



Instituto Politécnico
de Viana do Castelo

USING SMART CONTRACTS TO ENHANCE AN INDUSTRIAL SYMBIOSIS PROCESS

Ricardo Miguel Silva Gonçalves

Escola Superior de Tecnologia e Gestão



Instituto Politécnico
de Viana do Castelo

Nome completo do(a) candidato(a)

RICARDO MIGUEL SILVA GONÇALVES

Nome do curso de Mestrado

Mestrado em Cibersegurança

Trabalho efetuado sob a supervisão de

Professor Pedro Filipe Cruz Pinto

Professor António Alberto dos Santos Pinto

Setembro de 2022



Mestrado em
Cibersegurança
Master in
Cybersecurity

Using Smart Contracts to Enhance an Industrial Symbiosis Process

a master's thesis authored by

Ricardo Miguel Silva Gonçalves

and supervised by

Pedro Filipe Cruz Pinto

Professor Adjunto, Instituto Politécnico de Viana do Castelo

António Alberto dos Santos Pinto

Professor Coordenador, Instituto Politécnico do Porto

This thesis was submitted in partial fulfilment of the requirements for the
Master's degree in Cybersecurity at the Instituto Politécnico de Viana do Castelo



18 January, 2022



Abstract

Industrial Symbiosis is a subfield of Circular Economy that tries to solve the issue of the enormous amount of Industrial Waste. The main objective of this field is to create networks of companies where a by-product/waste of one is one of the principal resources of another one.

To run these types of Industrial Symbiosis networks, companies require the creation and maintenance of trusted and transparent relationships between all entities. These relationships are a constant challenge to maintain. When a new entity wants to join the network, it requires trust in each current member.

In an Industrial Symbiosis context, a blockchain-based system can reduce the work necessary to establish and maintain these networks. The system can serve as a ground truth between all the entities operating at a national or global scale, removing all the overhead of establishing and maintaining relationships between all parties.

This thesis proposes a scalable and modular blockchain architecture design using smart contracts to enhance the industrial symbiosis process of the Pulp, Paper, and Cardboard Production Sector companies in Portugal. The system will implement all the requirements using smart contracts. The design comprehends all entities participating in the network, namely the Pulp and Paper companies, the Sand Producers, the Mortar Producers, the organization that uses the resulting material mixture, and the Environmental Portuguese Agency (EPA) [13]. The base blockchain technology used to implement the architecture is Hyperledger Fabric, by being a permissioned ledger, prevents unwanted accesses to the network. Fabric also provides the required trust and transparency between all entities.

Keywords: blockchain. smart contracts. industrial symbiosis. hyperledger fabric.

Resumo

A Simbiose Industrial é um subcampo da Economia Circular que tenta solucionar o problema da quantidade de Resíduos Industriais. O principal objetivo deste campo é o de criar redes de empresas, onde um subproduto/resíduo de uma empresa é um dos principais recursos de outra.

Para operar esses tipos de redes de simbiose industrial, as empresas exigem a criação e manutenção de relacionamentos confiáveis e transparentes entre todas as entidades. A manutenção de tais relacionamentos configura-se como um desafio constante. Quando uma nova entidade deseja ingressar na rede, esta necessita confiar em todos membros atuais.

Num contexto de Simbiose Industrial, um sistema baseado em blockchain pode reduzir o trabalho necessário para estabelecer e manter essas redes. O sistema pode servir como uma verdade fundamental entre todas as entidades que operam em escala nacional ou global, removendo toda a sobrecarga de estabelecer e manter relacionamentos entre todas as partes que participam.

Esta tese propõe um projeto de arquitetura de blockchain escalável e modular utilizando smart contracts para potencializar o processo de simbiose industrial das empresas do Setor de Produção de Pasta, Papel e Cartão em Portugal. O sistema implementa todos os requisitos usando smart contracts. O projecto abrange todas as entidades participantes na rede, nomeadamente as empresas de Pasta e Papel, os Produtores de Areia, os Produtores de Argamassas e a Agência Portuguesa do Ambiente [13]. A tecnologia de blockchain de base usada para implementar a arquitetura é Hyperledger Fabric, sendo *permissioned* impede acessos indesejados à rede. O mecanismo também fornece a confiança e a transparência necessárias entre todas as entidades.

Palavras-chave: blockchain. smart contracts. simbiose industrial. hyperledger fabric.

Acknowledgements

The fulfillment of this thesis required enormous effort, commitment, devotion, and focus during this last year. It would be impossible to develop this work without the support and help of some people, so I would like to mention here my sincere gratitude.

I want to thank Professors Pedro Pinto and António Pinto, supervisors of this thesis, for their support, knowledge, and valuable contributions to this document. Overall, thank you for escorting me on this journey.

I also want to thank all friends and colleagues who, directly or indirectly, contributed and helped with all the patience, care, and energy they provided during this work.

Finally, I want to thank my family for always supporting me and pushing me further in my academic progress.

Contents

List of Figures	v
List of Tables	vi
List of Listings	vii
List of Abbreviations	ix
1 Introduction	1
1.1 Context	2
1.2 Objective	3
1.3 Contributions	3
1.4 Organization	4
2 Background and Related Work	5
2.1 Blockchain & Smart contracts	5
2.2 Ethereum	8
2.3 Hyperledger Fabric	10
2.3.1 Orderers	10
2.3.2 Peers	10
2.3.3 Chaincode	11
2.3.4 Channels	11
2.4 Ethereum vs Hyperledger	12
2.5 Related Work on Application of smart contracts	13

3	Proposed Solution	15
3.1	Requirements	16
3.2	Design	16
3.3	Architecture	19
4	Implementation	22
4.1	General files	23
4.2	Unit	27
4.2.1	Structure	27
4.2.2	Get Unit ID	28
4.2.3	Create Unit	28
4.2.4	Unit Exists	30
4.2.5	Get Unit	31
4.2.6	List Units	33
4.2.7	Delete Unit	34
4.3	Product	35
4.3.1	Structure	35
4.3.2	Get Product ID	36
4.3.3	Product Exist	37
4.3.4	Create Product	37
4.3.5	Get Product	39
4.3.6	List Products	41
4.3.7	Delete Product	42
4.4	Organization	43
4.4.1	Structure	43
4.4.2	Get Organization ID	45
4.4.3	Organization Exist	45
4.4.4	Create Organization	45
4.4.5	Update Organization	47
4.4.6	Get Organization	49
4.4.7	List Organizations	51

4.4.8	Delete Organization	52
4.5	Order	53
4.5.1	Structure	53
4.5.2	Order Type	55
4.5.3	Order Status	55
4.5.4	Get Order	56
4.5.5	Order Exist	57
4.5.6	Create Order	58
4.5.7	Close Order	61
4.5.8	Get Order	62
4.5.9	List Orders	64
4.6	Transaction	69
4.6.1	Structure	69
4.6.2	Transaction Status	70
4.6.3	Get Transaction ID	72
4.6.4	Transaction Exist	72
4.6.5	Make Transaction	73
4.6.6	Change Transaction Status	76
4.6.7	Get Transaction	79
4.6.8	List Transactions	81
5	Results and Analysis	84
5.1	Creation a New Organization	84
5.2	Removing an Organization	85
5.3	Creating a new sell order	85
5.4	Buying a product	86
6	Conclusion	90
	References	92

List of Figures

1.1	Industrial Symbiosis network scenario	2
2.1	Example blockchain	6
2.2	Broken blockchain	7
2.3	Valid blockchain	8
2.4	Trial and Error Proof-of-Work (PoW)	9
2.5	Endorsement and Commitment peer	11
2.6	Channels Example	11
3.1	Industrial Symbiosis Network for the Fluidized Bed Sands (FBS) case study	16
3.2	Data Relationship Diagram	17
3.3	Use Cases	18
3.4	Architecture of the proposed solution	19
4.1	File structure	22

List of Tables

2.1	Traditional vs Smart contracts	7
2.2	Differences between Hyperledger Fabric and Ethereum	12

List of Listings

4.1	main.go file contents	23
4.2	attributes roles definition	23
4.3	attributes.go file contents	24
4.4	doc.go file contents	25
4.5	contract.go file contents	26
4.6	Price structure	26
4.7	Unit Structure	27
4.8	Get Unit ID	28
4.9	Create Unit	29
4.10	Unit Exist	30
4.11	Get Unit and Get Unit Inner	31
4.12	List Units	33
4.13	Delete Unit	34
4.14	Product Structure	35
4.15	Get Product ID	36
4.16	Product Exist	37
4.17	Create Product	37
4.18	Create Product	39
4.19	List Products	41
4.20	Delete Product	42
4.21	Organization Structure	43
4.22	Get Organization ID	45
4.23	Organization Exist	45

4.24	Create Organization	45
4.25	Update Organization	47
4.26	Get Organization and Get Organization Inner	49
4.27	List Organizations	51
4.28	Delete Organization	52
4.29	Order Structures	53
4.30	Order Mapper	54
4.31	Order Type	55
4.32	Order Status	55
4.33	Build Order ID	56
4.34	Order Exists	57
4.35	Create Order	58
4.36	Close Order	61
4.37	Get Order	63
4.38	Get Order Inner	63
4.39	List Orders	64
4.40	Transaction Structure	69
4.41	Transaction Status	70
4.42	Get Transaction ID	72
4.43	Transaction Exist	72
4.44	Make Transaction	73
4.45	Change Transaction Status	76
4.46	Get Transaction and Get Transaction Inner	79
4.47	List Transactions	81
5.1	List Products	85
5.2	Create Order	86
5.3	List Orders	86
5.4	Make Transaction	88
5.5	List Transactions	88

List of Abbreviations

CA Certificate Authority

CLI Command-Line Interface

EPA Environmental Portuguese Agency

FBS Fluidized Bed Sands

PoS Proof-of-Stake

PoW Proof-of-Work

Chapter 1

Introduction

The term "symbiosis" refers to "a close connection between different types of organisms in which they live together and benefit from each other" [26]. In an industry context, an Industrial Symbiosis can be considered as a mutually beneficial relationship among companies to achieve a productive usage of by-products and waste [9]. The Industrial Symbiosis has a collection of approaches to gain an advantage by involving physical exchanges of materials, through local and regional economies [8]. This happens due to the trade of by-products and utility sharing that occurs between industries, which include the reuse and commercialization of waste that can be used as secondary raw material [17]. As [8] cites, Industrial Symbiosis engages traditionally separate industries in a collective approach to competitive advantage involving the physical exchange of materials, energy, water, and/or by-products.

For the Industrial Symbiosis network to succeed in this scenario it requires that all these companies create and maintain a relationship with each other; the bigger the network, the hardest it is to maintain and create the relationships. Incoming companies must trust and should be trusted by the companies already in the network, and the relation and transactions between companies in the network should be clearly defined and auditable. The Blockchain technology can meet this requirement by design since it allows Smart Contracts to run automatically, executing all or parts of an agreement [24] and, at the same time, each transaction that is recorded on the blockchain cannot be changed once created, and can be verified or audited. The use of a cryptocurrency was not considered, mainly because most countries still do not legally recognize cryptocurrencies as a legal

form of money.

Currently, the field of Industrial Symbiosis has several issues when creating new and larger networks. These issues are due to the requirement that all these companies create and maintain a relationship with each other. On larger systems, the problem is more noticeable because each entity on the system needs a connection with the necessary entities to be trustable. The problem additionally exists when a new entity joins the network, mainly due to the new entity being requiring to trust and be trusted by existing network members.

Implementing a blockchain architecture as a support basis for on an Industrial Symbiosis system can solve the previously mentioned problems. The entities would only need to trust the blockchain system and nothing else. The logic required to make the trades and transactions between every member of the network can be implemented by smart contracts that can run automatically on the network.

1.1 Context

The Portuguese Pulp, Paper, and Cardboard industry, an Industrial Symbiosis network in Portugal, as described in [13], was the one selected to be better analysed.

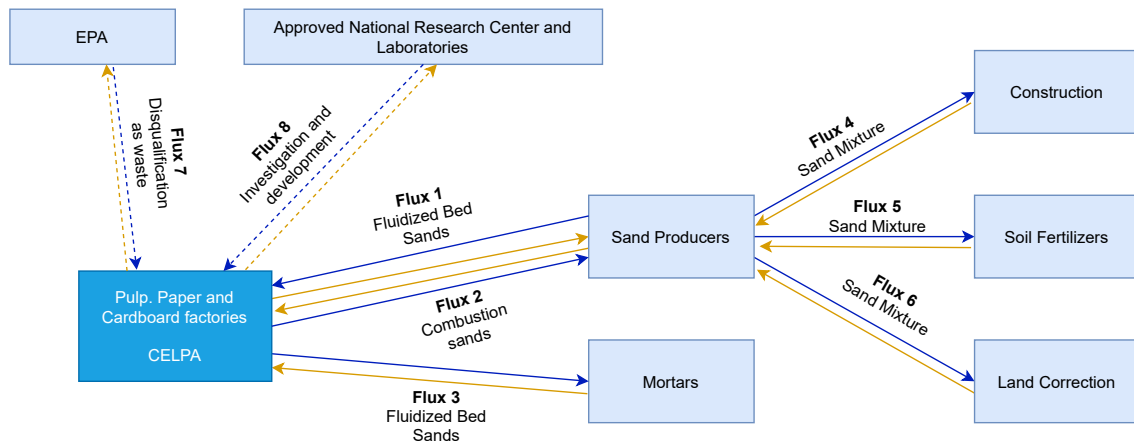


Figure 1.1: Industrial Symbiosis network scenario

Fig. 1.1 represents all the entities in the Industrial Symbiosis relationship and their relations. The Industrial Symbiosis network is composed of Environmental Portuguese Agency (EPA), National Research Centers, CELPA - Paper Industry Association, Sand

Producers, Mortars, Construction, Soil Fertilizers, and Land Correction companies. These companies exchange resources or fluxes with each other to gain advantages over their by-products and waste. These resources are cheaper from the buyer's perspective and earn more profit margin from the seller's perspective, as well as being more environmentally friendly. The dashed arrows in the figure represent the indirect relationships, and the solid arrows represent direct relationships, meaning they trade some resources between them. There is also a difference between blue and orange arrows, the first being the resource itself, and the latter is the monetary payment for the resource.

1.2 Objective

This thesis has the objective to propose and implement a blockchain architecture using blockchain and smart contracts to solve the challenges of the Industrial Symbiosis system depicted in Fig 1.1. To satisfy this objective the design has several requirements. The architecture needs to be scalable and modular so that new members can join the network without any issue and doing so as fast as possible. The network also needs a way for users to sell their extra resources to another entity on the system. Finally, the design has to have a way to monitor and audit the transactions made within the network. At the end of the work herein, the IS network described in Fig 1.1 is expected to be fully supported by a blockchain system, except for money transactions because, legally in Portugal, these can not be made using cryptocurrencies.

