Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis

Evaluating Various Transaction Processing Characteristics of Permissioned Blockchain Networks

Vikas Khinchi

Course of Study: Infotech

Examiner: Prof. Dr. Dr. h. c. Frank Leymann

Supervisor: Ghareeb Falazi, M.Sc.

Commenced: April 30, 2018

Completed: October 30, 2018

Abstract

Blockchain is a distributed ledger technology that completely removes the requirement of third parties like banks in exchanging assets and performing business transactions and every participant maintains a local copy of the data. In permissionless blockchains like Ethereum or Bitcoin, any entity can participate in submitting and validating a transaction as the entire blockchain network is public. On the other hand, permissioned blockchains only allow certain entities that have the right permission to be included in the blockchain network and to participate in transaction execution and validation. Hyperledger Fabric is one such example of a permissioned blockchain system that provides privacy and confidentiality by using smaller private networks characterized by channels. Thus, Hyperledger Fabric can be used to study the business transactions.

The main aim of this thesis is to investigate the transactional properties of permissioned blockchain networks like Hyperledger Fabric and evaluate them by comparing them to the ACID properties of Relational Database Management Systems (RDBMS). A supply chain application has been developed that demonstrates the various business transactions like getting product details, creating a new product and updating a product. Hyperledger Fabric performs business transactions with the help of chaincodes that are deployed on peers and channels are introduced to provide privacy. A web front-end has been created that serves as a 2-PC transaction monitor for writing data to different channels within a single transaction. Business transactions are mostly characterized by many nested sub-transactions and hence, an equivalent system that fulfills this requirement with blockchain is developed. An Android application has also been developed that provides the same functions with a focus on end user. Also the immutability of blockchain involving number of peers is studied and it is found that the blockchain itself is immutable but the state database on a single peer can be modified in absence of efficient endorsement policy.

Contents

1 Introduction **17**
1.1 Problem Statement . . . 18
1.2 Scope of Work . . . 19
1.3 Outline . . . 19

2 Fundamentals **21**
2.1 Introduction to Permissioned Blockchains . . . 21
2.2 Hyperledger Fabric as Permissioned Blockchain Network . . . 21
2.2.1 Consensus . . . 22
2.2.2 Membership Service Provider (MSP) . . . 25
2.2.3 Chaincodes . . . 27
2.2.4 Endorsement Policy . . . 28
2.2.5 Channels . . . 30
2.2.6 Ledger . . . 31
2.2.7 Data Storage . . . 33
2.3 Hyperledger Sawtooth as Permissioned Blockchain Network . . . 33
2.3.1 Characteristics of Hyperledger Sawtooth . . . 34
2.3.2 Transactions and Batches in Sawtooth . . . 34

3 Related Work **37**
3.1 A Transaction Processing Perspective on Blockchains . . . 37
3.1.1 Motivation . . . 37
3.1.2 Study Approach . . . 37
3.2 Hyperledger Fabric: A distributed Operating System for Permissioned Blockchains 38
3.2.1 Motivation . . . 38
3.2.2 Study Approach . . . 38
3.3 Blockchain Consensus Protocols in the Wild . . . 38
3.3.1 Motivation . . . 39
3.3.2 Study Approach . . . 39

4 Blockchain Transaction Characteristics **41**
4.1 Transaction Processing Models . . . 41
4.1.1 Order-Execute Model . . . 41
4.1.2 Execute-Order-Validate Model . . . 43
4.2 Transaction Flow in Hyperledger Fabric . . . 45
4.3 Hyperledger Fabric Transactional Properties from an ACID Perspective . . . 46
4.3.1 Atomicity . . . 46
4.3.2 Consistency . . . 47
4.3.3 Isolation . . . 48

4.3.4 Durability . . . 48
4.4 Permissionless Blockchain Transaction Properties . . . 49
4.4.1 Transaction View for Blockchain Transactions . . . 49
4.4.2 System View for Blockchain Transactions . . . 50
4.5 Differences between Permissioned and Permissionless Blockchains based on Transactional Properties . . . 51
4.6 Effect of Consensus Algorithms on Transactional Properties . . . 52
4.7 Effect of Channels on Transactional Properties . . . 53
4.8 Effect of Chaincodes on Transactional Properties . . . 53
4.9 Effect of Endorsement Policy on Transactional Properties . . . 54
4.10 Other Miscellaneous Transaction Terms . . . 54
4.10.1 Peer Transaction Manager . . . 54
4.10.2 Savepoint . . . 55
4.10.3 Rollback Transactions in Hyperledger Fabric . . . 55
4.10.4 Compensating Transactions . . . 55

5 Concept, Specification and Design of a Permissioned Blockchain Network 57
5.1 System Overview . . . 57
5.2 System Architecture . . . 58
5.3 User Roles . . . 59
5.4 Use Cases . . . 61
5.5 Data Model . . . 65
5.6 Chaincode functions . . . 66
5.7 Web Front-End Application . . . 70
5.7.1 Get All Products . . . 70
5.7.2 Get Product by Id . . . 71
5.7.3 Create a Product . . . 72
5.7.4 Update a Product . . . 73
5.8 Android Application . . . 73

6 Experimental Design and Results 77
6.1 Experiment 1: Chaincode to Chaincode Call on Same Channel . . . 77
6.1.1 Read from Same Channel . . . 77
6.1.2 Write on same channel . . . 82
6.2 Experiment 2: Chaincode to Chaincode Call on Different Channels . . . 85
6.2.1 Read from Different Channels . . . 85
6.2.2 Write on Different Channels . . . 88
6.3 Experiment 3: Implementation of an Intermediate Layer for Transaction Management 90
6.3.1 Use Case Scenario . . . 91
6.3.2 Results . . . 92
6.3.3 Conclusion of Results . . . 93
6.4 Experiment 4: Test for Immutability of Ledger . . . 93
6.4.1 Modifying the World State . . . 93
6.4.2 Modifying the Peer Blockchain File . . . 94
6.4.3 Modifying the Peer and Orderer Blockchain and then Joining a New Peer 95

7 Conclusion 99

List of Figures

2.1 Comparison of commonly used Consensus algorithms in Permissioned blockchains [WG18b] 24
2.2 Figure showing how a local MSP is stored on a local file system [Fab18g] 26
2.3 An example of a blockchain network with two organizations, their corresponding MSP, the peer, the orderer and the channel [Fab18g] 30
2.4 Figure showing a Ledger consisting of the world state and the blockchain [Fab18e] 31
2.5 Figure showing the structure of the blockchain ledger [Fab18e] 32

3.1 The consensus resilience properties in different blockchain networks [CV17] . . . 39

4.1 Figure showing the order-execute architectural style in blockchain network [ABB+18] 42
4.2 The execute-order-validate architecture model of transactions in Hyperledger Fabric [ABB+18] 43
4.3 Transaction flow in Hyperledger Fabric that follows an execute-order-validate architectural style [Fab18a] 45
4.4 Differences in transactional properties of permissioned and permissionless blockchains at the transaction level. 51
4.5 Differences in transactional properties of permissioned and permissionless blockchains at the block level. 52

5.1 The architecture of the developed application. Methods marked with (*) will not be accessible to the corresponding peers. The method $getPrice()^1$ will fetch price for channel between Producer and Supplier, $getPrice()^3$ will fetch price for channel betwwen Supplier and Buyer, $getPrice()^2$ will fetch prices for both the channels, $getPrice()^4$ will fetch price for channel between buyer and consumer 58
5.2 Figure showing different participants in the blockchain application. 59
5.3 Use case diagram for the different participants. The first figure shows the use case diagram for producer, second figure shows the use case diagram for supplier, third figure for the buyer and fourth figure for the consumer. 60
5.4 Figure showing the schema of product and price that are maintained in two different ledgers. 65
5.5 The view for displaying all the products in the blockchain. 70
5.6 The view for fetching a product by its id in the blockchain. 71
5.7 The view for creating a new product and the acknowledgement of the transaction with transaction id 72
5.8 The view for updating a product with the new owner. 73
5.9 The mobile app screens designed for the blockchain network 74
5.10 The architecture of the mobile application interacting with the blockchain network. 75
5.11 The extension of the app that can be used to view the status and other information about a product. 75

6.1 The overall flow of read operation from a different chaincode on the same channel. 81
6.2 The overall flow of write operation from a different chaincode on the same channel. 84
6.3 The overall flow of read operation from a different chaincode on different channels 87
6.4 The overall flow of write operation from a different chaincode on a different channel. 89
6.5 The overall flow of two separate transactions managed within a single transaction context. 91
6.6 The view of the software layer that handles transaction on multiple channels. . . 92

List of Tables

2.1 Table showing classification of consensus types. 22

5.1 Description for the use case *Get all Products* for the participants *Producer*, *Supplier*, *Buyer*. 61
5.2 Description for the use case *Get Product by Id* for the participant *Producer*, *Buyer*, *Supplier*, *Consumer*. 62
5.3 Description for the use case *Create a Product* for the participant *Producer*. 63
5.4 Description for the use case *Update a Product* for the participant *Buyer*, *Supplier*. 64

6.1 Different values for the keys during different chaincode operations for read from same channel. 82
6.2 Different values for the keys during different chaincode operations during write on same channel. 84
6.3 Different values for the keys during different chaincode operations during read from different channels. 88
6.4 Different values for the keys during different chaincode operations during write on different channels. 90

List of Listings

2.1 Endorsement policy specification at the time of chaincode instantiation. 29
2.2 Different peer commands to create a channel and to join the peer to the created channel [Fab18c]. 31
5.1 Code snippet for the Invoke method in the product-chaincode.go 66
5.2 Code snippet for the *queryProduct* method in the product-chaincode.go 67
5.3 Code snippet for the *createProduct* method in the product-chaincode.go 67
5.4 Code snippet for the *queryAllProducts* method in the product-chaincode.go . . . 68
5.5 Code snippet for the *changeProductOwner* method in the product-chaincode.go . 69
5.6 Code snippet for the *main* method in the product-chaincode.go 69
5.7 Code snippet for the connecting to the Fabric Client SDK. 70
5.8 Code snippet for the *queryAllProducts* method in the controller function that calls the chaincode with the chaincodeId 71
5.9 Code snippet for the *queryProduct* method in the controller function that calls the chaincode with the chaincodeId 71
5.10 Code snippet for the *createProduct* method in the controller function that calls the chaincode with the chaincodeId 72
5.11 Code snippet for the *changeProductOwner* method in the controller function that calls the chaincode with the chaincodeId 73
6.1 Code snippet for the *invoke* method in the chaincode1 78
6.2 Code snippet for the *invoke* method in the chaincode2 79
6.3 CLI command for chaincode *install* on same channel for read operation. 80
6.4 CLI command for chaincode *instantiate* on same channel for read operation. . . . 80
6.5 CLI command for chaincode *query* on same channel for read operation. 80
6.6 CLI command for chaincode *invoke* on same channel for read operation. 81
6.7 CLI command for chaincode *query* on same channel for read operation. 81
6.8 Code snippet for the *write* method in the chaincode2 83
6.9 CLI command for chaincode *invoke* with write operation on same channel. 83
6.10 CLI command for chaincode *install* for read operation on different channels. . . . 85
6.11 CLI command for chaincode *instantiate* for read operation on different channels. . 86
6.12 CLI command for chaincode *query* for read operation on different channels. . . . 86
6.13 CLI command for chaincode *invoke* for read operation on different channels. . . . 86
6.14 CLI command for chaincode *query* for read operation on different channels. . . . 87
6.15 CLI command for chaincode *invoke* for write operation on different channels. . . 89
6.16 The different objects created in the experiment to test the immutability of ledger. . 96

