Design and Implementation of a Blockchain Shipping Application

by

Maher M. Bouidani

B.Sc., Philadelphia University, Jordan 2015

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Applied Science

in the Department of Electrical and Computer Engineering

© Maher M. Bouidani, 2019
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Design and Implementation a Blockchain Shipping Application

by

Maher M. Bouidani

B.Sc., Philadelphia University, 2015

Supervisory Committee

Dr. Fayez Gebali, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Hausi A. Müller, Co-Supervisor
(Department of Computer Science)

ABSTRACT

The emerging Blockchain technology has the potential to shift the traditional centralized systems to become more flexible, efficient and decentralized. An important area to apply this capability is supply chain. Supply chain visibility and transparency has become an important aspect of a successful supply chain platform as it becomes more complex than ever before. The complexity comes from the number of participants involved and the intricate roles and relations among them. This puts more pressure on the system and the customers in terms of system availability and tamper-resistant data. This thesis presents a private and permisioned application that uses Blockchain and aims to automate the shipping processes among different participants in the supply chain ecosystem. Data in this private ledger is governed with the participants' invocation of their smart contracts. These smart contracts are designed to satisfy the participants' different roles in the supply chain. Moreover, this thesis discusses the performance measurements of this application results in terms of the transaction throughput, transaction average latency and resource utilization.

Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **iv**

List of Tables **vii**

List of Figures **viii**

List of Acronyms **x**

Glossary **xi**

Acknowledgements **xiv**

Dedication **xv**

1 Introduction **1**

1.1 Motivation 1
1.2 Problem Definition 2
1.3 Approach 3
1.4 Thesis Outline 3

2 Blockchain Fundamentals **4**

2.1 Public Key Cryptography 5
2.2 Blockchain Transaction 6
2.3 Hash Function 8
2.4 Consensus 9
2.4.1 Proof of Work 11
2.4.2 Proof of Stake 13

2.4.3 Practical Byzantine Fault Tolerance 13
2.5 Merkle Tree 15
2.6 Timestamp Server 16
2.7 Public and Private Blockchain 16
2.7.1 Hyperledger Fabric 18
2.8 Summary 20

3 Related Technologies **22**

4 Approach and Methods **25**

5 Design and Implementation of a Blockchain Shipping Application **27**
5.1 User Story as a Proof of Concept 27
5.2 System Architecture 29
5.3 Design and Implementation 32
5.3.1 Data Model Implementation 32
5.4 Smart Contract Implementation 36
5.4.1 CreateOrder Smart Contract 38
5.4.2 Buy Smart Contract 39
5.4.3 OrderFromVendor Smart Contract 40
5.4.4 RequestShipping Smart Contract 41
5.4.5 InDelivering Smart Contract 42
5.4.6 Deliver Smart Contract 43
5.4.7 PaymentReq Smart Contract 44
5.4.8 Pay Smart Contract 45
5.4.9 ReturnOrder Smart Contract 46
5.4.10 ReturnOrderVendor Smart Contract 47
5.4.11 RefundRequest Smart Contract 48
5.4.12 Refund Smart Contract 49
5.4.13 CancelOrder Smart Contract 50
5.5 Query Implementation 50
5.6 Access Control List Implementation 51
5.7 Summary 52

6 Discussion and Results **54**
6.1 Development Environment 54

6.2 Implementation Results 56
6.3 Summary 69

7 Performance Evaluation **72**
7.1 Experiment I: Network Load 75
7.1.1 Experiment I: Method A Fixed Rate Controller 75
7.1.2 Experiment I: Method B Fixed Feedback Rate Controller . . 77
7.2 Experiment II: send TPS rate 79
7.2.1 Experiment II: Method A Fixed Rate Controller 79
7.2.2 Experiment II: Method B Fixed feedback controller 81
7.3 Results Analysis 82
7.3.1 Analysis of Experiment I. Method A 82
7.3.2 Analysis of Experiment I. Method B 83
7.3.3 Summary of Experiment I 84
7.3.4 Analysis of Experiment II. Method A 85
7.3.5 Analysis of Experiment II. Method B 85
7.3.6 Summary of Experiment II 86
7.3.7 Summary of Experiment I and Experiment II 86
7.3.8 Summary 87

8 Conclusion and Future Work **88**
8.1 Summary of Contributions 88
8.2 Future Work 89

A Data Model Implementation Code **97**

B Smart Contracts Implementation Code **102**

C Query and Access Control List Implementation Codes **111**

D Experiments Configuration **115**

List of Tables

Table 5.1 Asset Order 33
Table 5.2 JSON Object: StatusList 37

Table 6.1 Participants JSON Data 56

Table 7.1 Experimental Hardware and Software Setup 73
Table 7.2 Experiment I: Data Sets 75
Table 7.3 Experiement I: Method A Fixed Rate Controller 76
Table 7.4 Resource Utilization for Experiment I Method A 76
Table 7.5 Experiement I: Method B Fixed Feedback Controller 77
Table 7.6 Resource Utilization for Experiment I Method B 78
Table 7.7 Experiment II: Data Sets 79
Table 7.8 Experiement II: Method A Fixed Rate Controller 80
Table 7.9 Resource Utilization for Experiment II Method A 80
Table 7.10 Experiment II: Method B Fixed Feedback Controller 81
Table 7.11 Resource Utilization for Experiment II Method B 82

List of Figures

Figure 2.1 Double Spending Problem 5
Figure 2.2 Public Key Cryptography 6
Figure 2.3 Chain of Transactions 7
Figure 2.4 Transaction Mechanism 7
Figure 2.5 Hash Function 8
Figure 2.6 Hash Function SHA-256 Result 9
Figure 2.7 Hash Mechanism in Blockchain 9
Figure 2.8 Hash Function SHA-256 of Four Zeros Nonce 12
Figure 2.9 PBFT Consensus Algorithm 14
Figure 2.10 Merkle Tree 15
Figure 2.11 Timestamp Server 16
Figure 2.12 Blockchain Classification 17
Figure 2.13 Blockchain World State 19

Figure 5.1 Application User Story 28
Figure 5.2 Application Architecture Overview 29
Figure 5.3 System Architecture Overview of the Application 31
Figure 5.4 CreateOrder Workflow 38
Figure 5.5 Buy Workflow 39
Figure 5.6 OrderFromVendor Workflow 40
Figure 5.7 RequestShipping Workflow 41
Figure 5.8 InDelivering Workflow 42
Figure 5.9 Deliver Workflow 43
Figure 5.10 PaymentReq Workflow 44
Figure 5.11 Pay Workflow 45
Figure 5.12 ReturnOrder Workflow 46
Figure 5.13 ReturnOrderVendor Workflow 47
Figure 5.14 RefundRequest Workflow 48

Figure 5.15 Refund Workflow 49
Figure 5.16 CancelOrder Workflow 50

Figure 6.1 Application's REST API Server 55
Figure 6.2 HTTP POST Response for Each Participant 56
Figure 6.3 HTTP POST: CreateOrder 58
Figure 6.4 HTTP POST: Buy 59
Figure 6.5 HTTP POST: OrderFromVendor 60
Figure 6.6 HTTP POST: RequestShipping 61
Figure 6.7 HTTP POST: InDelivering 62
Figure 6.8 HTTP GET Resposne: History of the Order 63
Figure 6.9 HTTP POST: Deliver 64
Figure 6.10 HTTP POST: Payment Request 65
Figure 6.11 HTTP POST: Pay 66
Figure 6.12 HTTP POST: ReturnOrder 66
Figure 6.13 HTTP POST: ReturnOrderVendor 67
Figure 6.14 HTTP POST: RefundRequest 68
Figure 6.15 HTTP POST: Refund 68
Figure 6.16 Access Control for Smart Contract Invocation 70

List of Acronyms

API Application Programming Interface

BFT Byzantine Fault Tolerance

HTTP Hypertext Transfer Protocol

IoT Internet of Things

JSON JavaScript Object Notation

PBFT Practical Byzantine Fault Tolerance

REST Representational State Transfer

SDK Software Development Kit

TPS Transaction Per Second

Glossary

Asset Tangible or intangible digital value in the Blockchain space

Business Logic Operating and regulation requirements of business services

Consensus Protocol The series of distributed processes to achieve an agreement among nodes

Data Model The application's data structure (classes)

Decentralization Eliminates the need for a central authority in the Blockchain space

Event Notifies a certain participant that a smart contract has been invoked in the Blockchain ledger

Hash Function Takes an input data and turns it into a fixed size hash value string

Hyperledger Caliper Blockchain benchmark tool and one of the Hyperledger projects hosted by The Linux Foundation

Hyperledger Fabric Blockchain open development framework and one of the Hyperledger projects hosted by the Linux Foundation

Network Load The number of transactions that are submitted to the Blockchain ledger, and the number of assets that are exchanged in the Blockchain space

Participant A client in the Blockchain space who can create an asset and submit a smart contract

Permissioned Blockchain Performing consensus on transaction is restricted to a predefined list of peers with known identities

Private Blockchain Accessing the data and submitting transactions are limited to a predefined list of participants

Proof of Concept The realization of a certain method or idea in order to demonstrate its feasibility

Provenance The place of origin and the full history of the asset in the Blockchain space

HTTP Response Body The JSON payload data that is returned by a server to a client

HTTP Response Code Indicates the status of the HTTP request (e.g., code 200 is a standard response for successful HTTP request)

Smart Contract A code written in Go or JavaScript to define the Blockchain's implementation methods to create and maintain the value of digital assets among participants

Transaction Latency The time from the point that the transaction is submitted to the point that it is confirmed and committed in the Blockchain ledger

Transaction Throughput The maximum rate in which valid transactions are committed in the Blockchain ledger

ACKNOWLEDGEMENTS

I woke up and looked out my friend's window seat as the Lufthansa Flight LH0492 was approaching the Vancouver International Airport. At that moment, I thought about how uncertain my future is but my mind interrupted this thought with one core and principle rule that I set for myself a long time ago: **"Beat down uncertainty and fears with absolute certainty and break through those fears."**

I would like to thank:

My Parents for giving me the reason to what I want to be.

My Supervisor Dr. Fayez Gebali for his support, encouragement, advice, and monitoring.

My Supervisor Dr. Hausi A. Müller for his support, feedback, encouragement, and caring.

WUSC, Michelle Manks, Dr. Marlea Clarke and Dr. Scott Watson for giving me this opportunity and for their enormous support.

My close friends Walid Al-Habboul, Marillia Techy and AlMontaser Al-Jundi for their support during hard moments and for sharing the experience.

Last, but not least, my partner **Claire Mackie** for her love and support.

DEDICATION

I would like to dedicate this work to my parents and my supervisors Dr. Fayez Gebali and Dr. Hausi A. Müller.

Chapter 1

Introduction

1.1 Motivation

Shipping and logistics in supply chains extend across borders in international trade and industries, and involve several phases in which information needs to be shared among the relevant parties, whether these parties are private companies or public organizations. These parties include suppliers, providers, customers, regulatory agencies, sellers, manufacturers, and buyers. The collected information helps with the essential business decision making process that may affect market capitalization and efficiency. IHS Global Insight, a recognized global leader in the financial analysis industry reported that the 2009 annual global contribution for the shipping industry was USD183.3 billion, created 4.2 million direct jobs, and compensated those employees, with USD27.2 billion [1]. However, the shipping industry still has significant challenges related to transparency and immutability that are associated with their current shipping processes.

Advanced automation technology, and more specifically, Blockchain promises to change the aspect of how shipping and logistic companies operate. Blockchain provides high-level supply chain flexibility and security, reduces bottlenecks in third-party certifications, and eliminates the need for paper-based documentation [2].

Recently, Pacific International Lines, a Singaporean shipping company, announced its collaboration with International Business Machines Corporation (IBM) to improve the process of documentation in the supply chain using Blockchain technology. The announcement gained support from the Maritime and Port Authority of Singapore and promised to increase efficiency and diminish unnecessary handling costs [3].

1.2 Problem Definition

In the digital world today, clients' expectations surrounding the supply chain ecosystem are changing rapidly. Clients are now demand that their orders and inventory become more transparent and visible throughout the entire production cycle [4]. Supply chain transparency refers to stakeholders' (e.g., customers, suppliers and regulatory bodies) ability to access the collected information of goods in the entire supply chain, from order to delivery [5] [6].

Supply chain visibility means that the products' information is properly collected and recorded throughout the entire production cycle [7]. Visibility enables stakeholders to track an asset anywhere along the entire supply chain. For instance, if a stakeholder requests a manufacturing company to design a product, the manufacturing company may collect and record information related to each stage during the production life cycle (i.e., brainstorming, designing, implementation and delivering). However, the manufacturing company may not disclose all this information to the stakeholder. In this case, the manufacturing company is transparent but not visible to the stakeholder. This is because the information that the manufacturing company shared may be selective and insufficient. In another scenario, if the manufacturing company decides to not share the information with the stakeholder, the manufacture company is not transparent. Therefore, what is defined as visible is transparent, if the visible information is complete and accurate and the information is provided to the stakeholders with full disclosure [8].

Maersk, the world's largest container shipping company [9], tracked a container filled with roses from Kenya to the Netherlands in 2014. The company concluded that approximately 30 organizations were directly involved in the process of shipping the container. During the 44-day shipping process, one of the essential documents went missing but was later found, amid a large pile of papers. Following this incident, Maersk stated that "the paperwork and processes vital to global trade are also one of its biggest burdens" [10].

Shipping systems today involve labour-intensive and inefficient processes because each organization has its own set of ledgers where they correlate and understand the shipping process differently across many intermediary organizations.

1.3 Approach

This thesis proposes a Blockchain, distributed, shared, paperless, private and permissioned solution-based ledger that aims to automate the shipping processes among different participants in the supply chain ecosystem (e.g., sellers, buyers, providers). The application's user story is designed to implement the Blockchain smart contracts and enhance the trading capabilities of assets among different participants. This application also has a private and permissioned ledger to secure data flow and meet common business requirements. These requirements define the data accessibility rights among different parties and thereby protect the privacy of the participants data. Moreover, this work aims to evaluate and measure the performance of this Blockchain application in terms of different indicators (e.g., transaction throughput and transaction latency).

1.4 Thesis Outline

Chapter 2 explains the Blockchain fundementals that are essential to understand the rest of this thesis as well as realize the potential of Blockchain technology. Chapter 3 provides a summary of related technologies with a focus on Blockchain. Chapter 4 describes the approach and methods of this research. Chapter 5 presents a detailed description of the design and implementation of our subject application including Data Model, Smart Contracts, Events, Query and Access Control. Chapter 6 discusses the results that are obtained from the implementation stage. Chapter 7 provides the performance measurement's experimental setup and its results. Finally, Chapter 8 concludes with the summary of the contributions and proposes directions for future work.

Chapter 2

Blockchain Fundamentals

This chapter discusses the Blockchain fundamentals and provides knowledge about Blockchain technology which will be helpful in understanding the remainder of this thesis. Public key cryptography is introduced first and followed by the transaction mechanism in Blockchain. Next, the hash function and consensus are described, along with different types of consensus protocols. Then, Merkle tree that relies on hash fucntion methodology and timestamp server, is explained. Finally, Blockchain technology, which includes public and private Blockchain as well as Hyperleger Fabric, is explored. In the aftermath of the 2008 global financial crisis [11], *Satoshi Nakamoto* published Bitcoin's white paper "P2P (peer to peer) Electronic Cash System" [12]. In this paper, Satoshi defines Bitcoin as a decentralized digital currency which can be sent from peer to peer within the network without any central authority or intermediary layer involvement [13]. In fact, the development of Bitcoin was to solve the double spending problem by implementing the concept of atomicity [14] [15]. The double spending problem occurs when a given set of coins is spent in more than one transaction as illustrated in Figure 2.1 [16].

Atomicity guarantees that the given transaction either happens before or after other transactions or does not happen at all. For example, sending a $20 transaction from A to B must happen at a specific timestamp and before or after any other transactions. In this case, it would not allow the $20 to be spent more than once in separate and simultaneous transactions.

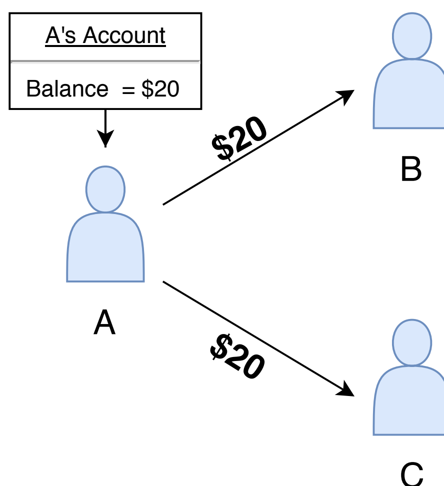

Figure 2.1: Double Spending Problem

2.1 Public Key Cryptography

One of Blockchain's core fundamentals is cryptography. Blockchain employs cryptography as a method to protect the user's identity and to ensure the authenticity and integrity of any given transaction [17]. Cryptography is a method of applying mathematical principles in a specific form to transmitted data, in order to control who can access and process this data [18]. While encryption is the process of encoding transmitted data to an unreadable format that third parties cannot intercept.

Blockchain uses public key cryptography to validate transactions in the network. It relies on the asymmetrical mathematical complexity that uses public and private keys. The use of these keys is to deliver the following two specifications:

- Data encrypted with the public key can only be decrypted using the private key.
- Data signed with the private key can be verified using the public key.

The private key cannot be derived from the public key while the public key can be derived from the private key. The public key is designed to be shared over the network and therefore can be exposed to anyone [19] [20].
E_p represents the set of encryption transformations while D_v is the set of corresponding decryption transformations. The transmitted message is m and c signifies a random ciphertext. As illustrated in Figure 2.2, Alice and Bob are participating in

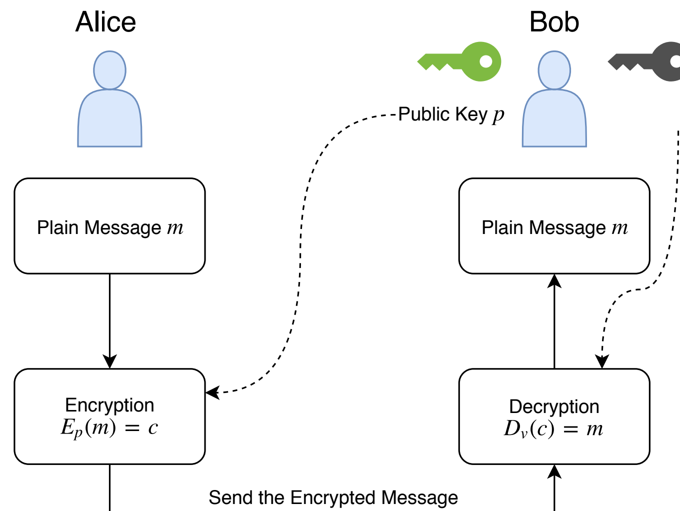

Figure 2.2: Public Key Cryptography

a two-party network communication. If Alice wants to send a secure and encrypted message to Bob. Bob, therefore, chooses the key pair (public key p, private key v) and sends the public key p to Alice. Alice must encrypt m using Bob's public key and the encryption transformation E_P [21]:

$$E_p(m) = c \tag{2.1}$$

Once Bob receives the encrypted message m, Bob has to decrypt the ciphertext c by applying the inverse decryption transformation D_v, associated with the private key v:

$$D_v(c) = m \tag{2.2}$$

On a large-scale network, any participant can send encrypted messages to Bob, which only Bob can decrypt. Moreover, senders cannot deny their sent message authorship. Hence, public key cryptography enhances the system's authenticity, integrity, non-repudiation, and privacy [21].

2.2 Blockchain Transaction

Within Bitcoin's P2P network, a transaction is the process of transferring Bitcoin's credit ownership between two participants [22]. If the network validates the authen-

ticity of any given transaction, the network adds this transaction to the validated transaction list, inside the block as illustrated in Figure 2.3.

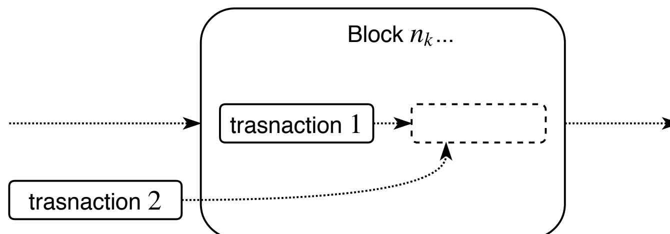

Figure 2.3: Chain of Transactions

Figure 2.4 illustrates how to initiate transactions in Bitcoin:

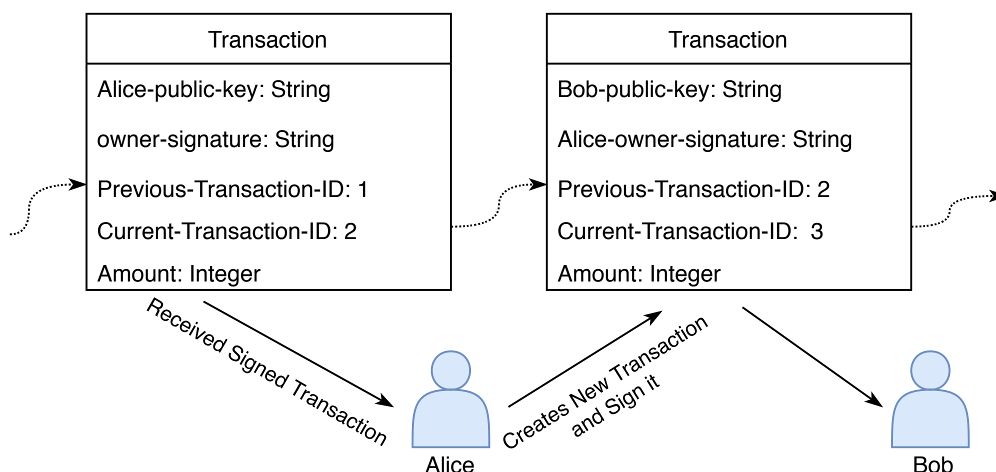

Figure 2.4: Transaction Mechanism

In Figure 2.4, Alice received the Bitcoin credit amount as a result of a previous transaction. Then, she decides to transfer this credit to Bob. He must validate that Alice is the one who sent this credit, in order to maintain the network's non-repudiation. Furthermore, Bob needs to trace the source of Alice's credit to also sustain the network's authenticity [12] [23]. To achieve these processes, the network implements public key cryptography. Alice creates the transaction payload which contains the following fields:

- Bob-public-key field is to designate Bob as the new authentic owner of the transferred credit.

- Alice-owner-private-key is to designate Alice as the old authentic owner of the transferred credit.

- Previous-transaction-ID signifies the origin of Alice's credit through the previous transaction.
- Current-transaction-ID indicates the order of a current transaction within the transaction list.
- Amount indicates how much credit Alice wants to transfer to Bob.

On a large scale, each transaction points to the previous one using the *previous-owner-private-key* and *previous-transaction-ID* fields. This creates a traceable and linked list of transactions.

2.3 Hash Function

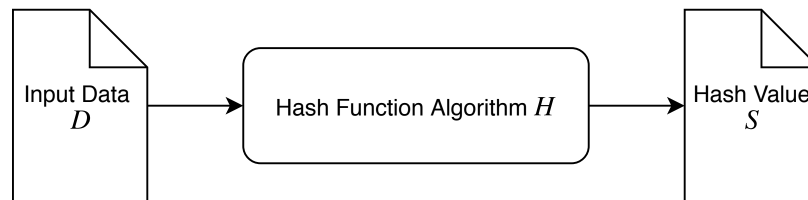

Figure 2.5: Hash Function

One of Blockchain's foremost fundamentals is hashing. Figure 2.5 illustrates the hashing process. This process takes input data of any length, D, and turns it into a fixed size hash value string, S, using a specific hash function algorithm, H. The famous hash functions include SHA-256, SHA-1, MD5, and CRC32 along with many others [24]. The hashing process satisfies that only the same input data can result in the same hash value. A slight change in the input data would generate a completely different hash value. Therefore, hashing is an irreversible process because recreating the input data from its hash is not possible [25].

For example, the hash function, $H =$ SHA-256[1], is implemented to hash the University of Victoria's student number, $D =$ "V00863910", which generates its associated hash value as follows:

[1]Github Repository: https://github.com/MaherBouidani/hash-encode

$$\text{Var D} = (\text{V00861039})$$

$$\text{S} = \text{SHA-256('D')}$$

$$\text{S} = 4694\text{ae}11066....$$

Figure 2.6 shows the resulting hash value S.

```
[Mahers-MacBook-Pro:hash-encode maherbouidani$ node SHA256.js
4696ae11066b1ca43b1f11af2b163ed051174109bcbc6108dff6034902ecd0ff
Mahers-MacBook-Pro:hash-encode maherbouidani$ █
```

Figure 2.6: Hash Function SHA-256 Result

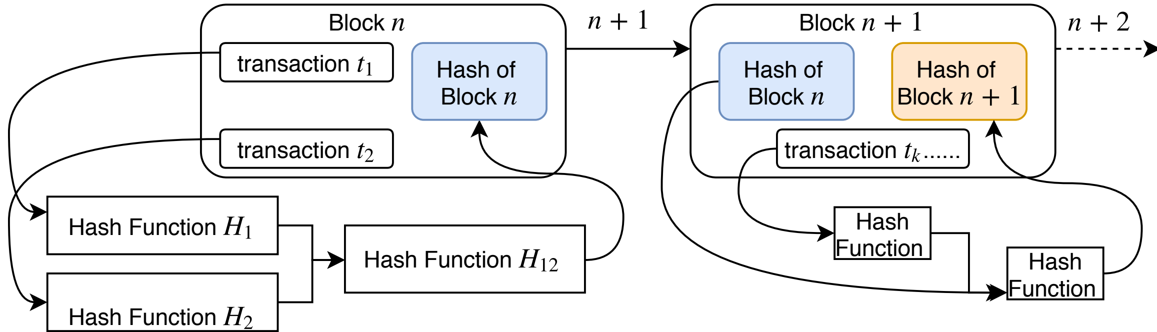

Figure 2.7: Hash Mechanism in Blockchain

In particular, Blockchain implements a hashing mechanism to preserve the traceability and security in Blockchain ledger, as illustrated in Figure 2.7. Block n applies a hash function algorithm to hash all transactions in this block and to produce a unique hash value. The newly created Block $n+1$ has new transactions that combine with the hash of the previous block n. This combination produces a new hash value for Block $n+1$ which is repeated with each newly created block. As a result, each block has a link to the previous block through its hash, therefore, creating an immutable traceable chain of blocks [26].

