Department
of Information Engineering and Computer Science

Master's Degree in
Computer Science

FINAL DISSERTATION

SECURE AND TRANSPARENT DISPUTE RESOLUTION SYSTEM IN SUPPLY CHAINS USING HYPERLEDGER FABRIC

Supervisor UniTrento

Bruno Crispo

Supervisor NEC Laboratories Europe

Ghassan Karame

Student

Cristian Rotari 213321

ACADEMIC YEAR 2019/2020

Acknowledgements

This dissertation would not have been possible without my thesis supervisors from NEC Laboratories Europe, that assisted me during the research, development and writing process of my thesis.
Firstly, I would like to thank Dr. Ghassan Karame, that had my best interests at heart when times were tough and uncertain. He was a great mentor by motivating me in my work and by knowing when to offer constructive criticism or when to show appreciation.
Secondly, I would like to show my thankfulness to Alessandro Sforzin, that assisted me with the technical difficulties that I faced and who was always available in helping me clarify things.

Next, I would like to thank my supervisor from University of Trento, Professor Bruno Crispo, which advised and accommodated me in the process of making this thesis possible.
Similarly, I would like to express my gratitude to Ms. Volha Tarasevich, who continuously provided me throughout my internship and thesis with valuable practical information in a timely manner. Also putting efforts into solving bureaucratic issues.

I would also like to take this time to mention and thank one of my university professors at University of Twente, namely: Dr. Andreas Peter. Thank you for your professionalism, sharing your expertise with me and for always being eager to help.

Moreover, I want to show my appreciation towards EIT Master School, who changed my life for the better on an academic, professional as well as personal level by providing me with a scholarship that made this masters experience possible.

Additionally, thank you to the University of Twente and University of Trento for the support throughout this period and for making this all possible.

A big thank you goes towards anyone who I may have missed that has contributed to this thesis by giving me advice or support when I've asked.

Lastly, I would like to sincerely thank my family and friends who I could always turn to and who continuously had my back.

Abstract

Consumerism has become one of the main pillars of today's world economies. This is putting increased pressure on supply chains to have a good and efficient management. The increased size and complexity of supply chains along with fraud and misunderstandings also results in a high number of disputes between parties. These have the potential not only to disrupt supply chain partnerships and activities, but also lead to time and financial losses.
Blockchain is one of the most promising technologies that can improve the way supply chains are managed. It has the potential to bring transparency and traceability in the system, increase security and reduce administrative costs.
This thesis proposes a dispute resolution blockchain system built with the permissioned Hyperledger Fabric technology, which helps manage and solve disputes in the supply chain easier. The developed system allows easy tracking and access to information among supply chain participants such as the source, current and past states and holder information of assets. It also enables dispute creation, examination and settlement within the app. Because the designed system is meant to be as automated as possible, it implements automated shipment deliveries among organizations. To showcase the system functionalities, a web Graphical User Interface has been developed. Finally, the system design is analyzed from the security point of view and implementation details are discussed.
The paper concludes with the idea that blockchain technology, especially permissioned blockchains, have great applicability in the supply chain management process. Moreover, the dispute resolution process can benefit from such a technology firstly by preventing disputes and secondly by offering means to a transparent and efficient resolution.

Index Terms: Blockchain, distributed ledger, supply chain, dispute resolution, security, traceability.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

We live in a world in which consumerism is a key factor driving world economies. It refers to the consumption of goods in larger quantities which usually motivates the production of goods. Furthermore, it offers a broader range of available products and services, creates jobs and in turn grows the economy of a country. The technological advancements in the last 20 years and the fact that the world is becoming more and more digitalized has impacted the way we live our lives and perform day to day activities. Among those it has also impacted the way people do shopping and acquire goods. Nowadays the number of people preferring online shopping is growing steadily and the current and future pandemic challenges of our society are going to stimulate that trend even more.
An important factor enabling consumerism are the **supply chains**. Supply chains refer to the systems of activities, organizations, information and resources involved in the process of creating and delivering a product to a consumer [1]. A system can be considered a supply chain if it involves three or more organizations or individuals and it may contain upstream and downstream flows of resources. Organizations can take part in multiple supply chains which may vary in size and complexity. The supply chain can not exist without the organizations that form it and perform different functions within it. The typical participants in any supply chain are [2]:

- *Producers*: these organizations are also called manufacturers that produce goods, which in turn can be raw materials consumed by other producers or final products that need to be delivered to the customer. Among these can be companies that mine for natural resources, farms land, grow animals or those manufacturing technological goods such as phones or laptops.
- *Distributors*: also known as wholesalers, buy products from producers in big quantities and sell them to other customers. Their usual customers are other businesses that are interested in buying bigger quantities than the regular customer. This participant has the role of offering demanded products to customer at the right time in the right place. For this they have large inventories of products and act as a buffer between the customers and producers which protects both parties from fluctuations in demand or supply.
- *Retailers*: these participants are the ones having moderately large quantities of products and they sell directly to the end customer. Their primary goal is to make the product attractive to buy by combining service, marketing and pricing strategies.
- *Customers*: these are the identities or organizations that buy and use products. They can either be end customers or another type of supply chain participant that uses the bought product for the creation of other products which in turn get sold to other customers.

Supply chains are considered bidirectional because of the different flows running from different ends. One flow represents the distribution of goods and is going from the producers to the customers. Another one is the financial flow which represents the customers paying for the products they are buying. The third flow describes how information travels within the supply chain and is bidirectional. This is because suppliers get informed about the demand of goods while customers have access to information about the products representing the supply. The participants together with the aforementioned flows are represented in Figure 1.1. One must keep

in mind that this diagram is an extreme simplification of the numerous and different participants from a real supply chain and the intertwined flows between them.

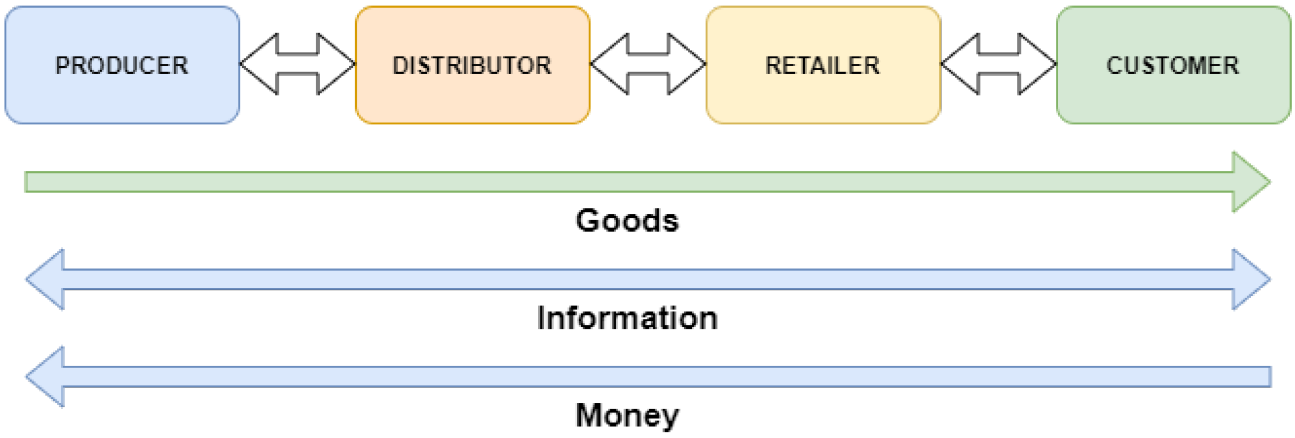

Figure 1.1: Process flow in a simplified supply chain

Supply chain management is the domain responsible for assuring a seamless flow of resources from producers to consumers and vice versa, and for reducing the number of disputes between parties. Even though supply chain management is directly connected to supply chains, the latter can exist without the former. It has been shown that increased complexity within the supply chains has negative consequences on it such as increased operational costs, delays and lack of cooperation and integration between business partners [3].

1.2 Motivation

Because of the previously described tendencies, there is an increased pressure on the supply chains to have a good and efficient management. A simple mistake at the beginning of the chain could have a potential snowballing effect on the consequent processes, which in turn results in financial losses for the involved parties.
If we only look at employees of a company that take part in processing a product, we can understand that only inside this organization a multitude of things can go wrong at various levels. Taking into account the size and complexity of the supply chains and the vast differences between involved participants, it is only natural for conflicts to arise at some point. A lot of the problems arise because of lack of information, information mismatch or lack of trust between parties. As Min mentions in [4], ”contractual disputes resulting from fraud, misunderstanding, and performance failures can not only destroy the supply chain partnerships but also disrupt supply chain activities with prolonged time for resolution”. Frequent disputes between business partners leads to time and financial losses, broken commercial ties and dissatisfaction among involved participants. Therefore, timely and efficient resolution of these problems are essential for a good supply chain management. Currently, the dispute resolution process in a supply chain is a costly process in terms of time and resources and often leads to bigger ruptures. Also, some of the biggest challenges currently faced by supply chains are the lack of transparency and traceability, quality assurance and ultimately minimizing operational costs. Having this in mind, new technologies present huge potential in improving the way supply chains are managed and how disputes are solved within it.

One of the most promising technologies that can assist in managing supply chains more successfully is the distributed ledger technology and more specifically *blockchain*. Having in mind blockchain’s design, it is believed that it can improve transparency, traceability and reduce administrative costs in businesses, therefore it is also applicable inside supply chains. Moreover, by providing a central distributed database which can act as a source of truth, it can assist dispute resolution processes and enhance trust between parties.
Finally, because of the lack of trust, organizations are reticent with sharing information between them as it might give other organizations a competitive advantage over them. For this reason, the privacy and security of information systems and supply chain processes acquires a higher value. This leads us to the idea of private and permissioned blockchains that can have a perfect fit to the current problem.

1.3 Objectives

The main topic of this dissertation represents the applicability study of permissioned blockchains, namely Hyperledger Fabric to supply chain management and more specifically to the dispute resolution process inside supply chain systems.

1.3.1 Main Objectives

The main objectives which contribute directly to the stated topic can be formulated as:

- Design and implement a proof of concept prototype representing a supply chain management system enhancing the dispute resolution process. This objective can be considered achieved when the developed application will allow every supply chain participant to perform normal operational activities as well as open disputes and provide remedies for solving them.
- Analyze the applicability of permissioned blockchains to dispute resolution processes within the supply chains.

1.3.2 Secondary Objectives

The secondary objectives represent a series of steps that are required to be taken in order to achieve the main ones:

- The study of supply chains, supply chain management and disputes within them.
- The study of blockchain technology: the functioning mode of a blockchain network, ways of reaching consensus and validating transactions, types of blockchains, security analysis of the technology, advantages and limitations.
- The study of Hyperledger Fabric: mode of operation, advantages and limitations over traditional blockchain networks.
- Set up a Hyperledger Fabric test network having supply chain participants (e.g. producer, distributor and retailer) run blockchain nodes.
- Design and write chaincode for a supply chain dispute resolution system.
- Implement a REST server for client applications, serving as a gateway to the Hyperledger Fabric network.
- Design and implement a graphical user interface that provides means to interact with the developed network and which showcases the functionalities of the system.

1.4 Dissertation Structure

The rest of this paper is structured in 7 chapters:

- Chapter 2: Related Work, that provides an insight into the current state of the art in the applicability of blockchains in supply chain and the dispute resolution process. Also, other examples are given of how blockchain and distributed ledger technology can be used to improve present technological systems.
- Chapter 3: Background, which describes the most important theoretical concepts needed in order to understand the rest of this work. In the first part, the blockchain technology is described from a more general point of view, the blockchain types and how it can be useful in our world. The second part focuses on the architecture and concepts behind the Hyperledger Fabric network, its advantages and limitations.
- Chapter 4: System Design, represents the most important part of this dissertation and describes the reasoning standing behind the made design choices, how does the system function, the functionalities that get enabled by it as well as its limitations plus some possible alternatives to certain parts of the design.

- Chapter 5: Security analysis, looks into the system design from the security point of view. It specifies what are the security implications from using this technology and this design.
- Chapter 6: Implementation, dives into the technical details relevant to implementing the system such as used libraries or important steps in setting up the Fabric network.
- Chapter 7: Validation and testing, showcases the resulted application, its use cases, how the functionalities were tested and an user guide.
- Chapter 8: Conclusion, gathers all the achieved results and gives an overview of what has been done. Ultimately it specifies how the system can be improved in potential future works.

Chapter 2

Related Work

The topic of supply chain dispute resolution systems is relatively new and there is not much scientific literature or industrial projects addressing this issue. Nevertheless, there is a multitude of papers regarding the applicability of blockchain in the supply chain and supply chain management.

For example, Feng Tian proposes in [5] an agri-food supply-chain traceability system based on RFID (Radio Frequency Identification) tags and blockchain technology.The information from RFID tags can be recorded and uploaded to the blockchain using wireless networks in different links of the supply chain such as:

- *Production* link where information relevant to the production process can be recorded for each item or batch of items. For example plants can have all the information recorded from the time they were planted to the time they got harvested, strain of the plant, when and what kind of pesticides were used, location or fertilization details. The process can be similarly done for meat products where the breed, parents, vaccination or diseases information can be recorded together with information regarding the personnel operating with it. In this way, cases of accidents or anomalies can be easily detected and responsible employees can intervene in a timely manner.

- *Processing* link uses the information provided by the production link and on top of that records valuable information such as keeping conditions, expiration date, how the products have been processed and how they should be handled consequently.

- Warehouse management link could make use of information used in RFID tags in order to manage the stock dynamically and make sure the losses due to expiration or lack of proper preserving conditions are minimal. Stored information can also be used to check the real-time environmental conditions during storage or delivery.

- *Distribution* link uses temperature and humidity sensors to record the transportation conditions and make sure it complies with the requirements stored on blockchain for the transporting items. Furthermore, GPS can be useful to improve the delivery efficiency and keep track of the products in real time.

- *Sales* link can finally benefit from all the efforts put into recording the information by previous supply chain participants. Therefore, customers are assured that the products they are buying are fresh, as sales organizations can easily manage products close to the expiration date. Another key benefit is transparency of products' supply chain journey, as customers can use RFID tag readers to obtain the history of the relevant item. Finally, accident risks can be mitigated as products can be easily localized and action would be taken immediately.

This paper is a good example of how blockchains can be used in practice in the supply chain of agro-products. Taking into account that food products are the ones with the most requirements, it can be concluded that this approach is applicable to other fields with looser requirements. As it proposes a more general approach on a real life example, we can consider it a confirmation that blockchains have real use within supply chains.

Saberi, Sara et al. in [6] discuss and review the blockchain technology adoption in the supply chains. It is stated that blockchain's ability to guarantee traceability, reliability and authenticity, basically removing partially the need of trust in the system, brings the necessity to rethink how supply chains are generally designed. It also

mentions that more often than not, a private, permissioned blockchain might be required to be used, but does not dismiss the applicability of public ones in certain scenarios. In comparison to regular supply chains, four major entities required in a blockchain supply chains are mentioned: *registrars* which issue identities, *standards organizations* responsible for defining and maintaining policies and requirements, *certifiers* that provide certificates needed in order to participate in the network and *actors* which include participant roles defined previously such as producers, distributors or retailers. One important noted thing is the need of having a mechanism for checking the permissions of different parties within the system and how parties would get into sell or buy agreements. As a possible solution, smart contracts are mentioned, that can design the business rules required in real life.
In the second part of the paper the authors turn their attention to how blockchains can make supply chains more sustainable. One angle is social sustainability in which corrupt individuals or organizations can be prevented and held accountable for their misdeeds. Also it would help assure human rights by giving customers access to product history which in turn would promote ethical production sources. Another one is the environmental one, by enhancing product quality and therefore reducing the number of returns and reworks, which in turn result in less energy consumption and emissions. Another way is by giving another meaning to ”eco” or ”green” products, whose authenticity could become verifiable. Other mentioned positive effects on sustainability are improved recycling processes, better tracking of carbon emissions and removal of intermediaries.
This article proves that the adoption of blockchains in the supply chain has various benefits not only for the involved participants but also for the whole society and encourages further research in the domain, which this dissertation is doing.

Abeyratne and Monfared propose in [7] a decentralized system with blockchain for collection, storage, and managing of most relevant information of products.
All items have unique profiles that would get populated through their life cycle. The link between the software identity of the product and the physical one is made through RFID tags or QR codes which contain cryptographic identifiers. Actors can create profiles on the network through designated parties, representing their identity and containing useful information such as location, descriptions or owned certifications. The actors of the system are similar to the ones mentioned in [6] and they interact with the system through a commonly developed software interface. The application would differ for every type of application and profile.

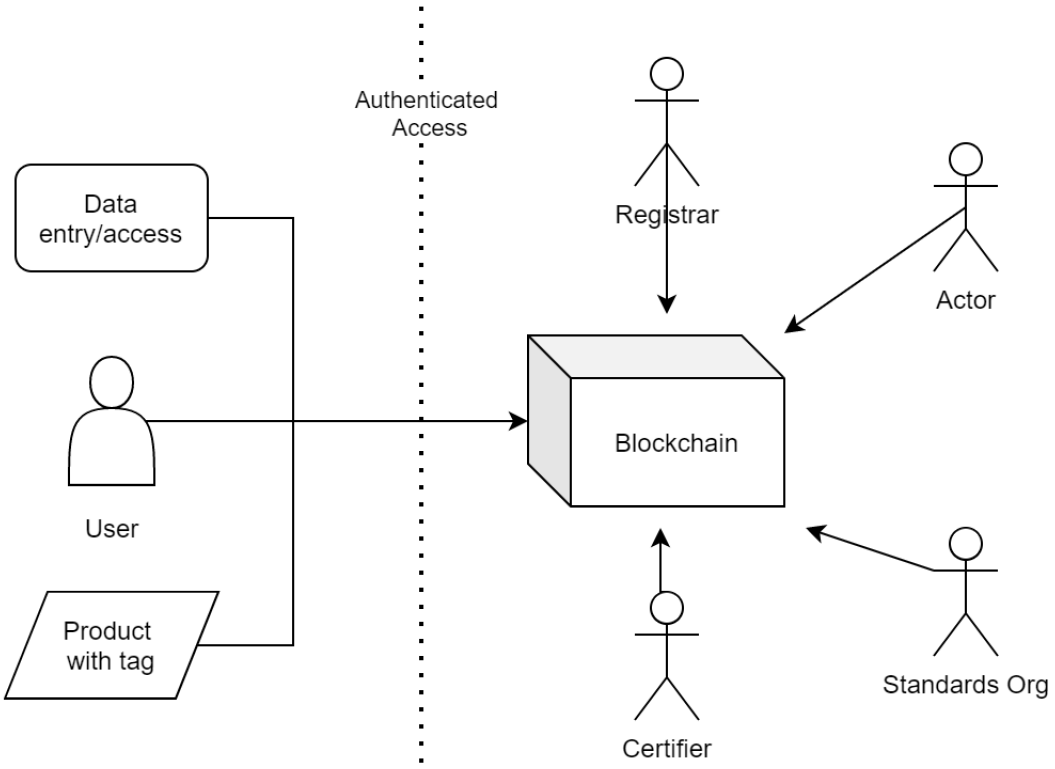

Figure 2.1: Overview of proposed concept. Adapted from [7]

The code is executed and data is stored on a a blockchain network such as Ethereum [8]. To perform any action on the ledger actors need to identify themselves with a valid set of public-private keys. It is proposed that

for each item, a set of rules would be created on the ledger as a smart contract, which decides who is allowed to access what and checks permissions for certain actions. In terms of data access, each product would have certifications stating what data is available for that identity. Therefore, a customer interested in checking the history of events would have access only to the type of events, while a participant that handles the product could access more information such as the specific identities that managed the product previously. The system overview is shown in 2.1
Even though the authors provide sufficient evidence that such a system would benefit consumers, suppliers and the environment, the proposed design has also disadvantages. Because of the need to store all the information related to products, the use of the Ethereum network is unsuitable. Not only it is an unsuitable place to store huge amounts of sensitive data [9], but also code execution in the network is too costly for the system to be able to scale in a real world scenario.

In [10], Casado-Vara et al. proposes an improvement to the current agriculture supply chain with the addition of blockchain technology. The main noted benefits added by this technology are the availability of data from the beginning to the end of the chain as well as higher security in the transactions. Authors state that by incorporating this technology, a circular type of economy gets enabled in which consumers and producers are linked through the recycling process.

In another paper, Caro et al. [11] present AgriBlockIoT - a blockchain based decentralized traceability solution for the supply chain management of agro-food. In their solution, the following actors are identified: *providers* which provide raw materials, *producers* that produces the food, *processors* whose responsibility varies from packaging to complex actions, *distributors* responsible for the delivery of the product, *retailers* which ultimately sell the products directly to the *customer* - the last actor. The system is composed of the following modules:

- *Blockchain* representing the main component and which implements the business logic through smart contracts. Depending on the needs and desired complexity, the actual implementation of the blockchain can vary.
- Controller which has the job of a middleman between the blockchain and upper levels acting as an adapter between low-level and high-level calls.
- REST API having the role of exposing the system functionalities in a easy and accessible way.

The main use-case of AgriBlockIoT is to offer consumers complete information about the products they are purchasing, with the requirement that all actors and IoT devices are registered on the blockchain with a public-private key pair, and that all product related activities are recorded on the ledger.
The performance of the system has been assessed with two blockchain implementations: Hyperledger Sawtooth and Ethereum. The results show clearly that Hyperledger Sawtooth has much lower latency, transaction size or CPU load. This can be further taken into account when designing systems with a realistic applicability that needs to scale.

The authors of [12], try to analyze on a theoretical level how the adoption of blockchain can influence the supply chain management process. The paper concludes that one of the biggest benefits would be the reduction in costs for supply chain operations, as it reduces the time needed to look for reputable partners thanks to the records regarding their reputation. Also by implementing business contracts in logic, some of the processes would become automated reducing needed resources to be put in activities related to it. At the same time, this technology may have a disruptive effect on the established supply chain and the relations and governing within. These findings therefore motivate further research in the domain and makes the blockchain adoption in supply chain more promising.

A blockchain based tracking system is proposed in [13]. Just like in previously mentioned articles, the data is shared between participants using the ledger network. Smart contracts are interpreted as state machines responsible for tracking the status changes of goods. Moreover, interested stakeholders can track or react to specific state changes depending on the terms of agreement stated by the smart contracts. In this way the information from the supply chain is delivered to the participants through a push mechanism rather than pull as it currently is. This would reduce the needs to keep backlogs and result in lower operation costs. It also concludes that an on-chain payment method would be beneficial and reduce payment lead time.

Regarding the dispute resolution process within the supply chain, the available literature uses blockchain as a way of storing information and a mediation point that does not take decisions about who is right or wrong, but rather offers means to help solve the issues.
For example the authors in [14], introduce a blockchain-based solution for cross-border e-commerce supply-chain, to solve the traceability problem. The data stored on the blockchain is classified into five categories:

- Digital documents which replace paper documents, increase processing speed and reduce transaction costs.
- IoT Data which is essential in providing product traceability.
- Transaction records that are afferent to the implementation of the blockchain system.
- Traceability tags representing the digital identities of products on the ledger.
- Execution records of smart contracts, which are stored on the ledger and are used to look for solutions in case of disputes.

Big quantities of data are stored outside the ledger (e.g. IPFS) and only the hash of the uploaded data is stored on the blockchain. The most important element providing security and traceability in the system is the traceability tag, which is generated using elliptic curve cryptography, is difficult to crack or copy and ensures privacy and security. While suppliers and producers upload product information manually or through IoT devices, consumers and auditors have access to this information stored either fully or in IPFS, and can conduct checks on the history and validity of the product.
Depending on the need for read and write speed, access frequency and data encryption requirements, the data is stored on three different chains that satisfy the needs. Namely these are account chain, transaction chain and IoT chain, which operate on different levels and interact between them accordingly as shown in 2.2 [14].

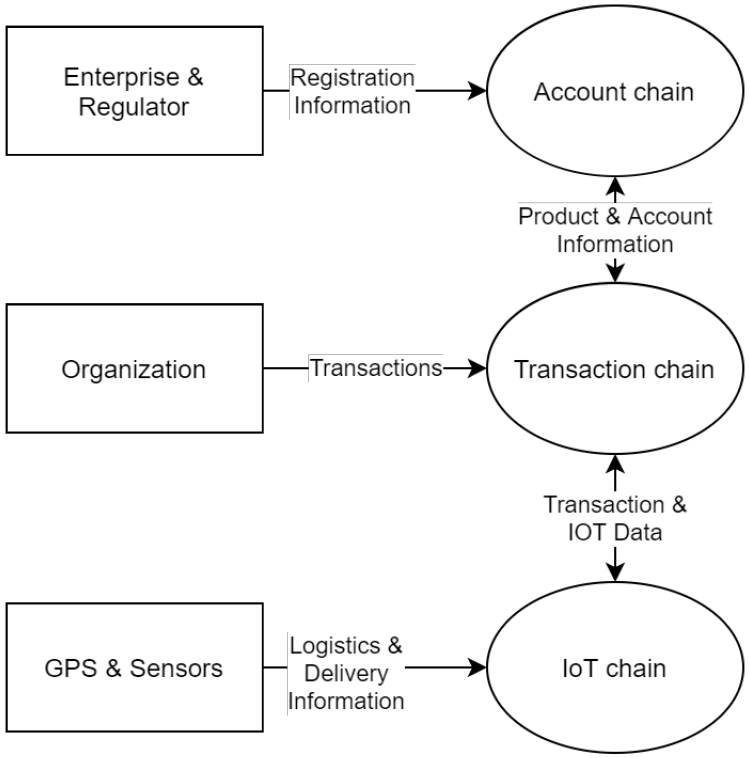

Figure 2.2: Multi-chain architecture data flow. Adapted from [14]

The biggest challenge in the adoption of such a system is the fear of using the immutable blockchain technology among participants, which would be from different countries and that do not trust each other in the first place. As mentioned previously, the the system can provide means and data for collaboration, but it can not solve disputes within it. It can offer evidence and accessible data to parties across countries but the issues still

need to be addressed in the legal and commercial system. Also, the interfaces between the developed systems and the enterprises need to be developed accordingly to the needs of each and every organization.

Wang et al. in [15] built an information management framework in the construction domain using blockchain. The system is used by project owners, contractors, off-site plants and logistic companies by serving as a mean to collect and query information during the whole process of precast ordering, starting from its production to final use. Relevant information such as digital identity, delivery truck information, quantity or check-in time are recorded on the ledger. Because the information shared is of sensitive nature and the actors know each other, a permissioned ledger is preferred over the public one. In order for the process to flow smoothly, parties need to collaborate and communicate between them, therefore the process is not fully automated. For a transaction regarding the precast to be considered valid, all parties must agree on the details to be written.
Organizations which are using the network also participate in the transaction validation process on the network level, by including business rules in it. That is, the off-site plants, logistic providers, construction providers and project owners start by proposing a transaction, after which the proposal is verified and executed by the peers of other organizations, then the transaction gets ordered by the ordering service and finally it gets validated by the peers. The clients gets notified about the outcome of this process. For the implementation part, Hyperledger Fabric 1.0 was used together with Golang as the chaincode programming language. The resulting framework does not only make the flow of information smoother between parties, but also offers a way to access traceable information at any time, therefore facilitating the dispute resolution in case of claims. It is stated that such a system can enhance supply chain management process from the following perspectives:

1. Accessible and easy way of sharing information between participants.
2. Real-time control of the process and ability to intervene at any moment during the process, thus reducing the number of disputes.
3. Traceable information, which allows for easy auditing and dispute resolution.

The authors of [16] came up with a solution that uses Ethereum blockchain and smart contracts to track and trace transactions in the agricultural soybean supply chain. The solution makes use of smart contracts, which can notify the interested parties when a specific transaction occurs. This allows for an automated way of monitoring and preventing frauds and disputes. The actors of the system vary from the seed company, through processors, to retailers and finally customers. The plants are registered on the blockchain using Global Trade Identification Numbers (GTIN), which make it easy to track information in the system. Bigger quantities of information are hashed and only the hash is stored on the ledger. Such an example are the images of the growth of the plants which are stamped and uploaded on IPFS. The farmer can either be responsible for uploading accurate video materials, or secured video cameras can also be installed that track the growth of plants, allowing for easy verification or dispute opening. Even though the system checks for validity of input data, it can not check if the actors inputting it are honest. To make up for this fact, smart contracts can have functionalities to invalidate shipments or flows, once fraudulent data is eventually discovered. This will allow new correct data to be linked to untrue one, thus ensuring correct information auditability and traceability.