List of Abbreviations

BFT Byzantine Fault Tolerant. 21

CA Certificate Authority. 25

CFT Crash Fault Tolerant. 21

CSS Cascading Style Sheet. 18

DLT Distributed Ledger Technology. 18

DoS Denial of Service. 43

ESCC Endorsement System Chaincode. 27

HTML Hyper Text Markup Language. 18

JSON JavaScript Object Notification. 33

MSP Membership Service Provider. 18

OSN Ordering Service Node. 51

OU Organizational Unit. 26

PBFT Practical Byzantine Fault Tolerance. 23

PoET Proof of Elapsed Time. 22

PoW Proof of Work. 22

PTM Peer Transaction Manager. 54

RBFT Redundant Byzantine Fault Tolerance. 23

RDBMS Relational Database Management System. 3

UTXO Unspent Transaction Output. 28

VSCC Validation System Chaincode. 27

1 Introduction

ACID defines the transaction properties in RDBMS that help to achieve reliability in business transactions. The main focus is on achieving database consistency. The significance of the ACID properties is described as follows :

1. Atomicity: The transaction is considered as an atomic unit of work and a transaction can either be committed successfully in the database or it can be aborted due to a failure but it can never be partially committed. There are two main operations **Abort** and **Commit**. It is also called as `All or nothing` property [MOE09].

2. Consistency: The total value of asset in a database before a transaction and its value after the transaction (committed or aborted) is always the same. In other words, it maintains consistency by adhering to the database constraints. Inconsistent databases can lead to catastrophic events that cannot be rolled back in most cases.

3. Isolation: Isolation ensures that multiple transactions can take place parallely without resulting in an inconsistent state in the database. This is equivalent to the transaction result execution in sequential manner with the advantages of concurrency control. Isolation guarantees that the changes made by a transaction are only visible only after its written to the main memory [MOE09].

4. Durability: It is expected that the database remain robust in case of hardware failures and the data generated in the process of transaction execution does not get lost due to this catastrophic failure. Durability ensures that the data remains persistent in case of transaction commitment and never gets lost due to system failure. Physical storages like Hard disks are used to guarantee durability of transactions.

The ACID properties in RDBMS provides a way to maintain consistency, data persistence, robustness, parallel transaction execution and avoid double spending. Typically, it is the responsibility of the *transaction manager* component of a Transaction Processing system that involves co-ordination with different *resource managers* [TEK17].

Blockchain has evolved as a peer to peer system that eliminates the need to have a trusted third party that manages the business transactions. It consists of immutable ledgers that are the only source of truth in the system and which cannot be modified by a single entity and every peer maintains a copy of the ledger. It is due to these important properties that blockchains have found applications in various industries like Supply Chain Management, Financial organizations and any network involving many participants having potential trust issues [CNS15]. As blockchains have found way into the business use cases, it becomes extremely essential to study the transaction properties of blockchains. A typical business transaction involves many sub-transactions that are related to each other. It becomes important to study the behaviour of a blockchain system as monetary values are involved in the business transactions and the transactions are spanned across different entities in the network and are sometimes inter-linked with each other. It is important to study the behaviour of

blockchain systems by comparing it to the ACID properties of the traditional RDBMS. A blockchain system is generally classified into two different types of networks based on the ability of having controlled or uncontrolled admission in the blockchain network: **Permissionless Blockchains** and **Permissioned Blockchains**.

Permissionless Blockchain: In a permissionless blockchain network, anyone can join the channel and act as miner/validator and anyone can submit a transaction as there is no central authority that authorizes who can join the network [FF18]. Miners are given incentives to remain in the network and provide resources for calculating the hash and that results in a new block creation. Typically, the transactions are visible to the public but the identity of the node submitting a transaction is not known. The blockchain network is mostly driven by a cryptocurrency that is used for submitting a transaction. Examples of such network include *Ethereum* and *Bitcoin*.

Permissioned Blockchain: A permissioned blockchain network has a central authority that governs the rules about who can join the network and submit the transactions. Most of the permissioned blockchain networks are built around the B2B use cases and hence, involves proper management about the right entities joining the network. *Hyperledger Fabric* is one such example of a permissioned blockchain network that follows a modular architectural style and can be adapted according to the needs of the business organizations [ABB+18]. It has an important concept of channels that provides privacy and confidentiality to a subset of participants in the network.

Both the permissioned and permissionless blockchain networks have their application in the B2B scenario and therefore, it becomes imperative to compare and evaluate them with the existing traditional RDBMS and verify if the ACID properties are as strong as in the RDBMS. This will provide us with an overview if the Blockchain technology can still be used efficiently in the B2B scenarios and the steps towards making it robust, secure and immutable.

1.1 Problem Statement

In the past, studies have been conducted to evaluate the transaction processing characteristics of permissionless blockchain networks, which use consensus algorithms like Proof-of-Work or Proof-of-Stake [Eth] for transaction processing, and it is found that they follow different semantics as compared to the RDBMS [TEK17]. From the transactions perspective, permissionless blockchain networks are generally classified as *Sequential*, *Agreed*, *Ledgered* and *Tamper-resistant* [TEK17]. On the other hand, these permissionless blockchain networks are characterized as *Symmetric*, *Admin-free*, *Ledgered* and *Time-consensual* from a systems perspective [TEK17]. Moreover, only little research has been done on the transaction processing properties of permissioned blockchain networks that incorporate the same underlying concept of Distributed Ledger Technologys (DLT) as the permissionless blockchains, but differ substantially in the transaction processing and execution. The primary aim of this thesis is to put forward the important aspects of *Permissioned Blockchains*, to identify and evaluate the various transaction processing characteristics of such a network and to implement a supply chain application that demonstrates the various business transactions. The permissioned blockchain framework that is studied in this thesis is based on Hyperledger Fabric.

1.2 Scope of Work

This thesis is divided into two main parts. In the first part we discuss the various concepts in Permissioned Blockchains like *Membership Service Providers (MSPs)*, *consensus algorithms*, *chaincodes*, *ledger*, *endorsement policy*, *peers* and *channels*. We will dive into the details of all these components that are part of the Hyperledger Fabric framework. We will study the transaction processing characteristics of Hyperledger Fabric in relation to these intrinsic components. In the second part, we will create a supply chain application that includes the basic business transactions like querying a product, getting all the product details, creating a new product and updating the current owner of a product. We will also evaluate the business transaction spanning multiple channels and how they are managed in permissioned blockchain in case of failure. Here, we will study about commit, rollback, save-point and nested transactions. We will also study the immutability property of the distributed ledger in Hyperledger Fabric and understand its impact on the business transactions. For the implementation, we will create an Angular web application and an Android application. The key technologies utilized for this use case are Cascading Style Sheet (CSS), Hyper Text Markup Language (HTML), chaincodes written in Go language, JAVA, Android SDK and JavaScript.

1.3 Outline

The remaining content of the document is arranged as follows:

Chapter 2 - Fundamentals : In this chapter we will discuss the fundamentals of permissioned blockchains based on Hyperledger Fabric, difference between permissioned and permissionless blockchains on the basis of transaction processing characteristics and the types and classification of consensus algorithms and how they impact the transaction flow.

Chapter 3 - Related Work : This chapter will give an overview of the different work being carried on permissioned blockchains that impact its transaction properties.

Chapter 4 - Blockchain Transaction Characteristics : This chapter will highlight the differences in the transaction processing characteristics of permissioned blockchains by comparing it with the properties of RDBMS and permissionless blockchains.

Chapter 5 - Concept, Specification and Design of a Permissioned Blockchain Network : In this chapter we will discuss the steps required to create a permissioned blockchain network and how a blockchain network can be used with web and mobile application.

Chapter 6 - Experimental Design and Results : In this chapter we will define different experiments performed to understand the transaction characteristics of permissioned blockchains and their corresponding results.

Chapter 7 - Conclusion : This chapter will summarize the entire thesis work and will give us some future research possibilities.

2 Fundamentals

In this chapter, we will learn about permissioned blockchains explicitly focussing on Hyperledger Fabric, which is a widely used permissioned blockchain network and has a modular architectural style, channels for privacy and confidentiality, pluggable consensus algorithms and chaincodes written in Go programming language for transaction execution [ABB+18].

2.1 Introduction to Permissioned Blockchains

In a permissioned blockchain, the participants are usually known and identifiable. It is mostly used in scenarios where the participants are known to each other, but they have partial trust or no trust on each other [ABB+18]. In this situation, permissioned blockchain plays an important role to create a trust environment between the participants and also to take advantage of the benefits of blockchain. Permissioned blockchain is mostly used in business applications where there is exchange of money, assets, goods or any kind of important information. In order to validate and execute a transaction, permissioned blockchains use a *Smart Contract Layer* that modifies data in the distributed ledger [WG18a]. In addition to a *Smart Contract Layer*, there is a central authority that governs the rules for the participants to join the network and to authorize the participants to submit a transaction. There is also a *Consensus Layer* that is used by the participants of the network to reach consensus on a block being appended and to validate the transaction executed by the Smart Contract [WG18b]. With all these important concepts, permissioned blockchains prove to be the right candidate to be considered for business blockchain use cases. *Hyperledger Fabric* is one such example of such a permissioned blockchain network that is widely discussed and is used as a reference for the validation and study in this thesis [Fou18].

2.2 Hyperledger Fabric as Permissioned Blockchain Network

Hyperledger Fabric is a project under the Hyperledger group of projects hosted by The Linux Foundation [1]. It has a modular architectural design pattern where the different components are pluggable and can be switched depending upon the business requirements [ABB+18]. For e.g. depending upon the requirements of the business, the consensus algorithm can be either Crash Fault Tolerant (CFT) or Byzantine Fault Tolerant (BFT). Similarly, channels can also be included to provide a higher degree of confidentiality and privacy among a subset of participants. Compared to the *Order - Execute* architectural style followed by most permissionless blockchains, *Hyperledger*

[1] https://www.hyperledger.org/projects/fabric

Fabric follows an *Execute - Order - Validate* architectural style that yields more deterministic results [ABB+18]. The different components of *Hyperledger Fabric* and their importance is described in the coming sections.