2.4 Consensus

Current centralized systems rely on central administrators to check the validity of submitted data [27]. For example, Graduate Admission and Records Office (GARO) is the University of Victoria's (UVic) central authority to validate students submitted

papers and issue related documents (e.g, official transcript or letter of completion) [28]. For UVic systems, GARO is a trustworthy and honest node.

C. Cashin *et al.*, [29] defined Blockchain as a “distributed ledger for recording transactions, maintained by many nodes without central authority through a distributed cryptographic protocol.” In Section 2.3, Alice sends a transaction to Bob, however, the used system needs to ensure that Alice is an honest node first before proceeding to process this transaction. On a large scale with hundreds of nodes, Blockchain must apply a fault-tolerance consensus protocol to ensure that participating nodes are honest to eliminate the negative effect of malicious or dishonest nodes. For instance, the general reliability assumption for a ledger with n total nodes states that no more than either a third or a half of the total nodes are faulty (f) in a ledger [30] as expalined in Equation 2.3:

$$f < n/k \quad for \quad quorum \quad k = 2, 3 \tag{2.3}$$

As shown in Equation 2.3, when quorum, $k = 2$, then $f < n/2$, the number of faulty nodes is less than a half of the total nodes. Moreover, when quorum, $k = 3$, then $f < n/3$, the number of faulty nodes is less than a third of the total nodes. Therefore, the number of honest nodes (h) in a ledger is calculated as shown in Equation 2.4.

$$h = n - f \quad nodes \quad are \quad honest \tag{2.4}$$

A consensus protocol is a series of distributed processes to achieve an agreement among nodes on any given submitted transaction. It is a decision-making process where nodes support the decision that is best for the Blockchain ledger [31]. More precisely, developing a consensus protocol ensures the following properties [29] [32]:

- Validity: if an honest node, h, broadcast a message, m, then h eventually delivers m to itself.
- Agreement: if some honest nodes deliver a message, m, then m is eventually endorsed (delivered) by all other honest nodes.
- Integrity: an honest node will not duplicate the same broadcasted message and the message received, m, is therefore identical to the message that was sent (non-repudiation).

- Safety: each ledger's node is guaranteed the same sequence of inputs and results in the same output on each node. This property promotes a replication service where eventually each node has an updated copy of all messages to preserve consistency in the ledger [33].

Algorithm 1: Reliable Multicast Algorithm [34]

```
initialization received ⟵ ∅;
while multicast (m) process is executed by node p do
    send message m to all nodes;
end
foreach m is received by node q do
    if m ∉ received then
        received ⟵ received ∪ m;
        multicast (m);
        send m to all nodes
    end
end
```

Algorithm 1 shows a reliable multicast algorithm with primitive $multicast(m)$ process [34]. The $multicast(m)$ defines the process of multicasting message m to a group of nodes. To multicast a message, a node p executes $multicast(m)$ to send the message to all nodes including itself. When node q receives m, node q checks if m is not included in its set of received messages. If the condition is true, node q adds m to its set of received messages and executes the $multicast(m)$ process to send the message to all the other nodes (node q is not the original sender). This algorithm achieves the integrity property since the duplicated message is detected and not delivered. Furthermore, it satisfies the validity property since an honest node will eventually deliver m to itself. Agreement is reached when each honest node executes $multicast(m)$ after the event *receive* (m) has occurred.

2.4.1 Proof of Work

The earliest implementation of a distributed consensus algorithm in Blockchain is Bitcoin's proof of work (PoW) algorithm [35] [36]. The proof of work is a process to confirm that a node contributes a certain amount of computational work in solving a hard cryptographic puzzle where any solution is easy to verify [37]. As previously

mentioned in Section 2.3, each block includes the hash of the previous block. Furthermore, each block's hash must match a certain pattern which is a precise leading digit of zero bits in Bitcoin known as $Nonce$ [38]. This nonce rule is called the hash value difficulty. For example, it takes up to 60,000 attempts in order to find a hash input data that results in a four leading zero hash value difficulty [39]. Using the hashing process in Section 2.3, the result of hashing input data, D = "debearded", with $Nonce$ of four zeros is a hash value, S, also with four leading zeros as illustrated in Figure 2.8.

```
Mahers-MacBook-Pro:hash-encode maherbouidani$ node SHA256.js
0000fa20ee307c4f15a53bf69ae0ca1c66c675b0ae94781ff5b6f8cff90ebcca
```

Figure 2.8: Hash Function SHA-256 of Four Zeros Nonce

As hashing the same input data always leads to the same hash value, a nonce is therefore included as part of the block. Using different nonce rules changes the hash of the block. This nonce rule is enforced by the network of nodes since they only accept blocks generated by other nodes if the hash of that block matches the nonce rule [40].

A node that contributes computational work to achieve the above process, is called a $Miner$. The Bitcoin system awards the $Miner$ with a bitcoin currency reward if this $Miner$ completes the PoW task and satisfies the nonce rule successfully. This is what incentivises individual users to become $Miners$ and to provide the Bitcoin network with their computational resources [41]. The more computing capabilities $Miners$ have, the faster they find a hash that meets the $Nonce$ requirements.
The time period $T(t)$ for a $Miner$ to perform r operations per second to find a valid block is distributed exponentially with the rate r/D where D is the target difficulty [42]:

$$P\{T(r) \leq t\} = 1 - exp(-rt/D) \tag{2.5}$$

However, this process requires the power of a large number of GPUs (i.e., electricity that is comparable to the energy consumption in both Denmark and Ireland [43] [44]).

2.4.2 Proof of Stake

An alternative to the PoW algorithm is the proof of stake (PoS) consensus algorithm which does not apply "mining" in its mechanism [45]. The main advantage of the proof of stake approach is the diminished need for computational power and hence a lower entry barrier for block generation rewards. The PoS is a process to prove that a node owns a certain amount of network stakes (coins) before being cleared to participate in forming consensus in the network. However, this process includes the possibility for an attacker or malicious node to hold enough stakes and hence becomes the node with the highest decision weight which relates to achieving consensus in the network [46].

2.4.3 Practical Byzantine Fault Tolerance

Byzantine Fault Tolerance (BFT) is the ability for a distributed system to correctly reach a sufficient consensus despite the presence of malicious nodes from the failed system which propagates incorrect information to other nodes. BFT defends against system failures by mitigating the influence that these malicious nodes have on the correct function of the system. Derived from the Byzantine Generals' Problem, this topic has been researched and optimized with a diverse set of solutions [47] [48]. Miguel *et al.* introduced Practical Byzantine Fault Tolerance algorithm (PBFT) as one of these optimization in 1999. The PBFT scheme focuses on delivering a State Machine Replication (SMR) that tolerates Byzantine faults by assuming that there are independent node failures.

The algorithm is designed to work in asynchronous systems and is developed to be high-performance with an overhead run-time and a slight increase in latency. All of the nodes in the PBFT scheme are ordered in a sequence with one node being the primary node (leader). The algorithm provides both integrity, agreement and safety when the condition of Equation 2.4 is true ($f < n$-1/3). Therefore, the replies that are received by nodes from their requests are correct due to atomic consistency [49].

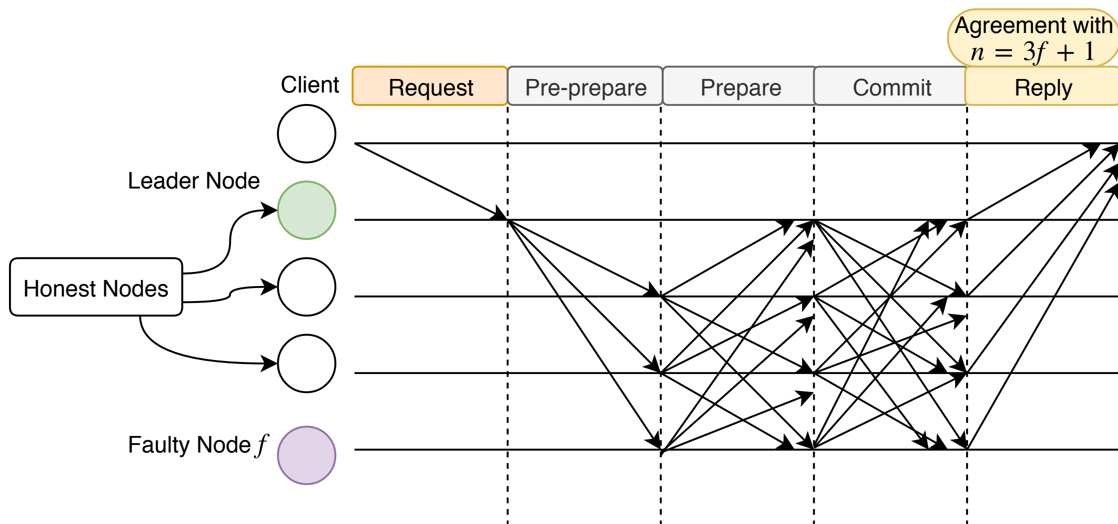

Figure 2.9: PBFT Consensus Algorithm

As illustrated in Figure 2.9, each round of PBFT consensus consists of five phases that are executed in sequence as follows:

- Request: the client sends the leader node a request to invoke a service operation.
- Pre-prepare: the leader node multicasts the request to all other nodes.
- Prepare: the nodes multicast the request to ensure that a non-faulty leader node agrees on the total order of the request.
- Commit: the leader nodes commit the request to all other nodes to achieve the consensus's agreement property.
- Reply: if the nodes reach an agreement, they reply back to the client with their approval.

The client awaits f + 1 (f represents the maximum number of nodes that may be faulty) replies from different nodes with the same result. This effectively eliminates attempts to defraud the system since the client waits for at least one honest node to vote on the integrity of the decision that is taken by all faulty nodes. One of the primary advantages of the PBFT model is its ability to provide finality without the need for confirmations like in the PoW models. If a proposed block is confirmed as legit by the nodes in a PBFT system, then that block is final. This is enabled because all honest nodes agree on the state of the system at that specific time from their communication with each other [50] [51].

2.5 Merkle Tree

The concept of Merkle tree was first introduced by Ralph Merkle in 1979 [52]. A Merkle tree is a binary tree that summarizes and verifies the integrity of a large set of transactions. Similar to the binary tree, any Merkle tree's node has, at most, two child nodes [53]. Bitcoin and thus Blockchain uses Merkle trees to summarize all the transactions in a block and verify whether a specific transaction is invoked in a block or not, using an efficient search algorithm with time complexity $O(\log_2(n)$ [54].

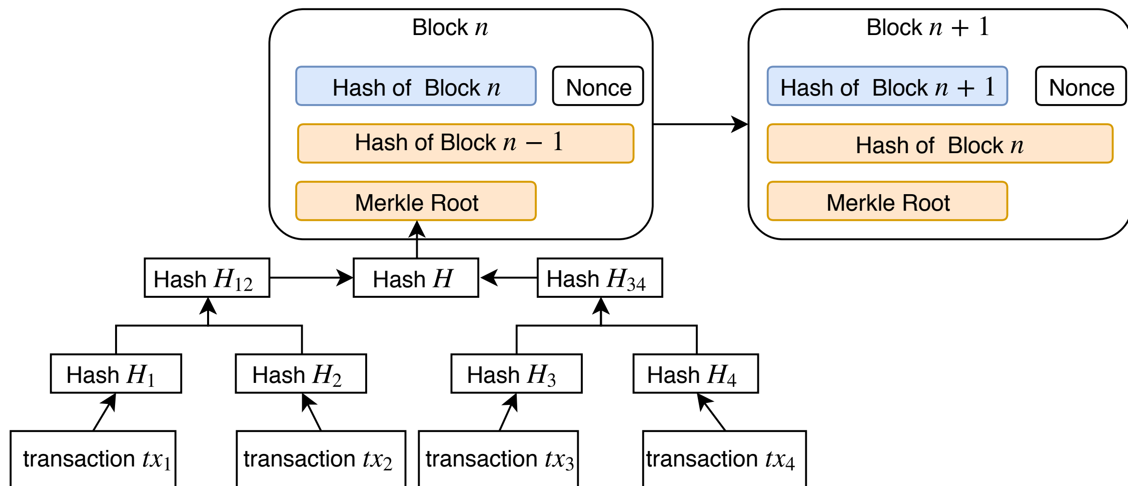

Figure 2.10: Merkle Tree

Each block header in Bitcoin and Blockchain contains a summary of all the transactions in the block using a Merkle tree [55] as illustrated in Figure 2.10. Using the hashing mechanism from Section 2.3, each block will construct its Merkle tree from the bottom to up by running the hash function recursively on pairs of nodes. This process will continue until there is only one hash called Merkle root. Therefore, the resulting hashed transactions are stored cumulatively in each child node and percolating up to the Merkle root, forming a Merkle path [56].

Simplified Payment Verification (SPV) is a notable use case that implements the Merkle tree technique [31]. Nodes in SPV do not have to download all transactions in the blocks in order to check whether a certain transaction is included or not; instead, they use the Merkle path [57].

2.6 Timestamp Server

Blockchain defines the Timestamp server as a method to record the time that the nodes received and agreed on each transaction [35]. A Timestamp server takes a hash from every block (hash of all transactions in a block as explained in Sections 2.2 and 2.3, respectively) and stamps it with the time as illustrated in Figure 2.11. This Timestamp contains the previous block's Timestamp and proves the existence of data at that time.

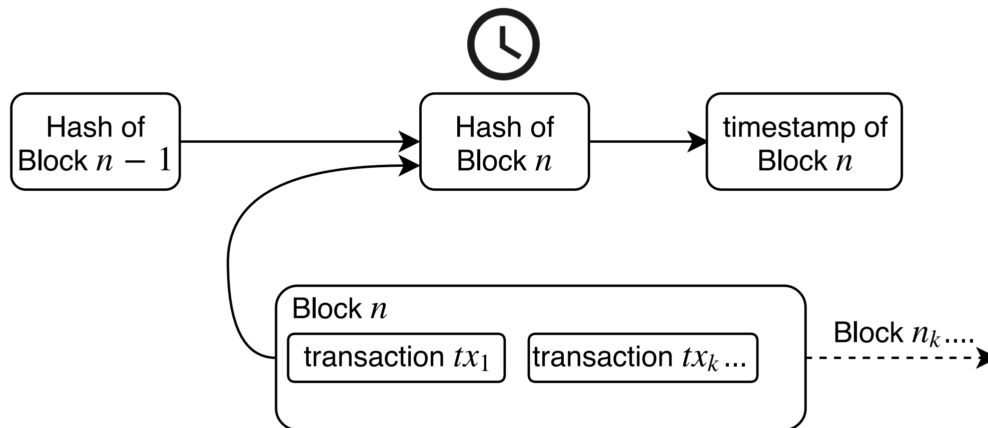

Figure 2.11: Timestamp Server

2.7 Public and Private Blockchain

Blockchain technology has recently received notable attention as enterprises and institutions started to invest in research and development to leverage its potential for applications beyond cryptocurrency [58]. As observed in Section 2.1, Blockchain is characteristically known as an immutable distributed ledger of records that are shared and verified using a specific consensus algorithm among participating parties. It opens the door to enhance digital transformation development without centralized authority interference. Blockchain can be classified based on many factors like the ledger's accessibility rights for participants and who can perform a consensus algorithm that is used in the ledger as shown in Figure 2.12 [59].

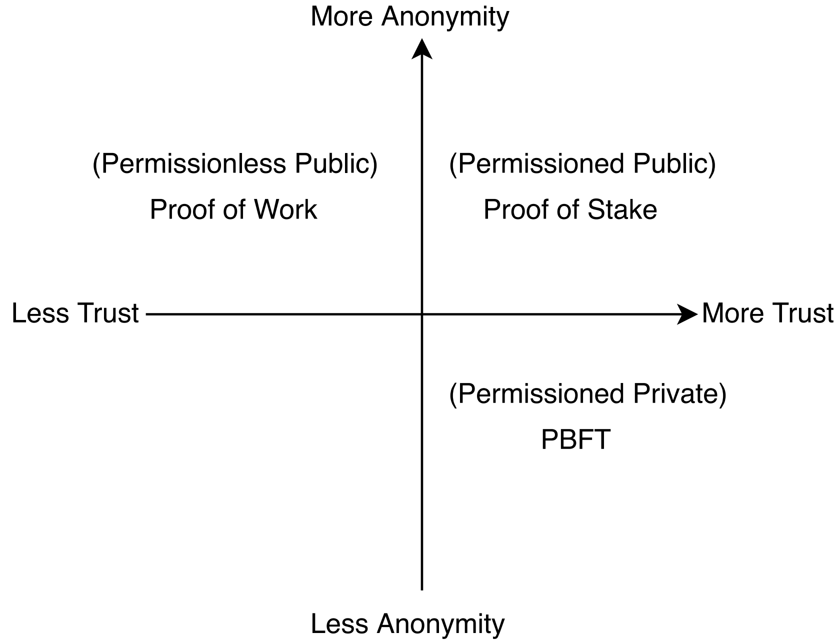

Figure 2.12: Blockchain Classification

Definition I. A public Blockchain is a Blockchain, in which, there are no restrictions for participants to read the data and to submit transactions for inclusion into the Blockchain ledger [60].

- Bitcoin and Ethereum are examples of a public Blockchain [35] [61].
- the proof of work, proof of stake and Federated Byzantines Agreement (FBA) are examples of consensus used in a public Blockchain framework [62].

Definition II. A private Blockchain is a Blockchain, in which access to the data and submitting transactions is limited to a predefined list of participants [60].

- Hyperledger Fabric is an example of a private Blockchain [63].
- Apache Kafka and Practical Byzantines Fault Tolerance (PBFT) are examples of consensus used in a private Blockchain framework [64].

Definition III. A permissioned Blockchain is a Blockchain, in which performing consensus on transactions is restricted to a predefined list of participants with known identities (they are known to the Blockchain network) [65].

- Hyperledger Fabric supports a permissioned Blockchain architecture.

Definition IV. A permissionless Blockchain is a Blockchain, in which performing consensus on transactions is permitted to all participants that joined the network (they may not be known to the blockchain network) [66].

It is essential to note that a public Blockchain may not always be permissionless. For example, a proof of stake Blockchain is a public and permissioned Blockchain since participants must prove upfront that they own a certain amount of stake in order to be involved in achieving consensus in the network.

2.7.1 Hyperledger Fabric

The Hyperledger initiative was first introduced by Linux Foundation in early 2016. It is an open development and global collaborative approach that aims to advance Blockchain technology. Currently, it includes more than 50 enterprises and *R&D* institutions [67].

Linux Foundation adopted the Fabric framework that was initially developed by Digital Asset and IBM in late 2016 [68]. Hyperledger Fabric is a Blockchain framework that endeavours to develop a distributed ledger for consensus implementation and to enhance its modular architecture for executing smart contract.

A smart contract is a code written in Go[2] or JavaScript[3]. It is designed to define Blockchain's implementation methods that are executed when predefined conditions are met [69]. Sorine *et al* defined smart contract as "being able to perform useful functions to create, maintain or augment the value of digital assets" [70].
Smart contracts initialize and manage the Blockchain ledger state through transactions submitted by ledger's nodes. The Blockchain ledger state (also known as world state) represents the latest values of all keys included in Blockchain's transaction log [71] as illustrated in Figure 2.13. Furthermore, Fabric supports the CouchDB implementation, in order to perform operations like set and query by ID.
Nodes in Hyperledger Fabric plays a vital role in reaching consensus in the ledger. These nodes can be classified into three types [72]:

---

[2]https://golang.org/
[3]https://www.javascript.com/

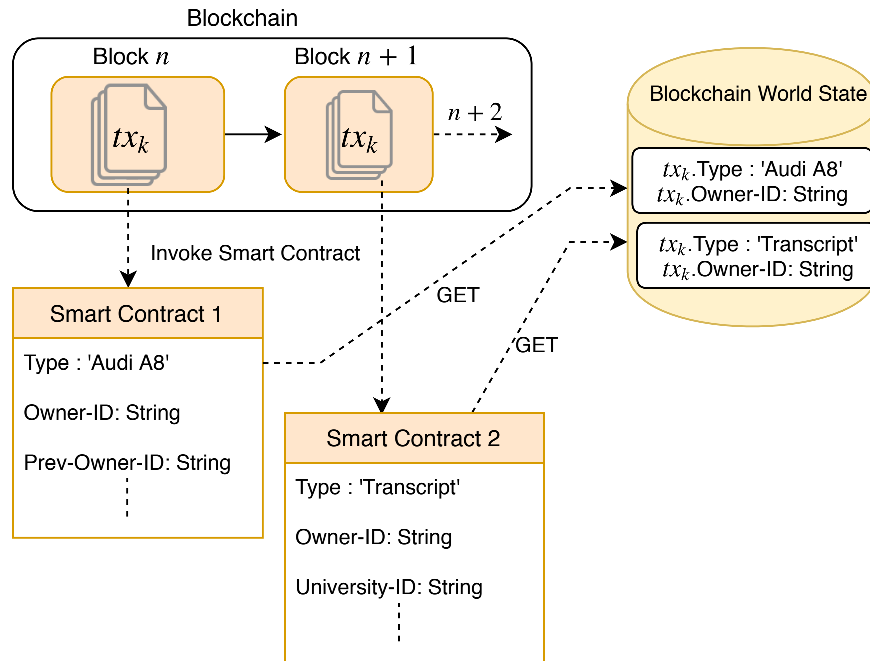

Figure 2.13: Blockchain World State

- **Client**: clients act on behalf of an end-user and submit the transaction-invocation request to the endorsers, and broadcasts transaction proposals to the ordering service.

- **Peer**: a node that commits transactions and maintains the state of the ledger. Peers can either be an endorser or validator and can be switched from one role to another as needed.

- **Orderer**: a node that runs a communication service between an endorser and validator peers

The consensus protocol in Hyperledger Fabric is broken into five phases [73]:

- **Submission**: client submits the transaction-invocation request to the endorser peer.

- **Endorsement**: endorser checks the client's identity information to ensure that this client has the right to submit the transaction first. The next step is to sign this transaction request with the endorser's endorsement policy and return the response back to the client. The Endorsement policy is a set of rules that define

how to write an acceptable transaction code (Smart Contract) in Hyperledger Fabric.

- **Broadcasting**: the client broadcasts the received endorsement transaction to the orderer.
- **Orderer**: the orderer adds the transaction to a newly created block and delivers it to the validator. Note that the orderer does not check or verify the transaction. Therefore, the orderer does not consume a lot of the computational resources.
- **Validator**: the validator validates the correctness of the block and checks the correctness of the endorsement policy of each transaction inside a block.

2.8 Summary

This chapter described how public key cryptography, hash functions and merkle trees are used to achieve integrity, authenticity and tracebility in Blockchain. Public key cryptography is an attractive approach to solve trust issues since a user can sign the transactions with the private key that is secured in a user's wallet, while distributing the public key to other users as verification. Furthermore, each transaction links to the previous one using the user's private key. Additionally, the hash function mechanism combines two methods to deliver an immutable traceable chain of Blocks. Two methods were shown that recreating the input data from its hash is not possible and each newly created block links to its previous one through its hash.

A description of Consensus and its different types was provided to show the decentralization capability of Blockchain. Consensus eliminates the need for a third-party or central authority to validate the correctness of processes in modern systems. The energy consumption of the proof of work's consensus is comparable to the consumption in both Ireland and Denmark. However, even though the proof of stake consumes less energy than the proof of work, the proof of stake's protocol includes a possibility for an attacker or malicious node to hold enough stakes and hence become the majority.

Blockchain is an emerging technology for distributed and decentralized architectures to share data among untrusted parties. The public and private Blockchain can be

leveraged for applications beyond the traditional cryptocurrency use cases. The conceptual definitions for each concept of private, public, permissioned and permissionless show the development scope of Blockchain. Each combination among these concepts delivers a certain capability as shown in Figure 2.12. Finally, an overview of a private and permissioned Blockchain Hyperledger Fabric was presented, as it is the development environment of the subject in this thesis.

Chapter 3

Related Technologies

This chapter introduces the Blockchain's application domain and recent Blockchain use case. Additionally, an overview of the supply chain's centralized and enterprise management system (ERP) is presented.
Blockchain technology has gained recognition in wider audiences since Bitcoin achieved a significant market capitalization of USD237 billion in 2017 [74]. However, Blockchain's frameworks and its potential capabilities extend beyond cryptocurrencies. It enables existing technology applications to transform and evolve into more efficient and transparent forms. The following examples are the potential platforms that Blockchain technology could deliver in the coming years:

- Digital Identity
  A self-sovereign ID can be used to verify an identity without needing an individual. It eliminates the need to produce numerous documents and paperwork each time they need their identity verified. Switzerland-based Uport is an example of an RD foundation working in this field [75].

- Energy Market
  Blockchain technology enables the smart metering of electricity generated through a user's solar panels to be recorded and traded on a ledger. This potential application may lead to reduced costs and improved efficiency by eliminating the need to rely on a centralized grid. Grid Plus is an example of a company that develops solutions in this field [76].