A real-time supply-chain architecture for project deliveries is built in [17]. By using RFID, QR codes and IoT devices, the real-time characteristic of the system is ensured. Each transaction is linked to a generated ID, user ID, transaction timestamp and location of the device. The information is inputted to the system manually using either the web or mobile application. Also implemented with Ethereum, the system's performance has a ceiling because of the maximum number of possible transactions per second. Because of these reasons, the operational use of the system is more similar to the Observe - Orient - Decide - Action loop, meaning that the management of the system checks the normal functioning of it, and intervenes only in case of necessity.

The paper [18] develops a system prototype that can track back a product to its source by using the transactions log associated with it. It works as an event-based system in which every transaction is verified before being validated.
The supply-chain participants are grouped into suppliers, manufacturers, distributors and retailers, which all can have the demand or supply function. There is an additional regulatory department that monitors the activity in the system and acts in case of necessity. The system logic consists of three smart contracts: for registering new products in the system, to add and manage batches of previously added products and to add to the ledger transactions relevant for a batch. To achieve traceability of goods, another level of distributed ledger is implemented

on top of the blockchain architecture within the third smart contract. More specifically, validated transactions related to the batch are linked together using the same chaining mechanism (e.g. linking with the hash of the previous block/transaction) and they serve as a source of truth when verifying new transactions. The agreement process in a purchase transaction consists of the following steps and events shown in Figure 2.3 [18]:

1. Buyer purchases goods and the system triggers the Buy event with the relevant necessary information (i.e. addresses, batch and product information and signatures).
2. Seller verifies validity of signature in the events and responds accordingly with Respond event.
3. Seller sends goods to buyer and system triggers a Send event which contains the important information.
4. Buyer signals the receipt of the shipment by triggering the Receive event.

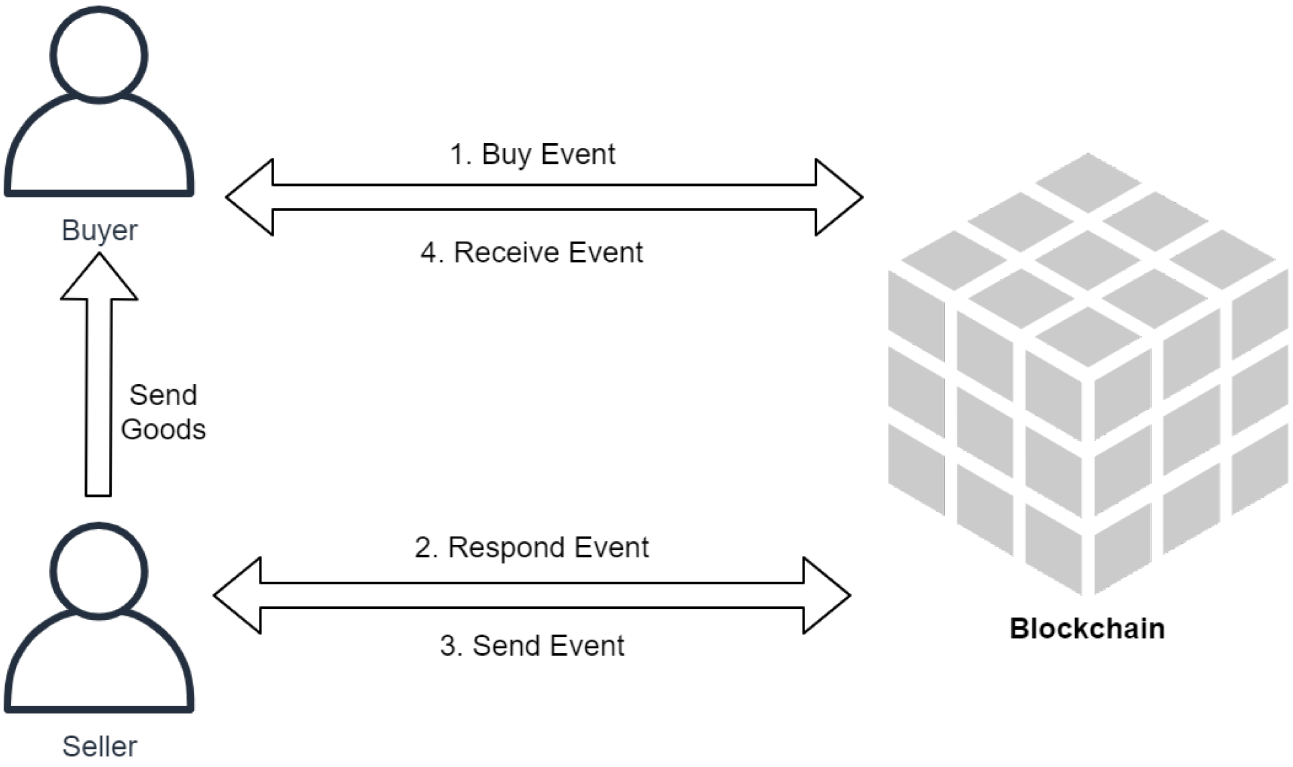

Figure 2.3: Buy process event responses. Adapted from [18]

The prototype has been built on truffle framework using Ethereum test network and from a security perspective is characterized by data accessibility and immutability, system autonomy and resistance to man-in-the-middle attacks by using signature schemes.

One similar piece of work is [19]. The author analyzes and builds a system with Hyperledger Fabric and Composer, that is able to allow users create and manage products and shipments in the supply chain, query and audit the transactions written on the ledger, submit complaints and update status of the products. In order to facilitate payments between parties, a cryptocurrency is introduced to the system. Among the normal users of the system (i.e. supplier, manufacturer, distributor, retailer, customer), there are also regulatory users able to audit and observe the normal functioning of the system and admins that have close to full control over the information on the ledger. The assets that the system operates with are:

- Commodity - representing a single product with all the relevant necessary information about it.
- ShipmentBatch - representing a physical shipment from a seller, containing commodities, delivery information and is associated to a contract.
- OrderContract - representing the digital buying contract associated to a ShipmentBatch. It holds information regarding the agreement between parties and conditions that need to be met. Based on this information, fraud and mistakes in the system can be detected.

In order to expose the functionalities of the blockchain system to the outer environment, a REST API was developed on top of the network. The resulting system proves that a feasible supply chain management system can be implemented to improve security, traceability and operational process. Nevertheless, the system does not address the disputes that can arise between parties and only mentions that the design can assist in the dispute resolution process.

Judging from the presented scientific literature, we can conclude that supply chain management systems are possible to implement in various domains and they would bring clear added value to the current ones. Traceability and security are among the main advantages that these implementations bring. Even though the mentioned designs try to reduce the number of disputes or aid by providing a shared way of accessing information, they do not go in details into the process of resolving issues. Therefore, in this dissertation we are going to attempt to contribute to the exploration of dispute resolution process within blockchain supply chains.

Chapter 3

Background

This chapter will briefly describe the main technologies required to understand the content of this dissertation. First it will start by describing the blockchain technology from a wider perspective, history and types of blockchains. In the second part we will dive a bit deeper into the permissioned Hyperledger Fabric network and the specifics about it.

3.1 Blockchain Technology

The beginning of blockchain technology is marked by the introduction of Bitcoin by notoriously famous, but still unknown group or individual Satoshi Nakamoto in [20], where a peer-to-peer electronic cash system is proposed, which allows sending payments between parties without a central authority. Several years later, Bitcoin represents a multibillion dollar market with a multitude of varieties and adaptations of the original network.
Currently the world relies on third party actors whose main role is to offer security, provide confirmation of our online actions or certify identities. The presence of these third parties in every online process makes systems more complex, opening more possibilities for security vulnerabilities, fraud, corruption and manipulation. Blockchain has the advantage of being able to remove these limitations by introducing a distributed consensus. Moreover it does so by preserving the privacy and anonymity of involved parties. With such advantages at hand, the current way of how things work can be challenged, putting a new system in place with fundamental, game-changing differences. This potential outweighs all the complexity, regulatory and technical challenges that come with this technology.

3.1.1 Blockchain Architecture

A blockchain essentially represents a distributed database containing records, transactions or events information that are shared among all the participants of the network [21].

Blocks

Every block has only one parent block, to which it is linked through a hash. The only exception is the first block recorded on the ledger, which has no parent and is called *genesis block*. An example of a blockchain architecture corresponding to Hyperledger can be seen in Figure 3.1.

A block consists of three sections [22]:

1. **Block Header** consisting of the following fields which are cryptographically derived internally:

    - *Block Number* representing the sequence number of the current block since the creation of the genesis one, which has the 0 index.
    - *Block Hash*, which is made of all the contained transactions from a block, hashed together.
    - *Previous Block Hash*, which points to the anterior block header hash. In this way blocks are linked between them.

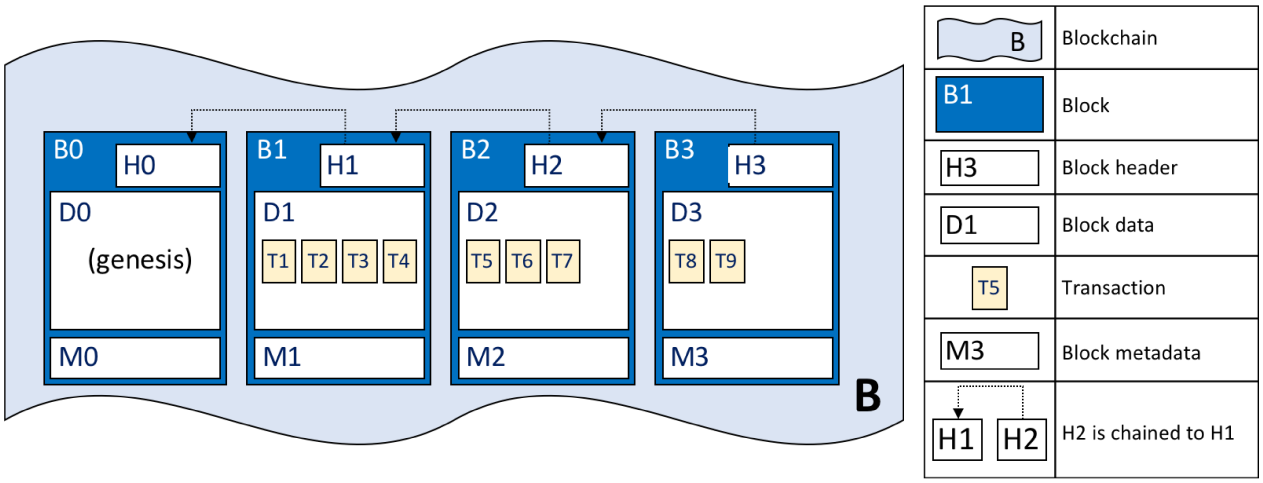

Figure 3.1: Sequence of blocks in a blockchain. Source: [22]

2. **Block Data** containing a bunch of ordered transactions.

3. **Block Metadata**, containing identity information of the block creator and used in the process of validating the block by commiters. This field is not included in the hashing process.

Digital Signatures

Digital signatures are at the base of identities and privacy in blockchain networks. Users have a pair of cryptographically secure private and public keys, which are related between them through modular arithmetic properties. The keys belong to digital signature cryptosystems such as ECDSA, RSA or DSA. The private key, as the name suggests, must be kept private and represents a person's identity on the blockchain. On the other hand, the public key is shared to the network for everyone to see. Transactions are signed with the private key before they get broadcast to the network. The process of digital signature is usually composed of two steps [23]:

- Signing phase in which the person that wants to send the message uses his private key to encrypt the message.

- Verification phase in which the recipient of the message uses the sender's public key in the decryption process. Based on modular arithmetic, if the public key is related to the private key, the decryption process is successful, otherwise it is impossible to recover the message. In this way, one can sign messages in a way that proves that he is the sender, and that the data has not been tampered.

Main Characteristics

Generally, blockchains are described by the following main aspects, which also represent their strength [23]:

- *Decentralization* - as there is no need to have a third party entity validating and managing transactions, the system relies on the collective power of participants in the network. This not only reduces costs, but also eliminates bottlenecks, enhances trust in the system and makes it more secure by removing the single point of failure.

- *Auditability* - as the network represents a chain of blocks of transactions, the validity of every input and output can be verified by consulting the history of transactions. This greatly reduces the possibility of fraud and data manipulation.

- *Security* - thanks to the Public Key Infrastructure, transaction data on blockchains is encrypted, removing the need for having a gate-keeper.

- *Persistency* - once a valid record is added to the ledger, it is nearly impossible to remove it or to add an invalid record. This property lies in the nature of the system of data redundancy. By having the necessity to store the ledger on every network node, the system becomes highly secured.
- *Anonimity* - depending on the type of network, users can interact with the blockchain only by having a pair of valid keys and without providing any information about their real identity.
- *Consensus* - one key element making blockchains so trusted are consensus algorithms, which help in the decision making process within the network. This is responsible for making the blockchain trustless, meaning that there will always be a single truth among all the participants that do not know or trust each other.

Blockchain Taxonomy

Since the creation of Bitcoin, blockchain technology has been under various transformations. This is because different systems and architectures are suitable for different use cases. There is no silver-bullet blockchain type that is suitable to any kind of problem. That is why it is important to know the differences between them and their characteristics. According to [24], blockchains can generally be attributed to one of the following groups:

- *Public Blockchain* - this type of blockchain is permission-less, meaning that anyone with an internet connection can take part in the network and submit transactions. In this version, each peer has a copy of the ledger and participating nodes are responsible for keeping it working by validating transactions and reaching consensus. This is one of the most popular types of blockchains, mainly because the first created blockchains are public and because it has a wide adoption. Among its advantages are:
  - Requires no intermediaries for it to work.
  - The bigger the network, the more secure they become.
  - Can be joined by anyone and brings trust among participants that do not trust each other.
  - Anyone can verify the data.

  Despite their wide adoption by users, this type of blockchain is not suitable for all use cases, namely because of the following limitations:
  - Transaction speed: the strength in numbers of this network comes with this drawback. This is mainly because it takes a lot of time to reach consensus and propagate a decision between huge number of users. This makes public blockchains unscalable, as they become slower with time, making them unsuitable for many current business applications.
  - The fact that the network is public can come as an disadvantage in cases where parties want to engage in private interactions or when they do not want to disclose certain information to the public.
- *Private Blockchain* - these types of networks are the ones functioning in environments with certain restrictions, usually under the control of an entity. In this type of network, in order to be part of it, a trusted entity needs to give such permission. They are also considered partly centralized, as the controlling entity monitors and gives permissions to users, while the participating users maintain the decentralized nature.

  The advantages are:
  - Transaction throughput - because of the smaller size of the network and simplified way of reaching consensus, the number of possible transaction for a time interval increases considerably. This makes it suitable for use cases where scalability is an important factor.
  - Privacy - as the access to the network is permissioned, this type of blockchain becomes suitable for more private interactions such as business agreements that should not be disclosed to non-members.

  Respectively, the disadvantages are:

  - Network is not fully decentralized anymore, which goes against the definition of blockchain and distributed ledger technology.
  - Lower security by having a smaller number of running nodes.
  - Trust in the network has a changed definition as there is a central authority.

- *Consortium Blockchain* - this type of network has some features of the public and private one. It has a decentralized nature because there are predefined nodes that take part in the decision process, and it has a private nature because access to the network is permissioned. It can be looked at as a private blockchain that shares the decision making process between multiple entities. Their strength lies in:

  - More customizable than its predecessors.
  - More efficient than public blockchains.
  - Suitable for cases in which the ruling entities are well-defined and do not change.

  The weaknesses are:

  - Reduced transparency in the decision process.
  - Less secure due to the ruling entities integrity.
  - Less anonymous than other types.

- *Hybrid Blockchain* - this type represents a combination of public and private blockchains. It can have permissioned or permission-less access at the same time. Some of the data can be kept private while the other can go public. An example can be a private blockchain that consists of more public ones or that have connections to public ones.

  It has the following strong points:

  - Customizable depending on the needs.
  - Immune to 51% attacks.
  - Better scalability than the public network.
  - Works in a closed environment keeping public properties.

  while the weak points are:

  - More complex to create.
  - Lack of incentive to participate and maintain the network.
  - Less transparent.

3.2 Hyperledger Fabric

Hyperledger Fabric is a free and open-source distributed ledger platform. The participants in the network are known to each other, unlike in the public distributed ledgers, meaning that there should be some kind of trust mechanisms established between them in order to have some kind of authority in the system. This makes the platform best suited and meant to be used at enterprise level between partners, competitors and regulators [25].

3.2.1 Traditional distributed ledgers architecture

Usually distributed ledgers follow an order-execute architecture which means that initially the nodes agree on an order of the transactions through a consensus protocol such as Proof of Work, followed by the execution of these transactions on all the nodes in the network. This architecture is shown in Figure 3.2.

This traditional architecture is fairly easy to understand and it achieves the desired results, making it vastly adopted in different blockchain networks. Nevertheless, it suffers from a series of limitations and drawbacks such as [26]:

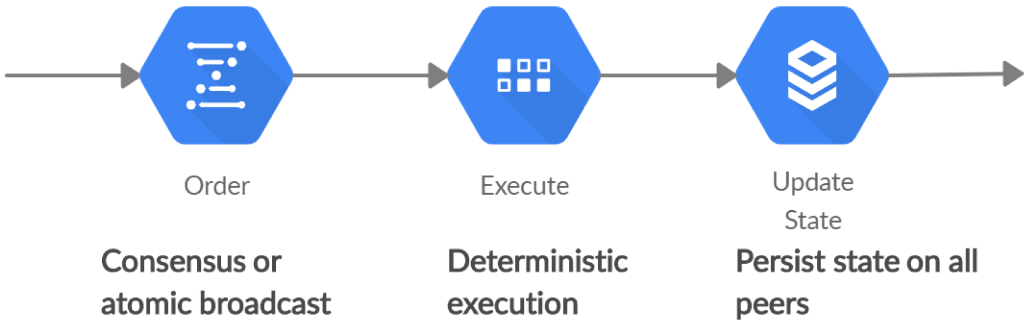

Figure 3.2: Order-Execute architecture

- *Limited throughput*: because of the sequential execution of transactions on all the nodes, the resulting throughput is affected, thus increasing latency and making the network susceptible to denial of service attacks. This can happen because the blockchain acts like a distributed computing machine, and a smart contract whose execution does not terminate can make the whole network unavailable. In order to overcome this threat, some distributed ledgers adopt different limiting mechanisms (e.g. gas for Ethereum) which might become unsuitable for permissioned ledgers.

- *Domain-specific languages*: because the structure and content of the blockchain should be kept consistent among all the nodes, the code executed on it must be deterministic in order to avoid network forking. The most popular blockchain networks achieve this by introducing domain-specific programming languages, which represent another inconvenience for the programmer and influences negatively the adoption rate of the technology. On the other hand, integrating blockchain with general-purpose languages can be a real challenge as some of them are not always deterministic, which can potentially lead to the inconsistency of the whole network.

- *Lack of confidentiality*: this limitation is a consequence of one of the strengths of many permissionless and public networks - transparency. Because the code has to be executed on all the nodes, peers have access to smart contract logic, execution parameters and data. This might pose a limitation for many businesses that can not and do not wish to expose confidential business details. Even though there are techniques such as encryption to keep the data confidential, they significantly increase the overhead making them unsuitable for practical cases.

- Another limitation of architectures that currently exist is *fixed consensus protocol*: prior to Hyperledger Fabric, blockchain networks allowed only one type of consensus mechanisms, which therefore reduces the flexibility and adaptability of the network to case by case business needs. It is also known that the performance of different consensus protocols differs significantly when introduced to different environments. For example BFT protocols' performances vary considerably when performing in adversarial environments.

3.2.2 Fabric architecture

Fabric is a system operating on permissioned blockchain networks which executes applications written in general purpose programming languages such as Java, NodeJs or Go and has no cryptocurrency built in. Because of the previously mentioned limitations of traditional distributed ledgers limitations, it follows an execute-order-validate architecture that can be seen in Figure 3.3

Simply put, a Fabric distributed application is composed of two entities:

- A *chaincode* which represents the smart contract that implements the logic of the application. It is executed at the execute phase, can be written by a malicious developer and represents the central part of the application.

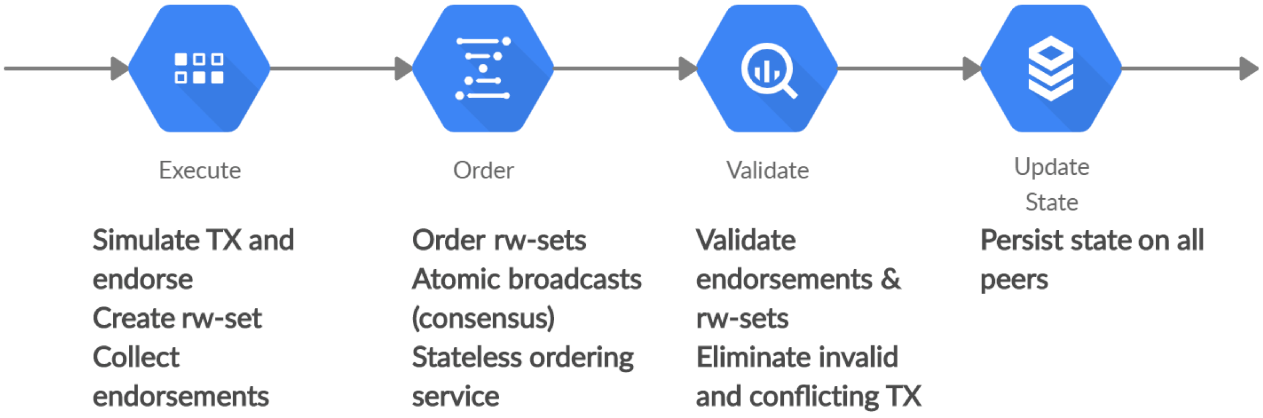

Figure 3.3: Execute-order-validate architecture

- An *endorsement policy* that represents a static library used for validating transactions in the validation phase by specifying the peers that are necessary for endorsement. It can be modified by system administrators and chaincode developers do not have access or influence over it.

Because Fabric is a permissioned network, all the peers in the network have an identity which is given to them by the membership service provider (MSP). The network is composed of the following roles:

- *Clients* which propose transactions for execution, take part in organizing the execution phase and distribute transactions to be ordered;
- *Peers* that store the blockchain ledger data which contains a chain of transaction hashes and the latest state of the ledger. Peers that are specified by the endorsement policies or chaincode are also called endorses and they have the role of execution of transaction proposals and transaction validation.
- *Orderers* that are unaware of the application state and do not get involved in the previous execution or validation phases. These nodes are responsible for creating a total order of transactions which contain ledger state changes and cryptographic signatures of the endorsers.

3.2.3 Transaction flow

The transaction flow (Figure 3.4) starts with the **execution phase** (or proposal), in which clients sign and send to the endorsers the transaction proposal to be executed (1). The proposal contains data such as client's identity, payload, parameters, transaction identifier and a nonce used to uniquely bind each client with each transaction. The endorsers execute the operation in isolation and therefore simulate the proposal against his local ledger's state (2). As a result of this, each endorser produces and sends back to the client an endorsement containing a readset which are the dependencies of the simulation to previous local states and a writeset that consists of changes made by the simulation on the state (3). After the client collects enough endorsements on a proposal, he creates the transaction and forwards it to the orderers (4).

In the **ordering phase**, a total order is established in each channel by the ordering service that groups multiple transactions into blocks and chains these blocks using their hashes (5), thus improving the throughput of the protocol. The ordering services has two high-level functionalities that can be called by the clients:

- Broadcasting a transaction containing its payload and the signature.
- Retrieving a block by the sequence number containing a list of transactions and a hash of the previous block. Because the content and sequence numbers of the blocks do not change, this operation is executed only upon first invocation.

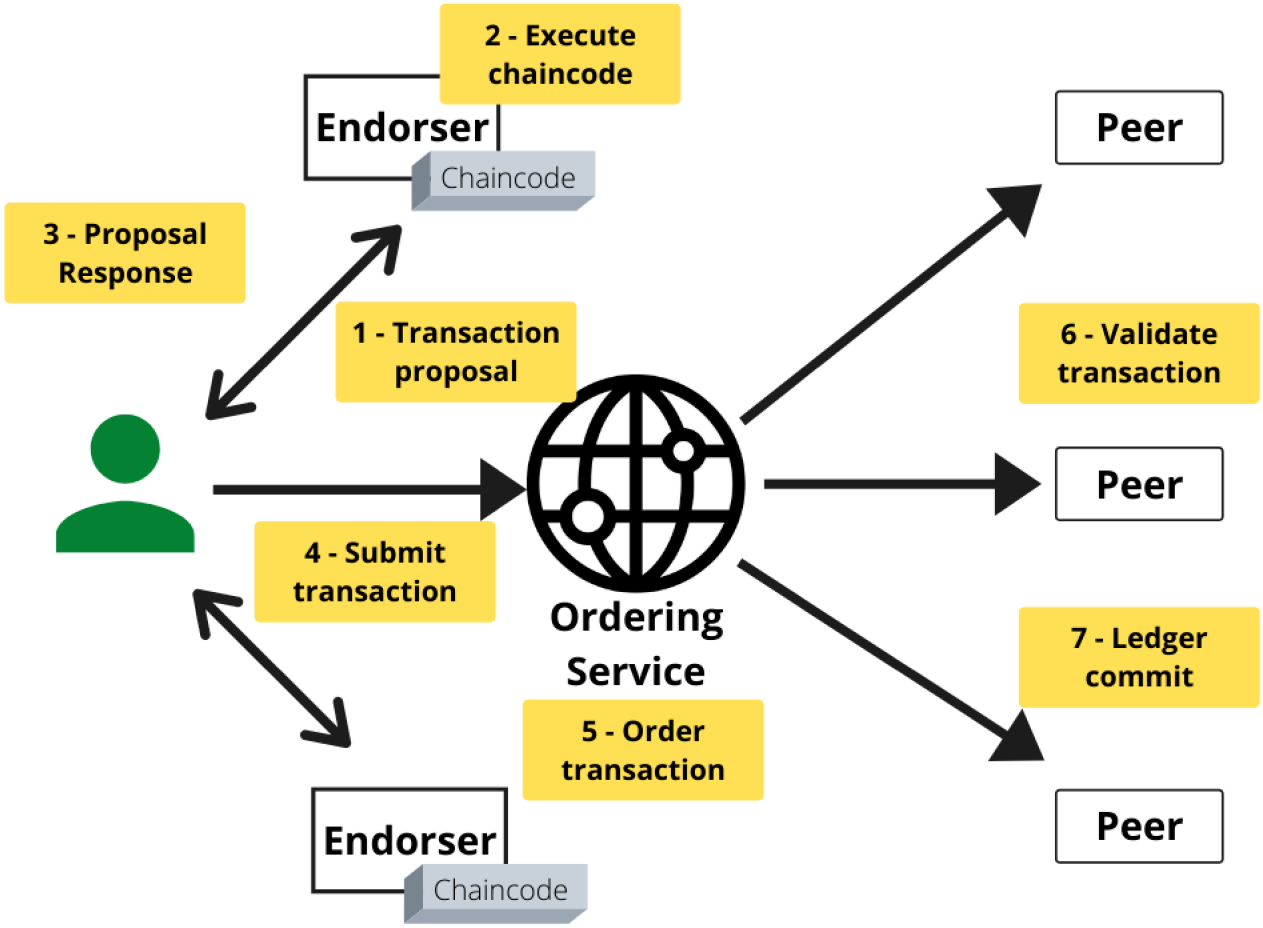

Figure 3.4: Hyperledger Fabric transaction flow

Orderers ensure that the blocks that get delivered in channels follow a set of safety properties that guarantee the well-formed state of the transaction blockchain. The fact that the ordering services do not execute transactions or maintain the state of the blockchain ensures the separation of consensus from validation and execution. This leads to a highly modular design of Fabric, making the use of different consensus protocols possible in the same network.

Before entering the **validation phase**, a block has to be forwarded to the peers by the orderers or through gossip dissemination protocol. At this stage, the peers check the transactions in parallel using the validation system chaincode (6). Transactions that do not meet the endorsement policy rules of the channel are deemed invalid, after which a read-write conflict check is performed in sequential order to make sure that for each transaction the readset corresponds to the actual state of the ledger locally stored (6), otherwise the transaction is flagged as invalid and its effects are nullified.

