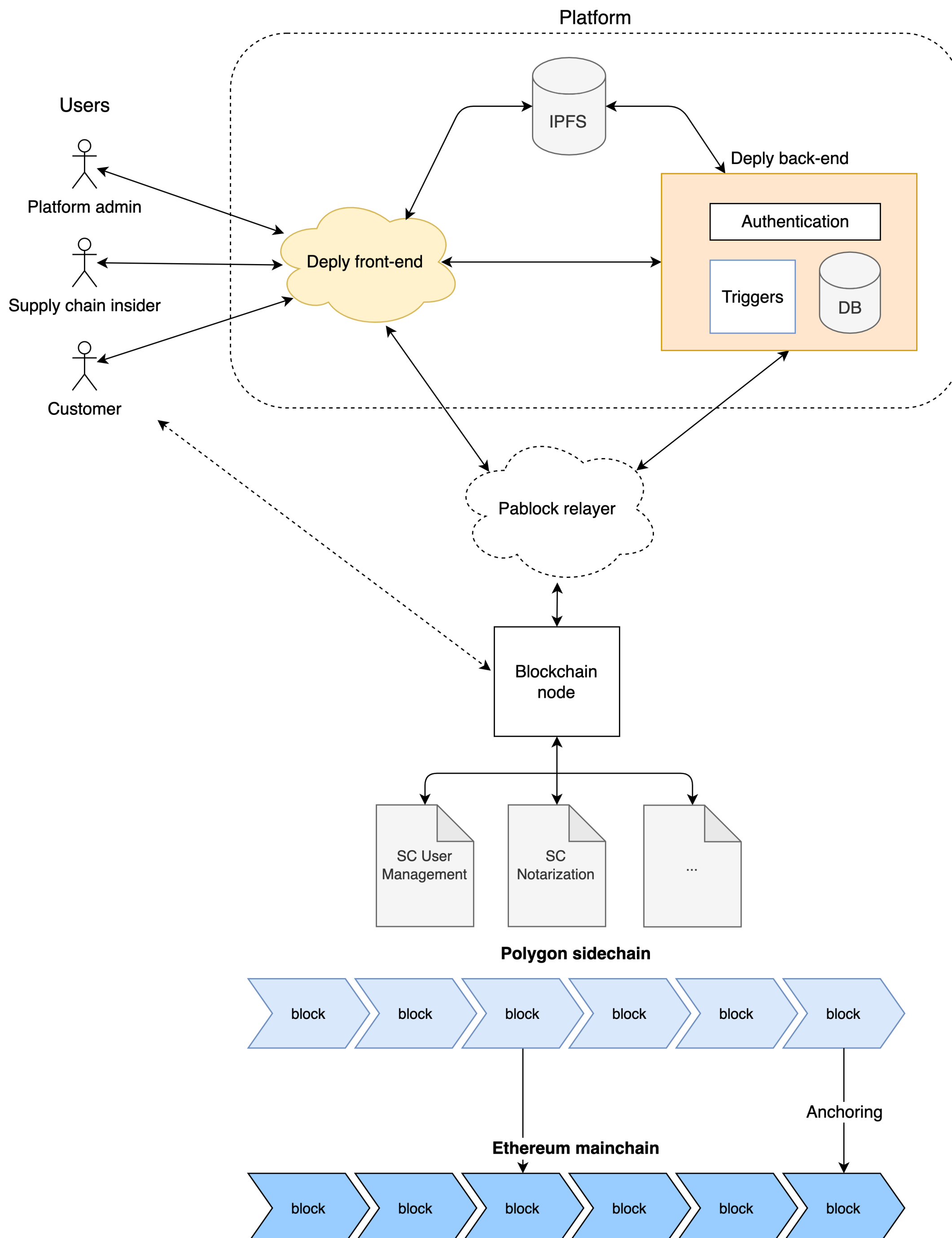

Figure 4.3: System architecture of Deply.

The figure above provides a detail level comparable to the most refined one of Chapter 3 (see Figure 3.5). This is because a more specific scheme is needed in order to be able to concretely map the theoretical functionalities to the actual ones.

Comparing the two schemes, the mapping between the theoretical model and the concrete solution is immediate and provided in Table 4.1. The correspondences identified are discussed below.

Theoretical model	Concrete solution
Front-end	Deply front-end
Back-end	Deply pseudo back-end with subcomponents
Database	IPFS + Back-end DB
Relayer	Pablock relayer
Sidechain	Polygon sidechain
Mainchain	Ethereum mainchain
Smart contract user management	Smart contract user management
Smart contract notarization	Smart contract notarization
Users (platform admin, supply chain insiders, customers)	Users (platform admin, supply chain admins and workers, customers)

Table 4.1: Component mapping from the proposed model and Deply solution.

The first component is the **front-end**, mapped to Deply front-end. It consists of a user interface, accessible both from the web and from mobile, allowing the users to easily interact with the rest of the system. The technological choices for the component and its internal settlement are exposed in the next section.

Regarding the **back-end**, there is something more to be discussed. Deply back-end is not a real back-end, as motivated by the technological choice presented in Section 4.3, but it is provided by a cloud service. In particular, it consists of a serverless architecture. As pointed out by its sub-components in Figure 4.3, it performs the role of user authentication, includes an internal database, and contains the functions to be triggered by the front-end, in order to modify the state of the various storage components.

The **database** is strongly related to the back-end, since they both play the role of data storage. The *DB* component of the theoretical model is mapped to the InterPlanetary File System (IPFS), which is not only a database but a distributed system in general [27]: its advantages are better explained in the next section. The choice of specifying the

technology employed for the database is due to the need to distinguish between the two mentioned components.

Nevertheless, a question arises: why is there another database inside the back-end? The short answer is that both IPFS and the back-end DB are needed, since they store different kinds of data. Also this point is investigated in the implementation section (4.3).

As previously explained, the **relayer** allows to execute transactions on the blockchain without paying for them, relying on its service. The relayer is the way through which the platform (specifically the back-end) accesses the smart contracts on the blockchain. The specific one exploited by Deply is called Pablock [28] and comes with all the described features.

The platforms chosen for the **sidechain** and for the mainchain of this project are the exact same proposed by theoretical model, i.e., the Polygon blockchain as the sidechain and Ethereum as the mainchain. We recall that the use of a sidechain improves network scalability, by increasing the transaction rate and lowering their cost. Since most of the explanation needed was already given, these technological choices are reported in Figure 4.3, even if this was not done for the front-end and the back-end.

As far as the **users** are concerned, Deply introduces a more detailed distinction with respect to the generic one of the model. Three plus one different roles are planned for the users:

- *Platform administrator* (platform admin) - belongs to the client company, manages the system, creates the user profiles from the different supply chains, and enable or disable the subscriptions.
- *Supply chain administrator* (supply admin) - belongs to a supply company adopting Deply system. Created by the platform admin, this actor has the privileges to notarize and add or remove workers under his supervision.
- *Supply chain worker* (supply worker) - belongs to a supply company adopting Deply system. Created by the admin of the same supply chain, a supply worker may have or not have the privileges to notarize on the blockchain. Anyway, he can record the complete transactions as a draft, requiring approval.
- *Customer* - is the generic person who bought a product certified using Deply and is interested into viewing its provenance and processing up to him. The customer is not considered a proper role, because does not need a registered profile neither on the platform nor on the smart contracts, since he performs only read operations.

The last part of the system that needs to be discussed are the **smart contracts**. Deply system, at least in its present form, relies on two smart contracts: one for user management and one for notarization. These correspond to the homonymous contracts of the initial model, and each node of the sidechain provides access to them.

The former contract is in charge of handling the users: the authorisations are distributed according to the role. The platform admin can add and enable the subscriptions of the supply admins, which in turn can add the supply workers. The notarization contract, instead, relies on the user management one to check for the authorisations. After the check is successful, the notarization is executed and an event is emitted. The actual implementation of the smart contracts is provided in Section 4.3.2, as well as the relevant parts of the code.

A remarkable question about the system emerging at this point is how to deal with availability problems: e.g., what to do if the relayer or the web application is **offline**? An interesting aspect about the employment of a blockchain such as Polygon is the fact that it is public. Therefore, everyone can read the information stored there, as also pointed out by the dashed line connecting the users and the generic blockchain node in Figure 4.3. Specifically, the hash is read on the blockchain and it allows to retrieve the related information, e.g., in a document on IPFS.

Furthermore, an authorised user - as, for example, a supply worker subscribed to Deply - can notarize the information out of band with respect to the platform. Thus, notarization could also be performed if the application is offline for some reason. This fact enforces the decentralisation property of the blockchain, guaranteeing a high degree of availability, only dependent on the blockchain availability. However, this chance has a quite strong requirement in terms of usability, as anticipated in Section 3.2.3. In order to execute a transaction, a user must own a cryptocurrency wallet with enough funds to pay for it. This is not granted, given the technical knowledge required to do it.

4.3. Implementation

The current section is about the project implementation. In the first part, the components presented before are better investigated: the design choices taken are pointed out, as well as the technologies actually employed. Particular attention is devoted to the decisions taken for the front-end and the back-end. In the second part, the smart contract implementation is addressed, by actually looking into the code.

4.3.1. Implementation of the application

The aim of the current section is to give a more detailed description of the web application, including the technological choices for the front-end and the back-end. The smart contract part, instead, is presented within Section 4.3.2. Figure 4.4 below shows the architectural details about the platform.

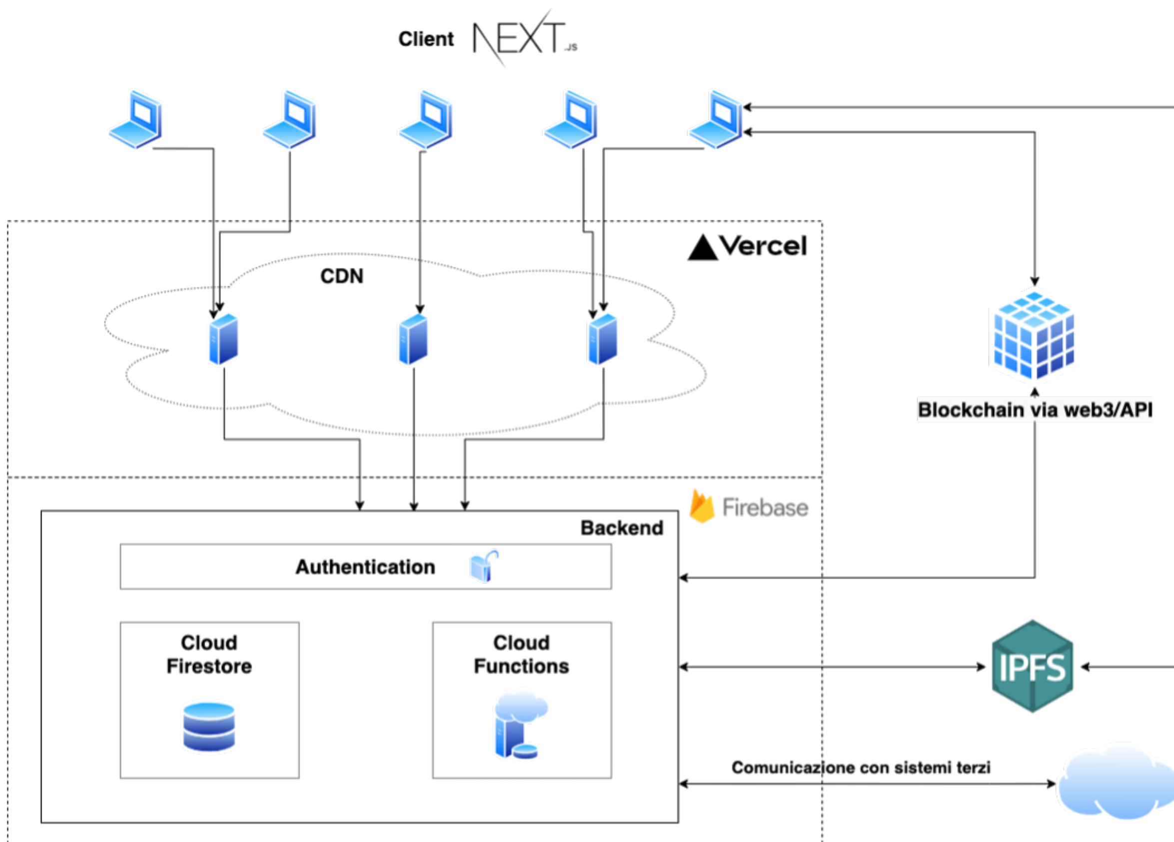

Figure 4.4: Architectural view of the application.

First, the technologies employed for the front-end and the back-end are presented are they are discussed as whole components; then each specific component is individually analysed. Eventually, some considerations are done that were not included before.

For the implementation of the **front-end**, Deply exploits the Next.js framework, provided by Vercel [29]. Next.js is a React web development framework, built on top of Node.js (a multi-platform JavaScript runtime environment), enabling functionalities such as server-side rendering, to create fast and responsive web applications. Then, it uses a hosting service (also provided by Vercel), on which the application bundle is loaded, autonomously published and distributed to the various requesting clients.

As for the **back-end** part, it must be said that there is not a real back-end, but it is a *backend-as-a-service* solution, i.e., a cloud solution. Deply mainly exploits Google's Firebase ecosystem as the foundation of its architecture [30]. The serverless infrastructure provided by Firebase offers a series of out-of-the-box solutions and tools that facilitate the development of this core part of the web application. Firebase manages all the functionalities mentioned before: authentication, the database (i.e., Cloud Firestore) and the triggers (i.e., Cloud Functions).

Let's now provide a detailed overview of Figure 4.4 components.

- **Client** - are all the end users of the service that, by connecting to Deply web page via their browser, receive the bundle of the application;
- **CDN** (Content Delivery Network) - takes care of serving the static bundle of the application to the various clients that require it. It is a geographically distributed structure of servers or nodes collaborating in order to meet and optimise the various requests for static content. This infrastructure is entirely maintained by Vercel, that deals with both the distribution of the bundle and its management (caching...). The network just described allows users to have less latency in the normal use of the application, thus improving the usability;
- **Authentication** - is the layer exposed by the Firebase SDK, that allows the authentication of the client HTTPS calls to the back-end services. In addition to the classic user authentication, we also have a set of so-called *rules*, through which read and write accesses to the various schemes of the application database are defined;
- **Cloud Firestore** - is the application serverless database. It is a non-relational NoSQL data structure, that allows: an automatic scaling of the resources, based on the number of incoming requests from the various clients; their synchronization in real-time; and an offline mode not to lose data, useful also in case of network latency or lack of connection;
- **Cloud Functions** - is a server environment, where to perform serverless functions in response to certain events (e.g., associated with the database or HTTPS requests from clients). The space is protected, not exposed to the client, automatically scales according to the incoming load, and can also be used for data integration from third-party systems;
- **Blockchain via web3/API** - the link with the blockchain part of the Deply architecture can be performed directly, through appropriate libraries (`web3.js`, `ethers.js`), or indirectly, through API calls to third-party services;

- **IPFS** (InterPlanetary File System) - is a peer-to-peer network that implements a communication and data sharing protocol in a distributed file system. In this way, documents and images of the supply chain can be stored in a completely decentralised, immutable manner, with high scalability;
- **External Services** - Deply back-end is able to leverage Cloud Functions to interact with any services offered by third-parties, such as SAP management systems or other data sources, as for example IoT devices, as long as the latter expose APIs or SDKs for the interfacing between systems.

Among the pending questions, the database matter is a remarkable one. As anticipated before, the question was about the reason behind the use of two different databases. The answer is that they are due to two distinct roles.

Firestore is the application (serverless) database: it allows for accessing the data in real-time, in a really fast way, and contains data about the users, the supply chain products and processes and the subscriptions. Only authenticated users can perform read and write operations within their competence. IPFS, instead, is a distributed file system, employed for storing larger files, related to the supply chain processing. Through the hash of a specific blockchain transaction, it is possible to retrieve the associated images and documents. The fact that it is distributed improves scalability and immutability properties, while confidentiality can be enforced by encryption.

At last, a note on the relayer. Pablock relayer is the intermediary between the application and the blockchain, allowing - by means of meta-transactions - to execute state-changing transactions without paying the fees for them. However, it is the only way to make calls to the blockchain from the application and this could be a problem. Indeed, if the service is offline, there is no fallback planned, so it is impossible to notarize information. Anyway, it still holds the chance to directly execute transactions to the smart contracts without passing through the application, although it is a more complicated path. Finally, we remark that read operations do not require relayer intervention and can be performed even if Pablock is offline.