1.3 Contributions

The contribution of this dissertation is a novel system for blockchain architecture design to enhance the Industrial Symbiosis process of the Pulp, Paper, and Cardboard Production Sector Companies in Portugal, providing the required trust and transparency to a network to enhance their Industrial Symbiosis process for the scenario depicted in Fig. 1.1. This contribution resulted in the following publication:

- **Ricardo Gonçalves**, Inês Ferreira, Radu Godina, Pedro Pinto, and António Pinto. (2021). "A Smart Contract Architecture to Enhance the Industrial Symbiosis Pro-

cess Between the Pulp and Paper Companies - A Case Study". In *Blockchain and Applications*, Springer International Publishing, Cham, pages 252-260, 202. https://doi.org/10.1007/978-3-030-86162-9_25 [16]. This paper received the MDPI Systems Best Paper Award.

1.4 Organization

The organization of this document is as follows. In Chapter 2 the fundamentals of blockchain are described, as well as details and comparisons between some blockchain networks and frameworks. In Chapter 3 the requirements, design, and specification of the proposed solution are described. Chapter 4 describes the implementation of the proposed solution, as well as the explanation of some of its the source code. Chapter 5 describes some common use cases for the network and all the processes around it. Finally, in Chapter 6, conclusions are drawn, the results of the work are enumerated, and some future work is identified.

Chapter 2

Background and Related Work

This chapter describes the background and related work on the current context. Section 2.1 describes the blockchain and smart-contracts technologies. Section 2.2 describes the fundamentals of Ethereum blockchain and the Section 2.3 the fundamentals of Hyperledger Fabric blockchain, two common blockchains supporting smart contracts. Next, Section 5.4 compares the two blockchain technologies and identified the one best suited for the work herein. Finalizing this chapter, Section 2.5 presents the identified related work.

2.1 Blockchain & Smart contracts

This section will describe the blockchain technology and will present the main logic behind smart contracts, comparing them to regular contracts and showing how these work in practice.

The popularity of blockchain technology has been increasing over the past years. This exponential growth was mainly caused by bitcoin [25] popularity increase, as it also uses blockchain technology. This technology is based on a chain of blocks linked to each other, as represented in Fig. 2.1. Once a new block is created and inserted on the chain, the new block will have a link to the previous block, in a form of an hash value, creating a chain [25]. They all include the hash of the previous block except for the first one, which is called the genesis block.

There are two types of blockchain: permissionless and permissioned [23, 12]. The permissionless blockchain, also called public blockchain, is available to everyone. Anyone

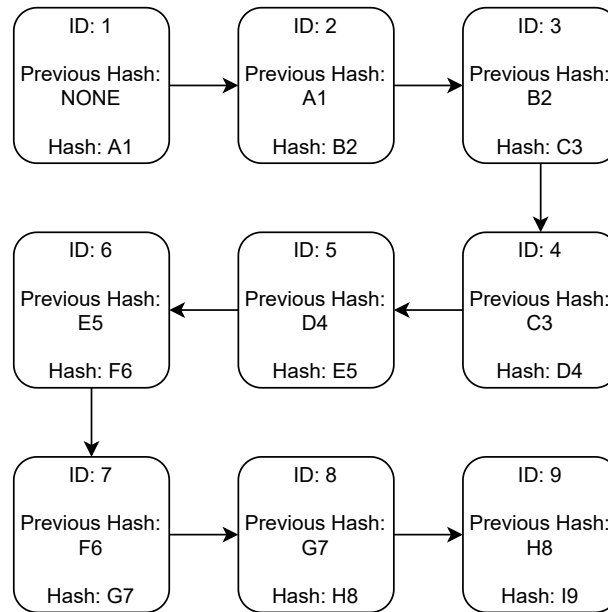


Figure 2.1: Example blockchain

can create and track down blocks. On the other side, the permissioned blockchain, or often referred to as private blockchain, has limited availability, only being accessible to a certain group and not being publicly available. On this type of blockchain, there is no anonymity like in a public blockchain.

Every full node in the network has a copy of the whole chain of blocks, this means that every transaction is available to each member, making the transactions traceable [28]. This makes every transaction more transparent and removes the need for intermediaries, allowing the process to be more efficient.

Information in the blockchain is immutable. To better exemplify this, let's take a look at Fig. 2.1. If a block in the middle of the chain is changed, Fig. 2.2, the chain is no longer valid. To be valid, the blocks in the front must be recalculated, Fig. 2.3 [20]. To apply the previous changes in every node in the network, a consensus algorithm should be used in every block that was modified. This work would be too demanding in terms of resource usage.

This is one of the biggest advantages of the blockchain, being a decentralized system allows for the lack of a central authority controlling the transactions. If someone wants to update the chain, they need to run the consensus algorithm of the network. In the case of the bitcoin, the most popular one, the adopted consensus algorithm is the Proof-of-Work

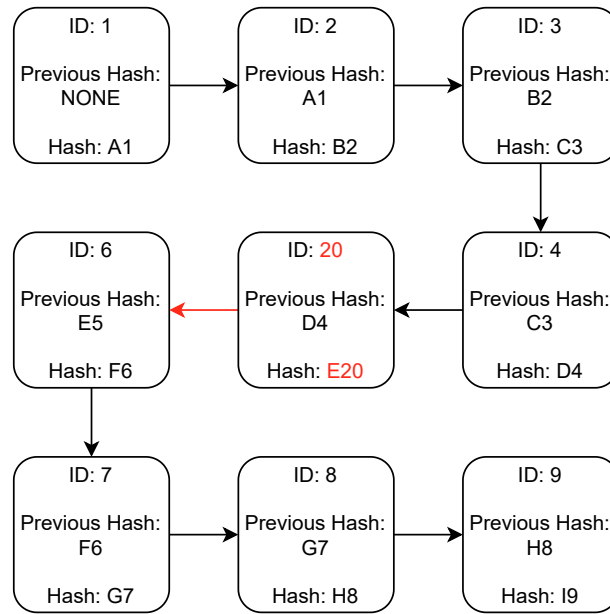


Figure 2.2: Broken blockchain

(PoW) [25]. One of the main disadvantages of this type of consensus algorithm is the amount of resources, including energy, required to perform it.

Table 2.1: Traditional vs Smart contracts

	Traditional contract	Smart contract
Third Parties	Government, etc	None
Execution Time	Days	Minutes
Process	Manual	Automatic
Transparency	None	Full
Security	Limited	Cryptographic Methods
Cost	High	Low

Smart contracts are pieces of code, or programs, that contain rules on how a contract must be executed. They automatically execute the terms described in them. When a smart contract is created it is sent to the blockchain and validated by the network members. Table 2.1 compares traditional contracts with smart contracts. This retrieves some advantages in using smart contracts vs traditional contracts, these being:

- No third parties - There is no need for someone else other than the blockchain and the parties on the contract in order to validate it.
- Automated - No need for human interaction.

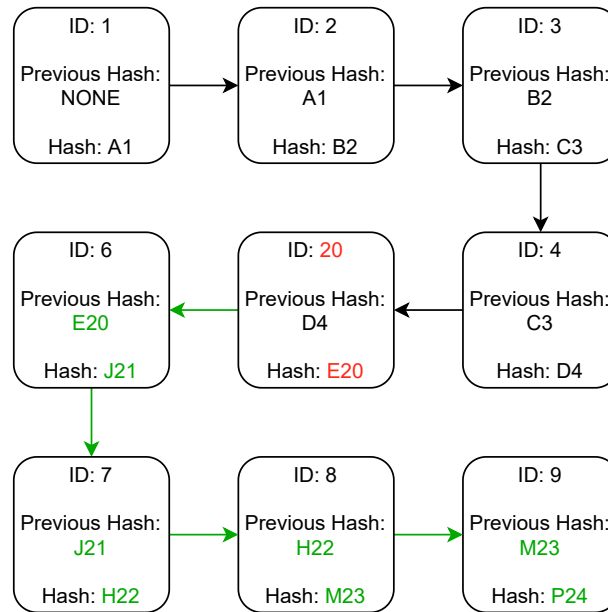


Figure 2.3: Valid blockchain

- Fast - As there is no need for human validation, the process is much faster.
- Secure - Uses cryptographic algorithms to implement security and can't be tampered with.
- Trust - Removes the need for trust between the entities of the contract.

2.2 Ethereum

Ethereum is a public and open-source, blockchain-based, decentralized software platform with the ability to run smart contracts released in 2015 [27]. Ethereum has its own cryptocurrency, Ether, and allows anyone to send cryptocurrency to anyone else, for a small fee. It also powers applications that everyone can use and no one can takedown, the smart contracts. Once a smart contract is deployed in the Ethereum network it cannot be modified or removed. Ether is the second-largest cryptocurrency currently on the market [18].

Unlike Bitcoin that only has one goal, being a peer-to-peer virtual currency, Ethereum has the capability of being programmable. This means that anyone can run programs inside the network, the smart contracts [6]. Using smart contracts, written in Solidity, Go, or any other programming language, developers can create their apps and run them

in a decentralized way. A popular analogy for smart contracts is vending machines, they are programmed to automatically deliver specific items based on specific inputs.

Running these computer programs on the network requires some computational power. To meet its computational requirements, every transaction or smart contract ran by the network pays a fee, named "gas". These fees are then paid to the users of the network who provide their computational power. These users have the name of miners. Requiring payment also provides some form of spam protection [6].

Miners work together in order to keep a consensus between all the nodes, Ethereum now uses the PoW mechanism, it is moving to a Proof-of-Stake (PoS) mechanism gradually in the next years. PoW and PoS are two examples of consensus algorithms that allow miners to agree on account balances, transactions and their order, preventing users from "double spending" their coins and ensuring that the Ethereum blockchain is tamper proof [15].

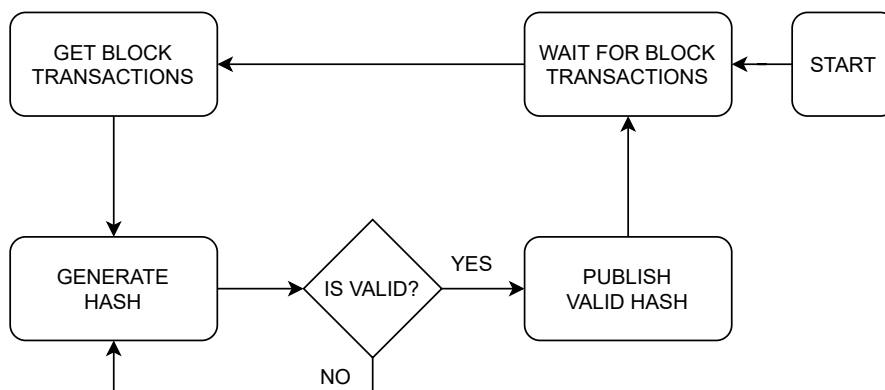


Figure 2.4: Trial and Error PoW

PoW requires users to go on an intensive competition of trial and error to find a valid hash for a block, Fig. 2.4. A valid hash is required in order for a block to be added to the chain. The first user to find a valid hash gets the reward. PoW makes an intensive use of useless computational power due to the the trial and error step of the process. Moreover, as the size of the PoW network increases, so does its energy requirements [15]. To address this issue, Ethereum is moving towards a PoS algorithm. PoS has the same end goal of the PoW but is much more environmentally friendly by significantly reducing the amount of energy required to run the network [4].

2.3 Hyperledger Fabric

The Hyperledger Project was first created in December of 2015 by the Linux Foundation. Hyperledger is a blockchain project that offers an open-source suite of frameworks and tools to allow enterprises to create their blockchain networks. Using the enterprise-ready permissioned blockchain tools, the business can create several modular blockchain solutions to enhance the efficiency of their systems [10].

Hyperledger Fabric is one of the tools provided by the Hyperledger suite. This is intended as the foundation layer for creating applications with a modular architecture. It has a plug-and-play design and every component can be changed independently [2]. Hyperledger Fabric was built from the ground up with the purpose of enterprise use.

2.3.1 Orderers

Other blockchains that are not permissionless anyone can participate in the consensus process. Fabric, being permissioned, features a node called the orderer that does the transaction ordering. Orderers also perform access control for channels, defining who can read and write data to them, and who can configure them [7].

The ordering service has three phases, the proposal, the ordering and packaging, and the validation and commit. In the first phase, the application sends the proposed ledger change to the endorsing peers, the peers then return an endorsed transaction to the application.

Then, in the second iteration, the returned endorsed transactions will be submitted to the ordering service that will create the block of transactions. The block will eventually be communicated to all peers on the channel for final validation and commit. The number of transactions depends on the configuration made by the administrator. If the block is considered as valid, it will be added to the ledger [2].

2.3.2 Peers

The peer nodes are the fundamental elements of the blockchain network. Nodes host the chaincode and the ledgers, and are the entry point for applications or administrators to send requests to the blockchain. As shown in Fig. 2.5, there are two types of peers:

commitment peers and endorsement peers. Commitment peers only store the ledger. Endorsement peers store the ledger and also run chaincode [2].

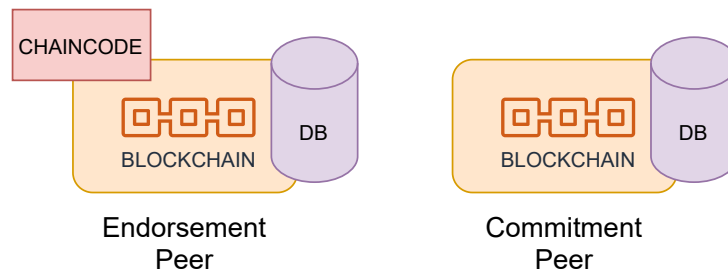


Figure 2.5: Endorsement and Commitment peer

Two types of requests can be made: query or update. The return of a query is immediate since all the required information is the peer’s local ledger. An update request is different because the peer can’t simply update its ledger, it firstly needs to obtain the approval from the other peers. In this case, peers send the desired transaction to the orderer component, and return a proposed update to the application.

2.3.3 Chaincode

Chaincode is a program that implements a specific interface. It can be written in different languages like Go, JavaScript or Java. This code runs in a docker container inside the endorsing peer. It initializes and manages the ledger state according to actions submitted by applications [7]. These are considered "smart contracts" because they handle business logic as agreed by its members.

2.3.4 Channels

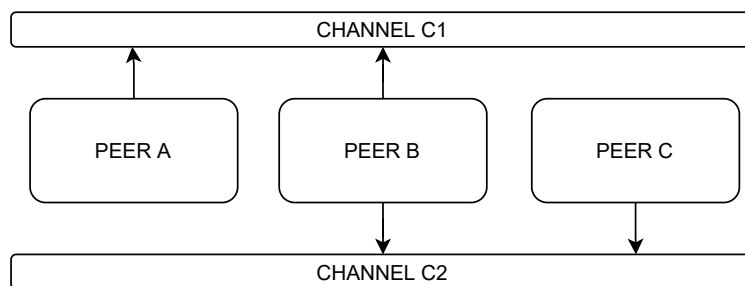


Figure 2.6: Channels Example

Channels are private subsets of interaction between two or more specific network mem-

bers. A channel enables confidential transaction exchange between such subsets of network members. There is no maximum number of channels on a Fabric network and every transaction must run within the context of a channel. A member can only view the transactions of the channels they take part in [3]. For example, Fig. 2.6 assumes a system with three entities (A, B, and C), and that two channels were set up (C1 and C2) as follows: channel C1 connecting entities A and B; channel C2 connecting entities B and C. The transactions made by entity A are not visible to entity C, and consequently, transactions from C are not visible to A. This separation of ledger data, by channel, enables the coexistence, for instance, of market competitors on the same blockchain network.