After all the checks have been performed, the ledger is updated (7) by appending the new block to the ledger and updating its state from the block writeset.

3.2.4 Hyperledger Fabric Components

Membership Service Provider

On a permissioned blockchain network, all the operations, messages and interactions between members have to be authenticated, usually through digital signatures. For this, members need a way to prove their identity on the network. The membership service provider (MSP) is responsible for issuing and maintaining credentials for the nodes, which can be used for authorization and authentication. This is done through Root Certificate Authorities (RCA) and Intermediate Certificate Authorities (ICA). The MSP consists of a component at each node, where it is in charge of signing and validating endorsements, authentication of operations, verification of

the integrity and authentication of transactions.

In practice, MSPs are implemented as a set of folders added to the network configuration which determine who the organization's admins are, the permissioned entities and which other organizations have the right to validate that entities are authorized to do a specific action [27]. It is the MSP that assigns roles to simple identities by specifying the privileges the identity has in a channel or a node. There are two types of MSPs:

- *Local MSPs* that are bound to peers, orderers or clients and they state what permissions do users have on that specific node. All the nodes need to define a local MSP in order to define the admins and participants at a level. This type of MSPs exist only on the file system of the nodes.
- *Channel MSPs* which function at the channel level. Similar to local MSPs, the channel ones specify the identities that take part in a channel (which are MSPs themselves) and the possible actions associated with them. All channel members have the same channel MSP which they consult to decide if a transaction is authorized or should be ignored. In order for a node to be part of a channel, the organization's MSP states that its trusted members should be included in the channel configuration. Moreover, the channel MSP includes all the MSPs related to organizations that specify the orderers that take part in running the ordering service. Even though conceptually a channel MSP is one for all the participants, in practice every node has a copy of it and it is being synchronized and updated through consensus.

Whether we are talking about multinational corporations or small start-ups, the MSP allows employees' identities to be attributed to the specific organization. Because of the one to one relation of MSPs and organizations, it is usually the case that they are named the same. In cases which an organization has multiple departments and membership groups that are naturally separated, multiple MSPs can be created to outline these differences. An example is shown in 3.5

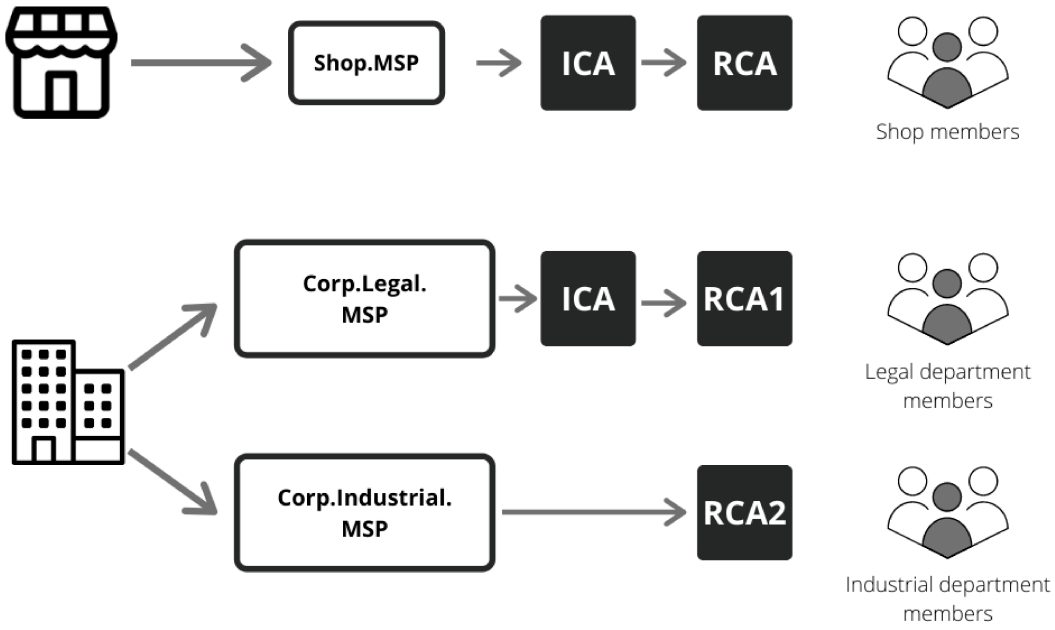

Figure 3.5: Membership service provider (MSP) hierarchy diagram. ICA/RCA - Intermediate/Root Certificate Authority

Ordering Service

While public distributed ledgers offer the possibility to participate in the probabilistic consensus process, permissioned ledgers work differently. Fabric introduces a new type of node called ordering node or just orderer whose role is to perform atomic broadcast to establish order on transactions. Orderers inject transactions for broadcasting and during the batching process of transactions, a new block is formed when one of the following conditions gets fulfilled:

- The maximum number of transactions in the block has been reached;

- The maximum size of the block has been reached;
- A predefined amount of time has passed since the last formed block.

All the ordering nodes together form the ordering service and because it achieves consensus in a deterministic way, any block that gets validated is correct and final. This ensures the impossibility of forks to happen and separates the consensus level from chaincode execution, increasing the performance and transaction throughput in the network [28].

Besides ordering, another responsibility of the orderers is to maintain which organizations have the right to channel creation. Also they enforce access control at the channel level by allowing or forbidding reading or writing data or channel configurations. It makes these decisions based on policies set at the creation of the channel or consortium. When an orderer received a configuration request, it checks if the requestor has the necessary rights to perform the action, validates it based on the current configuration and packs the update in a block. The block then gets distributed to all the peers on the channel which then verify if the approved modifications follow the channel defined policies.

Peers

Another type of nodes are peer nodes, which mainly form the blockchain network by hosting ledgers and smart contracts, also called chaincodes. Every peer keeps a copy of the ledgers and the chaincodes, which even though increases redundancy, it represents the distributed nature of the blockchain and avoids single point of failure vulnerabilities.

Even though it is theoretically possible that peers hold ledgers without chaincodes, practically every peer needs a chaincode associated with the ledger in order to be able to read, write and interact with it.

In order to access the ledger and chaincodes, applications have to connect to peers using the Fabric Software Development Kit (SDK) [29]. Applications can either query or update the ledger and the required steps for these actions differ in complexity. The necessary steps can be seen in Figure 3.6

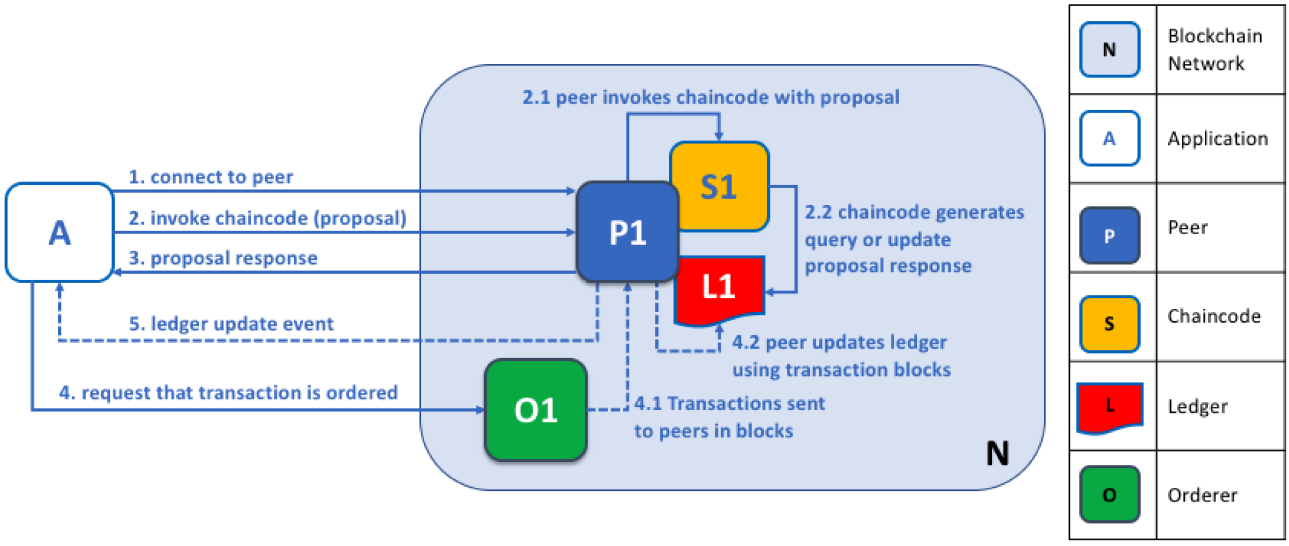

Figure 3.6: Peer-query (1-3) and Peer-update (1-5) operation steps [29]

In order to query the ledger, an application should connect with a peer (1), invoke the chaincode (2) and receive the proposal response (3). As all the information required for the query is contained in peer's local copy of the ledger, it does not need to consult with others and the response can be sent instantly.

To update the ledger, additional to the 3 mentioned steps before, the process involves 2 more steps. In order to perform the update operation, a consensus needs to be reached first among the peers. This is achieved by peers sending back to the application signed proposed updates which then all get sent to the orderer to get packaged into blocks and be distributed to the entire network of peers (4). Peers then verify the validity of the change before applying it to the local ledger and the application is notified asynchronously (5).

Ledger

From the logical point of view, in Hyperledger Fabric there is only one ledger while in reality, every peer stores and maintains a copy of the ledger through the consensus process, hence the term Distributed Ledger Technology. The ledger is used to store current and previous information about the business objects, and it consists mainly of two components: a world state and a blockchain [22].

In order to keep information about the current state of the objects ready for use, the *world state* holds all the current attributes of business objects as (key, value) pairs. In this way, if an application wants to query the ledger for certain information, instead of traversing the whole blockchain to get the updated information, it can read it straight from the world state. In practice, the world state acts like a database that through the ledger API offers get, put and delete functionality as well as the ability to perform more advanced queries. After every valid transaction on the ledger (i.e. endorsed with enough signatures), the world state gets updated and the application is notified. Along with the (key, value) pair, a version number is also stored to keep track of how many times the attribute has been changed. Before updating the present state, the version is checked to make sure it matches with the one at the time of endorsement.

The other component is the *blockchain*, and in comparison to the world state, the blockchain holds all transactions ever recorded and shows how the world state ended up in the current state. Contrary to how the world state is implemented, the blockchain represents a file with interlinked sequential blocks as the majority of operations performed on it are append and sometimes query. Each block represents a set of sequential transactions ordered by the ordering service, which either query or update the world state. While the world state gets affected only by valid transactions, the blockchain stores both valid and invalid transactions. The transactions are linked to the created block by including the cryptographic hash of the transactions inside the block header along with the hash of the previous block, therefore linking the block to the already existing chain. In this way, the blockchain has an append-only nature, making it practically impossible to tamper with the previously written blocks once they have been added to the chain.

Chaincode and Smart Contracts

While the ledger serves as the storage of Hyperledger Fabric, the chaincode represents the executable logic associated with it. The business model rules that model the interactions between parties are dictated in the blockchain by smart contracts. While the terms chaincode and smart contract are sometimes used interchangeably, there is a slight difference between them. The smart contracts define the transaction logic specific to the business model, but the chaincode represents the way these smart contracts are implemented and deployed to the network. The term chaincode is relevant mostly to those that administer the network, while everyone else can view the blockchain logic in terms of smart contracts. Smart contracts can belong to the same or different chaincodes. Because contracts can invoke functionalities of other contracts, there is a slight difference when they belong to the same chaincode or not. Contracts packed in the same chaincode share the same world state (readset and writeset), therefore it is recommended to group smart contracts in a chaincode based on application logic and endorsement policy.

On the basic level, most of the smart contracts perform put, get and delete transactions on the world state, which immutably get saved on the blockchain. Associated to every smart contract is an endorsement policy, which defines how many and which organizations in the network are required to sign a contract transaction in order for it to be considered valid. Smart contracts implement the application logic in an *application chaincode*, which is closely tied to the real business world. Additionally, Fabric provides *system chaincodes* or *lifecycle chaincodes* that are domain independent and offer low level functionalities to interact with the ledger. Among these functionalities are: handling changes to system configurations, querying blocks and transactions, signing transaction responses, installing chaincodes on peers or validating transactions [30]. The interaction with the system chaincodes can be done through command line interfaces (CLI) or software development kits (SDK).

The Fabric **chaincode lifecycle** is a series of steps that organizations follow in order to install, upgrade or deploy chaincodes to the channel. Before a chaincode can be ready to use, organizations must agree on parameters such as chaincode version, name or endorsement policy. The agreement is achivied using the following steps [31]:

1. *Package the chaincode*: each organization needs to package the smart contracts in a *.tar* file containing

chaincode files and metadata that specifies the path to the code, programming language (Go, Java or JavaScript) and a short label that would identify it.

2. *Install chaincode on peers*: all the peers that are going to run the chaincode need to install the package using the Peer Administrator. In order to have the exact same package on all the peers, a good practice is to package the chaincode once on one peer and distribute it to other peers for it to get installed. The install command will issue a package identifier formed of the label and a package hash.

3. *Approve chaincode definitions*: organizations need to approve a chaincode definition by voting that it accepts the parameters is is composed of. This process needs to be done by at least one peer per organization. These parameters include:

    - Package identifier which can be different for different organizations. This parameter can also be missing in case the organization does not need to use the chaincode but still is required to approve the definition;
    - Name of the chaincode;
    - Version number associated with the package;
    - Sequence number indicating the number of times a chaincode has been upgraded;
    - Endorsement policy specifying who needs to sign transactions from this chaincode in order to be considered valid;
    - Collection Configuration file associated with the chaincode;
    - Whether the *Initialization* function is required to be called before any other function;
    - Optionally a custom validation or endorsement plugin to be used.

4. *Submit chaincode*: after enough members approved the chaincode definition, one Organization Administrator from the ordering service can commit it to the channel. Before the definition is sent to the ordering services, it gets sent to the peers in the channel which endorse it depending on whether their organization has approved it or not. The number of organizations that need to approve the definition before being committed is defined in the LifecycleEndorsement policy. This policy is different from the chaincode endorsement policy because it defines who needs to endorse the committing transaction, while the latter defines who needs to endorse the chaincode transactions in order for them to be valid. After this step, the channel members can invoke the Init() method to instantiate the chaincode if it requires so, and start using the newly installed chaincode.

The lifecycle is illustrated in Figure 3.7.

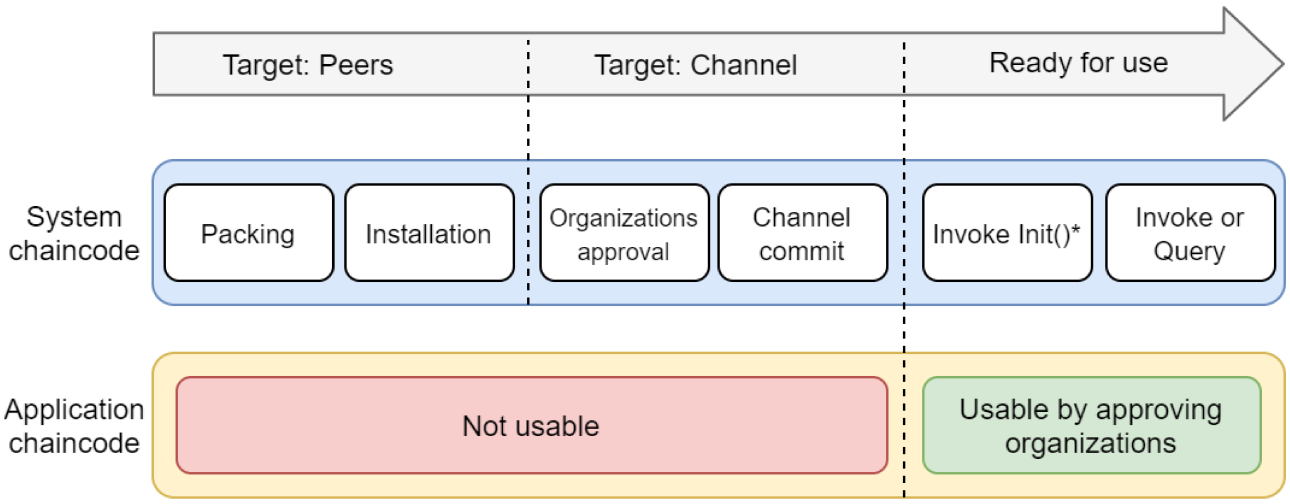

Figure 3.7: Chaincode lifecycle relative to the system and application chaincodes

Chapter 4

System Design

In this chapter we will dive into the system design process and will try to explain the design choices that were made. This will be done by describing each component individually, their roles and challenges associated with them. We will start by describing the most important element of the system, which is the Hyperledger chaincode, followed by the REST server and finally we will talk about the front-end application.

4.1 Requirements specification

This project is developed as a proof of concept, which has the role of validating the feasibility of blockchain supply chain implementations in the supply chain management process and the dispute resolution process in it. The objective is not to prototype a product that can be used by participants, but more of an integrated common system that is able to showcase that the necessary functionalities can be achieved in practice.
Therefore, the scope of this project is the development of an automated blockchain system, together with the underlying infrastructure supporting it, in order to facilitate supply chain participants to interact between them in the supply chain. The development of individual systems, tailored for the needs of every participant is outside of the scope of this project.

4.1.1 Functional Requirements

Judging from the desired system description, we can form the following functional requirements. Members should be able to:

1. Read information about assets according to their permission levels.
2. Change and update information about themselves.
3. Create new assets in the system.
4. Place orders through the system and query the information related to the order.
5. Complete orders using the system.
6. Initiate on own assets deliveries assisted by the system.
7. Specify delivery rules that must be applied to the delivery process.
8. Check the current and past locations of assets.
9. See the history of actions related to an asset.
10. Check information and current state of deliveries created by or destined for them.
11. Participate in the delivery process of a shipment as an intermediary.
12. Dispute assets from a shipment.

13. Query the history and status of disputes.

14. Solve disputes opened for them.

4.1.2 Non-functional Requirements

The system should also be described by the following characteristics:

- Speed: the transaction throughput of the system should be high enough to allow users perform continuous tasks while being in the flow.
- Availability: the system should be able to process transactions at any moment, with almost no downtime.
- Immutability: it should be impossible to modify the contents of the ledger unless through a valid transaction.
- Scalability: it should be possible to add a big number of nodes and users without affecting the good functioning of the system.
- Usability: the source code and configuration files should be written in a clear way, with proper documentation and explanations. It should be possible to build personalized systems on top of the developed components with ease.
- Privacy: transactions and data should be possible to be kept private from other system users.

4.2 System Overview

Flow Description: The system uses a dynamic way of finding the path for a shipment and assumes organizations can physically send packages to any organization unless mentioned otherwise in the policy. Organizations buy available items from other organizations and pay for them off-chain. After the buying contract is created by the buyer organization, the seller gets notified through a Hyperledger Fabric event, after which he creates the delivery shipment attaching desired delivery rules. The way the shipment gets delivered through the supply chain is that each node is responsible for sending it to the next destination, by following the specified constraints in the policy. Each organization has a profile which contains necessary information such as its location, country or what type of organization it is. The policy for a Shipment has a fixed size as it uses two bloom filters to whitelist and blacklist organizations. When the total number of organizations increases, a bigger bloom filter is automatically used.
After the delivery shipment is created, the system filters all available organizations based on the attached policy. Out of those filtered organizations, one is chosen as the next step based on a heuristic function (in this case minimizing distance to destination and distance to the node). The system converts the next step to a QR code and the holder sends the shipment with the instruction attached to it. After its arrival, the QR code gets scanned by the receiving organization and the process is repeated until it reaches the final destination.

In a supply chain, the type and number of participants can vary greatly from the supplier of producer to the final customer. For this proof of concept system, a simplified version of supply chain is adopted. According to [1], a group can be considered a supply chain if it involves 3 or more organizations. Therefore, the main participants considered for this system are:

- Producers
- Distributors
- Retailers
- Customers

These 4 actors can generally represent the majority of supply chains. Even though the customer is also part of the supply-chain, it does not take part in the system as a full member, because it only consumes what the supply-chain has to offer, and for the sake of simplicity, we can assume that Retailers are the ones responsible for developing and maintaining applications that give identities and access to the network to other clients and end customers. In the current supply chain, the flow of good would start at the producer and end at the customer, but for the design of this system, we will assume that any organization can interact directly with any other organization, thus enabling a circular economy. This can be seen in 4.1

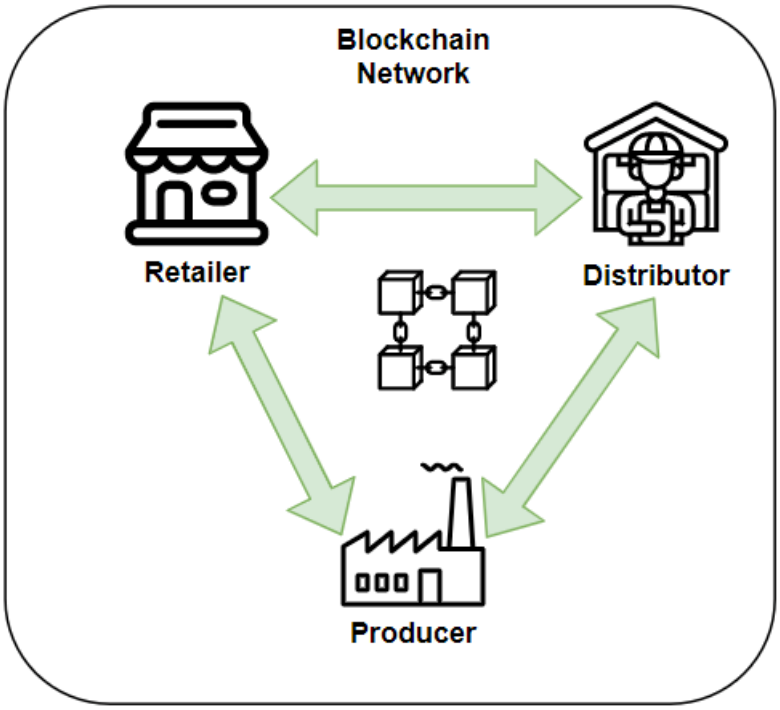

Figure 4.1: Circular interaction between system actors

The underlying technology used to build the network is Hyperledger Fabric for the following reasons:

- Because of the private and permissioned nature, it is ideal for the supply chain application as only its members should have access to the system, and the network can be run by participating nodes.
- Does not require a cryptocurrency to execute transactions, therefore reduces costs when the application scales.
- Because there is no proof of work consensus algorithm, the process of validating transactions is much faster, which is suitable for enterprise cases.
- Channels can be used to partition the data between different domains of the supply chain, and in this way an organization can be part of a channel together with his suppliers, and at the same time part of another channel with his buyers.

To keep the system simple but still make it able to capture all the functionalities in a real supply chain, assets are grouped per:

- Items: represent single physical individual assets, such as products.
- Orders: representing the buying contract of one or more items, between a seller and a buyer.
- Shipments: represent physical shipments, associated to orders and that contain items.
- Disputes: represent problems associated with items that are contained in shipments, opened by the client for the seller.

4.2.1 Network Topology

A Hyperledger Fabric network has three types of nodes:

- Clients - which are applications that need to connect through a gateway to a peer in order to communicate with the blockchain.
- Peers - that endorse and commit transactions and physically keep a copy of the blockchain.
- Orderers - which order transactions in blocks for the peers and does not need a copy of the ledger.

A Hyperledger network is formed by all organizations that provide resources (i.e. orderer and peer nodes) to its normal functioning. Because running an orderer or a peer node can not be done on a lightweight hardware device (e.g. smartphone), it is not expected for customers to run peer or orderer nodes. They will have to connect to the network using applications provided by organizations, which will make use of gateways to connect to peers. For example, the retailer's application or website will take care of identity management for its customers, and will facilitate connection to the blockchain through their peer node.
Therefore, we can conclude that the blockchain will be run by all other actors interested in taking part in the system and in the dispute resolution process and which are able to run their own nodes. These are producers, distributors and retailers. If an organization is too small to have its own peer node, it can communicate with the blockchain through a bigger organization that would provide them an identity and application, potentially at a cost.
The ordering service can either be run by a single organization that will handle the whole process and does not need to provide peer nodes or each organization can contribute with their own orderer node to participate in the ordering services. All the employees from organizations would communicate with the network through applications created by their organization, under the organization's identity. Transportation companies can also use the applications given by their customer (producers, distributors or retailers). The network topology can be seen in Figure 4.2.

In the implemented system, each organization runs a peer node, while the orderer node is a separate entity.

4.3 System Architecture

The designed system is composed of three main components, which can be seen in Figure 4.3

- *Dispute Resolution Fabric network* - this is the most important component as it represents the brain of the whole architecture. It is responsible for the system logic and is the one bringing the desired characteristics such as immutability, security and traceability.
- *REST Server* - this component is responsible for exposing the blockchain component functionalities to the outside environment. It does so by using a gateway and packing the needed methods in an Application Interface that can be used when building external applications. Without this component, anyone willing to interact with the blockchain would have to go through the cumbersome process of developing their own system that implements the connection.
- *Graphical User Interface* - this is the component that will be used by application users and that has the responsibility of presenting the system functionalities in an accessible and clear way. It connect to the REST API server through HTTPS. Without this component, the designed system would be usable only by individuals with expertise in application development and information technology.

In this section we will describe the design of each component individually.

4.3.1 Dispute Resolution Fabric Network

The logic of this component is implemented through chaincode, which consists of two smart contracts:

- *OperationsContract* - responsible for providing functionalities for normal supply chain operations. This contract implements all the logic that is being exposed to users.
- *StateContract* - responsible for providing functionalities for managing the current and past states of Items, Shipments and Disputes. The functionalities can be called only by the OperationsContract.

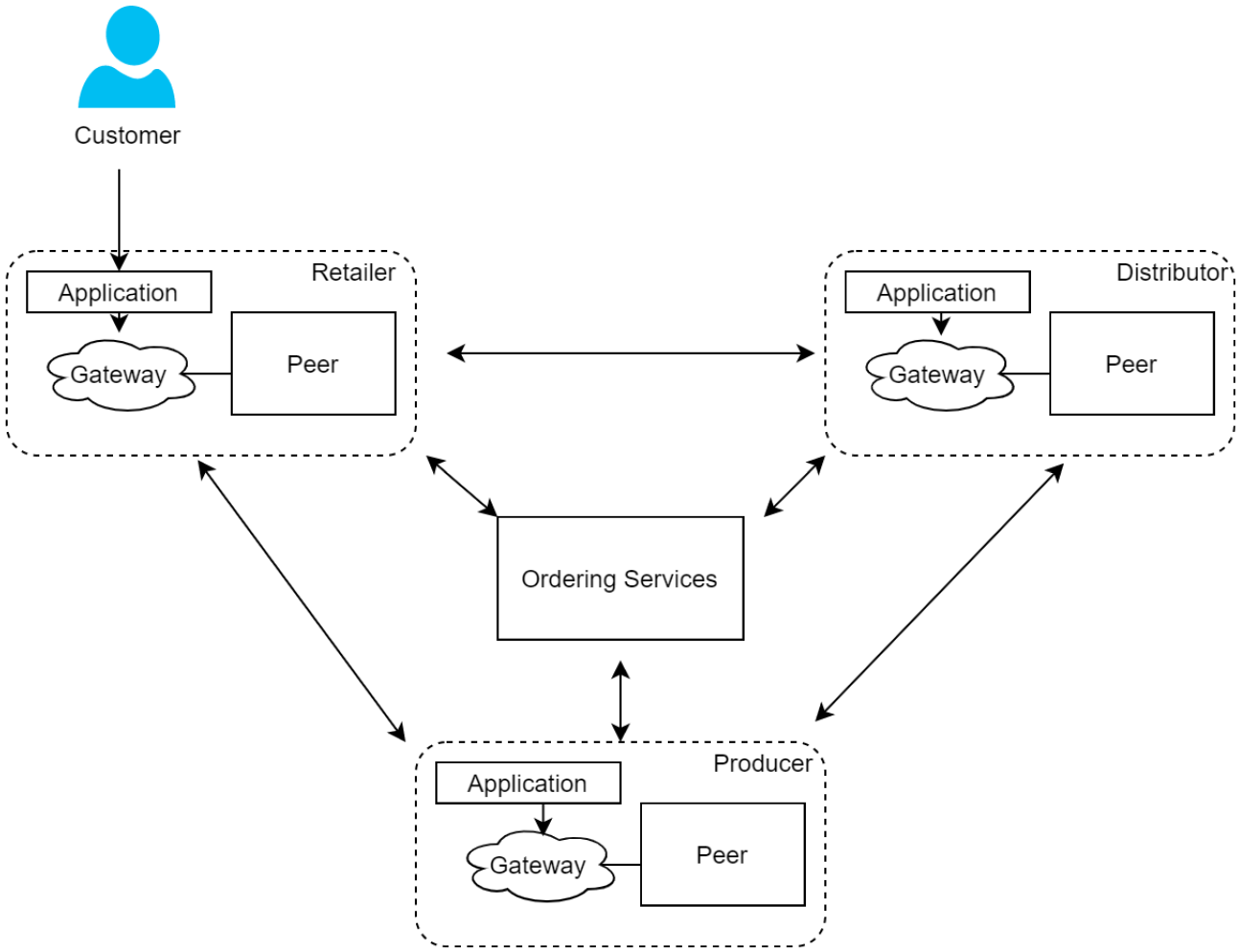

Figure 4.2: Network topology

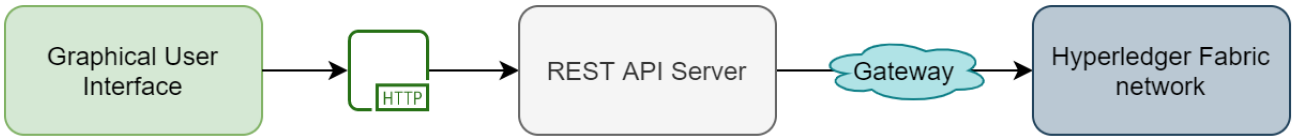

Figure 4.3: System components

ID generation & Identity Management

In order to have a system that is as automated as possible, the entity IDs are generated by the OperationsContract. This is done by maintaining a variable for each type of entity, that gets incremented each time a new ID is required. This ensures that no two entities of the same type can have the same ID, because of the fact that Hyperledger Fabric transactions are atomic in their nature. An example for retrieving the next item ID can be seen in Algorithm 1. For more complex and secure systems, cryptographically secure identifiers should be used and generated randomly.