2.2.1 Consensus

Hyperledger Fabric includes a pluggable consensus algorithm functionality that can be switched depending upon the business requirements [CV17][WG18b]. Due to the distributed architecture, it becomes extremely important to agree to a generated block based on some rules. These rules are called as the *Consensus* [Org18]. *Consensus* in *Hyperledger Fabric* guarantees the ordering of transactions by the *Ordering Service* and also validates the blocks generated by transactions [WG18b]. The important functions that *Consensus* provides are:

1. It verifies the rightness of the transactions in a block generated after the success of the endorsement and consensus policies [WG18b].
2. With the help of Consensus, the Ordering Service verifies the correct ordering of transactions in a block and hence, helps in achieving deterministic results globally [ABB+18].
3. Works in combination with the *Smart Contracts* to verify the correctness of transactions that have to be included in the block. In Hyperledger Fabric, Consensus also verifies if the proper peers have endorsed the transactions to be considered for the transaction as valid [ABB+18].

Types of Consensus Systems and their Classification

Consensus can be classified on the basis of the fault-tolerant model of the blockchain architecture, transaction execution model of the blockchain transaction flow, method of block selection by the network, the types of peers or depending upon the method of ordering of the transactions.

Classification basis	Type 1	Type 2
Method of block selection	Lottery-based	Voting-based
Type of Fault Tolerance models	Byzantine Fault Tolerant	Crash Fault Tolerant
Type of peers	Competing peers	Non-competing peers
Method of transaction ordering	Reputation systems	Ordering service
Transaction execution model	Order - Execute architecture	Execute - Order - Validate architecture

Table 2.1: Table showing classification of consensus types.

Method of Block Selection

1. **Lottery - based algorithms**: In *lottery-based* algorithms like Proof of Elapsed Time (PoET) [Com17e] and Proof of Work (PoW) [IV17], the winner of the lottery process is selected that proposes the new block that has to be appended in the blockchain [CV17]. This block is then transmitted to the rest of the network [WG18b]. The major advantage of such a model is

that it is highly scalable since only the winner of the lottery proposes the block. *Forking* is a major disadvantage in case there are multiple winners and this increases the time to achieve a final state or *finality* [WG18b].

2. **Voting - based algorithms**: In *voting-based* algorithms majority of the nodes validate the blocks or transactions and only after that it is considered for inclusion in the blockchain given that the policies are met [WG18b]. Examples of *voting-based* algorithms include Redundant Byzantine Fault Tolerance (RBFT) and Paxos [WG18b]. Since majority of the nodes agree upon the validity of the blocks, *finality* is achieved very fast. The major disadvantage of such a network is the trade-off between speed and scalability as nodes transfer messages to each other. More the number of nodes in the network, more time it will take to agree on a result [WG18b].

Type of Fault Tolerance Models

1. **BFT**: These type of consensus protocols have the ability to tolerate *Byzantine* nodes. Such models take care of reaching the consensus even in case of a malicious node in the network. Practical Byzantine Fault Tolerance (PBFT) is one such example of a BFT algorithm. In a system of n nodes, PBFT protocol can tolerate a maximum of $f < n/3$ nodes, where f is the number of faulty or malicious nodes [CV17].

2. **CFT**: In these models the main focus is on the reliable broadcast and not on the importance of the content in the broadcast [CV17]. It is also called as *atomic broadcast* which means either the broadcast is successful on all the nodes or on none [CV17]. Every node can broadcast a transaction x by calling *broadcast(x)* and this will result in an event *deliver(x)* to the application [CV17]. Example of such an algorithm is Apache Kafka [Kaf18].

Type of Peers

1. **Competing peers**: In this type of consensus protocols, the peers are always competing with each other on the next block to be appended in the blockchain. The major disadvantage is that all the peers are working on the same data and this results in considerable wastage of resources. Examples include the PoW family of algorithms.

2. **Non-competing peers**: In this type of consensus model, not all the peers have the same responsibility and the work can be divided that can save a lot of resources. Some of the peers can work on block creation, while other peers can validate or endorse the block for its correctness. Examples include PBFT.

Method of Transaction Ordering

1. **Reputation systems**: Reputation systems are used for ordering the transactions in a block. The system assigns reputation to the nodes based on certain factors like the time that each node is registered in the system, total number of successful transactions and if the node was successful in the past to detect failures. Example of such a consensus protocol include *Sumeragi* [WG18b].

Consensus Algorithms	Consensus Approach	Pros	Cons
Apache Kafka	Voting-based approach. Permissioned in nature. Only leader does ordering	Crash Fault Tolerant. Finality is achieved immediately.	In case of malicious nodes, agreement cannot be reached.
RBFT	Voting-based approach. Permissioned in nature.	Byzantine Fault Tolerant and Finality achieved immediately.	More time to reach consensus if more nodes are present.
Sumeragi	Server reputation system. Permissioned in nature.	Byzantine Fault Tolerant and Finality achieved immediately. Scales to a large amount of data	More time to reach consensus if more nodes are present.
PoET	Lottery-based approach. Permissioned in nature	Provides Byzantine fault tolerance and greater scalability	Finality is delayed due to forks.

Figure 2.1: Comparison of commonly used Consensus algorithms in Permissioned blockchains [WG18b]

.

2. **Ordering service**: In this type of models, a fixed node is always responsible for ordering the transactions in the block and this node is called as the *Ordering Service node*. The ordering service can be composed of multiple nodes but the results generated by all the nodes are deterministic and this helps to achieve total-order in the blocks generated. Examples of such type of model include PBFT algorithm [ABB+18].

Transaction Execution Model

1. **Order-Execute architecture**: This is related to the architectural style of the system to agree on a transaction or block. In this style, first the ordering of transaction takes place and then the execution of transactions by the nodes. All the peers have to execute the transaction sequentially within a block and across the block. This results in less throughput due to the sequential execution. Example includes the PoW family of algorithms [ABB+18].

2. **Execute-Order-Validate architecture**: In this model, client sends transactions to peers that are specified by the endorsement policy. The ordering service then orders the executed transactions into blocks. The committing peers then validate the output against the endorsement and consensus policies and then commit the block into the blockchain. Examples based on such model include the PBFT algorithm implemented in Hyperledger Fabric ordering service.

Consensus Properties

The consensus protocols must always satisfy two important properties among the peers: **safety** and **liveness** [WG18b].

1. **Safety**: Safety property of any consensus algorithm ensures deterministic results on all the nodes. A transaction is called as deterministic if the same operation performed across different peers returns the same output no matter the time and order of execution. Safety property ensures that the algorithm behaves like a single node system that executes and validates transactions atomically and one at a time.
2. **Liveness**: Liveness is an important property that deals with the eventual delivery of transactions. It ensures that every non-faulty peer in the system will receive the transaction at one point of time or the other given that the communication mechanism is not down and the system can synchronize in case of failures.

2.2.2 Membership Service Provider (MSP)

In a permissioned blockchain network, every participating entity is assigned a unique identity. This identity is wrapped around in an X.509 digital certificate [Fab18f]. With the help of these identities, the permissioned blockchain network decides who has the proper rights and authorization to submit a transaction or who can access the resources. An MSP defines the Root Certificate Authoritys (CAs) and intermediate CAs of a blockchain trust network. In a permissioned blockchain scenario, an organization is the trusted entity [Fab18g]. An MSP also defines the access rights for different participants of the network and the channel. For e.g. some peers in a Hyperledger Fabric network may only have read access and no write access to the ledger. This is taken care by the MSP. The MSP configuration is available on the channels as well as on the peers, orderers, clients to authenticate a member outside the context of the channel [Fab18g].

Channel and Local MSPs

MSPs can be either present on the channel configuration information and is called as *channel MSP* or it may be present locally on the peer, orderer or the client and is called as a *local MSP*. In case of peers and orderers, the MSP defines the access privileges for them whereas in case of the client the MSP allows to identify the user in the transactions on the channel or if the user has a specific role like *admin* in the system.

MSP Structure

In addition to the root or intermediate CAs, there are many other important aspects in a membership for the organization. Figure 2.2 shows the local MSP structure on a local file system [Fab18g]. The importance of the various components of an MSP can be described as follows:

- **Root CAs**: This is an important folder in the MSP file system that contains the X.509 certificates of the Root CAs that is related to the organization's MSP. From the Root CAs all other certificates for other members are derived .

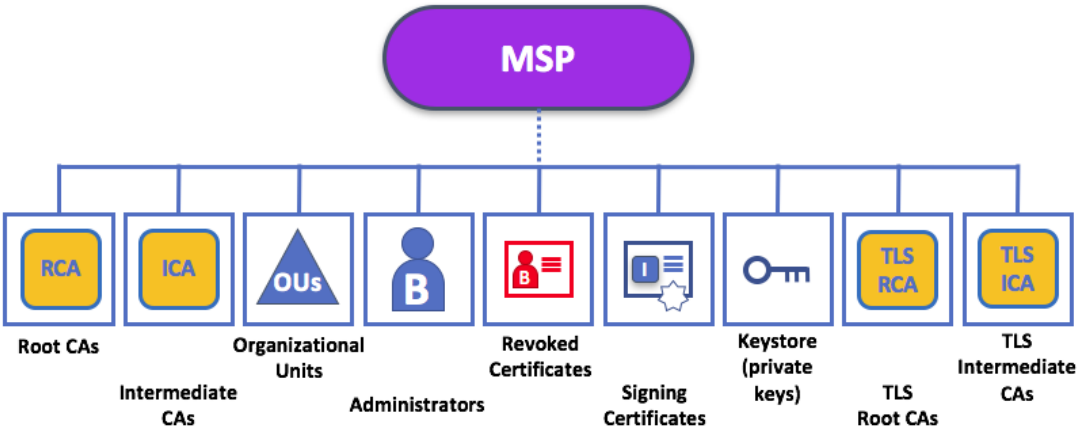

Figure 2.2: Figure showing how a local MSP is stored on a local file system [Fab18g]

.

- **Intermediate CAs**: This is a folder on the local MSP file system that contains the X.509 certificates of the intermediate CAs. The intermediate CA certificate has to be signed by at least one of the Root CAs of that organization. The intermediate CA is important when a particular organization is part of a larger blockchain network and where it has more than one subdivisions. For e.g. *Org1* can have two subdivisions as *Org1-Manufacturer* and *Org1-Supplier*. Depending upon the channel on which the organization is present, it can take a particular subdivision role and the intermediate CA for that subdivision is then used to issue certificates to the members of the organization on the channel.
- **Organizational Units (OUs)**: The OUs provide a list of the organizational units that are the members of the network and represented by the organization issuing the MSP. This is useful when the members of an organization have to be restricted to be having a particular identity signed by the designated CAs. The OUs specification is optional and if its is not specified, all the identities identified by the Root CA and Intermediate CA will become the members of the organization with the write privileges.
- **Administrators**: There is a folder for the administrators in the local MSP that defines the administrator identity for a given organization represented by the specific MSP. Generally, at least one administrator identity should be defined for an organization. The admin rights are not with the management of resources but it is determined by the specific policies used for managing the system resources. For example, the policy of the channel may define that *Org1-Manufacturer* administrator has the rights to add new organizations to a channel and the *Org1-Supplier* has no such administrator rights. The **Role** attribute is also defined in the X.509 certificate as admin. The role is strictly within the organization and not on the entire blockchain network.