2.4 Ethereum vs Hyperledger

Table 2.2 compares both solutions [29]. Ethereum is permissionless, offers a built-in currency, and its transactions are transparent and open to everyone. While on the other hand, Hyperledger Fabric is permissioned and does not have a built-in cryptocurrency [2].

Table 2.2: Differences between Hyperledger Fabric and Ethereum

	Hyperledger Fabric	Ethereum
Purpose	Business to Business	Business to Contract and generalized applications
Consensus	Pluggable Consensus Algorithm	Proof of Work
Access	Permissioned	Permissionless
Transaction visibility	Confidential	Transparent
Native Currency	No	Yes, Ether
Programming Languages	C++, C#, Go, Haskell, Java, JavaScript, Python, Ruby, Rust, Elixir, Erlang	Solidity, Go

Comparing both, one can conclude that, for the envisioned scenario, Hyperledger Fabric is the adequate solution. The required use of a cryptocurrency by Ethereum, and the associated costs for smart contract execution are a clear drawback. These add complexity and introduce operational cost volatility due to the fluctuation of Ether's price. Another issue for the adoption of Ethereum relies on its permissionless nature; in the envisioned

scenario only selected entities should be able to create transactions or alter the state of the ledger.

2.5 Related Work on Application of smart contracts

Alexandris et al, in [1], proposed a blockchain-based system as the basis for a collaborative circular economy business model that consists of assets transitioning between operators. The proposed mechanism enables assets monitoring by the involved entities. It also allows auditing by third parties such as regulators of the state. They concluded that the adoption of a blockchain-based in a Industrial Symbiosis network brings benefits in regulated environmental jurisdictions, allowing entities to monitor the system easier.

Alexa Böckel et al, in [5], addresses the field of blockchain for a circular economy and review current developments. In the article, he conducts a research-practice gap analysis. The process uses a methodical review and qualitative examination of 57 distinct documents. They concluded three findings after the review. Two of them are that a clear terminology of blockchain types, their technical properties, and benefits is lacking in research, and is that trust and verification are the major possible advantages but a challenge to create.

Kouhizadeh et al, in [22], identified both blockchain and circular economy as the two new concepts that can positively impact the way we live in the future. In their work, they survey companies' cases and assess how blockchain will promote advances in the circular economy by linking blockchain to the multiple dimensions of the ReSOLVE model (Regenerate, Share, Optimize, Loop, Virtualize, and Exchange). They conclude with a list of research propositions, restraints, and future investigations. In a previous work [21], the same authors focused on the particular problem of product deletion in the same circular economy context while using blockchain as a supporting technology. Despite being a field of the circular economy, product deletion companies sometimes forget about it. The authors present a hardy research plan to assess the multiple links amongst technology, policy, commerce, and the natural environment.

The PlasticCoin cryptocurrency [19], proposed within the PlasticTwist H2020 research project, was developed to foster plastic reuse by citizens, promoting the circular economy

while maintaining trust among its users. They make use of the Hyperledger Fabric and developed a token following the concept described in the ERC-20 [11] standard of the Ethereum platform. These tokens are then printed and handed out, as scratchpads, to deserving people so that they can later claim their coins for reusing plastic materials.

From the surveyed related work, one can conclude that the circular economy in general, and network in particular, can benefit from the adoption of blockchain and related concepts, such as cryptocurrencies and smart contracts. Moreover, the use of smart contracts or cryptocurrencies will be dependent on each specific case; while smart contracts can be used for overall system automation, the cryptocurrencies can be used to attract citizens and foster their involvement. The solutions found do not fulfil the entirety of the requirements for this network, missing the fact of being able to trade resources, not just plastic, in the network.

Chapter 3

Proposed Solution

The proposed solution aims to support the existing Industrial Symbiosis network of the Pulp, Paper and Cardboard industry companies in Portugal, better described in [14, 13]. In this Industrial Symbiosis network, companies create the Fluidized Bed Sands (FBS) as a by-product and require normal sand for combustion. Fig. 3.1 depicts the network that is formed within this Industrial Symbiosis process, considering the different entities and their relationships.

Five actors are depicted with direct or indirect flows: Pulp and Paper companies, Sand Producers, Mortar Producers, the organizations that use the resulting material mixture, and EPA. Pulp and Paper companies use fluidized bed boilers for energy production. In fluidized bed boilers, a small amount of sand (regular sand) is used to maintain the energy production, this generates processed sand named FBS. The agency that represents every company in the pulp and paper industry is CELPA, the Portuguese Association of Pulp, Paper and Cardboard Producers.

The mortar and sand industries have the potential to use this FBS in their industrial process. Finally, industrial manufacturers of fertilizer, building materials, and other industries, can use the sand mixture to manufacture their products. The dashed lines indicate that the EPA monitors the trades between each entity.

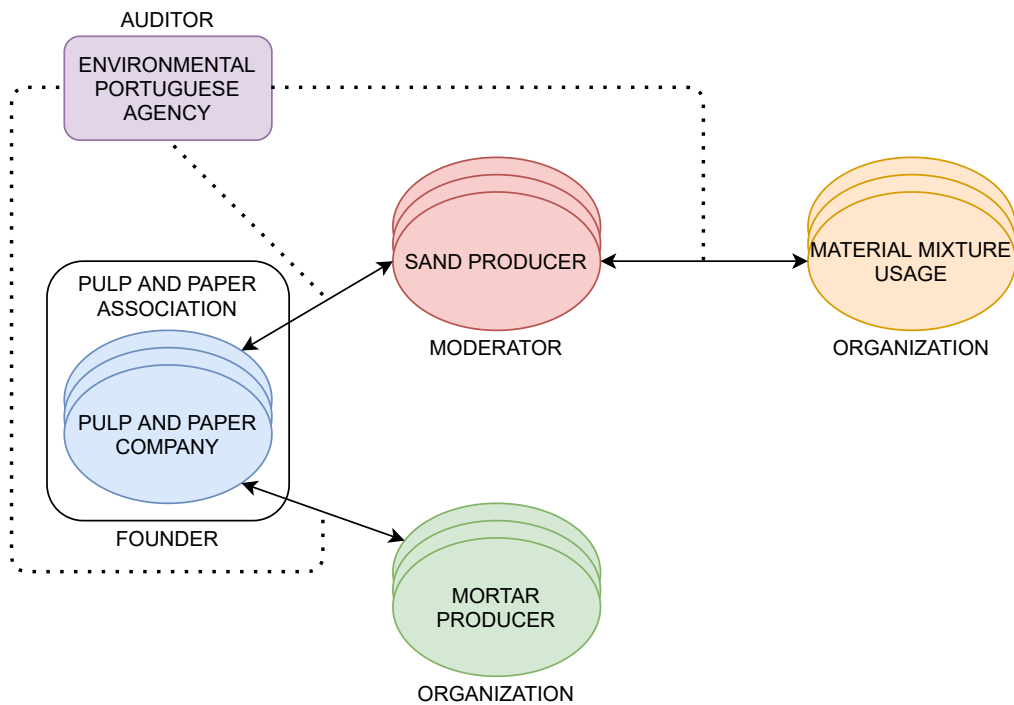


Figure 3.1: Industrial Symbiosis Network for the FBS case study

3.1 Requirements

The case study requires that the design follows some specific requirements. The architecture must be easily scalable. This requirement requires that new nodes can join without any downtime and not demanding a long setup time. The network must also be modular to allow for swapping or removal of any component if needed. Auditability is also a must-have in the system, this means that every transaction must be visible to EPA, the organization that audits all transactions. It also has to be private, in this case, permissioned, to keep all prevent unauthorized access to the data. Finally, as the last main requirement, the network must allow its participants to trade their by-products/waste materials between them. This requirement means allowing them to list and buy products from the network.

3.2 Design

The information is stored in five different structures, each one for a different purpose (see Fig. 3.2). These data types are organizations, transactions, orders, products, and

units. Organizations represent each entity in the network, it will have a unique identifier so it can be identified inside the network. The structure also has the name, description, address, and phone number of the organization, some basic information to better identify the entity for the user.

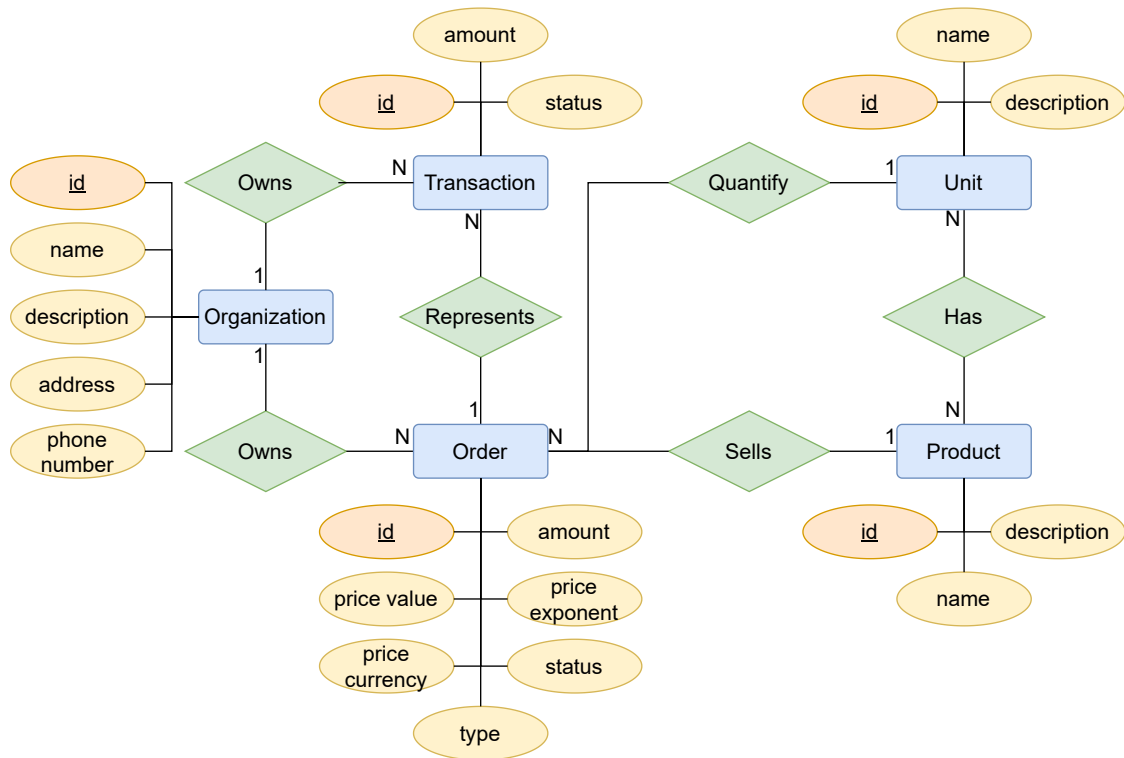


Figure 3.2: Data Relationship Diagram

To sell products, users need to create Orders. This data type has a unique identifier, the amount of product that wants to sell. It also has the price represented by the value in an integer format, currency, and exponent. Orders also have different statuses, for example, open or closed status. If another company buys products from these orders a transaction type is created. The structure has a unique identifier, the amount bought, and the status, it also has the order id to link back to the order. Finally, there is two data type that just represents the products that can be traded in the system, as well the units of those products. Both types have a unique identifier, name, and description.

There are four types of entities, the founder, the moderator, the auditor, and the normal organization. Each entity will have two or three of the five roles in this design, the founder, the moderator, the auditor, the organization admin, and the organization user. The envisioned use cases are shown in Fig. 3.3.

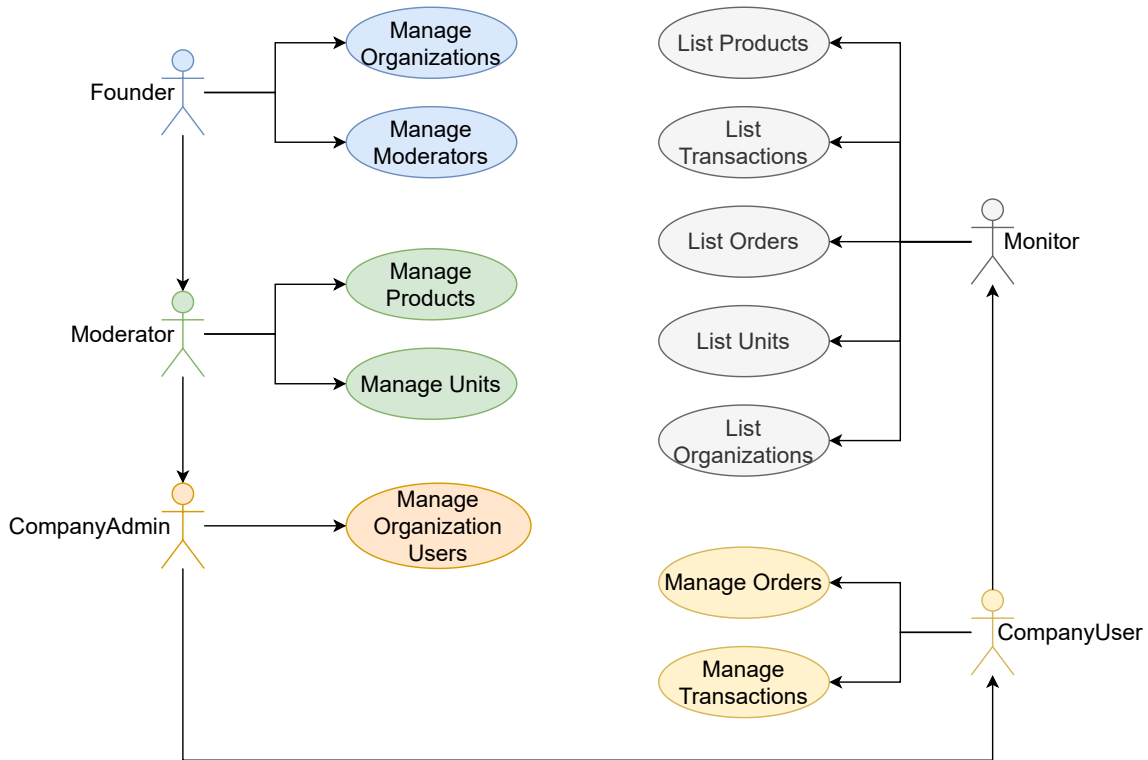


Figure 3.3: Use Cases

The auditor role will permit the users to read all the information on the network. The next two higher roles, the organization admin and user, can also manage the order and transactions of the user’s company. The admin also has the job of managing the company’s users. The moderator and the founder roles have all the previous permissions with the addition of extra ones. The moderator can create new products and units so they can be traded in the network. The founder, the highest role in the network, will have the task of creating and managing the company admins, this means when an organization wants to join the network, that is responsible to create the company admin for that entity.

The design is based on the Hyperledger Fabric architecture. Fig. 3.4 shows the architecture of the proposed solution. It includes one channel where all transactions will run, this way it is possible to accomplish the audit requirement. The auditor will have access to the channel, thus enabling access to all transactions in the network.