Organizations have their identities corresponding to their MSPs. This way, any employee of an organization will be identified as the the organization itself. In systems where a more refined identity management is required, client identities can be used instead of MSPs. Therefore, the identities present in the system are:

- *ProdMSP* - representing the Producer organization and its employees.
- DistMSP - representing the Distributor organization and its employees.
- RetMSP - representing the Retailer organization and its employees.

```
1 Function GetNextID:
2 |   nextID ⟵ read(itemsID);
3 |   if nextID = NULL then
4 |   |   nextID ⟵ 0
5 |   write(itemsID, nextID + 1)
6 |   return nextID
7 End Function
```

Algorithm 1: ID generation function for items

OperationsContract Data Structures

The data structures used by the *OperationsContract* can be seen in Figure 4.4. The data structures which contain the ID field are stored on the ledger, while those missing the ID field are created only for a more clear representation and easier interaction. The arrow between two entities represents the fact that one entity points to the other entity by having its ID stored as a field in case both entities are stored on the ledger. Otherwise, the entity that is not on the ledger and does not have an ID is included in the referring entity as a structure field. This choice was made for two reasons:

1. In order to secure the data, it needs to be managed by the system. In this case, saving an entity object on the system ensures that it will be the same for any party that tries to retrieve it at any moment in time.

2. Not all entities require to be saved independently on the ledger. For example there is no reason to save all the locations in a data structure with an ID if the location itself does not represent anything without the data structure that contains it. This also reduces the amount of data that needs to be saved and reduces data throughput.

In a practical example, additional fields and data which are not mandatory to be stored on the ledger (e.g. photos and videos) can be stored off-chain with a link to the ledger ID. Such a solution would reduce the transaction size and make sure the system is having a desired performance.

Items represent physical assets and are identified by their ID. It is worth mentioning that an item does not represent an item type or model, but a real object. Two identical items will receive different IDs in the system. Items are described by:

- ID - the unique number that identifies the physical item and links it with the logical entity. This field gets auto generated by the system.

- Owner - the MSP of the organization that owns this item.

- Description - the name or a short description of the item.

Contracts (or orders) represent buying agreements between a buyer and a seller. Both parties should be members of the network. Contracts can be created only by the Buyer organization as it needs payment proof. This ensures that nobody can make purchases in the name of another organization. When a contract is created, the seller gets notified through events that specify that there is a new order to be processed by him. It has the following fields:

- ID - the unique number that identifies the contract. This field gets auto generated by the system.

- Seller - the MSP of the organization that is selling the owned items. This field is automatically inferred by the system by checking who is the owner of the included items.

- Buyer - the MSP of the organization that is buying the included items. This field is automatically inferred by the system by retrieving the identity of the organization which creates the contract.

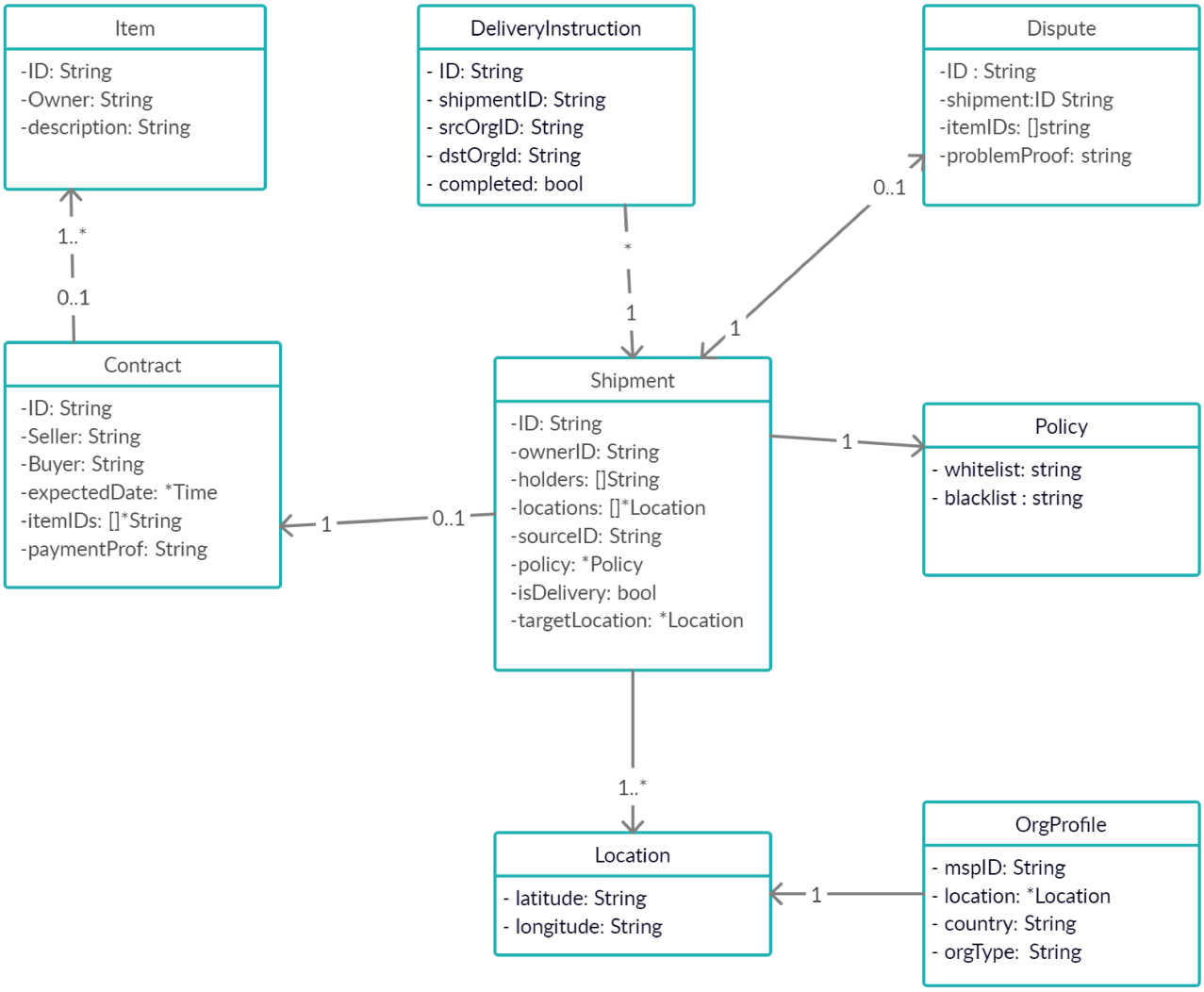

Figure 4.4: OperationsContract data structures

- ExpectedDate - represents the date on which this order should be completed. This field is automatically calculated by the system by applying the formula

$$expectedDate = currentTime + \frac{Distance(Seller.Location, Buyer.Location)}{avgDistancePerDay}$$

  The seller and buyer locations can be retrieved from their profiles, while the average distance per day is a parameter set by the system administrator.

- ItemIDs - represents an array with all the IDs of the items that are bought. These must belong to the same organization, must be different than the buying organization and items must be in the right state to get sold. At least one item must be contained in a contract.

- PaymentProof - a piece of information that proves that the buyer has paid for the ordered items. As the system does not implement a cryptocurrency, this field is present only to represent that Contracts can not be created without paying.

Shipments (or packages) are the system representation of physical packages. Because in real life shipments usually represent simple boxes, the life cycle of a shipment inside the system is short. After it gets delivered, it can not be reused after. Shipments can be of two types:

- *Delivery* shipments, that are created for an order, and are being delivered to the buyer from the seller.

- *Return* shipments, which represent shipments going back from the buyer to the seller. This type of shipments are created for a dispute.

A shipment can not be created if there is no reason to do so, namely if there is no contract or dispute that requires its creation. A shipment has the following fields:

- ID - the unique number that identifies the Shipment and links it with the physical package. This field gets auto generated by the system.
- OwnerID - the MSP of the organization that owns the Shipment. The owner remains the same from the creation until the shipment is delivered at the target location. When a shipment is created, the organization that called the creation transaction becomes the owner.
- Holders - a list of organization identities (i.e. MSPs), which had the shipment under their control. This list gets automatically updated as the shipment goes through the delivery process. The first holder is the organization that created the shipment, followed by all intermediary organizations that received the shipment and sent it forward, and the last holder is the organization which keeps the shipment at the current moment.
- Locations - similar to the Holders field, this one keeps a list of locations that the shipment has been located at. The first location is the location of the creating organization and the last one is where it is currently located. Locations get automatically updated every time the shipment is exchanged between organizations, with the location from the organization's profile.
- SourceID - the ID of the entity for which this shipment was created for. It either points to the contract entity in case of delivery shipments or dispute entity in case of return shipments. Based on this field, it is possible to infer what items does the shipment contain.
- Policy - a set of rules in terms of whitelisted and blacklisted elements that are checked in the delivery process. It is set by the creator of the shipment, which can dictate the delivery rules that he prefers.
- isDelivery - a flag which is set in case of delivery shipments and unset in case of returns. This field is inferred automatically by the system judging by the logic.
- targetLocation - the final destination of the shipment. This field is set automatically as the location of the buyer in case of deliveries and as location of seller in case of returns.

Disputes represent a disagreement or a problem that was detected by the buyer on a shipment. Disputes can only be opened by the buying organization, after the shipment has been marked as delivered. When a buyer opens a dispute, the seller gets notified through an event that signals that one of his shipments are disputed. The fields have the following meaning:

- ID - the unique number that identifies the dispute. This field gets auto generated by the system.
- ShipmentID - the ID of the shipment that is disputed.
- ItemIDs - the IDs of the disputed items contained in the disputed shipment. This allows to open disputes on specific items only, if the rest are as expected. In case the problem refers the Shipment as a whole (e.g. missing or wrong shipment), all the items are disputed.
- ProblemProof - a piece of information proving that the dispute is valid (e.g. photo, video, description). This field has been added to make sure clients can not open disputes without a reason.

Delivery Instructions represent steps automatically generated by the system, which aid organizations in the delivery process of a shipment. The instruction specifies to which organization the shipment needs to be sent. Delivery instructions are generated when the organization which currently holds the shipment retrieves the next instruction. After it gets generated, the instruction must be encoded in a QR code and attached to the shipment. The receiver then scans the QR code and if it is the destination organization, then the instruction is marked as completed. Instructions have the following fields:

- ID - the unique number that identifies the instruction. This field gets auto generated by the system.
- ShipmentID - the ID of the Shipment for which this instruction has been generated for.
- SrcOrgID - the MSP ID of the organization that needs to send the package.
- DstOrgID - the MSP ID of the organization that the package needs to be sent to and that has to receive it.
- Completed - a flag which is set if the instruction has been performed (i.e. the destination organization received the Shipment by scanning the current Delivery Instruction).

OrgProfiles are entities associated with organizations. It holds necessary information about the respective organization which is required to take part in the system, such as:

- MSPID - the unique identifier of the data structure and is the same as the MSP ID of the organization that it describes.
- Location - the coordinates where the organization is located. For the sake of simplicity, in this proof of concept we assume that an organization has a single fixed location.
- Country - the country in which the organization is located.
- OrgType - the type of organization. In the context of this proof of concept, this field can take the value of: Producer, Distributor or Retailer.

Policies are entities that describe rules in the delivery process. Because they live together with the shipment and are only relevant in that context, they are not stored on the ledger independently. Rules are in the form of whitelist or blacklist, which allow or forbid organizations, countries or organization types.

Usually, white or black lists represent arrays or hash tables, which contain specified elements. In the worst case scenario, these lists can contain half of the total number of available elements. Because in a real system the number of organizations can be big, so will be the policies in the system. Having the Policy space complexity be linear to the number of organizations not only increases the amount of data that needs to be written on the ledger, but also increases transaction size when reading or writing every Shipment. This would represent a bottleneck in a real system.
In order to make the Policy be constant size, the data structure chosen to implement white or black lists are *bloom filters*. This makes the process of querying and adding elements to the Policy instant, and offers the desired functionality. Nevertheless, bloom filters are also characterized by false positives. In order to reduce the false positive rate to almost 0, we need to choose the correct parameters for the size of the filter. In the current system, the size is automatically calculated judging by the total number of organizations registered on the network.
When delivering a shipment, judging by the organizations' profiles, the ones that have characteristics (e.g. country, MSP ID or organization Type) included in the blacklist (if any) are removed, and the ones having characteristics in the whitelist are picked. The resulting set of organizations can further participate in the delivery process.

Locations are used to represent physical locations, described by latitude and longitude degrees. As this data structure does not make sense to be kept separately, it is saved as part of the entity that contains it.

StateContract Data Structures

The *StateContract* is characterized by the data structures presented in Figure 4.5. This contract is mainly responsible for keeping track of the current and previous states of certain entities. This is done by maintaining state machines, with predefined rules that decide if a state transition is possible or not. State machines are developed only for Items, Shipments and Disputes and can be created only by the OperationsContract when the transactions for creating items, shipments and disputes are invoked. For every entity, there exists a state machine with the past and current states it has been in. If the transition is considered invalid, the operation that caused this transition will be reverted and the transaction will fail.

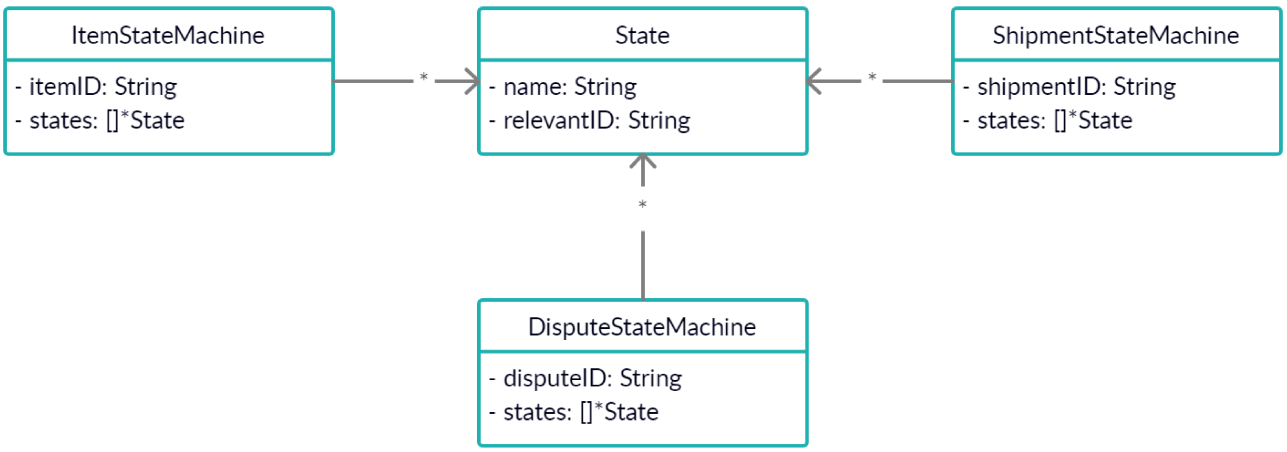

Figure 4.5: StateContract data structures

States represent generic states in which items, shipments or disputes can be. States do not have an ID and are not saved as a separate entity on the ledger as they do not make sense outside the context of the object they belong to. In this way, states are part of the StateMachine entities and saved as their field. A state is described by the following fields:

- Name - the name that describes the state. Possible state names are presented in the following sections.
- RelevantID - an ID which points to an entity stored on the ledger, which is relevant for the current state. Every state has a specific type of relevant ID, depending on the type of state machine it belongs to and the state itself. For example, the state ”Created” of an ItemStateMachine must hold the MSP ID of the organization that created it. The decision to use generic relevant IDs has been made in order to facilitate easy data retrieval and to have proof of validity of the state transition stored inside the state. As an example, when an Item goes in the ”Shipped” state, one can check that the relevant ID corresponds to the ID of the shipment containing it. Without having the relevant ID field, one would have to look through all the Shipments and find the one containing the item. The specification for the states and their relevant IDs are mentioned in the following sections.

ItemStateMachines represent state machines associated with every item on the ledger. This data structure has the following fields:

- ItemID - the ID of the item that this state machine is associated with. In this way, from a logical point of view, the ID of the state machine is the same as the ID of the item.
- States - the past states that the item has been into. The last state in this array represents the current state of the item. This field gets populated automatically when specific transactions are invoked in the OperationsContract, and the user can not access or modify this field directly.

ShipmentStateMachines represent state machines associated with every shipment on the ledger. This data structure has the following fields:

- ShipmentID - the ID of the shipment that this state machine is associated with. In this way, from a logical point of view, the ID of the state machine is the same as the ID of the shipment.
- States - the past states that the shipment has been into. The last state in this array represents the current state of the shipment. This field gets populated automatically when specific transactions are invoked in the OperationsContract, and the user can not access or modify this field directly.

DisputeStateMachines represent state machines associated with every dispute on the ledger. This data structure has the following fields:

- DisputeID - the ID of the dispute that this state machine is associated with. In this way, from a logical point of view, the ID of the state machine is the same as the ID of the dispute.
- States - the past states that the dispute has been into. The last state in this array represents the current state of the dispute. This field gets populated automatically when specific transactions are invoked in the OperationsContract, and the user can not access or modify this field directly.

State Transitions

As mentioned in the previous section, every item, shipment and dispute has a state machine associated with it, responsible for validating and keeping track of the transitions the entity has been into. The rules validating the transitions are represented in the following state transition diagrams, in which the arrows represent conditions for the transitions to take place. States are described in the form of [*StateName*, *RelevantID*].

Item State Diagram represents the states and transitions that an item can be in. It can be seen in Figure 4.6.

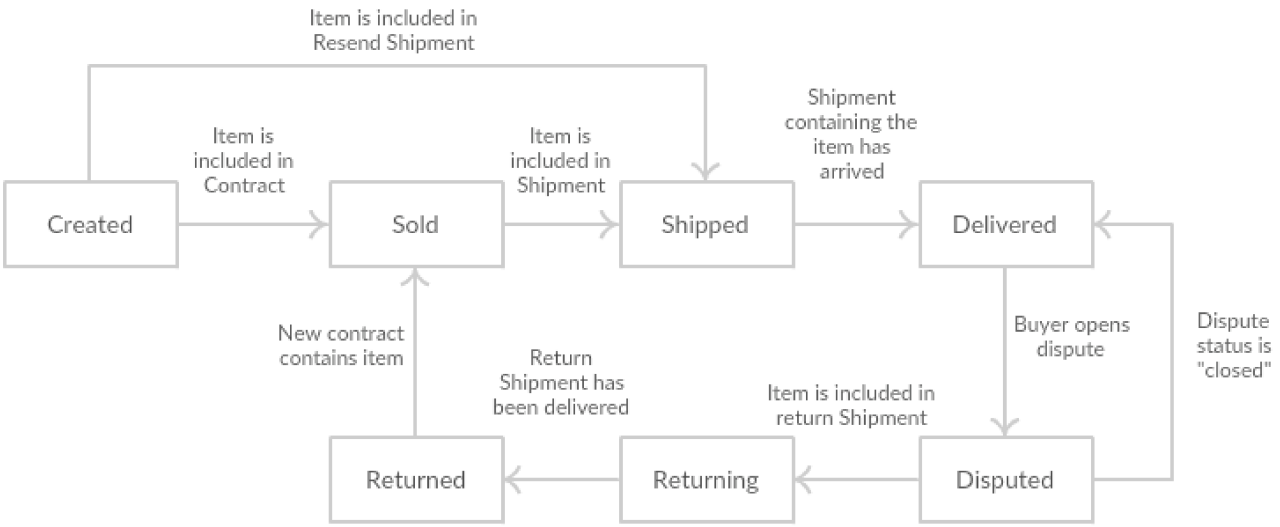

Figure 4.6: Items state transition diagram

The state meanings together with their relevant IDs are the following:

- [*Created*, *CreatorID*] - represents that the item has been created and no other operations have been performed with it. The relevant ID for this state is the MSP ID of the organization that created the item.
- [*Sold*, *ContractID*] - represents the fact that the item has been bought, therefore it has been included in a contract. Only items that are either freshly created or that have been returned can be bought again. The relevant ID represents the ID of the contract that contains this item.
- [*Shipped*, *ShipmentID*] - means that the item has been packed in a shipment. Items must either be bought first and then packed in case of delivery shipments, or they can be directly shipped after being created if it is part of a shipment that serves as a resend. The relevant ID for this state is the ID of the shipment which contains the item.
  Starting from this state, the item will stay in the ”Shipped” state until the shipment will get delivered, as they represent the same entity. This ensures that the state of the item will match the shipment state, as long as it will be part of it. This results in less computations and data stored, as shipments can have hundreds of items, which would end up duplicating the shipment’s states in the delivery process.

- [*Delivered*, *BuyerID*] - means that the shipment containing the item has arrived at the target location, therefore it has been delivered. From this moment on, the item is considered an individual entity again and its states no longer match the ones of the parent shipment. An item also ends up in this state when it is involved in a dispute, the dispute gets resolved and the item stays at the buyer organization. The relevant ID represents the ID of the organization that ordered the shipment, therefore received it.
- [*Disputed*, *DisputeID*] - represents the fact that the buyer organization has opened a dispute on the shipment that delivered this item, and the item has been marked as disputed as well. If the shipment has been disputed but this specific item has not been reported as disputed, the item stays in the ”Delivered” state. When the dispute gets solved and the item stay at the buyer organization, the item goes back in the ”Delivered” state. The relevant ID represents the ID of the Dispute entity saved on the ledger.
- [*Returning*, *ShipmentID*] - represents the fact that the buyer and seller organizations have agreed on a resolution mechanism and the buyer will return the disputed items through a return shipment. This state is the equivalent of ”Shipped” state, in case of returning shipments.
  From this moment until the return shipment will get delivered, the state of the item will correspond to the state of the shipment that contains it. The relevant ID is the ID of the returning shipment that contains the item.
- [*Returned*, *SellerID*] - means that the return shipment containing the item has been delivered back to the seller organization. From this moment on, the item state machine is detached from the shipment state machine and is considered an individual entity. The relevant ID represents the MSP ID of the organization that received the shipment, therefore, of the seller organization.

Shipment State Diagram represents the states and transitions that a shipment can be in. One important thing to note is that shipments are not reusable, therefore even if it gets returned, another shipment will be created instead of using the same logical entity. It can be seen in Figure 4.7

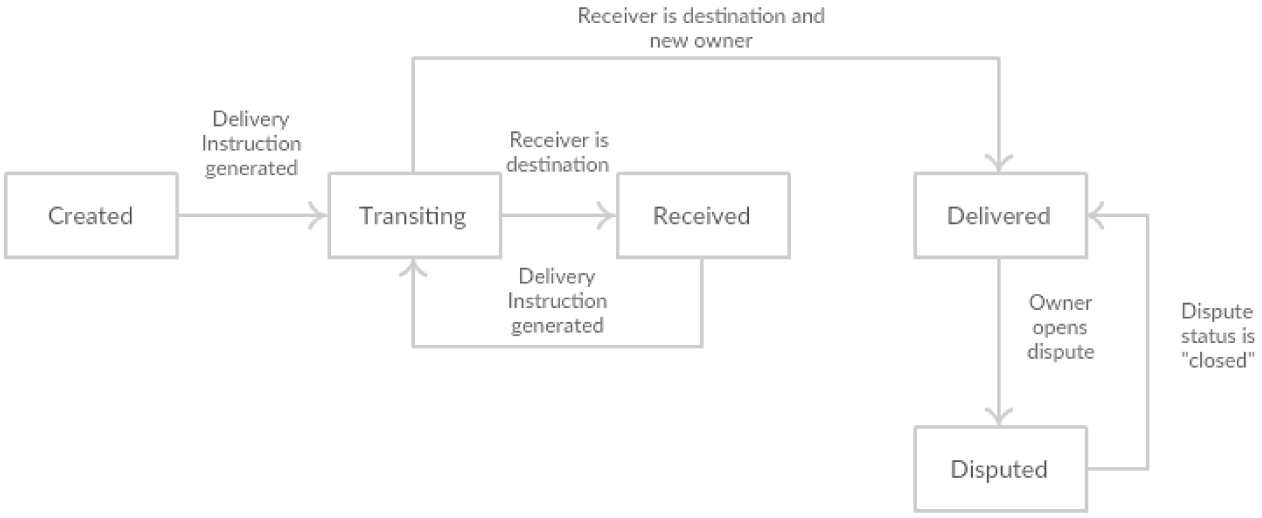

Figure 4.7: Shipments state transition diagram

The states meanings are:

- [*Created*, *SourceID*] - represents the beginning of the shipment life cycle. Shipment state machine gets assigned this state during the shipment creation transaction. At the creation of the shipment, all items contained in the shipment go to ”Shipped” state. The relevant ID represents the ID of the contract or dispute that this shipment has been created for.
- [*Transiting*, *InstructionID*] - represents that the shipment is in the delivery process and is being sent to the next destination. Transition to this state is happening when the organization that currently holds the shipment receives a delivery instruction. The relevant ID represents the ID of the generated instruction.

- [*Received*, *ReceiverID*] - signals the fact that the shipment has been received by the target intermediate organization and is not at the target location. In order to transition to this state, an organization needs to receive the shipment by scanning the QR code containing the delivery instruction. The relevant ID represents the MSP ID of the receiving organization.
- [*Delivered*, *BuyerID*] - means that the shipment has been received by the destined organization (that is - the buyer) and is at the target location. In order to transition to this state, the buyer must receive the shipment by scanning the QR code, and the system will automatically perform the required transition. Additionally, when a dispute gets solved, the disputed shipment also transitions to this state as shipments are not reusable. When Shipments get delivered, all the items contained in the shipment transition to the ”Delivered” state as well. The relevant ID represents the MSP ID of the organization that ordered the items contained in the shipment.
- [*Disputed*, *DisputeID*] - represents the fact that the buyer has opened a dispute on the delivered shipment. Shipments get disputes if any of the items contained by it get disputed. After the dispute gets solved, the shipment transitions back to the ”Delivered” state as it can not get reused. The relevant ID represents the ID of the dispute entity saved on the ledger.