- Health Care
  Developing Blockchain to record patient information on a distributed ledger

can deliver a timestamped audit trail. This makes access to a patient's health information more secure. The Medicalchain company is working in this field [77].

- Automating Regulatory Compliance Blockchain could improve efficiency in the area of anti-money laundering compliance. This helps banks automate regulatory reporting on transactions [78].

SAP's ERP platfrom[79] [80] and E2Open's supply chain management application [81] [82] are examples of leading-edge supply chain systems that provide well-developed functionalities and interoperability with enterprise resource planning (ERP) systems. These systems are cloud-based and adopt a centralized software and hardware architecture. Given the number of stakeholders in the system, a centralized architecture has limitations related to its bureaucratic nature, and the number of third party auditors and facilitators that are involved in each of these processes. However, a centralized system provides a degree of scalability that is associated with redundancy.

Hyperfacture is an example of a Manufacturing as a Service (MaaS) application that is based on Blockchain's Hyperledger Fabric which is based on Golang [83]. It aims to match clients that want to deliver their product, with manufacturers that own spare machines. Although the Golang programming language has experienced a recent rise in popularity, JavaScript [84] includes various rich libraries and frameworks that support the implementation of modular and asynchronous architectures. This feature determines the degree of the public adoption in the developers' community and RD enterprises.

Tangle [85] and Corda [86] are examples of Blockchain platforms that aim to avoid the need of global consensus. According to their schemes, each node has its own hash chain. Therefore, transactions between nodes are recorded on their respective chains. This results in a directed acyclic graph. Given that both platforms avoid global consensus, this approach achieves desirable properties similar to BitTorrent [87]. However, the applications and security properties of this approach are limited. Given their architecture schemes, a malicious node may lie to the nodes regarding its own chain by submitting the same data with a different set of versions to the other nodes in the system. Thus, the network will have a conflicting view on the state of the malicious node.

This chapter described Blockchain's ecosystem and its recent potential uses. Furthermore, it described the limitations of the ERP centralized systems in supply chain. These limitations are due to the nature of centralized systems that depend on third-parties to validate the correctness of the processes in the system. Additionally, this chapter presents an overview of the programming languages used in Hyperledger Fabric in terms of the degree of potential public adoption, which is an important factor in collaborative development.

Chapter 4

Approach and Methods

To demonstrate the method and approach, a supply chain's shipping application was designed and implemented based on Blockchain private and permissioned Hyperldeger Fabric to deliver decentralization, data accessibility rights, visibility, and transparency capabilities. The main objective is to design a platform that can be used to exchange assets and values among untrusted participants (i.e, participants do not trust each other) and thus improve the efficiency of the current supply chain practices by creating a traceable and immutable chains of transactions and blocks.

The nature of supply chain ecosystems requires a certain degree of information privacy along with an access control layer over the application. This layer defines who can access the application by implementing the concept of participant's identity. A participant uses the identity card that is issued by the application's owner in order to access it and perform a pre-defined role or task.

The current supply chain processes was identified to determine the relationships among participants in the system as well as the supply chain's desired restrictions on these participants. This research helped design the proof of concept (user story) of this application.

The development phase of the application included the design and implementation of the application's system architecture. This system architecture includes the definition and implementation of the Data Model, Smart Contracts, Query, Events and Access Control List along with the integration of the development dependencies (i.e., JavaScript SDK and Execution Runtime). The application was designed using JavaScript and validated by simulation using the Loop Back REST API server.

The REST API server consists of a set of end points to communicate with the application and thus the Blockchain Hyperledger Fabric's ledger.

Finally, the performance evaluation and measurements using the Hyperldeger Caliper tool was conducted. This tool was integrated and configured to communicate with the Hyperledger Fabric's application ledger to derive the performance results of transactions throughput, transactions latency and resource utilization. The performance was measured in two experiments and each experiment features method A and B.

Chapter 5

Design and Implementation of a Blockchain Shipping Application

This chapter presents the desing and implementation of a Blockchain shipping application that is motivated by real supply chain industry demands and designed according to actual industry requirements. The platform consists of a set of different actors, in which each has an assigned role and task within the system. Furthermore, transactions' execution depends on the participants' roles in fulfilment of corresponding shipping service definition behind each smart contract. This chapter describes the Proof of Concept(i.e., User Story) and the system architecture to be implemented. The system architecture includes the design and implementation of the Data Model, Smart Contracts, Query and Access Control List. Furthermore, it describes the Loop Back REST API server methods used to interact with the Hyperledger Fabric's shipping application. All codes for this chapter are listed in **Appendix**A for Data Model, **Appendix**B for Smart Contracts, and **Appendix**C for Query and Access Control List.

5.1 User Story as a Proof of Concept

Shipping processes in the supply chain include different actors in which each has an assigned task and role. This thesis proposes the following major participants as key players in the user story:

- **Buyer**
- **Seller**
- **Shipper**
- **Vendor**
- **Bank**

These participants deliver certain capabilities in the blockchain system in order to build an efficient shipping application's user story as illustrated in Figure 5.1.

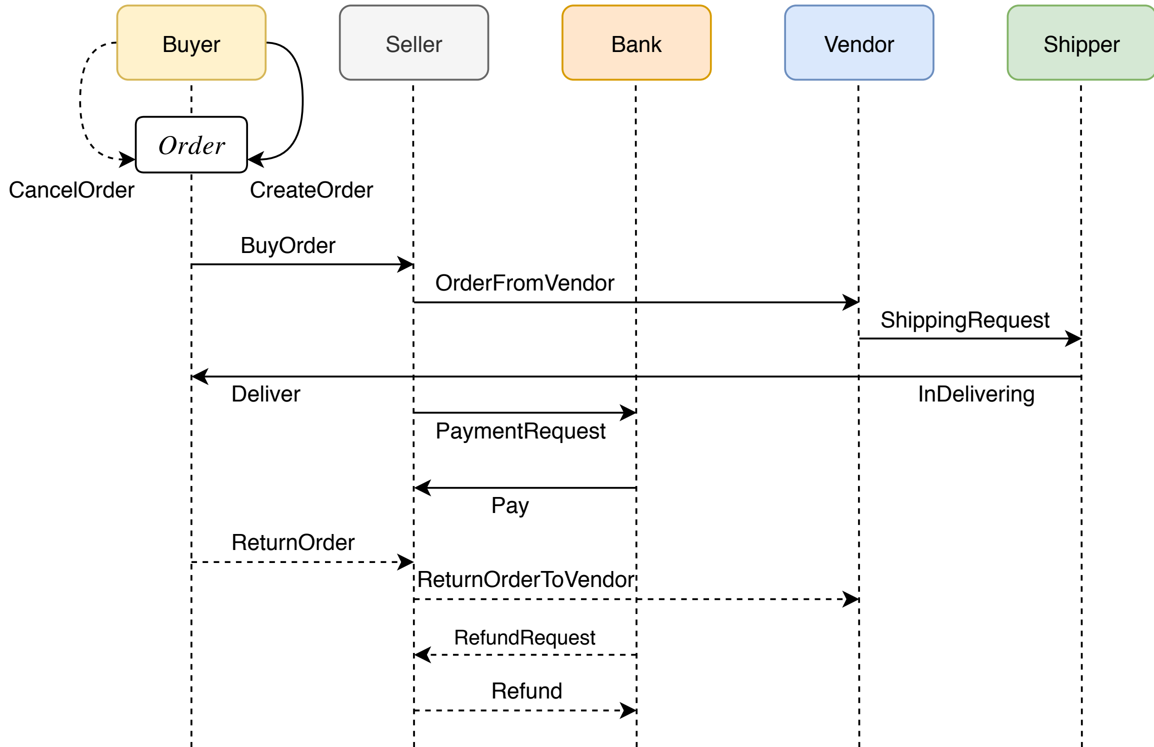

Figure 5.1: Application User Story

A buyer should be able to create an order and send it then to a seller as well as return the order if it is not matched with the initial order or cancel the order. A seller should be able to forward an order to a vendor and request a payment from a bank as well as to return an order to a vendor and refund a bank for a payment refund request. A bank has the capabilities to process a seller's payment request and initialize a payment to a seller. A vendor can create an order ship request to a shipper if a vendor received an order from a seller in earlier stage. Finally, the shipper is responsible to process a ship request and then ship an order to a buyer. A Buyer should be notified if the shipment package has arrived.

Whether the third party is for auditing or inspecting the order, this proposed system eliminates the need for their involvements, since the smart contracts' transactions between these participants are tamper-free and transparent. Furthermore, the proposed system is decentralized since there is no central authority holding the verification processes and information flow within the system.

5.2 System Architecture

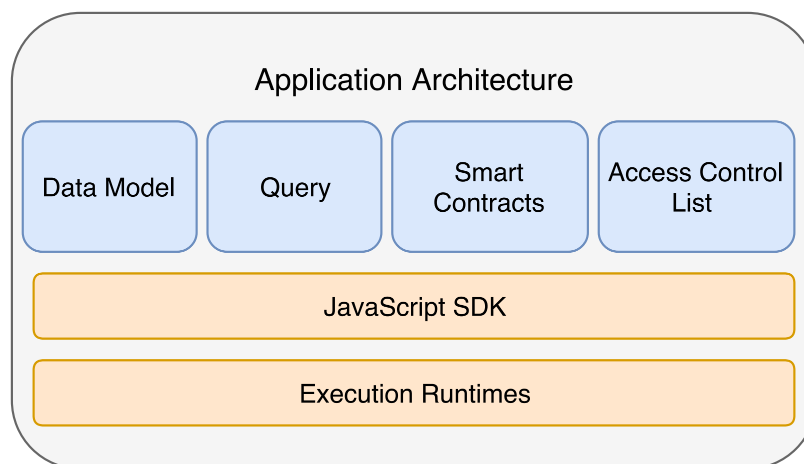

Figure 5.2: Application Architecture Overview

Figure 5.2 shows the architecture overview of the shipping application. This application includes five components which collaborate together to deliver the designated application's capabilities.

Data Model
The Date model is designed to define the data structures in the application network definition. The application network definition includes an organization class name space ***Phoenicia*** in which all resource declarations within this file belong. These resources are Asset, Transaction class, Participants and Events. Asset allows the exchange of values over the Blockchain network whether the value is tangible like Houses or intangible like Order. A participant is an assigned player in the network who can create an asset and also exchange it with other participants. A participant interacts with an asset by submitting transactions. A transaction class includes references to an asset and participants, in which a smart contract's execution is realized. An event is emitted to notify a certain participant that a smart contract has been invoked in the Blockchain ledger. It has references to asset and participants that are assigned to receive a particular event.

Smart Contracts:
Smart contracts are written in JavaScript and execute the associated transaction class (and events if any) for each service capability's logic, as explained in Section 5.1, within the application. The execution's runtime is parsed into the Blockchain

ledger as a Blockchain transaction.

Query

Queries are written in a query language. It retrieves JSON data from Blockchain ledger when pre-defined query conditions are met. Named queries are implemented in the application and invoked into the Blockchain ledger as GET method within the REST API component. Queries contain a description and a statement. Query description is a String that determines the function of the query, while query statement contains the operators and functions that control the query behavior. These operators are SELECT, WHERE, AND, OR, ORDER BY and might be used in combination depends on the desired result of performing a specific query function.

Access Control List

Access control list is implemented to provide declarative access control over the participants in the name space domain Phoenicia. It controls the access permission right of participants to perform Create, Read, Update or Delete operations on the network resources level. Furthermore, It controls the authorization right of a system administrator to access the application itself. System administrator can perform a Create Identity operation for participants in order to allow them to interact with the network resources. Moreover, the identity documents can be validated to prove the identity of a certain participant.

JavaScript SDK

JavaScript SDK is a set of Node.JS APIs within the Hyperledger Fabric Composer environment to manage and interact with the deployed application. The APIs are split into two NPM (Foot REF) modules: composer-client and composer-admin. Composer-client is implemented in order to submit transactions to the application and to perform Create, Read, Update, and Delete operations on assets and participants. However, composer-admin is implemented to deploy, install and launch the application on Hyperledger Fabric version 1.1 (v1.1).

Hyperledger Fabric Execution Runtime

It specifies how to connect the application to the execution runtime of the Hyperledger Fabric version 1.1 that is installed and deployed locally on this thesis's development machine (Mac Book Pro).

Figure 5.3 illustrates the system architecture overview of the shipping application. As the application is deployed successfully on Hyperledger Fabric v1.1, the client initiates the REST API services in order to manipulate their various capabilities. These

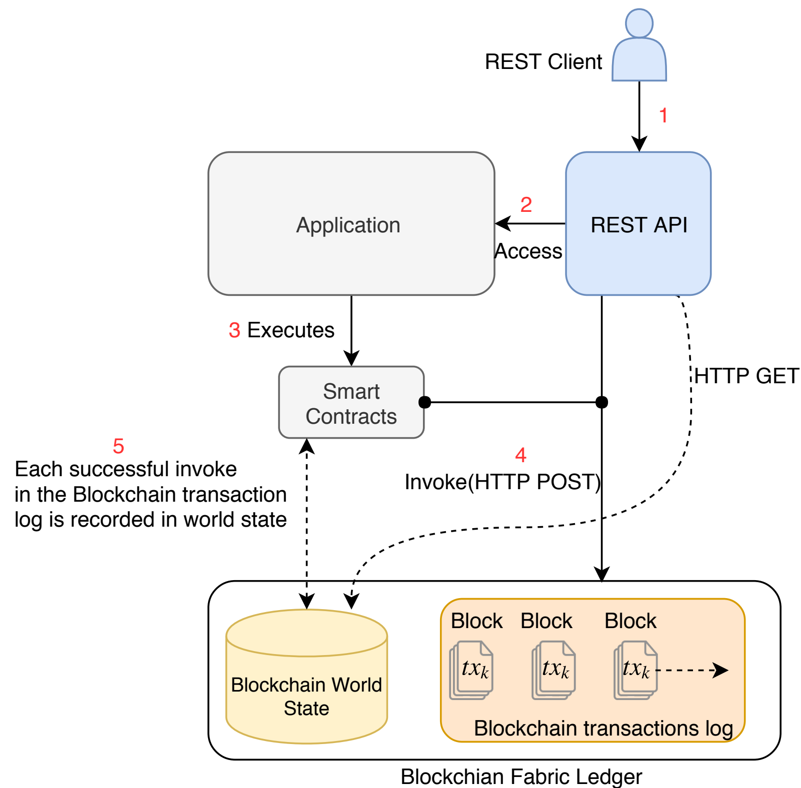

Figure 5.3: System Architecture Overview of the Application

capabilities enable the client to interact efficiently with the system and determine how system resources are defined and addressed. The REST API protocol simplifies the interactions between the REST client and the system using the following HTTP methods as implemented for this thesis:

HTTP POST
Create a resource by requesting the application to execute the desired smart contract and to then submit the transaction's JSON payload (result of execution) to the Hyperledger Fabric's peer (running consensus) as a means to invoke it.

HTTP GET
Retrieve a resource from the Blockchain world state in form of JSON payload.

HTTP PUT
Update a resource.

HTTP DELETE
Delete a resource.

Every time the transaction is invoked in the Hyperledger Fabric's transactions log, the JSON payload is shared and recorded in the Blockchain world state. This process ensures that what is recorded in the world state reflects the invoked transactions in the transactions log.
If the client sends an HTTP GET or queries the ledger with a certain condition, it should return the JSON payload data as accurately expected from the client perspective (i.e., the retrieved data matches the data that invoked through HTTP POST in previous stage). Furthermore, this expectation expands to the situation If HTTP GET is sent after either HTTP PUT or HTTP DELETE is applied on the same resource.

5.3 Design and Implementation

This system architecture enables a decentralized and a proof of concept supply chain's shipping application to be designed and implemented across different participants. This section describes the technologies, which are used in the implementation of this architecture. The implementation utilizes Hyperledger Fabric Blockchain, which is a platform for the deployment of private and permissioned Blockchains.

5.3.1 Data Model Implementation

The implementation uses different resources in order to define the data structure domain of the application. Each of these resources has a list of attributes that are assigned to them as part of the network definition as follows:

Definition I. Participants

- Participant **Buyer** is identified by **buyerID** of type string.
- Participant **Seller** is identified by **sellerID** of type string.
- Participant **Shipper** is identified by **shipperID** of type string.
- Participant **Vendor** is identified by **VendorID** of type string.
- Participant **Bank** is identified by **bankID** of type string.

Definition II. Asset
The implementation defines Order as the asset in this Blockchain network. Table 5.1 shows the asset **Order** that is identified by **orderID** of type string and has different properties that are used by the transactions and the smart contracts.

Table 5.1: Asset Order

Asset: Order		
Property	**Type**	**Description**
product	String	The product to be ordered
status	String	The order's status in the application life cycle
quantity	Integer	Quantity of the order's product
ReasonForCancelling	String	Buyer's reason to cancel an order
reasonForReturning	String	Buyer's reason to return an order
refundAmount	Integer	Refund amount paid to bank
totalBalanceDue	Integer	Total balance payment requested from seller
reference to Vendor	String	vendorID
reference to Shipper	String	shipperID
reference to Buyer	String	buyerID
reference to Seller	String	sellerID
reference to Bank	String	bankID
OrderCreatedDate	String	When buyer creates the order
OrderBuyDate	String	When buyer requests order form seller
OrderVendorDate	String	When seller requests order from vendor
requestShipmentDate	String	When vendor requests shipping order from shipper
deliveringDate	String	When order's product is being sent to buyer
deliveredDate	String	When order's product is delivered to buyer
paymentRequestDate	String	When seller requests payment from bank for an order
payDate	String	When bank pays seller for an order
OrderCancelDate	String	When buyer cancels order
returnOrderDate	String	When buyer returns order to seller
returnOrderToVendorDate	String	When seller returns order to vendor
RefundRequestDate	String	When bank requests refund from seller
OrderRefundDate	String	When seller refunds bank for a cancelled or returned an Order

Definition III. Transaction Class
Each transaction class includes references to asset and participants, in which its associated smart contract consumes them while executing. We made the following transactions for this thesis:

CreateOrder
This transaction class includes a set of specific data attributes that entitle a Buyer to create an Order. It has references to an asset Order and a participant Buyer as well as a property quantity of type integer and property product of type string. This transaction's data are supplied by a Buyer in JSON format and then consumed by its associated smart contract while executing. **Buy**:
This transaction class includes a set of specific data attributes that entitle a Buyer to buy an Order from a Seller. It has references to an asset Order and the both participants Buyer and Seller. This transaction's data are supplied by a Buyer in JSON format and then consumed by its associated smart contract while executing.

OrderVendor
This transaction class includes a set of specific data attributes that entitle a Seller to request an Order from a Vendor. It has references to an asset Order and the both participants Seller and Vendor. This transaction's data are supplied by a Seller in JSON format and then consumed by its associated smart contract while executing.

RequestShipping
This transaction class includes a set of specific data attributes that entitles a Vendor to request a Shipper to ship an Order. It has references to an asset Order and the both participants Vendor and Shipper. This transaction's data are supplied by a Vendor in JSON format and then consumed by its associated smart contract while executing.

InDelivering
This transaction class indicates the status in which an Order is being shipped to a Buyer. It has references to an asset Order and a Shipper since the required properties are appended into an Order in previous transactions. The transaction is triggered by a Shipper in JSON format and then executed by its associated smart contract.

Deliver

This transaction class indicates the status in which an Order is delivered to a Buyer. It has a reference to an asset Order and a Shipper. It is triggered by a Shipper in JSON format and then executed by its associated smart contract.

PaymentReq

Once an Order is received by a participant Buyer, a participant Seller initiates a transaction to request a payment from a Bank. This transaction class has references to an asset Order, participant Bank and special property totalBalanceDue of type float that indicates the Order's payment value. This transaction is triggered by a Seller in JSON format in which is then executed by its associated smart contract.

Pay

Once a payment request is received by a participant Bank, this Bank initiates a transaction to indicate the status in which an order's total balance is being paid to a Seller. Furthermore, it has references to an asset Order and the both participants Seller and Bank. This transaction is triggered by a Bank in JSON format and then executed by its associated smart contract.

ReturnOrder

This transaction class indicates the status in which a Buyer returns an Order to a Seller. It has references to an asset Order, participant Seller, participant Buyer and special property reasonForReturning of type String. The transaction is triggered by a Buyer in JSON format and then executed by its associate smart contract.

ReturnOrderVendor

Once the returning order is received by a participant Seller, this Seller initiates a transaction to forward an Order back to a Vendor. It has references to an asset Order and the both participants Seller and Vendor. This transaction is triggered by a Seller in JSON format and then executed by its associated smart contract.

RefundRequest

A Seller initiates a transaction as a result of ReturnOrder transaction in order to request an Order refund from a Bank. It has references to an asset Order and the both participants Seller and Bank. This transaction is triggered by a Seller in JSON format and then executed by its associated smart contract.

Refund

Once the RefundRequest transaction is received by a participant Bank, this bank initiates a transaction to refund a Seller. It has references to an asset Order, participant Seller, participant Bank and special property refundAmount of type Integer. This transaction is triggered by a Bank in JSON format and then executed by its associated smart contract.

CancelOrder

This transaction class indicates the status in which a Buyer cancels Buy or CreatOrder transaction. It has references to an asset Order, participant Buyer, participant Seller and special property ReasonForCancelling of type String.

Definition IV. Events

Events are declared in the Data Model and emitted during the smart contracts' execution runtime.

- Event **BuyNotification** is to notify a **Seller** with the Buy transaction.
- Event **CancelNotification** is to notify a **Buyer** with the CancelOrder transaction.
- Event **VendorNotification** is to notify a **Vendor** with the OrderVendor.
- Event **OnRouteNotification** is to notify a **Buyer** with the InDelivering transaction.
- Event **DeliveredNotification** is to notify a **Seller** with the Pay transaction.
- Event **RefundNotification** is to notify a **Bank** with the Refund transaction.
- Event **ReturnNotification** is to notify a **Seller** with the ReturnOrder.

5.4 Smart Contract Implementation

Each Smart Contract consumes different resources (i.e., Participants, Asset, Events, Transactions), and is realized within a pre-defined logic that leads to deliver the Hyperledger Fabric application capabilities as explained in Section 5.1.

As shown in Table 5.2, the JSON object's StatusList defines Key-Value pairs that are used to append Value into a property Order Status of type String during the execution

runtime of a smart contract. This appended value is meant for the participants (REST clients) to realize the changes of the Status in an asset Order during the executions of different smart contracts. For example, the status of an Order before the execution of the smart contract Buy was a "Created Order" while after the execution, the status becomes "Bought Order." In this case, a participant Seller (REST client) can look up the history of a specific asset Order through the changes of the Status in this Order.

Table 5.2: JSON Object: StatusList

JSON Object: StatusList	
Key	**Value**
OrderCreateStatus	Created Order
OrderBuyStatus	Bought Order
OrderCancelStatus	Cancelled Order
OrderFromVendorStatus	Seller Sent Order to the Vendor
ShipRequestStatus	Vendor Requests Shipper to Ship Order
DeliveredStatus	Deliver Order
InDeliveringStatus	Order On The Way
OrderReturnStatus	Return Order to Seller
PaymentReq	Seller Requests Bank to Pay
PayStatus	Payment Processed
RefundReq	Bank Requests Seller to Pay Back
Refunded	Refunded Order
ReturnOrderVendorStatus	Returned Order to Vendor

The Following subsections describe the Smart Contracts implementation.

5.4.1 CreateOrder Smart Contract

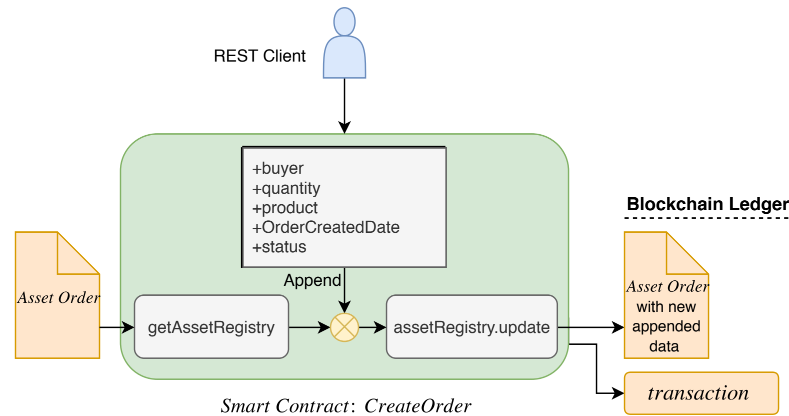

Figure 5.4: CreateOrder Workflow

This smart contract as illustrated in Figure 5.4, has the following characteristic:

- **Execution** It is executed by a participant Buyer using REST client (HTTP POST) method and consumes both the asset Order and transaction CreateOrder during the execution runtime.

The supplied data is appended by Buyer into the Order using the Hyperledger Fabric's (HL) getAssetRegistry and assetRegistry.update modules. These Fabric's modules define the process of retrieving the Order from Blockchain ledger and update it with this supplied data. The smart contract is invoked as a new transaction that is added into the Blockchain transactions log. This transaction includes the information of transactionID (transaction hash) and timestamp that describes the transaction's occurrence date.