It will also introduce one peer and one Certificate Authority (CA) for each entity that is in the network, this helps to have a scalable and modular blockchain. Each entity just needs to a peer node and a CA to join the network. The design also includes an orderer

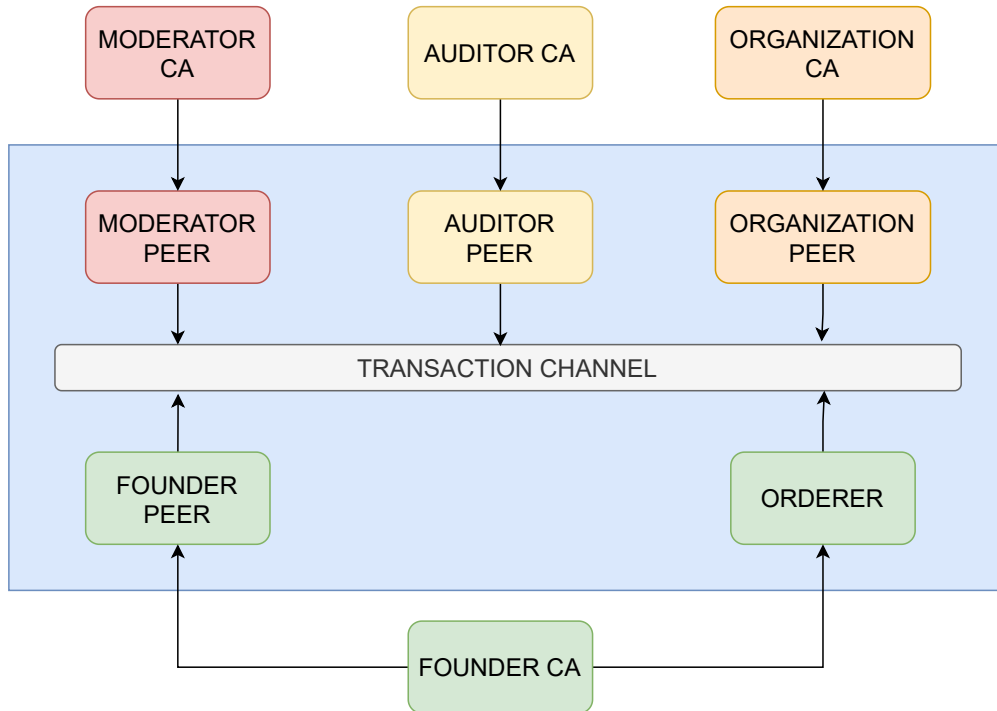


Figure 3.4: Architecture of the proposed solution

node that handles the packing and ordering of transactions into blocks.

The steps to be performed by two entities, A (the seller) and B (the buyer), are as follows. Firstly, both entities must be registered and receive their corresponding certificates, including their permissions. A can then create an order by identifying the product, its unit, quantity, and price. After this, B can browse the order's list and select the order issued by A . By selecting the order and specifying the amount they want to receive, will trigger the creation of a new transaction. At this point, B receives an invoice from A and has to make the payment, subsequently marking it as "paid" by changing the transaction state. A , after receiving the payment, changes the state of the transaction to "deposit received". When the cargo is ready to start the delivery, the transaction state is changed again to "in progress". Finally, when the products are delivered to B , the process is done, and the transaction is closed.

3.3 Architecture

The proposed architecture (see Fig. 3.4) assumes an implementation on a blockchain network built with Hyperledger Fabric and comprehends the following 4 peers: the founder

peer (which is the network founder and maintainer), the moderator peer (that has the role to moderate the network), the auditor peer (who audits all the transactions and relations within the network), and the organizations' peer (a peer per any other company in the network). Furthermore, the founder entity will also have an extra component, the orderer node.

The architecture also comprehends one CA for each entity, which will provide their users with credentials. The last component in the design is the transaction channel, it comprises only one channel to convey transactions between all entities because there was no need for companies to keep their transactions confidential, or hidden from the other entities. Using only one channel means that all transactions are available to all entities, but also facilitates the existence of a monitoring agency, one of the identified requirements. Another identified requirement is that every company, except the monitoring agency, needs to create trade offers for their by-products. These offers then need to be disseminated within the network so that other entities will be aware of them, thus being able to buy the related by-product and registering a new transaction in the blockchain.

To access the system, users need a certificate proving their identity. Organization Admins generate these certificates for users inside each organization. The Founder entity generates a new certificate for each Organization Admin when it joins the network. Scaling the platform can be done easily by adding a new organization CA and peer and connect it to the transaction channel.

In this setup, two key decisions were necessary: what database to use on the peers and what type of ordering service should be adopted by the Orderer. With respect to the selected database of the two available in Hyperledger Fabric, LevelDB and CouchDB, the CouchDB was the one adopted. It has support for richer queries than those supported by LevelDB, a key-based database. Regarding the ordering service, Raft was selected because of being the one recommended by Hyperledger Fabric in its documentation¹. The other two options available are Solo and Kafka. Solo only allows for the use of a single Orderer. Kafka is a distributed platform with multiples functionalities other than data storage, which would increase the overall system complexity, hampering its manageability.

For access control and authorization, the proposed solution will use the Hyperledger

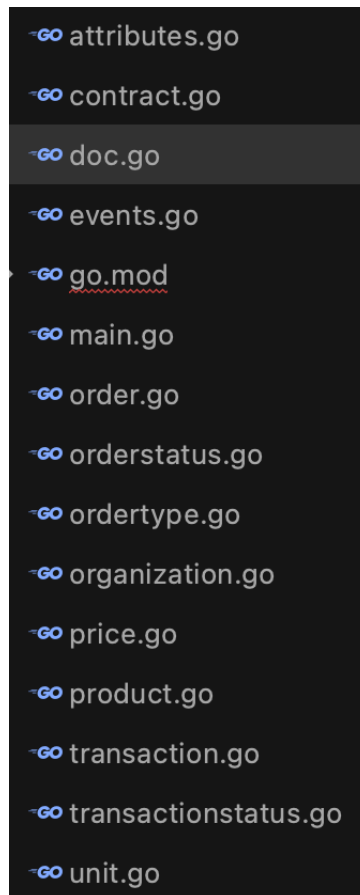
¹See https://hyperledger-fabric.readthedocs.io/en/latest/couchdb_tutorial.html

Fabric Attribute-Based Access Control. This method depends on attributes that are added to the certificates of the users when these certificates are created. These attributes are created by concatenating the data type and the type of access to be authorized, separated by a dot. Taking the example of product creation, only users with the attribute "products.create" will be allowed to create new products in the proposed solution.

Chapter 4

Implementation

The implementation was in Go, this was chosen because is my main language. Go appeared at Google in 2009, it was designed to mimic the core of C. The code is structure as Fig.4.1, it has individual files for each entity, the main.go file is the entry point for the chain code.



```
-go attributes.go
-go contract.go
-go doc.go
-go events.go
-go go.mod
-go main.go
-go order.go
-go orderstatus.go
-go ordertype.go
-go organization.go
-go price.go
-go product.go
-go transaction.go
-go transactionstatus.go
-go unit.go
```

Figure 4.1: File structure

4.1 General files

In the `main.go` file, the `SmartContract` structure is initialized with the Hyperledger fabric package contract API, `fabric-contract-api-go/contractapi`, Listing 4.1. It has a field named `checkPermissions` has `true`. This field is used to make the chain code check for the ABAC permissions for debug purposes. If the creation of the chain code or has an error during the runtime it just returns it to this main function, logs it, and stops the chain code.

Listing 4.1: `main.go` file contents

```
func main() {
    assetChaincode, err := contractapi.NewChaincode(&SmartContract{
        checkPermissions: true,
    })
    if err != nil {
        log.Panicf("Error_creating_asset-transfer-basic_chaincode:_%v"
    }

    if err := assetChaincode.Start(); err != nil {
        log.Panicf("Error_starting_asset-transfer-basic_chaincode:_%v"
    }
}
```

The attributes file, `attributes.go` in his content, has the attributes keys in constants variables with the values that the CA issues in the certificates, Listing 4.2. The file also has a function that takes the `TransactionContext` and the attribute it wants to check for, Listing 4.3. If the attribute is present or `checkPermissions` is false it returns no error, nil, otherwise, it will return an error `"not authorized"`.

Listing 4.2: attributes roles definition

```
const (
    UnitsCreate Attribute = "units.create"
    UnitsDelete Attribute = "units.delete"
```

```
UnitsRead    Attribute = "units.read"
UnitsUpdate  Attribute = "units.update"

ProductsCreate Attribute = "products.create"
ProductsDelete Attribute = "products.delete"
ProductsRead   Attribute = "products.read"
ProductsUpdate Attribute = "products.update"

OrganizationsCreate Attribute = "organizations.create"
OrganizationsDelete Attribute = "organizations.delete"
OrganizationsRead   Attribute = "organizations.read"
OrganizationsUpdate Attribute = "organizations.update"

OrdersCreate Attribute = "orders.create"
OrdersDelete Attribute = "orders.delete"
OrdersRead   Attribute = "orders.read"
OrdersUpdate Attribute = "orders.update"

TransactionsCreate Attribute = "transactions.create"
TransactionsDelete Attribute = "transactions.delete"
TransactionsRead   Attribute = "transactions.read"
TransactionsUpdate Attribute = "transactions.update"
)
```

Listing 4.3: attributes.go file contents

```
type Attribute string

func (a Attribute) String() string {
    return string(a)
}
```

```
func (s *SmartContract) HasPermission(ctx contractapi.TransactionContextInterface) bool {  
    if !s.checkPermissions {  
        return nil  
    }  
  
    err := ctx.GetClientIdentity().AssertAttributeValue(att.String(), "true")  
    if err != nil {  
        return fmt.Errorf("not_authorized")  
    }  
  
    return nil  
}
```

The doc.go file has two helper types to help with storing in the CouchDB database, Listing 4.4. It has *DocType* type that represents the document type of document in the database to be easier to search. The *Doc* structure has the fields that every structure needs to be identified in the database, it will compose all the other structures saved in the database.

Listing 4.4: doc.go file contents

```
type DocType string  
  
type Doc struct {  
    Type DocType 'json:"doc_type" '  
    CreatedBy string 'json:"created_by" '  
    UpdatedBy string 'json:"updated_by" '  
}
```

The file in Listing 4.5, *contract.go*, has the main structure composed by the *contractapi.Contract* from the hyperledger fabric package. It also has the implementation two methods for the *SmartContract* type. The first one, *GetSubmittingClientIdentity*, receives as parameter the *TransactionContext* and returns the unique identifier of the user requesting the transaction or an error. The second one, *GetSubmittingClientOrganization*,

receives as parameter the *TransactionContext* and returns the organization of the user or an error.

Listing 4.5: contract.go file contents

```
type SmartContract struct {  
    contractapi.Contract  
    checkPermissions bool  
}  
  
func (s *SmartContract) GetSubmittingClientIdentity(ctx contractapi.TransactionContext) (string, error) {  
    b64ID, err := ctx.GetClientIdentity().GetID()  
    if err != nil {  
        return "", fmt.Errorf("failed to read clientID: %v", err)  
    }  
    decodeID, err := base64.StdEncoding.DecodeString(b64ID)  
    if err != nil {  
        return "", fmt.Errorf("failed to base64 decode clientID: %v", err)  
    }  
    return string(decodeID), nil  
}  
  
func (s *SmartContract) GetSubmittingClientOrganization(ctx contractapi.TransactionContext) (string, error) {  
    return ctx.GetClientIdentity().GetMSPID()  
}
```

The price data structure, Listing 4.6 is composed of three fields, amount, this represents the amount in an integer format number without decimal places, currency, the currency type of the amount, and the exponent, the exponent of the currency, this shows how many decimal cases the amount has.

Listing 4.6: Price structure

```
type Price struct {  
    Amount      uint32 'json:"amount"'  
    Currency    string  
    Exponent    int32
```



```
    Exponent    uint32 'json:"exponent" '  
    Currency    string 'json:"currency" '  
}
```

4.2 Unit

4.2.1 Structure

There is two structures representing units in Listing 4.7, *Unit* and *UnitInner*, these two have different purposes, *UnitInner* represents the data stored in the database with the additional composing *Doc* type. There is a function to parse between the *UnitInner* and *Unit* easier.

Listing 4.7: Unit Structure

```
const (  
    UnitDoc DocType = "unit"  
)  
  
type UnitInner struct {  
    Doc  
  
    ID          string 'json:"id" '  
    Name        string 'json:"name" '  
    Description string 'json:"description" '  
    Exponent    uint32 'json:"exponent" '  
}  
  
type Unit struct {  
    ID          string 'json:"id" '  
    Name        string 'json:"name" '  
    Description string 'json:"description" '  
    Exponent    uint32 'json:"exponent" '  
}
```

```
}

func FromUnitInner(u *UnitInner) *Unit {
    if u == nil {
        return nil
    }

    return &Unit{
        ID:          u.ID,
        Name:        u.Name,
        Description: u.Description,
        Exponent:    u.Exponent,
    }
}
```

4.2.2 Get Unit ID

The *GetUnitID*, Listing 4.8, build a storage unit id of the input from the user by adding the unit document type.

Listing 4.8: Get Unit ID

```
func (s *SmartContract) GetUnitID(_ contractapi.TransactionContextInterface, i
    return string(UnitDoc) + "_" + id
}
```

4.2.3 Create Unit

To create a new unit the user need to invoke the *CreateUnit* method with the id, name, description, and exponent. These parameters are then pass through to the method in Listing 4.9. It first if the current context user has the permissions for creating a unit. Then checks if a unit with the same id already exists, if so returns an error saying it already exists, otherwise continues to the next step. After that, the client id is retrieved

from the context, and the new unit is built. Finally, the unit is committed to the state, creating a new unit ready to be retrieved.

Listing 4.9: Create Unit

```
func (s *SmartContract) CreateUnit(ctx contractapi.TransactionContextInterface
    if err := s.HasPermission(ctx, UnitsCreate); err != nil {
        return err
    }

    exists, err := s.UnitExist(ctx, id)
    if err != nil {
        return err
    }
    if exists {
        return fmt.Errorf("the asset %s already exists", id)
    }

    clientID, err := s.GetSubmittingClientIdentity(ctx)
    if err != nil {
        return err
    }

    doc := Doc{
        Type:      UnitDoc,
        CreatedBy: clientID,
        UpdatedBy: clientID,
    }

    unit := UnitInner{
        ID:      s.GetUnitID(ctx, id),
        Name:    name,
```

```
        Description: description ,
        Exponent:    exponent ,
        Doc:         doc ,
    }

    assetBytes , err := json.Marshal(unit)
    if err != nil {
        return err
    }

    err = ctx.GetStub().PutState(unit.ID, assetBytes)
    if err != nil {
        return err
    }

    return nil
}
```

4.2.4 Unit Exists

Both methods *UnitExist* and *UnitsExist*, Listing 4.10, checks if units exist, the first one takes a single id, the second one receives a list of ids and checks if they exist in the current state.