- **Revoked Certificates**: The identifying information about a revoked identity in a network is held in this folder. This contains a pair of strings known as Subject Key Identifier (SKI) and Authority Access Identifier (AKI). The administrator of an organization has the responsibility to advertise the updated Certificate Revocation List (CRL) to revoke an actor of its rights in the network [Fab18g].
- **Node Identity**: This folder contains the identifying information for the nodes or peers in the blockchain network. This is used whenever a node sends a transaction proposal for endorsement, the signed endorsed transaction response and when a node wants to commit the transaction or block on the blockchain. This is a mandatory folder for the local MSPs and there should be exactly one X.509 certificate for the peer.
- **KeyStore for Private Key**: This folder contains the node's *signing key*. The *signing key* is used as an identifier for the node whenever a transaction response is signed by the node. This signature is then used by other nodes in the network to verify the authenticity of the transaction result at the endorsement phase. This is a mandatory folder for the local MSPs. Channel MSPs do not have this folder as the main aim of channel MSPs is to work on the validation and not on verification of signing by the nodes.
- **TLS Root CA**: This folder contains the X.509 certificates of the Root CAs for TLS communications. This is important in case the nodes want to communicate with each other regarding the ledger updates and when new nodes want to securely join the channel. It is mandatory to have at least one TLS Root CA X.509 certificate in the respective folder.
- **TLS Intermediate CA**: This folder contains the X.509 certificates of the Intermediate CAs for TLS communications. It is important when there is a need of commercial CAs for generating TLS certificates of an organization. This is an optional attribute and can remain empty.

2.2.3 Chaincodes

Chaincodes are programs that contain the actual business logic of the blockchain application and it is used to access the *world state* data of the blockchain. Chaincode is similar to a *Smart Contract* in Ethereum or a *Stored Procedure* in the traditional RDBMS [ABB+18]. It is a self executing piece of code that is installed and instantiated on the peers of a Hyperledger Fabric channel. The client application invokes the various chaincode functions by coupling it with a network peer. The transactions executed by the chaincode are checked for validation by the committing peers and upon complete validation the transactions results from a chaincode execution are used to update the shared ledger and change the world state. Currently, Hyperledger Fabric supports different programming languages in which chaincode can be written. These includes *Go*, *node.js* or *Java*. Chaincode has its own secure docker environment that runs loosely coupled from the corresponding peer containers [ABB+18]. The state generated by a chaincode cannot be directly accessed by a different chaincode. If the chaincodes are in the same network, the state of chaincode can be accessed with right permissions [Fab18b].

Types of Chaincodes in Hyperledger Fabric

- **System chaincode**: Transactions like the chaincode lifecycle management and Fabric policy configuration are handled by some special system transactions executed by the *System chaincodes* [WG18a]. The System chaincodes can also be used to specify the application requirements depending upon the requirements of the network and the chaincode developers can program it according to the needs of the application. The *Endorsement System Chaincode (ESCC)* is a special system chaincode that handles the endorsement of related transactions. The ESCC takes a transaction proposal and the result as input and produces a response with the results and the endorsement signature [ABB+18]. The *Validation System Chaincode (VSCC)* is responsible for checking the validity of the transaction. The input is the transaction and the output is a response indicating if the transaction is valid or not [ABB+18].
- **Application chaincode**: The applications states on the distributed ledger is maintained by the application chaincode that includes assets, data records or funds [WG18a]. The client invokes the application chaincode methods with the help of the API and submit a request for transaction. Every transaction is associated with a transaction ID that is stored in the ledger.

Chaincode Usage Scenarios

There are generally two different ways to define and develop chaincodes for the Hyperledger Fabric architecture:

- **Using chaincodes to design business contracts**: The chaincode can be used as a standalone application instance that gets invoked automatically when a condition is met or it can be programmed and accessed with the help of an API and called from the client application whenever required [WG18a].
- **Using chaincodes to manage assets in a network**: Chaincodes can be used to transfer the assets in a network between the participants. Account balances can be encoded into past transactions records and it is called as the Unspent Transaction Output (UTXO) model that is stateless. Chaincodes can also use the ledger state to store the asset value and its current holder and regularly modifying the world state as the asset moves in the network [WG18a].

2.2.4 Endorsement Policy

Endorsement policy is defined on the chaincode and it states the set of peers or nodes on a particular channel that have to execute the chaincode method and endorse the results obtained from execution for the transaction to be designated as valid. It is a way of defining which organizations have to approve of a result before finally committing it to the ledger and modifying the world state [Fab18d]. It is the responsibility of the validating peers to check for the appropriate number of endorsements for a transaction from the right sources as defined in the endorsement policy. The validating peers also check for the signatures of peers in the endorsement for authenticity and authorization.

Listing 2.1 Endorsement policy specification at the time of chaincode instantiation.

```
peer chaincode instantiate -C <channelid> -n cc -P "OR('Org1.member', 'Org2.member')"
```

Endorsement Policy Types

Generally, endorsement is defined at the time of chaincode instantiation or upgrade and it common for all the functions of the chaincode execution. However, there are some instances where a better approach for endorsement policy has to defined at the key level. Lets us see the difference between the two scenarios:

- **Chaincode level endorsement policies**: When the chaincode is instantiated on a channel, it is possible to define the endorsement policy for that chaincode on that specific channel. It is specified with the **-P** switch in the CLI command. Listing 2.1 shows the sample CLI command used to instantiate the chaincode on a channel with the endorsement policy specified with a **-P** switch. The command specifies that a chaincode with name *cc* will be deployed on the peer and channel specified by the *channelid* with an endorsement policy *'Org1.member', 'Org2.member'*. This endorsement policy states that the transaction on the channel has to be either endorsed by a member of organization *Org1* or *Org2*. Whenever a new organization is added to the network channel, the endorsement policy has to be modified [Fab18d].
- **Key-level endorsement policies**: The chaincode level endorsement policies are bound to the lifecycle of the chaincode. The key-level endorsement policies can be updated from within a chaincode function and is considered as a transaction. If there is no key-level endorsement policy present, the chaincode endorsement policies are executed in general. Even if the key-level endorsement policy is present, the transaction to create a key-level endorsement policy has to met the chaincode endorsement policy specification. If a transaction involves modification of a key containing a key-level endorsement policy, the key-level policy will override the chaincode level policies. A particular key can have multiple key-level endorsement policies and all of them have to be satisfied for key update [Fab18d].

Roles in Endorsement Policy

The endorsement policies are always specified in combination with the MSPs and the role of different nodes in the network. In general it takes the form *'MSP.ROLE'* where MSP defines the associated MSP ID in the network and ROLE represents the associated roles with the MSP in the network [Fab18d]. Some examples are:

- **'Org1.admin'**: This specifies the administrator of organization *Org1* MSP.
- **'Org2.member'**: This specifies any member of organization *Org2* MSP.
- **'Org3.client'**: This specifies any client of organization *Org3* MSP.
- **'Org4.peer'**: This specifies any peer of organization *Org4* MSP.

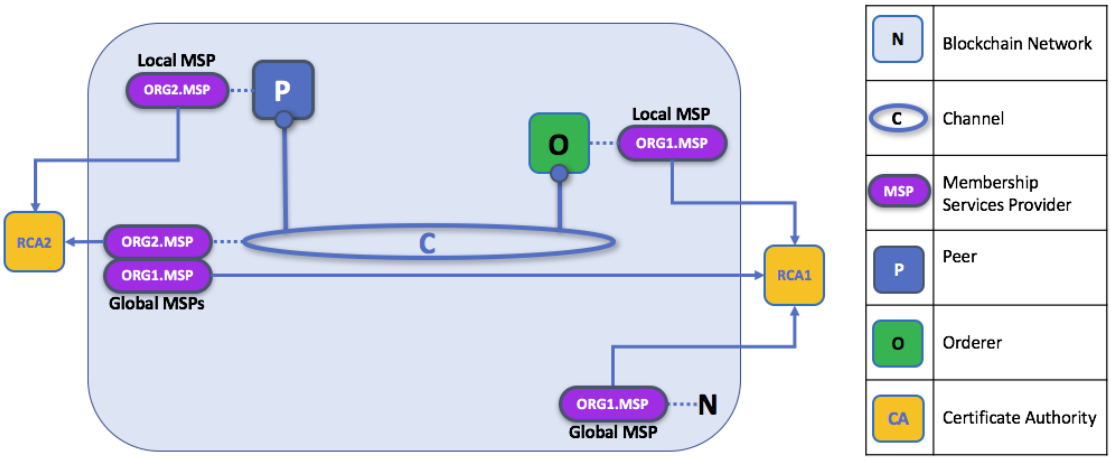

Figure 2.3: An example of a blockchain network with two organizations, their corresponding MSP, the peer, the orderer and the channel [Fab18g]

.

Syntax of the Language

There is a way of defining the policy in combination with the roles and the msp and a principal like *AND*, *OR* and *OutOf*. Some examples are:

- **AND('Org2.peer', 'Org3.peer')**: This specifies that at least one signature from both the organization's peers.
- **OR('Org2.peer', 'Org3.peer')**: This specifies that at least one signature from any one of the organization's peers.
- **OutOf(1, 'Org2.peer', 'Org3.peer')**: This specifies that at least one signature from any one of the organization's peers. Similar to *OR* policy in this case.

2.2.5 Channels

Hyperledger Fabric has the channel capabilities for a subset of participants in a blockchain network to provide some form of confidentiality and privacy. Channels are a useful feature when there are competitors involved on the same blockchain network and they do not want to share any sensitive information that might undermine their business profits. Channels are basically used to create a subset of private networks within a large business network that has some confidentiality issues with other participants. The channel identifies each peer joining the channel by the MSP of that organization to authenticate it [Fab18c].

The client SDK calls the configuration system chaincode and checks the properties like the anchor peers, members of the organizations to create a new channel. The first step in channel creation is the generation of a genesis block for the common channel ledger. The genesis block stores the channel configuration information about the channel members, anchor peers and the endorsement policies.

Listing 2.2 Different peer commands to create a channel and to join the peer to the created channel [Fab18c].

```
peer channel create -o <ordering-service url> -c <channelName> -f <channel.tx file> --cafile<
path to the peer identity file>

peer channel join -b <genesis-block.block>
```

When a new member is added to an existing channel, the genesis block containing all the required information is shared with the new member [Fab18c]. Figure 2.3 shows a typical channel created for the two organizations. The peers that are part of this channel can submit and invoke transactions on the channel ledger. Listing 2.2 shows the sample peer commands that can be used to create a channel and join the peer to the channel [Fab18c].

It is not possible to pass data from one channel to another which is secured by configuration chaincode, the gossip data dissemination protocol and the identity membership service. Channels are similar to the private access modifiers in a object oriented class providing access only to the right members of the channels. It is however important to note that chaincodes can be called from different channels but only for querying and it cannot be used for updating the ledger.