4.3.2. Smart contracts

The current section focuses on the smart contracts. They are presented in detail, by also introducing some snippets of code, in order to better explain the concepts. The full code of the user management and notarization contract is presented in Appendix A and Appendix B, respectively. The description starts with an outline of the design, the programming language and the development environment chosen, moving then to the actual

content and functionalities provided. The testing phase, which has been accomplished as well, will be examined in the chapter about the evaluation (Chapter 5).

The smart contract design takes place after the design of the blockchain architecture and the draft of some flowcharts, representing the most common use cases for the system.

The choice of the programming language was made subsequently to that of the blockchain platform. Since the project relies on the Ethereum ecosystem, the programming language choice fell on Solidity. Solidity is an object-oriented language for developing smart contracts, which runs on the Ethereum Virtual Machine (EVM). As far as the development environment is concerned, Truffle framework has been employed. Truffle is one of the possible tools allowing for all the development, testing, and deployment processes to be performed with the same program.

The starting point for the implementation is the requirement gathering. Such requirements are transformed into a high-level representation of the data structures required and the functionalities. Concretely, they were initially written in an informal way as a list of data structures and functions, that - given Solidity's object-oriented nature - could be easily translated into a set of attributes and methods. That's exactly what happened in practice for the two contracts.

Before entering into the code, we give a note about the interaction with Pablock relayer. For the correct working of the system, the development of Deply smart contracts must be contextual to their integration to Pablock existing contracts. Therefore, they are both declared as a sub-class of the `PablockMetaTxReceiver` contract defined by Pablock and their constructor is derived from the one of the father contract. We will point it out when showing the contract constructors.

Following, we describe the contents of the user management and the notarization contracts, with the support of some pieces of code.

User management

The following section is devoted to the description of the smart contract for the user management, whose complete code is provided in Appendix A.

The `UserManagement` contract, as said above, extends `PablockMetaTxReceiver` and calls the father contract constructor inside its own, in order to support meta-transactions. As shown in the following snippet of code (Listing 1), the constructor takes as parameter the address of another smart contract from Pablock, `EIP712MetaTransaction`, and uses it to call a function from the father contract and is only needed for internal purposes. The

father constructor receives as input two strings: the name and the version of the current instance as a meta-transaction receiver.

Apart from those parameters - needed for the integration - two operations are performed by the constructor. The first is the setting of the `owner` variable to the address who deployed the contract. This allows for access control management: some operations can be restricted to be done only by the owner. Notice that `msgSender()` function is called to retrieve the sender, instead of the usual `msg.sender`. This is due to the meta-transactions mechanism and allows to obtain the actual transaction sender. The second operation is to initialise the `userId` variable to 1. Such a variable is visible in the whole contract and it is incremented every time a new user is added.

Listing 1 User management - Contract declaration and constructor.

```
1  contract UserManagement is PablockMetaTxReceiver {
2      address owner;
3      uint256 userId;
4      ...
5
6      constructor(
7          address metaContract    // address of EIP712MetaTransaction contract
8      ) PablockMetaTxReceiver("deply_user", "0.0.1") {
9          owner = msgSender();
10         userId = 1;
11         setMetaTransaction(metaContract);
12     }
13     ...
14 }
```

Regarding the data structures, Listing 2 snippet below includes their definitions.

First of all, we have the distinction between two roles, `LdF` and `AdF`, respectively, referring to the roles of supply worker (in italian *Lavoratore di Filiera*) and of supply admin (*Amministratore di Filiera*). This distinction is realised as an `enum` and it is one of the attributes of the main data structure of the contract, the `User` object.

The other attributes are the already-mentioned `id`, the address of the current user's `admin` (if not an admin himself), an array of the handled `workers`, and two counters: `PTKs` and `workersCounter`. The latter is set by the platform admin for supply admins only; the former requires a further comment. It has been implemented from a utility provided by the well-known OpenZeppelin library [31], by slightly modifying `Counters`. Methods

for setting a specific value and resetting the counter have been added. The name PTKs means *Pablock Tokens*.

Finally, we have two mappings. The mapping `users` maps each registered user with the corresponding address. Notice that the role of platform admin is covered by the contract owner, not registered to the user list.

About that, it must be noted that not all the user types are supported by the smart contract. The reason is simply that some of them are not worth recording, and they would also be overhead. In particular, we refer to two types of users, the customers, and the supply workers without notarization capabilities. The former only need to perform read operations on publicly available information. The latter, although they can record supply chain data on the platform, are not allowed to notarize them, since confirmation is required. Thus, they never interact with the smart contracts, making it unnecessary to register them.

The second mapping, `contracts`, is required for the reception of the meta-transactions: if they come from a white-listed contract, then it can be executed.

Listing 2 User management - Variables and data structures.

```
1    enum UserType{ LdF, AdF }
2
3    struct User {
4        uint256 id;
5        UserType userType;
6        address admin;
7        address[] workers;
8        bool isEnrolled;
9        Counters.Counter PTKs;
10       uint256 workersCounter;
11   }
12
13   mapping(address => User) users;
14   mapping(address => bool) contracts; // whitelisted contracts for notarization
```

Continuing with the overview, we have the modifiers. Below, we provide some of them as an example. Their name are self-explanatory. A remarkable aspect is the use of assembly code in order to check if a blockchain address belongs to a contract. Other modifiers are omitted for the sake of brevity and can be viewed in Appendix A.

Listing 3 User management - Modifiers.

```
1    modifier onlyOwner {
2        require(msgSender() == owner, "Only the contract owner can call this.");
3        _;
4    }
5
6    modifier isContract(address _address) {
7        uint32 size;
8        assembly {
9            size := extcodesize(_address)
10       }
11       require(size > 0, "Not a contract address.");
12       _;
13   }
```

A simple but core function is the `signUp` function, whose mechanism about the `id` has already been explained before. We recall that such a function is used to register only supply admins.

Listing 4 User management - Sign-up function.

```
1    function signUp() public notRegistered(msgSender()) {
2        users[msgSender()].id = userId;
3        userId++;
4        users[msgSender()].userType = UserType.AdF;
5    }
```

A fundamental part in object-oriented programming are the *getters*, used to retrieve the attributes of a class. Some of them are provided in Listing 5 below.

`UserManagement` has some standard getters, as `getId`, and some others that serve as auxiliary functions for the corresponding modifiers. For instance, the function `isRegistered` is only called by the `registered` and `isAdmin` modifiers. This is useful because the wrapping can be done also from other contracts linked to the current one. As an example, the `Notarization` contract exploits the two functions `isRegistered` and `isActionable` to define their own modifiers.

Listing 5 User management - Getters.

```
1        function isActionable(address _address) public view returns (bool) {
2            if(users[_address].userType == UserType.LdF){ // if worker
3                return (users[users[_address].admin].PTKs.current() > 0);
4            } else {
5                return (users[_address].PTKs.current() > 0);
6            }
7        }
8
9        function isRegistered(address _address) public view returns (bool) {
10           return (users[_address].id > 0);
11       }
12
13       function getId(address _address) public view registered(_address)
14       returns (uint256) {
15           return users[_address].id;
16       }
17
18       function userType(address _address) public view registered(_address)
19       returns (UserType) {
20           return users[_address].userType;
21       }
22       ...
```

Moving to another feature, the contract owner has the right to add or remove user subscriptions. By doing it, it specifies as `PTKs` parameter the amount of operations that the given admin can perform. A remarkable fact about the `setSubscription` method is that it can be called by providing the address of a user non registered to the smart contract. If that's the case, then the user is first registered and then the subscription is added.

The snippet below, Listing 6, shows the function related to what just explained. Beside that, also another function is included in such snippet, performing the opposite operation. The function, called `removeSubscription` sets to 0 the amount of the admin's available `PTKs`.

Listing 6 User management - Subscriptions.

```
function setSubscription(address _address, uint256 _PTKs) public onlyOwner {
    if (users[_address].id == 0) { // if not registered: sign-up as a new user
        users[_address].id = userId;
        userId++;
        users[_address].userType = UserType.AdF;
    } else {
        require(users[_address].userType == UserType.AdF, "Not an admin.");
    }
    if (!users[_address].isEnrolled) { // if counter is null
        users[_address].isEnrolled = true;
        users[_address].PTKs = Counters.Counter(0);
    }
    users[_address].PTKs.set(_PTKs);
}

function removeSubscription(address _address) public onlyOwner isAdmin(_address) {
    users[_address].isEnrolled = false;
    users[_address].PTKs = Counters.Counter(0);
}
```

We mentioned above that only permitted contracts can perform meta-transactions. In order to better understand the statement, we provide some further information about the *meta-transactions* mechanism.

The latter works so that the meta-transaction sender is the contract who executes it and is in charge of the fees. In principle, any contract could execute a meta-transaction that encapsulates another transaction. Though, in order to prevent it from happening to Deply smart contracts, a security check is provided, following explained. A smart contract, before executing meta-transactions in place of Deply contracts, must be white-listed with a specific method, i.e., `whitelistContract` presented below.

A method to make the reserve operation, called `blacklistContract`, is provided as well. After calling `blacklistContract` on a address, the contract corresponding to such address is not allowed anymore to ask Pablock for the execution of meta-transactions.

Listing 7 User management - White-listing contracts.

```
1 	function whitelistContract(address _address) public onlyOwner isContract(_address)
2 	{
3 		contracts[_address] = true;
4 	}
5 
6 	function blacklistContract(address _address) public onlyOwner isContract(_address)
7 	{
8 		contracts[_address] = false;
9 	}
```

If a subscription provides `PTKs` that can be spent to execute operations, the following methods manage the reduction of `PTKs` upon the execution. `_pay` is an internal function, called also during the addition and removal of workers from a supply admin. If it is called by a supply worker, the tokens are not taken from him, but from the related supply admin.

The function `pay`, instead, is called also from the outside of the contract in order to pay for notarizing. It requires that the sender of the meta-transaction, `msg.sender`, is a contract which has been white-listed. If all the requirements are successful, it calls the internal `_pay` function to complete the payment.

Listing 8 User management - Pay function.

```
1 	function _pay(address _address) internal {
2 		if(_address == owner) {
3 			return;
4 		}
5 		if (users[_address].userType == UserType.LdF) { // if worker
6 			users[users[_address].admin].PTKs.decrement();
7 		} else {
8 			users[_address].PTKs.decrement();
9 		}
10 	}
11 
12 	function pay(address _address) public isContract(msg.sender) actionable(_address)
13 	{
14 		require(isWhitelisted(msg.sender), "Contract is not whitelisted.");
15 		_pay(_address);
16 	}
```

As mentioned, a supply admin has the chance to add a worker under its supervision. He can do it with the `addWorker` method in the following snippet of code.

Listing 9 User management - Add worker.

```
function addWorker(address _address) public notRegistered(_address)
actionable(msgSender()) {
    // check that msgSender() is an AdF
    (, uint256 slots) = getWorkers(msgSender());
    require(slots > 0, "There are no free slots for the worker.");
    users[_address].id = userId;
    userId++;
    users[_address].userType = UserType.LdF;
    users[_address].admin = msgSender();
    users[msgSender()].workers.push(_address);
    _pay(msgSender());
}
```

The use of the blockchain guarantees complete traceability, but what happens if a user is deleted? This is a planned feature for the system, so it must be possible to perform such operation. In order to trace this in a convenient way, the event mechanism is used.

Every time a user is removed, a `Deletion` event is emitted. In this way, it is easy to keep track also of the removed users. The emission of such an event is exploited in the methods presented next.

```
event Deletion(uint256 userId, address user, uint256 userType, address _from);
```

The Listing 10 snippet shows what just explained for the deletion of a worker by his admin. The procedure (reported in Appendix A) is similar also for the removal of an admin by the owner, with the difference that, if such an admin has some workers, also the latter are deleted, causing the emission of more `Deletion` events.

Listing 10 User management - Remove worker.

```
function removeWorker(address _address) public registered(_address)
actionable(msgSender()) {
    require(users[_address].admin == msgSender(),
        "You are not the admin for the worker."
    );
    uint256 len = users[msgSender()].workers.length;

    for (uint256 i = 0; i < len; i++) {
        // swap with the last worker of the array
        if (users[msgSender()].workers[i] == _address) {
            if (i < len - 1)
                users[msgSender()].workers[i] = users[msgSender()].workers[len-1];
            users[msgSender()].workers.pop(); // deletion through pop function
        }
    }
    emit Deletion(
        users[_address].id, _address, uint256(UserType.LdF), msgSender()
    );
    delete users[_address];
    _pay(msgSender());
}
```

At last, a method is available to perform the recovery of the address of a supply admin. Such a method moves all the workers from the old to the new admin, by also setting the other attributes to the same values.

Listing 11 User management - Recovery of an admin.

```
function recoverAdF(address _old, address _new) public onlyOwner isAdmin(_old)
isAdmin(_new) {
    users[_new].PTKs.set(users[_old].PTKs.current());
    users[_new].workersCounter = users[_old].workersCounter;
    users[_new].workers = users[_old].workers;
    delete users[_old].workers;
    for (uint256 i = 0; i < users[_new].workers.length; i++) {
        users[users[_new].workers[i]].admin = _new;
    }
    emit Deletion(users[_old].id, _old, uint256(UserType.AdF), owner);
    delete users[_old];
}
```

This method is implemented on the smart contract, but at the current state of Deply is not directly supported by the web application (it could cause mismatching with the data on the database). Anyway, the recovery can be done with some precautions, like registering the supply admin in advance to the database and then perform the method call manually.

Notarization

The following section is devoted to the description of the smart contract for the notarization, whose complete code is provided in Appendix B.

The `Notarization` contract, with respect to the `UserManagement` one, is way smaller, since it contains a smaller amount of application logic. As the previous, it extends `PablockMetaTxReceiver` and the call to the constructor is done in the same way. The contract declaration and constructor is shown in Listing 12.

Listing 12 Notarization - Contract declaration and constructor.

```
import "./UserManagement.sol";

contract Notarization is PablockMetaTxReceiver {
    address owner;
    UserManagement public userContract;
    ...

    constructor(
        address _userAddress,
        address metaContract    // address of EIP712MetaTransaction contract
    ) PablockMetaTxReceiver("deply_notar", "0.0.1") {
        owner = msgSender();
        userContract = UserManagement(_userAddress);
        setMetaTransaction(metaContract);
    }
    ...
}
```

As this regards, it is important that the name passed in the constructor (i.e., `deply_notar`) is different from the one passed in the user management contract (i.e., `deply_user`), allowing to distinguish between the two when performing the meta-transactions.

Another remarkable point is the link to the `UserManagement` contract. The latter is imported at the beginning, and it is declared as a public variable inside the contract. In

order to actually connect to it, `UserManagement` contract address is passed as a constructor parameter.

The whole objective of all the contract infrastructure is notarization. The cornerstone that enables the notarization resides in a single instruction, specified below. It consists of an event, named `Record`, which contains the hash of the data to be notarized, as well as the transaction sender and a URI, that can be a link, to optionally provide additional information.

```
event Record(bytes32 hash, string uri, address applicant);
```

As mentioned for the previous contract, `Notarization` regulates the transactions by means of two modifiers, presented in Listing 13: `registered` and `actionable`. Such modifiers, exactly like before, make use of the corresponding methods defined in the contract `UserManagement`, i.e., they are *wrappers*.

Listing 13 Notarization - Modifiers.

```
1 modifier registered(address _address) {
2     require(userContract.isRegistered(_address), "User not found.");
3     _;
4 }
5
6 modifier actionable(address _address) {
7     require(userContract.isActionable(_address), "no more PTKs");
8     _;
9 }
```

The `Record` event and the modifiers just introduced are used into two notarization functions, called `notarizeDirect` and `notarize`, both requiring to be called by a registered user. Their implementation is shown in Listing 14 below.

Both functions emit a `Record` event, under different conditions. `notarizeDirect`, the first one, is the standard one: it relies on Pablock service and thus, in order to be executed, it must be actionable, i.e., the caller user must own at least one token to pay for it.

The second one, simply named `notarize`, is a fallback function. It can be called without having PTKs and does not pass through the relayer. Consequently, it also works if the web application and/or the relayer are offline, but, on the other hand, the caller is in charge of paying the transaction fees.

Listing 14 Notarization - Notarize functions.

```
function notarizeDirect(bytes32 _hash, string memory uri) public
registered(msgSender()) actionable(msgSender()) {
    userContract.pay(msgSender());
    emit Record(_hash, uri, msgSender());
}

function notarize(bytes32 _hash, string memory uri) public registered(msg.sender)
{
    emit Record(_hash, uri, msg.sender); // without using PTKs
}
```

As mentioned above, the implementation of both contracts came with the relative testing. However, they are out of the scope of this chapter, therefore they are not explained here. They are addressed in a specific section of Chapter 5.

5 | Evaluation

The current chapter is the last chapter adding new content to the thesis, only followed by the conclusions. This work so far described a platform, based on the blockchain, to improve supply chain efficiency with a special focus on the traceability aspect. The model was first presented from a theoretical perspective, then from a practical one. The aim of this chapter is to present some considerations about the proposed system, with reference to the concrete development of Deply project, described in Chapter 4.