Listing 4.10: Unit Exist

```
func (s *SmartContract) UnitExist(ctx contractapi.TransactionContextInterface ,
    assetJSON , err := ctx.GetStub().GetState(s.GetUnitID(ctx , id))
    if err != nil {
        return false , fmt.Errorf("failed to read from world state: %v"
    }

    return assetJSON != nil , nil
```

```
}
```

```
func (s *SmartContract) UnitsExist(ctx contractapi.TransactionContextInterface
    for _, id := range ids {
        e, err := s.UnitExist(ctx, id)
        if err != nil {
            return err
        }
        if !e {
            return fmt.Errorf("unit %s does not exists", id)
        }
    }

    return nil
}
```

4.2.5 Get Unit

The *GetUnit*, Listing 4.11 receives a unit id from the parameters and returns a unit or an error. It checks to see if the client requesting the operation has the correct attribute role and gets the unit from the state. Before sending the unit back to the client it removes the storage id that adds the document type. The Listing 4.11 represent the same method but instead of returning the type *Unit*, returns *UnitInner*.

Listing 4.11: Get Unit and Get Unit Inner

```
func (s *SmartContract) GetUnitInner(ctx contractapi.TransactionContextInterface
    if err := s.HasPermission(ctx, UnitsRead); err != nil {
        return nil, err
    }

    assetBytes, err := ctx.GetStub().GetState(s.GetUnitID(ctx, id))
    if err != nil {
```

```
        return nil, fmt.Errorf("failed to get asset %s: %v", id, err)
    }

    if assetBytes == nil {
        return nil, fmt.Errorf("asset %s does not exist", id)
    }

    var unit UnitInner
    err = json.Unmarshal(assetBytes, &unit)
    if err != nil {
        return nil, err
    }

    unit.ID = strings.TrimPrefix(unit.ID, string(UnitDoc)+"_")
    return &unit, nil
}

func (s *SmartContract) GetUnit(ctx contractapi.TransactionContextInterface, id string) (UnitInner, error) {
    if err := s.HasPermission(ctx, UnitsRead); err != nil {
        return nil, err
    }

    assetBytes, err := ctx.GetStub().GetState(s.GetUnitID(ctx, id))
    if err != nil {
        return nil, fmt.Errorf("failed to get asset %s: %v", id, err)
    }

    if assetBytes == nil {
        return nil, fmt.Errorf("asset %s does not exist", id)
    }
}
```

```
var unit UnitInner
err = json.Unmarshal(assetBytes, &unit)
if err != nil {
    return nil, err
}

unit.ID = strings.TrimPrefix(unit.ID, string(UnitDoc)+"_")
return FromUnitInner(&unit), nil
}
```

4.2.6 List Units

The method in Listing 4.12, *ListUnits*, lists all the units currently saved in the state. It first checks for permissions, then returns the units to the client.

Listing 4.12: List Units

```
func (s *SmartContract) ListUnits(ctx contractapi.TransactionContextInterface)
    if err := s.HasPermission(ctx, UnitsRead); err != nil {
        return nil, err
    }

    results, err := ctx.GetStub().GetQueryResult(fmt.Sprintf("{ selector":
    if err != nil {
        return nil, fmt.Errorf("failed to get assets: %v", err)
    }
    defer results.Close()

    var assets []*Unit
    for results.HasNext() {
        queryResult, err := results.Next()
        if err != nil {
```

```
        return nil, err
    }
    var unit UnitInner
    err = json.Unmarshal(queryResult.Value, &unit)
    if err != nil {
        return nil, err
    }

    unit.ID = strings.TrimPrefix(unit.ID, string(UnitDoc)+"_")
    assets = append(assets, FromUnitInner(&unit))
}

return assets, nil
}
```

4.2.7 Delete Unit

To delete a unit the method in Listing 4.13 is used. It receives from the parameter the id of the unit to be deleted and returns an error if not possible to delete. To delete the code checks if the client has the correct role to be able to do the action.

Listing 4.13: Delete Unit

```
func (s *SmartContract) DeleteUnit(ctx contractapi.TransactionContextInterface
    if err := s.HasPermission(ctx, UnitsDelete); err != nil {
        return err
    }

    err := ctx.GetStub().DelState(s.GetUnitID(ctx, id))
    if err != nil {
        return fmt.Errorf("failed to delete asset %s: %v", id, err)
    }
}
```



```
    return nil
}
```

4.3 Product

4.3.1 Structure

There is two structures representing products, Listing 4.14, *Product* and *ProductInner*, these two have different purposes, *ProductInner* represents the data stored in the database with the additional composing *Doc* type. There is a function to parse between the *ProductInner* and *Product* easier.

Listing 4.14: Product Structure

```
const (
    ProductDoc DocType = "product"
)

type ProductInner struct {
    Doc

    ID          string  'json:" id" '
    Name        string  'json:" name" '
    Description string  'json:" description" '
    UnitIDs     []string 'json:" unit_ids" '
}

type Product struct {
    ID          string  'json:" id" '
    Name        string  'json:" name" '
    Description string  'json:" description" '
    Units       []*Unit 'json:" units" '
}
```

```
func (s *SmartContract) FromProductInner(ctx contractapi.TransactionContextInterface, p *Product, units []Unit) (*Product, error) {
    units := make([]Unit, 0, len(p.UnitIDs))
    for _, unitID := range p.UnitIDs {
        unit, err := s.GetUnit(ctx, unitID)
        if err != nil {
            continue
        }

        units = append(units, unit)
    }

    return &Product{
        ID:          p.ID,
        Name:        p.Name,
        Description: p.Description,
        Units:       units,
    }
}
```

4.3.2 Get Product ID

The *GetProductID* build a storage product id of the input from the user by adding the product document type, Listing 4.15.

Listing 4.15: Get Product ID

```
func (s *SmartContract) GetProductID(_ contractapi.TransactionContextInterface, id string) (*Product, error) {
    return string(ProductDoc) + "_" + id
}
```

4.3.3 Product Exist

The method *ProductExist*, Listing 4.16, checks if a product exists, it takes a single id and checks if it exists in the current state.

Listing 4.16: Product Exist

```
func (s *SmartContract) ProductExist(ctx contractapi.TransactionContextInterface,
    assetJSON, err := ctx.GetStub().GetState(s.GetProductID(ctx, id))
    if err != nil {
        return false, fmt.Errorf("failed to read from world state: %v")
    }

    return assetJSON != nil, nil
}
```

4.3.4 Create Product

To create a new product the user need to invoke the *CreateProduct* method with the id, name, description, and units. These parameters are then pass through to the method in Listing 4.17. It first if the current context user has the permissions for creating a product. Then checks if a product with the same id already exists, if so returns an error saying it already exists, otherwise continues to the next step. It also checks if the units ids coming from the request are valid and exist. After that, the client id is retrieved from the context, and the new product is built. Finally, the product is committed to the state, creating a new product ready to be retrieved.

Listing 4.17: Create Product

```
func (s *SmartContract) CreateProduct(ctx contractapi.TransactionContextInterface,
    if err := s.HasPermission(ctx, ProductsCreate); err != nil {
        return err
    }

    exists, err := s.ProductExist(ctx, id)
```

```
if err != nil {
    return err
}
if exists {
    return fmt.Errorf("the asset %s already exists", id)
}

if err := s.UnitsExists(ctx, units); err != nil {
    return err
}

clientID, err := s.GetSubmittingClientIdentity(ctx)
if err != nil {
    return err
}

doc := Doc{
    Type:      ProductDoc,
    CreatedBy: clientID,
    UpdatedBy: clientID,
}

unit := ProductInner{
    ID:          s.GetProductID(ctx, id),
    Name:        name,
    Description: description,
    UnitIDs:     units,
    Doc:         doc,
}
```

```
assetBytes, err := json.Marshal(unit)
if err != nil {
    return err
}

err = ctx.GetStub().PutState(unit.ID, assetBytes)
if err != nil {
    return err
}

return nil
}
```

4.3.5 Get Product

The *GetProduct*, Listing 4.18 receives a product id from the parameters and returns a product or an error. It checks to see if the client requesting the operation has the correct attribute role and gets the product from the state. Before sending the product back to the client it removes the storage id that adds the document type. The Listing 4.18 represent the same method but instead of returning the type *Product*, returns *ProductInner*.

Listing 4.18: Create Product

```
func (s *SmartContract) GetProductInner(ctx contractapi.TransactionContextInterface, id string) (*ProductInner, error) {
    if err := s.HasPermission(ctx, ProductsRead); err != nil {
        return nil, err
    }

    assetBytes, err := ctx.GetStub().GetState(s.GetProductID(ctx, id))
    if err != nil {
        return nil, fmt.Errorf("failed to get asset %s: %v", id, err)
    }
}
```

```
    if assetBytes == nil {
        return nil, fmt.Errorf("asset %s does not exist", id)
    }

    var product ProductInner
    err = json.Unmarshal(assetBytes, &product)
    if err != nil {
        return nil, err
    }

    product.ID = strings.TrimPrefix(product.ID, string(ProductDoc)+"_")
    return &product, nil
}

func (s *SmartContract) GetProduct(ctx contractapi.TransactionContextInterface
    if err := s.HasPermission(ctx, ProductsRead); err != nil {
        return nil, err
    }

    assetBytes, err := ctx.GetStub().GetState(s.GetProductID(ctx, id))
    if err != nil {
        return nil, fmt.Errorf("failed to get asset %s: %v", id, err)
    }

    if assetBytes == nil {
        return nil, fmt.Errorf("asset %s does not exist", id)
    }

    var product ProductInner
    err = json.Unmarshal(assetBytes, &product)
```

```
    if err != nil {
        return nil, err
    }

    product.ID = strings.TrimPrefix(product.ID, string(ProductDoc)+"_")
    return s.FromProductInner(ctx, &product), nil
}
```

4.3.6 List Products

The method in Listing 4.19, *ListProducts*, lists all the products currently saved in the state. It first checks for permissions, if the correct role match, then returns the products to the client.

Listing 4.19: List Products

```
func (s *SmartContract) ListProducts(ctx contractapi.TransactionContextInterface) {
    if err := s.HasPermission(ctx, ProductsRead); err != nil {
        return nil, err
    }

    results, err := ctx.GetStub().GetQueryResult(fmt.Sprintf("{ selector }"))
    if err != nil {
        return nil, fmt.Errorf("failed to get assets: %v", err)
    }
    defer results.Close()

    var assets []*Product
    for results.HasNext() {
        queryResult, err := results.Next()
        if err != nil {
            return nil, err
        }
    }
}
```

```
var unit ProductInner
err = json.Unmarshal(queryResult.Value, &unit)
if err != nil {
    return nil, err
}

unit.ID = strings.TrimPrefix(unit.ID, string(ProductDoc)+"_")
assets = append(assets, s.FromProductInner(ctx, &unit))
}

return assets, nil
}
```

4.3.7 Delete Product

To delete a product the method in Listing 4.20 is used. It receives from the parameter the id of the product to be deleted and returns an error if not possible to delete. To delete the code checks if the client has the correct role to be able to do the action.

Listing 4.20: Delete Product

```
func (s *SmartContract) ListProducts(ctx contractapi.TransactionContextInterface)
    if err := s.HasPermission(ctx, ProductsRead); err != nil {
        return nil, err
    }

    results, err := ctx.GetStub().GetQueryResult(fmt.Sprintf("{selector}"))
    if err != nil {
        return nil, fmt.Errorf("failed to get assets: %v", err)
    }
    defer results.Close()

    var assets []*Product
```



```
for results.HasNext() {
    queryResult, err := results.Next()
    if err != nil {
        return nil, err
    }
    var unit ProductInner
    err = json.Unmarshal(queryResult.Value, &unit)
    if err != nil {
        return nil, err
    }

    unit.ID = strings.TrimPrefix(unit.ID, string(ProductDoc)+"_")
    assets = append(assets, s.FromProductInner(ctx, &unit))
}

return assets, nil
}
```

4.4 Organization

4.4.1 Structure

There is two structures representing organizations, *Organization* and *OrganizationInner*, Listing 4.21, these two have different purposes, *OrganizationInner* represents the data stored in the database with the additional composing *Doc* type. There is a function to parse between the *OrganizationInner* and *Organization* easier.

Listing 4.21: Organization Structure

```
const (
    OrganizationDoc DocType = "organization"
)
```

```
type OrganizationInner struct {  
    Doc  
  
    ID          string 'json:"id"'  
    Name        string 'json:"name"'  
    Description string 'json:"description"'  
    Address     string 'json:"address"'  
    PhoneNumber string 'json:"phone_number"'  
}
```

```
type Organization struct {  
    ID          string 'json:"id"'  
    Name        string 'json:"name"'  
    Description string 'json:"description"'  
    Address     string 'json:"address"'  
    PhoneNumber string 'json:"phone_number"'  
}
```

```
func (s *SmartContract) FromOrganizationInner(_ contractapi.TransactionContext)  
    return &Organization{  
        ID:          p.ID,  
        Name:        p.Name,  
        Description: p.Description,  
        Address:     p.Address,  
        PhoneNumber: p.PhoneNumber,  
    }  
}
```

4.4.2 Get Organization ID

The *GetOrganizationID*, Listing 4.22, build a storage organization id of the input from the user by adding the organization document type.

Listing 4.22: Get Organization ID

```
func (s *SmartContract) GetOrganizationID(_ contractapi.TransactionContextInterface) return string(OrganizationDoc) + "_" + id
}
```

4.4.3 Organization Exist

The method *OrganizationExist*, Listing 4.23 checks if an organization exists, it takes a single id and checks if it exists in the current state.

Listing 4.23: Organization Exist

```
func (s *SmartContract) GetOrganizationID(_ contractapi.TransactionContextInterface) return string(OrganizationDoc) + "_" + id
}
```

4.4.4 Create Organization

To create a new organization, the user needs to invoke the *CreateOrganization* method with the id, name, description, address, and phone number in string. These parameters are then pass through to the method in Listing 4.24. It first if the current context user has the permissions for creating an organization. Then checks if an organization with the same id already exists, if so returns an error saying it already exists, otherwise continues to the next step. After that, the client id is retrieved from the context, and the new organization is built. Finally, the organization is committed to the state, creating a new organization ready to be retrieved.

Listing 4.24: Create Organization

```
func (s *SmartContract) CreateOrganization(ctx contractapi.TransactionContextInterface) {
    if err := s.HasPermission(ctx, OrganizationsCreate); err != nil {
```

```
        return err
    }

    exists, err := s.OrganizationExist(ctx, id)
    if err != nil {
        return err
    }
    if exists {
        return fmt.Errorf("the asset %s already exists", id)
    }

    clientID, err := s.GetSubmittingClientIdentity(ctx)
    if err != nil {
        return err
    }

    doc := Doc{
        Type:      OrganizationDoc,
        CreatedBy: clientID,
        UpdatedBy: clientID,
    }

    unit := OrganizationInner{
        ID:          s.GetOrganizationID(ctx, id),
        Name:        name,
        Description: description,
        Address:     address,
        PhoneNumber: phoneNumber,
        Doc:        doc,
    }
```

```
    assetBytes, err := json.Marshal(unit)
    if err != nil {
        return err
    }

    err = ctx.GetStub().PutState(unit.ID, assetBytes)
    if err != nil {
        return err
    }

    return nil
}
```

4.4.5 Update Organization

To update an organization, the user needs to invoke the *UpdateOrganization* method with the id, name, description, address, and phone number in string. These parameters are then pass through to the method in Listing 4.25. It first if the current context user has the permissions for updating its organization or if it's an admin with the create organization role. Then checks if an organization with the same id already exists, if it does not exist returns an error saying it does not exist, otherwise continues to the next step. After that, the client id is retrieved from the context, and the new organization is built. Finally, the updated organization is committed to the state.