Dispute State Diagram represents the states and transitions that a dispute entity can be in. It can be seen in Figure 4.8

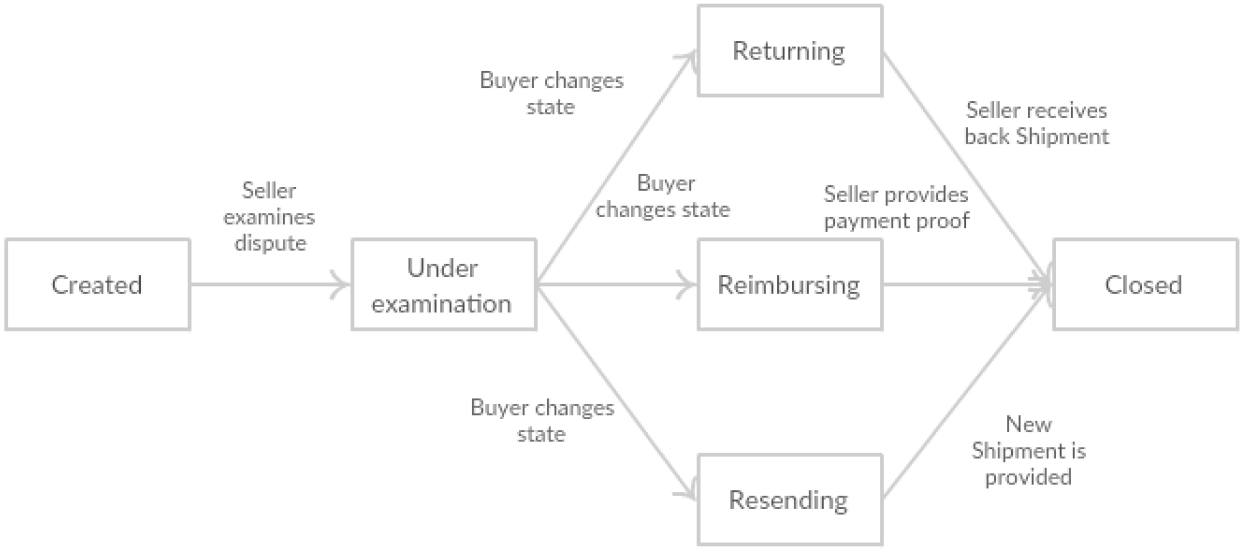

Figure 4.8: Disputes state transition diagram

The states represent the following:

- [*Created*, *ShipmentID*] - means that a dispute has only been opened by the buyer organization. Transition to this state happens together with the dispute opening transaction. The relevant ID represents the shipment which is being disputed.
- [*Under Examination*, *SellerID*] - represents the fact that the seller organization has started looking into the dispute. During this state, off-chain negotiations between the seller and buyer organizations must happen and they must agree on a way to solve the dispute, by using the information offered by the system. Transition to this state is done when the seller signals that he initiated the resolution process. This is done also to let the buyer know that things are moving towards finding a solution for the reported problem. The relevant ID represents the ID of the buyer organization.
- [*Returning*, *BuyerID*] - represents that the involved parties have agreed on a resolution method, and the buyer has chosen to return some or all the items from the shipment. The transition to this state will allow

the creation of a return shipment, which otherwise would not be possible. The relevant ID represents the MSP ID of the party that decided the settlement method - the buyer.

- [*Reimbursing*, *BuyerID*] - represents the fact that the buyer prefers to get all or part of his money back for the ordered shipment. Transition to this state will allow the seller organization to start the reimbursement process. The relevant ID represents the MSP ID of the party that decided the settlement method - the buyer.
- [*Resending*, *BuyerID*] - means that the parties have agreed to resend some or all the items contained in the disputed shipment in a new shipment. This allows the seller organization to start the resend process. The relevant ID represents the MSP ID of the party that decided the settlement method - the buyer.
- [*Closed*, *ShipmentID*] - signals the resolution of the dispute through the chosen settlement method. The condition for transitioning to this state and the relevant ID change depending on the previous state:
  - From ”Returning”: seller organization must receive back the returning shipment. Relevant ID is the ID of the return shipment destined for the seller.
  - From ”Reimbursing”: seller organization must provide proof of off-chain payments representing the reimbursed money for the disputed items. Relevant ID is the ID of the disputed shipment that is being reimbursed.
  - From ”Resending”: seller organization must provide as solution the ID of the shipment which resends the disputed items. Relevant ID is the ID of the new resending shipment.

Delivery Path Computation

In order to have an automated system as much as possible, the system also needs to decide how shipments are delivered. For this, a dynamic path computation algorithm has been implemented. It is dynamic in the sense that the path is decided during the delivery process, after every completed step. This design choice makes the system resilient in front of network changes in case new organizations join the network or if others leave it. Another advantage is that the path does not need to be stored anywhere, but a limitation is that it is more computationally intensive.

ComputeInstruction The algorithm for calculating the next instruction in the delivery path is presented in Algorithm 2. ComputeInstruction function receives as parameter the shipment for which the instruction is being requested, and verifies the identity of the calling organization (line 2). As only the organization currently holding the shipment is allowed to get the next instruction, any other different organization than the holder, will get an error. In order to pick the next organization in the path, all available organization profiles in the network are filtered based on the black and white lists given by the policy (line 4). Next, based on a heuristic function, one organization is chosen as the next step in the delivery process (line5), after which the instruction entity is generated and returned to the calling organization (lines 6 and 7).

Heuristic The heuristic function is supposed to pick from a list of available organization, the most suitable organization which should serve as the next step in the delivery process from source (src) and destination (dst). In the current design, the heuristic chooses the organization which is the closest to the source location, while minimizing the distance to the final destination. In other words, it picks the closest organization which is in the way to the destination. For this, the heuristic uses the location of each organization which is stored in their profiles. Firstly, the variables are instantiated (lines 10 and 11), then the distance from the source to the destination is calculated (line 12). For every individual organization from the filtered ones, a score is calculated that shows how good of a candidate is this organization to be the next step in the path. The score is calculated by the following formula:

$$Score = \frac{Distance(src, dst) - Distance(org.location, dst)}{Distance(src, org.location)^2}$$

The numerator represents how much shorter the distance to the destination is from the new organization than from the current location. This number gets bigger for organizations that are closer to the destination.
If we would only use the above mentioned parameter to calculate the score, then the algorithm would pick the closest organization to the destination as the next step, which is not how we want it to be designed. In order to pick the new organization to be closer to the source, we need to use the denominator. It represents the distance from the source to the new organization, and the bigger it is, the less likely it will be selected as the next organization. In conclusion, the score is higher for those organizations that are closer to the source, while minimizing the distance to the destination.
The function considers only those organizations that are closer than the destination itself (line 15), calculates the above mentioned score (line 17) and keeps track of the organization with the highest score (line 18, 19 and 20).
In case of need, any other heuristic function can be implemented which would prioritize organizations based on the business needs. For example the cost or time of delivery could also be taken into account.

```
1  Function ComputeInstruction(shipment):
2      if tx.getMSPID! = shipment.holder then
3          return Error
4      filteredOrgs ⟵ filterOrgs(shipment.policy);
5      nextOrg ⟵ heuristic(filteredOrgs, tx.getMSPID.location, shipment.targetLocation);
6      nextInstruction ⟵ DeliveryInstruction(shipment, shipment.holder, nextOrg, false);
7      return nextInstruction
8  End Function
9  Function heuristic(filteredOrgs, src, dst):
10     maxHeur ⟵ −Inf;
11     maxHeurOrg ⟵ NULL;
12     distToDestination ⟵ distance(src, dst);
13     foreach org ∈ filteredOrgs do
14         distToOrg ⟵ distance(src, org.location);
15         if distToOrg <= disteToDestination then
16             distFromOrgToDestination ⟵ distance(org.location, dst);
17             heur ⟵
                (distToDestination − distFromOrgToDestination)/(distToOrg ∗ distToOrg);
18             if heur > maxHeur then
19                 maxHeurOrg ⟵ org;
20                 maxHeur ⟵ heur;
21     end
22     return maxHeurOrg
23 End Function
```

Algorithm 2: Delivery Instruction generation algorithm

Delivery Flow

To understand the systems' way of functioning on a general level, we can go through the flow of actions in the delivery process of a shipment. These steps can be seen in Figure 4.9.

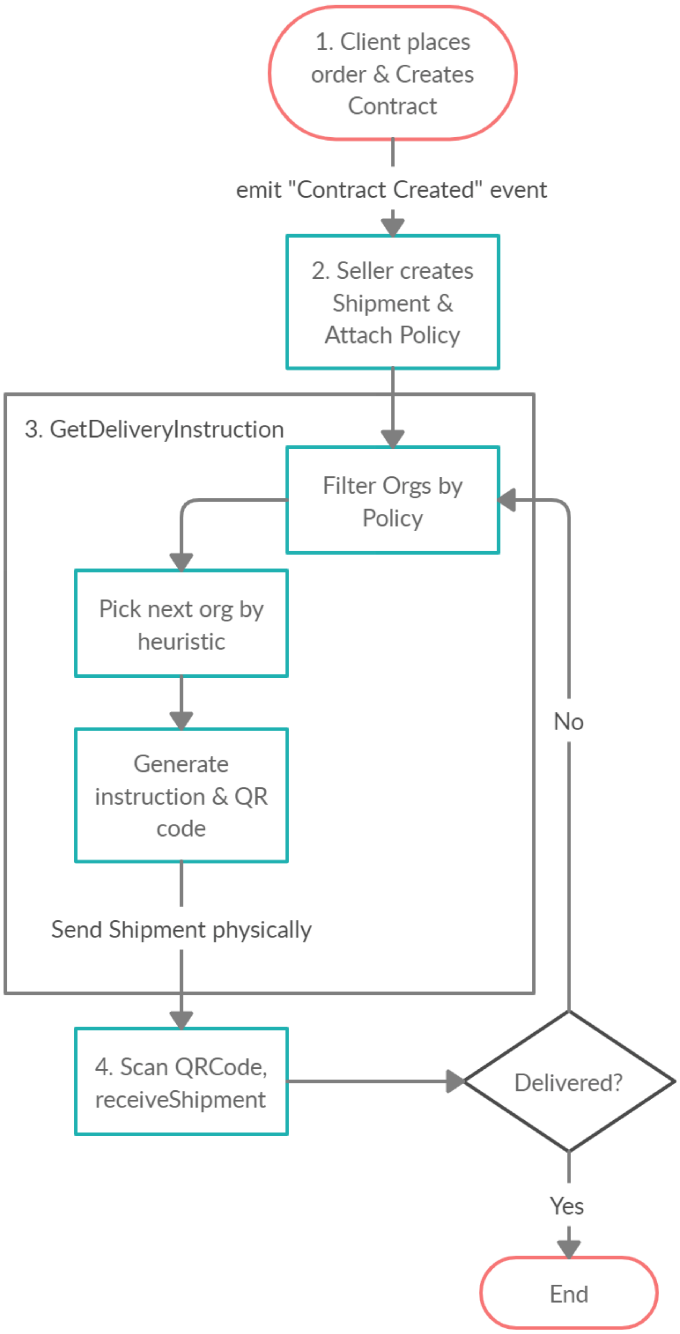

Figure 4.9: Shipment delivery flowchart

The flow is composed of the following actions:

1. The client uses the system to browse the available items in the system offered by other organizations. After deciding which ones he wants to order, he makes the off-chain payment for those items and places order, for which the system creates a Contract that contains all the needed information. An event is emitted by the system to let the seller know there is a new order that he needs to process.

2. The seller examines the placed order and creates a shipment for it. In the creation process, he attaches a policy consisting of a white and black list, which specify the delivery rules.

3. The current holder of the shipment, which initially is the seller, requests from the server the next DeliveryInstruction. In order to compute it, the system filters all organizations in the network based on the attached policy. Then, based on the heuristic function developed, the best suited organization is chosen as the next step. After having the next organization picked, the DeliveryInstruction is generated, and it gets converted to a QR code that is attached to the physical shipment. The holder sends the shipment to the destination mentioned in the received instruction.

4. The receiving organization scans the QR code that is on the shipment and completes the shipment reception operation. If the shipment has arrived at the target destination, the flow is complete. Otherwise, the flow continues starting with step 3.

From the system interactions point of view, the above mentioned flow would be similar to the one in Figure 4.10. Every step represents a transaction called by the respective actor trough their application. As we can see, the actors interact strictly with the OperationsContract. This is mainly because the StateContract publicly exposes only the methods for retrieving the machine states, while other methods that are used for state transitioning are kept private. In this way, the only way one can change the state of an asset is through the OperationsContract.
The bidirectional flow between the OperationsContract and StateContract mean that they communicate with each other. This happens when the OperationsContract tries to update the state of the assets or when checking if the operations are valid by querying the current state the asset is in.

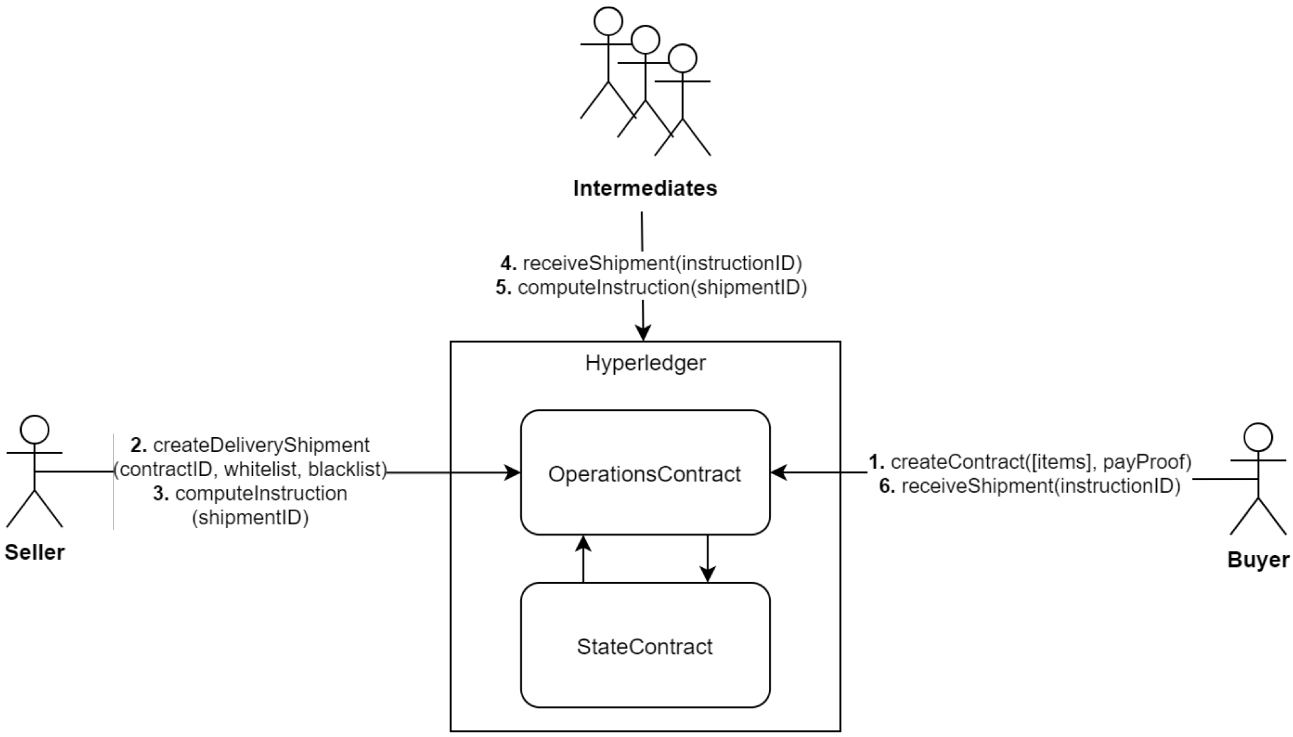

Figure 4.10: Shipment delivery interaction diagram

From the state transition's point of view, a delivery flow would result in state transitions similar to the one in Figure 4.11. In this example, two items are being ordered by the client, and in that moment the items' states transition to "Sold". When the shipment containing the items gets created, the items switch to "Shipped" state. This state contains the ID of the shipment, therefore it is possible to track the shipment state transitions by querying the item states. Next, the shipment gets in the "Transiting" state when the holder requests a DeliveryInstruction, followed by "Received" when the QR code gets scanned by the destined organization. This process continues until it reaches the final destination, after which the items together with the shipment switch to the final "Delivered" state.

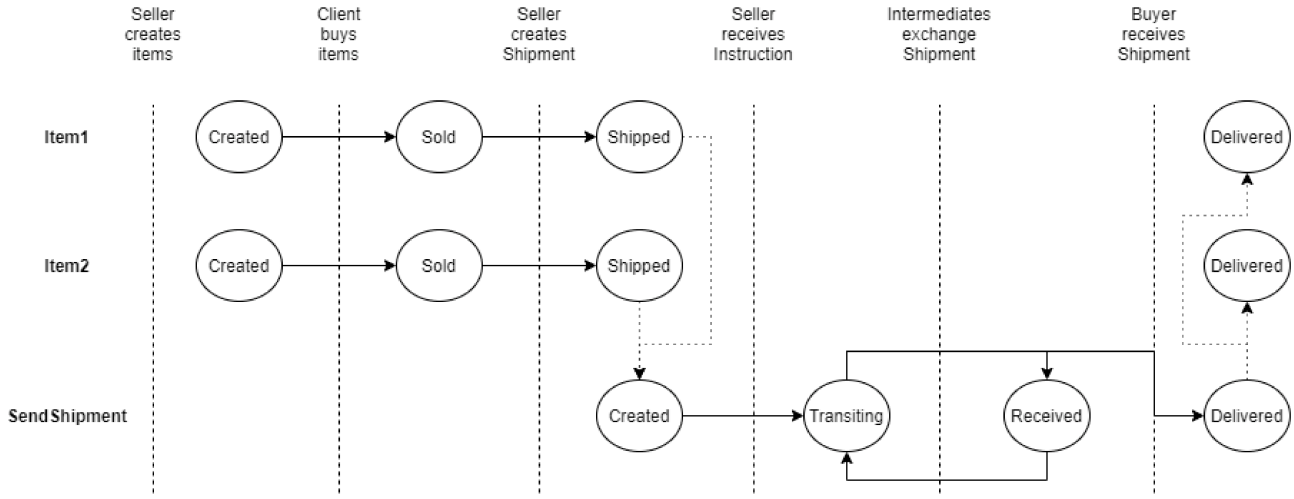

Figure 4.11: Shipment delivery state transitions example

Dispute Flow

As an example for the dispute flow, we consider the case in which the buyer wants to return the items from the shipment. The steps can be seen in Figure 4.12.

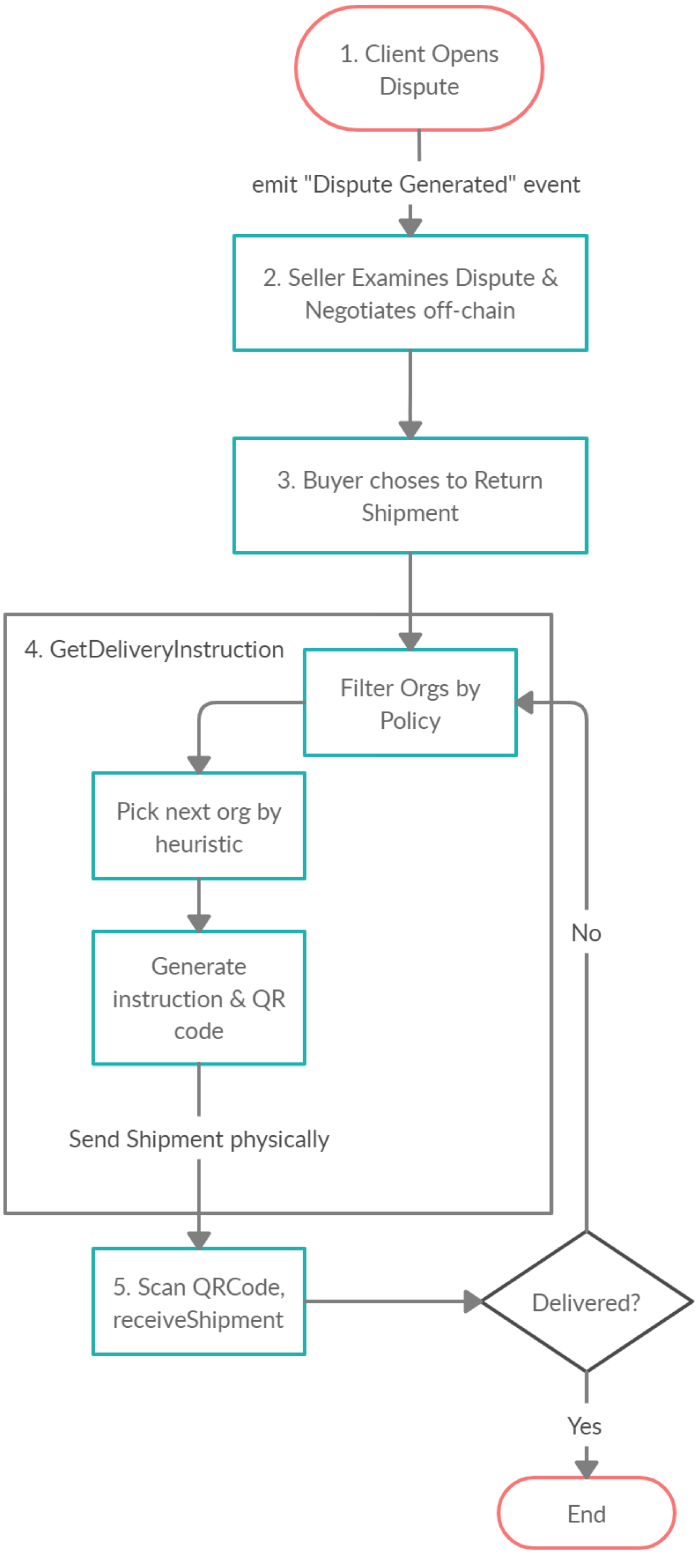

Figure 4.12: Shipment return flowchart

The flow is composed of the following actions:

1. After receiving the ordered items, the client reports one or more items from the shipment as disputed and the system creates a dispute entity. After this step, the system generates a ”Dispute Generated” event which lets the seller know that one of his shipments are being disputed.

2. When the seller looks into the details of the dispute and disputed shipment, the client gets notified that his dispute is ”Under examination”. During this time, seller and buyer investigate the dispute by querying the information stored on the ledger and negotiate off-chain a suitable solution for the client in order to solve the dispute.

3. After the buyer decides that his organization will return one or more items from the shipment, he uses the system to report the return of the shipment. After this step, the system creates a new logical entity - return shipment, with the disputed items. The newly created shipment gets attached the same policy that was used in the delivery process and has the location of the seller as the targetLocation.

4. The shipment holder (the client in this case) requests a new DeliveryInstruction to find out where he needs to send the shipment. Just like in the delivery flow, the system filters all available organizations based on the attached policy, picks next organization based on the heuristic and generates a new instruction with the new organization as the destination. After this, the shipment gets physically delivered where the instruction specifies.

5. After receiving the shipment, the destined organization scans the QR code and the system finalizes the reception operation. If the shipment has reached the final destination (that is, the seller organization), then the process is finalized. Otherwise, the process continues starting from step 4.

The interactions with the system for the shipment return are presented in Figure 4.13. Similarly to the normal delivery case, actors interact only with the OperationsContract. For this system, the ”examineDispute” transaction would happen automatically when the seller decides to view the opened dispute, but depending on the business rules, it can also be explicitly invoked, which would signal that the seller is open for negotiating a solution for the dispute.

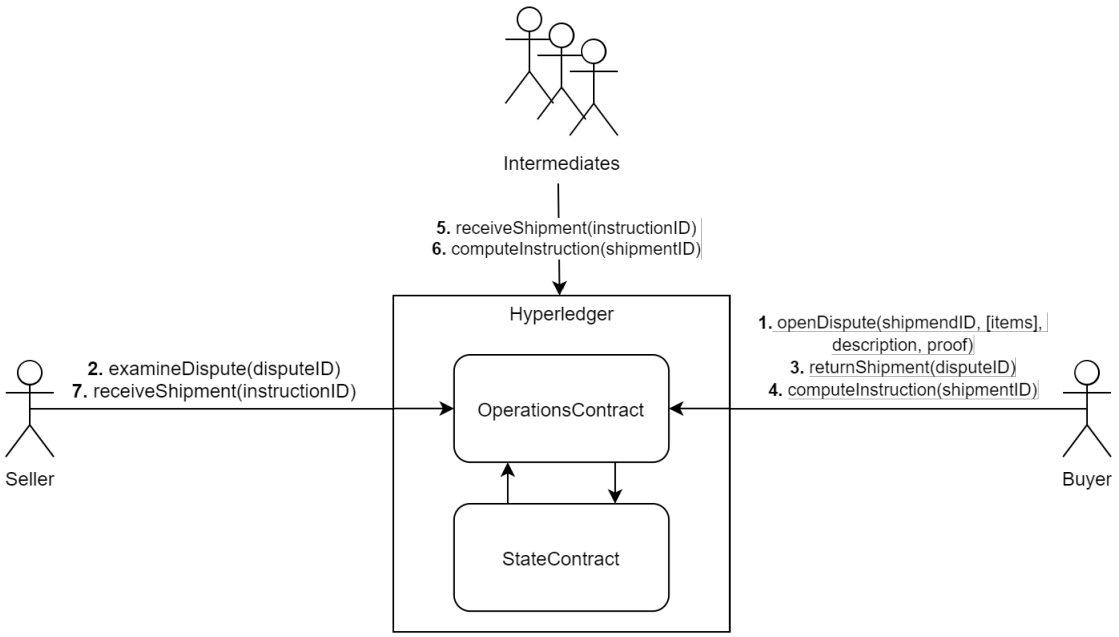

Figure 4.13: Shipment return interaction diagram

The state transitions in this case can be seen in Figure 4.14. Once the buyer opens a dispute, the disputed items together with the shipment go in the ”Disputed” state. From this moment until the dispute gets closed, the items’ and the shipment’s states correspond to the state of the dispute. When the seller investigates the dispute and negotiates with the buyer, the dispute stays in the ”Under examination” state. After deciding on a way to solve the issue, the buyer reports in the system that he wants to perform a return and a return shipment is created. Just like for normal deliveries, items go into the ”Retuning” state, with the relevant ID from the state pointing to the newly created return shipment. Then the shipment passes through the delivering process and ends up in the ”Delivered” state. When this happens, the dispute gets ”Closed”, the initial shipment goes in the ”Delivered” state to show that it is no longer disputed, and the items go in the ”Returned” state, from which they can be bought again by other clients.