An evaluation of the system is made, from two perspectives. A quantitative evaluation is conducted on some aspects, a qualitative one on others. The quantitative evaluation regards the blockchain-side, where the implemented smart contracts are the main point. They are assessed through a completeness analysis, involving their testing.

From a qualitative point of view, the implementation choices taken are analysed and the differences between Deply and the model proposed in Chapter 3 pointed out. Furthermore, usability issues introduced in the previous chapters are addressed.

About the latter, the decisions taken about usability need to be specified, and the weaknesses discussed. Alongside usability, other aspects stand out. Specifically, functional aspects related to the system behaviour in unexpected situations are a main concern.

The outline of the chapter is now presented. First, Section 5.1 contains a comparison between the theoretical model and concrete model described in this work. The components and features of the two are discussed, highlighting similarities and differences between them. Then, Section 5.2 is about the two implemented smart contracts: in particular, a completeness analysis is made, by means of testing. After that, the chapter is concluded with Section 5.3, devoted to the review of the platform usability aspects and to some functional aspects. The unresolved questions from the previous chapter are addressed, as well as some notable issues.

5.1. Features of the model

This section provides a comparison between the theoretical model from Chapter 3 and its actuation described in Chapter 4. This is accomplished by analysing the components one by one, pointing out the functionalities not implemented, but also the ones that have been expanded or modified.

In order to meet the obligations about the content of this section, we need a structured and clear starting point. The one considered the most suitable is a table, containing all the features of both the theoretical and the concrete solution. The aim of Table 5.1 is to summarise the functionalities available in each model, all in the same place. In particular, the reference schemes are the ones provided in Figure 3.5 and Figure 4.3.

(1) Theoretical solution		**(2) Concrete solution**		**Is in (1)**	**Is in (2)**
Component	*Subcomponents*	*Component*	*Subcomponents*		
Front-end	(not specified)	Vercel application	CDN	✓	✓
Back-end	Authentication	Firebase infrastructure	Authentication	✓	✓
	Triggers		Cloud Functions	✓	✓
	Inner DB		Cloud Firestore	✗	✓
Database	-	IPFS	-	✓	✓
Relayer	(not specified)	Pablock relayer	A set of interoperating smart contracts	✓	✓
Sidechain	-	Polygon sidechain	-	✓	✓
Mainchain	-	Ethereum mainchain	-	✓	✓
Smart contract	User management	Smart contract	User management	✓	✓
	Notarization		Notarization	✓	✓
	IoT		IoT	✓	✗
	Token economy		Token economy	✓	✗
	Contract factory		Contract factory	✓	✗
Users	Platform admin	Users	Platform admin	✓	✓
	Supply chain insider		Supply admin	✓	✓
	Supply chain insider		Supply worker (with notarization)	✗	✓
	Supply chain insider		Supply worker (without notarization)	✗	✓
	Customer		Customer	✓	✓

Table 5.1: Component comparison between the theoretical solution and Deply concrete solution.

The table above compares the components and subcomponents of the system, denoting the solutions as (1) and (2), and specifies whether they are present in each of the two. This table slightly resemble Table 4.1 from the previous chapter, since it has a similar

objective. However, there is a difference: that table only considered the components of Deply system; this one maps all the components mentioned in both models. Furthermore, Table 5.1 has a higher granularity level, with the subcomponents specification.

In the following, the contents provided are commented upon, beginning with the application components, and specifically the front-end. The front-end component in Deply solution is better defined, since the Vercel solution is well-outlined and the subcomponent of the Content Delivery Network is specified.

As far as the back-end is concerned, again model (1) proposed a more general solution, with some subcomponents to accomplish the main functions. Model (2), as described in Section 4.3, does not really have a back-end, but it is a cloud solution, including more functions. Among them, there are Firebase Authentication and Cloud Functions, covering respectively the authentication and triggers theoretical features, but there is also an additional feature. Indeed, Firebase offers an inner database solution, Cloud Firestore, in addition to the other database of the infrastructure. This was not present in the theoretical model.

The actual database is realised as the distributed solution IPFS, guaranteeing all the advantages of a distributed network.

Let's now consider the blockchain part. First, the relayer is concretely implemented by the Pablock service. While the theoretical solution does not go into the details about it, the practical one clarifies that it is made up of different smart contracts, interconnected to each other. Some of them have been mentioned during the description of Deply smart contracts in Section 4.3.2. Their features include the reception and execution of meta-transactions, according to the standards, the token management and the notarization.

As for the blockchain architecture, the proposed solution, i.e., sidechain anchored to permissionless blockchain, is the one adopted for this use case. Also the proposed platforms for the roles of sidechain and mainchain are the same, respectively Polygon and Ethereum.

The remaining aspects to be analysed from Table 5.1 are the smart contracts and the users. Contrary to what it may seem in the first place, they are strongly related by the user management contract. Indeed, establishing roles for users is inherent in the definition of the contract.

If in the first model the distinction is only between platform managers, supply chain insiders and customers of the platform, the concrete implementation is more exhaustive. The design of the `UserManagement` contract states that the users can play four roles: the platform administrator, the supply chain administrators, the supply chain workers and

the customers of the platform. As shown, the supply chain insider role is split. Furthermore, the supply chain worker can in turn be of two types, with or without notarization capabilities.

Moving to the other contracts, the notarization smart contract is included in both models with analogue tasks. The other three subcomponents (each made up of one or more smart contracts) are present in the theoretical model, but not in the concrete one.

There are multiple reasons for such a decision. The main one is related to market demands. Legacy systems of supply chain management include a small involvement of advanced computer technologies. The integration of Deply system, as described in Chapter 4, already poses a sufficient challenge from the perspective of the customers' acceptance. By introducing more features into the new system, the gap with the legacy ones would become too large.

A progressive change, instead, is something that is accepted in an easier way. Based on this, it must be said that the solution previously described is just a first version of Deply project, that will be expanded in the future. Among the foreseen enhancements, the ones described by the theoretical model are taken into account by Deply.

In particular, the most attractive among the three is the IoT enhancement. Making the recording of IoT data on the blockchain automatic would imply a great improvement to supply chains in terms of efficiency. In fact, it allows to eliminate the waiting time and provide the data as soon as they are produced.

The implementation of the token economy and of the smart factory features, instead, requires something more. First, the customer companies should become familiar with the newly-developed platform; then, a minimum training is necessary in order to get to know the mechanisms introduced.

5.2. Smart contracts

The current section switches from the general view of the previous one to the specific view on the smart contracts. In particular, the focus is on the evaluation of the user management and notarization contracts, implemented within the project. A completeness analysis is provided by means of testing procedures. The first part describes how smart contracts have been tested and the tools employed. Following, the actual implementation is presented, with comments about the code snippets supporting the description.

The completeness analysis is carried out through a unit and interaction testing, getting

to reach a 100% coverage. Besides testing the standard execution flows, full coverage is accomplished by artificially creating the conditions under which certain uncommon but wanted situations occur.

Regarding the programming environment used for the tests, it is still used the Truffle suit, already mentioned for the contract implementation in Section 4.3.2. After coding the tests, Truffle allows for the automatic execution of all of them, with a simple terminal command. If the contracts are implemented in Solidity, a language *ad hoc* for this purpose, the tests instead better fit the use of a scripting language, as JavaScript. The latter is exactly the one exploited.

The testing methodology is now described. The underlying idea is to avoid using mocks or stubs, unnecessary in this context. Indeed, Pablock contracts that interact with Deply contracts are already guaranteed to work perfectly, so they can be used for the tests. Moreover, the `Notarization` contract calls `UserManagement`, but the vice versa is not true. Therefore, it is possible to first do unit testing on `UserManagement` contract, and verify its operations alone. Subsequently, it can be performed the testing on `Notarization` contract.

For each smart contract, a test scenario was set, in which calls to the methods are sequential. The tests covered all methods, one by one, in an incremental way. Thus, the data structures were progressively populated, supporting the construction of such situations that needed to be verified.

The tests are of two kinds: the ones on the proper method calls (*positive*), checking for the correct operation of the system, and the ones on improper calls (*negative*), designed to capture the error messages thrown. They are both important for the system availability and robustness. The latter, in particular, are used to capture smart contracts behaviour in unintended situations. Their aim is to prevent unexpected use-cases, exploring all the possible flows of execution.

Concerning the context in which the tests are executed, there are three of them to be gone through one after the other. During contract development, the tests are executed in a local environment, on a blockchain running on the developer's computer. There are various solutions that enable this. The one employed by the author for this specific case is the Ganache application, belonging to the Truffle suite. Ganache provides a ready-to-use local blockchain, along with some accounts in order to execute transactions. Such a blockchain can be tuned from the application, and the accounts can be imported/exported for a future use.

When the contracts are fully implemented and all the local tests work correctly, it is possible to move to the next step. It consists of the deployment and testing on a really distributed environment, though still a testing one: a *testnet*.

A testnet is a copy of a real-world blockchain, devoted to experiments and tests only, without affecting the real chain (called the mainnet, as opposed to the testnet). In order to pay for the deployment and the transactions, an active wallet is required, provisioned with some funds. For the testnet, the tokens are given for free by the so-called *faucets*. Polygon testnet, Mumbai, is where Deply contracts were deployed for further tests after the ones in local. The deployment on Mumbai was done for the two implemented contracts only, since Pablock service already ran on it. Thus, Mumbai was a good benchmark also to confirm the correct interaction with Pablock.

About that, a further step is needed when working on the testnet. Since the Pablock contracts are already deployed and controlled by their developers, the whitelisting process for Deply smart contracts cannot be performed, but must be done by Pablock developer.

Another effect of having the smart contracts deployed on a testnet, with respect to having them only in local, is the possibility to reach them also with external API calls. This makes it possible to perform the system test before the deployment on Polygon mainnet.

At last, after the system test works with the contracts on the testnet, the final step of the testing can be addressed. The deployment is done on the mainnet (i.e., Polygon), in an analogous way as that on the testnet, and all the considerations done before are valid as well.

Following, the unit (and interaction) testing are presented for the user management and the notarization smart contracts. As done for the contract implementation, the full code for the tests is provided in Appendix C.

5.2.1. User management testing

This section describes the unit testing on the `UserManagement` contract. As said, the tests are written in JavaScript and they make use of the `truffle-assertions` package, allowing to write tests for smart contracts in a convenient way.

All the tests are wrapped by the `contract` structure and receive as input the `accounts` list provided by Ganache. The indexes of the `accounts` array ranges from 0 (the contract owner) to 9, but they are only employed up to index 5. This is also valid for the tests on the `Notarization` contract.

Since the blockchain is a decentralised environment, the response to a method call is not immediate: in particular, the returned object is a `Promise`, that is substituted by the actual result as soon as it is ready. This mechanism is made possible by the async/await model. Whenever a function internally includes asynchronous function calls, it must be declared as `async`. All the methods inside the function that requires an asynchronous response are to be prepended by the `await` keyword. This technique is broadly exploited in the following tests.

A peculiar keyword used in the testing context, is `before`. This clause contains a series of instructions to be executed *before* the test snippets, consisting of preliminary settings. For the current contract, the `before` clause contains the definition of the contracts instances, useful to call the contract methods. Beyond the `UserManagement` contract, also the `EIP712MetaTransaction` contract from Pablock is deployed: this is required to execute the meta-transactions, while testing the `pay` function, specified later on.

The following snippet, Listing 15, shows the test definition and some initial assignments of variables. Before the declaration, all the external components are imported, as, for example, the artifacts, providing an abstraction for the smart contracts. The aspects mentioned above are all present in the snippet, as well.

Listing 15 User management test - Definition and preliminary settings.

```
1   const UserManagement = artifacts.require("UserManagement");
2   const MetaTransaction = artifacts.require("EIP712MetaTransaction");
3   ...
4
5   contract("User Management", (accounts) => {
6       let tx = null;
7       ...
8
9       before(async () => {
10          instance = await UserManagement.deployed();
11          metaTransactionInstance = await MetaTransaction.deployed();
12      });
13
14      ...
15  }
```

Before going into the actual tests, a comment is needed. Every test is encapsulated into an `it` clause, which accepts two parameters: a string with a name for the test, and a function - in our case, an asynchronous arrow function, containing the code. The first

allows to distinguish the tests from each other during their execution. The second contains in turn two macro-categories of instructions. It can contains calls to smart contracts methods, that change the state, and *assertions*. The latter allow to test conditions and check for data values. The most used instruction for both tests is the `assert.equal(..)` instruction, that verifies the equality of its two parameters.

Listing 16 shows the first test, on the `signUp` function. In each line of the code, an asynchronous method from the smart contract is called, prefixed with `await`. This testing framework allows to specify additional options to the method calls, devoted to tuning some blockchain parameters, such as the transaction sender, or the gas limit. The `signUp` function is called with the `from` option, that permits to set `accounts[0]` as the sender. The default value is `accounts[0]`, the one used to deploy the contract, i.e., the contract owner.

Furthermore, the test of a changing state function comes with the calls of one or more getters, in order to test if the correct changes occurred.

Listing 16 User management test - Sign-up.

```
1  it("Should register a user", async () => {
2      await instance.signUp({ from: accounts[1] });
3      assert.equal((await instance.getId(accounts[1])).toString(), "1");
4      assert.equal(await instance.userType(accounts[1]), 1); // 1 = UserType.AdF
5      assert.equal(await instance.getAdmin(accounts[1]), 0x0);
6      assert.equal(await instance.hasWorkerSlots(accounts[1]), false);
7  });
```

The test just presented is an example of positive testing. A functional test is considered positive if it verifies the proper execution of a function, in a standard context. However, it also exists the opposite technique, the negative testing. Negative testing consists of checking for anomalies in the execution of a method, by creating certain conditions, that could cause unwanted behaviours.

An example of negative testing is shown in Listing 17. Such example about sign-up is the negative test corresponding to the previous positive one.

Listing 17 User management test - Negative testing for sign-up.

```
it("Should not register an already-registered user", async () => {
    try {
        await instance.signUp({ from: accounts[1] });
    } catch (e) {
        assert(e.message.includes("User already registered."));
        return;
    }
    assert(false);
});
```

Negative testing captures exceptional behaviour, that deviates from the expected flow of execution. The `require` statements in the smart contract fail and revert the transaction, bringing back the values to the previous state.

Because of that, some errors arise, and they must be handled. The way in which it is done is by using the `try/catch` model. In the `try` clause a function call is made which raises an error, passed as a parameter to the `catch` block. The reason of the error `e` is read by the `e.message` attribute. In particular, the name is compared to the expected one by means of the method `includes`.

More `try/catch` statements can be concatenated one after the other; each of them is sequentially evaluated and resolved. The important thing is that a `return` instruction is located as the final instruction of the last `catch`. Indeed, the bad outcome of this kind of tests is established by the execution or not of the `assert(false)` instruction put in the end.

This is the only negative test reported in the current section, even if this pattern is followed during all the testing procedure. More tests of this kind are provided in Appendix C.

Going on with the description, it comes the testing of the `setSubscription` method, shown below.

It covers two possible use cases, the second one for convenience. Such method can be called only by the contract owner, and the two parameters are the address of the subscribing user and the number of PTKs (i.e., Pablock ToKens) provided. The two variants of the method are distinguished according to the address parameter. If the address belongs to an already registered supply admin, the method just sets (and checks for) the PTKs. If, on the other hand, it does not belong to any registered user, a new supply admin is registered with the provided PTKs value.

The tokens can be used to perform notarization but also other operations. In particular, they can be used by supply admins to add or remove workers under their control. PTKs must be a positive value: a negative one would not have any meaning, while 0 corresponds to an inactive subscription and is obtained through the `removeSubscription` method.

Listing 18 User management test - Subscribe an admin.

```
1  it("Should subscribe an admin", async () => {
2      await instance.setSubscription(accounts[1], 20); // 20 actions available
3      assert.equal(await instance.getPTKs({ from: accounts[1] }), 20);
4  });
5
6  it("Should add and subscribe a non-registered admin", async () => {
7      await instance.setSubscription(accounts[2], 30); // 30 actions available
8      assert.equal((await instance.getId(accounts[2])).toString(), "2");
9      assert.equal(await instance.userType(accounts[2]), 1); // 1 = UserType.AdF
10     assert.equal(await instance.getAdmin(accounts[2]), 0x0);
11     assert.equal(await instance.hasWorkerSlots(accounts[2]), false);
12     assert.equal(await instance.getPTKs({ from: accounts[2] }), 30);
13 });
```

Before concretely adding a worker, a supply admin must be given some worker slots, by means of the method `setWorkerBench`, tested in Listing 19. In this case, the values 0 and 1 have the same meaning, i.e., the admin cannot add supply workers. For values higher than 1, the counter parameter specifies the amount of slots (excluded one, reserved for the admin himself) available for the workers.