Listing 4.25: Update Organization

```
func (s *SmartContract) UpdateOrganization(
    ctx contractapi.TransactionContextInterface,
    id string, name string, description string, address string, phoneNumbers []string) error {
    if err := s.HasPermission(ctx, OrganizationsCreate); err != nil {
        if innerErr := s.HasPermission(ctx, OrganizationsUpdate); innerErr != nil {
```

```
        return innerErr
    }

    if orgID, innerErr := s.GetSubmittingClientOrganization(ctx);
        return err
    }
}

exists, err := s.OrganizationExist(ctx, id)
if err != nil {
    return err
}
if !exists {
    return fmt.Errorf("the asset %s does not exists", id)
}

clientID, err := s.GetSubmittingClientIdentity(ctx)
if err != nil {
    return err
}

org, err := s.GetOrganizationInner(ctx, id)
if err != nil {
    return err
}

org.Name = name
org.Description = description
org.Address = address
org.PhoneNumber = phoneNumber
```

```
org.UpdatedBy = clientID

assetBytes, err := json.Marshal(org)
if err != nil {
    return err
}

err = ctx.GetStub().PutState(org.ID, assetBytes)
if err != nil {
    return err
}

return nil
}
```

4.4.6 Get Organization

The *GetOrganization*, Listing 4.26 receives a product id from the parameters and returns an organization or an error. It checks to see if the client requesting the operation has the correct attribute role and gets the organization from the state. Before sending the organization back to the client, it removes the storage id that adds the document type. The Listing 4.26 represent the same method but instead of returning the type *Organization*, returns *OrganizationInner*.

Listing 4.26: Get Organization and Get Organization Inner

```
func (s *SmartContract) GetOrganizationInner(ctx contractapi.TransactionContext) (OrganizationInner, error) {
    if err := s.HasPermission(ctx, OrganizationsRead); err != nil {
        return nil, err
    }

    assetBytes, err := ctx.GetStub().GetState(s.GetOrganizationID(ctx, id))
    if err != nil {
```

```
        return nil, fmt.Errorf("failed to get asset %s: %v", id, err)
    }

    if assetBytes == nil {
        return nil, fmt.Errorf("asset %s does not exist", id)
    }

    var product OrganizationInner
    err = json.Unmarshal(assetBytes, &product)
    if err != nil {
        return nil, err
    }

    product.ID = strings.TrimPrefix(product.ID, string(OrganizationDoc)+".")
    return &product, nil
}

func (s *SmartContract) GetOrganization(ctx contractapi.TransactionContextInterface) (*OrganizationInner, error) {
    if err := s.HasPermission(ctx, OrganizationsRead); err != nil {
        return nil, err
    }

    assetBytes, err := ctx.GetStub().GetState(s.GetOrganizationID(ctx, id))
    if err != nil {
        return nil, fmt.Errorf("failed to get asset %s: %v", id, err)
    }

    if assetBytes == nil {
        return nil, fmt.Errorf("asset %s does not exist", id)
    }
}
```



```
var product OrganizationInner
err = json.Unmarshal(assetBytes, &product)
if err != nil {
    return nil, err
}

product.ID = strings.TrimPrefix(product.ID, string(OrganizationDoc)+"_")
return s.FromOrganizationInner(ctx, &product), nil
}
```

4.4.7 List Organizations

The method in Listing 4.27, *ListOrganizations*, lists all the organizations currently saved in the state. It first checks for permissions, if the correct role match, then returns the organizations to the client.

Listing 4.27: List Organizations

```
func (s *SmartContract) ListOrganizations(ctx contractapi.TransactionContextInterface) (
    if err := s.HasPermission(ctx, OrganizationsRead); err != nil {
        return nil, err
    }

    results, err := ctx.GetStub().GetQueryResult(fmt.Sprintf("{ selector }"))
    if err != nil {
        return nil, fmt.Errorf("failed to get assets: %v", err)
    }
    defer results.Close()

    var assets []*Organization
    for results.HasNext() {
        queryResult, err := results.Next()
```

```
        if err != nil {
            return nil, err
        }
        var unit OrganizationInner
        err = json.Unmarshal(queryResult.Value, &unit)
        if err != nil {
            return nil, err
        }

        unit.ID = strings.TrimPrefix(unit.ID, string(OrganizationDoc)+
        assets = append(assets, s.FromOrganizationInner(ctx, &unit))
    }

    return assets, nil
}
```

4.4.8 Delete Organization

To delete an organization the method in Listing 4.28 is used. It receives from the parameter the organization's id to be deleted and returns an error if not possible to delete. To delete the code checks if the client has the correct role to be able to do the action.

Listing 4.28: Delete Organization

```
func (s *SmartContract) DeleteOrganization(ctx contractapi.TransactionContextI
    if err := s.HasPermission(ctx, OrganizationsDelete); err != nil {
        return err
    }

    err := ctx.GetStub().DelState(s.GetOrganizationID(ctx, id))
    if err != nil {
        return fmt.Errorf("failed to delete asset %s: %v", id, err)
    }
}
```

```
        return nil
    }
}
```

4.5 Order

4.5.1 Structure

There is two structures representing orders, Listing 4.29, *Order* and *OrderInner*, these two have different purposes, *OrderInner* represents the data stored in the database with the additional composing *Doc* type. The function in Listing 4.30 is used to parse between the *OrderInner* and *Order* easier.

Listing 4.29: Order Structures

```
const (
    OrderDoc DocType = "order"
)

type OrderInner struct {
    Doc

    ID          string    'json:" id" '
    Amount      uint32    'json:" amount" '
    Price       Price     'json:" price" '
    Type        OrderType 'json:" type" '
    Status      OrderStatus 'json:" status" '
    OrganizationID string    'json:" organization_id" '
    ProductID   string    'json:" product_id" '
    UnitID      string    'json:" unit_id" '
}

type Order struct {
```

```

ID          string          'json:" id" '
Amount      uint32         'json:" amount" '
Price       Price          'json:" price" '
Type        OrderType     'json:" type" '
Status      OrderStatus   'json:" status" '
Organization *Organization  'json:" organization" '
Product     *Product      'json:" product" '
Unit        *Unit         'json:" unit" '
}

```

Listing 4.30: Order Mapper

```

func (s *SmartContract) FromOrderInner(ctx contractapi.TransactionContextInterface,
    org, _ := s.GetOrganization(ctx, p.OrganizationID)
    product, _ := s.GetProduct(ctx, p.ProductID)
    unit, _ := s.GetUnit(ctx, p.UnitID)

    return &Order{
        ID:          p.ID,
        Amount:      p.Amount,
        Price:       Price{
            Amount:    p.Price.Amount,
            Exponent:  p.Price.Exponent,
            Currency:  p.Price.Currency,
        }
        Type:       p.Type,
        Status:     p.Status,
        Organization: org,
        Product:    product,
        Unit:       unit,
    }
}

```

4.5.2 Order Type

Listing 4.31 shows the two order types present in the system, *BUY* and *SELL*, for now, the system will now use sell orders but, the type will be present anyway. The same figure also shows the mapper to map from a string to the type *OrderType*.

Listing 4.31: Order Type

```
const (  
    OrderTypeBuy    OrderType = "BUY"  
    OrderTypeSell   OrderType = "SELL"  
)  
  
type OrderType string  
  
func (o OrderType) String() string {  
    return string(o)  
}  
  
func ParseOrderType(_type string) (OrderType, error) {  
    switch _type {  
    case "BUY":  
        return OrderTypeBuy, nil  
    case "SELL":  
        return OrderTypeSell, nil  
    }  
    return "", fmt.Errorf("invalid _order _type")  
}
```

4.5.3 Order Status

Listing 4.32 shows the two order status present in the system, *OPEN* and *CLOSED*. The same figure also shows the mapper to map from a string to the type *OrderStatus*.

Listing 4.32: Order Status

```
const (
    OrderStatusOpen    OrderStatus = "OPEN"
    OrderStatusClosed  OrderStatus = "CLOSED"
)

type OrderStatus string

func (o OrderStatus) String() string {
    return string(o)
}

func ParseOrderStatus(status string) (OrderType, error) {
    switch status {
    case "OPEN":
        return OrderStatusOpen, nil
    case "CLOSED":
        return OrderStatusClosed, nil
    }
    return "", fmt.Errorf("invalid_order_status")
}
```

4.5.4 Get Order

The *GetOrderID*, Listing 4.33, build a storage order id of the input from the user by adding the order document type.

Listing 4.33: Build Order ID

```
func (s *SmartContract) GetOrderID(ctx contractapi.TransactionContextInterface
    return string(OrderDoc) + "_" + id
}
```

4.5.5 Order Exist

Both methods *OrderExist* and *OrdersExist*, Listing 4.34, checks if orders exist, the first one takes a single id, the second one receives a list of ids and checks if they exist in the current state.

Listing 4.34: Order Exists

```
func (s *SmartContract) OrderExist(ctx contractapi.TransactionContextInterface
    assetJSON, err := ctx.GetStub().GetState(s.GetOrderID(ctx, id))
    if err != nil {
        return false, fmt.Errorf("failed to read from world state: %v", err)
    }

    return assetJSON != nil, nil
}

func (s *SmartContract) OrdersExist(ctx contractapi.TransactionContextInterface
    for _, id := range ids {
        e, err := s.OrderExist(ctx, id)
        if err != nil {
            return err
        }
        if !e {
            return fmt.Errorf("order %s does not exist", id)
        }
    }

    return nil
}
```

4.5.6 Create Order

To create a new order the user need to invoke the *CreateOrder* method with the id, amount, price, price exponent, currency, type, organization id, product id, and unit id. These parameters are then pass through to the method in Listing 4.35. It first if the current context user has the permissions for creating an order. Then checks if the organization, product, and unit exist and if an order with the same id already exists, if so returns an error saying it already exists, otherwise continues to the next step. After that, the client id is retrieved from the context, and the new order is built. Finally, the order is committed to the state, creating a new order ready to be retrieved.

Listing 4.35: Create Order

```
func (s *SmartContract) CreateOrder(  
    ctx contractapi.TransactionContextInterface ,  
    id string , amount uint32 , price uint32 , priceExponent uint32 , currency  
    typeInput string , organizationID string , productID string , unitID string  
) error {  
    if err := s.HasPermission(ctx , OrdersCreate); err != nil {  
        return err  
    }  
  
    orgID , err := s.GetSubmittingClientOrganization(ctx)  
    if err != nil || orgID != organizationID {  
        return fmt.Errorf("unauthorized")  
    }  
  
    exists , err := s.OrderExist(ctx , id)  
    if err != nil {  
        return err  
    }  
    if exists {  
        return fmt.Errorf("the asset %s already exists" , id)
```



```
}

hasOrg, err := s.OrganizationExist(ctx, organizationID)
if err != nil {
    return err
}
if !hasOrg {
    return fmt.Errorf("organization %s does not exists", id)
}

hasProduct, err := s.ProductExist(ctx, productID)
if err != nil {
    return err
}
if !hasProduct {
    return fmt.Errorf("product %s does not exists", id)
}

hasUnit, err := s.UnitExist(ctx, unitID)
if err != nil {
    return err
}
if !hasUnit {
    return fmt.Errorf("unit %s does not exists", id)
}

clientID, err := s.GetSubmittingClientIdentity(ctx)
if err != nil {
    return err
}
```

```
_type, err := ParseOrderType(typeInput)
if err != nil {
    return err
}

doc := Doc{
    Type:      OrderDoc,
    CreatedBy: clientID,
    UpdatedBy: clientID,
}

unit := OrderInner{
    Doc:          doc,
    ID:           s.GetOrderID(ctx, id),
    Amount:       amount,
    Price:        Price{Amount: price, Exponent: priceExponent},
    Type:         _type,
    Status:       OrderStatusOpen,
    OrganizationID: organizationID,
    ProductID:    productID,
    UnitID:       unitID,
}

assetBytes, err := json.Marshal(unit)
if err != nil {
    return err
}

err = ctx.GetStub().PutState(unit.ID, assetBytes)
```

```
    if err != nil {
        return err
    }

    return nil
}
```

4.5.7 Close Order

To close an order, the user needs to invoke the *CloseOrder* method with the id. These parameters are then pass through to the method in Listing 4.36. It first if the current context user has the permissions for closing an order. Then checks if the order with the same id already exists, if it does not exist returns an error saying it does not exist, otherwise continues to the next step. It also checks if the order is already closed and returns an error if so. After that, the client id is retrieved from the context, and the new order is built. Finally, the updated order is committed to the state.

Listing 4.36: Close Order

```
func (s *SmartContract) CloseOrder(ctx contractapi.TransactionContextInterface
    if err := s.HasPermission(ctx, OrdersUpdate); err != nil {
        return err
    }

    exists, err := s.OrderExist(ctx, id)
    if err != nil {
        return err
    }
    if exists {
        return fmt.Errorf("the asset %s already exists", id)
    }

    order, err := s.GetOrderInner(ctx, id)
```

```
    if err != nil {
        return err
    }

    orgID, err := s.GetSubmittingClientOrganization(ctx)
    if err != nil || orgID != order.OrganizationID {
        return fmt.Errorf("unauthorized")
    }

    if order.Status == OrderStatusClosed {
        return fmt.Errorf("you_do_not_belong_in_that_org")
    }

    order.Status = OrderStatusClosed

    assetBytes, err := json.Marshal(order)
    if err != nil {
        return err
    }

    err = ctx.GetStub().PutState(order.ID, assetBytes)

    return nil
}
```

4.5.8 Get Order

The *GetOrder*, Listing 4.37 receives a order id from the parameters and returns a order or an error. It checks to see if the client requesting the operation has the correct attribute role and gets the order from the state. Before sending the data back to the client, it removes the storage id that adds the document type. The Listing 4.38 represent the same

method but instead of returning the type *Order*, returns *OrderInner*.

Listing 4.37: Get Order

```
func (s *SmartContract) GetOrder(ctx contractapi.TransactionContextInterface,
    if err := s.HasPermission(ctx, OrdersRead); err != nil {
        return nil, err
    }

    assetBytes, err := ctx.GetStub().GetState(s.GetOrderID(ctx, id))
    if err != nil {
        return nil, fmt.Errorf("failed to get asset %s: %v", id, err)
    }

    if assetBytes == nil {
        return nil, fmt.Errorf("asset %s does not exist", id)
    }

    var unit OrderInner
    err = json.Unmarshal(assetBytes, &unit)
    if err != nil {
        return nil, err
    }

    unit.ID = strings.TrimPrefix(unit.ID, string(OrderDoc)+"_")
    return s.FromOrderInner(ctx, &unit), nil
}
```

Listing 4.38: Get Order Inner

```
func (s *SmartContract) GetOrderInner(ctx contractapi.TransactionContextInterf
    if err := s.HasPermission(ctx, OrdersRead); err != nil {
        return nil, err
    }
}
```

```
assetBytes, err := ctx.GetStub().GetState(s.GetOrderID(ctx, id))
if err != nil {
    return nil, fmt.Errorf("failed to get asset %s: %v", id, err)
}

if assetBytes == nil {
    return nil, fmt.Errorf("asset %s does not exist", id)
}

var unit OrderInner
err = json.Unmarshal(assetBytes, &unit)
if err != nil {
    return nil, err
}

unit.ID = strings.TrimPrefix(unit.ID, string(OrderDoc)+"_")
return &unit, nil
}
```

4.5.9 List Orders

The method in Listing 4.39, *ListOrders*, lists all the orders currently saved in the state. It first checks for permissions, then returns the orders to the client. There are three variants of this method, the first one, *ListOrdersByStatus* filters orders by stats, Listing 4.39. The second, *ListOrdersByOrg*, filters orders by organizations, Listing 4.39. Finally, the third variant combines the two previous by filtering by organization and status, *ListOrdersByOrgAndStatus*, Listing 4.39. The methods all return the same type of data.