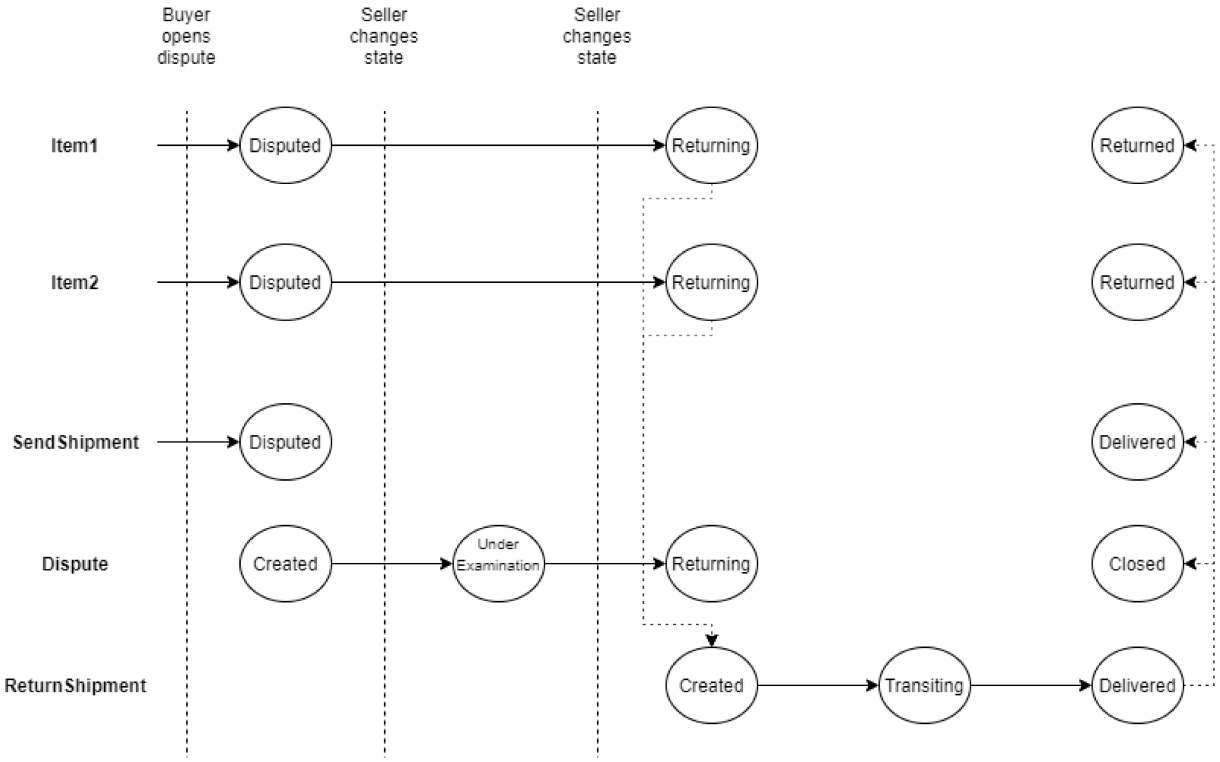

Figure 4.14: Shipment return state transitions example

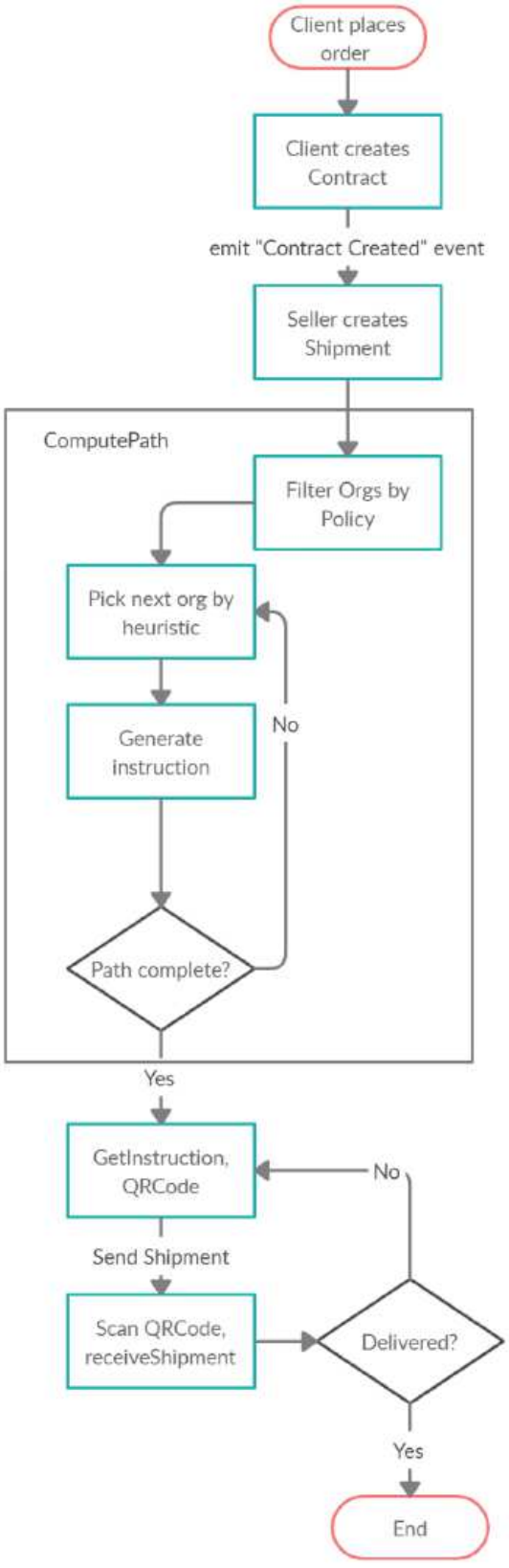

Figure 4.15: Shipment delivery flowchart - static path version

Design Alternatives

The current design has been chosen among other possible options for specific reasons. In this section, some design alternatives that were taken into account will be discussed.

Path Computation A possible alternative to dynamic path computation is the static one. This means that the delivery path is computed before the beginning of the delivery process itself. The flowchart of this approach can be seen in Figure 4.15.
This approach implies that the delivery instructions need to be stored on the ledger in a way that they can be queried individually by each intermediate organization. This can be achieved by storing each instruction with a composite key including the shipment ID and the MSP ID of the organization. In this way, the instruction with key ”ship-’ID-X’instr-’mspID-Y’” would retrieve the instruction destined for the organization with MSP ID equal to ’mspID-Y’ for the shipment with id ’ID-X’. This ensures that intermediates do not have to and can not query more instructions than the one meant for its organization (if there exist any). This is not only efficient but also secure from the privacy point of view. Another advantage is that there is less computation to be done, as the path is computed only once, instead of computing the best next destination for each intermediate.
The biggest disadvantage is that this approach is vulnerable to network changes. This would mean that either the path will not be the most optimal one in case a more suitable one becomes available or even worse, it will be impossible to complete it in case a chosen organization becomes unavailable.

Advanced Heuristic Currently in the proposed design, a simple heuristic that prioritizes the closest node in the direction towards the destination is chosen. This is because the system is a proof of concept and has the scope of demonstrating that an implementation in this domain is possible and feasible. In different scenarios, this heuristic can be designed to encompass different business needs. For example, organizations can be differentiated by their type and only those responsible for transportation could be picked in the delivery process. Also, delivery time and price can also be an important factor. It is also possible to design multiple heuristics and let the system user decide which one suits the business needs for that case.

Handling Payments In order to make the system more automated, a method for verifying or handling payments could be implemented. This can be done by using a built-in cryptocurrency, which would allow parties to make payments within the system for the items that they order. This would allow the system to act as an intermediate, which could handle returns, reimbursements or any other types of financial operations.
Alternatively, the system could have a connection with a payment system (e.g. VISA), that would allow him to verify the proof of payments given by the organizations. This approach is less automated, therefore the system can only partially verify the legitimacy of payments, but has the advantage of being able to integrate more easy with modern payment methods used nowadays.

4.3.2 REST Server

The REST server has the role of exposing the functionalities of the designed fabric network, through an API. It uses the Fabric SDK to connect to a peer of an organization, and manages the identity of the caller by using wallets. This component is necessary because client applications can not connect directly to Fabric network, as this requires libraries that can be used only on the server side.
In order to be able to connect to the network, every organization needs to enroll an admin user, which allows the connection through a gateway to one peer of that organization. The server is not designed to be used as an API to build any kind of client applications on top. This is because it is meant to be used as a server in a proof of concept application, which implies multiple drawbacks. One of them is that there are no advanced authentication methods, mainly because there is no need to have any, as the client application will be a common one for all the organizations.
The API is composed of HTTP GET and POST methods. Each method makes a new connection with the Hyperledger network and invokes the specified transaction or query using the identity that is currently set. The request data, response and the data passed as parameters to transactions are in JSON format. The server API methods are briefly presented in Table 4.1.

4.3.3 Graphical User Interface

The Graphical User Interface component is responsible for showcasing the system functionalities in an easy and user-friendly way. As the information on the ledger is always correct, the application is just a view that the organizations have of the underlying system. Sometimes, because of the connection problems or issues with the nodes, the data in the interface might not always correspond to the one on the ledger. For this, the interface needs to represent this issue by highlighting which information corresponds to the one on the ledger. Therefore, some data needs to be stored locally, which would represent the truth from the organization's point of view. As an example, if an organization places an order with some items, but the connection to the blockchain system gets broken, the interface should show this mismatch.
The data is fetched from the Hyperledger Fabric network using the developed REST Server described previously through HTTP, every time the user performs an action. The data fetched from the server is considered ground truth, and it populates the local storage of the organization. The records present on the ledger are considered "Confirmed" in the sense that it is up to date with the ground truth. When there is information stored locally but missing on the ledger, it has an "Unconfirmed" status, meaning that the actions which resulted in that data were not validated, and the issue needs to be looked into.

Chosen approach

During the design process, there were being take into consideration two approaches regarding the way the interface is going to be used.

The first one is to design the application as an end product indented to be used by each individual organization. This would imply that organizations would have to authenticate themselves, could see assets strictly relevant to them and some parts of the available information, such as past states of assets, would not be available because of privacy reasons. The advantage is that the flow of actions would be tailored specifically for the end user, making the use of the application much easier and enjoyable. For example, actions would be possible to perform with a single click only at the right location (e.g. an open dispute button in the page with shipment information). Also, error messages would communicate the status of the action results.

Method	URL	Parameters	Description
GET	/getItem	id: String	Retrieve an item by ID.
	/getContract		Retrieve a contract by ID.
	/getDispute		Retrieve a dispute by ID.
	/getShipment		Retrieve a shipment by ID.
	/getItemStateMachine		Retrieve the item state machine.
	/getShipmentStateMachine		Retrieve the shipment state machine.
	/getDisputeStateMachine		Retrieve the dispute state machine.
	/getAvailableItems	-	Retrieve items in ”Created” or ”Returned” state owned by other organizations and all items owned by the current organization.
	/getAllOrgProfiles		Retrieve all profiles of registered organizations on the network.
	/getOrganizationContracts		Retrieve contracts in which the current active organization is either seller or buyer.
	/getOrganizationShipments		Retrieve shipments created or currently held by the current organization.
	/getDisputesByOrganization		Retrieve all the disputes opened by the current organization.
	/getDisputesForOrganization		Retrieve all the disputes opened for the current organization.
	/currentOrganization		Retrieve the MSP ID of the current active organization.
	/enrollAdmins		Create a wallet with an admin identity for the Producer, Distributor and Retailer.
	/getOrgProfile	msp: String	Retrieve the profile of an organization.
	/changeOrganization	org: String	Change the current active organization.
	/computeInstruction	shipment_id: String	Compute the next DeliveryInstruction associated to a shipment.
	/receiveShipment	instruction_id: String	Receive a shipment associated with a DeliveryInstruction.
	/examineDispute	dispute_id: String	Examine a dispute.
POST	/createItem	description: String	Create an item
	/saveProfile	msp_id, country, org_type, long, lat: String;	Save the profile details of an organization.
	/createContract	payment_proof: String; items_ids: JSON(Array[String])	Create a contract.
	/createDeliveryShipment	contract_id: String whitelist, blacklist: JSON(Array[String])	Create a delivery shipment.
	/createDispute	shipment_id, proof: String; items_ids: JSON(Array[String])	Create a dispute.
	/initiateSettleDispute	dispute_id, settle_method: String;	Initiates the settlement process of a dispute.
	/returnShipment	dispute_id: String	Return a shipment involved in a dispute.
	/resendShipment	dispute_id: String; items_ids: JSON(Array[String])	Resend a shipment with new items for a dispute.
	/reimburseShipment	dispute_id, proof_payment: String	Reimburse a shipment involved in a dispute.

Table 4.1: REST API methods

Even though this is the closest design to how the application would actually be used in a real life scenario, it does not showcase all the functionalities of the system. Also, completing a full delivery operation through the system requires several actions from different organization, which in this case would imply logging out and logging in every time, making the process cumbersome.

The other option would intend a totally different outcome. It would represent a back office made for system admins, that could have access to all the data. This implies that privacy would not matter, and the actions would be performed in a more manual, less flowy or real way. For example, the admin would have a generic page for disputes and fields where he should put information in order to open a dispute. This design choice has the advantage that it can be easily implemented and it would show all the functionalities of the system. The disadvantage is that it is much harder to use without proper knowledge about the action flows. It also would not offer proper input or error handling and would result in a worse user experience.

Having the previous arguments in mind, the user interface has been designed as a combination of those approaches. More specifically, it tries to offer a flowy user experience to enable easier interaction with the system, while keeping some of the admin features in order to showcase most of the available functionalities.

Layout Design

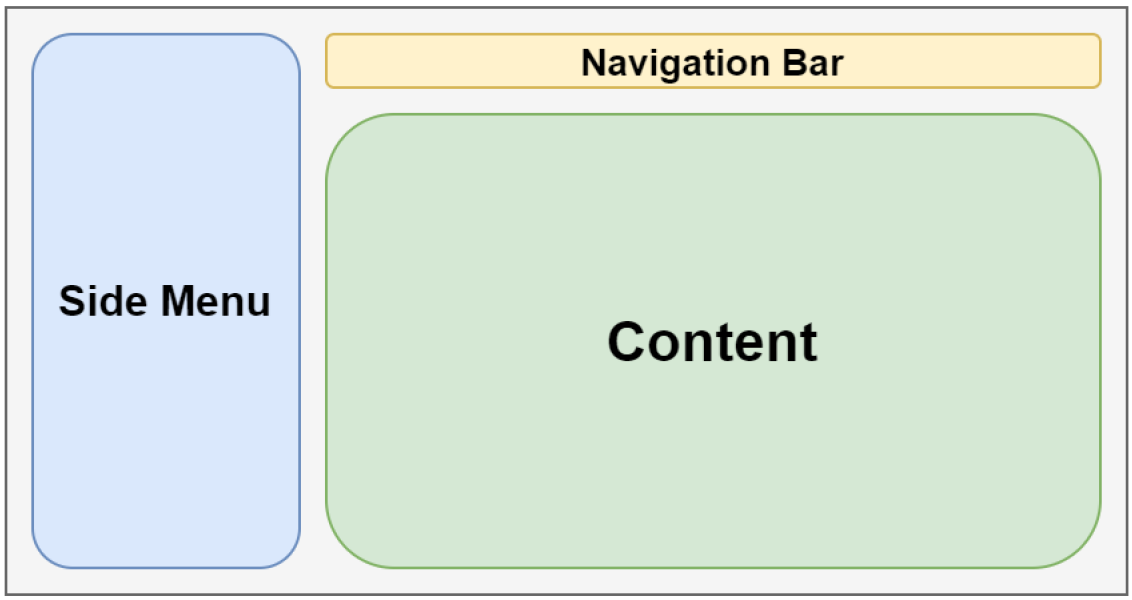

Figure 4.16: Graphical User Interface main components

From a general, user point of view, the interface layout has three main components, which can be seen in Figure 4.16. They have the following roles:

- *Navigation Bar* - this component is present all the time at the top of the screen and has the role of showing what is the current active organization and is offering functionalities to change it. This is an alternative to more complex authentication methods. It also gives the possibility to view and change the profile of the current active organization. For example, when a user wants to act as a different organization, he should be able to do so with a few clicks.

- *Side Menu* - this component, just like the Navigation Bar, is visible all the time on the screen and does not change. It should make the changes of tabs possible, which in turn should be reflected in the Content component. The menu items should represent pages with information and functionalities that make sense when grouped together. The current design gives the option to change tabs between the following pages:

  - Items - contains information about the available items and owned items. Should give the possibility to view item information, create new items and pick items for purchasing.

  - Orders - contains information about the orders opened for and by the current organization. Additionally, the user should be able to view order details and items, complete orders by creating delivery shipments and query the state of the order.
  - Shipments - contains information about the shipments created by the organization and those that have been received by it, whether they are in transit or destined for the organization. Similarly, the user should be able to query information about the shipments and perform actions related to it, such as opening disputes. In this page, organizations can also perform actions to simulate the scanning of a shipment, resulting in its reception.
  - Disputes - contains information related to the disputes opened by and for the current organization. In the case of those opened by the organization, it can see its status and decide on a settlement method. For those opened for the organization, it can examine the dispute and settle it according to the solution chosen by the buyer.

- *Content* - this component contains the main information and functionalities. It changes content based on the chosen tab in the Side Menu and reacts to the actions performed bt the user in the application.

Chapter 5

Security Analysis

The security aspects of the proposed design stem from two main sources. The first one is the security characteristics inherited from the blockchain, namely from the Hyperledger Fabric model. The second layer of security is given by the design choices that were made in order to maintain privacy, access-control and traceability .

5.1 Hyperledger Fabric Security

By building the system on top of the Hyperledger Fabric network model, the system acquires its security capabilities. Among them are the following:

5.1.1 Immutability

Since Fabric is a permissioned ledger, the immutability requirements change compared to a public blockchain. Before blocks are added to the ledger, they need to be signed by the ordering service. After what the transactions are ordered in blocks, they are sent to peer nodes, which can check the validity of the blocks by having access to the Certificate Authorities of the ordering node. Therefore it can know if the blocks are coming from a trusted source or not. If an orderer node is made out of thin air or his private keys are compromised, then it leads to the orderer being able to write any new blocks effortlessly. This implies that having a secure ordering service is quite important, especially when the orderer participates as a peer node as well.
Having a compromised orderer allows modifying previous existing blocks by censoring or reshuffling transactions. Even so, because transactions are signed by clients, this does not let the malicious party modify transactions. Even if we assume that the ordering service gets compromised, this means that the malicious party will result in having a different ledger than the other participants, and having a valid but different ledger does not help in any way if you can not convince others that your version is the right one. If in public blockchains you can convince other members by having a longer chain through forking, forks are by design impossible in Fabric, resulting in a bug. If an honest node receives a block with an unmatching hash having the same height, then it is sure that the block corresponds to a different ledger and a critical situation arises.
Because of the private nature of the network, private channels can be implemented even with two participants. In these cases where one party can modify the ledger and there are no other peers to compare it with, the immutability property lies in a strong endorsement policy. By having a policy that requires all the parties to sign the endorsement, one can be sure that the participant's states match. If the data is forged, the endorsement process would fail.
For above mentioned reasons, the data on the designed system can be considered immutable.

5.1.2 Privacy and Confidentiality

The privacy aspects of Hyperledger Fabric include multiple aspects. One of them is the asymmetric cryptography and zero-knowledge proofs, which have the role of separating the transaction data from the records on the ledger. This results in the protection of the data from the underlying algorithm. Thus, the orderers that order the transactions have no knowledge of the transaction data.

Another aspect is the fact that the organizations participating in the network are legitimate. This is achieved by the fact that in order to be a participant in the network, one must be given a valid digital certificate by the Membership Service Provider that would prove his identity. This achieves user privacy between the participating actors, by keeping the other non-involved users out of the network. Moreover, it separates the roles between different organizations in the system between simple users and governing ones. Based on the roles between organization, access-control rules can be implemented through chaincode, which we will discuss in the following section.

Other two aspects that are features of Fabric are separate channels which separate data between groups of nodes and leads to privacy among business partners. Channels share the same copy of the ledger, with separate chaincodes accessible only to its participants. This allows the development of any type of private relations between organizations, exactly as socio-economic rules dictate. Additionally, one other aspect is the ability to store private data within the same channel. This can further implement viewing rights on different levels over organizations' data. As the designed system does not implement or benefit significantly from this featurs we will not discuss them in detail here.
Ultimately, the ledger data can be encrypted via file system encryption on each peer node, while the data in transit is encrypted using TLS.

5.1.3 Consensus

Consensus is responsible for confirming the correctness of all transactions in a block and agrees on an order between them. It must guarantee two properties:

- *Safety* - each node is guaranteed to execute the transactions in the same way. In other words, given the same inputs, nodes among the network should output the same result.
- *Liveness* - assuming that there is a communication channel, every non-faulty node eventually should receive the submitted transactions.

In Hyperledger Fabric, consensus is a complex process, that is present in the transaction flow process described in the Background chapter. This process starts by making sure the set endorsement policies are met, which consists of checking if the transaction proposal is well-formed, if it has not been submitted in the past, thus preventing replay-attacks, checking the signature validity using the MSP and checking if the submitter has the right to invoke the transaction. Then the proposal is checked by the application and if the endorsements are valid, the ordering service orders the transaction. After this, transactions are validated again and committed by peer nodes.

We can conclude that consensus is reached in systems that have nodes acting honestly. It is worth mentioning that depending on the type of endorsement policy, the system's security varies. For example, if the endorsement policy requires half of the nodes to agree on a transaction, then the network is susceptible to 51% attacks.

One great benefit of Fabric is that it allows plugable consensus, thus the network administrators can decide on the tradeoff between performance and security depending on the network type.

5.2 Design security

When analyzing security aspects of the designed system, we assume the blockchain data corresponds to the one in real life and participants do not fabricate the data inputted to the system. The security aspects of human-blockchain interface are outside of the scope of this project. Additionally, another assumption is that a sufficient number of honest peer and orderer nodes are operating on the network. With these assumptions and depending on the endorsement policy, the designed system ensures the following security properties:

- Privacy from outside world - as participants need to have a valid certificate provided by MSP services, only the supply chain participants allowed to access the network can have access to the designed system and data.

- Asset traceability - because any operation performed on the ledger is recorded and saved in the asset's state, one can track items and shipments up to its creation on the blockchain.
- Transparency - having access to the state history of an asset, it is possible to know if products are new, returned or have been involved in any type of disputes. Also, there is a layer of transparency and trust between system participants because of the fact that the blockchain is permissioned and identities are known by administrating parties.
- Soundness - because of channel's endorsement policy, which requires the majority of organizations to agree on the outcome of a transaction, one can only invoke valid transactions which operate within the set business rules. Therefore, no invalid data can be accepted in the system as long as there is a majority of honest peer nodes.
- States validity - by performing checks on the validity of the operations both in the OperationsContract and StateContract through the state machines, the system ensures that items, shipments and disputes can not end in illegal states.
- Restricted data modification - even though the chaincodes offer functionalities to update the data on the ledger, they are made private and can be accessed only through proper operational use of OperationsContract. Therefore, in order to change an asset it must be involved in a valid transaction which takes into account access and business rules.
- ID uniqueness - as the IDs of the logical entities are generated through the chaincode's OperationsContract and because of transaction atomicity, the system ensures that no two different logical entities can have the same ID.
- Order validity - by checking the MSP ID, the system ensures that the ordered items are available to be sold, that they belong to the same organization and that the buyer has provided a payment proof.
- Timely delivery - because the expected delivery time is calculated on-chain, any delay in the delivery process can be automatically detected.
- Policy confidentiality - the bloom filter implementation of shipment policy whitelist and blacklist, ensures that it is impossible to know what the policy consists of. Therefore the policy rules are known only by the selling organization at the moment of shipment creation. In case of returns, the return shipment gets the same policy as the delivery shipment, thus the buyer does not find out anything about the delivery rules imposed by the seller.
- Accountability - organizations can not perform actions on behalf of other participants. Because of the permissioned nature of Fabric, for every operation, the identity of the caller is checked and validated and the transaction is signed by him. Using the Public Key Infrastructure of Fabric, it can be proven that the corresponding transaction has been submitted by him.
- Shipment validity - it is impossible for organizations to create shipments without having a valid buying contract or dispute for it. Therefore sending unsolicited shipments and requesting money for them becomes impossible.
- Return validity - as buyers are the one that choose the method of settlement they want for the dispute, it is always possible to return the items of the shipment.
- Shipment privacy from intermediates - even if organizations participate in the delivery process of a shipment, they can not know who the final client is, or where it is coming from. Having access only to the DeliveryInstruction destined for your own organization, one can only find the shipment ID, and where it needs to send the shipment.
- Path validity - shipments can not be lost or end up in unwanted destinations as long as the path finding algorithm works properly and participants follow the given instructions. It is impossible to exchange a shipment with an organization that is not part of the path as each step is checked using the DeliveryInstructions. Moreover, shipments can not be processed by organizations which are blacklisted by the policy,

because every time the next instruction is computed, the policy filters all available organizations, thus ensuring that the rules specified by the seller are followed.

- Shipment tracking - organizations are responsible for the shipments they hold. This ensures that at any point in time, the location and holder of a shipment can be known by querying the Locations and Holders lists of it. If the shipment is sent to a wrong destination, the system will refuse to record this change, leaving the sending organization responsible for the mistake.
- Dispute validity - buyers can only open disputes on shipments they received by giving proof of the problem. Having access to history of shipment and items, faking a problem with the shipment becomes increasingly difficult.

Still, there are security aspects that can be improved. One of them is the fact that seller organization can refuse to examine the disputes, forcing it to stay in the ”Created” state and making the client unable to return the shipment or ask for a reimbursement. Similarly, the system can not force the seller to create and send shipments that have been ordered already. This is left as the responsibility of the network administrators, to punish and exclude organizations that do not fulfill their responsibilities.

Chapter 6

Implementation

In this chapter we will describe the designed system from the implementation's point of view. The first part will refer to how the Fabric network has been created, the development process of the chaincode, as well as the challenges that were faced during this process. Then will follow some technical details about the REST server implementation. The last will be the implementation process of the graphical user interface and the framework used for that.

6.0.1 Process description

The implementation process of the whole system consisted of three steps:

- Blockhain system implementation - started with building a Fabric test network, that would be able to deploy and invoke chaincodes. Then it followed by modifying the built network by modifying names and adding an extra node in order to get the desired network topology and accommodate the design. After the network was deployed, the development of the chaincode has begun by implementing the designed system specification. Functionalities were being tested from the terminal while they were being implemented. The process was considered finalized once it was possible to perform all the action flows by transaction invocation.
- REST Server implementation - the design and implementation of the system came as a necessity after finding out that client applications can not connect to the Fabric network. This component implements only the methods that are necessary by the front-end side. Therefore the process consisted by outlining the needed functionalities, implementing them one by one and testing them with CURL tool.
- Front-end implementation - started with setting up a template application, followed by building the main empty components. Then the pages were being developed one by one until the action flows were possible to complete from the application.

6.1 Dispute Resolution Fabric system

6.1.1 Network Deployment

Prerequisites

The network setup consisted in following the test network setup tutorial from the Hyperledger Fabric documentation page [32]. For this, a series of prerequisites are necessary to be installed, such as:

- Git
- cURL
- wget
- Docker and Docker-compose

- NodeJS and NPM
- Golang and Python programming languages

This is followed by cloning the Hyperledger fabric-samples [33], which contain the necessary binaries and pre-made projects that can help in the learning process of Fabric. One of the most important things from the fabric-samples repository are the platform-specific binaries and configuration files which are saved in the /bin and /config directories of fabric-samples. Finally, the Hyperledger Fabric docker images are pulled.

Setup

As mentioned earlier, this project was built on top of the Fabric 2.0 test-network project by modifying the necessary configuration files and scripts in order to adapt the network to our needs. Every node is running in a separate docker container, just as the deployed chaincodes.
The network setup is strongly dependent on several configuration files. Among them are:

- configtx.yaml, which has channel transaction files and configuration information for building the genesis block. It is split in several sections:
  - Organizations - which hold the details about individual organizations and hold their identities that are referenced in the rest of the file. The identities contain the organization name, MSP ID, the directory that contains the MSP configuration and the read, write, admin and endorsement policies for this organization. Finally, the host and port of the anchor peers or orderer endpoint are specified, which get encoded in the genesis block. They are used for communication between nodes in the gossip protocol. Here we define one orderer organization OrdererMSP and three peer organizations: ProdMSP, DistMSP and RetMSP.
  - Capabilities - which define the capabilities of the Fabric network. These receive the profiles defined in the next sections.
  - Application - application defaults.
  - Orderer - the details regarding the orderer parameters. This includes the orderer implementation configuration, TLS certificates or block property configurations.
  - Channel - define the values to encode into a config transaction or genesis block for channel related parameters.
  - Profiles - which describe the organization structure of the network. Here we specify that the consortium is composed of the producer, distributor and retailer organizations and what kind of capabilities does the orderer have. Also, we define the channel configuration, capabilities and which organizations are part of it. For this proof of concept, one channel for the whole network is considered, as the network is composed only of three peers and there is no reason in having separate channels.
- crypto-config-*.yaml, that has the orderer and peer informations for generating their certificates. These files are required only if cryptogen tool is used for certificate generation. Here the organizations are defined with their name, domains and number of users.
- fabric-ca-config-*.yaml, that has the Certificate Authorities parameters. These files are required only if the organizations' certificates are using certificate authorities instead of the default cryptogen tool.
- docker-compose-*.yaml, these files contain the docker configurations of the containers, such as addresses, domains, ports, path to the genesis block, chaincode ports and other variables important for a correct network setup. These files are also important for setting up the certificate authorities containers.

The creation and deployment process of the network is done by executing commands from the network.sh script. In order to bring up the network, the network.sh up command should be executed, which performs the following actions:

1. Create the keys and certificates for the peer and orderer organizations. This can be done using the cryptogen tool from the binaries folder, by consuming configuration files crypto-config-*.yaml. It is also possible to bring the network up using Certificate Authorities by using the -ca flag. This uses the fabric-ca-config-*.yaml files. Both approaches generate MSP folders and crypto materials.