Listing 19 User management test - Set worker slots.

```
1  it("Should add worker slots for an admin", async () => {
2      await instance.setWorkerBench(accounts[1], 3);
3      assert.equal(await instance.hasWorkerSlots(accounts[1]), true);
4      let workers = await instance.getWorkers(accounts[1]);
5      assert.equal(workers[1].toString(), "2"); // verify the number of free slots
6  });
```

Figure 5.1 below gives an intuitive view of the worker slots for the supply admin. The first one is devoted to the admin himself, while the two other slots are free.

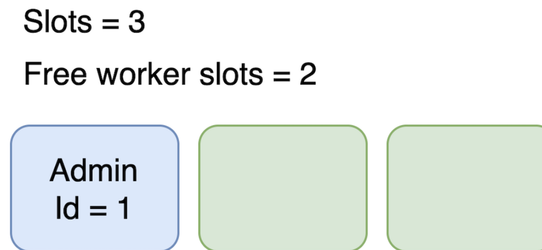

Figure 5.1: Schematic view of the worker slots.

One of the core operations for an admin is the addition of a worker, shown below. This function is normally accomplished by a meta-transaction call and, for this reason, it consumes PTKs. For the purposes of this test, the method is called directly without passing through a meta-transaction. This improper call to `addWorker` is not a problem, since the meta-transactions are covered by the subsequent tests.

The successful outcome of the test is testified by the assertion at line 9 of Listing 20: the remaining PTKs are 19, one less than the 20 set at the beginning.

A remark is made necessary about the two getters `isActionable` and `isRegistered`. They are not tested alone, because their testing is implicit into all the methods that exploit the corresponding modifiers and the `isAdmin` (which internally calls `isRegistered`). The `addWorker` method is an example of where such modifiers are applied.

Listing 20 User management test - Add worker.

```
1      it("Should add a new worker", async () => {
2          await instance.addWorker(accounts[3], { from: accounts[1] });
3          assert.equal((await instance.getId(accounts[3])).toString(), "3");
4          assert.equal(await instance.userType(accounts[3]), 0); // 0 = UserType.LdF
5          assert.equal(await instance.getAdmin(accounts[3]), accounts[1]);
6          // a supply worker never has worker slots
7          assert.equal(await instance.hasWorkerSlots(accounts[3]), false);
8
9          assert.equal(await instance.getPTKs({ from: accounts[1] }), 19);
10         let workers = await instance.getWorkers(accounts[1]);
11         assert.equal(workers[0][0], accounts[3]);
12         assert.equal(workers[1].toString(), "1"); // number of free slots
13     });
```

Again, a view of the admin's worker slots is given in Figure 5.2. The difference is the slot

occupied by the worker just added.

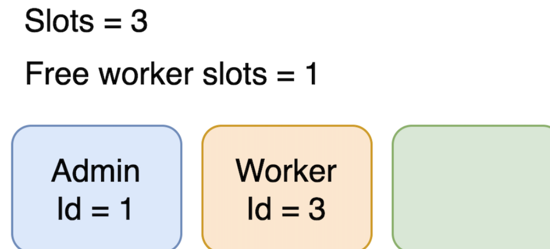

Figure 5.2: Schematic view of the worker slots after a worker has been added.

Listing 21 tests the use case in which a supply admin with 0 PTKs tries to add a worker. As might be expected, the call fails with a "*no more PTKs*" message.

This is just one specific example for the general use case of calling some method without having PTKs. However, notice that in this particular context, even if the admin had some PTKs, the transaction would have been reverted all the same, because he had no available worker slots.

Listing 21 User management test - Transaction failed with no PTKs.

```
1   it("Should not allow transactions without PTKs", async () => {
2       assert.equal(await instance.getPTKs({ from: accounts[2] }), 0);
3       try {
4           await instance.addWorker(accounts[4], { from: accounts[2] });
5       } catch (e) {
6           assert(e.message.includes("no more PTKs"));
7           return;
8       }
9       assert(false);
10  });
```

Let's now turn on a different function: the recover of a supply admin, shown in Listing 22. While, as an admin, recovering a worker can be simply achieved by removing it and adding a new one, the same procedure for a supply admin requires the intervention of Deply platform manager. The recovery allows to modify the admin address: this is accomplished by copying all the state from the old user admin to a new one just subscribed. At the end of the procedure the old admin is deleted and a `Deletion` event is emitted. In order

to test a more complex scenario, a worker was added before the recovery and a check for the presence of the two workers is done after the call of the recovery function.

Listing 22 User management test - Recover an admin.

```
it("Should recover an admin", async () => {
    // add a worker, to have a list of 2 workers
    await instance.addWorker(accounts[4], { from: accounts[1] });
    assert.equal((await instance.getId(accounts[4])).toString(), "4");
    assert.equal(await instance.getPTKs({ from: accounts[1] }), 18);

    await instance.signUp({ from: accounts[5] });
    tx = await instance.recoverAdF(accounts[1], accounts[5]);

    assert.equal((await instance.getId(accounts[5])).toString(), "5");
    assert.equal(await instance.userType(accounts[5]), 1); // 1 = UserType.AdF
    assert.equal(await instance.getPTKs({ from: accounts[5] }), 18);
    assert.equal(await instance.getAdmin(accounts[5]), 0x0);
    assert.equal(await instance.hasWorkerSlots(accounts[5]), true);

    let workersNew = await instance.getWorkers(accounts[5]);
    assert.equal(workersNew[0].length, 2);
    assert.equal(workersNew[0][0], accounts[3]);
    assert.equal(workersNew[0][1], accounts[4]);
    assert.equal(workersNew[1].toString(), "0"); // number of free slots

    assert.equal(await instance.getAdmin(accounts[3]), accounts[5]);
    assert.equal(await instance.getAdmin(accounts[4]), accounts[5]);

    assert.equal(await instance.isRegistered(accounts[1]), false);
    truffleAssert.eventEmitted(tx, "Deletion", (ev) => {
        return (
            ev.userId.toString() === "1" &&
            ev.user === accounts[1] &&
            ev.userType.toString() === "1" && // 1 = UserType.AdF
            ev._from === accounts[0]
        );
    });
});
```

In this respect, we explain how the events have been tested. The `truffleAssert` library provides support for detecting the event emitted during a transaction. First, the transaction outcome is stored into a variable (`tx` at line 8 of the code), then the `eventEmitted`

at line 26 allows to verify that the `Deletion` event occurred with the specified parameter values.

In particular, after the recovery, the admin having as address `accounts[1]` switched to the address of `accounts[5]`.

The situation before an after `recoverAdF` call at line 8 is shown in Figure 5.3. The associated workers and the number of slots and free slots does not change; what change is the admin id, which switches from 1 to 5.

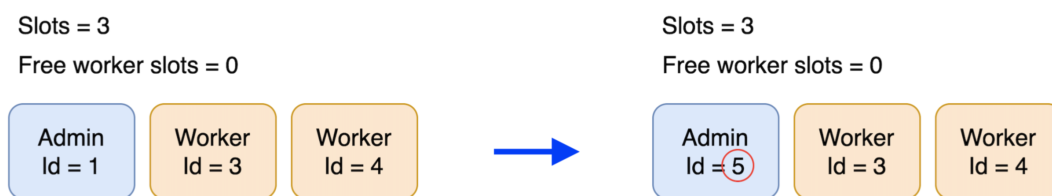

Figure 5.3: Schematic view of the worker slots after the admin's recovery.

Listing 23 presents the test for the `removeWorker` method, only callable from a subscribed supply admin.

Listing 23 User management test - Remove worker.

```
1    it("Should remove a worker", async () => {
2        tx = await instance.removeWorker(accounts[3], { from: accounts[5] });
3        assert.equal(await instance.isRegistered(accounts[3]), false);
4        truffleAssert.eventEmitted(tx, "Deletion", (ev) => {
5            return (
6                ev.userId.toString() === "3" &&
7                ev.user === accounts[3] &&
8                ev.userType.toString() === "0" && // 0 = UserType.LdF
9                ev._from === accounts[5]
10           );
11       });
12
13       assert.equal(await instance.getPTKs({ from: accounts[5] }), 17);
14       let workers = await instance.getWorkers(accounts[5]);
15       assert.equal(workers[0][0], accounts[4]);
16       assert.equal(workers[0].length, 1);
17       assert.equal(workers[1].toString(), "1"); // number of free slots
18   });
```

The worker, whose address is specified as a parameter is disconnected from the admin and deleted from users, emitting a `Deletion` event. Also this method should be called through a meta-transaction and consumes a PTK.

The worker slot scheme after that the worker with id equal to 3 has been removed is shown in the following image (Figure 5.4). Such a worker left a free slot, that may be used for another worker.

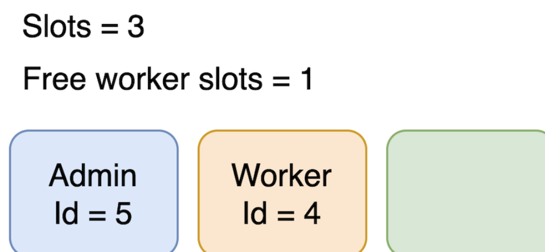

Figure 5.4: Schematic view of the worker slots after the removal of a worker.

A similar method not reported here is the one to remove a user. `removeWorker` works in the same way, but it can be called only by the contract owner. Moreover, if called on an admin, it has the side-effect of deleting also his workers.

Let's now switch to another sub-section of `UserManagement` testing, the one that enables meta-transactions. The following snippets except for the last one are about white-listing contracts and exploiting them to perform a meta-transaction. Listing 24 shows the Pablock contract `metaTransaction` being white-listed, before calling pay function through it.

Listing 24 User management test - White-listing.

```
1    it("Should whitelist a contract", async () => {
2        await instance.whitelistContract(metaTransactionInstance.address);
3        assert.equal(
4            await instance.isWhitelisted(metaTransactionInstance.address),
5            true
6        );
7    });
```

The negative test for this method, not reported here, tests the case in which an address not corresponding to any contract is provided. In that event, the transaction is reverted,

as expected. Black-listing test, also related to this, works in the same way as white-listing one, and it is included in Appendix C.

Going back to the original flow, once the meta-transaction contract is white-listed, the `pay` method can be tested. Two ingredients are introduced in the following snippets, before presenting the test code. The first one is the ABI (Application Binary Interface), imported from the JSON version of the contract, resulting from the building process.

```
const { abi } = require("../build/contracts/UserManagement.json");
```

The ABI is in turn exploited in the function signature definition. By relying on the well-known `web3` library, the function signature is easily defined starting from the ABI. The function signature is also featured by the calling parameters of the function, provided into an array as the second argument of the `encodeFunctionCall`. Below, the specific value assigned to the parameter of `pay` function is `accounts[2]`.

```
const functionSignature = web3.eth.abi.encodeFunctionCall(
    abi.find((el) => el.type === "function" && el.name === "pay"),
    [accounts[2]]
);
```

Hereafter, Listing 25 shows the test code. The starting point is an account set with a valid subscription (i.e., 10 PTKs). `metaTransactionInstance` provides a method to get a `nonce`. The nonce, as well as the function signature, the user address and private key and other parameters, is used by `getTransactionData` to compute the signing parameters `r`, `s`, `v`. The latter three identify the signature in a secure way. For what concerns what we called the *other parameters* mentioned before, they include the contract name, version, address: the first two must be the same as the ones specified in the contract constructor, while the third is known from the deployment.

This long pre-setting phase culminates in the call to a Pablock method, which actually accomplish the meta-transaction, called `executeMetaTransaction`. After execution ends, checks are made that the only tokens consumed are the PTKs (internal to Deply framework) and not the blockchain tokens, for which the balance remains the same.

Listing 25 User management test - Meta-transaction for paying PTKs.

```
    it("Should only call 'pay' through meta-transactions", async () => {
        assert.equal(await instance.getPTKs({ from: accounts[2] }), 10);
        const initBalance = await web3.eth.getBalance(accounts[2]);

        let nonce = await metaTransactionInstance.getNonce(accounts[2]);
        let { r, s, v } = await getTransactionData(
            nonce.toNumber(),
            functionSignature,
            accounts[2],
            privateKey2,
            { name: "deply_user", version: "0.0.1", address: instance.address }
        );

        await metaTransactionInstance.executeMetaTransaction(
            instance.address,
            accounts[2],
            functionSignature,
            r,
            s,
            v
        );

        assert.equal(await instance.getPTKs({ from: accounts[2] }), 9);
        assert.equal(initBalance, await web3.eth.getBalance(accounts[2]));
    });
```

The last use case for which the test is discussed is contained in Listing 26. The scenario motivating it is the following. The contract owner cannot call the method `removeWorker`, but can accomplish the same operation in two ways. The first, already mentioned before, consists of calling the more general method `removeUser` on the worker to remove.

The second - i.e., the one described - is based on the reduction of the worker slots for an admin, by means of `setWorkerBench` called with a sufficiently low parameter. If that happens while the admin has more submitted workers than the remaining slots, then the workers in advance (i.e., from the last on the list), are deleted.

The test shows that mechanism: first, a worker is added and the admin state highlighted. Then the slot number is reduced, implying the worker deletion. The other worker, still fitting into the slots, is not affected by this operation.

Listing 26 User management test - Remove workers by decreasing the number of slots.

```
it("Should remove workers if the number of slots is decreased", async () => {
    await instance.addWorker(accounts[3], { from: accounts[5] });
    assert.equal((await instance.getId(accounts[3])).toString(), "6");
    assert.equal(await instance.getPTKs({ from: accounts[5] }), 15);

    let workers = await instance.getWorkers(accounts[5]);
    assert.equal(workers[0][0], accounts[4]);
    assert.equal(workers[0][1], accounts[3]);
    assert.equal(workers[0].length, 2);
    assert.equal(workers[1].toString(), "0"); // number of free slots

    tx = await instance.setWorkerBench(accounts[5], 2);
    assert.equal(await instance.isRegistered(accounts[3]), false);
    truffleAssert.eventEmitted(tx, "Deletion", (ev) => {
        return (
            ev.userId.toString() === "6" &&
            ev.user === accounts[3] &&
            ev.userType.toString() === "0" && // 0 = UserType.LdF
            ev._from === accounts[0]
        );
    });

    workers = await instance.getWorkers(accounts[5]);
    assert.equal(workers[0][0], accounts[4]);
    assert.equal(workers[0].length, 1);
    assert.equal(workers[1].toString(), "0"); // number of free slots
});
```

Figure 5.5 below shows what just explained and tested. The right scheme represents the state before the execution of `setWorkerBench` at line 12; the left one is the final state, with a total of 2 slots, with no free slots.

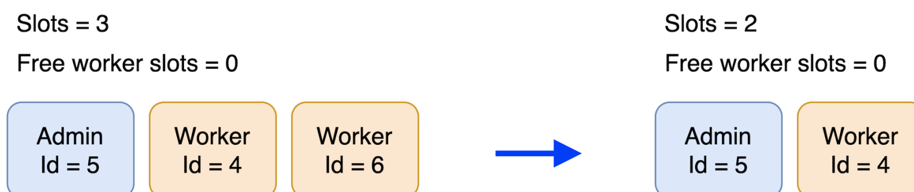

Figure 5.5: Schematic view of the worker slots when it is decreased.

Finally, Figure 5.6 summarises all the executed tests. As we can see, all the tests have been executed at once by means of the Truffle suite.

They tests every flow of execution of the methods defined into the `UserManagement` smart contract. All of them bring to a successful outcome, guaranteeing a high degree of confidence about the smart contract correctness.

```
Contract: User Management
  ✓ Should register a user (722ms)
  ✓ Should not register an already-registered user (3099ms)
  ✓ Should subscribe an admin (445ms)
  ✓ Should add and subscribe a non-registered admin (943ms)
  ✓ Should add worker slots for an admin (372ms)
  ✓ Should not allow for unauthorised calls to 'setWorkerBench' (458ms)
  ✓ Should add a new worker (770ms)
  ✓ Should not add an already-registered worker (144ms)
  ✓ Should not allow for unauthorised subscriptions (291ms)
  ✓ Should remove an admin's subscription (205ms)
  ✓ Should not allow for unauthorised subscription removal (436ms)
  ✓ Should not allow transactions without PTKs (195ms)
  ✓ Should recover an admin (1788ms)
  ✓ Should not allow for unauthosized recoveries of admins (611ms)
  ✓ Should remove a worker (526ms)
  ✓ Should not allow for unauthorised deletion of workers (858ms)
  ✓ Should whitelist a contract (235ms)
  ✓ Should not whitelist an address which is not a contract (156ms)
  ✓ Should only call 'pay' through meta-transactions (841ms)
  ✓ Should make an admin pay PTKs for his workers (810ms)
  ✓ Should not allow for unauthorised calls of 'pay' (1041ms)
  ✓ Should blacklist a contract (285ms)
  ✓ Should remove workers if the number of slots is decreased (1032ms)
  ✓ Should remove a user (2733ms)
  ✓ Should not allow for unathourised deletion of users (610ms)


25 passing (23s)
```

Figure 5.6: Summary of the tests for `UserManagement` smart contract.