5.4.2 Buy Smart Contract

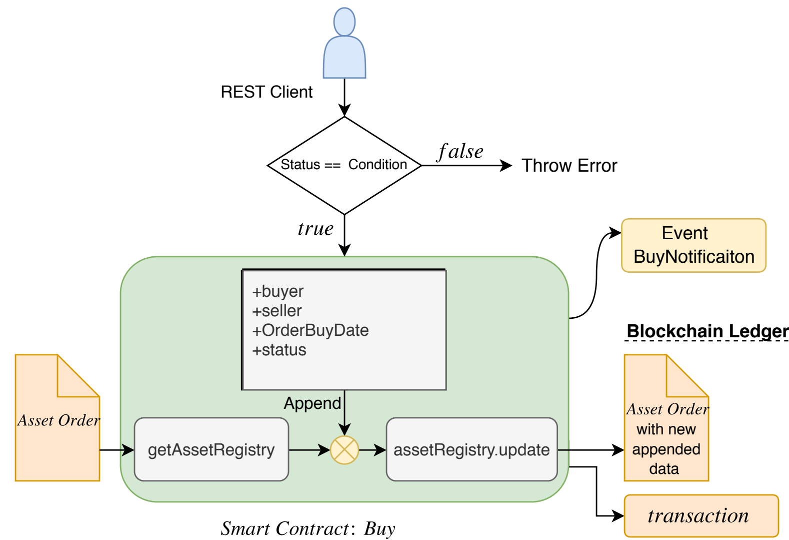

Figure 5.5: Buy Workflow

This smart contract as illustrated in Figure 5.5, has the following characteristics:

- **Execution** It is executed by a participant Buyer using HTTP POST method and consumes both the asset Order and transaction Buy during the execution runtime.

- **Execution's Condition** The status of received Order must match the StatusList's key-value of OrderCreateStatus, Else the following error is generated: create the order first.

The supplied data is appended by Buyer into the Order using the HL's modules. Additionally, the smart contract is invoked as a new transaction that is added into the Blockchain transactions log and emits the event BuyNotification as explained in Section 5.4.1 Definition III. This transaction includes the information of transactionID (transaction hash) and timestamp.

5.4.3 OrderFromVendor Smart Contract

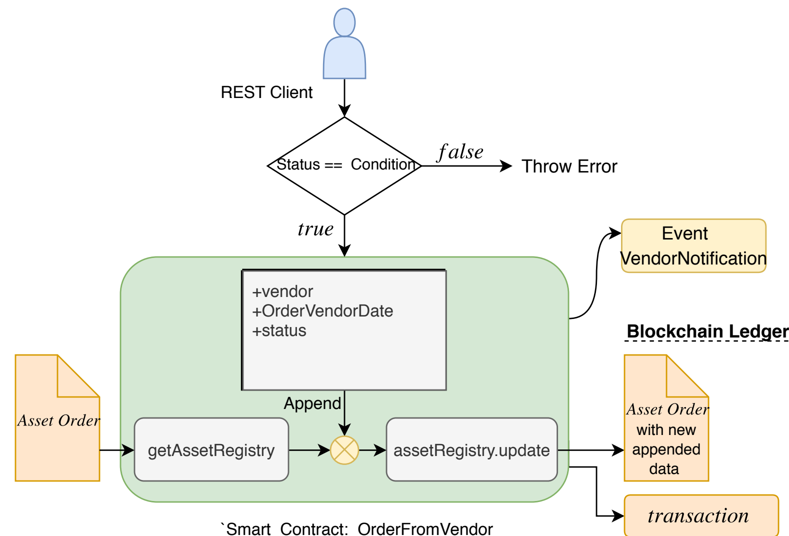

Figure 5.6: OrderFromVendor Workflow

This smart contract as illustrated in Figure 5.6, has the following characteristics:

- **Execution** It is executed by a participant Seller using HTTP POST method and consumes both the asset Order and transaction OrderFromVendor during the execution runtime.
- **Execution's Condition** The status of received Order must match the StatusList's key-value of OrderBuyStatus, Else the following error is generated: The vendor has already received this order.

The supplied data is appended by Seller into the Order using the HL's modules. Additionally, the smart contract is invoked as a new transaction that is added into the Blockchain transactions log. This transaction includes the information of transactionID (transaction hash) and timestamp.

5.4.4 RequestShipping Smart Contract

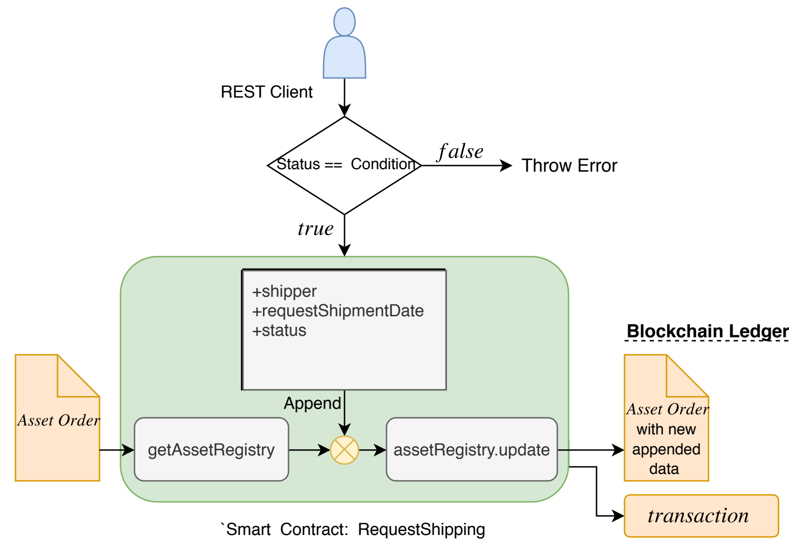

Figure 5.7: RequestShipping Workflow

This smart contract as illustrated in Figure 5.7, has the following characteristics:

- **Execution** It is executed by a participant Vendor using HTTP POST method and consumes both the asset Order and transaction RequestShipping during the execution runtime.

- **Execution's Condition** The status of received Order must match the StatusList's key-value of OrderVendorStatus, Else the following error is generated: The order has already been requested for shipping.

The supplied data is appended by Vendor into the Order using the HL's modules. Additionally, the smart contract is invoked as a new transaction that is added into the Blockchain transactions log. This transaction includes the information of transactionID (transaction hash) and timestamp.

5.4.5 InDelivering Smart Contract

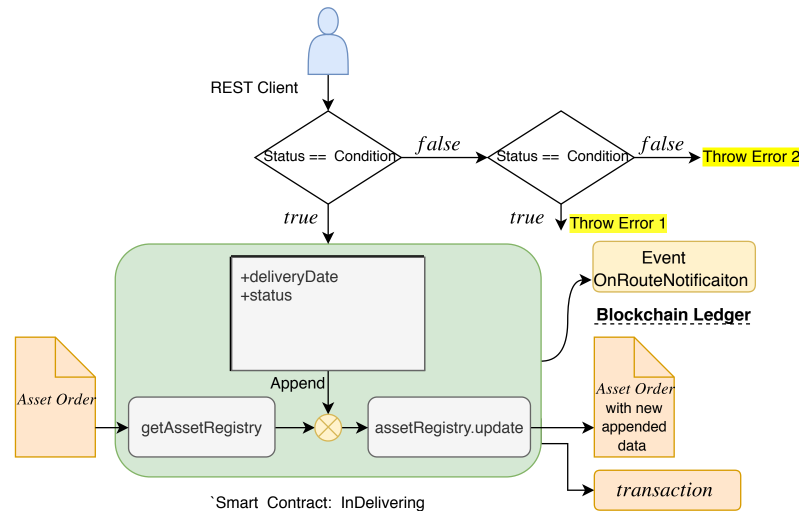

Figure 5.8: InDelivering Workflow

This smart contract as illustrated in Figure 5.8, has the following characteristics:

- **Execution** It is executed by a participant Shipper using HTTP POST method and consumes both the asset Order and transaction InDelivering during the execution runtime.

- **Execution's Condition** The status of received Order must match the StatusList's key-value of ShipRequestStatus, Else if it matches the key-value DeliveredStatus, the following error is generated: The buyer has already received this order. If neither previous condition is met, the following error is generated: The order has not been requested for shipping yet

The supplied data is appended by Shipper into the Order using the HL's modules. Additionally, the smart contract is invoked as a new transaction that is added into the Blockchain transactions log and emits the event OnRouteNotification. This transaction includes the information of transactionID (transaction hash) and timestamp.

5.4.6 Deliver Smart Contract

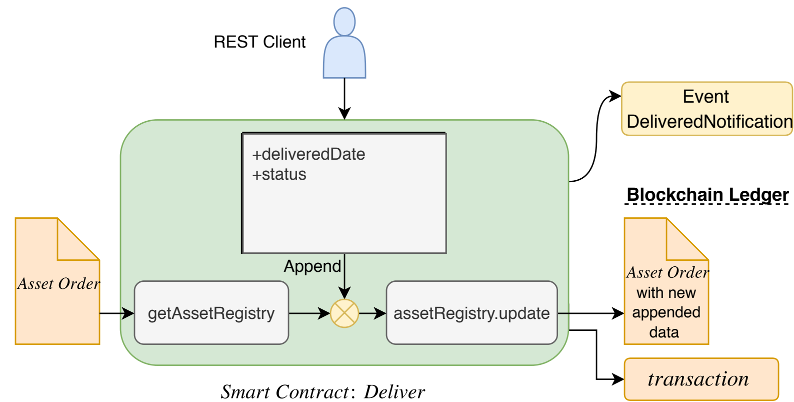

Figure 5.9: Deliver Workflow

This smart contract as illustrated in Figure 5.9, has the following characteristic:

- **Execution** It is executed by a participant Shipper using HTTP POST method and consumes both the asset Order and transaction Deliver during the execution runtime.

The supplied data is appended by Shipper into the Order using the HL's modules. Additionally, the smart contract is invoked as a new transaction that is added into the Blockchain transactions log and emits the event DeliveredNotification. This transaction includes the information of transactionID (transaction hash) and timestamp.

5.4.7 PaymentReq Smart Contract

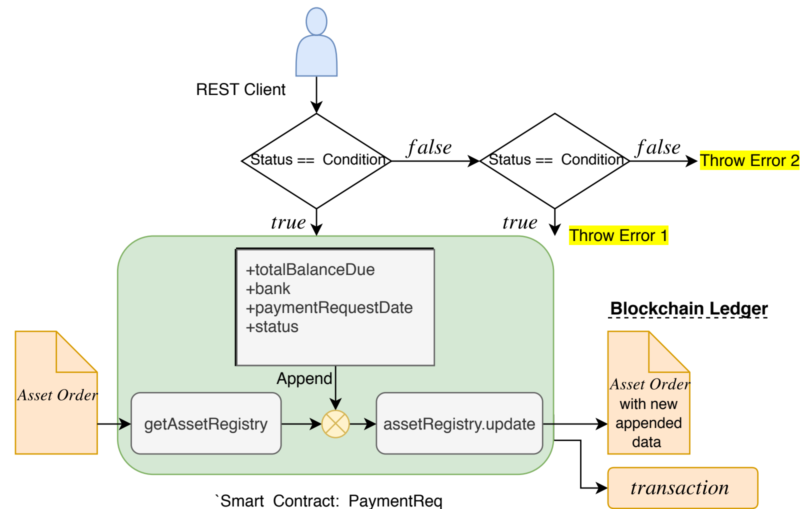

Figure 5.10: PaymentReq Workflow

This smart contract as illustrated in Figure 5.10, has the following characteristics:

- **Execution** It is executed by a participant Seller using HTTP POST method and consumes both the asset Order and transaction PaymentReq during the execution runtime.

- **Execution's Condition** The status of received Order must match the StatusList's key-value of DeliveredStatus, Else if it matches the key-value InDeliveringStatus, the following error is generated: The order is still on route and payment Request is not allowed. If neither previous condition is met, the following error is generated: The order has been already requested for payment.

The supplied data is appended by Seller into the Order using the HL's modules. Additionally, the smart contract is invoked as a new transaction that is added into the Blockchain transactions log. This transaction includes the information of transactionID (transaction hash) and timestamp.

5.4.8 Pay Smart Contract

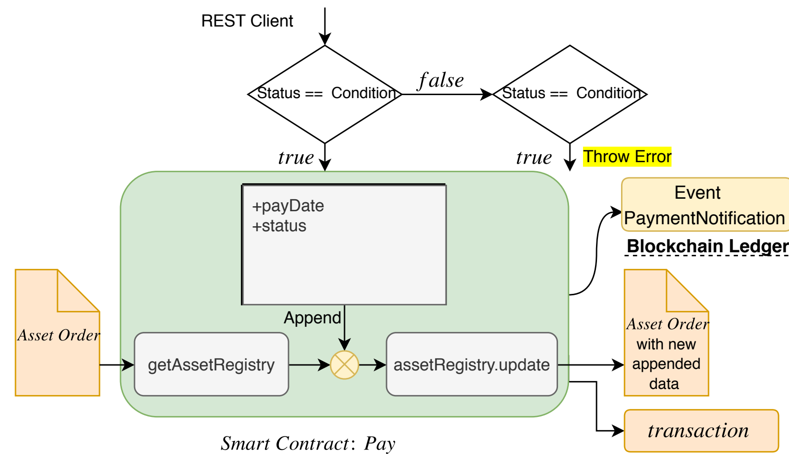

Figure 5.11: Pay Workflow

This smart contract as illustrated in Figure 5.11, has the following characteristics:

- **Execution** It is executed by a participant Bank using HTTP POST method and consumes both the asset Order and transaction Pay during the execution runtime.

- **Execution's Condition** The status of received Order must match the Status-List's key-value of PaymentReq, Else if it matches the key-value PayStatus, the following error is generated: The order has already been paid.

The supplied data is appended by Bank into the Order using the HL's modules. Additionally, the smart contract is invoked as a new transaction that is added into the Blockchain transactions log and emits the event PaymentNotification. This transaction includes the information of transactionID (transaction hash) and timestamp.

5.4.9 ReturnOrder Smart Contract

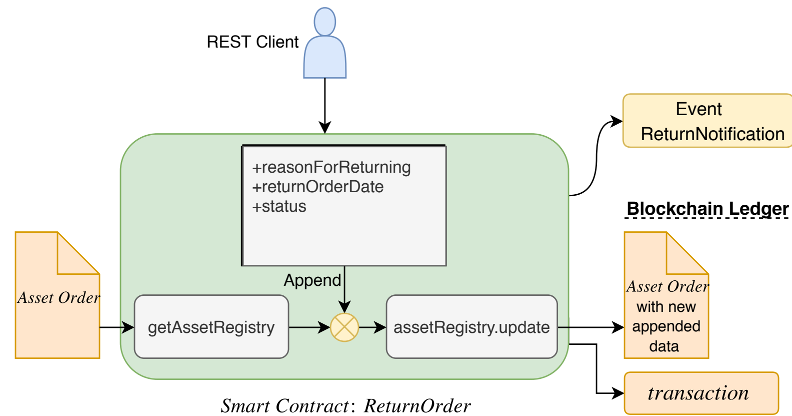

Figure 5.12: ReturnOrder Workflow

This smart contract as illustrated in Figure 5.12, has the following characteristic:

- **Execution** It is executed by a participant Buyer using HTTP POST method and consumes both the asset Order and transaction ReturnOrder during the execution runtime.

The supplied data is appended by Buyer into the Order using the HL's modules. Additionally, the smart contract is invoked as a new transaction that is added into the Blockchain transactions log and emits the event ReturnNotification. This transaction includes the information of transactionID (transaction hash) and timestamp.

5.4.10 ReturnOrderVendor Smart Contract

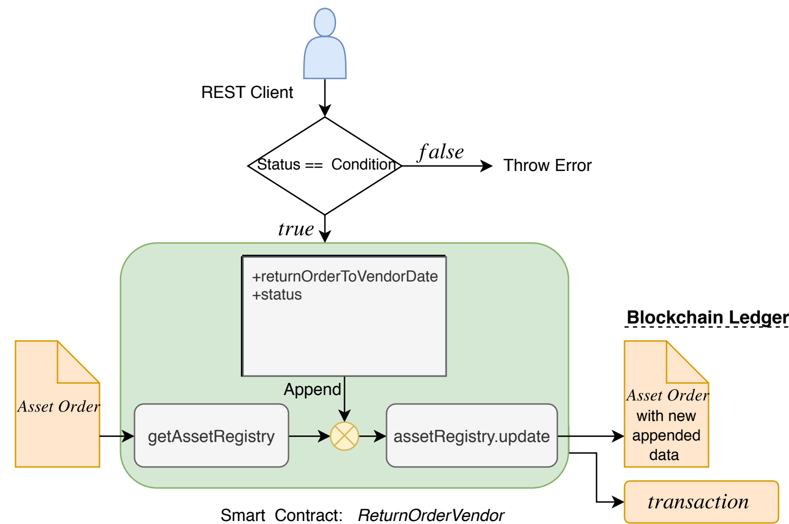

Figure 5.13: ReturnOrderVendor Workflow

This smart contract as illustrated in Figure 5.13, has the following characteristics:

- **Execution** It is executed by a participant Seller using HTTP POST method and consumes both the asset Order and transaction ReturnOrderVendor during the execution runtime.

- **Execution's Condition** The status of received Order must match the StatusList's key-value of OrderReturnStatus, Else the following error is generated: The buyer did not return the order.

The supplied data is appended by Seller into the Order using the HL's modules. Additionally, the smart contract is invoked as a new transaction that is added into the Blockchain transactions log. This transaction includes the information of transactionID (transaction hash) and timestamp.

5.4.11 RefundRequest Smart Contract

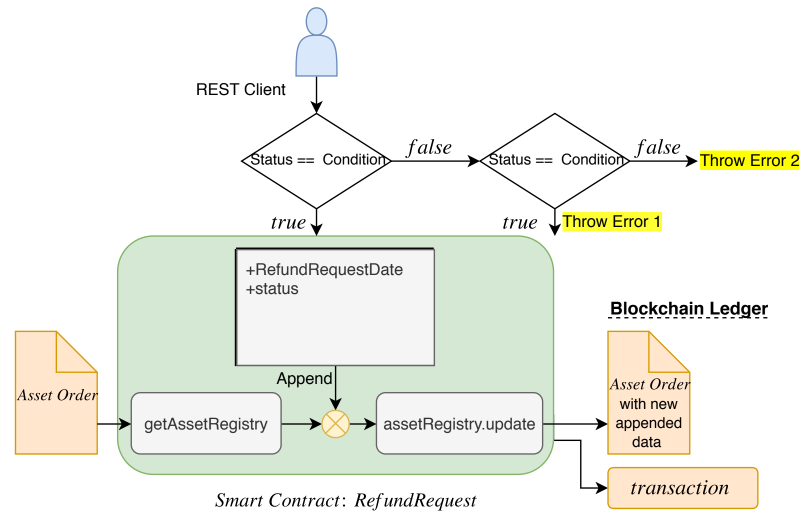

Figure 5.14: RefundRequest Workflow

This smart contract as illustrated in Figure 5.14, has the following characteristics:

- **Execution** It is executed by a participant Bank using HTTP POST method and consumes both the asset Order and transaction RefundRequest during the execution runtime.
- **Execution's Condition** The status of received Order must match the StatusList's key-value of OrderReturnStatus, Else if it matches the key-value RefundReq, the following error is generated: The order has already been requested for refund. If neither condition is met, the following error is generated: The order has not been returned to Seller

The supplied data is appended by Bank into the Order using the HL's modules. Additionally, the smart contract is invoked as a new transaction that is added into the Blockchain transactions log and emits the event PaymentNotification. This transaction includes the information of transactionID (transaction hash) and timestamp.

5.4.12 Refund Smart Contract

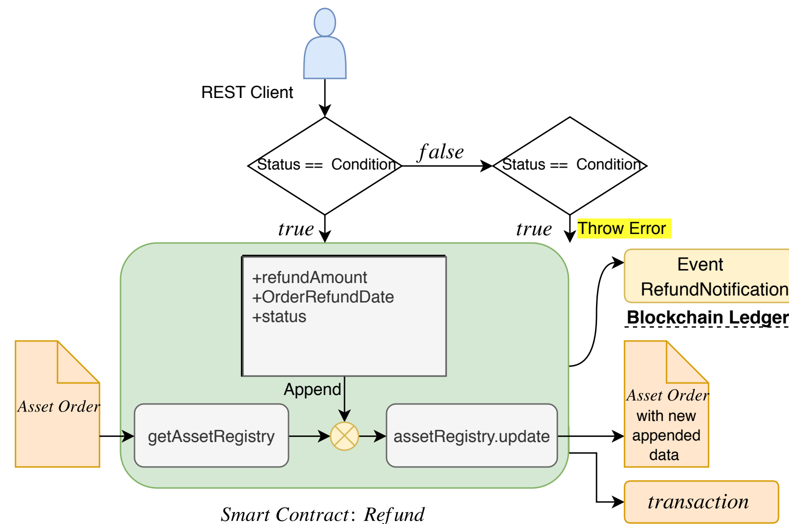

Figure 5.15: Refund Workflow

This smart contract as illustrated in Figure 5.15, has the following characteristics:

- **Execution** It is executed by a participant Seller using HTTP POST method and consumes both the asset Order and transaction Refund during the execution runtime.

- **Execution's Condition** The status of received Order must match the Status-List's key-value of RefundReq, Else if it matches the key-value Refunded, the following error is generated: The order has already been refunded.

The supplied data is appended by Bank into the Order using the HL's modules. Additionally, the smart contract is invoked as a new transaction that is added into the Blockchain transactions log and emits the event RefundNotification. This transaction includes the information of transactionID (transaction hash) and timestamp.

5.4.13 CancelOrder Smart Contract

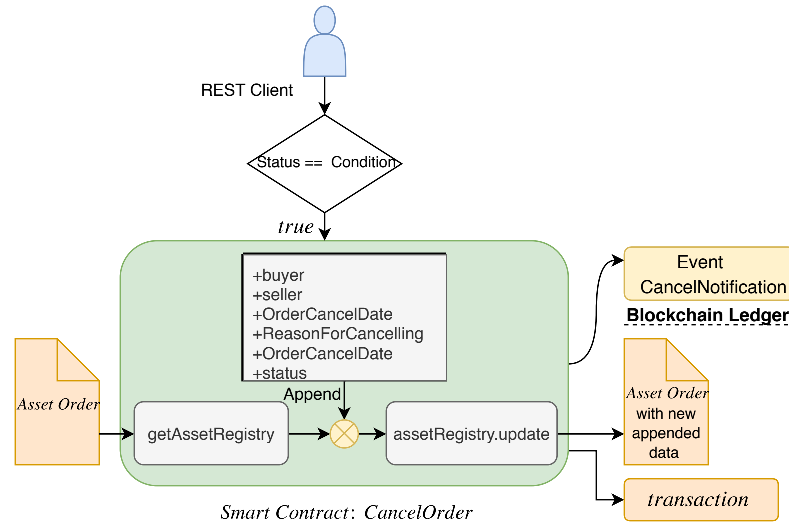

Figure 5.16: CancelOrder Workflow

This smart contract as illustrated in Figure 5.16, has the following characteristics:

- **Execution** It is executed by a participant Buyer using HTTP POST method and consumes both the asset Order and transaction Refund during the execution runtime.

- **Execution's Condition** The status of received Order must match the StatusList's key-value of OrderCreatedStatus or OrderBuyStatus

The supplied data is appended by Buyer into the Order using the HL's modules. Additionally, the smart contract is invoked as a new transaction that is added into the Blockchain transactions log and emits the event CancelNotification. This transaction includes the information of transactionID (transaction hash) and timestamp

5.5 Query Implementation

Query is a powerful method to retrieve JSON data that was written previously into the Blockchain ledger through Smart Contracts execution, when a query condition is met. The following query scenarios are implemented and delivered:

- **Query Orders** returns all orders that are in the Blockchain ledger.

- **OrderByBuyer** return the orders that are in the Blockchain ledger and belong to a certain participant Buyer.
- **OrderBySeller** returns the orders that are in the Bockchain ledger and belong to a certain participant Seller.
- **OrderByQuantity** returns the orders that are in the Blockchain ledger and have a quantity property's value that matches a pre-defined quantity's value.
- **OrdersByProduct** returns the orders that are in the Blockchain ledger and have a product property's value that matches a pre-defined product's value.
- **OrdersHistorian** returns the history of an order if the participant ID (that is supplied by the participant in Query payload request) matches the participant ID that is already in this order. The history includes the transactionIDs and timestamps along with the order's status and the information of the participant who received this order at each timestamp.

All queries are run as GET methods within the REST API component. Furthermore, those queries scenarios are chosen to demonstrate the capabilities of querying a Blockchain ledger. Thus, there are various scenarios that can be implemented; depending on the user's needs and the user case's requirements.

5.6 Access Control List Implementation

The implementation of access control has a vital role in applying the private Blockchain feature of this application. It ensures that no participant can do Create, Read, Update, and/or Delete (CRUD) operations on the network resources (i.e., Asset and Transactions) unless this participant has the permission to do that. Thus, it adds a powerful privacy and security layer to the application. The following access rule scenarios are implemented and delivered:

Asset Rule
Each participant in the network domain can only perform all operations only on their own asset.

Personal Information Rule

All participants in the network domain can only perform Create, Read and Update operations on their own information instance. For example, If a participant Buyer with buyerID = "UVic" was approved to join the network, this participant cannot perform Create, Delete or Update (CRUD) on a participant Seller instance. This to prevent participants to defraud the system by having access into other participants data.

Each participant in the network domain can only perform all operations on their own asset. For instance, participant Buyer cannot invoke CancelOrder smart contract on Order that belongs to another participant Buyer.

Transactions Rule

Participant Buyer can only perform all operations on the CreateOrder, CancelOrder, Buy and ReturnOrder transactions (using the associate smart contract). Participant Seller can only perform all operations on the OrderFromVendor, PaymentReq, Refund and ReturnOrderVendor transactions. Participant Bank can only perform all operations on the Pay and RefundRequest transactions. Participant Vendor can only perform all operations on the ReqeustShipping transaction. Participant Shipper can only perform all operations on the Deliver and InDelivering transactions.

Blockchain's Transactions History

All participants can only perform Read operation on their history of previous transactions.

5.7 Summary

This chapter described the design, implementation and delivery path of the Blockchain Shipping application's capabilities. Furthermore, this chapter reveals how the system architecture components are relational dependencies. To invoke the smart contract successfully, it needs to consume the JSON data that is declared in its designated data model's transaction class and the asset *Order*. This asset *Order* is updated with each invocation of a smart contract to reflect the most recent invocation. However, it is still possible to query the history of the asset *Order* to find when and how this asset *Order* has been updated.