Listing 4.39: List Orders

```
func (s *SmartContract) ListOrders(ctx contractapi.TransactionContextInterface
    if err := s.HasPermission(ctx, OrdersRead); err != nil {
```

```
        return nil, err
    }

    results, err := ctx.GetStub().GetQueryResult(fmt.Sprintf("%s selector":
if err != nil {
        return nil, fmt.Errorf("failed to get assets: %v", err)
    }
defer results.Close()

var assets []*Order
for results.HasNext() {
    queryResult, err := results.Next()
    if err != nil {
        return nil, err
    }
    var unit OrderInner
    err = json.Unmarshal(queryResult.Value, &unit)
    if err != nil {
        return nil, err
    }

    unit.ID = strings.TrimPrefix(unit.ID, string(OrderDoc)+"_")
    assets = append(assets, s.FromOrderInner(ctx, &unit))
}

return assets, nil
}

func (s *SmartContract) ListOrdersByStatus(ctx contractapi.TransactionContextI
    if err := s.HasPermission(ctx, OrdersRead); err != nil {
```

```
        return nil, err
    }

    status, err := ParseOrderStatus(statusInput)
    if err != nil {
        return nil, err
    }

    results, err := ctx.GetStub().GetQueryResult(fmt.Sprintf("{ selector":
    if err != nil {
        return nil, fmt.Errorf("failed to get assets: %v", err)
    }
    defer results.Close()

    var assets []*Order
    for results.HasNext() {
        queryResult, err := results.Next()
        if err != nil {
            return nil, err
        }
        var unit OrderInner
        err = json.Unmarshal(queryResult.Value, &unit)
        if err != nil {
            return nil, err
        }

        unit.ID = strings.TrimPrefix(unit.ID, string(OrderDoc)+"_")
        assets = append(assets, s.FromOrderInner(ctx, &unit))
    }
}
```



```
    return assets, nil
}

func (s *SmartContract) ListOrdersByOrg(ctx contractapi.TransactionContextInterface) ([]*Order, error) {
    if err := s.HasPermission(ctx, OrdersRead); err != nil {
        return nil, err
    }

    results, err := ctx.GetStub().GetQueryResult(fmt.Sprintf("{selector}"))
    if err != nil {
        return nil, fmt.Errorf("failed to get assets: %v", err)
    }
    defer results.Close()

    var assets []*Order
    for results.HasNext() {
        queryResult, err := results.Next()
        if err != nil {
            return nil, err
        }
        var unit OrderInner
        err = json.Unmarshal(queryResult.Value, &unit)
        if err != nil {
            return nil, err
        }

        unit.ID = strings.TrimPrefix(unit.ID, string(OrderDoc)+"_")
        assets = append(assets, s.FromOrderInner(ctx, &unit))
    }
}
```

```
    return assets, nil
}

func (s *SmartContract) ListOrdersByOrgAndStatus(ctx contractapi.TransactionContext) (
    if err := s.HasPermission(ctx, OrdersRead); err != nil {
        return nil, err
    }

    status, err := ParseOrderStatus(statusInput)
    if err != nil {
        return nil, err
    }

    results, err := ctx.GetStub().GetQueryResult(fmt.Sprintf("{ selector":
    if err != nil {
        return nil, fmt.Errorf("failed to get assets: %v", err)
    }
    defer results.Close()

    var assets []*Order
    for results.HasNext() {
        queryResult, err := results.Next()
        if err != nil {
            return nil, err
        }
        var unit OrderInner
        err = json.Unmarshal(queryResult.Value, &unit)
        if err != nil {
            return nil, err
        }
    }
}
```

```
        unit.ID = strings.TrimPrefix(unit.ID, string(OrderDoc)+"_")
        assets = append(assets, s.FromOrderInner(ctx, &unit))
    }

    return assets, nil
}
```

4.6 Transaction

4.6.1 Structure

There is two structures representing transactions, *Transaction* and *TransactionInner*, these two have different purposes, *TransactionInner* represents the data stored in the database with the additional composing *Doc* type, the other is the output structure. The function in Listing 4.40 is used to parse between the *TransactionInner* and *Transaction* easier.

Listing 4.40: Transaction Structure

```
const (
    TransactionDoc DocType = "transaction"
)

type TransactionInner struct {
    Doc

    ID          string          'json:" id" '
    Amount      uint32         'json:" amount" '
    Description  string         'json:" description" '
    Status       TransactionStatus 'json:" status" '
    OrganizationID string      'json:" organization_id" '
    OrderID      string         'json:" order_id" '
}
```

```
}
```

```
type Transaction struct {  
    ID          string          'json:" id" '  
    Amount      uint32           'json:" amount" '  
    Description  string          'json:" description" '  
    Status      TransactionStatus 'json:" status" '  
    OrganizationID string        'json:" organization_id" '  
    OrderID     string          'json:" order_id" '  
}
```

```
func (s *SmartContract) FromTransactionInner(_ contractapi.TransactionContextI  
    return &Transaction{  
        ID:          p.ID,  
        Amount:      p.Amount,  
        Description: p.Description,  
        Status:      p.Status,  
        OrganizationID: p.OrganizationID,  
        OrderID:     p.OrderID,  
    }  
}
```

4.6.2 Transaction Status

Listing 4.41 shows the ten transaction status present in the system, *OPEN*, *CLOSED*, *CANCELED*, *IN_REVIEW*, *WAITING_PAYMENT*, *PAID*, *READY*, *IN_PROGRESS*, *NOT_DELIVERED*, and *DELIVERED*. The Listing 4.41 shows the mapper to map from a string to the type *TransactionStatus*.

Listing 4.41: Transaction Status

```
const (  
    TransactionStatusOpen          TransactionStatus = "OPEN"
```

```
TransactionStatusClosed      TransactionStatus = "CLOSED"
TransactionStatusCanceled    TransactionStatus = "CANCELED"
TransactionStatusInReview    TransactionStatus = "IN_REVIEW"
TransactionStatusWaitingPayment TransactionStatus = "WAITING_PAYMENT"
TransactionStatusPaid        TransactionStatus = "PAID"
TransactionStatusReady       TransactionStatus = "READY"
TransactionStatusInProgress   TransactionStatus = "IN_PROGRESS"
TransactionStatusNotDelivered TransactionStatus = "NOT_DELIVERED"
TransactionStatusDelivered    TransactionStatus = "DELIVERED"
)
```

```
type TransactionStatus string
```

```
func (o TransactionStatus) String() string {
    return string(o)
}
```

```
func ParseTransactionStatus(status string) (TransactionStatus, error) {
    switch status {
    case "OPEN":
        return TransactionStatusOpen, nil
    case "CLOSED":
        return TransactionStatusClosed, nil
    case "CANCELED":
        return TransactionStatusCanceled, nil
    case "WAITING_PAYMENT":
        return TransactionStatusWaitingPayment, nil
    case "PAID":
        return TransactionStatusPaid, nil
    case "IN_REVIEW":
```

```
        return TransactionStatusInReview, nil
    case "READY":
        return TransactionStatusReady, nil
    case "IN_PROGRESS":
        return TransactionStatusInProgress, nil
    case "NOT_DELIVERED":
        return TransactionStatusNotDelivered, nil
    case "DELIVERED":
        return TransactionStatusDelivered, nil
    }
    return "", fmt.Errorf("invalid_transaction_type")
}
```

4.6.3 Get Transaction ID

The *GetTransactionID*, Listing 4.42, build a storage order id of the input from the user by adding the transaction document type.

Listing 4.42: Get Transaction ID

```
func (s *SmartContract) GetTransactionID(_ contractapi.TransactionContextInterface, id string) string {
    return string(TransactionDoc) + "_" + id
}
```

4.6.4 Transaction Exist

The method *TransactionExist*, Listing 4.43 checks if a transaction exists, it takes a single id and checks if it exists in the current state.

Listing 4.43: Transaction Exist

```
func (s *SmartContract) TransactionExist(ctx contractapi.TransactionContextInterface, id string) bool {
    assetJSON, err := ctx.GetStub().GetState(s.GetTransactionID(ctx, id))
    if err != nil {
        return false, fmt.Errorf("failed_to_read_from_world_state: %v", err)
    }
    return true
}
```

```
    }  
  
    return assetJSON != nil, nil  
}
```

4.6.5 Make Transaction

To create a new transaction, the user needs to invoke the *MakeTransaction* method with the id, amount, organization id, and the order id. These parameters are then pass through to the method in Listing 4.44. It first if the current context user has the permissions for creating transactions. Then checks if the organization and order exist and if a transaction with the same id already exists, if so returns an error saying it already exists, otherwise continues to the next step. It also gets all transactions for the order and checks if the amount of product in all the transactions exceeds the amount in the order, if so returns an error. After that, the client id is retrieved from the context, and the new order is built. Finally, the order is committed to the state, creating a new order ready to be retrieved.

Listing 4.44: Make Transaction

```
func (s *SmartContract) MakeTransaction(  
    ctx contractapi.TransactionContextInterface ,  
    id string, amount uint32, organizationID string, orderID string,  
) error {  
    if err := s.HasPermission(ctx, TransactionsCreate); err != nil {  
        return err  
    }  
  
    exists, err := s.TransactionExist(ctx, id)  
    if err != nil {  
        return err  
    }  
    if exists {
```

```
        return fmt.Errorf("the asset %s already exists", id)
    }

    hasOrg, err := s.OrganizationExist(ctx, organizationID)
    if err != nil {
        return err
    }
    if hasOrg {
        return fmt.Errorf("organization %s does not exist", id)
    }

    order, err := s.GetOrder(ctx, orderID)
    if err != nil {
        return err
    }

    transactions, err := s.ListTransactionsForOrderInner(ctx, orderID)
    if err != nil {
        return err
    }

    transactionsAmount := getTransactionsAmount(transactions)
    if transactionsAmount+amount > order.Amount {
        return fmt.Errorf("invalid amount to transact")
    }

    clientID, err := s.GetSubmittingClientIdentity(ctx)
    if err != nil {
        return err
    }
}
```



```
doc := Doc{
    Type:      TransactionDoc ,
    CreatedBy: clientID ,
    UpdatedBy: clientID ,
}

transaction := TransactionInner{
    Doc:      doc ,
    ID:      s.GetUnitID(ctx , id) ,
    Amount:  amount ,
    Status:  TransactionStatusOpen ,
    OrganizationID: organizationID ,
    OrderID: orderID ,
}

assetBytes , err := json.Marshal(transaction)
if err != nil {
    return err
}

err = ctx.GetStub().PutState(transaction.ID , assetBytes)
if err != nil {
    return err
}

eventBody , err := NewNewTransactionEvent(transaction.ID)
if err != nil {
    return err
}
```

```
err = ctx.GetStub().SetEvent(NewTransactionEventKey, eventBody)
if err != nil {
    return err
}

return nil
}
```

4.6.6 Change Transaction Status

To change a transaction status, the user needs to invoke the *ChangeStatus* method with the id, new status, and a message log. These parameters are then pass through to the method in Listing 4.45. It first if the current context user has the permissions for updating a transaction. Then checks if the transaction with the same id already exists, if it does not exist returns an error saying it does not exist, otherwise continues to the next step. It also checks if the transaction is already closed or canceled and returns an error if so. After that, the client id is retrieved from the context, and the new transaction is built, Listing 4.45. Finally, the updated transaction is committed to the state.

Listing 4.45: Change Transaction Status

```
func (s *SmartContract) ChangeStatus(ctx contractapi.TransactionContextInterface,
    if err := s.HasPermission(ctx, TransactionsUpdate); err != nil {
        return err
    }

    status, err := ParseTransactionStatus(inputStatus)
    if err != nil {
        return err
    }

    exists, err := s.TransactionExist(ctx, id)
```

```
    if err != nil {
        return err
    }
    if exists {
        return fmt.Errorf("the asset %s already exists", id)
    }

    transaction, err := s.GetTransactionInner(ctx, id)
    if err != nil {
        return err
    }

    if transaction.Status == TransactionStatusClosed || transaction.Status
        return fmt.Errorf("transaction already closed or canceled")
    }

    order, err := s.GetOrderInner(ctx, transaction.OrderID)
    if err != nil {
        return err
    }

    orgID, err := s.GetSubmittingClientOrganization(ctx)
    if err != nil {
        return err
    }

    switch orgID {
    case transaction.OrganizationID:
        if status == TransactionStatusPaid ||
            status == TransactionStatusInReview ||
```

```
        status == TransactionStatusWaitingPayment ||
        status == TransactionStatusReady ||
        status == TransactionStatusInProgress {
            return fmt.Errorf("you do not have permissions to do that")
        }
case order.OrganizationID:
default:
            return fmt.Errorf("you do not have permissions to do that")
    }

    clientID, err := s.GetSubmittingClientIdentity(ctx)
    if err != nil {
        return err
    }

    oldStatus := transaction.Status
    transaction.Status = status
    transaction.Description = message
    transaction.UpdatedBy = clientID

    assetBytes, err := json.Marshal(transaction)
    if err != nil {
        return err
    }

    err = ctx.GetStub().PutState(transaction.ID, assetBytes)
    if err != nil {
        return err
    }
}
```

```

eventBody, err := NewTransactionStatusChangedEvent(transaction.ID, old
if err != nil {
    return err
}

err = ctx.GetStub().SetEvent(TransactionStatusChangedEventKey, eventBo
if err != nil {
    return err
}

return nil
}

```

4.6.7 Get Transaction

The *GetTransaction*, Listing 4.46 receives a transaction id from the parameters and returns a transaction or an error. It checks to see if the client requesting the operation has the correct attribute role and gets the transaction from the state. Before sending the data back to the client, it removes the storage id that adds the document type. The Listing 4.46 represent the same method but instead of returning the type *Transaction*, returns a *TransactionInner* structure.