2. Generate the system channel genesis block, using configtxgen tool from the binaries. This uses the configtx.yaml file previously described.

3. After which the crypto material for organizations and the channel artifacts have been created, the network can be brought up. This is done by using the docker-compose-*.yaml files to create containers for network participants and the respective Certificare Authorities in case it is necessary.

After this step, the network is up and running, but no channels exist between the participants. For this, the ./network.sh createChannel command should be run, which executes the channel creation script. This uses the configtx.yaml file to create the channel creation transaction, and transactions that update the peers to be anchor peers. It then uses the peer binary to create the channel and join the organization peers to it, making them anchor peers.

Now there exists a channel between the organizations, but there is no chaincode deployed on it. To deploy a chaincode, the ./network.sh deployCC command must be run, which:

1. Installs the chaincode dependencies such as Fabric contract API and packages the source code into a chaincode.

2. After the chaincode is packaged, it can be installed on the peers and it must be installed on every peer that will take part in the endorsement process using the peer binary. After the install process, the chaincode will be build on each peer.

3. The approval process of the chaincode definition follows, in which the package ID and the version of the chaincode is used. Every time a new version of the chaincode is built, the version number should be incremented. In this step, one can specify the custom endorsement policies associated with each chaincode, otherwise the default option will be set. The default endorsement policy is a majority of nodes, which in our case would mean that 2 out of 3 organizations should endorse a transaction in order for it to be considered valid.

4. After enough organizations have approved the chaincode definition, one peer node can commit it to the channel. The commit transaction will pass only if the majority of nodes have it approved.

5. Now the chaincode is committed and ready to be invoked. If necessary, the initialization method of the chaincode should be invoked first.

With these steps behind, a fully working Hyperledger Fabric network with 3 peer nodes and one orederer is up and running. The implementation of the ordering services is solo, meaning that there is only one node taking part in the ordering process. This configuration is simple to set up and manage and is suitable for development environments. Nevertheless, in larger production networks it can become a bottleneck. At the same time, having only one ordering node, puts the network and production data under single point of failure risk.
Initially, the system has been implemented using cryptogen tool for creating certificates and crypto material, but after the implementation process was complete, Certificate Authorities were used. This choice was made because the connection to the network from the REST Server is easier and more documented if the network is set up with CAs. Moreover, in a production environment only Certificate Authorities are able to be used.

6.1.2 Chaincode Implementation

The two smart contracts have been implemented in the same chaincode. This is because the two contracts need to communicate between each other. Also, if the contracts are packed in the same chaincode, they share the same world state, making it easier to invoke each other's methods. It is also common to pack contracts together based on their domain and functionality, and it would make sense to separate them only in case they need

separate endorsement policies. In this case, the contracts are tightly coupled, and they do not require different endorsement policies. For these reasons, it makes more sense from the performance point of view to develop both contracts in the same chaincode.
The smart contracts have been implemented in Go programming language [34] using Visual Studio Code.

Data Structures

All the data structures and their fields are implemented according to the specifications shown in the Design chapter. In order to be able to operate with them properly, their definitions and fields have been made public and contain specifications on how to convert them to the JSON format.
As mentioned previously, data structures that do not contain IDs, are not stored on the ledger. This makes the reference between entities of two types, which can be seen in Figure 6.1:

- ID reference - in which one entity points to the other by keeping its ID. This is done between entities that are both stored on the ledger.
- Structure reference - when the entity is contained in the parent entity. This is implemented when the child entity is not stored on the ledger.

Figure 6.1: Types of relations between entities

Because of the inability to have pointers in JSON structures, ID referencing was necessary to implement in order to avoid circle dependencies and to reduce the size of data structures. Nevertheless, the Location and Policy structures are contained within their parent structures as they do not make sense outside that scope.

The *Location* longitude and latitude are implemented as floating point numbers. A negative/positive longitude would mean Western/Eastern hemisphere, while a negative/positive latitude means Southern/Northern hemisphere. This format makes it easy to store the coordinates and calculate distances in this proof of concept. The Location is stored as a whole structure within other structures.

The *Policy* data structure contains two bloom filters representing the whitelist and blacklist. Because all data structures need to be converted to JSON, the bloom filter implementations need to be marshaled to byte before converting them to JSON. In the bloom filter implementation, the [35] library is used. This implementation allows Marshal and Unmarshal operations. If the bloom filter is stored as a data structure, the conversion to and from JSON fails for some reason. This brought the requirement to convert the bloom filter implementation to an array of bytes first, which is stored as a String, and then convert it back when reading it. For this reason the bloom filters are stored as strings.

The filter can be created either by manually giving the parameters such as the number of hash function and the total number of elements, or it is possible to give the total number of elements and the maximum probability of False Positives allowed and leave the rest to the library implementation. In this way, a maximum FP probability of 0.001% has been used, with the total number of elements being:

$$3 * N + 10$$

where N is the total number of currently registered organizations on the network. The number 3 comes from the fact that each OrgProfile has three possible elements that can be added to the bloom filter: MSP ID, orgType and country. If we assume worst case, these elements would all be different, resulting in three times more distinct elements than the total number of organizations. The 10 is an extra margin added just to make sure that even if new organizations get added to the network, the policy will be able to function as intended.

In the *InitLedger* function, which is called before any other transaction, the IDs of all entities are initialized with 0. Also, the OrgProfiles of the three registered organizations are saved to the ledger, together with the total number of current organizations equal to 3. The function $\mathrm{getEntityNextID}$ has been implemented to retrieve the ID of the next entity. The entity is specified as a parameter, the ID is returned and the next ID is incremented no matter if the returned ID will be used or not. Entities are saved on the ledger with the key $\mathrm{entityID}$. For example, the item with ID 3 is saved under the key $\mathrm{item3}$. This allows to retrieve all entities of one type by using the function $\mathrm{GetStateByRange(startKey, endKey)}$.

In the calculation of delivery time, the current time is considered the timestamp of the submitted transaction. This is necessary because local times are different for each peer, while the transaction has a fixed timestamp, resulting in a deterministic execution. To calculate the number of days for delivery, an average of 100 km per day are considered.

All methods perform the necessary checks on the identity of the caller. This is done by retrieving the MSP ID and checking if it corresponds to the requirements for that specific actions.

Implementation Difficulties

Database Relations One of the first implementation difficulties was the fact that Hyperledger Fabric does not operate like a relational database but as a (key, value) database. This implies that entities can not point to each other. In order to associate two entities between them, they have to have as fields in their structure the key (ID) of the other entity. This ends up in forcing the development of uni and bidirectional relations, by storing string IDs in all the entities. For example, in Figure 6.2 the dispute entity points to the shipment that is disputed, which in turn points to the contract that it is created for. The contract contains the IDs of all the items that are ordered in it, and all entities except the dispute contain MSP ID of an organization.

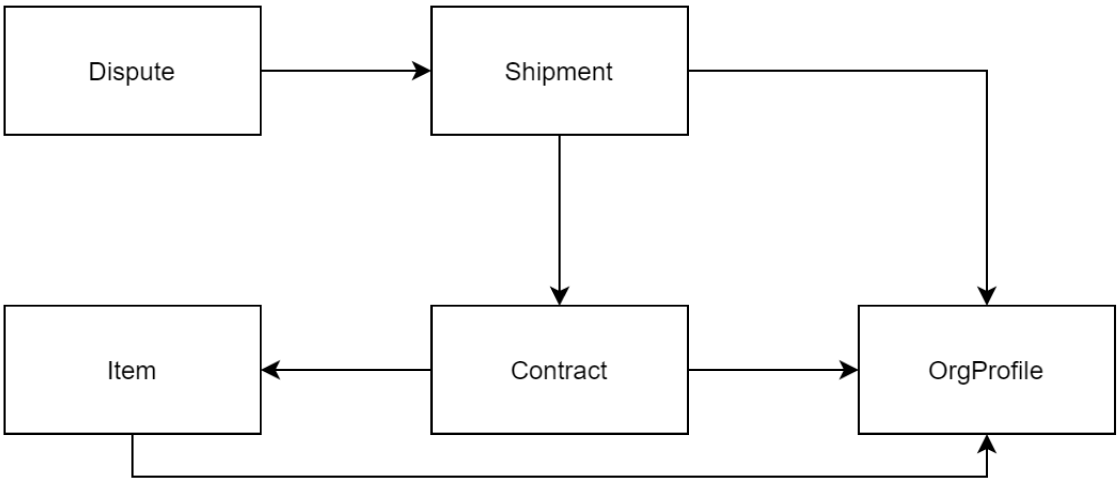

Figure 6.2: Exemplification of entity dependency

This makes the querying process more difficult. For example, in order to retrieve the items that are in a shipment, by having only the dispute ID, one would have to perform 4 different queries to get the dispute, then the shipment, followed by the contract and ending with all the items within it. In a relational database this operation could be executed more efficiently. Similarly, in order to find the shipment that contains a contract, one would have to iterate through all the shipments and check which one has the respective contract ID. This results in very inefficient queries and checks. In a relational database, any relation would be bidirectional, thus making it trivial to navigate between data objects. Even though this problem is known, this system is not designed to be as efficient as possible with Fabric database.

Risk of Data inconsistency The aforementioned problem can also result in problems with data integrity. This is because entities point to each other by storing their string IDs, and not by pointing to the actual entity. In case the entities are changed or removed from the ledger, it would result in consistency problems, as structures would point to non existing objects. To make up for this, the developer has to automate the update operations through the chaincode functionalities, not only making the development process cumbersome, but also making the project prone to bugs and hacks. As the current system is a proof of concept, the design does not take into consideration a solution for this problem.

Deterministic Bloom Filters Because the execution of the chaincode has to be identical for all the peers, no randomization in the operations are allowed. This implies that the bloom filter implementation and execution has to be deterministic for every endorser. As bloom filters contain multiple hash functions, usually initialized with a seed, the libraries that initialize the hash functions with random seeds are not suitable. This includes libraries that use the local time as seed, as the time for every endorser differs.
The current implementation produces the hash functions from the data itself, therefore the functions will be deterministic for all the peers as long as the data will be the same.

Inability to check changes to World State made by current transaction Initially, the idea of communication between StateContract and OperationsContract was that each of them would consult the other when performing changes. For example, before item state machine would transition to the ”Shipped” state, it would check in the OperationsContract if the Shipment has been created. Similarly, the OperationsContract would query the current state of entities to decide the validity of operations. This design was supposed to keep both smart contracts' functionalities public and independent.
During the implementation process, this choice turned out to be a blocking point. The problem lies in the fact that it is impossible to check the changes to the world state that the current transaction is going to make. To understand the problem better, let us consider the shipment creation transaction. Within the same transaction, the Shipment entity must be created, and items' states must be updated. In order for the StateContract to check if the change from ”Sold” to ”Shipped” is valid, it must check if a Shipment containing the items was created on the ledger. As the shipment creation is supposed to be done in the same transaction, the changes are only made to the world state, and are going to be committed at a later point in time. Therefore, it is impossible within the same transaction to create the Shipment in a contract and check its creation from another contract.
This made it necessary to make the StateContract's funnctionalities private, use it exclusively inside OperationsContract and leave only the getters public. In this way, instead of implementing security checks in the StateContract and trusting the implementation, the trust is shifted towards the OperationsContract implementation. Therefore, the state changing methods, only verify if the transition makes sense from the state diagram's point of view, without performing further more elaborated verification. In this way, OperationsContract is totally responsible for checking the validity of operations and uses the StateContract only as a way of storing the state machines.

6.2 REST Server

As the REST Server has a simple and clear design, the implementation process was straightforward as well. The server is implemented with NodeJS and ExpressJS [36] was used as Web framework. In order to connect to the Fabric network, `fabric-network` and `fabric-ca-client` have been used.

As the server needs to remember and change the current active organization as requested, this is done by keeping the path to different connection profiles as variables. When the current active organization needs to change, the path to the current connection profile changes accordingly.
Because the implementation complexity of GET /get* methods is low and the majority of them have a high degree in similarity (e.g. /getItem?id=0 and /getContract?id=0), it has been decided to implement a generic /get:entity method, that would pass the parameters to the Fabric network accordingly. Even though this defies the single responsibility principle, for this proof of concept it improves readability and reduces the size of these methods by around 14 times. In this way, the :entity parameter can be checked and the right transaction can be invoked depending on its value.
the methods are implemented in the same way: firstly the connection profile is read depending on the current active organization, then the wallet for the organization is retrieved and the presence of the admin identity is checked. Afterwards, the gateway to the Fabric is created, followed by the retrieval of the channel and chaincode. Then, depending on the method, the transaction is invoked with the right parameters and the result is returned in the response body.
As this component represents just a middleman between the front-end and Fabric network, it does not implement any logic and is responsible only for parsing the request and response accordingly.

6.3 Front-end application

The front-end application has been implemented using the Shards Dashboard [37] as initial template. For this, ReactJS library [38] has been chosen, while the components are provided by Shards React [39] library. Axios library was used to perform calls to the REST Server, whose IP was available in the whole project tree by saving it as an environmental variable in the ./env file.
In the whole application, it was necessary to know at any moment which organization is the current one, which decides the content shown on the page. For this, the React Context has been used, which removes the need to pass props down the project tree. The context is known in every child element of the tree, and together with the context, the context changing function is passed as well. This allows the component responsible for changing the current organization to be able to change its value. Every time the current organization is changed, the page gets refreshed in order to repopulate the components with the correct data.
As mentioned in the Design chapter, there is one layout containing a Navigation Bar, Side Menu and Content area. The content area can be made up of one of the following views:

- ItemsView - a view in which the organization can see its items, other available items, create items and create orders.
- OrdersView - a view where organizations can see lists of orders ordered for and by him. These are implemented using HTML tables.
- ViewOrder - view in which information about the Order, corresponding shipment (if any) and included items is shown. This view is shown when the button to view the order is clicked.
- ShipmentsView - view in which lists of created and received shipments are shown.
- ViewShipment - a view containing details about shipments and offers functionalities to manage them. This view can be accessed from the ShipmentsView by clicking the view shipment button.
- DisputesView - a view containing disputes opened by and for the current organization.
- ViewDispute - view in which dispute information can be seen together with the disputed items and shipment.

In the whole application, item details can be seen by clicking the eye button next to it. This will create a modal on top of the application where item details are shown together with their state history.
States are represented visually as colored badges, where every color corresponds to a specific state. The themes and the associated states can be seen in Figure 6.3.

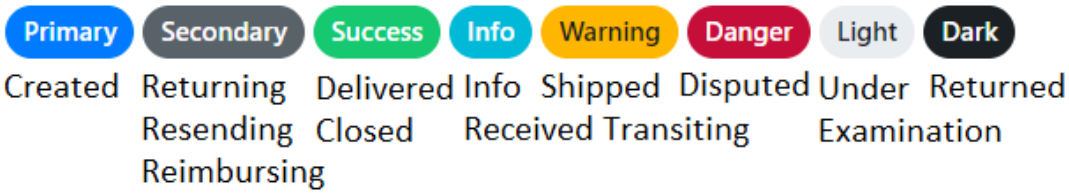

Figure 6.3: State representations

In order to show which data corresponds to the one on the ledger, it must be kept locally for each individual organization. For this, the local storage at the client side has been used. Every time an operation would be attempted but it would fail, the local storage would be updated accordingly. When the data from the ledger will be fetched, there will be a mismatch between the local and ledger data, resulting in an ”Unconfirmed” status of that asset. For example, if an organization tries to place an order, it provides payment proof but the connection to either the server or Hyperledger network is broken, this item would be saved in the local storage of this organization. When the connection would be established again, items from the ledger will not match with the local ones, therefore the corresponding item will be ”Unconfirmed”.

Chapter 7

Validation and Testing

7.1 Testing

The system and its parts have been tested thoroughly during and after the development process. The development and testing has been done in the following environment:

- Host Machine:
  - Processor: Intel Core i7-7700HQ, 2.80 GHz
  - RAM: 8GB
  - Operating System: Windows 10 Home
  - System Type: 64 bit OS
  -
- Fabric Network and REST Server: 64 bit Ubuntu Virtual Machine, 4700 MB base memory, Virtual Box
- Front-end application: Host Machine

The request would be made on the host machine in the front-end application, it would target the VirtualBox's virtual machine IP, on the port 8080. The REST Server listens on localhost:8080 and forwards the calls to the Fabric network through the Gateway.

The testing of the Fabric system has started right after setup of the network. The ./network.sh script responsible for deploying the network contains operations that are meant to test whether the deployment steps have been performed with success. For example, in the chaincode deployment script, after the installation process, the peer lifecycle chaincode command is used to query whether the peers have successfully installed the chaincode. Also, after organizations approve the chaincode definition, the commit readiness is checked and the commit is queried afterwards using the command previously mentioned.
The chaincode functionalities were tested individually step by step after their implementation. This was done from the terminal using the peer chaincode invoke command by setting the right environment variables to act as one of the organizations and by providing the right parameters to the command. An example of such command is:
peer chaincode invoke -o ordererAddress:Port –ordererTLSHostnameOverride ordererDomain –tls –cafile ordererCAFile -C channelName -n chaincodeName –peerAddresses peerAddress:Port –tlsRootCertFiles peerCAFile -c '"function":"functionName","Args":[functionArgs]'

The REST Server methods implementations have also been tested individually. This was done using the cURL tool from the terminal. For example, a command used to test the create item POST method looks like the following:
curl -d '"description":"value1"' -H "Content-Type: application/json" -X POST http://localhost:8080/createIter
while one for retrieving an item is:
curl localhost:8080/getItem?id=0

7.2 System Validation

The validation process has been performed through the front-end application. To validate it, the delivery and dispute flows have been performed in-app, which demonstrates the presence of all necessary functionalities. The full screen captures can be seen in the Appendix.

The user can create items, orders and view the available items from the ”Items” tab shown in Figure 7.1.

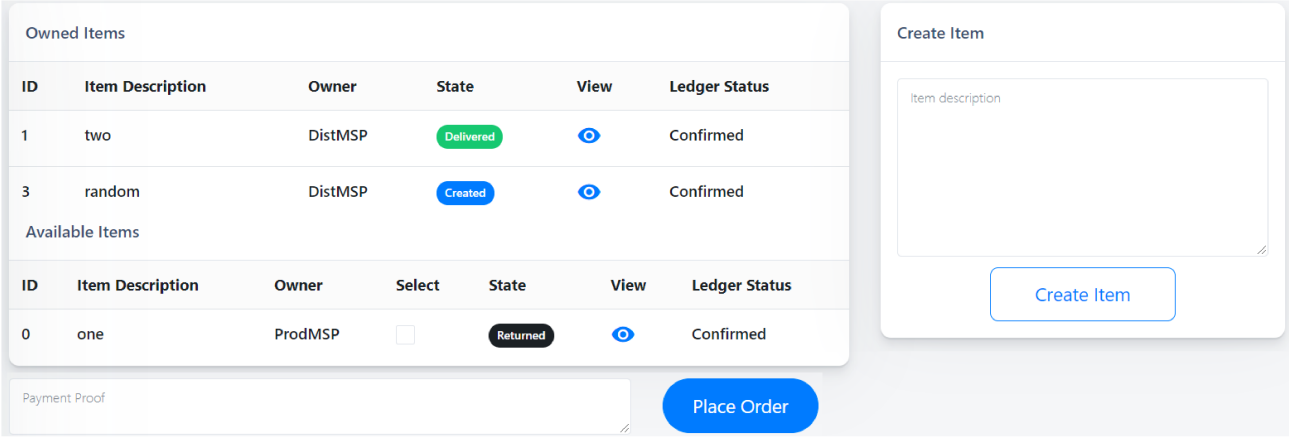

Figure 7.1: Items tab screen capture

After placing the order, the seller organization can view the order and create the Shipment for it, using the functionality shown in Figure 7.2.

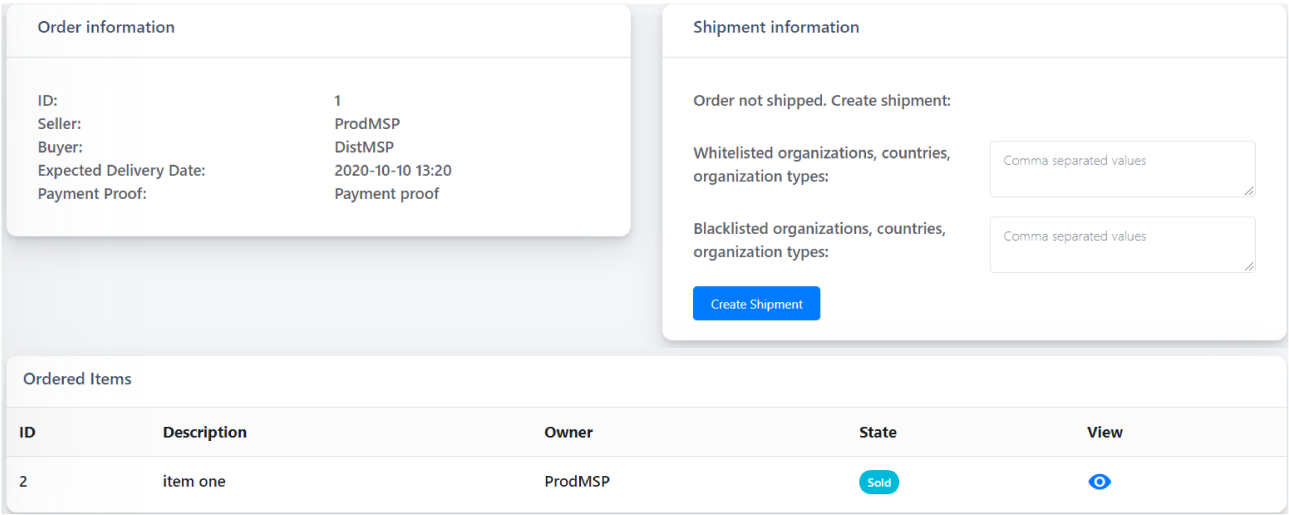

Figure 7.2: Shipment creation screen capture

Once the shipment is created, the holder can initiate the delivery process by asking for a DeliveryInstruction, which is shown in Figure 7.3.

In order to receive a Shipment and simulate the scanning of the QR code, the organization must input the instruction ID in the designated field for instructions in Figure 7.4. Moreover, the orders and disputes views have the same structure as the shipments view, with the exception that they lack the Shipment receival functionality. These can be seen in the Appendix.

After the shipment is received, the buyer organization can open the dispute by selecting the disputed item and providing problem proof in the screen from Figure 7.5.

At any moment of the flow, it is possible to view item, shipment and dispute details and states. Item and shipment information can be seen in Figure 7.6.

Also, changing the current active organization or editing its profile is possible by using the top-right toggle buttons, which trigger a modal.

Figure 7.3: DeliveryInstruction screen capture

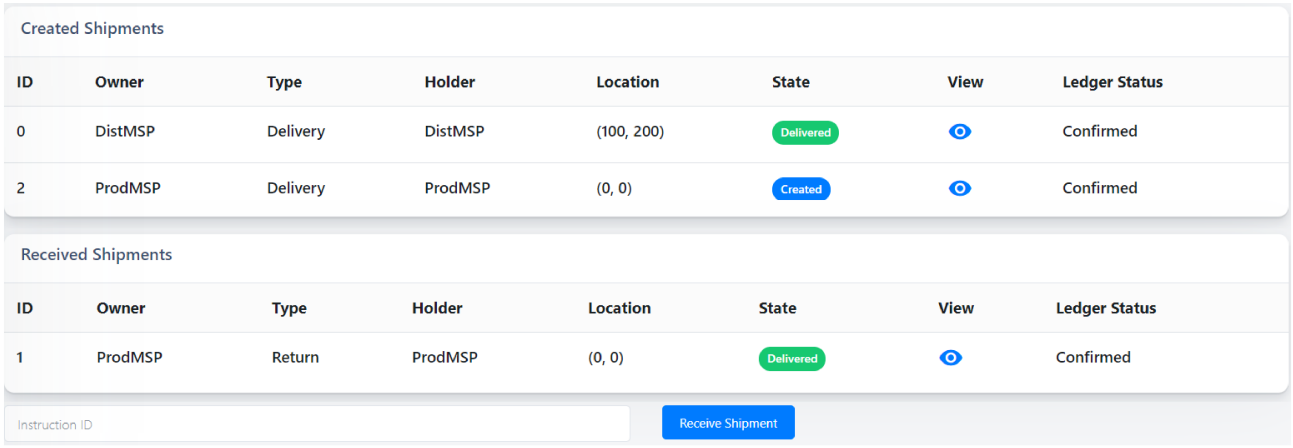

Figure 7.4: Shipments view screen capture

7.3 Graphical User Interface guide

7.3.1 Delivery Flow

In order to perform the delivery flow within the web-app, the following steps must be followed:

1. Using the top-right button, change the current active organization to the desired one and move to the Items tab.
2. Create one or more items by providing item descriptions.
3. Change the current active organization to any other organization, provide payment proof, select interested items and create order.
4. Change current active organization to the seller organization and go to Orders tab.
5. View the newly sold order and create a shipment by providing delivery rules as blacklist and whitelist.
6. Move to Shipments tab, view the shipment held, compute new instruction and save the ID and destination organization.
7. Change current active organization to the destination organization, move to Shipments tab and receive the shipment by providing the saved instruction ID.
8. Repeat steps 6 and 7 until shipment is delivered.

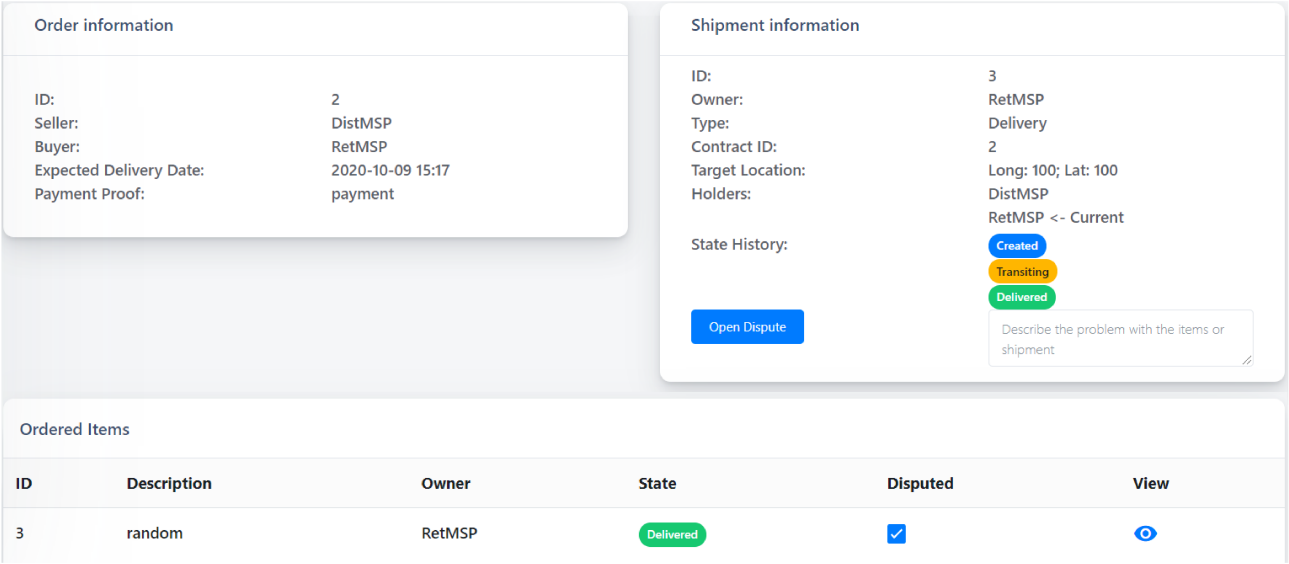

Figure 7.5: Open Dispute screen capture

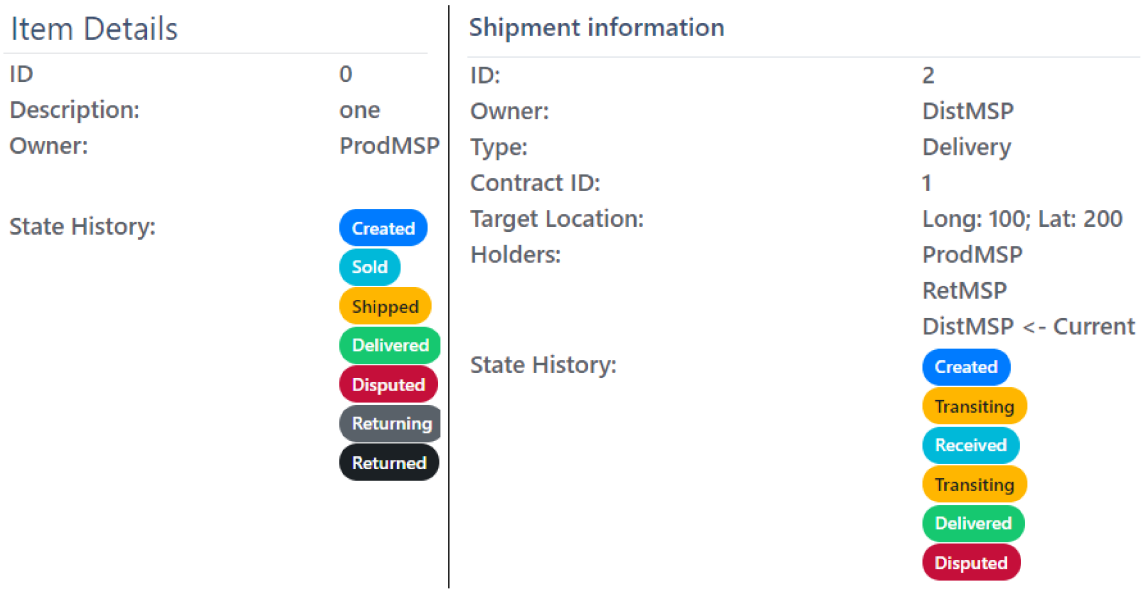

Figure 7.6: Item and Shipment information screen captures

7.3.2 Dispute Flow

In order to perform one of the dispute flows, the following steps are required:

1. As the buyer organization, view the received shipment and create a dispute by picking at least a disputed item and providing problem proof.
2. As the seller organization, go to Disputes tab and view the dispute.
3. As the buyer organization, view the dispute and pick a settle method.
    - For Return:
        - Pick ”Returning” option.
        - Perform steps 6 and 7 from delivery flow until shipment is received by selling organization.
    - For Resend:
        - Pick ”Resending” option.
        - As the seller organization, view the dispute and pick items to be resent.