Events are emitted as soon as the smart contract is invoked to notify a participant that the invocation has occurred. Furthermore, the access control list was defined while

the conditions were coded in the Smart Contract code to meet the common supply chain requirements with whom and what each participant can do in the application. For instance, participant Sellers cannot submit a request to participant Vendors to cancel the *Order*. Moreover, Sellers must meet a condition in which they cannot submit a payment request smart contract until the shipment has been successfully delivered to Buyers. This is to ensure the correctness of the application's smart contracts execution work flow and thus meets the application user story's work flow.

Chapter 6

Discussion and Results

This chapter presents the capabilities of the subject application discussed in this thesis. These capabilities are the result of the implementation stage explained in Chapter 5. We present a summary of the development and the application's launching key steps to run this application. The results of the user story's implementation are shown in Figure 5.1. Finally, we discuss the results obtained.

6.1 Development Environment

The following pre-requisite steps are taken in order to set up the development environment:

- Node Version Manager (NVM) package was installed to allow a seamless switch between different versions of npm for Node.JS framework.
- JavaScript SDK was installed and then connected with the application by setting a package.json connection file.
- The Hyperledger Fabric execution runtime was installed and deployed locally.
- Docker container was installed to host the application and the Hyperledger Fabric runtime images locally.

The following steps are taken in order to run the application:

- A peer admin card was created and contains this application developer's credentials to deploy the application on the Hyperledger Fabric execution runtime

and perform operations on the application (e.g., submit transactions, retrieve data and create identity cards for each participant).

- The application participants' identity cards were created and contain the participants' credentials to access the application and perform operations according to their designated roles in the application.
- The Hyperledger Fabric execution runtime was installed and deployed locally.
- The REST API server was launched in order to allow this application's developer as well as the participants to interact with the application and the Hyperledger Fabric's local environment.

Figure 6.1: Application's REST API Server

6.2 Implementation Results

Table 6.1: Participants JSON Data

Create Participants		
Participant	**Value**	**HTTP POST Response**
Buyer	buyerID : UVic ContactInformation : uvic@me.com	200 Successful
Seller	sellerID: EquibCO ContactInformation: equip@me.com	200 Successful
Vendor	vendorID : VenCO ContactInformation : vendor@me.com	200 Successful
Bank	bankID: BankCO ContactInformation: bank@me.com	200 Successful
Shipper	shipperID : ShipCO ContactInformation : ship@me.com	200 Successful

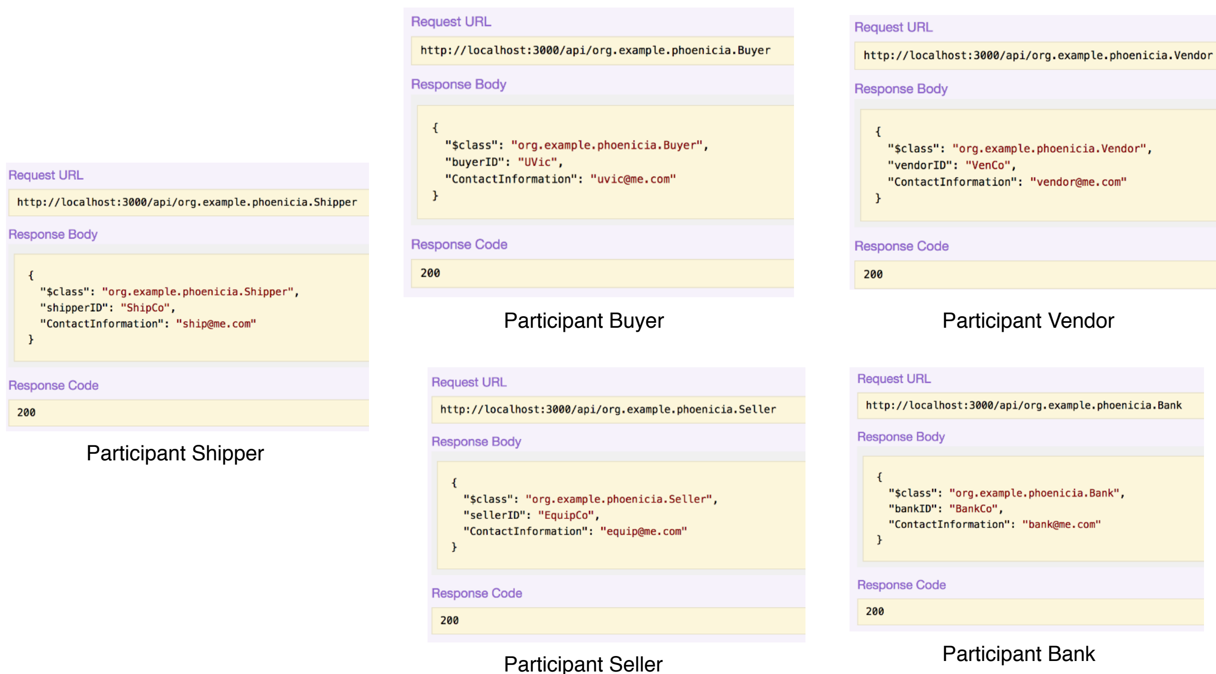

Figure 6.2: HTTP POST Response for Each Participant

Figure 6.1 shows the application's REST API server is up and running. It holds the application APIs' endpoints (i.e., RequestURL as shown in the HTTP Response in Figure 6.2) to allow successful interactions with the application's smart contracts and resources (i.e., Participants, Asset, Transactions, and Queries) as well as the Fabric Blockchain ledger.
In order to demonstrate the capabilities of the application, the participants' JSON data are first created as illustrated in Table 6.1. Figure 6.2 shows the HTTP POST invocation response for each JSON data request. The response code 200 means that the POST request is successfully invoked in the Ledger.

A template of asset Order is created in order to allow the participant Buyer to use it in the CreateOrder transaction. Figure 6.3 (1) and (2) shows the HTTP POST invocation response of CreateOrder smart contract that is executed by the participant Buyer. The POST request is sent to the CreateOrder end point (Request URL) and a body response is retuned as a result of successfully invoking the smart contract in the Fabric Ledger. This response contains the JSON data that was supplied by the participant Buyer before sending the POST request (CreateOrder transaction data as explained in Definition III. Transactions Section 5.3.1). Furthermore, This response has the field timestamp that determines the time in which the transaction (transactionID) is successfully invoked in the Blockchain ledger. The transactionID represents the hash of the transaction CreateOrder that was invoked in the Fabric ledger and added into Fabric transactions log.

(1)HTTP POST Request


```
{
"$class": "org.example.phoenicia.CreateOrder",
"quantity": 30,
"product": ["Monitors"],
"order": {org.example.phoenicia.Order#1},
"buyer": {org.example.phoenicia.Buyer#UVic},
}
```


Parameter content type:
application/json

Request URL

http://localhost:3000/api/org.example.phoenicia.CreateOrder

Response Body


```
{
  "$class": "org.example.phoenicia.CreateOrder",
  "quantity": 30,
  "product": [
    "Monitors"
  ],
  "order": "org.example.phoenicia.Order#1",
  "buyer": "org.example.phoenicia.Buyer#UVic",
  "transactionId": "0b2ffaf329bc5041b260af60eebd2834f2f11fa64076b5c908afed078a5409d3",
  "timestamp": "2018-12-01T10:19:48.105Z"
}
```


Response Code

200

(2) HTTP POST Response

Response Body


```
[
  {
    "$class": "org.example.phoenicia.Order",
    "orderID": "1",
    "product": [
      "Monitors"
    ],
    "status": "\"Created Order\"",
    "OrderRefundedDate": "string",
    "RefundRequestDate": "string",
    "quantity": 30,
    "OrderCreatedDate": "2018-12-01T10:30:34.312Z",
    "OrderBuyDate": "string",
    "OrderCancelDate": "string",
    "OrderVendorDate": "string",
    "returnOrderDate": "string",
    "returnOrderToVendorDate": "string",
    "requestShipmentDate": "string",
    "deliveredDate": "string",
    "deliveringDate": "string",
```


Response Code

200

(3) Asset Order

Figure 6.3: HTTP POST: CreateOrder

HTTP GET result for asset Order is called to show the status of the Order after the CreateOrder smart contract is successfully invoked as shown in Figure 6.3(3). The status of the Order becomes “Created Order” and contains the invocation timestamp as well as the other data that was supplied by the participant Buyer as part of the smart contract invocation.

Figure 6.4 (1) and (2) shows the HTTP POST invocation response of Buy smart contract that is executed by the participant Buyer. The POST request is sent to the Buy end point (Request URL) and a body response is retuned as a result of successfully invoking the smart contract in the Fabric Ledger. This response contains the JSON data that was supplied by the participant Buyer before sending the POST request (Buy transaction data). Furthermore, This response has the field timestamp that determines the time in which the transaction (transactionID) is successfully invoked in the Blockchain ledger. The transactionID represents the hash of the transaction that was invoked in the Fabric ledger and added into Fabric transactions log.

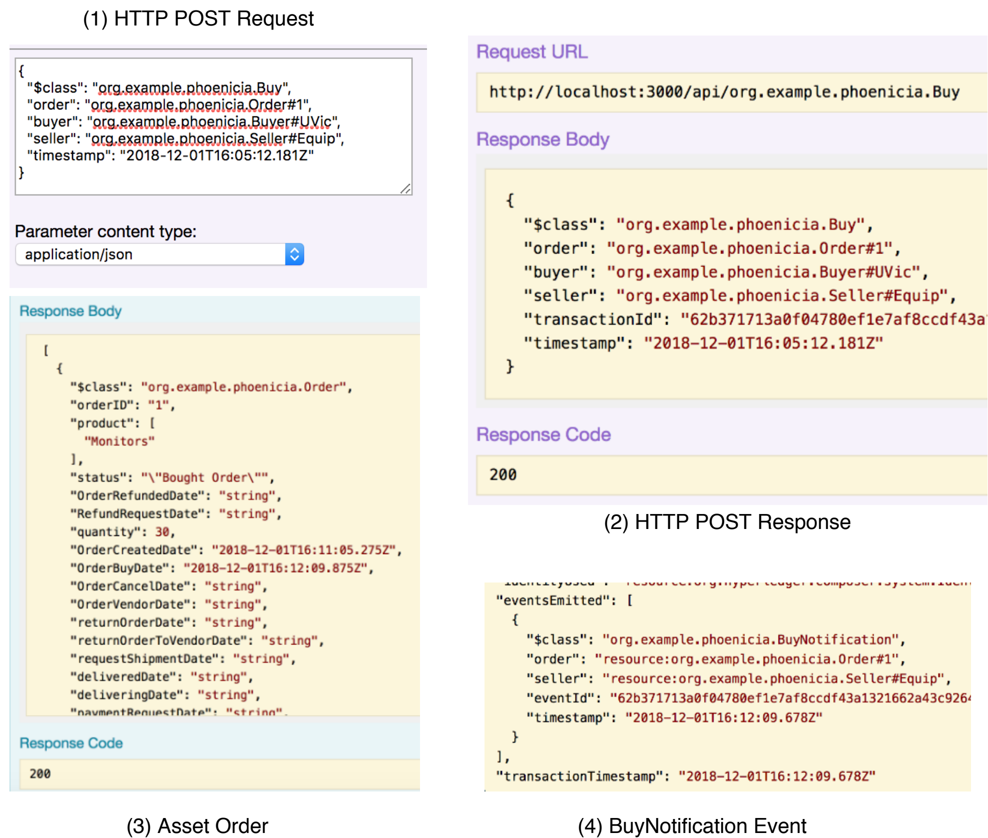

Figure 6.4: HTTP POST: Buy

Figure 6.4 (3) shows the HTTP GET result for asset Order that is called to show the updated status of the order after the Buy smart contract is successfully invoked. The status of the order becomes "Bought Order" and contains the timestamp as well as the other data that was supplied by the participant Buyer.
As a Buy transaction is invoked, the event BuyNotification is emitted to notify the Seller with the buy request as shown in Figure 6.4 (4). The event contains the information about the Order and the sellerID as well as the event ID which represents the hash of transaction BuyNotification event in the transactions log, and the timestamp.

Figure 6.5 (1) and (2) shows the HTTP POST invocation response of OrderFromVendor smart contract that is executed by the participant Seller. The POST request is sent to the Request URL and a body response is retuned as a result of successfully invoking the smart contract in the Fabric Ledger. This response contains the JSON data that was supplied by the participant Seller before sending the POST

request (OrderFromVendor transaction data). Furthermore, This response has the field timestamp that determines the time in which the transaction (transactionID) is successfully invoked in the Blockchain ledger. The transactionID represents the hash of the transaction that was invoked in the Fabric ledger and added into Fabric transactions log.

(1) HTTP POST Request

{
"$class": "org.example.phoenicia.OrderFromVendor",
"order": "org.example.phoenicia.Order#1",
"vendor": "org.example.phoenicia.Vendor#VenCo",
"seller": "org.example.phoenicia.Seller#EquipCo",
"timestamp": "2018-12-01T21:14:18.985Z"
}

Parameter content type:
application/json

Request URL

http://localhost:3000/api/org.example.phoenicia.OrderFromVendor

Response Body

{
"$class": "org.example.phoenicia.OrderFromVendor",
"order": "org.example.phoenicia.Order#1",
"vendor": "org.example.phoenicia.Vendor#VenCo",
"seller": "org.example.phoenicia.Seller#EquipCo",
"transactionId": "e834787742f515bd9724b4b6b6aac55defb02bff3
"timestamp": "2018-12-01T21:14:18.985Z"
}

Response Code

200

(2) HTTP POST Response

Response Body

{
"$class": "org.example.phoenicia.Order",
"orderID": "1",
"product": [
"Monitors"
],
"status": "\"Seller Sent Order to the Vendor\"",
"OrderRefundedDate": "string",
"RefundRequestDate": "string",
"quantity": 30,
"OrderCreatedDate": "2018-12-01T21:19:46.654Z",
"OrderBuyDate": "2018-12-01T21:20:42.261Z",
"OrderCancelDate": "string",
"OrderVendorDate": "2018-12-01T21:22:04.337Z",
"returnOrderDate": "string",
"returnOrderToVendorDate": "string",
"requestShipmentDate": "string",
"deliveredDate": "string",
"deliveringDate": "string",
"paymentRequestDate": "string",
"payDate": "string",

Response Code

200

(3) Asset Order

"eventsEmitted": [
{
"$class": "org.example.phoenicia.VendorNotification",
"order": "resource:org.example.phoenicia.Order#1",
"vendor": "resource:org.example.phoenicia.Vendor#VenCo",
"eventId": "e834787742f515bd9724b4b6b6aac55defb02bff3922f6c
"timestamp": "2018-12-01T21:22:04.094Z"
}
],
"transactionTimestamp": "2018-12-01T21:22:04.094Z"

(4) VendorNotification Event

Figure 6.5: HTTP POST: OrderFromVendor

Figure 6.5 (3) shows the HTTP GET result for asset Order that is called to show the updated status of the order after the OrderFromVendor smart contract is successfully invoked. The status of the order becomes "Seller Sent Order to the Vendor" and contains the timestamp as well as the other data that was supplied by the participant Seller. As a Buy transaction is invoked, the event VendorNotification is emitted to notify the Vendor with the OrderFromVendor request as shown in Figure 6.5 (4). The event contains the information about the Order and the sellerID as well as the event ID which represents the hash of transaction VendorNotification event in the transactions log, and the timestamp.

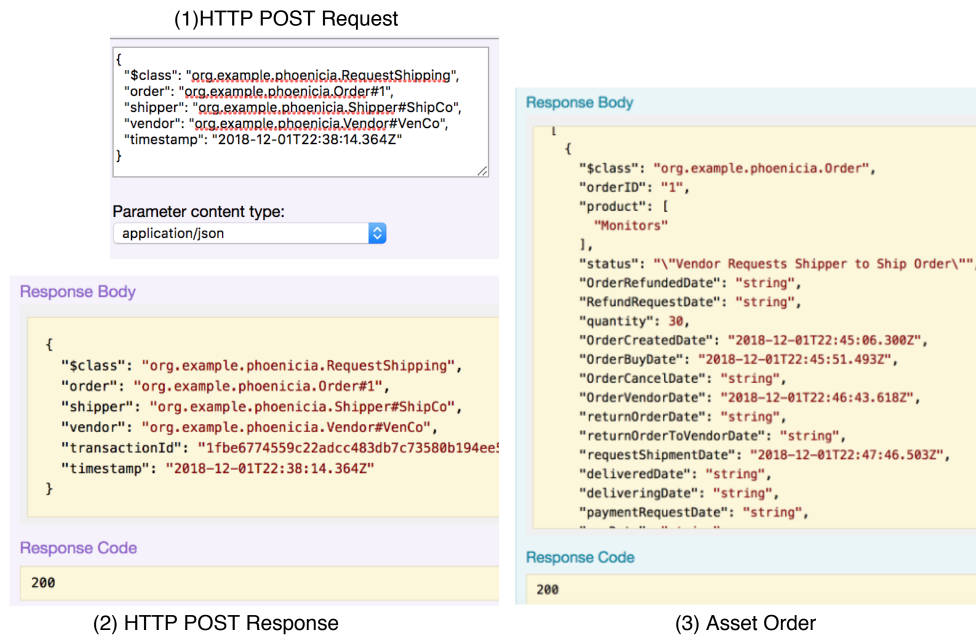

Figure 6.6: HTTP POST: RequestShipping

Once the Order is received by the Vendor. The Vendor checks if the status of the order is valid (as explained in Section 5.4.4). If it is valid, the Vendor sends a request to the Shipper for shipping. This process is done by the invocation of RequestShipping smart contract as shown in Figure 6.6 (1), (2), and (3). The Post request is sent to the RequestShipping end point and a body response is returned. This response includes the JSON data that was supplied by the participant Vendor and both the fields timestamp and transactionID. Furhtemore, the HTTP GET result for asset Order shows the updated status of the order "Vendor Requests Shipper to Ship Order" after this smart contract is invoked successfully.

The Shipper processes the shipping request and invoke the InDelivering smart contract. Thus, the event OnRouteNotification is emitted to notify the Buyer that the shipment package has left the Shipper's facility and is on its way as shown in Figure 6.7 (1), (2), (3), and (4).

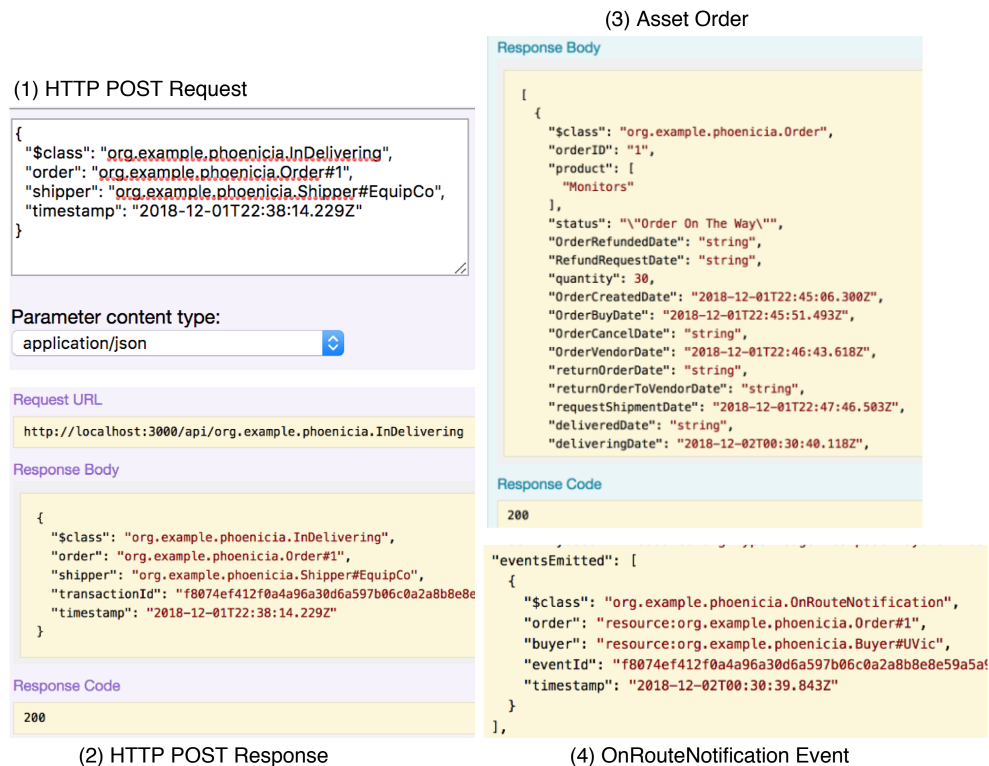

Figure 6.7: HTTP POST: InDelivering

The full history of the order can be inspected by the Order's issuer (The Buyer) at any point in the supply chain using OrderHistorian query as explained in Section 5.5. At this point in the supply chain, if the Buyer runs an HTTP GET for OrderHistorian query, a JSON data response is retrieved and contains the history of this order at this point (i.e. InDelivering is the most recent invocation) as shown in Figure 6.8. The following list of transactions is retuned in JSON query response (i.e. the history of the Order that is obtained after invoking all smart contracts, is around 13 pages):

- **CreateOrder**: Buyer, timestamp, and transaction ID
- **Buy**: Buyer, Seller, timestamp, BuyNotification event data, and transaction ID
- **OrderFromVendor**: Buyer, Seller, Vendor, VendorNotification event data, and transaction ID.
- **RequestShipping**: Buyer, Seller, Vendor, Shipper, timestamp and transactionID.

- **InDelivering**: Buyer, Seller, Vendor, Shipper, OnRouteNotification event data, and transactionID

Request URL

```
http://localhost:3000/api/system/historian
```

Response Body


```
  {
    "$class": "org.hyperledger.composer.system.HistorianRecord",
    "transactionId": "59f3cbdd21dae91d6ccfa168edb384e744ca3a1bc594
    "transactionType": "org.example.phoenicia.OrderFromVendor",
    "transactionInvoked": "resource:org.example.phoenicia.OrderFrc
    "participantInvoking": "resource:org.hyperledger.composer.syst
    "identityUsed": "resource:org.hyperledger.composer.system.Ider
    "eventsEmitted": [
      {
        "$class": "org.example.phoenicia.VendorNotification",
        "order": "resource:org.example.phoenicia.Order#1",
        "vendor": "resource:org.example.phoenicia.Vendor#VenCo",
        "eventId": "59f3cbdd21dae91d6ccfa168edb384e744ca3a1bc5949a
        "timestamp": "2018-12-01T22:46:43.393Z"
      }
    ],
    "transactionTimestamp": "2018-12-01T22:46:43.393Z"
  },
```


Figure 6.8: HTTP GET Resposne: History of the Order

As a result, the Buyer has a full visibility capability on the Order at any point in the supply chain. Moreover, this Blockchan's application delivers the supply chain transparency feature by enabling the Buyer to access these data at any chosen time in the supply chain. The transparency and visibility features were explained in Chapter 1 Section 1.2.

The Buyer can ensure that these previous transactions (e.g., JSON query response) have been truthfully invoked in the Fabric ledger by checking both transactionID and timestamp properties. Therefore, the immutability and validity features are enabled by this Blockchain's application since the transactionID(hash of the transaction) could be only generated if the transaction is truthfully invoked in the Fabric ledger. Moreover, since the Buyer has successfully checked the validity of these transactions, the Buyer can trust that the data, that is included in JSON query response, is valid and accurate. Therefore, this Blockchain's application enables a trust among unknown participants (i.e., the Buyer does not trust the other participants and vice versa).
Each transactionID links to the previous transactionID as explained in Chapter 1 Section 2.2 and Section 2.3. Therefore, this creates an immutable and traceable chain

of transactions. Once the shipment package is received by the Buyer, the Shipper invokes the Deliver smart contract by sending HTTP POST request to the Request URL. The successful body response is returned and contains the data, transactrionID, and timestamp. The event DeliveredNotification is emitted in order to notify the Buyer that the shipment has arrived. Figure 6.9 shows the HTTP POST Request, the HTTP POST Response, the HTTP GET Order, and the DeliveredNortification event.

(1) HTTP POST Request


```
{
  "$class": "org.example.phoenicia.Deliver",
  "order": "org.example.phoenicia.Order#1",
  "shipper": "org.example.phoenicia.Shipper#ShipCo",
  "buyer": "org.example.phoenicia.Buyer#UVic",
  "timestamp": "2018-12-01T22:38:14.207Z"
}
```


Parameter content type:
application/json

Request URL

http://localhost:3000/api/org.example.phoenicia.Deliver

Response Body


```
{
  "$class": "org.example.phoenicia.Deliver",
  "order": "org.example.phoenicia.Order#1",
  "shipper": "org.example.phoenicia.Shipper#ShipCo",
  "buyer": "org.example.phoenicia.Buyer#UVic",
  "transactionId": "851999c57b3aaf375ca524771d0001964ca
  "timestamp": "2018-12-01T22:38:14.207Z"
}
```


Response Code

200

(2) HTTP POST Response

(3) Asset Order

Response Body


```
[
  {
    "$class": "org.example.phoenicia.Order",
    "orderID": "1",
    "product": [
      "Monitors"
    ],
    "status": "\"Deliver Order\"",
    "OrderRefundedDate": "string",
    "RefundRequestDate": "string",
    "quantity": 30,
    "OrderCreatedDate": "2018-12-01T22:45:06.300Z",
    "OrderBuyDate": "2018-12-01T22:45:51.493Z",
    "OrderCancelDate": "string",
    "OrderVendorDate": "2018-12-01T22:46:43.618Z",
    "returnOrderDate": "string",
    "returnOrderToVendorDate": "string",
    "requestShipmentDate": "2018-12-01T22:47:46.503Z",
    "deliveredDate": "2018-12-02T00:49:08.798Z",
    "deliveringDate": "2018-12-02T00:30:40.118Z",
```


Response Code

200


```
"eventsEmitted": [
  {
    "$class": "org.example.phoenicia.DeliveredNotification",
    "order": "resource:org.example.phoenicia.Order#1",
    "buyer": "resource:org.example.phoenicia.Buyer#UVic",
    "eventId": "851999c57b3aaf375ca524771d0001964cabf19b4d54ba
    "timestamp": "2018-12-02T00:49:08.464Z"
  }
],
"transactionTimestamp": "2018-12-02T00:49:08.464Z"
```


(3) DeliveredNotificaiton Event

Figure 6.9: HTTP POST: Deliver

The Seller invokes the PaymentReq smart contract in order to request payment from the participant Bank for the Order that has been delivered to the Buyer. The successful body response of the HTTP POST request is returned and contains the data, transactrionID, and timestamp as shown in Figure 6.10.