5.2.2. Notarization testing

This section describes the testing on the `Notarization` contract. It can't be considered unit testing, since it relies on the `UserManagement` contract and on Pablock contracts. Nevertheless, the tests are likely significant, given that the two mentioned sets of contracts were proven to be correct.

The following snippet, Listing 27, exactly like the corresponding one for `UserManagement`, shows the test definition and some first variable assignments. All the artifacts components are imported, including `Notarization`.

Differently from before, more variables are initialised. Two examples are `initBalance`, used by multiple tests, thus defined just once at the beginning, and a toy example of hash, passed to the notarization functions. In the `before` clause, besides the contract deployment is also included the setting of a user.

Listing 27 Notarization test - Definition and preliminary settings.

```
const Notarization = artifacts.require("Notarization");
const UserManagement = artifacts.require("UserManagement");
const MetaTransaction = artifacts.require("EIP712MetaTransaction");
...

contract("TestNotarization", (accounts) => {
    let initBalance = null;
    let hash =
        "0xe3f699c09d7a6b9a6a4c378bb8611fcfff787856fd629fab1768330e7c95a007";
    let tx = null;
    ...

    before(async () => {
        notarInstance = await Notarization.deployed();
        userManInstance = await UserManagement.deployed();
        metaTransactionInstance = await MetaTransaction.deployed();

        await userManInstance.signUp({ from: accounts[1] });
        await userManInstance.setSubscription(accounts[1], 20);
        // needed to call notarizeDirect
        await userManInstance.whitelistContract(notarInstance.address);
        await userManInstance.whitelistContract(metaTransactionInstance.address);
    });

    ...
}
```

If for the previous smart contract the role of meta-transactions was marginal, it is a main aspect to be tested for the `Notarization` contract. In particular, the function `notarizeDirect` is the one interested by meta-transactions, while `notarize` can be called directly. Following, the signature for the former function is provided. The specified parameters are the hash defined before and a string *uri*, which is a placeholder for an eventual URL.

```
const functionSignature = web3.eth.abi.encodeFunctionCall(
    abi.find((el) => el.type === "function" && el.name === "notarizeDirect"),
    [hash, "uri"]
);
```

The function signature above is exploited for setting up the meta-transaction parameters and executing it. Listing 28 shows the notarization test. All the considerations about meta-transaction made above (see Listing 25) remain valid in this case. At this regard, some code is omitted for the sake of brevity; we highlight only the notable aspects.

First, the contract name specified at line 7 is different with respect to before, since so is the contract. Second, the meta-transaction works so that the call to `notarizeDirect` function is performed within the transaction wrapped by the meta-transaction. Thus, the event emitted is not detected by the usual library call. To do that, it is necessary to define a `result` object (see line 14), specifying the contract instance (`notarInstance`) and the meta-transaction variable (`tx`), to retrieve such event.

Listing 28 Notarization test - Notarization through meta-transaction.

```
1    it("Should call 'notarizeDirect' to notarize through Pablock", async () => {
2        initBalance = await web3.eth.getBalance(accounts[1]);
3        ...
4
5        let { r, s, v } = await getTransactionData(
6          ...,
7          { name: "deply_notar", version: "0.0.1", address: notarInstance.address }
8        );
9
10       tx = await metaTransactionInstance.executeMetaTransaction(
11         ...
12       );
13
14       let result = await truffleAssert.createTransactionResult(
15         notarInstance,
16         tx.tx
17       );
18       truffleAssert.eventEmitted(result, "Record", (ev) => {
19         return (
20           ev.hash === hash && ev.uri === "uri" && ev.applicant === accounts[1]
21         );
22       });
23
24       assert.equal(await userManInstance.getPTKs({ from: accounts[1] }), 19);
25       assert.equal(initBalance, await web3.eth.getBalance(accounts[1]));
26     });
```

Another test exists, not reported in this section, to assess that the execution of a meta-

transaction by a worker achieves the expected outcome. In that setting, as scheduled, the PTKs are consumed by the admin, and no other blockchain tokens are spent.

Continuing the description, Listing 29 provides an example of negative testing for a meta-transaction. To avoid repetition, only the essential parts of the code are included (e.g., function parameters at line 6 are omitted). The use case of the test is that a meta-transaction should fail if the user does not own at least one PTK to pay for it.

Listing 29 Notarization test - Not allowed notarization.

```
it("Should not notarize with Pablock, if not owning PTKs", async () => {
    await userManInstance.removeSubscription(accounts[1]);
    ...

    let { r, s, v } = await getTransactionData(
      ...
    );

    try {
        await metaTransactionInstance.executeMetaTransaction(
            notarInstance.address,
            accounts[1],
            functionSignature,
            r, s, v
        );
    } catch (e) {
        assert(e.message.includes("no more PTKs"));
        ...
        return;
    }
    assert(false);
});
```

As mentioned, `notarize` function allows to perform notarization (by emitting `Record` events) without Pablock intervention. Then, the user sending the transaction must own enough blockchain tokens. `notarize` function was tested in two use cases, i.e., if the user is equipped with PTKs and if he is not.

Listing 30 below shows the test for the second use case and a sketch of the testing for the first one. Since the two tests are pretty much analogous, it is not necessary to provide both. The test checks that the user PTKs do not decrease, while the blockchain tokens are consumed instead.

Listing 30 Notarization test - Notarization without meta-transactions.

```
  it("Should notarize without Pablock by using 'notarize'", async () => {
      initBalance = await web3.eth.getBalance(accounts[1]);
      assert.equal(await userManInstance.getPTKs({ from: accounts[1] }), 0);
      tx = await notarInstance.notarize(hash, "uri not sub", {
          from: accounts[1],
      });

      truffleAssert.eventEmitted(tx, "Record", (ev) => {
          return (
              ev.hash === hash &&
              ev.uri === "uri not sub" &&
              ev.applicant === accounts[1]
          );
      });
      assert(initBalance > (await web3.eth.getBalance(accounts[1])));
      initBalance = await web3.eth.getBalance(accounts[1]);

      // check that PTKs are not consumed
      await userManInstance.setSubscription(accounts[1], 10);
      ...
    });
```

At last, we show a negative test for `notarize` function. The test case is the following: if the user trying to execute a notarization is not registered, as for the user management contract, the transaction is reverted and not executed. The snippet below shows it for a simple transaction, but a further test was also performed for a meta-transaction, with the same result.

Listing 31 Notarization test - Negative test for notarize.

```
  it("Should not call 'notarize' if the user is not registered", async () => {
      try {
          await notarInstance.notarize(hash, "uri", { from: accounts[2] });
      } catch (e) {
          assert(e.message.includes("User not found."));
          return;
      }
      assert(false);
  });
```

A final summary of all the executed tests is displayed by Figure 5.7. As we can see, the amount of tests is lower than before, due to the smaller size of the `Notarization` smart contract. Again, the tests cover every flow of execution of the methods defined into the smart contract, giving guarantees on the correct operation.

```
Contract: Notarization
  ✓ Should call 'notarizeDirect' to notarize through Pablock (1313ms)
  ✓ Should allow for notarization through Pablock also to the workers (1766ms)
  ✓ Should not call 'notarizeDirect' if the user is not registered (3283ms)
  ✓ Should not notarize with Pablock, if not owning PTKs (731ms)
  ✓ Should notarize without Pablock by using 'notarize' (627ms)
  ✓ Should not call 'notarize' if the user is not registered (280ms)


6 passing (12s)
```

Figure 5.7: Summary of the tests for `Notarization` smart contract.

5.3. Usability and functional aspects

The current section gives an analysis about issues related to the usability and user experience, concerning functional and architectural aspects, in the context of Deply project implementation.

For instance, about the functional aspects, the question of how a user can trace a product is deepened; as for the *architectural* ones, matters falling into this categories are the management of the private key, but also the implications of using a relayer.

For each aspect mentioned during the platform description in Section 3.2, it is addressed and the implementation decisions taken are pointed out. Other issues considered important are presented as well.

5.3.1. Integration to legacy systems

The first point presented in Chapter 3 analysis is that the new model must not constitute a disruptive change. This quality is obtained through more measures. The first one is the design of the **user interface**. To have an intuitive user interface is important for every software: it has been enforced during the design phase, and then slightly modified due to refinements during the implementation phase. We remark that the application is accessible from the web and on mobile from the vast majority of the browsers and devices.

In particular, we present the use case of a customer tracing the production processing of some goods. Figure 5.8 displays three screens from the design of the user interface. The views give an outline of the application from the customer's point of view.

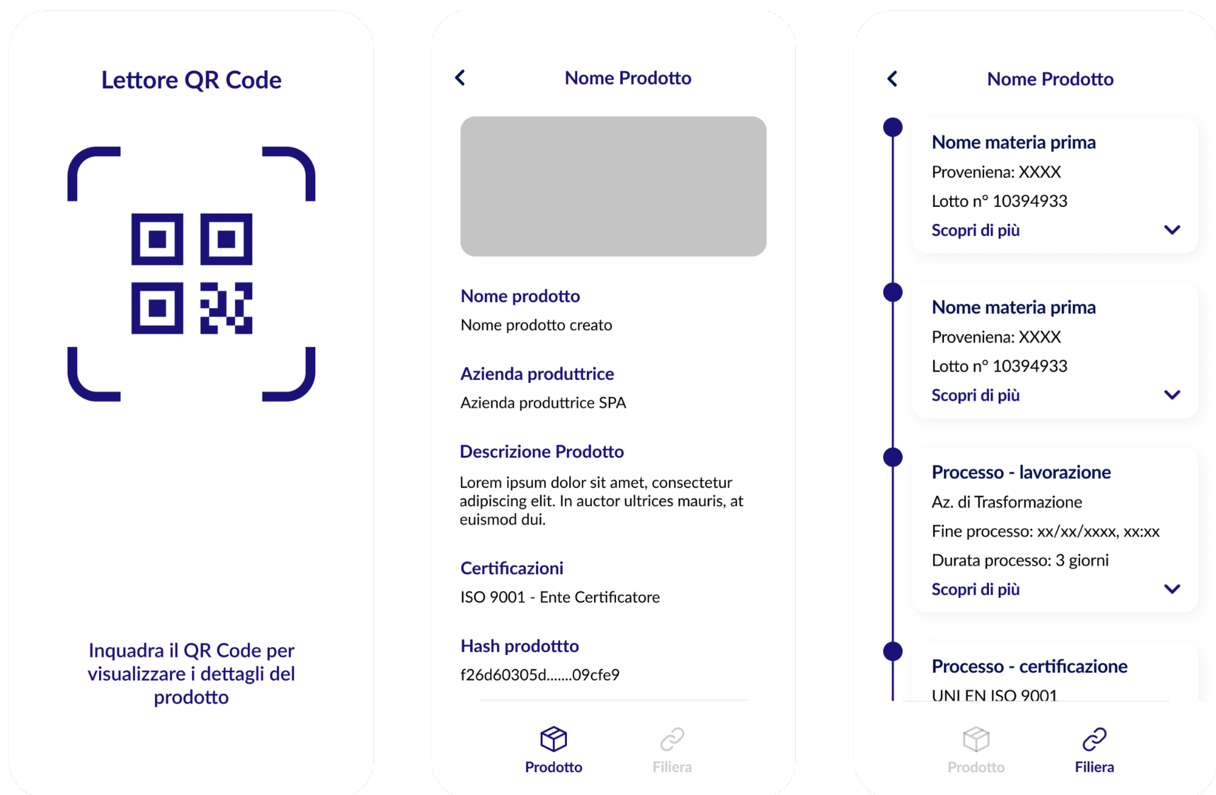

Figure 5.8: Three mock-up screens from Deply mobile application.

From left to right, a QR reader is represented, that allows to retrieve the information about the product, by scanning the QR code printed on the product. Then an information view opens, divided into two sections: product and supply chain. The middle screen shows the product section, containing information about the product name and description, the manufacturer company, and also the hash that certifies it. The screen on the right shows the supply chain processing phases that the product went through before getting into the customer's hands. Notice that these views are just a mock-up, while the actual application reserves a higher space for images, that immediately let the customers identify products and processing steps.

From a user experience perspective, for a customer it is really easy and fast to have access to the information he is seeking, since can be accomplished through an operation as simple as opening the smartphone camera and scanning the QR code. The navigation is intuitive as well, organised and with a restricted number of sections to consult.

A second measure in order to making the platform more usable is the use of a **relayer** service as Pablock. The addition of a further layer between users and blockchain eases the interaction and makes it transparent to all the technicalities usually required. Thus, also for a supply chain insider - not expert in the blockchain field - it is possible to notarize

and benefit from the good properties of the blockchain.

5.3.2. Management of cryptographic keys

Moving to the blockchain aspect, one of the most important aspects is the management of the cryptographic keys. As explained before, two solutions are possible for this question: a custodial or non-custodial handling. The difference between the two is that for a custodial wallet, the private key is held by a third party which is in charge of making the operations required by the users, while a non-custodial wallet is entirely handled by the customer. In a fully decentralised scenario, all the wallets should be non-custodial, giving anyone full control over their currencies.

For this reason, Deply model does not comply with the custodial solution, but assigns to any smart contract user a unique wallet and private key. Anyway, as other non-custodial solutions, Deply stores each user's private key locally on his browser, in order to let him access it and visualise it if the key is forgotten. This solution is secure, since the local storage of the browsers is fully detached from the internet.

To enforce usability and make it easier to remember the private keys, the standard solution is to memorise instead an equivalent information. We are talking about a seed phrase (or recovery phrase), consisting of a 12-word or 24-word series of common words that do not form any semantically meaningful sentence. From the seed phrase, it is possible to retrieve the cryptographic key or set of keys of a wallet, and vice versa.

5.3.3. Notarization of incorrect information

The notarization of incorrect information is another open point that deserves a comment. The aim of this short section is to address it.

The problem of notarizing incorrect information is mitigated in two main ways. First, by preventing some users to actually notarize on the blockchain. This is obtained with the splitting of the supply worker type in two roles, one of which does not have notarization privileges. However, the supply workers without restrictions and the supply admins can still notarize, and, most of all, the nature of the platform is to let users notarize, thus preventing them from doing it is not a solution.

The second mitigation is to introduce an intermediate step in the user interface. This step consists of requiring the approval of an authorised user before notarizing (i.e., authorised users are, again, supply worker with notarization and supply admin). This step imposes a double check on the written data before writing them in an irreversible way, favouring

the detection of errors.

5.3.4. Transaction fees

As anticipated at the end of Section 5.3.1, the interaction between users and blockchain is one of the most critical issues. Besides the point of view of the user interface, which must be intuitive, there is the problem of the transaction fees. Such problem was already presented in Section 3.2.2, during the description of the model: this section only comments the solution taken.

Deply exploits a relayer service - specifically Pablock - to solve the transaction fees problem. The tokens required for executing the meta-transactions are moved internally to the platform. Indeed, the user management smart contract allows to distribute and handle the so-called Pablock ToKens (i.e., PTKs) to pay for transactions within the platform. In particular, each PTK corresponds to a notarization chip. This mechanism and its presentation is really intuitive for the users, that see them as a *Notarizzazioni disponibili* panel on top of their application menu, shown in Figure 5.9 below.

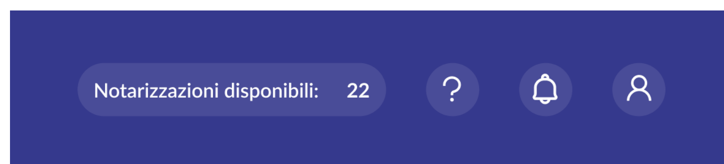

Figure 5.9: View of available operations on Deply application.

However, this system has a weakness related to the centralisation issue, corresponding to the subsequent concern to be addressed. For this reason, the discussion continues hereafter in the next section.

5.3.5. Centralisation

As pointed out from a theoretical perspective in Section 3.2.3, the relayer - as well as the Firebase framework - constitutes a *single point of failure* for the system operation, since it is the only way to interact with the blockchain.

In the actual Deply implementation, the problem can be partially bypassed, in the following way. If Pablock and/or the Deply platform is down, notarization can be directly performed on the blockchain, by personally taking charge of the fees. The additional documents must be separately stored in the IPFS database. IPFS, being distributed, is independent from the application and guarantees a high availability.