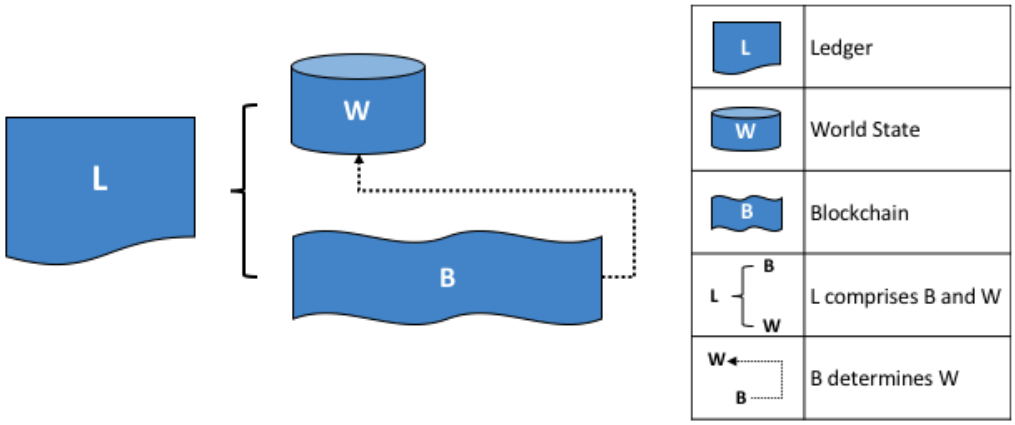

Figure 2.4: Figure showing a Ledger consisting of the world state and the blockchain [Fab18e]

.

2.2.6 Ledger

The Ledger in Hyperledger Fabric consists of two different components, the **world state** and the **blockchain**. The world state holds the current value of a key. So whenever a new transaction is invoked and if it is successful the world state gets updated with the value obtained in the transaction execution. Whenever a peer queries the chaincode for a key value, the queries are run against the world state to give the result back. The world state is stored as a key-value pair dataset. The blockchain contains the history of all the transactions and can be considered as an historical log.

The world state can be generated any time by traversing the entire blockchain. It is immutable and once transactions are appended they cannot be modified [Fab18e] [ABB+18]. Figure 2.4 shows a Ledger L comprising of a world state W and the blockchain B. The world state W can always be generated from the blockchain B. The different members of a blockchain network contain the same consistent copy of the ledger.

World State

World state contains the current value for any key and can be accessed from the chaincode without the need to traverse the entire history of the blockchain. The chaincodes can access the world state key values and modify it with the API interfaces like **GetState**, **PutState** and **DeleteState**. The state values can be either simple with just a single value like *{key=Vehicle1, value="Truck"}* or complex consisting of multiple value fields like *{key=Vehicle1, value={type:"2-wheeler", color:"red"}}*.

It is important to note that only those transactions that have the required number of endorsements as per the endorsement policy can change the world state. The state or the key-value pair are always associated with a version number. Every time a key gets updated, its version number gets incremented by 1. Initially when a ledger is created the world state is empty and subsequently gets populated when transactions are endorsed and committed in the blockchain[Fab18e].

Blockchain

The blockchain part of the ledger contains the history of transaction logs and it is considered as append only and immutable [Fab18e]. The blocks in the blockchain are interlinked with each block containing a sequence of transactions. The block's header includes a hash of the previous block header and the hash of all the transactions in the block. This hashing makes the blockchain secure

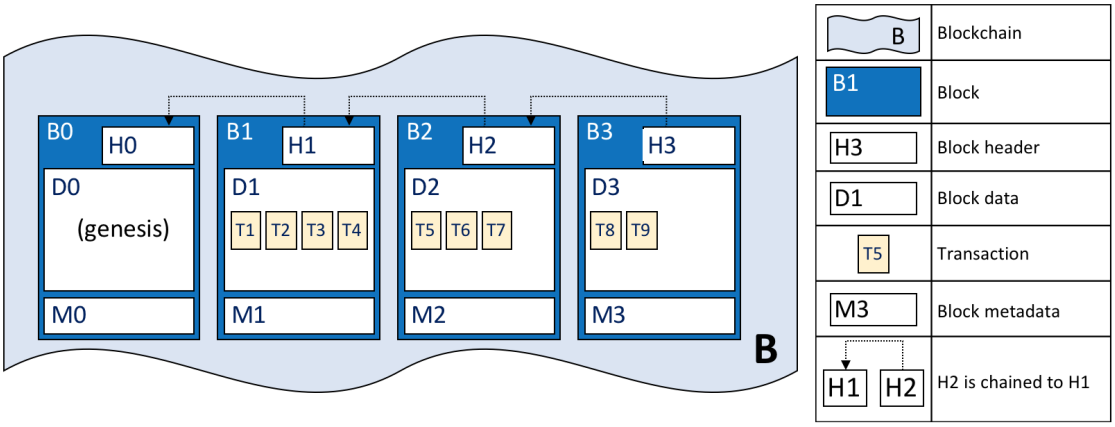

Figure 2.5: Figure showing the structure of the blockchain ledger [Fab18e]

.

and tampering can be easily detected as the blockchain itself is distributed. The world state is implemented as a database whereas the blockchain is designed as a file. Since, its append only, the file structure is much more appropriate [Fab18e].

Figure 2.5 shows the typical blockchain file structure consisting of interlinked blocks. It states that the Blockchain B is composed of blocks B0, B1, B2, B3. B0 is the genesis block in the network that contains configuration transaction and is the starting point of the blockchain. The block B3 contains block data D3 consisting of all the transactions T8 and T9. Header H3 of block B3 contains the hash of transactions T8 and T9 as well as the hash of the previous block B2's header H2.

2.2.7 Data Storage

As discussed in the previous section, the world state is implemented as a database unlike the file system of the blockchain. The world state key values can be simple or complex and Hyperledger Fabric provides two different database options for the world state : *LevelDB* [Lev18] and *CouchDB* [Cou18].

LevelDB

LevelDB is the default data store in Hyperledger Fabric and it is embedded within the peer operating system process. In other words, it is tightly coupled with the peer process and its life cycle is associated with the peer node. It is especially useful when the ledger states are simple key-value pairs [Fab18e].

CouchDB

CouchDB supports rich and complex queries and supports simple datatypes and JavaScript Object Notification (JSON) documents for the ledger states. As a result CouchDB cab be used for efficient queries that is the main requirement of a business transaction. It runs in a separate environment than the peer operating system process but has a one to one mapping with the corresponding peer process [ABB+18].

Thus with the requirements of the business, different databases can be implemented as a relational data store, temporal data store or a graph structure. Hyperledger Fabric provides pluggable data store functionality [Fab18e].

2.3 Hyperledger Sawtooth as Permissioned Blockchain Network

Hyperledger Sawtooth is a part of the Hyperledger group of projects hosted by the Linux Foundation. It is designed particularly for enterprise use due to its modular design that allows the applications to choose access control, transaction rules and consensus mechanisms in the blockchain network [Saw17].

2.3.1 Characteristics of Hyperledger Sawtooth

1. **Clear separation between the core system and application**: Sawtooth provides a smart contract abstraction layer that facilitates the applications developers to use a language of their choice to develop the smart contracts.
2. **Private groups with access control**: Separate access control can be specified for a group of nodes as the blockchain maintains stores the configuration information for access control for all participants to quickly access it.
3. **Parallel transaction execution**: With the help of a parallel scheduler, Sawtooth executes transactions parallely while still taking into consideration the transaction dependencies.
4. **Event system**: Sawtooth provides a way to subscribe to events so that the nodes can be notified.
5. **Pluggable consensus protocols**: Just like Fabric, Sawtooth also provides a pluggable mechanism to select the consensus algorithms depending upon the need of the network.
6. **Sample transaction families**: Just as Fabric has chaincodes, Sawtooth provided predefined transaction families and user-defined transaction transaction families as models. For example, the **IntegerKey** transaction family is used to test the deployed ledger for datatypes as Integer.

2.3.2 Transactions and Batches in Sawtooth

In Sawtooth, transactions that are successfully executed result in state changes. Transactions are included in **batches**. A batch is an atomic unit of work in Hyperledger Sawtooth [Saw17]. This implies that either all the transactions in a batch are committed or none at all.

Dependencies between Transactions

There are instances when a transaction execution is dependent on the successful execution of some other transactions and this results in a form of *dependency*. There is *dependency* field in the transaction proposal in Sawtooth that specifies the dependent transaction [Saw17]. This is called **explicit dependency** if it is specified by the client and allows two dependent transactions to be included in different batches but executed in right order. The parallel scheduler in Sawtooth takes care of **implicit dependencies** by interacting with the state. There is a input and output field in transaction proposal that specifies the address of the state.

Importance of Batches

Batches are the atomic unit of work in Sawtooth. If there is an invalid transaction in a batch, the batch is not applied and it is rejected. This helps to avoid the explicit dependency for transactions in the same batch as transactions are sequentially executed. External dependencies have to be specified only if the dependent transactions are in different batches [Saw17].

If there is a *cyclic dependency* between the transactions, it cannot be solved with *explicit dependency* specification at the transaction level. Suppose there are three transactions **X**, **Y**, **Z** that have to be applied in the same order. When a cyclic dependency is present between them, this cannot be defined with explicit dependencies alone. Batches solve this problem by executing the transactions in an ordered manner and rejecting the entire batch if any transaction fails. Dependencies specify the ordering of transaction execution and this fails in scenarios where there is a cyclic dependency as it cannot be specified in the transaction [Saw17].

3 Related Work

In this chapter, we will discuss the three important reference work done on blockchains related to the transaction characteristics analysis. These three are important as they highlight the differences between the permissionless and permissioned blockchain networks. In Section 3.1, the permissionless blockchains are studied for their transactional properties. In Section 3.2, Hyperledger Fabric and its underlying architecture is discussed which is used as a reference for further study in the thesis. Section 3.3 compares the different consensus algorithms present in different blockchain networks.

3.1 A Transaction Processing Perspective on Blockchains

In this keynote paper, Tai et.al. discuss the transactions characteristics of blockchains [TEK17]. A different model called **SALT** is proposed for the blockchain transactions.

3.1.1 Motivation

The RDBMS transactions focus on achieving database consistency while BASE is a model that neglects some **consistency** by favoring **availability**. With the introduction of blockchains that emphasize on distributed peer-to-peer technology and immutable ledger, another model in place of ACID and BASE has to be introduced as blockchain transactions are consensus driven by the participants.This model introduced by Tai et.al. is called as **SALT** [TEK17].

3.1.2 Study Approach

Blockchain transactions are consensus driven and have different properties compared to ACID and BASE models. Blocks contain transactions and it is important to study the blockchains from transaction perspective as well as blocks or system perspective. In this study, two real world application example of blockchain are used for comparison and evaluation of transaction properties. One use case involves Monegraph [Mon17] which is an online digital media platform to share or sell media rights to different media users. The second use case uses Provenance blockchain [Pro17] that is a supply chain traceability application providing information about a product as it flows in the supply chain.