  - Perform steps 6 and 7 from normal flow until shipment is received by buying organization.
- For Reimbursing:
  - Pick ”Reimbursing” option.
  - As the seller organization, view the dispute and provide reimbursement proof.

Chapter 8

Conclusion

Distributed ledger technology is a revolutionary innovation with huge potential to improve many current existing systems by making them more transparent, secure and efficient.

In this dissertation, the main characteristics of blockchain technology are reviewed, with emphasis on Hyperledger Fabric and the benefits brought by it. More importantly, the focus was put on the applicability of blockchain technology in the supply chain management domain.

For this, a Dispute Resolution system for supply chains has been developed using Hyperledger Fabric 2.0 blockchain. The designed proof of concept prototype facilitates the prevention of accidental or intentional system misuses, thus reducing the number of potential dispute situations between parties. One such example is the prevention of sending packages to wrong destinations by automating the delivery process. The system also serves as a common source of truth, which can be consulted in cases a dispute arises. Moreover, product traceability is achieved by having access to the history of holders, owners, locations and actions performed on an asset. This results in increased transparency for customers in the buying process within supply chain.
The dispute resolution process is facilitated by allowing parties to dispute delivered shipments and items, inspect the problem and ultimately settle the dispute on-chain. The proposed system along with transparency, immutability and traceability, also brings security to the supply chain. Participants in the network can be sure that no parties can bend the rules in their favor or circumvent imposed regulations which are imposed by the chaincode. Nevertheless, these benefits are achieved without sacrificing the privacy of participants, which makes it suitable for establishing private business relations.

Having the proposed system and the main objectives of this paper in mind, we can conclude that blockchain technology can have a great utility in the supply chain management. More specifically, the dispute resolution process in the supply chain can be assisted and improved by making use of the transparency, immutability and security which this technology brings.

Among the limitations of this system is the fact that blockchain technology is a relatively immature technology, which can result in resistance or doubt in its adoption process. Also, this technology shifts the trust from within the system to its frontier, but securing the blockchain-real world interface is a separate research domain, outside of the scope of this work.

As this project represents a proof of concept, future work might include the development of a more elaborate system, which would take into account the business needs and the topology of a real supply chain. Another direction of research can be the further automation of those decisions in the supply chain, which currently are done manually by its participants, such as settlement negotiation in a dispute.

Bibliography

[1] J. T. Mentzer, W. DeWitt, J. S. Keebler, S. Min, N. W. Nix, C. D. Smith, and Z. G. Zacharia, "Defining Supply Chain Management," *Journal of Business Logistics*, vol. 22, no. 2, pp. 1–25, 2001.

[2] mhugos, "Four Participants in Every Supply Chain | SCM Globe."

[3] F. Isik, "Complexity in Supply Chains: A New Approachto Quantitative Measurement of the Supply-Chain-Complexity," *Supply Chain Management*, Apr. 2011.

[4] H. Min, "Blockchain technology for enhancing supply chain resilience," *Business Horizons*, vol. 62, no. 1, pp. 35 – 45, 2019.

[5] Feng Tian, "An agri-food supply chain traceability system for China based on RFID blockchain technology," in *2016 13th International Conference on Service Systems and Service Management (ICSSSM)*, pp. 1–6, June 2016. ISSN: 2161-1904.

[6] S. Saberi, M. Kouhizadeh, J. Sarkis, and L. Shen, "Blockchain technology and its relationships to sustainable supply chain management," *International Journal of Production Research*, vol. 57, pp. 2117–2135, Apr. 2019.

[7] S. A. Abeyratne and R. Monfared, "Blockchain ready manufacturing supply chain using distributed ledger," 2016.

[8] "Ethereum Whitepaper."

[9] M. Wohrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pp. 2–8, Mar. 2018.

[10] R. Casado-Vara, J. Prieto, F. D. la Prieta, and J. M. Corchado, "How blockchain improves the supply chain: case study alimentary supply chain," *Procedia Computer Science*, vol. 134, pp. 393–398, Jan. 2018.

[11] M. P. Caro, M. S. Ali, M. Vecchio, and R. Giaffreda, "Blockchain-based traceability in agri-food supply chain management: A practical implementation," in *2018 IoT Vertical and Topical Summit on Agriculture - Tuscany (IOT Tuscany)*, pp. 1–4, 2018.

[12] C. G. Schmidt and S. M. Wagner, "Blockchain and supply chain relations: A transaction cost theory perspective," *Journal of Purchasing and Supply Management*, vol. 25, no. 4, p. 100552, 2019.

[13] S. E. Chang, Y.-C. Chen, and M.-F. Lu, "Supply chain re-engineering using blockchain technology: A case of smart contract based tracking process," *Technological Forecasting and Social Change*, vol. 144, pp. 1 – 11, 2019.

[14] Z. Liu and Z. Li, "A blockchain-based framework of cross-border e-commerce supply chain," *International Journal of Information Management*, vol. 52, p. 102059, 2020.

[15] Z. Wang, T. Wang, H. Hu, J. Gong, X. Ren, and Q. Xiao, "Blockchain-based framework for improving supply chain traceability and information sharing in precast construction," *Automation in Construction*, vol. 111, p. 103063, 2020.

[16] K. Salah, N. Nizamuddin, R. Jayaraman, and M. Omar, "Blockchain-based soybean traceability in agricultural supply chain," *IEEE Access*, vol. 7, pp. 73295–73305, 2019.

[17] P. Helo and A. Shamsuzzoha, "Real-time supply chain—a blockchain architecture for project deliveries," *Robotics and Computer-Integrated Manufacturing*, vol. 63, p. 101909, 2020.

[18] S. Wang, D. Li, Y. Zhang, and J. Chen, "Smart Contract-Based Product Traceability System in the Supply Chain Scenario," *IEEE Access*, vol. 7, pp. 115122–115133, 2019.

[19] "FEUP - Supply Chain Management with Blockchain Technologies."

[20] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," tech. rep., Manubot, Nov. 2019.

[21] "Blockchain Technology: Beyond Bitcoin (2016), Crosby, Nachiappan, Pattanayak, Verma & Kalyanaraman," Nov. 2017.

[22] "Ledger — hyperledger-fabricdocs master documentation." Available at https://hyperledger-fabric.readthedocs.io/en/release-2.0/ledger/ledger.html.

[23] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends," in *2017 IEEE International Congress on Big Data (BigData Congress)*, pp. 557–564, June 2017.

[24] "4 Different Types of Blockchain Technology & Networks," Apr. 2020.

[25] "Introduction — hyperledger-fabricdocs master documentation." Available at https://hyperledger-fabric.readthedocs.io/en/release-2.0/whatis.html.

[26] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–15, Apr. 2018. arXiv: 1801.10228.

[27] "Membership service provider (msp) — hyperledger-fabricdocs master documentation." Available at https://hyperledger-fabric.readthedocs.io/en/release-2.0/membership/membership.html.

[28] "The ordering service — hyperledger-fabricdocs master documentation." Available at https://hyperledger-fabric.readthedocs.io/en/release-2.0/orderer/ordering_service.html.

[29] "Peers — hyperledger-fabricdocs master documentation." Available at https://hyperledger-fabric.readthedocs.io/en/release-2.0/peers/peers.html.

[30] "Smart contracts and chaincode — hyperledger-fabricdocs master documentation." Available at https://hyperledger-fabric.readthedocs.io/en/release-2.0/smartcontract/smartcontract.html.

[31] "Fabric chaincode lifecycle — hyperledger-fabricdocs master documentation." Available at https://hyperledger-fabric.readthedocs.io/en/release-2.0/chaincode_lifecycle.html.

[32] "Using the Fabric test network — hyperledger-fabricdocs master documentation."

[33] "hyperledger/fabric-samples," Oct. 2020. original-date: 2017-06-20T00:15:53Z.

[34] "The Go Programming Language."

[35] W. Fitzgerald, "willf/bloom," Oct. 2020. original-date: 2011-05-21T14:18:41Z.

[36] "Express - Node.js web application framework."

[37] "DesignRevision/shards-dashboard-react," Oct. 2020. original-date: 2019-01-07T16:56:11Z.

[38] "React – A JavaScript library for building user interfaces."

[39] "Shards React."

Appendix A

Application Screen Captures

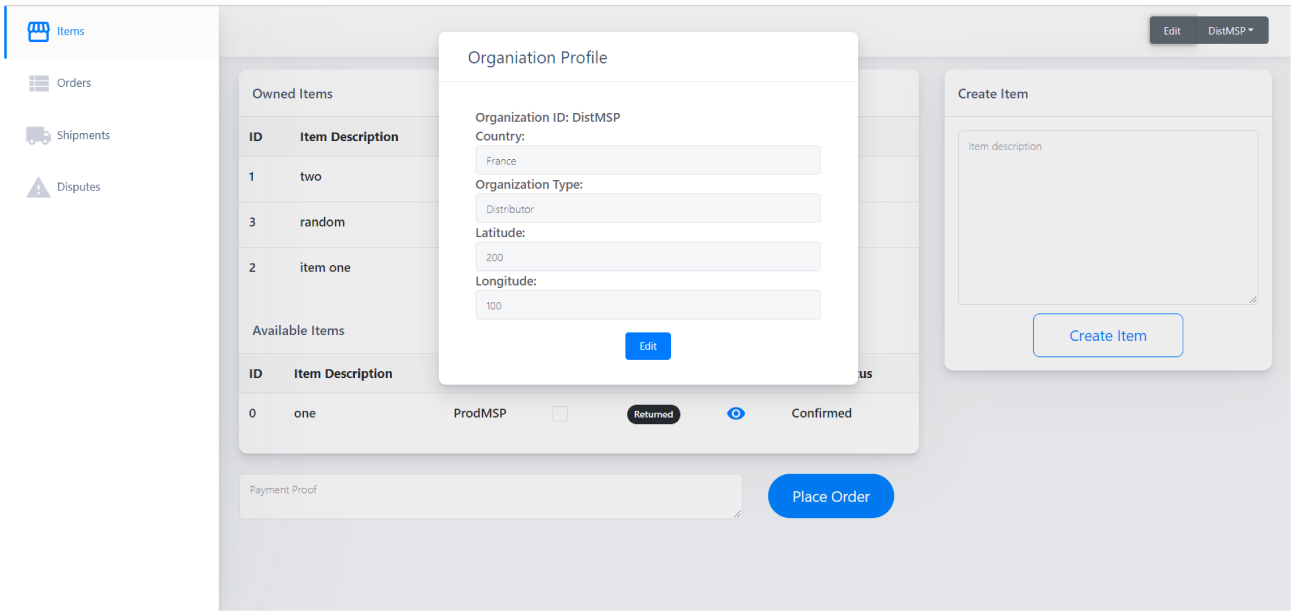

Figure A.1: Profile editor screen capture

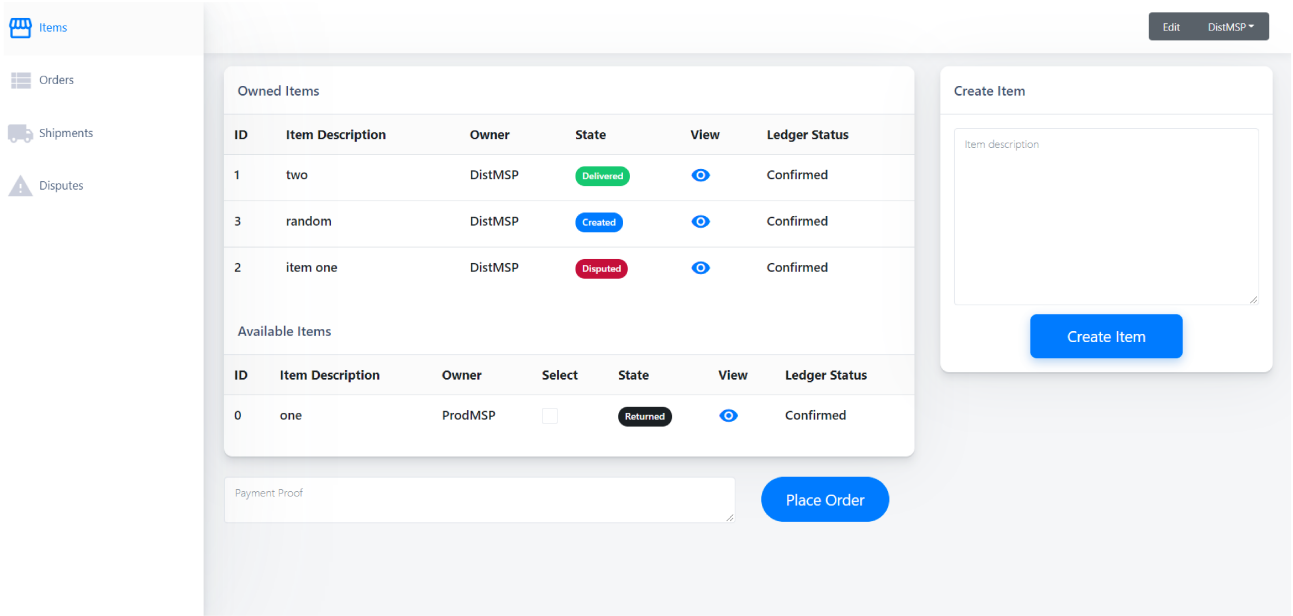

Figure A.2: Items View screen capture

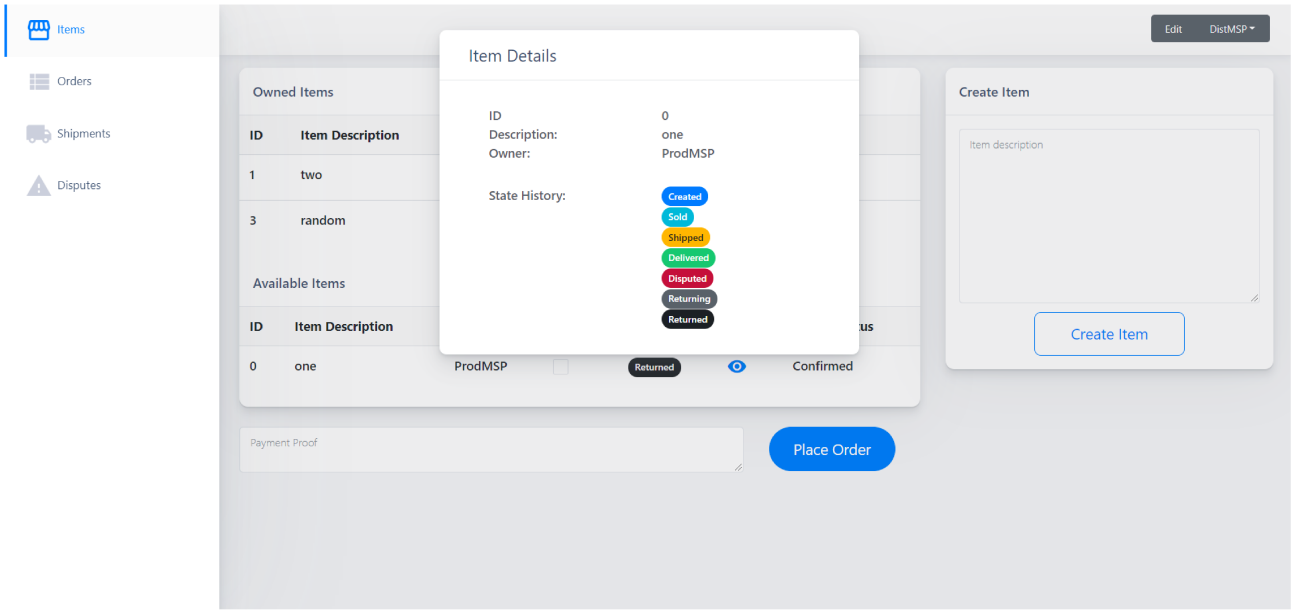

Figure A.3: Item details modal screen capture

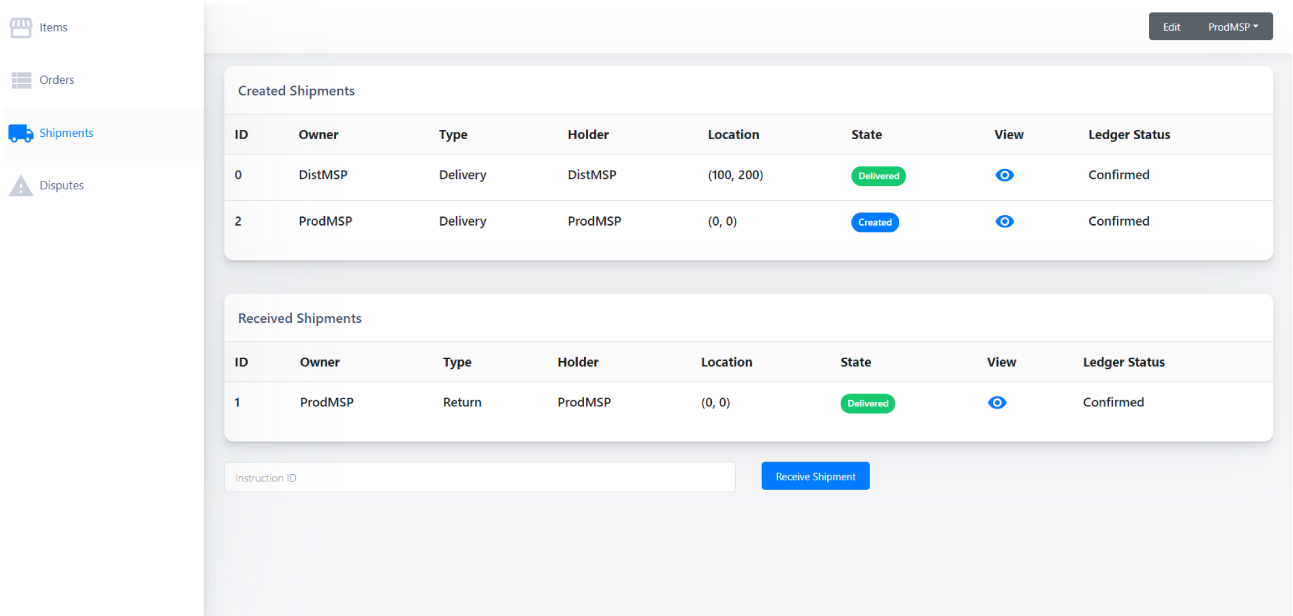

Figure A.4: Shipments View screen capture

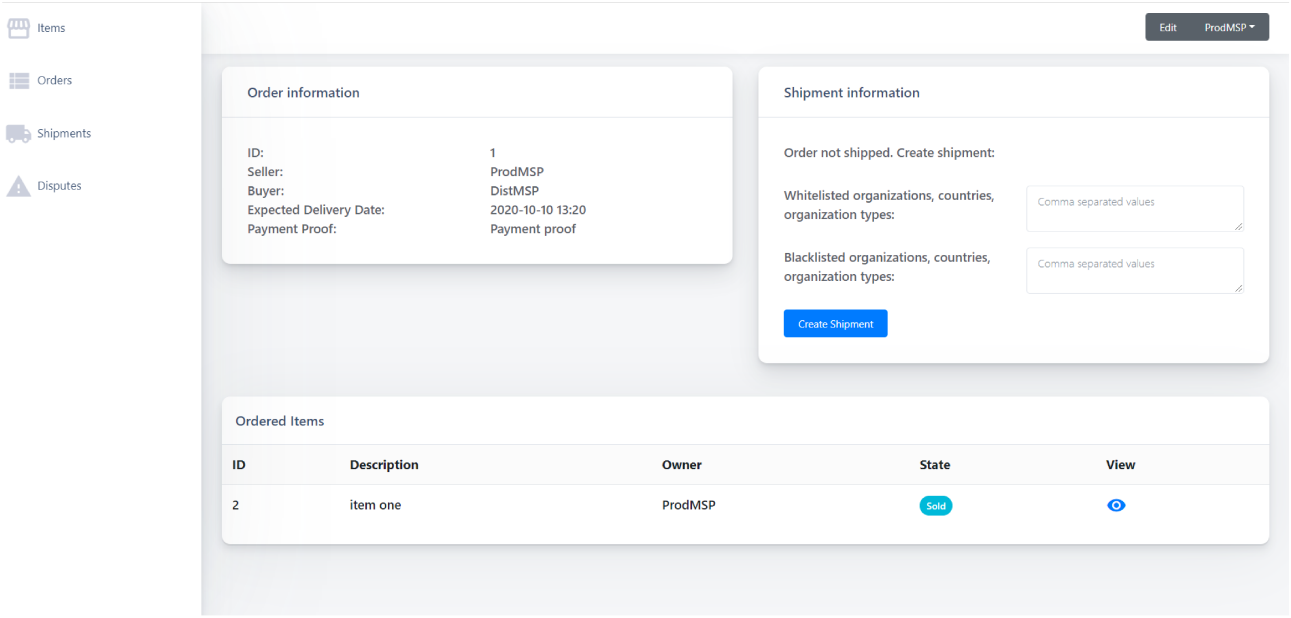

Figure A.5: Shipment creation screen capture

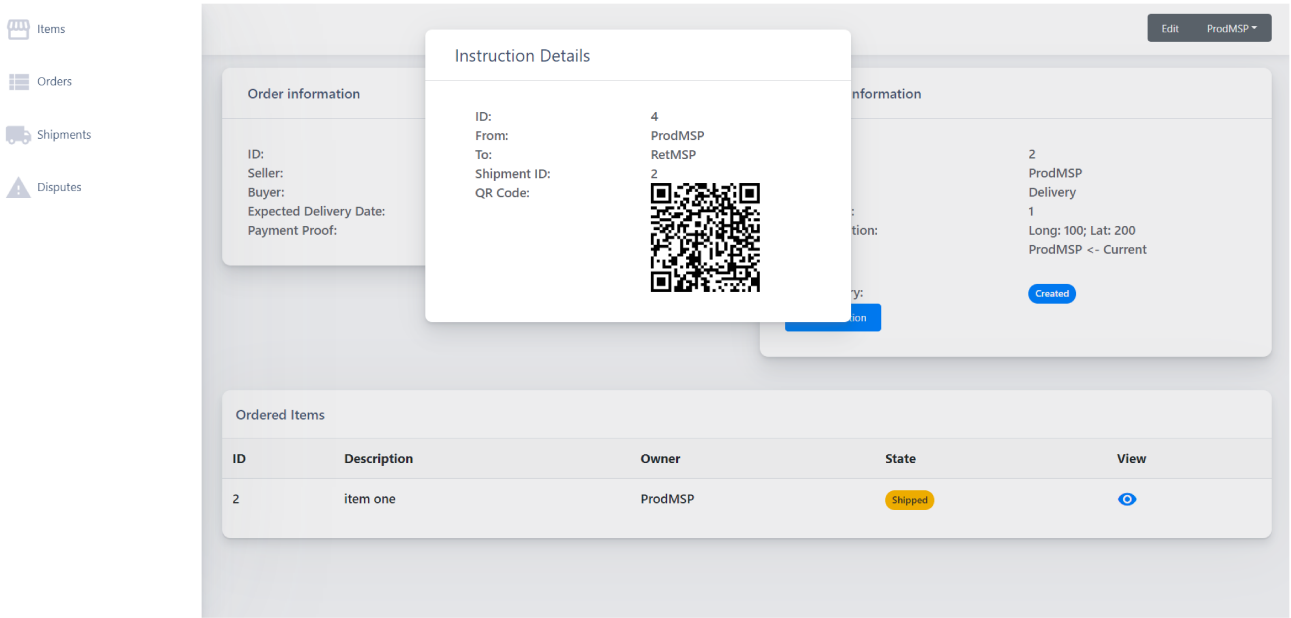

Figure A.6: Instruction details screen capture

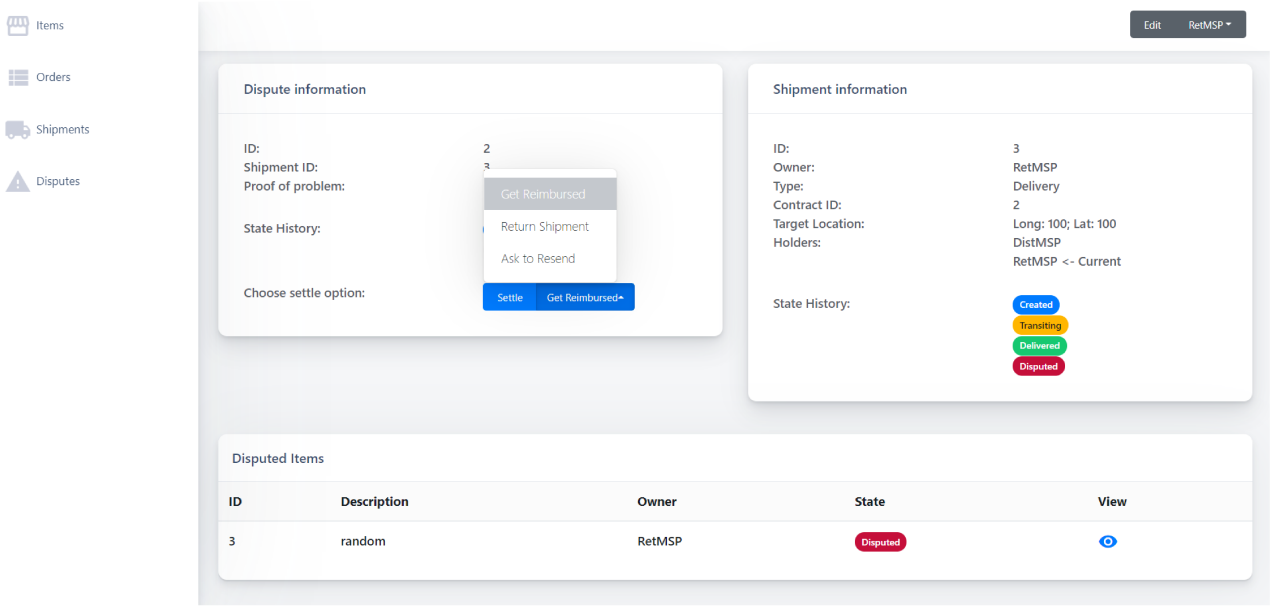

Figure A.7: Dispute Settlement process screen capture