The problem is thus bypassed, at the cost of losing in terms of usability: paying the gas cost requires technical knowledge of the blockchain and also the upload to IPFS must be made manually.

However, it is quite hard that the application is offline for a long period, since it relies on Google's Firebase ecosystem, which is really robust (think about the availability of Google search engine and services, which are constantly exploited from all over the world).

Not having a backup path is a choice, also motivated by the intention of Deply client company to keep a certain control over the application. This decision favours usability over decentralisation, and relies on the proper operation of the web application and the relayer. Even if decentralisation is lost, such a requirement is reasonable, given the high reliability of Firebase.

6 | Conclusions and future developments

This is the last chapter of the thesis, containing a summary of the work done and the source of its innovative character. The theoretical framework is summed up, and the concrete implementation results are pointed out.

The rationale behind this work is to provide tools that improve efficiency in the context of supply chain management. Such an idea is realised in a platform model. So, the aim of the thesis becomes to implement a platform, based on the blockchain technology, that could handle supply chains in different sectors, improving their traceability.

The use of blockchain on the one hand brings benefits; on the other hand, poses a large number of challenges, all properly introduced and addressed in the previous chapters.

As specified in Section 3.3, there are some features enforced by the given model, that grant efficiency to the solution. The main ones are: improved traceability, reached through the blockchain transparency and immutability; usability, a critical aspect that has been carefully studied; and versatility to multiple use cases, enforced by modularity. From a blockchain perspective, innovative techniques have been exploited in order to further provide scalability, without losing the other benefits presented.

Among these, the really innovative aspects - not present in any of the works considered for the state of the art - are pointed out. The first are the application *flexibility* and *versatility* to different sectors, also providing traceability to all the supply processing phases. Another is the blockchain *scalability*, obtained by means of a sidechain and anchoring techniques. A third remarkable aspect is the use of a *relayer* to handle transaction costs.

The main limitation of the solution, instead, is that the web application, though distributed, is centralised under the administrators control and constitutes a single point of failure, which is an obstacle to real decentralisation.

Finally, it is worth remarking that Deply model has been implemented, with some adjustments to fit the real-world scenario. Currently, the platform is being put into production,

after system testing ended up successfully. This fact proves that the theoretical model (or something really close to it) can be concretely realised and be effectively useful in reality.

6.1. Future developments

This section is devoted to the discussion of possible enhancements for the implemented system. The model considered is Deply, the realised one, object of Chapter 4. The possible enhancements are mainly located on the blockchain side, being it the most innovative one. As anticipated in Section 5.1, Deply smart contract infrastructure can be provisioned with some further smart contracts, implementing the missing features from the theoretical model.

The first development in chronological order is the development of smart contracts for the IoT. Once realised, such contracts will allow to notarize in an automatic way, as soon as the sensors collect them. Given the increasing spread of IoT technology, this feature will likely be needed in the near future.

The enhancement related to the token economy consists of defining a fungible token, that can be exchanged between the parties subscribed to Deply system. This token is given or taken away according to automatic rules. The design of the smart contracts providing such a feature is certainly feasible, but it would make the platform more complex to understand. That implication would negatively affect system usability. Thus, it must be introduced progressively and in a not invasive way.

The last set of smart contracts is the one devoted to the contract factory. As explained before, a contract factory provides a simplified way for defining smart contracts, containing personalised rules. This feature integrates with the token economy one, since the rules may include a reward or a penalty, according to the occurrence of different events. Also this enhancement, as the previous, is a game changer into the supply chain context, and the considerations made for the token economy also apply here.

Other developments concern the addition of a supply chain from a new sector. This requires to modify the implementation, but not to make breaking changes. Since the system is modular, only the interested part can be changed without affecting the rest.

Furthermore, the analysis of the smart contracts presented in Section 5.2 can be extended to other points of view. For instance, tests could be made about performance. Some interesting properties to be analysed could be their scalability. The kind of questions that such tests could address are: what happens if a huge amount of users interact with the smart contracts at the same time? How many transactions can they support before they

saturate? In particular, the gas fees required to perform the transactions are a remarkable aspect.

Finally, the main limitation of the system is the centralisation introduced by the application, which manages the user accesses to the platform. As anticipated in Section 3.2.3, a possible way to overcome this limit could be delegating the access management to a DAO, a Decentralised Autonomous Organisation. In this case, system entry is handled according to fixed rules, automated by some smart contracts. The characteristics of the DAO and its management should be carefully defined. If that is properly addressed, the system governance would move from being company-driven to being community-driven, achieving complete decentralisation.

Bibliography

[1] Vishal Gaur and Abhinav Gaiha. Building a transparent supply chain blockchain can enhance trust, efficiency, and speed. *Harvard Business Review*, 98(3):94–103, 2020.

[2] Alan T. Sherman, Farid Javani, Haibin Zhang, and Enis Golaszewski. On the origins and variations of blockchain technologies. *IEEE Security & Privacy*, 17(1):72–77, 2019.

[3] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Dec 2008.

[4] Wong Eugene. Retrieving dispersed data from sdd-1: A system for distributed databases. In *Berkeley Workshop*, 1977.

[5] R.C. Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford University, 1979.

[6] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3:99–111, 1991.

[7] Melanie Swan. *Blockchain: Blueprint for a new economy.* " O'Reilly Media, Inc.", 2015.

[8] A.M. Antonopoulos and G.W.P. D. *Mastering Ethereum: Building Smart Contracts and DApps.* O'Reilly Media, 2018.

[9] Anders Brownworth. Blockchain demo - anders brownworth, 2019. https://andersbrownworth.com/blockchain.

[10] Ana Reyna, Cristian Martín, Jaime Chen, Enrique Soler, and Manuel Díaz. On blockchain and its integration with iot. challenges and opportunities. *Future generation computer systems*, 88:173–190, 2018.

[11] Du Mingxiao, Ma Xiaofeng, Zhang Zhe, Wang Xiangwei, and Chen Qijun. A review on consensus algorithm of blockchain. In *2017 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 2567–2572. IEEE, 2017.

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

[13] Samer Hassan and Primavera De Filippi. Decentralized autonomous organization. *Internet Policy Review*, 10(2):1–10, 2021.

[14] Fran Casino, Thomas K. Dasaklis, and Constantinos Patsakis. A systematic literature review of blockchain-based applications: Current status, classification and open issues. *Telematics and Informatics*, 36:55–81, 2019.

[15] M. Casey, J. Crane, G. Gensler, N. Narula, and S. Johnson. *The Impact of Blockchain Technology on Finance: A Catalyst for Change*. Geneva Reports on the World Economy. Centre for Economic Policy Research, 2018.

[16] John Joseph Coyle, Brian J Gibson, C John Langley, and Robert A Novack. *Managing supply chains: A logistics approach*. South-Western Cengage Learning, 2013.

[17] Brian J Gibson, Joe B Hanna, C Clifford Defee, and Haozhe Chen. *The Definitive Guide to Integrated Supply Chain Management: Optimize the Interaction Between Supply Chain Processes, Tools, and Technologies*. Pearson Education, 2014.

[18] A Alicke, J Rachor, and A Seyfert. Supply chain 4.0–the next-generation digital supply chain, mckinsey & company. *Supply Chain Management*, 2016.

[19] Gülçin Büyüközkan and Fethullah Göçer. Digital supply chain: literature review and a proposed framework for future research. *Computers in Industry*, 97:157–177, 2018.

[20] Carrefour. A technological innovation guaranteeing secure and tamperproof product traceability, 2019. https://www.carrefour.com/en/group/food-transition/food-blockchain.

[21] CNBC. Uk hospitals are using blockchain to track the temperature of coronavirus vaccines, 2021. https://www.cnbc.com/2021/01/19/uk-hospitals-use-blockchain-to-track-coronavirus-vaccine-temperature.html.

[22] Hyperledger Foundation. Case study: How walmart brought unprecedented transparency to the food supply chain with hyperledger fabric, 2019.

[23] Jessi Baker, Jutta Steiner, and Gavin Wood. Blockchain: the solution for transparency in product supply chains, 2015.

[24] Sohail Jabbar, Huw Lloyd, Mohammad Hammoudeh, Bamidele Adebisi, and Umar Raza. Blockchain-enabled supply chain: analysis, challenges, and future directions. *Multimedia Systems*, 27(4):787–806, 2021.

[25] ISO-9241-11:2018. Ergonomics of human-system interaction — part 11: Usability: Definitions and concepts. Technical report, International Organization for Standardization, 3 2018.

[26] Texpo srl. Deply, 2022. https://deply.it.

[27] IPFS. Ipfs docs. https://docs.ipfs.io.

[28] BCode. Pablock - notarizzazione.cloud. https://www.pablock.it.

[29] Vercel. Next.js. https://nextjs.org.

[30] Google. Firebase. https://firebase.google.com.

[31] OpenZeppelin. Openzeppelin, 2020. https://openzeppelin.com.

A | Appendix A

This appendix contains the code of the smart contract `UserManagement`, written in the Solidity language.

```
// SPDX-License-Identifier: MIT

pragma solidity >= 0.6.0 < 0.9.0;

import "./pablock/PablockNotarization.sol";
import "./Counters.sol";


contract UserManagement is PablockMetaTxReceiver {
    using Counters for Counters.Counter;

    enum UserType{ LdF, AdF }
    address owner;
    uint256 userId;

    struct User {
        uint256 id;
        UserType userType;
        address admin;
        address[] workers;
        bool isEnrolled;
        Counters.Counter PTKs;
        uint256 workersCounter;
    }

    mapping(address => User) users;
    mapping(address => bool) contracts; // whitelisted contracts for notarization

    event Deletion(uint256 userId, address user, uint256 userType, address _from);


    constructor(
      address metaContract     // address of EIP712MetaTransaction contract
```

```
34    ) PablockMetaTxReceiver("deply_user", "0.0.1") {
35      owner = msgSender();
36      userId = 1;
37      setMetaTransaction(metaContract);
38    }
39
40    modifier onlyOwner {
41      require(msgSender() == owner, "Only the contract owner can call this function."
42      );
43      _;
44    }
45
46    modifier isContract(address _address) {
47      uint32 size;
48      assembly {
49        size := extcodesize(_address)
50      }
51      require(size > 0, "Not contract address.");
52      _;
53    }
54
55    modifier notRegistered(address _address) {
56      require(users[_address].id == 0, "User already registered.");
57      _;
58    }
59
60    modifier registered(address _address) {
61      require(isRegistered(_address), "User not found.");
62      _;
63    }
64
65    modifier actionable(address _address) {
66      require(isActionable(_address), "no more PTKs");
67      _;
68    }
69
70    modifier isAdmin(address _address) {
71      require(isRegistered(_address), "User not found.");
72      require(users[_address].userType == UserType.AdF, "Not an admin.");
73      _;
74    }
75
76
77    // Allow a user to register as AdF, without any subscriptions
78    function signUp() public notRegistered(msgSender()) {
```

```
79        users[msgSender()].id = userId;
80        userId++;
81        users[msgSender()].userType = UserType.AdF;
82      }
83
84      // Check if @param _address belongs to a subscribed user, returns a boolean
85      function isActionable(address _address) public view returns (bool) {
86        if (_address == owner)
87          return false;
88        if (users[_address].userType == UserType.LdF) { // if worker
89          return (users[users[_address].admin].PTKs.current() > 0);
90        } else {
91          return (users[_address].PTKs.current() > 0);
92        }
93      }
94
95      // Check if @param _address belongs to an existing user, returns a boolean
96      function isRegistered(address _address) public view returns (bool) {
97        return (users[_address].id > 0);
98      }
99
100     // Given @param _address, returns the id of the associated user
101     function getId(address _address) public view registered(_address) returns
102     (uint256) {
103       return users[_address].id;
104     }
105
106     // Given @param _address, returns the type of the associated user
107     function userType(address _address) public view registered(_address)
108     returns (UserType) {
109       return users[_address].userType;
110     }
111
112     // Given @param _address, returns the address of the associated user's admin
113     function getAdmin(address _address) public view registered(_address) returns
114     (address) {
115       return users[_address].admin;
116     }
117
118     // Returns the number of available tokens
119     function getPTKs() public view registered(msgSender()) isAdmin(msgSender())
120     returns (uint256) {
121       return users[msgSender()].PTKs.current();
122     }
123
```

```
124    // Given @param _address, checks if belongs to a user with some worker slots
125    // Returns a boolean
126    function hasWorkerSlots(address _address) public view registered(_address)
127    returns (bool) {
128      return (users[_address].workersCounter > 0);
129    }
130
131    // Given @param _address, check that it belongs to an AdF
132    // Returns the list of associated LdF and the number of available worker slots
133    function getWorkers(address _address) public view isAdmin(_address)
134    returns(address[] memory, uint256) {
135      if (users[_address].workersCounter == 0)
136        return (users[_address].workers, 0);
137      else
138        return (users[_address].workers,
139            users[_address].workersCounter - users[_address].workers.length - 1
140        );
141    }
142
143    // Given @param _address, checks if belongs to a whitelisted contract
144    // Returns a boolean
145    function isWhitelisted(address _address) public view returns (bool) {
146      return contracts[_address] == true;
147    }
148
149    // Add new contract @param _address to the whitelist
150    function whitelistContract(address _address) public onlyOwner isContract(_address)
151    {
152      contracts[_address] = true;
153    }
154
155    // Remove contract @param _address from the whitelist
156    function blacklistContract(address _address) public onlyOwner isContract(_address)
157    {
158      contracts[_address] = false;
159    }
160
161    // Internal function to pay PTKs given @param _address
162    function _pay(address _address) internal {
163      if(_address == owner) {
164        return;
165      }
166      if (users[_address].userType == UserType.LdF) { // if worker
167        users[users[_address].admin].PTKs.decrement();
168      } else {
```

```
169        users[_address].PTKs.decrement();
170      }
171    }
172
173    // Public function to pay PTKs outside this contract from @param _address
174    function pay(address _address) public isContract(msg.sender) actionable(_address)
175    {
176      require(isWhitelisted(msg.sender), "Contract is not whitelisted.");
177      _pay(_address);
178    }
179
180    // Given @param _address, set a new subscription with @param _PTKs actions
181    function setSubscription(address _address, uint256 _PTKs) public onlyOwner {
182      if (users[_address].id == 0) { // if not registered: sign-up as a new user
183        users[_address].id = userId;
184        userId++;
185        users[_address].userType = UserType.AdF;
186      } else {
187        require(users[_address].userType == UserType.AdF, "Not an admin.");
188      }
189      if(!users[_address].isEnrolled) { // if counter is null
190        users[_address].isEnrolled = true;
191        users[_address].PTKs = Counters.Counter(0);
192      }
193      users[_address].PTKs.set(_PTKs);
194    }
195
196    // Remove subscription for the AdF with address @param _address
197    function removeSubscription(address _address) public onlyOwner isAdmin(_address) {
198      users[_address].isEnrolled = false;
199      users[_address].PTKs = Counters.Counter(0);
200    }
201
202    // Given an AdF with address @param _address, set the number of worker slots
203    // to @param counter
204    // If counter is decreased, exceeding LdF in workers are removed
205    function setWorkerBench(address _address, uint256 counter) public onlyOwner
206    isAdmin(_address) {
207      if (users[_address].workersCounter > counter) {
208        while (users[_address].workers.length > 0 &&
209        users[_address].workers.length >= counter) {
210          address addrToDelete = users[_address].workers[
211            users[_address].workers.length - 1
212          ];
213          users[_address].workers.pop();
```

```
214         emit Deletion(
215           users[addrToDelete].id, addrToDelete, uint256(UserType.LdF), owner
216         );
217         delete users[addrToDelete];
218       }
219     }
220     users[_address].workersCounter = counter;
221     (, uint256 slots) = getWorkers(_address);
222     assert(slots >= 0);
223   }
224 
225   // A subscribed AdF with enough slots adds a new LdF with address @param _address
226   function addWorker(address _address) public notRegistered(_address)
227   actionable(msgSender()) {
228     (, uint256 slots) = getWorkers(msgSender()); // check that msgSender() is an AdF
229     require(slots > 0, "There are no free slots for the worker.");
230     users[_address].id = userId;
231     userId++;
232     users[_address].userType = UserType.LdF;
233     users[_address].admin = msgSender();
234     users[msgSender()].workers.push(_address);
235     _pay(msgSender());
236   }
237 
238   // A subscribed AdF removes its LdF with address @param _address
239   function removeWorker(address _address) public registered(_address)
240   actionable(msgSender()) {
241     require(users[_address].admin == msgSender(),
242       "You are not the admin for the worker."
243     );
244     uint256 len = users[msgSender()].workers.length;
245 
246     for (uint256 i = 0; i < len; i++) {
247       // swap with the last worker of the array
248       if (users[msgSender()].workers[i] == _address) {
249         if (i < len - 1)
250           users[msgSender()].workers[i] = users[msgSender()].workers[len-1];
251         users[msgSender()].workers.pop(); // deletion through pop function
252         break;
253       }
254     }
255     emit Deletion(users[_address].id, _address, uint256(UserType.LdF), msgSender());
256     delete users[_address];
257     _pay(msgSender());
258   }
```

```
259
260     // The contract owner deletes the user with @param _address
261     // If the user has some LdFs associated, they are deleted as well
262     function removeUser(address _address) public onlyOwner registered(_address) {
263       for (uint256 i = 0; i < users[_address].workers.length; i++) {
264         address addrToDelete = users[_address].workers[i];
265         emit Deletion(
266           users[addrToDelete].id, addrToDelete, uint256(UserType.LdF), owner
267         );
268         delete users[addrToDelete];
269       }
270       emit Deletion(
271         users[_address].id, _address, uint256(users[_address].userType), owner
272       );
273       delete users[_address];
274     }
275
276     // The contract owner deletes the user with address @param _old and
277     // substitutes him with the already registered AdF with address @param _new
278     function recoverAdF(address _old, address _new) public onlyOwner isAdmin(_old)
279     isAdmin(_new) {
280       users[_new].PTKs.set(users[_old].PTKs.current());
281       users[_new].workersCounter = users[_old].workersCounter;
282       users[_new].workers = users[_old].workers;
283       delete users[_old].workers;
284       for (uint256 i = 0; i < users[_new].workers.length; i++) {
285         users[users[_new].workers[i]].admin = _new;
286       }
287       emit Deletion(users[_old].id, _old, uint256(UserType.AdF), owner);
288       delete users[_old];
289     }
290
291 }
```