Both the use cases were studied from a transaction and system perspective and the results were concurrent. The blockchain transactions were characterized as *Sequential*, *Agreed*, *Ledgered* and *Tamper-Resistant* from a transactions perspective. From a system perspective, these transactions were characterized as *Symmetric*, *Admin-free*, *Ledgered* and *Time-consensual* [TEK17].

However, it is also important to note that the study was done on the permissionless blockchain networks and thus, it highlights the transactional properties of permissionless blockchains.

3.2 Hyperledger Fabric: A distributed Operating System for Permissioned Blockchains

This paper [ABB+18] describes the modular architecture of Hyperledger Fabric and its various components like consensus algorithms, smart contract layer, transaction flow, peer communication, ledger, etc. With the help of this study, we get a deep understanding of permissioned blockchain network like Hyperledger Fabric.

3.2.1 Motivation

Androulaki et. al. describe the working of Fabric and its underlying architectural style and the programming setup required to build a permissioned blockchain model. It is emphasized that Fabric is the first blockchain system that does not use cryptocurrencies and still provide the advantages of distributed applications. This is quite opposite to the permissionless blockchains that require cryptocurrency for transaction execution. Fabric uses portable membership concept combined with identity management for access control. It uses the architectural model for transaction processing called as execute-order-validate model that removes non-determinism from the smart contracts. These properties make Fabric an ideal candidate for study related to the transaction processing [ABB+18].

3.2.2 Study Approach

The study in this paper highlights the difference between the order-execute architecture model used in permissionless blockchains and early permissioned blockchains and the execute-order-validate architecture model of Hyperledger Fabric. For the purpose of study, a authority minted cryptocurrency similar to that of Bitcoin [Nak] is used which is defined as *Fabcoin* in the paper [ABB+18]. The study is focussed on the customization of the validation phase and the endorsement policy. The developed model is used for benchmarking of permissioned blockchains.

3.3 Blockchain Consensus Protocols in the Wild

Cachin and Vukolić [CV17] wrote a paper on the different consensus models for blockchain that include crash tolerant and byzantine tolerant models.

3.3.1 Motivation

Blockchain transactions are based on consensus mechanism and it is essential to have an efficient consensus algorithm so that the transactions are secure and resilient. The main aim of consensus is to agree on the data and the order in which the transactions are executed. There are chances that a particular blockchain network can have adversarial nodes in the system and hence, a blockchain system should be able to tolerate Byzantine faults.

3.3.2 Study Approach

Cachin and Vukolić review the different consensus protocols in some prominent permissioned blockchain platforms and analyze their fault models and resilience against attacks. The protocol comparison covers Hyperledger Fabric, Tendermint, Symbiont, Iroha, Kadena, Chain, Ripple, Stellar, and others [CV17].

Which faults are tolerated by a protocol?	**Special-node crash**	**Any $t < n/2$ nodes crash**	**Special-node subverted**	**Any $f < n/3$ nodes subverted**
Hyperledger Fabric/Kafka	.	✓	.	–
Hyperledger Fabric/PBFT	.	✓	.	✓
Tendermint	.	✓	.	✓
Symbiont/BFT-SMaRt	.	✓	.	✓
R3 Corda/Raft	.	✓	.	–
R3 Corda/BFT-SMaRt	.	✓	.	✓
Iroha/Sumeragi (BChain)	.	✓	.	✓
Kadena/ScalableBFT	?	?	?	?
Chain/Federated Consensus	–	(✓)	–	–
Quorum/QuorumChain	–	(✓)	–	–
Quorum/Raft	.	✓	.	–
MultiChain +	.	✓	.	–
Sawtooth Lake/PoET	⊕	✓	⊕	–
Ripple	⊗	(✓)	⊗	–
Stellar/SCP	?	?	?	?
IOTA Tangle	?	?	?	?

Figure 3.1: The consensus resilience properties in different blockchain networks [CV17]

.

Figure 3.1 shows the comparison of different blockchains based on their consensus resilience properties. The symbol **'X'** denotes that the system has a resilient model against faults whereas **'-'** denotes it is not. With the symbol **'.'**, no special node is present in the protocol whereas **'?'** denotes that the properties can't be accessed [CV17].

4 Blockchain Transaction Characteristics

In this chapter, we will discuss the various transaction characteristics of permissioned blockchain networks by studying the Hyperledger Fabric framework. We will compare the transaction properties of Hyperledger Fabric with that of the permissionless blockchain networks and the RDBMS for a proper understanding. The effect of various components like *channels*, *chaincodes*, *endorsement policies* and *consensus algorithms* on the transaction properties will be studied here.

4.1 Transaction Processing Models

Initially, transactions in blockchain followed an *order-execute* architectural style in which the network orders the transactions before sending them to all the peers in the network and then the peers execute all the transactions sequentially [ABB+18]. The most important requirement of such a model is that it requires the transactions to be *deterministic*. Hyperledger Fabric follows a different architectural style for transaction processing and it is called as *execute-order-validate* model [ABB+18]. In this model, the transactions are executed first, then the transactions are ordered with the help of a consensus algorithm and at the last the transactions are validated by the peers before committing it to the blockchain. Let us discuss the details of the two models.

4.1.1 Order-Execute Model

In most blockchain networks, inclusive of permissionless and permissioned blockchains, the transaction processing model is based on the Order-Execute architectural style. In this architectural style, the network first orders the transactions with the help of a consensus algorithm and then executes the transactions in the order in which they are processed on all the nodes in the network. The different steps in an order-execute architecture are as follows:

1. The nodes that participate in the consensus mechanism collect a block that contains valid transactions. The transactions are validated by the node itself by executing and verifying the results.
2. Depending upon the type of consensus protocol, the node or peer processes the valid transaction. For example, in PoW based system, the peer will solve a PoW mathematical problem.
3. The peer transmits the block to the network if it is successful in solving the mathematical problem.
4. The other peers in the network that receive the block, validate the solution to the mathematical problem and also the transactions contained in the block. The receiving peers repeat the same execution steps as that carried out by the peer that solved the problem.

All the peers execute the transactions in a block one after the other sequentially. The order-execute architecture is shown in Figure 4.1. Ordering is done by the *ordering service*, execution is done on the peers having the required *smart contract* and the successful transactions are then used to update the state.

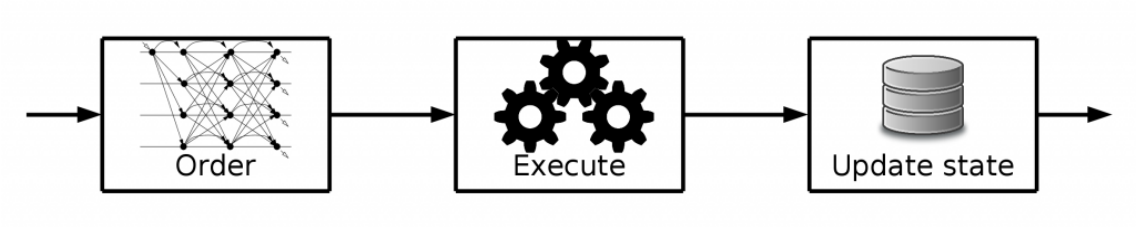

Figure 4.1: Figure showing the order-execute architectural style in blockchain network [ABB+18]
.

Drawbacks of Order-Execute Model

The order-execute is a simple architectural model but has many serious drawbacks associated with it which can be described as follows [ABB+18] :

- **Non-determinism**: Non-deterministic transactions are a major problem in the order-execute architectural style. In the order-execute model, consensus on the order of transaction is reached before the execution of the transactions. This involves that the transactions on all the peers produce deterministic results irrespective of the time of execution. If the transactions are non-deterministic, it can produce *forks* in the blockchain and it is against the basic principle of blockchain that every peer holds the same copy of data. Determinism can be achieved by using a domain-specific programming language like **Solidity** [Mas16].
- **Sequential transaction execution**: In an order-execute model, all the peers execute the transactions in a sequential manner and this limits the overall throughput of the network to process the transactions. In addition, if a smart contract has some programming faults like an infinite loop, this can stop the entire blockchain network and proceeding transactions will remain halted forever. To solve this issue, permissionless blockchains like Ethereum use cryptocurrency to process a transaction. So even if the smart contract has programming error like an infinite loop, the cryptocurrency will get exhausted at some point of time. In permissioned blockchains, this can be solved by executing the transactions in parallel. But in order to do this, transaction dependencies have to be considered before parallel transaction execution.
- **Confidentiality of execution**: Most blockchain networks run smart contracts on all the nodes in the network. But if the network requires confidentiality for the transaction, ledger state or smart contract logic, this becomes difficult to achieve without neglecting the poor system performance. The solution could be to create a network of trusted peers that execute the smart contract and propagate the results to other peers.

4.1.2 Execute-Order-Validate Model

As seen in the previous subsection, the order-execute model suffers from a number of drawbacks that limits its application in the blockchain networks. This becomes very critical when business transactions are involved that have business value only if executed in a secure and timely manner. To overcome the drawbacks in order-execute model, Hyperledger Fabric introduced the execute-order-validate model that solves all the major problems of the order-execute model [ABB+18]. Figure 4.2 shows the transaction execution in Hyperledger Fabric by the execute-order-validate model. Fabric has smart contracts called the *chaincodes* that handle the *execution phase*. The transactions after execution are ordered by an *ordering service* and this is called as the *ordering phase*. In the *validation phase*, the ordered transactions are checked for the fulfillment of the endorsement policy.

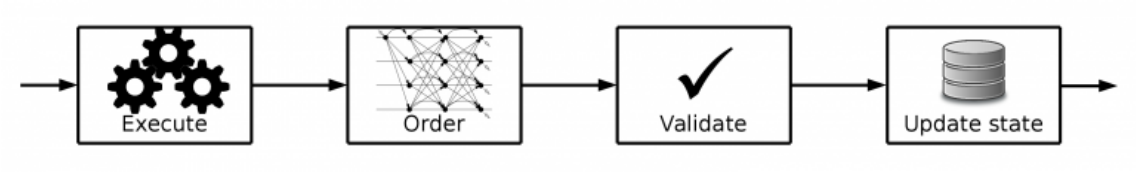

Figure 4.2: The execute-order-validate architecture model of transactions in Hyperledger Fabric [ABB+18]

.

The importance of the three steps can be described as follows:

Execution Phase

The client submits a transaction proposal by sending it to the endorsing peers in the network. The endorsing peers are specified at the time of chaincode instantiation on the channel. The transaction proposal submitted by the client contains the identifier of the chaincode, a counter or random variable, the transaction payload, parameters of the chaincode, a transaction identifier and the identity of the submitting client. The chaincode is installed for the endorsing peers to generate the transaction results. This is called as creation of a *proposal*. The proposal is generally executed against the local world state of the peer and the other peers have no knowledge about this state. The endorsers then produce a *writeset* that consists of the keys and the new values of keys and a *readset* that consists of the keys and its values read before endorsement. The signing of the message that contains the readset and the writeset is called as *endorsement* that is delivered to the client as a response.