Listing 4.46: Get Transaction and Get Transaction Inner

```

func (s *SmartContract) GetTransactionInner(ctx contractapi.TransactionContext
    if err := s.HasPermission(ctx, TransactionsRead); err != nil {
        return nil, err
    }

    assetBytes, err := ctx.GetStub().GetState(s.GetOrderID(ctx, id))
    if err != nil {
        return nil, fmt.Errorf("failed to get asset %s: %v", id, err)
    }

```

```
    if assetBytes == nil {
        return nil, fmt.Errorf("asset %s does not exist", id)
    }

    var unit TransactionInner
    err = json.Unmarshal(assetBytes, &unit)
    if err != nil {
        return nil, err
    }

    unit.ID = strings.TrimPrefix(unit.ID, string(TransactionDoc)+"_")
    return &unit, nil
}

func (s *SmartContract) GetTransaction(ctx contractapi.TransactionContextInterface) (TransactionInner, error) {
    if err := s.HasPermission(ctx, TransactionsRead); err != nil {
        return nil, err
    }

    assetBytes, err := ctx.GetStub().GetState(s.GetOrderID(ctx, id))
    if err != nil {
        return nil, fmt.Errorf("failed to get asset %s: %v", id, err)
    }

    if assetBytes == nil {
        return nil, fmt.Errorf("asset %s does not exist", id)
    }

    var unit TransactionInner
```

```
err = json.Unmarshal(assetBytes, &unit)
if err != nil {
    return nil, err
}

unit.ID = strings.TrimPrefix(unit.ID, string(TransactionDoc)+"_")
return s.FromTransactionInner(ctx, &unit), nil
}
```

4.6.8 List Transactions

The method in Listing 4.47, *ListTransactions*, lists all the transactions currently saved in the state. It first checks for permissions, then returns the transactions to the client. There are one variant of this method, *ListTransactionsInner* that is same method but instead of returning a list of *Transaction*, returns *TransactionInner*, Listing 4.47.

Listing 4.47: List Transactions

```
func (s *SmartContract) ListTransactionsForOrderInner(ctx contractapi.TransactionContext) (*[]*TransactionInner, error) {
    if err := s.HasPermission(ctx, TransactionsRead); err != nil {
        return nil, err
    }

    results, err := ctx.GetStub().GetQueryResult(fmt.Sprintf("{selector}"))
    if err != nil {
        return nil, fmt.Errorf("failed to get assets: %v", err)
    }
    defer results.Close()

    var assets []*TransactionInner
    for results.HasNext() {
        queryResult, err := results.Next()
        if err != nil {
```

```
        return nil, err
    }
    var unit TransactionInner
    err = json.Unmarshal(queryResult.Value, &unit)
    if err != nil {
        return nil, err
    }

    unit.ID = strings.TrimPrefix(unit.ID, string(TransactionDoc)+")")
    assets = append(assets, &unit)
}

return assets, nil
}

func (s *SmartContract) ListTransactionsForOrder(ctx contractapi.TransactionContext) (
    if err := s.HasPermission(ctx, TransactionsRead); err != nil {
        return nil, err
    }

    results, err := ctx.GetStub().GetQueryResult(fmt.Sprintf("%s selector", s.Name))
    if err != nil {
        return nil, fmt.Errorf("failed to get assets: %v", err)
    }
    defer results.Close()

    var assets []*Transaction
    for results.HasNext() {
        queryResult, err := results.Next()
        if err != nil {
```



```
        return nil, err
    }
    var unit TransactionInner
    err = json.Unmarshal(queryResult.Value, &unit)
    if err != nil {
        return nil, err
    }

    unit.ID = strings.TrimPrefix(unit.ID, string(TransactionDoc)+")
    assets = append(assets, s.FromTransactionInner(ctx, &unit))
}

return assets, nil
}
```

Chapter 5

Results and Analysis

The proposed architecture has identified advantages when compared with traditional systems. It has quicker transactions as there is no need to wait for email responses. The system also allows faster entity integration into the network as it does not require going through the process of building trust with every participant. The network facilitates the role of the auditor, as it does not need to ask for the records of each entity, as it already has direct access to everything recorded in the network.

There are also some identified upgrades to the system. Currently, users interact with it by using a Command-Line Interface (CLI). This setup for the target audience can be a problem and be hard to learn. In this case, a simple front-end app could help to fix the situation. Transactions speed can improve by adding a built-in cryptocurrency, removing the manual intervention on payments. This cryptocurrency would be a stable representation of the local currency, for example, the euro.

This section gives four examples of how to interact with the network. The examples are how a new organization joins the network, how to remove an organization from the system, creating a new sell order, and finally, how to buy a product.

5.1 Creation a New Organization

The first thing for a new organization to be able to join the networks is its nodes, a way to communicate with it, for that a peer node is required. Once the peer node is created and configured, it's time to set up the necessary admin user. The founder creates

a new certificate for the admin user and enrolls him into the system. The rest of the users are created by this admin, this will require a CA to create certificates for each one of the users he wants to give access to.

5.2 Removing an Organization

The process of removing an organization is simple as revoking the CA certificate authority from accessing the network. After that is just required to cancel all the sell orders which the organization has active.

5.3 Creating a new sell order

To create a sell order, the user needs to have the necessary role on his certificate. If the user's certificate has the correct permission, the CLI tool will perform the request using a fabric peer.

Listing 5.1: List Products

```
peer chaincode query -C mychannel -n basic-1 -c '{"Args":["ListProducts"]}' |  
  
[  
  {  
    "id": "areia",  
    "name": "Areia",  
    "description": "Areia",  
    "units": [  
      {  
        "id": "kg",  
        "name": "Kilogramas",  
        "description": "Kilogramas",  
        "exponent": 2  
      }  
    ]  
  }  
]
```

```
    }  
  ]
```

First, the user uses the `ListProducts` query to select and check the product id and unit id of the product he wants to put on in the sell order. Second, the invoking of the `CreateOrder` using as arguments the order id, amount of the product, price of each unit, price exponent, currency code, order type, organization id of the user, selected product id, and unit id.

Listing 5.2: Create Order

```
peer chaincode invoke -o localhost:7050 --ordererTLSTLSHostnameOverride orderer.e  
  
2021-11-30 00:32:34.997 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001
```

The request then gets a response from the peer node with a success or failure message. If successful, the order is correctly inserted into the ledger, otherwise, the problem is presented to the user and he can fix the issue. Some failure messages examples are:

- "product id not found" - the product does not exist
- "unit id not found" - unit does not exist
- "the asset already exists" - already exists an order with that id

5.4 Buying a product

To buy a product, the user needs to have the necessary role on his certificate. If the user's certificate has the correct permission, the CLI tool will perform the request using a fabric peer.

Listing 5.3: List Orders

```
peer chaincode query -C mychannel -n basic-1 -c '{"Args":["ListOrders"]}' | jq  
  
[  
  {
```

```
"id": "order-1",
"amount": 2000,
"price": {
  "amount": 1134,
  "exponent": 2,
  "currency": "EUR"
},
"type": "SELL",
"status": "OPEN",
"organization": {
  "id": "produtor-areia",
  "name": "Produtor_de_Areia_#1",
  "description": "Produtor_de_Areia_Testes",
  "address": "Morada_Exemplo",
  "phone_number": "222444666"
},
"product": {
  "id": "areia",
  "name": "Areia",
  "description": "Areia",
  "units": [
    {
      "id": "kg",
      "name": "Kilogramas",
      "description": "Kilogramas",
      "exponent": 2
    }
  ]
},
"unit": {
```

```

    "id": "kg" ,
    "name": "Kilogramas" ,
    "description": "Kilogramas" ,
    "exponent": 2
  }
}
]

```

First, the user uses the ListOrders query to select and check the order id of the order he wants to buy. Second, the invoking of the MakeTransaction using as arguments the transaction id, amount of the product, organization id of the user, and order id.

Listing 5.4: Make Transaction

```

peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.e
2021-11-30 00:32:34.997 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001

```

The request then gets a response from the peer node with a success or failure message. If successful, the order is correctly inserted into the ledger, otherwise, the problem is presented to the user and he can fix the issue. Some failure messages examples are:

- "order id not found" - the order does not exist
- "organization id not found" - the organization does not exist
- "invalid amount to transact" - the amount overflows the amount left for selling
- "the asset already exists" - already exists a transaction with that id

Listing 5.5: List Transactions

```

peer chaincode query -C mychannel -n basic-1 -c '{"Args":["ListTransactionsFor
[
{
  "id": "unit_transaction-1" ,

```

```
"amount": 100,  
"description": "",  
"status": "OPEN",  
"organization_id": "produtor-papel",  
"order_id": "order-1"  
}  
]
```

Chapter 6

Conclusion

The Industrial Symbiosis movement is trying to help the world by preventing the waste of by-products and making industries greener. Blockchain solutions, when applied to Industrial Symbiosis systems, can help it reach a global scale, making the world more sustainable. In particular, blockchain-based system can improve the Industrial Symbiosis process between companies by assuring trust between of entities and transparency of their transactions. This thesis proposes a blockchain architecture design to enhance the Industrial Symbiosis process of the Pulp, Paper, and Cardboard Production Sector Companies in Portugal, providing the required trust and transparency to a network to enhance their Industrial Symbiosis process.

The proposed architecture uses a permissioned ledger. Hyperledger Fabric is the kernel of the network allowing the necessary customization for the adopted scenario. Chaincode and Attribute-Based Access Control of the Hyperledger Fabric are the key pieces of the implementation. Using chaincode it is possible to implement the required logic that allows users to create orders and transactions.

The proposed blockchain-based solution was found to have advantages when compared to the traditional systems. Firstly, it enables fast transactions and quick integration of a new organization into the network. Then, the fact that the blockchain records every transaction makes it easier to audit. Finally, the blockchain-based system makes it easier to scale as it just requires each new company to setup one peer and one CA node.

As future work, some improvements were already identified. The first improvement is the development of a front-end web application to prevent users from using the Hy-

perledger Fabric's command-line interface. Another improvement can be in the speed of the transactions, which can improve by adding a built-in cryptocurrency to remove the manual intervention on payments.

References

- [1] G. Alexandris et al. “Blockchains as Enablers for Auditing Cooperative Circular Economy Networks”. In: *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. 2018, pp. 1–7. DOI: 10.1109/CAMAD.2018.8514985.
- [2] Elli Androulaki et al. “Hyperledger fabric: a distributed operating system for permissioned blockchains”. In: *Proceedings of the thirteenth EuroSys conference*. 2018, pp. 1–15.
- [3] Fabrice Benhamouda, Shai Halevi, and Tzipora Halevi. “Supporting private data on hyperledger fabric with secure multiparty computation”. In: *IBM Journal of Research and Development* 63.2/3 (2019), pp. 3–1.
- [4] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. “Cryptocurrencies without proof of work”. In: *International conference on financial cryptography and data security*. Springer. 2016, pp. 142–157.
- [5] Alexa Böckel, Anne-Katrin Nuzum, and Ilka Weissbrod. “Blockchain for the Circular Economy: Analysis of the Research-Practice Gap”. In: *Sustainable Production and Consumption* 25 (2021), pp. 525–539. ISSN: 2352-5509. DOI: <https://doi.org/10.1016/j.spc.2020.12.006>. URL: <https://www.sciencedirect.com/science/article/pii/S2352550920314056>.
- [6] Vitalik Buterin. “What is Ethereum?” In: *Accessed on: Dec 15* (2013).
- [7] Christian Cachin et al. “Architecture of the hyperledger blockchain fabric”. In: *Workshop on distributed cryptocurrencies and consensus ledgers*. Vol. 310. 4. Chicago, IL. 2016.

- [8] Marian R Chertow. “Industrial symbiosis: literature and taxonomy”. In: *Annual review of energy and the environment* 25.1 (2000), pp. 313–337.
- [9] Shauhrat S Chopra and Vikas Khanna. “Understanding resilience in industrial symbiosis networks: Insights from network analysis”. In: *Journal of environmental management* 141 (2014), pp. 86–94.
- [10] Vikram Dhillon, David Metcalf, and Max Hooper. “The hyperledger project”. In: *Blockchain enabled applications*. Springer, 2017, pp. 139–149.
- [11] Vitalik Buterin Fabian Vogelsteller. “EIP-20: ERC-20 Token Standard”. In: *Ethereum Improvement Proposals* 20 (2015).
- [12] Ghareeb Falazi et al. “Process-based composition of permissioned and permissionless blockchain smart contracts”. In: *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE. 2019, pp. 77–87.
- [13] Inês A Ferreira, Marta S Barreiros, and Helena Carvalho. “The industrial symbiosis network of the biomass fluidized bed boiler sand—Mapping its value network”. In: *Resources, Conservation and Recycling* 149 (2019), pp. 595–604.
- [14] Inês A Ferreira, Marta S Barreiros, and Helena Carvalho. “The industrial symbiosis network of the biomass fluidized bed boiler sand—Mapping its value network”. In: *Resources, Conservation and Recycling* 149 (2019), pp. 595–604.
- [15] Arthur Gervais et al. “On the security and performance of proof of work blockchains”. In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 3–16.
- [16] Ricardo Gonçalves et al. “A Smart Contract Architecture to Enhance the Industrial Symbiosis Process Between the Pulp and Paper Companies - A Case Study”. In: *Blockchain and Applications*. Ed. by Javier Prieto et al. Cham: Springer International Publishing, 2022, pp. 252–260. ISBN: 978-3-030-86162-9.
- [17] Wenting Jiao and Frank Boons. “Toward a research agenda for policy intervention and facilitation to enhance industrial symbiosis based on a comprehensive literature review”. In: *Journal of Cleaner Production* 67 (2014), pp. 14–25.

- [18] Paraskevi Katsiampa. “Volatility co-movement between Bitcoin and Ether”. In: *Finance Research Letters* 30 (2019), pp. 221–227.
- [19] Mirko Koscina, Mariusz Lombard-Platet, and Pierre Cluchet. “Plasticcoin: an erc20 implementation on hyperledger fabric for circular economy and plastic reuse”. In: *IEEE/WIC/ACM International Conference on Web Intelligence-Companion Volume*. 2019, pp. 223–230.
- [20] Mahtab Kouhizadeh and Joseph Sarkis. “Blockchain practices, potentials, and perspectives in greening supply chains”. In: *Sustainability* 10.10 (2018), p. 3652.
- [21] Mahtab Kouhizadeh, Joseph Sarkis, and Qingyun Zhu. “At the nexus of blockchain technology, the circular economy, and product deletion”. In: *Applied Sciences* 9.8 (2019), p. 1712.
- [22] Mahtab Kouhizadeh, Qingyun Zhu, and Joseph Sarkis. “Blockchain and the circular economy: potential tensions and critical reflections from practice”. In: *Production Planning & Control* 31.11-12 (2020), pp. 950–966.
- [23] Andrew Miller. “Permissioned and permissionless blockchains”. In: *Blockchain for Distributed Systems Security* (2019), pp. 193–204.
- [24] Bhabendu Kumar Mohanta, Soumyashree S Panda, and Debasish Jena. “An overview of smart contract and use cases in blockchain technology”. In: *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. IEEE. 2018, pp. 1–4.
- [25] Satoshi Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: *Decentralized Business Review* (2008), p. 21260.
- [26] David Cecil Smith, Angela Elizabeth Douglas, et al. *The biology of symbiosis*. Edward Arnold (Publishers) Ltd., 1987.
- [27] Christof Ferreira Torres, Mathis Steichen, et al. “The art of the scam: Demystifying honeypots in ethereum smart contracts”. In: *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 2019, pp. 1591–1607.
- [28] Sarah Underwood. “Blockchain beyond bitcoin”. In: *Communications of the ACM* 59.11 (2016), pp. 15–17.

- [29] Martin Valenta and Philipp Sandner. “Comparison of ethereum, hyperledger fabric and corda”. In: *Frankfurt School Blockchain Center 8* (2017).