B | Appendix B

This appendix contains the code of the smart contract `Notarization`, written in the Solidity language.

```
// SPDX-License-Identifier: MIT

pragma solidity >= 0.6.0 < 0.9.0;

import "./UserManagement.sol";


contract Notarization is PablockMetaTxReceiver {
    address owner;
    UserManagement public userContract;

    event Record(bytes32 hash, string uri, address applicant);


    constructor(
      address _userAddress,
      address metaContract    // address of EIP712MetaTransaction contract
    ) PablockMetaTxReceiver("deply_notar", "0.0.1") {
      owner = msgSender();
      userContract = UserManagement(_userAddress);
      setMetaTransaction(metaContract);
    }

    modifier registered(address _address) {
      require(userContract.isRegistered(_address), "User not found.");
      _;
    }

    modifier actionable(address _address) {
      require(userContract.isActionable(_address), "no more PTKs");
      _;
    }

```

```
34
35      // Exploits Pablock service, check that msgSender() is authorised and
36      // emits a Record event with @param _hash and @param uri
37      function notarizeDirect(bytes32 _hash, string memory uri) public
38      registered(msgSender()) actionable(msgSender()) {
39        userContract.pay(msgSender());
40        emit Record(_hash, uri, msgSender());
41      }
42
43      // Notarize without using PTKs, check that msg.sender is authorised and
44      // emits a Record event with @param _hash and @param uri
45      function notarize(bytes32 _hash, string memory uri) public registered(msg.sender)
46      {
47        emit Record(_hash, uri, msg.sender);
48      }
49
50  }
```

C | Appendix C

This appendix contains the code for testing the smart contracts `UserManagement` and `Notarization`. They are written in the JavaScript language, and allow to interact with the deployed smart contracts. Notice that the private keys specified are only used for the tests in local.

C.1. TestUserManagement

```
const UserManagement = artifacts.require("UserManagement");
const MetaTransaction = artifacts.require("EIP712MetaTransaction");

const { abi } = require("../build/contracts/UserManagement.json");
const { getTransactionData } = require("../utility");
const { assert } = require("chai");
const truffleAssert = require("truffle-assertions");

contract("User Management", (accounts) => {
  let tx = null;
  let privateKey2 =
    "5327f2a728d4eaf8ddd668072fc4b3aa58f4f8f867dfd0730e356b3f36a8e9dd";
  let privateKey4 =
    "b009e51652b49d17bee269269fa291aaafb1d867f6de32ec00e588f4c2361130";

  const functionSignature = web3.eth.abi.encodeFunctionCall(
    abi.find((el) => el.type === "function" && el.name === "pay"),
    [accounts[2]]
  );

  before(async () => {
    instance = await UserManagement.deployed();
    metaTransactionInstance = await MetaTransaction.deployed();
  });

  it("Should register a user", async () => {
```

```
27      await instance.signUp({ from: accounts[1] });
28      assert.equal((await instance.getId(accounts[1])).toString(), "1");
29      assert.equal(await instance.userType(accounts[1]), 1); // 1 = UserType.AdF
30      assert.equal(await instance.getAdmin(accounts[1]), 0x0);
31      assert.equal(await instance.hasWorkerSlots(accounts[1]), false);
32    });
33
34    it("Should not register an already-registered user", async () => {
35      try {
36        await instance.signUp({ from: accounts[1] });
37      } catch (e) {
38        assert(e.message.includes("User already registered."));
39        return;
40      }
41      assert(false);
42    });
43
44    it("Should subscribe an admin", async () => {
45      await instance.setSubscription(accounts[1], 20); // 20 actions available
46      assert.equal(await instance.getPTKs({ from: accounts[1] }), 20);
47    });
48
49    it("Should add and subscribe a non-registered admin", async () => {
50      await instance.setSubscription(accounts[2], 30); // 30 actions available
51      assert.equal((await instance.getId(accounts[2])).toString(), "2");
52      assert.equal(await instance.userType(accounts[2]), 1); // 1 = UserType.AdF
53      assert.equal(await instance.getAdmin(accounts[2]), 0x0);
54      assert.equal(await instance.hasWorkerSlots(accounts[2]), false);
55      assert.equal(await instance.getPTKs({ from: accounts[2] }), 30);
56    });
57
58    it("Should add worker slots for an admin", async () => {
59      await instance.setWorkerBench(accounts[1], 3);
60      assert.equal(await instance.hasWorkerSlots(accounts[1]), true);
61      let workers = await instance.getWorkers(accounts[1]);
62      assert.equal(workers[1].toString(), "2"); // verify the number of free slots
63    });
64
65    it("Should not allow for unauthorised calls to 'setWorkerBench'", async () => {
66      try {
67        await instance.setWorkerBench(accounts[1], 3, { from: accounts[1] });
68      } catch (e) {
69        assert(
70          e.message.includes("Only the contract owner can call this function.")
71        );
```

```
72      }
73      try {
74        await instance.setWorkerBench(accounts[3], 3);
75      } catch (e) {
76        assert(e.message.includes("User not found."));
77        return;
78      }
79      assert(false);
80    });
81
82    it("Should add a new worker", async () => {
83      await instance.addWorker(accounts[3], { from: accounts[1] });
84      assert.equal((await instance.getId(accounts[3])).toString(), "3");
85      assert.equal(await instance.userType(accounts[3]), 0); // 0 = UserType.LdF
86      assert.equal(await instance.getAdmin(accounts[3]), accounts[1]);
87      // a LdF never has worker slots
88      assert.equal(await instance.hasWorkerSlots(accounts[3]), false);
89
90      assert.equal(await instance.getPTKs({ from: accounts[1] }), 19);
91      let workers = await instance.getWorkers(accounts[1]);
92      assert.equal(workers[0][0], accounts[3]);
93      assert.equal(workers[1].toString(), "1"); // number of free slots
94    });
95
96    it("Should not add an already-registered worker", async () => {
97      try {
98        await instance.addWorker(accounts[3], { from: accounts[1] });
99      } catch (e) {
100       assert(e.message.includes("User already registered."));
101       assert.equal(await instance.getPTKs({ from: accounts[1] }), 19);
102       return;
103     }
104     assert(false);
105   });
106
107   it("Should not allow for unauthorised subscriptions", async () => {
108     try {
109       await instance.setSubscription(accounts[1], 20, { from: accounts[1] });
110     } catch (e) {
111       assert(
112         e.message.includes("Only the contract owner can call this function.")
113       );
114       assert.equal(await instance.getPTKs({ from: accounts[1] }), 19);
115     }
116     try {
```

```
117        await instance.setSubscription(accounts[3], 10);
118      } catch (e) {
119        assert(e.message.includes("Not an admin."));
120      }
121      try {
122        await instance.setSubscription(accounts[2], -10);
123      } catch (e) {
124        assert(e.message.includes("value out-of-bounds"));
125        return;
126      }
127      assert(false);
128    });
129
130    it("Should remove an admin's subscription", async () => {
131      await instance.removeSubscription(accounts[2]);
132      assert.equal(await instance.getPTKs({ from: accounts[2] }), 0);
133    });
134
135    it("Should not allow for unauthorised subscription removals", async () => {
136      try {
137        await instance.removeSubscription(accounts[1], { from: accounts[2] });
138      } catch (e) {
139        assert(
140          e.message.includes("Only the contract owner can call this function.")
141        );
142      }
143      try {
144        await instance.removeSubscription(accounts[4]);
145      } catch (e) {
146        assert(e.message.includes("User not found."));
147      }
148      try {
149        await instance.removeSubscription(accounts[3]);
150      } catch (e) {
151        assert(e.message.includes("Not an admin."));
152        return;
153      }
154      assert(false);
155    });
156
157    it("Should not allow transactions without PTKs", async () => {
158      assert.equal(await instance.getPTKs({ from: accounts[2] }), 0);
159      try {
160        await instance.addWorker(accounts[4], { from: accounts[2] });
161      } catch (e) {
```

```
162          assert(e.message.includes("no more PTKs"));
163          return;
164        }
165        assert(false);
166      });
167
168      it("Should recover an admin", async () => {
169        // add a worker, to have a list of 2 workers
170        await instance.addWorker(accounts[4], { from: accounts[1] });
171        assert.equal((await instance.getId(accounts[4])).toString(), "4");
172        assert.equal(await instance.getPTKs({ from: accounts[1] }), 18);
173
174        await instance.signUp({ from: accounts[5] });
175        tx = await instance.recoverAdF(accounts[1], accounts[5]);
176
177        assert.equal((await instance.getId(accounts[5])).toString(), "5");
178        assert.equal(await instance.userType(accounts[5]), 1); // 1 = UserType.AdF
179        assert.equal(await instance.getPTKs({ from: accounts[5] }), 18);
180        assert.equal(await instance.getAdmin(accounts[5]), 0x0);
181        assert.equal(await instance.hasWorkerSlots(accounts[5]), true);
182
183        let workersNew = await instance.getWorkers(accounts[5]);
184        assert.equal(workersNew[0].length, 2);
185        assert.equal(workersNew[0][0], accounts[3]);
186        assert.equal(workersNew[0][1], accounts[4]);
187        assert.equal(workersNew[1].toString(), "0"); // number of free slots
188
189        assert.equal(await instance.getAdmin(accounts[3]), accounts[5]);
190        assert.equal(await instance.getAdmin(accounts[4]), accounts[5]);
191
192        assert.equal(await instance.isRegistered(accounts[1]), false);
193        truffleAssert.eventEmitted(tx, "Deletion", (ev) => {
194          return (
195            ev.userId.toString() === "1" &&
196            ev.user === accounts[1] &&
197            ev.userType.toString() === "1" && // 1 = UserType.AdF
198            ev._from === accounts[0]
199          );
200        });
201      });
202
203      it("Should not allow for unauthosized recoveries of admins", async () => {
204        try {
205          await instance.recoverAdF(accounts[1], accounts[5], {
206            from: accounts[1],
```

```
207         });
208       } catch (e) {
209         assert(
210           e.message.includes("Only the contract owner can call this function.")
211         );
212       }
213       try {
214         await instance.recoverAdF(accounts[3], accounts[5]);
215       } catch (e) {
216         assert(e.message.includes("Not an admin."));
217       }
218       try {
219         await instance.recoverAdF(accounts[5], accounts[4]);
220       } catch (e) {
221         assert(e.message.includes("Not an admin."));
222       }
223       try {
224         await instance.recoverAdF(accounts[1], accounts[6]);
225       } catch (e) {
226         assert(e.message.includes("User not found."));
227         return;
228       }
229       assert(false);
230     });
231 
232     it("Should remove a worker", async () => {
233       tx = await instance.removeWorker(accounts[3], { from: accounts[5] });
234       assert.equal(await instance.isRegistered(accounts[3]), false);
235       truffleAssert.eventEmitted(tx, "Deletion", (ev) => {
236         return (
237           ev.userId.toString() === "3" &&
238           ev.user === accounts[3] &&
239           ev.userType.toString() === "0" && // 0 = UserType.LdF
240           ev._from === accounts[5]
241         );
242       });
243 
244       assert.equal(await instance.getPTKs({ from: accounts[5] }), 17);
245       let workers = await instance.getWorkers(accounts[5]);
246       assert.equal(workers[0][0], accounts[4]);
247       assert.equal(workers[0].length, 1);
248       assert.equal(workers[1].toString(), "1"); // number of free slots
249     });
250 
251     it("Should not allow for unauthorised deletion of workers", async () => {
```

```
252     try {
253       await instance.removeWorker(accounts[6]);
254     } catch (e) {
255       assert(e.message.includes("User not found."));
256     }
257     try {
258       await instance.setSubscription(accounts[2], 10);
259       await instance.removeWorker(accounts[4], { from: accounts[2] });
260     } catch (e) {
261       assert(e.message.includes("You are not the admin for the worker."));
262     }
263     try {
264       await instance.removeWorker(accounts[5], { from: accounts[2] });
265     } catch (e) {
266       assert(e.message.includes("You are not the admin for the worker."));
267       return;
268     }
269     assert(false);
270   });
271 
272   it("Should whitelist a contract", async () => {
273     await instance.whitelistContract(metaTransactionInstance.address);
274     assert.equal(
275       await instance.isWhitelisted(metaTransactionInstance.address),
276       true
277     );
278   });
279 
280   it("Should not whitelist an address which is not a contract", async () => {
281     try {
282       await instance.whitelistContract(accounts[1]);
283     } catch (e) {
284       assert(e.message.includes("Not contract address."));
285       return;
286     }
287     assert(false);
288   });
289 
290   it("Should only call 'pay' through meta-transactions", async () => {
291     assert.equal(await instance.getPTKs({ from: accounts[2] }), 10);
292     const initBalance = await web3.eth.getBalance(accounts[2]);
293 
294     let nonce = await metaTransactionInstance.getNonce(accounts[2]);
295     let { r, s, v } = await getTransactionData(
296       nonce.toNumber(),
```

```
297        functionSignature,
298        accounts[2],
299        privateKey2,
300        { name: "deply", version: "0.0.1", address: instance.address }
301      );
302
303      await metaTransactionInstance.executeMetaTransaction(
304        instance.address,
305        accounts[2],
306        functionSignature,
307        r,
308        s,
309        v
310      );
311
312      assert.equal(await instance.getPTKs({ from: accounts[2] }), 9);
313      assert.equal(initBalance, await web3.eth.getBalance(accounts[2]));
314    });
315
316    it("Should make an admin pay PTKs for his workers", async () => {
317      assert.equal(await instance.getPTKs({ from: accounts[5] }), 17);
318      const initBalance = await web3.eth.getBalance(accounts[4]);
319
320      const functionSignatureW = web3.eth.abi.encodeFunctionCall(
321        abi.find((el) => el.type === "function" && el.name === "pay"),
322        [accounts[4]]
323      );
324
325      let nonce = await metaTransactionInstance.getNonce(accounts[4]);
326      let { r, s, v } = await getTransactionData(
327        nonce.toNumber(),
328        functionSignatureW,
329        accounts[4],
330        privateKey4,
331        { name: "deply_user", version: "0.0.1", address: instance.address }
332      );
333
334      await metaTransactionInstance.executeMetaTransaction(
335        instance.address,
336        accounts[4],
337        functionSignatureW,
338        r,
339        s,
340        v
341      );
```

```
342
343     assert.equal(await instance.getPTKs({ from: accounts[5] }), 16);
344     assert.equal(initBalance, await web3.eth.getBalance(accounts[4]));
345   });
346
347   it("Should not allow for unauthorised calls of 'pay'", async () => {
348     try {
349       await instance.pay(accounts[5]);
350     } catch (e) {
351       assert(e.message.includes("Not contract address."));
352     }
353     try {
354       await instance.pay(accounts[5], { from: instance.address });
355     } catch (e) {
356       assert(e.message.includes("Contract is not whitelisted."));
357     }
358     try {
359       await instance.removeSubscription(accounts[2]);
360       let nonce = await metaTransactionInstance.getNonce(accounts[2]);
361       let { r, s, v } = await getTransactionData(
362         nonce.toNumber(),
363         functionSignature,
364         accounts[2],
365         privateKey2,
366         { name: "deply_user", version: "0.0.1", address: instance.address }
367       );
368
369       await metaTransactionInstance.executeMetaTransaction(
370         instance.address,
371         accounts[2],
372         functionSignature,
373         r,
374         s,
375         v
376       );
377     } catch (e) {
378       assert(e.message.includes("no more PTKs"));
379       return;
380     }
381     assert(false);
382   });
383
384   it("Should blacklist a contract", async () => {
385     await instance.blacklistContract(metaTransactionInstance.address);
386     assert.equal(
387       await instance.isWhitelisted(metaTransactionInstance.address),
```

```
388       false
389     );
390   });
391 
392   it("Should remove workers if the number of slots is decreased", async () => {
393     await instance.addWorker(accounts[3], { from: accounts[5] });
394     assert.equal((await instance.getId(accounts[3])).toString(), "6");
395     assert.equal(await instance.getPTKs({ from: accounts[5] }), 15);
396 
397     let workers = await instance.getWorkers(accounts[5]);
398     assert.equal(workers[0][0], accounts[4]);
399     assert.equal(workers[0][1], accounts[3]);
400     assert.equal(workers[0].length, 2);
401     assert.equal(workers[1].toString(), "0"); // number of free slots
402 
403     tx = await instance.setWorkerBench(accounts[5], 2);
404     assert.equal(await instance.isRegistered(accounts[3]), false);
405     truffleAssert.eventEmitted(tx, "Deletion", (ev) => {
406       return (
407         ev.userId.toString() === "6" &&
408         ev.user === accounts[3] &&
409         ev.userType.toString() === "0" && // 0 = UserType.LdF
410         ev._from === accounts[0]
411       );
412     });
413 
414     workers = await instance.getWorkers(accounts[5]);
415     assert.equal(workers[0][0], accounts[4]);
416     assert.equal(workers[0].length, 1);
417     assert.equal(workers[1].toString(), "0"); // number of free slots
418   });
419 
420   it("Should remove a user", async () => {
421     await instance.setWorkerBench(accounts[5], 3);
422     workers = await instance.getWorkers(accounts[5]);
423     assert.equal(workers[1].toString(), "1"); // number of free slots
424 
425     await instance.addWorker(accounts[3], { from: accounts[5] });
426     assert.equal((await instance.getId(accounts[3])).toString(), "7");
427     assert.equal(await instance.getPTKs({ from: accounts[5] }), 14);
428 
429     tx = await instance.removeUser(accounts[5]);
430 
431     // also delete the workers
432     assert.equal(await instance.isRegistered(accounts[4]), false);
```

```
433     truffleAssert.eventEmitted(tx, "Deletion", (ev) => {
434       return (
435         ev.userId.toString() === "4" &&
436         ev.user === accounts[4] &&
437         ev.userType.toString() === "0" && // 0 = UserType.LdF
438         ev._from === accounts[0]
439       );
440     });
441     assert.equal(await instance.isRegistered(accounts[3]), false);
442     truffleAssert.eventEmitted(tx, "Deletion", (ev) => {
443       return (
444         ev.userId.toString() === "7" &&
445         ev.user === accounts[3] &&
446         ev.userType.toString() === "0" && // 0 = UserType.LdF
447         ev._from === accounts[0]
448       );
449     });
450
451     assert.equal(await instance.isRegistered(accounts[5]), false);
452     truffleAssert.eventEmitted(tx, "Deletion", (ev) => {
453       return (
454         ev.userId.toString() === "5" &&
455         ev.user === accounts[5] &&
456         ev.userType.toString() === "1" && // 1 = UserType.AdF
457         ev._from === accounts[0]
458       );
459     });
460   });
461
462   it("Should not allow for unathourised deletion of users", async () => {
463     assert.equal(await instance.isRegistered(accounts[6]), false);
464     assert.equal(await instance.isRegistered(accounts[2]), true);
465
466     try {
467       await instance.removeUser(accounts[6]);
468     } catch (e) {
469       assert(e.message.includes("User not found."));
470     }
471     try {
472       await instance.removeUser(accounts[2], { from: accounts[1] });
473     } catch (e) {
474       assert(
475         e.message.includes("Only the contract owner can call this function.")
476       );
477       return;
```

```
    }
    assert(false);
  });
});
```