Execution of the transaction before ordering them helps to overcome the drawback of non-deterministic chaincode. It is also possible to abort a transaction in case of Denial of Service (DoS) attacks that was not possible with the order-execute model.

Ordering Phase

The proposal response from the endorsing peers is collected by the client. As per the endorsement policy specification, the client waits till it gets the right number of valid responses and assembles a transaction and delivers it to the *ordering service* for ordering. The transaction proposal is composed of the endorsements, the chaincode methods with parameters and the metadata about the transactions [ABB+18]. Ordering is a process of atomically broadcasting [CGR11] the endorsed transactions by establishing consensus on the transactions. The ordering service does the work of putting transactions in blocks that helps to improve the throughput of the system. The ordering service supports two operations:

- *broadcast(txn)*: This method is called by the client to broadcast a transaction *txn* in the network. The transaction contains a payload and signature of the sending client.
- $B \leftarrow$*deliver(x)*: This operation is performed by the client on the orderer to retrieve a block B and sequence number as x. The block B contains a number of transactions that were included into it at the time of committing it to the ledger.

It is the responsibility of the ordering service to ensure the blocks are fully ordered. It is important to note that the ordering service in Fabric does not execute or validate a transaction and it is concerned with the ordering of transactions. This makes the consensus process in Fabric quite modular by separating the execution and validation stages with the ordering stage.

Validation Phase

This is an important step before the blocks can be finally committed to the ledger [ABB+18]. In this step the blocks are validated by the committing peers to check the content of the transactions within the block and the block itself. The steps in the validation stage are performed sequentially as follows:

1. *Evaluation of endorsement policy*: This step involves parallel evaluation of the transactions in the block. The step is performed by the Validation System Chaincode (VSCC), that is responsible for checking the endorsement policy against the chaincode. Unsatisfied endorsements for a transaction results in the transaction becoming invalid but it is still included in the block and its effect does not take place on the ledger. It is logged to maintain a history of all submitted transactions that can be audited in the future.
2. *Read-write conflict check*: This is a sequential step that is done for all the transactions in the block. In this step, the key versions in the *readset* is compared with the current version of the key in the ledger state. Non-matching versions result in invalid transactions and its effect on the ledger is restricted.
3. *Ledger update stage*: This is the last step in which the block is finally appended to the ledger and the world state is also updated with the new key values. The state changes are done by updating the key values with the *writeset* of the valid transactions. Invalid transactions are still included in the block but have no effect on the world state.

The result of the validation stage may also contain invalid transactions in the block. This is very useful in cases when the auditing is done to identify peers that submitted faulty transactions.

4.2 Transaction Flow in Hyperledger Fabric

Hyperledger Fabric follows an execute-order-validate architecture to overcome the drawbacks present in the order-execute architectural model of transaction processing [ABB+18]. Transactions in Fabric can be either *Deploy transactions* that create a new chaincode for the specified peer or *Invoke transactions* that perform a particular operation on an already installed chaincode. The execution may result in modification of world state because of *write* or just a simple query to return a key value (read).

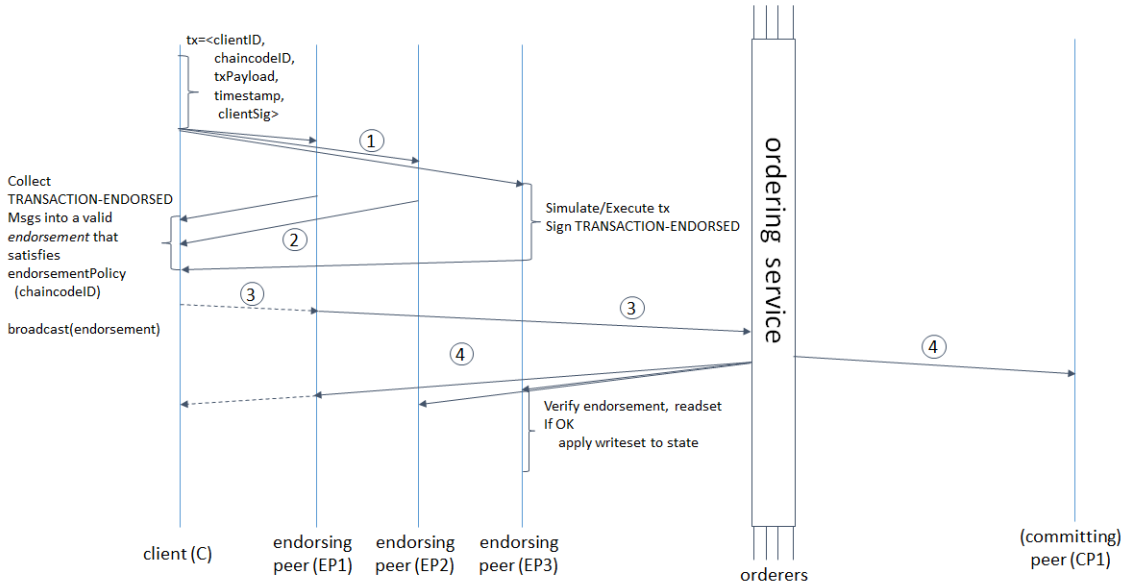

Figure 4.3: Transaction flow in Hyperledger Fabric that follows an execute-order-validate architectural style [Fab18a]

.

Figure 4.3 shows a typical transaction flow from execution, ordering to the validation of transactions in Hyperledger Fabric. The steps can be outlined as below:

1. The client submits a transaction request that is passed to all the endorsing peers in the blockchain network. The transaction proposal is composed of the client identifier, chaincode identifier, transaction payload, timestamp and the client signature.

2. The endorsing peers simulate the transactions and produce a transaction response. This process is called as *endorsement* and is part of the *execution phase*. The endorsing peers submit the proposal response to the requesting client.

3. In the execution phase, the client verifies the minimum amount of correct endorsements received on the transaction response and sends it to the ordering service to be included in the block and ultimately into the blockchain. The ordering service then orders the transactions as it receives in a sequence. This is called as *ordering phase*.

4. In the last step, the ordering service forwards the blocks containing the transaction simulation results to all the peers in the network and if needed to the client. The peers validate the blocks and the transactions in the block that is finally appended to the ledger. This is called as *validation phase*.

4.3 Hyperledger Fabric Transactional Properties from an ACID Perspective

We have already seen the transaction flow in Hyperledger Fabric. There are various aspects to consider that affect its transaction processing characteristics. In this section we will evaluate the transactional properties of Hyperledger Fabric from an ACID perspective both at the transaction level and at the system level or block level and also see ways by which the properties can be made stronger.

4.3.1 Atomicity

Atomicity is a property in which traditional RDBMS guarantees that the transaction will either be committed fully or not committed at all [MOE09]. In blockchain, atomicity can be evaluated at the individual transaction level and at the block level.

Transaction Level Atomicity

Atomicity states that a transaction is either fully committed or not committed at all. In Fabric, a transaction can either be a query or invoke on the chaincode that may modify the world state. Proper number of endorsements on the transaction execution result is required by the endorsing peers specified with the endorsement policy in order for the transaction to be committed to the blockchain. If the endorsement is met, the transaction is sent to the orderer to be included in a block. On the other hand, if a transaction does not meet the endorsement policy specification, it not sent to the orderer. If a transaction fails the *validation system check* and *multiversion concurrency check*, it is marked as invalid but it is not removed from the block and still appended to the blockchain but the world state is not updated. This is different from the permissionless blockchain networks where invalid transactions are not included in the block. Since, the state changes occur only if the transaction is valid, it can be considered as atomic. It is also important to note that if the endorsement criteria is not met, the transaction is not sent to the orderer to be included in the block and the transaction is, therefore, not committed to the blockchain making it *atomic* [ABB+18].

Block Level Atomicity

Once a transaction passes the endorsement criteria, it is sent to the ordering service to be included in a block. Blocks contain transactions that have satisfied the endorsement policy. If, at the *validation phase*, the transaction is deemed as invalid due to a version check failure or validation failure, it is not removed from the block but marked as invalid. The generated block will always be committed to the blockchain. This makes the blocks *non-atomic*.

Further work can be done to avoid committing a block in which all the transactions are invalid. This would keep the blockchain short.

Guarantee of Atomicity

There are scenarios in business transactions, where one transaction might be dependent on some other transaction. *Transaction dependencies* should be taken care of so that the transactions execute in the right order. Endorsement policy should be strong enough so that valid transactions are included in the block. In some cases it might happen that a block contains two different transactions that write to the same key in the world state. In this scenario, if the transaction model is not properly implemented only the first transaction will be valid and the second one will be deemed as invalid. Thus strong endorsement policy and *multiversion concurrency checks* and *validation system checks* are required to provide high degree of atomicity.

4.3.2 Consistency

Consistency states that the total asset value before and after a transaction in a system is always the same and it satisfies all the constraints [MOE09]. In blockchains, consistency can be either *weak*, *eventual* or *strong* depending upon the peer behaviour after transaction execution. As every peer holds the same copy of the ledger, but the update might take time due to geographical location of the peer or due to unavailability, consistency in blockchain can be considered as eventual consistency.

Steps to Improve Transaction Consistency

- **Removal of non-deterministic behaviour**: Non-determinism in blockchain results in different transaction results on different peers for the same transaction [ABB+18]. This is against the basic rule of distributed applications where the same copy is maintained on all the peers. The non-determinism in blockchain can be removed as follows :
    1. Using execute-order-validate model removes inconsistent states in the blockchain prior to ordering them in the block and provides high throughput by executing them in parallel.
    2. Using a deterministic execution model like Ethereum [WOO16] and use of a deterministic language like Solidity that avoids non-determinism in smart contracts.
    3. Parameters that affect the determinism of smart contract execution should always be passed from the client. For example, timestamp should not be generated from the smart contract but passed from the client for every peer to get the same timestamp value.
    4. External API calls should be avoided which may result in non-deterministic execution results [WG18a].
- **Avoid Forking**: *Forking* in blockchain creates two different chains and leads to inconsistent state in the blockchain. This is usually solved by taking the longest chain as the legitimate blockchain. Using execute-order-validate model can remove inconsistent states early, thereby avoiding forking. Also avoiding *lottery-based* consensus algorithms can help in avoiding forking in the chain [WG18b].

4.3.3 Isolation

Isolation ensures that multiple transactions can take place parallely without resulting in an inconsistent state in the database. This is equivalent to the transaction result execution in sequential manner with the advantages of concurrency control. Isolation guarantees that the changes made by a transaction are only visible only after its written to the main memory [MOE09].

Steps to Increase Transaction Isolation

Transaction dependencies should be taken care to guarantee strong Isolation property. Dependency is created when the execution result of one transaction is dependent on the execution of another transaction. This dependency can be handled by the smart contract as they handle the transaction execution logic. Dependencies can be either **explicit** or **implicit** [WG18a].

Implicit dependencies has to be taken care of by the smart contract and it is very hard to achieve. This can be implemented by using *mempool* of transactions and continuously applying different transactions till the required result is obtained. With **explicit** dependency, the client application or the user has to specify the order of transactions. In simple words, the user has to wait till one transaction finishes, fetch the result from that transaction and use those results to simulate the dependent transaction.