(1) HTTP POST Request

{
"$class": "org.example.phoenicia.PaymentReq",
"totalBalanceDue": 0,
"order": "org.example.phoenicia.Order#1",
"bank": "org.example.phoenicia.Bank#BankCo",
"timestamp": "2018-12-01T22:38:14.316Z"
}

Parameter content type:
application/json

Request URL

http://localhost:3000/api/org.example.phoenicia.PaymentReq

Response Body

{
"$class": "org.example.phoenicia.PaymentReq",
"totalBalanceDue": 0,
"order": "org.example.phoenicia.Order#1",
"bank": "org.example.phoenicia.Bank#BankCo",
"transactionId": "83a92fbdd4cc38775d2028ca726147de8d9db
"timestamp": "2018-12-01T22:38:14.316Z"
}

Response Code

200

(2) HTTP POST Response

Response Body

[
{
"$class": "org.example.phoenicia.Order",
"orderID": "1",
"product": [
"Monitors"
],
"status": "\"Seller Requests Bank to Pay\"",
"OrderRefundedDate": "string",
"RefundRequestDate": "string",
"quantity": 30,
"OrderCreatedDate": "2018-12-01T22:45:06.300Z",
"OrderBuyDate": "2018-12-01T22:45:51.493Z",
"OrderCancelDate": "string",
"OrderVendorDate": "2018-12-01T22:46:43.618Z",
"returnOrderDate": "string",
"returnOrderToVendorDate": "string",
"requestShipmentDate": "2018-12-01T22:47:46.503Z",
"deliveredDate": "2018-12-02T00:49:08.798Z",
"deliveringDate": "2018-12-02T00:30:40.118Z",

Response Code

200

(3) Asset Order

Figure 6.10: HTTP POST: Payment Request

Once the Bank receives the payment request, the Bank invokes the Pay smart contract by sending HTTP POST request to the Request URL. The successful body response is returned and contains the data, transactrionID, and timestamp. The event PaymentNotification is emitted in order to notify the Seller that the payment has been sent. Figure 6.11 shows the HTTP POST Request, the HTTP POST Response, the HTTP GET Order, and the PaymentNortification event.

In case the Buyer decided to return the Order to the Seller, The Buyer invokes the ReturnOrder smart contract as HTTP POST request. The successful body response is returned and contains the data, transactrionID, and timestamp. The event ReturnNotificationis emitted in order to notify the Seller that the Order has been returned. Figure 6.12 shows the HTTP POST Request, the HTTP POST Response, the HTTP GET Order, and the RetrunNotification event.

(1) HTTP POST Request


```
{
  "$class": "org.example.phoenicia.Pay",
  "order": "org.example.phoenicia.Order#1",
  "seller": "org.example.phoenicia.Seller#EquipCo",
  "bank": "org.example.phoenicia.Bank#BankCo",
  "timestamp": "2018-12-01T22:38:14.303Z"
}
```


Parameter content type:
application/json

(2) HTTP POST Response

Request URL

```
http://localhost:3000/api/org.example.phoenicia.Pay
```

Response Body


```
{
  "$class": "org.example.phoenicia.Pay",
  "order": "org.example.phoenicia.Order#1",
  "seller": "org.example.phoenicia.Seller#EquipCo",
  "bank": "org.example.phoenicia.Bank#BankCo",
  "transactionId": "2855775b55102f093ff45ceb8a10ad5dafe9fec
  "timestamp": "2018-12-01T22:38:14.303Z"
}
```


Response Code

```
200
```

(3) Asset Order

Response Body


```
[
  {
    "$class": "org.example.phoenicia.Order",
    "orderID": "1",
    "product": [
      "Monitors"
    ],
    "status": "\"Payment Processed\"",
    "OrderRefundedDate": "string",
    "RefundRequestDate": "string",
    "quantity": 30,
    "OrderCreatedDate": "2018-12-01T22:45:06.300Z",
    "OrderBuyDate": "2018-12-01T22:45:51.493Z",
    "OrderCancelDate": "string",
    "OrderVendorDate": "2018-12-01T22:46:43.618Z",
    "returnOrderDate": "string",
    "returnOrderToVendorDate": "string",
    "requestShipmentDate": "2018-12-01T22:47:46.503Z",
    "deliveredDate": "2018-12-02T00:49:08.798Z",
    "deliveringDate": "2018-12-02T00:30:40.118Z",
```


Response Code

```
200
```

(4) PayementNotification Event


```
"eventsEmitted": [
  {
    "$class": "org.example.phoenicia.PaymentNotification",
    "order": "resource:org.example.phoenicia.Order#1",
    "seller": "resource:org.example.phoenicia.Seller#EquipCo",
    "eventId": "2855775b55102f093ff45ceb8a10ad5dafe9fed2c21037c
    "timestamp": "2018-12-02T01:21:27.680Z"
  }
],
"transactionTimestamp": "2018-12-02T01:21:27.680Z"
,
```


Figure 6.11: HTTP POST: Pay

(1) HTTP POST Request


```
{
  "$class": "org.example.phoenicia.ReturnOrder",
  "reasonForReturning": "The Monitors are damages",
  "order": "org.example.phoenicia.Order#1",
  "seller": "org.example.phoenicia.Seller#EquipCo",
  "buyer": "org.example.phoenicia.Buyer#UVic",
  "timestamp": "2018-12-02T01:46:55.863Z"
}
```


Parameter content type:
application/json

(2) HTTP POST Response

Request URL

```
http://localhost:3000/api/org.example.phoenicia.ReturnOrder
```

Response Body


```
{
  "$class": "org.example.phoenicia.ReturnOrder",
  "reasonForReturning": "The Monitors are damages",
  "order": "org.example.phoenicia.Order#1",
  "seller": "org.example.phoenicia.Seller#EquipCo",
  "buyer": "org.example.phoenicia.Buyer#UVic",
  "transactionId": "43f7227a95d4ffd845fb95f4a23b703fc3b3398d
  "timestamp": "2018-12-02T01:46:55.863Z"
}
```


Response Code

```
200
```

(3) Asset Order

Response Body


```
  {
    "$class": "org.example.phoenicia.Order",
    "orderID": "1",
    "product": [
      "Monitors"
    ],
    "status": "\"Return Order to Seller\"",
    "OrderRefundedDate": "string",
    "RefundRequestDate": "string",
    "quantity": 30,
    "OrderCreatedDate": "2018-12-01T22:45:06.300Z",
    "OrderBuyDate": "2018-12-01T22:45:51.493Z",
    "OrderCancelDate": "string",
    "OrderVendorDate": "2018-12-01T22:46:43.618Z",
    "returnOrderDate": "2018-12-02T01:49:33.311Z",
    "returnOrderToVendorDate": "string",
    "requestShipmentDate": "2018-12-01T22:47:46.503Z",
    "deliveredDate": "2018-12-02T00:49:08.798Z",
    "deliveringDate": "2018-12-02T00:30:40.118Z",
    "paymentRequestDate": "2018-12-02T01:11:38.967Z",
```


Response Code

```
200
```

(4) ReturnNotification Event


```
"eventsEmitted": [
  {
    "$class": "org.example.phoenicia.ReturnNotification",
    "order": "resource:org.example.phoenicia.Order#1",
    "seller": "resource:org.example.phoenicia.Seller#EquipCo",
    "eventId": "43f7227a95d4ffd845fb95f4a23b703fc3b3398dab397e3
    "timestamp": "2018-12-02T01:49:33.096Z"
  }
],
"transactionTimestamp": "2018-12-02T01:49:33.096Z"
```


Figure 6.12: HTTP POST: ReturnOrder

(3) Asset Order

(1) HTTP POST Request

{
"$class":
"org.example.phoenicia.ReturnOrderVendor",
"order": "org.example.phoenicia.Order#1",
"seller": "org.example.phoenicia.Seller#EquipCo",
"vendor": "org.example.phoenicia.Vendor#VenCo",
"timestamp": "2018-12-02T01:46:55.884Z"
}

Parameter content type:
application/json

Request URL

http://localhost:3000/api/org.example.phoenicia.ReturnOrderVendor

Response Body

{
"$class": "org.example.phoenicia.ReturnOrderVendor",
"order": "org.example.phoenicia.Order#1",
"seller": "org.example.phoenicia.Seller#EquipCo",
"vendor": "org.example.phoenicia.Vendor#VenCo",
"transactionId": "c197c36745b628903012453270f394775d185316622e
"timestamp": "2018-12-02T01:46:55.884Z"
}

Response Code

200

(2) HTTP POST Response

Response Body

[
{
"$class": "org.example.phoenicia.Order",
"orderID": "1",
"product": [
"Monitors"
],
"status": "\"Returned Order to Vendor\"",
"OrderRefundedDate": "string",
"RefundRequestDate": "string",
"quantity": 30,
"OrderCreatedDate": "2018-12-01T22:45:06.300Z",
"OrderBuyDate": "2018-12-01T22:45:51.493Z",
"OrderCancelDate": "string",
"OrderVendorDate": "2018-12-01T22:46:43.618Z",
"returnOrderDate": "2018-12-02T01:49:33.311Z",
"returnOrderToVendorDate": "2018-12-02T01:57:12.269Z",
"requestShipmentDate": "2018-12-01T22:47:46.503Z",
"deliveredDate": "2018-12-02T00:49:08.798Z",
"deliveringDate": "2018-12-02T00:30:40.118Z",

Response Code

200

Figure 6.13: HTTP POST: ReturnOrderVendor

Once the Seller receives the returned order, the Seller invokes the ReturnOrderVendor smart in order to return the Order to the Vendor. The successful body response of the HTTP POST request is returned and contains the data, transactrionID, and timestamp as shown in Figure 6.13.

However, the Bank invokes the smart contract RefundRequest in order to request the refund from the Seller for the returned Order. Figure 6.14 shows the HTTP POST Request, the HTTP POST Response, and the HTTP GET Order. Once the

Seller receives the request for refund from the the Bank, the Seller invokes the smart contract Refund. Figure 6.15 shows the HTTP POST Request, the HTTP POST Response, and the HTTP GET Order, and the RefundNotification event. This event is to notify the Bank that the request for refund has been approved.

Figure 6.14: HTTP POST: RefundRequest

Figure 6.15: HTTP POST: Refund

The CancelOrder smart contract is invoked if the status of the Order is either "OrderCreateStatus" or "OrderBuyStatus" as expalined in Chapter 5 Section 5.4.13. However, since the most recent status of the Order is "Refunded", the application should be down and up again with empty Blockchain ledger.

HTTP POST requests and their responses were demonstrated in all previous Figures with a response code:200 for each one (i.e. refers to a successful POST request). Therefore, if HTTP GET request for each smart contract endpoint is sent (i.e. after a successful HTTP POST), HTTP GET response is returned and matches exactly the HTTP POST one that were demonstrated in all previous Figures. In a similar way, if query OrderbyBuyer is called and sent with parameter (Buyer = "UVic"), the response returns the Order that was bought by the Buyer as illustrated in Figure 6.4. Since this Order has a quantity field of 30 as illustrated in Figure 6.3, if query OrderByQuantity is called and send with parameter(quantity > 20) as coded in C, the response returns this Order. The similar response applied on query OrderByProduct if it is sent with parameter (product ="Monitors").

6.3 Summary

This chapter presents the fully operational and functional capabilities. An analysis of the POST request and POST response as well as the GET request and the GET response was presented which is exhibited in the figures. The following application's capabilities were successfully delivered.

Visiblity: participant Buyer = "UVic" and any other application's participants can query the Order at any timestamp to track the history of the Order throughout the supply chain. As shown in the HTTP POST figures, this participant can know exactly when the Order has been received by another participant and what changes have been made on it. Thus, the provenance feature is achieved and maintained. This promotes trust among the application's participants and enables the participants to hold a proof of the smart contract invocation (i.e., participants Sellers cannot deny that the Buy request from Buyers).

Transperency: participant Buyer and any other application's participants have access to the collected data stored in the ledger. Furthermore, these collected data are identical to the result of smart contracts that has been invoked in the ledger. Once a smart contract is invoked, the resulting data is stored in the ledger.

Decentralization: the core value of this Blockchain application allows data to be shared across boundaries without having a central trusted entity. This feature is important when independent and untrusted participants collaborate because none of them have a central authority. Decentralization eliminates the problems that arise from having a central entity such as data manipulation and surveillance, which guarantees transparency. Therefore, this application eliminates the need for third-party auditor to validate the correctness of each smart contract invocation since Blockchain transactions contain their own proof of authorization and validity.
Data consistency and Integrity: Data consistency and integrity are achieved in this application when the hash technique is applied and a transaction is added into the Blockchain Fabric's transactions log. Furthermore, each transaction links to its previous transaction and a timestamp is emitted for each transaction to determine the time in which a transaction has occurred.
Access Control: all participants in the network can only perform Create, Read and Update operations on their own information instance. Participant Buyer = "UVic" cannot update the Seller information. Furthermore, participant Buyer can only invoke CreateOrder, Buy, ReturnOrder and CancelOrder smart contracts on the Order that this Buyer creates. The same rule applies to all other participants in their designated asset *Order*. The following example is designed to demonstrate this capability.

(1) Add Participant Instance: BuyerCO

{
"$class": "org.example.phoenicia.Buyer",
"buyerID": "BuyerCo",
"ContactInformation": "buyer@me.com"
}

Parameter content type:
application/json

Request URL
http://localhost:3000/api/org.example.phoenicia.CreateOrder
Response Body

{
"$class": "org.example.phoenicia.CreateOrder",
"quantity": 25,
"product": [
"Laptops"
],
"order": "org.example.phoenicia.Order#2",
"buyer": "org.example.phoenicia.Buyer#BuyerCo",
"transactionId": "45f2b2f6f815a5765caf60dcb730067b9d40a0
"timestamp": "2018-12-02T02:24:30.998Z"
}

Response Code
200

(2) BuyerCo Created Order with "OrderID" = "2"

Request URL
http://localhost:3000/api/org.example.phoenicia.Buy
Response Body

{
"error": {
"statusCode": 500,
"name": "Error",
"message": "Error trying invoke business network. Er
"stack": "Error: Error trying invoke business networ
}
}

Response Code
500

(3)The Invocation Error of Buy Smart Contract

Figure 6.16: Access Control for Smart Contract Invocation

Figure 6.16 (1) illustrates creating another participant Buyer instance Buyer =”BuyerCo“. In (2), participant BuyerCo creates the asset Order with ”orderID“ = ”2“. However, when participant BuyerCo invokes a Buy smart contract ,as illustrated in (3), on asset Order with ”orderID“ = ”1“ (that belongs to Buyer = ”UVic“), the error is generated and the invocation request is rejected. This is because the participant BuyerCo is not allowed to perform the invocation on the *Order* that belongs to another participant (UVic).

Chapter 7

Performance Evaluation

This chapter reports on the experiments and evaluation of the subject application. The experiments are performed locally using the Hyperledger Caliper tool and the configuration files for this experiment provided in Appendix D.

Hyperledger Caliper (HL Caliper) is a Blockchain performance measurement tool aimed to evaluate the Blockchain solutions according to a number of performance indicators. These indicators are send TPS rate (Transactions Per Second), transaction latency, transaction throughput, and resource utilization. Hyperledger Caliper is an open source initially developed by developers from Huawei, Oracle, IBM and other companies.
To demonstrate the performance evaluation, the experimental environment was set up as follows:

- The local HL Caliper platform is configured to install the prerequisite development dependencies (e.g., HL Fabric v.1.1 dependencies).
- The shipping application is integrated with HL Caliper by setting up JSON configuration files. These files have the required information for HL Caliper to connect with the shipping application's architecture as explained in Section 5.2 as well as the desired experimental scenarios' methods.
- HL Caliper is configured to connect with the HL Fabric's architecture scheme using the configuration files.
- HL Caliper is run, and the results are observed in the Mac's terminal as well as in the form of auto-generated reports in HTML format.

Table 7.1: Experimental Hardware and Software Setup

Experimental Setup	
Performance Environment	**Specifications**
MacBook Pro Machine	**OS Version:** MacOS High Sierra **Processor:** Intel Core i5 2.8GHZ **Memory:** 8GB 1600MHZ DDR3 **Mac Version:** Retina, Mid 2014
Docker Container	**Docker Version:**: 18.06.1-ce **Composer Version:** 1.22.0

Table 7.1 describes the experimental hardware and software setup used in the experiments.

Transaction throughput, measured in TPS, is the maximum rate in which valid transactions are committed in the HL Fabric ledger:

$$Transaction_Throughput = Total_committed_transactions/total_time_in_seconds$$

Transaction latency is the time it takes for a transaction's effect to be usable across the network. The measurement includes the time from the point that the transaction is submitted to the point that it is confirmed and committed in the HL Fabric ledger. This includes the time that the REST client takes to submit the transaction to the network and any assigned time due to the HL Fabric's consensus mechanism (e.g., endorsement, validation and committing time as expalined in Section 2.7.1):

$$Transaction_Latency = Committed_Transaction_Time_Submit_time$$

Two experimental scenarios were designed to measure the performance indicators at the orderer node and validator peer of the Hyperledger Fabric, that were explained in Chapter 2 Section 2.7.1 . The two primary metrics investigated are: the effect of the network load and the send TPS rate on the application performance. The network load defines the number of the total transactions that is submitted to the application during the test round. Furthermore, it defines the number of the total assets that is

created in the application and consumed by the smart contract during its execution. The transaction send TPS rate defines the number of transactions per second that is submitted to the application. These metrics are important for real world applications because the network load and the send TPS rate can significantly impact the user experience and the application performance.
Two controllers are applied for both experimental scenarios (i.e., network load and send TPS rate):

Method A Fixed Rate Controller:
This controller submits transactions at a fixed interval that is specified as TPS. The following example is the fixed rate controller configuration at send TPS rate of 10 that is specified within a JSON configuration file:

$rateControl:$
$\{$
$type: fixed_rate,$
$opts: \{tps: 10\}$
$\}$

Method B Fixed Feedback Controller:
This controller submits transactions at a fixed send TPS rate. However, it implements both properties of **unfinished per client** as a fixed feedback and **Sleep Time**. These properties are used to give a solution for queuing unfinished transactions that are waiting to be committed in the HL Fabric's ledger. Moreover, this situation is observed as the difference between the number of the submitted transactions and the number of unfinished transactions that are shown in the terminal at each step of the running experiment.
The process of submitting new transactions is paused until the ledger commits the unfinished transactions. This impacts on the observing transaction throughput and latency during the network load experiment.
The following JSON example is the fixed feedback rate controller configuration at 40 TPS, 13 for unfinished per client, and 1 second for sleep time:
$rateControl:$
$[\{$
$type: fixed_feedback_rate,$
$opts: \{unfinished_per_client: 13, tps: 40, sleep_time: 1000\}$

}]

7.1 Experiment I: Network Load

In this experiment, the send TPS rate is set at 10 TPS and two different network load parameters' data set are applied in order to investigate the network load effect on the performance of the network. In Table 7.2, the two Data Set I and II are supplied to Methods A and B:

Table 7.2: Experiment I: Data Sets

Experiment I: Network Load Parameters Data Set	
Data Set I	
Parameter	**Value**
Total Transactions	10
Send TPS Rate	10
Assets	10
Data Set II	
Parameter	**Value**
Total Transactions	20
Send TPS Rate	10
Assets	20

The average latency, throughput and resource utilization are observed for Methods A and B.

7.1.1 Experiment I: Method A Fixed Rate Controller

In this experiment, three rounds of performance measurement are conducted for Data Sets I and II. Tables 7.3 and 7.4 show the results for average latency, transaction throughput, successful transactions, fail transactions, and resource utilization that observed for this method.

Table 7.3: Experiement I: Method A Fixed Rate Controller

Experiment I Method A Fixed Rate Controller					
Using Data Set I					
Rounds	**Send TPS**	**Succ. TX**	**Fail Tx**	**Avg. Latency**	**Throughput**
1	10	10	0	2.32s	3
2	10	10	0	2.40s	3
3	10	10	0	1.86s	4
Using Data Set II					
Rounds	**Send TPS**	**Succ. TX**	**Fail Tx**	**Avg. Latency**	**Throughput**
1	10	20	0	5.03	3
2	10	20	0	4.94	3
3	10	20	0	4.19	4

Table 7.4: Resource Utilization for Experiment I Method A

Resource Utilization For Data Set I Method A				
Process	**Avg. RAM**	**Avg. CPU**	**Traffic In**	**Traffic Out**
Peer1	34.2MB	17.01%	1.9MB	2.3MB
Peer2	32.5MB	11.54%	1.1MB	1.6MB
Orderer	10.8MB	1.75%	202.9KB	311.4KB
CouchDB	100.1MB	26.81%	395.4KB	0.864KB
HL Dev Peer1	105.1MBMB	11.44%	666.2KB	593.5KB
HL Dev Peer2	103.7MB	7.15%	311.6KB	282.6KB
Resource Utilization For Data Set II Method A				
Process	**Avg. RAM**	**Avg. CPU**	**Traffic In**	**Traffic Out**
Peer1	35.8MB	21.32%	3.1MB	3.3MB
Peer2	34.6MB	11.67%	1.7MB	2.0MB
Orderer	12.1MB	1.80%	319.1KB	611.9KB
CouchDB	100.35MB	29.97%	628.8KB	1.3MB
HL Dev Peer1	93.1MB	7.53%	531.8KB	473.4KB
HL Dev Peer2	106.4MB	14.92%	1.1MB	964.7KB

7.1.2 Experiment I: Method B Fixed Feedback Rate Controller

In this experiment, three rounds of performance measurement are conducted for Data Sets I and II. Tables 7.5 and 7.6 show the results for average latency, transaction throughput, successful transactions, fail transactions, and resource utilization observed for this method.

Table 7.5: Experiement I: Method B Fixed Feedback Controller

Experiment I: Method B Fixed Feedback Controller					
Using Data Set I					
Rounds	**Send TPS**	**Succ. TX**	**Fail Tx**	**Avg. Latency**	**Throughput**
1	10	10	0	2.41s	4
2	10	10	0	2.74s	3
3	10	10	0	2.08s	4
Using Data Set II					
Rounds	**Send TPS**	**Succ. TX**	**Fail Tx**	**Avg. Latency**	**Throughput**
1	10	20	0	3.98	4
2	10	20	0	4.34	4
3	10	20	0	3.31	5

Table 7.6: Resource Utilization for Experiment I Method B

Resource Utilization For Data Set I Method B				
Process	**Avg. RAM**	**Avg. CPU**	**Traffic In**	**Traffic Out**
Peer1	35.4MB	20.15%	1.5MB	1.8MB
Peer2	33.4MB	10.85%	971.1KB	1.2MB
Orderer	11MB	1.65%	114.9KB	238.5KB
CouchDB	99.4MB	28.9%	350.15KB	764.3KB
HL Dev Peer1	105.2MB	13.49%	574.1MB	398.9KB
HL Dev Peer2	104.2MB	6.88%	292.8KB	242.4KB

Resource Utilization For Data Set II Method B				
Process	**Avg. RAM**	**Avg. CPU**	**Traffic In**	**Traffic Out**
Peer1	36.1MB	22.32%	3.1MB	3.9MB
Peer2	35.1MB	12.42%	1.9MB	2.8MB
Orderer	11.8MB	1.85%	355KB	718KB
CouchDB	101.05MB	32.14%	691.45KB	1.4MB
HL Dev Peer1	106.9MB	14.61%	1.1MB	812.5KB
HL Dev Peer2	104.9MB	7.78%	520.4KB	478.9KB

7.2 Experiment II: send TPS rate

In this experiment, two different network's send TPS rate are applied while network load parameters (i.e. total transactions and assets) are fixed. In Table 7.7, the two Data Sets I and II are supplied to Methods A and B.

Table 7.7: Experiment II: Data Sets

Experiment II: Send TPS rate Data Set	
Data Set I	
Parameter	**Value**
Total Transactions	60
Send TPS Rate	20
Assets	60
Data Set II	
Parameter	**Value**
Total Transactions	60
Send TPS Rate	40
Assets	60

7.2.1 Experiment II: Method A Fixed Rate Controller

In this experiment, three rounds of performance measurement were conducted for Data Sets I and II. Table 7.8 and 7.9 show the results for average latency, transaction throughput, successful transactions, fail transactions, and resource utilization that observed for this method.