C.2. TestNotarization

```
const Notarization = artifacts.require("Notarization");
const UserManagement = artifacts.require("UserManagement");
const MetaTransaction = artifacts.require("EIP712MetaTransaction");

const { abi } = require("../build/contracts/Notarization.json");
const { getTransactionData } = require("../utility");
const truffleAssert = require("truffle-assertions");

contract("Notarization", (accounts) => {
  let initBalance = null;
  let hash =
    "0xe3f699c09d7a6b9a6a4c378bb8611fcfff787856fd629fab1768330e7c95a007";
  let privateKey1 =
    "5d24aff729c73821f2e0c7b8c096a918fdb430bb0209588cc52063318bdfbc35";
  let privateKey2 =
    "5327f2a728d4eaf8ddd668072fc4b3aa58f4f8f867dfd0730e356b3f36a8e9dd";
  let privateKey3 =
    "897621b568528037313fbc9a11acee616af4c7f5dc818351eda446dcd88e700a";
  let tx = null;

  const functionSignature = web3.eth.abi.encodeFunctionCall(
    abi.find((el) => el.type === "function" && el.name === "notarizeDirect"),
    [hash, "uri"]
  );

  before(async () => {
    notarInstance = await Notarization.deployed();
    userManInstance = await UserManagement.deployed();
    metaTransactionInstance = await MetaTransaction.deployed();

    await userManInstance.signUp({ from: accounts[1] });
    await userManInstance.setSubscription(accounts[1], 20);
    // needed to call notarizeDirect
    await userManInstance.whitelistContract(metaTransactionInstance.address);
    await userManInstance.whitelistContract(notarInstance.address);
```

```
36  });
37
38  it("Should call 'notarizeDirect' to notarize through Pablock", async () => {
39    initBalance = await web3.eth.getBalance(accounts[1]);
40    assert.equal(await userManInstance.getPTKs({ from: accounts[1] }), 20);
41    assert.equal(
42      await userManInstance.isWhitelisted(metaTransactionInstance.address),
43      true
44    );
45
46    let nonce = await metaTransactionInstance.getNonce(accounts[1]);
47    let { r, s, v } = await getTransactionData(
48      nonce.toNumber(),
49      functionSignature,
50      accounts[1],
51      privateKey1,
52      { name: "deply_notar", version: "0.0.1", address: notarInstance.address }
53    );
54
55    tx = await metaTransactionInstance.executeMetaTransaction(
56      notarInstance.address,
57      accounts[1],
58      functionSignature,
59      r,
60      s,
61      v
62    );
63
64    let result = await truffleAssert.createTransactionResult(
65      notarInstance,
66      tx.tx
67    );
68    truffleAssert.eventEmitted(result, "Record", (ev) => {
69      return (
70        ev.hash === hash && ev.uri === "uri" && ev.applicant === accounts[1]
71      );
72    });
73
74    assert.equal(await userManInstance.getPTKs({ from: accounts[1] }), 19);
75    assert.equal(initBalance, await web3.eth.getBalance(accounts[1]));
76  });
77
78  it("Should allow for notarization through Pablock also to the workers", async (
79  ) => {
80    await userManInstance.setWorkerBench(accounts[1], 3);
```

```
81      await userManInstance.addWorker(accounts[3], { from: accounts[1] });
82
83      assert.equal(await userManInstance.getPTKs({ from: accounts[1] }), 18);
84      initBalance = await web3.eth.getBalance(accounts[3]);
85
86      let nonce = await metaTransactionInstance.getNonce(accounts[3]);
87      let { r, s, v } = await getTransactionData(
88        nonce.toNumber(),
89        functionSignature,
90        accounts[3],
91        privateKey3,
92        { name: "deply_notar", version: "0.0.1", address: notarInstance.address }
93      );
94
95      tx = await metaTransactionInstance.executeMetaTransaction(
96        notarInstance.address,
97        accounts[3],
98        functionSignature,
99        r,
100       s,
101       v
102     );
103
104     let result = await truffleAssert.createTransactionResult(
105       notarInstance,
106       tx.tx
107     );
108     truffleAssert.eventEmitted(result, "Record", (ev) => {
109       return (
110         ev.hash === hash && ev.uri === "uri" && ev.applicant === accounts[3]
111       );
112     });
113
114     assert.equal(await userManInstance.getPTKs({ from: accounts[1] }), 17);
115     assert.equal(initBalance, await web3.eth.getBalance(accounts[3]));
116   });
117
118   it("Should not call 'notarizeDirect' if the user is not registered", async () => {
119     try {
120       await notarInstance.notarizeDirect(hash, "uri", { from: accounts[2] });
121     } catch (e) {
122       assert(e.message.includes("User not found."));
123     }
124
125     const nonce = await metaTransactionInstance.getNonce(accounts[2]);
```

```
126      let { r, s, v } = await getTransactionData(
127        nonce.toNumber(),
128        functionSignature,
129        accounts[2],
130        privateKey2,
131        { name: "deply_notar", version: "0.0.1", address: notarInstance.address }
132      );
133      try {
134        await metaTransactionInstance.executeMetaTransaction(
135          notarInstance.address,
136          accounts[2],
137          functionSignature,
138          r,
139          s,
140          v
141        );
142      } catch (e) {
143        assert(e.message.includes("User not found."));
144        return;
145      }
146      assert(false);
147    });
148
149    it("Should not notarize with Pablock, if not owning PTKs", async () => {
150      await userManInstance.removeSubscription(accounts[1]);
151      initBalance = await web3.eth.getBalance(accounts[1]);
152
153      const nonce = await metaTransactionInstance.getNonce(accounts[1]);
154      let { r, s, v } = await getTransactionData(
155        nonce.toNumber(),
156        functionSignature,
157        accounts[1],
158        privateKey1,
159        { name: "deply_notar", version: "0.0.1", address: notarInstance.address }
160      );
161
162      try {
163        await metaTransactionInstance.executeMetaTransaction(
164          notarInstance.address,
165          accounts[1],
166          functionSignature,
167          r,
168          s,
169          v
170        );
```

```
171      } catch (e) {
172        assert(e.message.includes("no more PTKs"));
173        assert.equal(await userManInstance.getPTKs({ from: accounts[1] }), 0);
174        assert.equal(initBalance, await web3.eth.getBalance(accounts[1]));
175        return;
176      }
177      assert(false);
178    });
179
180    it("Should notarize without Pablock by using 'notarize'", async () => {
181      initBalance = await web3.eth.getBalance(accounts[1]);
182      assert.equal(await userManInstance.getPTKs({ from: accounts[1] }), 0);
183      tx = await notarInstance.notarize(hash, "uri not sub", {
184        from: accounts[1],
185      });
186
187      truffleAssert.eventEmitted(tx, "Record", (ev) => {
188        return (
189          ev.hash === hash &&
190          ev.uri === "uri not sub" &&
191          ev.applicant === accounts[1]
192        );
193      });
194      assert(initBalance > (await web3.eth.getBalance(accounts[1])));
195      initBalance = await web3.eth.getBalance(accounts[1]);
196
197      // check that PTKs are not consumed
198      await userManInstance.setSubscription(accounts[1], 10);
199      tx = await notarInstance.notarize(hash, "uri sub", { from: accounts[1] });
200
201      truffleAssert.eventEmitted(tx, "Record", (ev) => {
202        return (
203          ev.hash === hash && ev.uri === "uri sub" && ev.applicant === accounts[1]
204        );
205      });
206      assert.equal(await userManInstance.getPTKs({ from: accounts[1] }), 10);
207      assert(initBalance > (await web3.eth.getBalance(accounts[1])));
208    });
209
210    it("Should not call 'notarize' if the user is not registered", async () => {
211      try {
212        await notarInstance.notarize(hash, "uri", { from: accounts[2] });
213      } catch (e) {
214        assert(e.message.includes("User not found."));
215        return;
```

```
216     }
217     assert(false);
218   });
219 });
```

List of Figures

1.1 Blockchain demo from Anders Brownworth [9]. 11
1.2 Sidechain anchored to mainchain. 18
1.3 Blockchain application domains. 24

3.1 Most general architectural view of the system. 38
3.2 Component view of the system architecture. 40
3.3 Detailed architecture of the blockchain part of the system. 42
3.4 Complete smart contract infrastructure. 45
3.5 Complete system architecture of the model. 49

4.1 Mock-up scheme of the pharmaceutical chain process and interaction with the blockchain. 56
4.2 Mobile view of a specific step of the process: a transportation company views the deliveries (on the left) and the details of one product (on the right). 56
4.3 System architecture of Deply. 60
4.4 Architectural view of the application. 64

5.1 Schematic view of the worker slots. 89
5.2 Schematic view of the worker slots after a worker has been added. 90
5.3 Schematic view of the worker slots after the admin's recovery. 92
5.4 Schematic view of the worker slots after the removal of a worker. 93
5.5 Schematic view of the worker slots when it is decreased. 96
5.6 Summary of the tests for `UserManagement` smart contract. 97
5.7 Summary of the tests for `Notarization` smart contract. 102
5.8 Three mock-up screens from Deply mobile application. 103
5.9 View of available operations on Deply application. 105

List of Tables

4.1 Component mapping from the proposed model and Deply solution. 61

5.1 Component comparison between the theoretical solution and Deply concrete solution. 80

List of Listings

1 User management - Contract declaration and constructor. 68
2 User management - Variables and data structures. 69
3 User management - Modifiers. 70
4 User management - Sign-up function. 70
5 User management - Getters. 71
6 User management - Subscriptions. 72
7 User management - White-listing contracts. 73
8 User management - Pay function. 73
9 User management - Add worker. 74
10 User management - Remove worker. 75
11 User management - Recovery of an admin. 75
12 Notarization - Contract declaration and constructor. 76
13 Notarization - Modifiers. 77
14 Notarization - Notarize functions. 78
15 User management test - Definition and preliminary settings. 85
16 User management test - Sign-up. 86
17 User management test - Negative testing for sign-up. 87
18 User management test - Subscribe an admin. 88
19 User management test - Set worker slots. 88
20 User management test - Add worker. 89
21 User management test - Transaction failed with no PTKs. 90
22 User management test - Recover an admin. 91
23 User management test - Remove worker. 92
24 User management test - White-listing. 93
25 User management test - Meta-transaction for paying PTKs. 95
26 User management test - Remove workers by decreasing the number of slots. 96
27 Notarization test - Definition and preliminary settings. 98
28 Notarization test - Notarization through meta-transaction. 99
29 Notarization test - Not allowed notarization. 100
30 Notarization test - Notarization without meta-transactions. 101

31 Notarization test - Negative test for notarize. 101

Acknowledgements

Desidero ringraziare in primo luogo i professori che mi hanno guidato lungo il percorso di scrittura della tesi, il mio relatore Francesco Bruschi e il mio correlatore Vincenzo Rana, per i loro preziosi consigli e suggerimenti.

Un ringraziamento speciale va a mia zia, alla quale questa tesi è dedicata, per la disponibilità e il sostegno che mi ha dato durante il mio percorso universitario. Oltre ad essersi resa disponibile per darmi un appoggio a Milano, mi ha sempre trasmesso un grande affetto.

Come dimenticare la mia famiglia: ringrazio mamma e papà, per avermi dato l'opportunità di intraprendere questo percorso di studi ed aver creduto in me. Ringrazio loro e i miei fratelli per avermi sopportato, in particolare mia mamma per i validi consigli.

Una menzione particolare va a Elly, che c'è stata per tutto il mio cammino universitario. La ringrazio di cuore per avermi supportato e sostenuto nei momenti di difficoltà, e per non aver sbottato (troppo) per le mie lamentele.

Ringrazio i miei compagni di università, quelli conosciuti dal primo giorno, Emanuele, Enrico, Mike e Agostino e quelli che si sono aggiunti durante il percorso, Alessandro, Alessio (gli "Ale"), Camilla e Davide. Grazie per i bei momenti passati insieme a voi e a tutti gli altri compagni, tra lezioni, pause pranzo al solito posto e partite di pallavolo. Ringrazio inoltre gli amici dell'Erasmus, con cui ho condiviso le belle e brutte esperienze.

Ringrazio gli amici di casa, che riesco a vedere poco (dicono che sono "sempre in giro"): in particolare, gli amici di "Serata Easy", quelli di Maccagno e i miei compagni di calcio, senza dimenticare chi ho perso di vista ultimamente, ma con cui ho vissuto dei fantastici momenti.

Vorrei ringraziare i colleghi di Knobs, con cui ho lavorato al progetto Deply e che, putroppo, ho conosciuto quasi solo da remoto. Infine, vorrei ringraziare tutti coloro che hanno contribuito alla mia formazione universitaria e alla mia crescita fino ad ora.

Alessandro
Maccagno, 28 aprile 2022