Table 7.8: Experiement II: Method A Fixed Rate Controller

Experiment II: Method A Fixed Rate Controller					
Using Data Set I					
Rounds	**Send TPS**	**Succ. TX**	**Fail Tx**	**Avg. Latency**	**Throughput**
1	20	60	0	10.93s	5
2	20	60	0	13.65s	4
3	20	60	0	10.30s	5
Using Data Set II					
Rounds	**Send TPS**	**Succ. TX**	**Fail Tx**	**Avg. Latency**	**Throughput**
1	40	60	0	11.37s	5
2	40	60	0	13.90s	4
3	40	60	0	11.31s	5

Table 7.9: Resource Utilization for Experiment II Method A

Resource Utilization For Data Set I Method A				
Process	**Avg. RAM**	**Avg. CPU**	**Traffic In**	**Traffic Out**
Peer1	43.4MB	14.01%	5.1MB	8.8MB
Peer2	44.5MB	25.43%	8.6MB	11.9MB
Orderer	14.4MB	1.50%	932.9KB	1.8MB
CouchDB	109.65MB	36.91%	1.85MB	3.85MB
HL Dev Peer1	111.3MB	15.57%	3.0MB	2.2MB
HL Dev Peer2	88.6MB	8.32%	1.4MB	1.3MB
Resource Utilization For Data Set II Method A				
Process	**Avg. RAM**	**Avg. CPU**	**Traffic In**	**Traffic Out**
Peer1	45.4MB	25.32%	8.3MB	8.9MB
Peer2	56.0MB	12.96%	4.9MB	6.6MB
Orderer	15.4MB	1.69%	928.2KB	1.8MB
CouchDB	109.7MB	34.93%	1.7MB	3.6MB
HL Dev Peer1	110.9MB	16.30%	3.0MB	2.2MB
HL Dev Peer2	91.0MB	8.31%	1.4MB	1.3MB

7.2.2 Experiment II: Method B Fixed feedback controller

In this experiment, three rounds of performance measurement are conducted for Data Sets I and II. Table 7.10 and 7.11 show the results for average latency, transaction throughput, successful transactions, fail transactions, and resource utilization that observed for this method.

Table 7.10: Experiment II: Method B Fixed Feedback Controller

Experiment II: Method B Fixed Feedback Controller					
Using Data Set I					
Rounds	**Send TPS**	**Succ. TX**	**Fail Tx**	**Avg. Latency**	**Throughput**
1	20	60	0	9.88s	5
2	20	60	0	12.07s	4
3	20	60	0	9.38s	5
Using Data Set II					
Rounds	**Send TPS**	**Succ. TX**	**Fail Tx**	**Avg. Latency**	**Throughput**
1	40	60	0	10.38s	5
2	40	60	0	12.13s	4
3	40	60	0	10.88s	5

Table 7.11: Resource Utilization for Experiment II Method B

Resource Utilization For Data Set I Method B				
Process	**Avg. RAM**	**Avg. CPU**	**Traffic In**	**Traffic Out**
Peer1	43.5MB	24.09%	8.3MB	8.7MB
Peer2	50.9MB	13.17%	5.0MB	7.1MB
Orderer	15.2MB	1.88%	932.3KB	1.8MB
CouchDB	108.7MB	33.9%	1.7MB	3.6MB
HL Dev Peer1	111.0MB	15.98%	3.0MB	2.2MB
HL Dev Peer2	88.6MB	8.53%	1.4MB	1.3MB
Resource Utilization For Data Set II Method B				
Process	**Avg. RAM**	**Avg. CPU**	**Traffic In**	**Traffic Out**
Peer1	52.0MB	13.22%	5.0MB	8.2MB
Peer2	44.6MB	24.44%	8.5MB	11.2MB
Orderer	17.2MB	1.90%	923.1KB	1.8MB
CouchDB	107.7MB	34.31%	1.85MB	3.75MB
HL Dev Peer1	96.1MB	8.30%	1.4MB	1.3MB
HL Dev Peer2	109.9MB	15.86%	3.0MB	2.2MB

7.3 Results Analysis

7.3.1 Analysis of Experiment I. Method A

- **The higher network load at 20 transactions, 20 assets, and fixed send TPS rate at 10, results in a significant increase in the average latency (by nearly a factor of 2) than the lower network load at 10 transactions, 10 assets, and fixed send TPS rate at 10**.

  This is because the number of transactions that are waiting at the orderer node is growing rapidly in network load of 20 transactions than the network load of 10 transactions. Therefore, the waiting time at the orderer node is increased and hence the average latency.

  As explained in Chapter 2 Section 2.7.1, the orderer node receives the transactions from the endorser peer to add them to a newly created block and then

deliver it to the validator peer.

- **Peers 1 and 2 (i.e., endorser and validator peers) consume larger CPU resources in the higher network load than in the lower network load**. The validator peer utilizes computational power of CPU to check the signature of the block of transactions and validates each transaction inside the block with the endorsement policy before it commits these transactions into the ledger. Therefore, since more transactions are required to be validated in the higher network load than the lower network load, the CPU consumption is increased. The role of the orderer node is to add the transactions to a block and deliver it to the validator peer. Therefore, the orderer node's resource consumption is changed slightly.

7.3.2 Analysis of Experiment I. Method B

- **The higher network load at 20 transactions, 20 assets, and fixed send TPS rate at 10, results in an increase in the average latency than the lower network load at 10 transactions, 10 assets, and fixed send TPS rate at 10. However, the Fixed Feedback Controller achieves a better latency performance (i.e., less latency) than the Fixed Rate controller in higher network load**.

  This is because the Fixed FeedBack Controller prevents the queue at the orderer node to go deeper by pausing and not submitting new transactions until the number of queuing transactions at the orderer node is lower than the Fixed Feedback Controller's **unfinished per client** threshold which is 13 transactions. Therefore, shorter queue results in a better latency performance. The value of **unfinished per client** is set after running several experiments.

- **The CPU utilization for Peers 1 and 2 in the higher (i.e., Data Set II) than lower network load (i.e.,Data Set I) for Method A, is less than The CPU consumption for Peers 1 and 2 in the higher than lower network load for Method B**.

  The difference in CPU utilization between Peers 1 and 2 in higher and lower network load in Method A is 4.31%, 0.13%, respectively, and in Method B is 2.17%, 1.57%, respectively. Hence, peers in Method B consumes less CPU resource than in Method A.

The Fixed Feedback Controller pauses submitting new transactions. Therefore, the number of transactions that is required to be validated is less than the case when applying the Fixed Controller. Thus, the CPU consumption by the validator when applying the Fixed Feedback Controller is less than the case when applying the Fixed Controller. Orderer CPU consumption is changed slightly in higher network load than lower network load.

7.3.3 Summary of Experiment I

- **Fixed Feedback Controller delivers a better latency performance (i.e., less latency) than Fixed Rate Controller**.

  The average latency difference between higher and lower network load when **the Fixed Feedback Controller is applied**, is [1.57s, 1.6s, 1.23s] in respect to the experiment's round index [1,2,3].

  The average latency difference between higher and lower network load when **the Fixed Rate Controller is applied**, is [2.81s, 2.54s, 2.33s] in respect to the experiment's round index [1,2,3].

- **In the Experiment I and applying the both methods, the increasing in the achieved throughput results in decreasing in the achieved latency**

  In Method A and the lower network load (i.e., Data Set I), the average latency decreases from 2.32s and 2.40s (i.e., round 1 and round 2, respectively) with 3 TPS throughput to 1.86s (i.e., round 3) with 4 TPS throughput.

  In Method A and the higher network load (i.e., Data Set II), the average latency decreases from 5.03s and 4.94s (i.e., round 1 and round 2, respectively) with 3 TPS throughput to 4.19s (i.e., round 3) with 4 TPS throughput.

  In Method B and the lower network load, the average latency decreases from 2.74s (i.e., round 2) with 3 TPS throughput to 2.41s and 2.08s (i.e., round 1 and round 3, respectively) with 4 TPS throughput.

  In Method B and the higher network load, the average latency decreases from 3.98s, 4.34s (i.e., round 1 and round 2, respectively) with 4 TPS throughput to 3.31s (i.e.,round 3) with 5 TPS throughput.

7.3.4 Analysis of Experiment II. Method A

- **The higher send TPS rate at 40 TPS results in a slight increase in latency than the lower send TPS rate at 20 TPS**

  The network is overloaded at send TPS rate of 20 and 40 TPS. The number of transactions, is waiting at the orderer node, is significantly increased. This is observed as the orderer's node traffic in and traffic out, in the lower send TPS rate, are 932.9KB and 1.8KB, respectively (i.e. traffic out is nearly a factor of 2 of traffic in). The orderer's node traffic in and traffic out, in the higher send TPS rate, are 928.2KB and 1.8KB, respectively (i.e. nearly by a factor of 2).

7.3.5 Analysis of Experiment II. Method B

- **The higher send TPS rate at 40 TPS and fixed network load parameters, increases the average latency than the lower send TPS rate at 20 TPS and fixed network load parameters. However, the Fixed Feedback Controller delivers a better latency performance (i.e., less latency) than Fixed Rate controller in higher send TPS rate**.

  This is because the Fixed Feedback Controller prevents the queue at the orderer node to go deeper by pausing and not submitting new transactions until the number of queuing transactions at the orderer node is lower than the Fixed Feedback Controller's **unfinished per client** threshold which is 13 transactions. Therefore, shorter queue results in a better latency performance. The value of **unfinished per client** is set after running several experiments.

- **The CPU utilization for peers 1 and 2 in the higher (i.e., Data Set II) than lower send TPS rate (i.e., Data Set I) when applying the Fixed Feedback controller, nearly equals to the CPU consumption for Peers 1 and 2 in the higher than lower send TPS rate when applying the Fixed Rate Controller.**

  Given that the network is overloaded at send TPS rate of 20 and 40 TPS, the number of transactions at the orderer node is significantly increased (i.e., more than the Experiment I as its send TPS rate is fixed at 10 TPS). Therefore, pausing and not submitting transactions, in the Fixed Feedback Controller, does not deliver that better performance in CPU consumption (i.e., compares to the Fixed Rate Controller).

7.3.6 Summary of Experiment II

- **The Fixed Feedback Controller delivers a better latency performance (i.e., less latency) than the Fixed Rate Controller**.

  The average latency difference between higher and lower send TPS rate when **the Fixed Feedback Controller is applied**, is [0.5s, 0.06s, 1.5s] in respect to the performance measurement round index [1,2,3].
  The average latency difference between higher and lower send TPS rate when **the Fixed Rate Controller is applied**, is [0.44s, 0.25s, 1.01s] in respect to the performance measurement round index [1,2,3]

- **In the Experiment II and applying the both methods A and B, the increasing in the achieved throughput results in decreasing in the achieved latency**

  In Method A and the lower send TPS rate (i.e., Data Set I), the average latency decreases from 13.64s (i.e., round 2) with 4 TPS throughput to 10.93s and 10.20s (i.e., round 1 and round 2, respectively) with 5 TPS throughput.
  In Method A and the higher send TPS rate(i.e., Data Set II), the average latency decreases from 13.90s (i.e., round 2) with 4 TPS throughput to 11.37s and 11.31s (i.e., round 1 and round 3, respectively) with 5 TPS throughput.
  In Method B and the lower send TPS rate, the average latency decreases from 12.07s (i.e., round 2) with 4 TPS throughput to 9.88s and 9.38s (i.e., round 1 and round 3, respectively) with 5 TPS throughput.
  In Method B and the higher send TPS rate, the average latency decreases from 12.13s (i.e., round3) with 4 TPS throughput to 10.38s and 10.88s (i.e., round 1 and round 2, respectively) with 5 TPS throughput.

7.3.7 Summary of Experiment I and Experiment II

- **In the Experiment I and II that are running on a local experimental setup, the maximum achieved throughput is 5 TPS (Saturation)**

- **The increasing in the achieved throughput results in decreasing in the achieved latency**

- **The average latency is significantly increased in the Experiment 2 compares to the Experiment I**
  This is because the sent TPS rate in the Experiment I is fixed at 10 TPS while in the Experiment II, the network is tested for the send TPS rate of 20 and 40 TPS.

7.3.8 Summary

In the Experiment I and II, the trade off was observed between using the Fixed Rate Controller and the Fixed Feedback Controller. The saturation (maximum) throughput is 5 TPS given that the limitation of the CPU computational resource of the experimental setup.

As observed, the Fixed Feedback Controller achieves a better latency performance (i.e., less latency) than the Fixed Rate Controller. Furthermore, the transaction throughput and the average latency follow a certain pattern. This pattern presents that the increasing in the achieved throughput results in decreasing in the achieved latency. Moreover, a significant increase in the average latency was observed when the send TPS rate is higher than the saturation throughput of the network (i.e., In the Experiment II, the send TPS rate is 20 and 40 TPS). However, the average latency is decreased when the send TPS rate is around the saturation throughput (i.e. Experiment I. send TPS rate is 10 TPS).

Chapter 8

Conclusion and Future Work

8.1 Summary of Contributions

This thesis described the realization of a Blockchain shipping application's user story as a Proof of Concept. The privacy and validity of the data being exchanged was considered. Moreover, untrusted participants were enabled to interact through a private but shared ledger that records the transfer of each order from the point of creating the order to delivery. This application achieved the visibility and transparency features since the participants can track their status of the order along the entire supply chain and can access the order's data at any time. The participants can verify the validity of the data that is being queried with the transaction ID and the time stamp. These two properties are appended into the ledger when a smart contract is invoked. This promotes the finality feature because this appended data cannot be deleted or revoked. Furthermore, an immutable chain of transactions was created since each transaction is linked to its previous transaction through its hash.
This application eliminates the need for a central authority or third-party involvement. For instance, there is no need for an auditor or inspector to validate the correctness of the data since the order carries all the tamper-free data, including when and where the order was at each timestamp along the entire supply chain. However, this data can only be accessed when the accessibility permission has been granted by the application's owner. This capability is demonstrated by implementing the Access Control List in which the participants can only perform Create, Read, and Updated (CRU) operations on their own information instance (e.g., buyerID and contact information). Furthermore, participants can only invoke the smart contract

that satisfies their pre-defined role in the application (e.g., Buyer is not allowed to invoke the shipping request smart contract). Finally, the participants can only invoke the smart contract on their own asset.

A performance measurement was conducted on the application to measure the transaction throughput, transaction average latency, and resource utilization. Experiment I focused on analyzing the effect of the network load with fixed send TPS rate and Experiment II focused on analyzing the effect of the send TPS rate with fixed network load. Furthermore, two methods of Fixed Rate controller and Fixed Feedback controller were applied in both experiments. The application achieved a transaction throughput of 5 TPS in the experimental setup and a pattern was observed in which the Fixed Feedback rate Controller achieves a better average latency performance than the Fixed Rate Controller. The observed transaction throughput increased and the average latency decreased.

8.2 Future Work

Blockchain technology is still in the early to mid-level research and development stage which means there are many untapped topics for researchers to investigate. This work has its limitations because a validator peer in Hyperledger Fabric needs a significant computational power to check the Block signature as well as each transaction for its compatibility with the endorsement policy. A performance measurement was conducted on a local machine with limited CPU resources. Therefore, this limited the valdiator's capability to perform fast validations when either high send TPS rate or high network load was applied. Thus, future work should apply an approach to deploy the application into a Cloud with high computational resource capabilities with more Hyperledger Fabric peers. However, this approach is expensive because of the average cost for deploying the application into the IBM cloud, costing USD1000 per month and an additional USD1000 per month for each peer deployed.

Integrating an Internet of Things (IoT) sensor into the application to collect real time information of the shipment package is an important next step for future work. However, this approach should focus on the security aspect since the IoT sensor is an external part to the Blockchain. If the IoT sensor ends up being controlled by an unauthorized participant, this participant can defraud the collected data before being appended and immutability recorded in the Blockchain ledger.

References

[1] (2009, December) Valuation of the liner shipping industry. IHS Global Insight. [Online]. Available: https://bit.ly/2G1hdaE

[2] P. Kauschke and A. Tipping. (2016) The future of the logistics industry. pwc. [Online]. Available: https://pwc.to/2Eziqjl

[3] (2018, October) Pil partners with ibm on blockchain. Champer of Shipping in British Columbia. [Online]. Available: https://bit.ly/2QfytgY

[4] G. Trousil. (2014) Top challenges for today's logistics providers. Supply Chain 24/7. [Online]. Available: https://bit.ly/2BS2quW

[5] N. Duckworth. (2018) Supply chain visibility and transparency: How everybody wins. TDWI. [Online]. Available: https://bit.ly/2zqi2X4

[6] D. Prokop. (2017) Good question — what's the difference between supply chain visibility and transparency? Inbound Logistics. [Online]. Available: https://bit.ly/2KTNM9n

[7] K. BARRIOS. (2018) What does supply chain visibility look like? Xeneta. [Online]. Available: https://bit.ly/2G17trC

[8] E. Noah. (2016) Transparency, visibility, and context. Percolate. [Online]. Available: https://bit.ly/2PkVrhA

[9] (2018) Connecting global trade. Maersek. [Online]. Available: https://www.maersk.com/

[10] A. P. Moller, "One of trade's biggest barriers," p. 43, May 2017.

[11] J. Davis. (2011, October) The crypto-currency. [Online]. Available: https://bit.ly/2y67mJ9

[12] N. Satoshi. (2008) Bitcoin: A peer-to-peer electronic cash system. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[13] C. S. Jennifer, "Evaluation report to united states senate," *Committee on Homeland Security and Government Affairs*, p. 15, November 2013.

[14] (2017) What is bitcoin double spending? Bitcoin. [Online]. Available: https://www.bitcoin.com/info/what-is-bitcoin-double-spending

[15] B. Vangie. Atomic operation. [Online]. Available: https://bit.ly/2SvV99E

[16] U. Chohan. (2017) The double-spending problem. [Online]. Available: https://bit.ly/2RACWqQ

[17] L. Hannig. (2018) Cryptography. Lisk. [Online]. Available: https://bit.ly/2IHwjyI

[18] M. Bellare and P. Rogaway, "Introduction to modern cryptography," *University of California*, p. 12, 2005.

[19] (2017) Symmetric vs Asymmetric Encryption – What are differences? SSL Information. [Online]. Available: https://bit.ly/2pZouPI

[20] (2017) How does cryptography protect the blockchain? [Online]. Available: https://bit.ly/2QfExWE

[21] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, "Handbook of applied cryptography," *CRC Press*, pp. 26–31–32, 1996.

[22] (2014) How a bitcoin transaction works? CCN. [Online]. Available: https://bit.ly/2UfugIC

[23] S. Delgado-Segura, C. Perez-Sol, J. Herrera-Joancomart, and G. Navarro-Arribas, "Bitcoin private key locked transactions," *Information Processing Letters*, pp. 37–41, 2018.

[24] (2018) What is hashing? Lisk. [Online]. Available: https://bit.ly/2BRVjm0

[25] Z. Yong-Xia and Z. Ge, "MD5 Research," *2010 Second International Conference on Multimedia and Information Technology, IEEE*, 2010.

[26] (2018) Hashing. [Online]. Available: https://bit.ly/2RzCvxq

[27] N. Greeshma and S. Shoney, "Centralised ledger to distributed ledger," *International Research Journal of Engineering and Technology*, p. 15, 2017.

[28] (2018) Graduate admission and records. University of Victoria. [Online]. Available: https://www.uvic.ca/graduatestudies

[29] C. Cashin and M. Vuklic, "Blockchain consensus protocols in the wild," *IBM Research Zurich*, 2017.

[30] H. Chris. (2017) Consensus in blockchain systems. [Online]. Available: https://bit.ly/2HwpUX4

[31] R. Margaret. (2017) consensus algorithm. [Online]. Available: https://bit.ly/2JFYvDv

[32] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," *Distributed Systems (2nd Ed.). ACM Press Addison-Wesley, Expanded version appears as Technical Report TR94-1425, Department of Computer Science, Cornell University*, 1994.

[33] H. Seif. (2014) Lecture 12. unit 1. total order broadcast. [Online]. Available: https://bit.ly/2D5j1w1

[34] C. George, D. Jean, K. Tima, and B. Gordon, "Distributed systems concepts and design fifth edition," *Cambridge University, University of London, Lancaster University*, 1994.

[35] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, "Handbook of applied cryptography," *CRC Press*, pp. 31–32, 1996.

[36] T. Andrew. (2018) Proof-of-work. [Online]. Available: https://bit.ly/2EUzhxH

[37] P. Nicol. (2016, August) If i only had 5 minutes to explain blockchain. IDG Contributor Network. [Online]. Available: https://bit.ly/2Qb0AxX

[38] B. Alex and K. Dmitry. Equihash: Asymmetric proof-of-work based on the generalized birthday problem. [Online]. Available: https://bit.ly/2PDiAQx

[39] M. Chris. (2012) Hash input for nonce rule. [Online]. Available: https://bit.ly/2JIEZX5

[40] L. Hartikka. (2017) A tutorial for building a cryptocurrency, chapter 2. [Online]. Available: https://lhartikk.github.io/

[41] W. Wang, D. T. Hoang, Z. Xiong, D. Niyato, P. Wang, P. Hu, and Y. Wen, "A survey on consensus mechanisms and mining management in blockchain networks," *IEEE COMST*, p. 39, May 2018.

[42] (2015, September) Proof of stake versus proof of work. Bitfury Group. [Online]. Available: https://bit.ly/2OlT2md

[43] (2017, November) Bitcoin energy consumption index - digiconomist. [Online]. Available: https://bit.ly/2zgLqht

[44] (2017, November) Bitcoin mining now consuming more electricity than 159 countries including ireland most countries in africa. Power Compare. [Online]. Available: https://bit.ly/2B3Pij2

[45] K. Georgios. (2017, December) Understanding blockchain fundamentals, part 2: Proof of work proof of stake. [Online]. Available: https://bit.ly/2HeNWoi

[46] V. Paul. (2017) Blackcoin's proof-of-stake protocol v2. [Online]. Available: https://bit.ly/2yUdmGd

[47] Q. Leonardo. (2010) Byzantine fault tolerance for replicated state machines. Sapienza University DI Rome, Middleware Lab. [Online]. Available: https://bit.ly/2D56Hfu

[48] L. Lamport, R. Shostak, and M. Pease. (2016) The byzantine generals problem. Microsoft. [Online]. Available: https://bit.ly/2EAWcxR

[49] M. Castro and B. Liskov, "Practical byzantine fault tolerance," *MIT Laboratory for Computer Science*, February 1999.

[50] K. Driscoll1, B. Hall1, H. Sivencrona, and P. Zumsteg, "Byzantine fault tolerance, from theory to reality," *International Conference on Computer Safety, Reliability, and Security*, 2003.

[51] A. Seredinschi. Byzantine fault tolerance and consensus. Distributed Programming Laboratory. [Online]. Available: https://bit.ly/2AS4Ojz

[52] M. Merkle, "Method of providing digital signatures," *US Grant Patent, US4309569A*, 1982.

[53] C. Liu, R. Ranjan, C. Yang, X. Zhang, L. Wang, , and J. Chen, "Mur-dpa: Top-down levelled multi-replica merkle hash tree based secure public auditing for dynamic big data storage on cloud," *IEEE Trans. Comput.,*, 2015.

[54] I. Musa. Data availability proof-friendly state tree transitions. Ethresear Switzerland. [Online]. Available: https://bit.ly/2KWxDjw

[55] C. Ronald. (2018, February) Blockchain data structure. [Online]. Available: https://bit.ly/2SMI5gQ

[56] I. Labi. (2018, July) Blockchain technology primer. [Online]. Available: https://bit.ly/2Ph1sRi

[57] Electrum. (2017) Simple payment verification. [Online]. Available: http://docs.electrum.org/en/latest/spv.html

[58] T. Don and T. Alex. (2018, March) How canada can be a global leader in blockchain technology. [Online]. Available: https://tgam.ca/2s3thMP

[59] P. Kravchenko. (2016, October) Does a blockchain really need a native coin? [Online]. Available: https://bit.ly/2U3axvA

[60] J. Praveen. (2017, May) The difference between public and private blockchain. [Online]. Available: https://ibm.co/2tabuot

[61] Ethereum Foundation. (2018) Blockchain app platform. [Online]. Available: https://www.ethereum.org/

[62] K. Pavel. (2017, November) Consensus explained. [Online]. Available: https://bit.ly/2qwIbMM

[63] E. Androulaki, S. Cocco, and C. Ferris. (2018, October) Private and confidential transactions with hyperledger fabric. [Online]. Available: https://ibm.co/2F24jZh

[64] M. Castro and B. Liskov, "Practical byzantine fault tolerance," *Laboratory for Computer Science, Massachusetts Institute of Technology*, 1999.

[65] (2018) Permissioned blockchains. MONAX. [Online]. Available: https://bit.ly/2DnZ58S

[66] D. Eric. (2018, April) The blog of content protection. [Online]. Available: https://bit.ly/2ztRpNY

[67] (2018) Blockchain technologies for business. Hyperledger Foundation. [Online]. Available: https://www.hyperledger.org/

[68] (2018) Hyperledger fabric. Hyperledger Foundation. [Online]. Available: https://github.com/hyperledger/fabric/releases?after=v1.0.0-alpha2

[69] C. Cachin. (2016, July) Architecture of the hyperledger blockchain fabric. IBM Research Zurich. [Online]. Available: https://bit.ly/2NP1E9E

[70] M. Sorin, S. Senyuk, D. Gelemter, Y. Flaumernhaft, J. Cooper, and H. Zachor. (2016, February) A hotspot for blockchain innovation. Delloite.

[71] (2018) Couchdb as the state database. Hyperledger Fabric. [Online]. Available: https://bit.ly/2yWvpM5

[72] (2018) Architecture explained. Hyperledger Fabric. [Online]. Available: https://bit.ly/2zB5d9Y

[73] (2017) Hyperledger fabric consensus. Hyperledger Fabric. [Online]. Available: https://bit.ly/2vZfz0q

[74] (2018) Market capitalization of bitcoin from 1st quarter 2012 to 3rd quarter 2018 (in billion u.s. dollars). Statista. [Online]. Available: https://bit.ly/2EKbfdC

[75] (2017, November) First official registration of a zug citizen on ethereum. UPort. [Online]. Available: https://bit.ly/2Igk8g8

[76] (2018) Grid+ creates products that enable mainstream use of digital assets and cryptocurrencies. GRID PLUS. [Online]. Available: https://gridplus.io/

[77] (2018) Medicalchain uses blockchain technology to securely store health records. Medicalchain. [Online]. Available: https://medicalchain.com/en/

[78] B. Nolan. (2018) What are the applications and use cases of blockchains? [Online]. Available: https://bit.ly/2JKZfYg

[79] F. Andrew. (1998) systems, applications, products in data processing sap: Implications for computer information systems.

[80] (2018) Know what's best for your business with real-time insight into all of your data. SAP. [Online]. Available: https://www.sap.com/canada

[81] R. Karpinski. (2001) E2open at one.

[82] (2018) Spark change with your supply chain. E2Open. [Online]. Available: https://www.e2open.com/

[83] (2018) Go is an open source programming language that makes it easy to build simple, reliable, and efficient software. Golang. [Online]. Available: https://golang.org/

[84] (2018) Javascript is a resource built by the pluralsight team for the javascript community. JavaScript. [Online]. Available: https://www.javascript.com/

[85] F. David. (2018, January) What is the tangle, and is it blockchain's 'next evolutionary step'? [Online]. Available: https://bit.ly/2L2CKy8

[86] (2018) The open source blockchain for business. Corda R3. [Online]. Available: https://www.corda.net/

[87] W. Mea and L. Baochun, "R2: Random push with random network coding in live peer-to-peer streaming," *IEEE*, Decemeber 2017.

Appendix A

Data Model Implementation Code

```
namespace org.example.phoenicia

abstract participant Member {
  o String ContactInformation
}
// participants
participant Buyer identified by buyerID extends Member{
    o String buyerID
}
participant Seller identified by sellerID extends Member{
    o String sellerID
}
participant Shipper identified by shipperID extends Member {
    o String shipperID
}
participant Vendor identified by vendorID extends Member {
    o String vendorID
}
participant Bank identified by bankID extends Member {
    o String bankID
}

// assets
asset Order identified by orderID {
    o String orderID
    o String[] product
    o String status
    o String OrderRefundedDate
```

```
29      o String RefundRequestDate
30      o Integer quantity
31      o String OrderCreatedDate
32      o String OrderBuyDate
33      o String OrderCancelDate
34      o String OrderVendorDate
35      o String returnOrderDate
36      o String returnOrderToVendorDate
37      o String requestShipmentDate
38      o String deliveredDate
39      o String deliveringDate
40      o String paymentRequestDate
41      o String payDate
42      o String ReasonForCancelling
43      o Integer refundAmount
44      o String reasonForReturning
45      o Integer totalBalanceDue
46      --> Vendor vendor
47      --> Shipper shipper
48      --> Buyer buyer
49      --> Seller seller
50      --> Bank bank
51
52  }
53  // transactions
54    transaction CreateOrder {
55      o Integer quantity
56      o String[] product
57      --> Order order
58      --> Buyer buyer
59  }
60    transaction CancelOrder {
61      o String ReasonForCancelling
62      --> Order order
63      --> Buyer buyer
64      --> Seller seller
65  }
66    transaction Buy {
67      --> Order order
68      --> Buyer buyer
69      --> Seller seller
70  }
```

```
71   transaction OrderFromVendor {
72     --> Order order
73     --> Vendor vendor
74     --> Seller seller
75 }
76   transaction RequestShipping {
77     --> Order order
78     --> Shipper shipper
79     --> Vendor vendor
80 }
81   transaction Deliver {
82     --> Order order
83     --> Shipper shipper
84     --> Buyer buyer
85 }
86   transaction InDelivering {
87     --> Order order
88     --> Shipper shipper
89 }
90   transaction ReturnOrder {
91     o String reasonForReturning
92     --> Order order
93     --> Seller seller
94     --> Buyer buyer
95 }
96 transaction ReturnOrderVendor {
97   --> Order order
98   --> Seller seller
99   --> Vendor vendor
100 }
101   transaction PaymentReq{
102     o Integer totalBalanceDue
103     --> Order order
104     --> Bank bank
105 }
106   transaction Pay {
107     --> Order order
108     --> Seller seller
109     --> Bank bank
110 }
111   transaction RefundRequest {
112     --> Order order
```

```
113     --> Seller seller
114     --> Bank bank
115   }
116   transaction Refund {
117     o Integer refundAmount
118     --> Order order
119     --> Seller seller
120     --> Bank bank
121 }
122 // Events
123 event BuyNotification{
124   --> Order order
125   --> Seller seller
126 }
127 event CancelNotification{
128   --> Order order
129   --> Buyer buyer
130 }
131 event VendorNotification {
132   --> Order order
133   --> Vendor vendor
134 }
135 event OnRouteNotification {
136   --> Order order
137   --> Buyer buyer
138 }
139 event DeliveredNotification {
140   --> Order order
141   --> Buyer buyer
142 }
143 event PaymentNotification {
144   --> Order order
145   --> Seller seller
146 }
147 event RefundNotification {
148   --> Order order
149   --> Bank bank
150 }
151 event ReturnNotification {
152   --> Order order
153   --> Seller seller
```

```
154 }
```

Listing A.1: Data Model Implementation

Appendix B

Smart Contracts Implementation Code

```
1  'use strict';
2
3  var StatusList = {
4      "OrderCreateStatus": "Created Order",
5      "OrderBuyStatus": "Bought Order",
6      "OrderCancelStatus": "Cancelled Order",
7      "OrderFromVendorStatus": "Seller Sent Order to the Vendor",
8      "ShipRequestStatus": "Vendor Requests Shipper to Ship Order",
9      "DeliveredStatus": "Deliver Order",
10     "InDeliveringStatus": "Order On The Way",
11     "OrderReturnStatus": "Return Order to Seller",
12     "PaymentReq": "Seller Requests Bank to Pay",
13     "PayStatus": "Payment Processed",
14     "RefundReq": "Bank Requests Seller to Pay Back",
15     "Refunded": "Refunded Order",
16     "ReturnOrderVendorStatus": "Returned Order to Vendor"
17 };
18
19 /**
20  * Buyer creates the order to Buy
21  * @param {org.example.phoenicia.CreateOrder} tx - the order to be
       processed
22  * @transaction
23  */
24 function CreateOrder(tx) {
25     tx.order.buyer = tx.buyer;
```

```
26      tx.order.quantity = tx.quantity;
27      tx.order.product = tx.product;
28      tx.order.OrderCreatedDate = new Date().toISOString();
29      tx.order.status = JSON.stringify(StatusList.OrderCreateStatus);
30      return getAssetRegistry('org.example.phoenicia.Order')
31          .then(function (assetRegistry) {
32              return assetRegistry.update(tx.order);
33          });
34  }
35  /**
36   * Buyer requests Buy from Seller
37   * @param {org.example.phoenicia.Buy} tx - the order to be processed
38   * @transaction
39   */
40  function Buy(tx) {
41      if (tx.order.status == JSON.stringify(StatusList.
        OrderCreateStatus))
42      {
43          tx.order.buyer = tx.buyer;
44          tx.order.seller = tx.seller;
45          tx.order.OrderBuyDate = new Date().toISOString();
46          tx.order.status = JSON.stringify(StatusList.OrderBuyStatus);
47          const buy = getFactory().newEvent('org.example.phoenicia', '
            BuyNotification');
48          buy.order = tx.order;
49          buy.seller = tx.order.seller;
50          emit(buy);
51          return getAssetRegistry('org.example.phoenicia.Order')
52              .then(function (assetRegistry) {
53                  return assetRegistry.update(tx.order);
54              });
55      }
56      else {
57          throw new Error ('Create the order first');
58      }
59  }
60  /**
61   * Buyer cancels the order
62   * @param {org.example.phoenicia.CancelOrder} tx - the order to be
       processed
63   * @transaction
64   */
```

```
65 function CancelOrder(tx) {
66     if ((tx.order.status == JSON.stringify(StatusList.
       OrderCreateStatus)) || (tx.order.status == JSON.stringify(
       StatusList.OrderBuyStatus)))
67     {
68         tx.order.buyer = tx.buyer;
69         tx.order.seller = tx.seller;
70         tx.order.OrderCancelDate = new Date().toISOString();
71         tx.order.ReasonForCancelling = tx.ReasonForCancelling;
72         tx.order.status = JSON.stringify(StatusList.
           OrderCancelStatus);
73         const cancel = getFactory().newEvent('org.example.phoenicia'
           , 'CancelNotification');
74         cancel.order = tx.order;
75         cancel.buyer = tx.order.buyer;
76         emit(cancel);
77         return getAssetRegistry('org.example.phoenicia.Order')
78             .then(function (assetRegistry) {
79                 return assetRegistry.update(tx.order);
80             });
81         }
82 }
83 /**
84  * Seller sends the Order request to Vendor
85  * @param {org.example.phoenicia.OrderFromVendor} tx - the order to
    be processed
86  * @transaction
87  */
88 function OrderFromVendor(tx) {
89     if (tx.order.status == JSON.stringify(StatusList.OrderBuyStatus)
       )
90     {
91         tx.order.vendor = tx.vendor;
92         tx.order.OrderVendorDate = new Date().toISOString();
93         tx.order.status = JSON.stringify(StatusList.
           OrderFromVendorStatus);
94         const vendorNotify = getFactory().newEvent('org.example.
           phoenicia', 'VendorNotification');
95         vendorNotify.order = tx.order;
96         vendorNotify.vendor = tx.order.vendor;
97         emit(vendorNotify);
98         return getAssetRegistry('org.example.phoenicia.Order')
```

```
99              .then(function (assetRegistry) {
100                 return assetRegistry.update(tx.order);
101             });
102     }
103     else {
104         throw new Error ('The vendor has already received this order
                ');
105     }
106 }
107 /**
108  * Vendor requests Shipper to deliver the Order to Buyer
109  * @param {org.example.phoenicia.RequestShipping} tx - the order to
        be processed
110  * @transaction
111  */
112 function RequestShipping(tx) {
113     if (tx.order.status == JSON.stringify(StatusList.
            OrderFromVendorStatus))
114     {
115         tx.order.shipper = tx.shipper;
116         tx.order.requestShipmentDate = new Date().toISOString();
117         tx.order.status = JSON.stringify(StatusList.
                ShipRequestStatus);
118         return getAssetRegistry('org.example.phoenicia.Order')
119             .then(function (assetRegistry) {
120                 return assetRegistry.update(tx.order);
121             });
122         }
123     else {
124         throw new Error ('The order has already been requested for
                shipping');
125     }
126 }
127 /**
128  * The shipment is on its way to Buyer
129  * @param {org.example.phoenicia.InDelivering} tx - the order to be
        processed
130  * @transaction
131  */
132 function InDelivering(tx) {
133     if (tx.order.status == JSON.stringify(StatusList.
            ShipRequestStatus))
```

```
134     {
135         tx.order.deliveringDate = new Date().toISOString();
136         tx.order.status = JSON.stringify(StatusList.
                InDeliveringStatus);
137         const onroute = getFactory().newEvent('org.example.phoenicia
                ', 'OnRouteNotification');
138         onroute.order = tx.order;
139         onroute.buyer = tx.order.buyer;
140         emit(onroute);
141         return getAssetRegistry('org.example.phoenicia.Order')
142             .then(function (assetRegistry) {
143                 return assetRegistry.update(tx.order);
144             });
145         }
146     else if (tx.order.status == JSON.stringify(StatusList.
            DeliveredStatus))
147     {
148         throw new Error ('The buyer has already received this order'
                );
149     }
150     else {
151         throw new Error ('The order has not been requested for
                shipping yet');
152     }
153 }
154 /**
155  * The shipment has arrived
156  * @param {org.example.phoenicia.Deliver} tx - the order to be
        processed
157  * @transaction
158  */
159 function Deliver(tx) {
160         tx.order.deliveredDate = new Date().toISOString();
161         tx.order.status = JSON.stringify(StatusList.DeliveredStatus)
                ;
162         const delivered = getFactory().newEvent('org.example.
                phoenicia', 'DeliveredNotification');
163         delivered.order = tx.order;
164         delivered.buyer = tx.order.buyer;
165         emit(delivered);
166         return getAssetRegistry('org.example.phoenicia.Order')
167             .then(function (assetRegistry) {
```

```
168                 return assetRegistry.update(tx.order);
169             });
170 }
171  /**
172  * Seller request payment from Bank
173  * @param {org.example.phoenicia.PaymentReq} tx- the order to be
        processed
174  * @transaction
175  */
176 function PaymentReq(tx) {
177     if (tx.order.status == JSON.stringify(StatusList.DeliveredStatus
           ))
178         {
179         tx.order.totalBalanceDue = tx.totalBalanceDue;
180         tx.order.bank = tx.bank;
181         tx.order.status = JSON.stringify(StatusList.PaymentReq);
182         tx.order.paymentRequestDate = new Date().toISOString();
183 
184     return getAssetRegistry('org.example.phoenicia.Order')
185         .then(function (assetRegistry) {
186             return assetRegistry.update(tx.order);
187         });
188     }
189     else if (tx.order.status == JSON.stringify(StatusList.
           InDeliveringStatus)) {
190         throw new Error ('The order is still on route and payment
               Request is not allowed');
191     }
192     else {
193         throw new Error ('The order has been already requested for
               payment');
194     }
195 }
196  /**
197  * Bank makes a payment to Seller
198  * @param {org.example.phoenicia.Pay} tx - the order to be processed
199  * @transaction
200  */
201 function Pay(tx) {
202     if (tx.order.status == JSON.stringify(StatusList.PaymentReq))
203     {
204         tx.order.status = JSON.stringify(StatusList.PayStatus);
```

```
205         tx.order.payDate = new Date().toISOString();
206         const paid = getFactory().newEvent('org.example.phoenicia',
                'PaymentNotification');
207         paid.order = tx.order;
208         paid.seller = tx.order.seller;
209         emit(paid);
210         return getAssetRegistry('org.example.phoenicia.Order')
211         .then(function (assetRegistry) {
212             return assetRegistry.update(tx.order);
213         });
214     }
215     else if (tx.order.status = JSON.stringify(StatusList.PayStatus))
           {
216         throw new Error ('The order has already been paid');
217     }
218 }
219 /**
220  * Bank request payment refund from Seller
221  * @param {org.example.phoenicia.RefundRequest} tx - the order to be
        processed
222  * @transaction
223  */
224 function RefundRequest(tx) {
225     if (tx.order.status == JSON.stringify(StatusList.
           OrderReturnStatus)){
226     tx.order.status = JSON.stringify(StatusList.RefundReq);
227     tx.order.RefundRequestDate = new Date().toISOString();
228 
229     return getAssetRegistry('org.example.phoenicia.Order')
230     .then(function (assetRegistry) {
231         return assetRegistry.update(tx.order);
232       });
233     }
234     else if (tx.order.status = JSON.stringify(StatusList.RefundReq))
            {
235         throw new Error ('The order has already been requested for
                refund');
236     }
237     else {
238         throw new Error ('The order has not been returned to Seller'
                );
239     }
```

```
240
241 }
242  /**
243  * Seller makes a refund payment to Bank
244  * @param {org.example.phoenicia.Refund} tx - the order to be
        processed
245  * @transaction
246  */
247 function Refund(tx) {
248     if (tx.order.status == JSON.stringify(StatusList.RefundReq)) {
249         tx.order.status = JSON.stringify(StatusList.Refunded);
250         tx.order.refundAmount= tx.refundAmount;
251         tx.order.OrderRefundedDate = new Date().toISOString();
252         const refunded = getFactory().newEvent('org.example.
            phoenicia', 'RefundNotification');
253         refunded.order = tx.order;
254         refunded.bank = tx.order.bank;
255         emit(refunded);
256         return getAssetRegistry('org.example.phoenicia.Order')
257         .then(function (assetRegistry) {
258             return assetRegistry.update(tx.order);
259         });
260     }
261     else if (tx.order.status = JSON.stringify(StatusList.Refunded)){
262         throw new Error ('The order has already been refunded');
263     }
264 }
265  /**
266  * Buyer returns the Order to Seller
267  * @param {org.example.phoenicia.ReturnOrder} tx - the order to be
        processed
268  * @transaction
269  */
270 function ReturnOrder(tx) {
271         tx.order.status = JSON.stringify(StatusList.
            OrderReturnStatus);
272         tx.order.reasonForReturning = tx.reasonForReturning;
273         tx.order.returnOrderDate = new Date().toISOString();
274         const returned = getFactory().newEvent('org.example.
            phoenicia', 'ReturnNotification');
275         returned.order = tx.order;
276         returned.seller = tx.order.seller;
```

```
277         emit(returned);
278         return getAssetRegistry('org.example.phoenicia.Order')
279         .then(function (assetRegistry) {
280             return assetRegistry.update(tx.order);
281         });
282 }
283  /**
284  * Seller returns the Order to Vendor
285  * @param {org.example.phoenicia.ReturnOrderVendor} tx - the order
        to be processed
286  * @transaction
287  */
288 function ReturnOrderVendor(tx) {
289     if (tx.order.status == JSON.stringify(StatusList.
        OrderReturnStatus)){
290     tx.order.status = JSON.stringify(StatusList.
        ReturnOrderVendorStatus);
291     tx.order.returnOrderToVendorDate = new Date().toISOString();
292
293     return getAssetRegistry('org.example.phoenicia.Order')
294     .then(function (assetRegistry) {
295         return assetRegistry.update(tx.order);
296     });
297 }
298     else {
299       throw new Error ('the buyer did not return the order');
300     }
301
302 }
```

Listing B.1: Smart Contracts Implementation

Appendix C

Query and Access Control List Implementation Codes

```
/* Query all orders in the network
*/

query Orders {
  description: "Select all orders"
  statement:
     SELECT org.example.phoenicia.Order
}

/* Query orders by Buyer in the network
* List all orders for a certain Buyer
*/

query OrderByBuyer {
  description: "Select all orders based on their Buyer"
  statement:
    SELECT org.example.phoenicia.Order
      WHERE (buyer == _$buyer)
}

/* Query orders by Seller in the network
* List all orders for a certain Seller
*/

query OrderBySeller {
  description: "Select all orders based on their Seller"
```

```
27    statement:
28      SELECT org.example.phoenicia.Order
29        WHERE (seller == _$seller)
30  }
31
32  /* Query orders by Quantity in the network
33  * List all orders for a certain Amount
34  */
35
36  query OrderByQuantity {
37    description: "Select all orders based on certain amount"
38    statement:
39      SELECT org.example.phoenicia.Order
40        WHERE (quantity > 20)
41  }
42
43  /* Query orders by Product Type in the network
44  * List all orders for a certain Product
45  */
46
47  query OrderByProduct {
48    description: "Select all orders based on their product type"
49    statement:
50      SELECT org.example.phoenicia.Order
51        WHERE (product == _$product)
52  }
```

Listing C.1: Query Implementation Code

```
1
2   \* Access Control List */
3   rule ParticipantCanSeeOnlyTheirRecords {
4   description: "Participants can perfrom on their own information
       instance"
5   participant(t): "org.example.phoenicia.participant"
6   operation: READ, UPDATE, CREATE
7   resource(v): "org.example.phoenicia.participant"
8   condition: (v.getIdentifier() == t.getIdentifier())
9   action: ALLOW
10  }
11
12  rule ParticipantCanPerformOnTheirAsset {
13  description: "Participants can see with their own asset"
```

```
14 participant(t): "org.example.phoenicia.participant"
15 operation: ALL
16 resource(c): "org.example.phoenicia.Order"
17 condition: (c.owner.getIdentifier() == t.getIdentifier())
18 action: ALLOW
19 }
20
21
22 rule ParticipantCanSeeTheirHistoryOfTransactions {
23 description: "Participants can see history of their own transactions
     "
24 participant(t): "org.example.phoenicia.participant"
25 operation: READ
26 resource(v): "org.hyperledger.composer.system.HistorianRecord"
27 condition: (v.participantInvoking.getIdentifier() != t.getIdentifier
     ())
28 action: DENY
29 }
30
31 rule EverybodyCanReadEverything {
32     description: "Allow all participants read access to all
         resources"
33     participant: "org.example.phoenicia.*"
34     operation: READ
35     resource: "org.example.phoenicia.*"
36     action: ALLOW
37 }
38
39 rule EverybodyCanSubmitTransactions {
40     description: "Allow all participants to submit transactions"
41     participant: "org.example.phoenicia.*"
42     operation: CREATE
43     resource: "org.example.phoenicia.*"
44     action: ALLOW
45 }
46
47 rule SystemACL {
48   description:  "System ACL to permit all access"
49   participant: "org.hyperledger.composer.system.Participant"
50   operation: ALL
51   resource: "org.hyperledger.composer.system.**"
52   action: ALLOW
```

```
53  }
54
55  rule NetworkAdminUser {
56      description: "Grant business network administrators full access
           to user resources"
57      participant: "org.hyperledger.composer.system.NetworkAdmin"
58      operation: ALL
59      resource: "**"
60      action: ALLOW
61  }
62
63  rule NetworkAdminSystem {
64      description: "Grant business network administrators full access
           to system resources"
65      participant: "org.hyperledger.composer.system.NetworkAdmin"
66      operation: ALL
67      resource: "org.hyperledger.composer.system.**"
68      action: ALLOW
69  }
```

Listing C.2: Access Control List Implementation Code

Appendix D

Experiments Configuration

```
{
  "blockchain": {
    "type": "composer",
    "config": "network/fabric/2-org-1-peer/composer.json"
  },
  "command" : {
    "start": "docker-compose -f network/fabric/2-org-1-peer/docker-
       compose.yaml up -d"
  },
  "test": {
    "name": "Performance test",
    "clients": {
      "type": "local",
      "number": 1
    },
    "rounds": [{
                "label" : "sc-maher",
                "txNumber" : [10],
                "rateControl" : [{"type": "fixed-rate", "opts": {"
                   tps" : 10}}],
                "arguments": {"testAssets": 10},
                "callback" : "benchmark/sc-maher.js"
            },
            {
                 "label" : "sc-maher",
                 "txNumber" : [20],
                  "rateControl" : [{"type": "fixed-rate", "opts": {"
                     tps" : 10}}],
```

```
26                    "arguments": {"testAssets": 20},
27                    "callback" : "benchmark/sc-maher.js"
28              }
29              ]
30   }
31 }
```

Listing D.1: Experiement I. Method A

```
1  {
2    "blockchain": {
3      "type": "composer",
4      "config": "network/fabric/2-org-1-peer/composer.json"
5    },
6    "command" : {
7      "start": "docker-compose -f network/fabric/2-org-1-peer/docker-
          compose.yaml up -d"
8    },
9    "test": {
10     "name": "Performance test",
11     "clients": {
12       "type": "local",
13       "number": 1
14     },
15     "rounds": [{
16                 "label" : "sc-maher",
17                 "txNumber" : [10],
18                 "rateControl" : [{"type": "fixed-feedback-rate", "
                      opts": {"unfinished_per_client": 13, "tps": 10, "
                      sleep_time": 1000}}],
19                 "arguments": {"testAssets": 10},
20                 "callback" : "benchmark/sc-maher.js"
21             },
22             {
23                  "label" : "sc-maher",
24                  "txNumber" : [20],
25                   "rateControl" : [{"type": "fixed-feedback-rate", "
                        opts": {"unfinished_per_client": 13, "tps": 10,
                         "sleep_time": 1000}}],
26                   "arguments": {"testAssets": 20},
27                   "callback" : "benchmark/sc-maher.js"
28             }
29             ]
```

```
30   }
31 }
```

Listing D.2: Experiement I. Method B

```
1
2  {
3    "blockchain": {
4      "type": "composer",
5      "config": "network/fabric/2-org-1-peer/composer.json"
6    },
7    "command" : {
8      "start": "docker-compose -f network/fabric/2-org-1-peer/docker-
          compose.yaml up -d"
9    },
10   "test": {
11     "name": "Performance test",
12     "clients": {
13       "type": "local",
14       "number": 1
15     },
16     "rounds": [{
17                  "label" : "sc-maher",
18                  "txNumber" : [60],
19                  "rateControl" : [{"type": "fixed-rate", "opts": {"
                       tps" : 20}}],
20                  "arguments": {"testAssets": 60},
21                  "callback" : "benchmark/sc-maher.js"
22              },
23              {
24                   "label" : "sc-maher",
25                   "txNumber" : [60],
26                    "rateControl" : [{"type": "fixed-rate", "opts": {"
                        tps" : 40}}],
27                    "arguments": {"testAssets": 60},
28                    "callback" : "benchmark/sc-maher.js"
29              }
30              ]
31   }
32 }
```

Listing D.3: Experiement II. Method A

```
1  {
```

```
2    "blockchain": {
3      "type": "composer",
4      "config": "network/fabric/2-org-1-peer/composer.json"
5    },
6    "command" : {
7      "start": "docker-compose -f network/fabric/2-org-1-peer/docker-
         compose.yaml up -d"
8    },
9    "test": {
10     "name": "Performance test",
11     "clients": {
12       "type": "local",
13       "number": 1
14     },
15     "rounds": [{
16                 "label" : "sc-maher",
17                 "txNumber" : [60],
18                 "rateControl" : [{"type": "fixed-feedback-rate", "
                     opts": {"unfinished_per_client": 13, "tps": 20, "
                     sleep_time": 1000}}],
19                 "arguments": {"testAssets": 60},
20                 "callback" : "benchmark/sc-maher.js"
21             },
22             {
23                  "label" : "sc-maher",
24                  "txNumber" : [60],
25                   "rateControl" : [{"type": "fixed-feedback-rate", "
                       opts": {"unfinished_per_client": 13, "tps": 40
                       , "sleep_time": 1000}}],
26                   "arguments": {"testAssets": 60},
27                   "callback" : "benchmark/sc-maher.js"
28             }
29             ]
30   }
31 }
```

Listing D.4: Experiement II. Method B