Similarly, **dependency graph** also affect the order of transaction execution in a block. The dependency graph can be of two types, **cyclic** or **acyclic** [WG18a]. Consider a cyclic dependency between three transactions X, Y and Z such that X depends on Y, Y depends on Z and Z depends on X. In this cyclic dependency, all the three transactions must be in the same block and design should be done in such a way that *multiversion concurrency check* should allow execution of all the transactions.

With the acyclic dependency, the transactions can be included in different blocks but care has to be taken about the sequence in which transactions are committed. Explicit references can also allow for parallel execution of non-dependent transactions included in the same block.

4.3.4 Durability

Durability is the property by which the transaction results are kept persistent on the system even if there is a crash or failure. This is achieved by saving the data on a physical memory on the system. In blockchain, the ledger is composed of the immutable blockchain and the constantly updated world state consisting of the key value store with increasing version number [Fab18e].

The blockchain in the ledger is a file on the peer system and every peer maintains a copy of the blockchain. Since, it is an append only data structure, it makes sense to use files for the blockchain. It consists of blocks that are linked to each other. It is important to note that since the transactions are validated before committing, there is no guarantee that any successfully executed transaction will change the world state but it will be written on the blockchain. This ensures *strong durability* for transactions written to the blockchain.

The world state on the other hand consists of keys and their corresponding latest values that are updated regularly. The chaincode query is run against the world state. The world state can have different implementations ranging from a relational database model, NoSQL database or a JSON document. It is important to note here that a successfully executed transaction might fail the *validity checks* and the world state will not be updated.

4.4 Permissionless Blockchain Transaction Properties

We have already seen the transactional properties of permissioned blockchains in the previous sections. Permissionless blockchains like Ethereum have been studied by comparing them to traditional RDBMS's ACID model and with the cloud systems and NoSQL data store's BASE model. Permissionless blockchains can be characterized as *Sequential*, *Agreed*, *Ledgered*, and *Tamper-resistant* from the transaction perspective, and as *Symmetric*, *Admin-free*, *Ledgered*, and *Time-consensual* from the transaction processing systems perspective [TEK17]. The focus of blockchain transactions is mainly on being trustless at the cost of scalability whereas ACID favors consistency at the cost of availability [Tai17]. The aforementioned perspectives on blockchain transactions can be described as in the following subsections.

4.4.1 Transaction View for Blockchain Transactions

As mentioned before, permissionless blockchain transactions can be described as *Sequential*, *Agreed*, *Ledgered* and *Tamper-resistant*. Let us discuss these properties in detail in the following paragraphs.

Sequential

The transactions in a blockchain network are executed sequentially. ACID transactions have the important characteristics of parallel transaction execution that increase the throughput of the system. However, in permissionless blockchains the transactions are not processed parallely and are only executed after one transaction finishes execution.

Agreed

In ACID transactions, there is only one central authority that decides what data has to be added to the network. In permissionless blockchains, the transactions are added to the blocks and finally to the ledger after a consensus is achieved on the given transaction. The consensus can be reached in a variety of ways depending upon the type of consensus algorithm chosen and therefore, it is a distributed agreed transaction model.

Ledgered

The successful and consensual transactions are added to the blockchain and they cannot be tampered or changed. All the transactions are recorded in the blockchain and it is append only ledger that does not allow modifying a transaction once it is committed to the blockchain. It is also important to note that the *Ledgered* property is similar to the durability aspect of ACID transactions.

Tamper-resistant

The blockchain is immutable and any effort to change the ledger can make the nodes aware of the tampering and it will ultimately be discarded. Transactions are signed by the identity of the nodes and this helps the system to identify the right user who can submit a transaction. Furthermore, the blockchain is distributed among the network participants that work with consensus. Even if a node is compromised, other nodes in the system will discard the compromised data of the malicious or attacked node.

4.4.2 System View for Blockchain Transactions

As mentioned before the system perspective of permissionless blockchain transactions classifies it as *Symmetric*, *Admin-free*, *Ledgered*, and *Time-consensual* [TEK17]. Let us discuss the importance and meaning of these terms.

Symmetric

The peers in permissionless blockchains symmetrically perform their responsibilities stating that every peer have the same processing tasks. This includes functions like data storage on the peer system, the transaction processing by signing the transactions and verifying the transactions and transmission of information.

Admin-free

As blockchain is a peer-to-peer system and there is no administrator that has the responsibilities like network maintenance, access privileges or infrastructure management. These responsibilities are performed by all the nodes in the system in a consensual manner that helps to create a trust environment.

Ledgered

The nodes in a blockchain network maintain the ledger that is kept consistent on all the nodes with the help of the consensus property of blockchain. It is an append only transaction log that requires validation from all the nodes before adding a transaction in the logs. Blocks are a type of data structure that group transactions together. Consensus is generally reached on the blocks to reduce the number of consensus rounds and this block is then appended to the blockchain.

Time-consensual

The peers in a blockchain network may be geographically separated and hence, there is a high possibility that the blocks reach different peers at different times. To solve this, the consensus algorithms defines an average time between which a new block can be created. This time is termed as *block interval*. A transaction is included into the ledger within half the block interval time but this is not always true.

4.5 Differences between Permissioned and Permissionless Blockchains based on Transactional Properties

The differences in the transactional properties of the two blockchain systems can be evaluated at the transaction level and at the block or system level. The main concept in both the blockchain networks is to have a peer-to-peer transaction processing system and having a distributed ledger that is secure and tamper-proof. Figure 4.4 shows a differences table from a transaction perspective. Permissionless blockchains can be considered as *Sequential*, *Agreed*, *Ledgered* and *Tamper-Resistant* at the transaction level [TEK17]. On the other hand, permissioned blockchains can be considered as *Sequential and Conditionally Parallel*, *Agreed and endorsed*, *Ledgered* and *Tamper-proof*. At

Permissionless Blockchains	Permissioned Blockchains
Sequential transaction execution	Sequential execution of dependent transactions and parallel execution of non-dependent transactions
Agreement by consensus	Agreement by consensus monitored by endorsement policy
Only valid transactions are appended in the blockchain	Both valid and invalid transactions are appended in the blockchain.
Transactions cannot be altered and are tamper-resistant	Transactions cannot be changed without changing all the blocks in the blockchain

Figure 4.4: Differences in transactional properties of permissioned and permissionless blockchains at the transaction level.

.

the block level, the transaction processing is a bit different. Figure 4.5 shows the differences in transactional properties at the block level for the permissionless and permissioned networks. In permissionless networks, the transaction at the block level can be considered as *Symmetric*, *Admin-free*, *Ledgered* and *Time-consensual* [TEK17]. On the other hand, for permissioned blockchains [1], it can be considered as *Asymmetric* due to different roles associated with an identity and access

[1] the reference is Hyperledger Fabric for the permissioned blockchains

control, *Admin-controlled* due to the presence of MSPs in the network for resource access, *Ledgered* as both valid and invalid transactions in a block are committed by the peer and *Time-consensual* by defining a time-to-cut transaction by the Ordering Service Nodes (OSNs) in the network that creates deterministic blocks on all the nodes. This timer starts when a transaction is included in a block and when the timer expires and a block is still not cut, a time-to-cut transaction is broadcast to finish the block creation [ABB+18].

Permissionless Blockchains	Permissioned Blockchains
Symmetric transaction processing, state storage and information broadcast	*Asymmetric* as some nodes might not have the permission to perform certain transaction
Admin-free as there is no central authority for infrastructure management and access control	*Admin-control* due to presence of MSP for access control in the network
Ledgered as the agreed on block is appended to the ledger after consensus	*Ledgered* as both the valid and invalid transactions are included in the block
Time-consensual as there is defined time interval between creation of blocks	*Time-consensual* by defining a time-to-cut transaction by the Ordering Service Nodes (OSNs)

Figure 4.5: Differences in transactional properties of permissioned and permissionless blockchains at the block level.

.

4.6 Effect of Consensus Algorithms on Transactional Properties

Consensus is a process by which the nodes in a blockchain network agree on the order and content of data. The peers in the blockchain network execute a *consensus protocol* to create blocks from transactions, to validate the transactions and building a hash chain over the created blocks [ABB+18]. Therefore consensus is a necessary requirement to achieve **consistency** in the blockchain networks.

The *order-execute* architecture with the consensus implementation can lead to inconsistent state due to non-deterministic results. In Hyperledger Fabric, the *ordering phase* involves the use of consensus protocols for ordering the transactions in blocks. Consensus also affects the **atomicity** of transactions specifying either a transaction execution result is valid or not. Only after the peers reach consensus on a transaction or block, it is considered valid and appended to the ledger by updating the world state. A strong consensus protocol can guarantee **atomicity** of transactions.

4.7 Effect of Channels on Transactional Properties

The concept of channels in Hyperldger Fabric provides confidentiality and privacy to the peers of the channel [Fab18c]. However, since a business transaction can span different channels and be composed of many *nested transactions* that are scattered over different channels, it is important to consider the impact of channels on transaction execution.

Having a channel is like creating a private blockchain network within a larger blockchain network. It can also be said that a blockchain network in Hyperledger Fabric is composed of several smaller channels. A peer can be on more than one channel but it maintains a separate ledger for the different channels. For example, if a peer P is present on channel A and channel B, it will have two different ledgers $L1$ for channel A and $L2$ for channel B.

As the main purpose of channels is to isolate data, it becomes problematic when a large business transaction involves sub-transactions from different channels. Consider that a chaincode call on one channel requires data from a chaincode call on a different channel (same chaincodes are installed on both the channels but their states are different). In this scenario, if the first transaction fails, the second transaction will not fail automatically and will require a layer of implementation on the top that handles this logic. Without proper implementation, the large transaction will be considered **non-atomic** and may result in **inconsistent state**. There are different ways for achieving transaction **atomicity** and **consistency** in channels discussed as follows :

1. Use channels only when the data is not going to be used in different channels at all.
2. Use of a single channel and private data collection to provide privacy for only the data that is sensitive and cannot be shared with other peers on the channel.
3. Use of single channel with data encryption.
4. Use of multiple channels with an intermediate implementation of a software layer like a 2-PC transaction monitor on the application side that has access to all the channel's data and manages the dependent transactions. This layer has to take care of **rollback**, **compensation action** and **commit** for all the transactions. However, this type of implementation will make the network slow.

4.8 Effect of Chaincodes on Transactional Properties

Chaincodes contain the application logic for the entire blockchain network, and for a peer to access the world state and ledger the chaincode has to be installed on the corresponding peer [Fab18b]. A business transaction may span different chaincodes on same channel or on different channels. Proper transaction management has to be taken care of in such a scenario.

Chaincodes affect **consistency**, **isolation** and **atomicity** properties of a transaction and requires special effort on the side of the application developer to program the chaincodes in such a way that these properties can be guaranteed. A chaincode operation can be either a **read** or a **write**. This is also impacted by the introduction of a number of channels. Let us consider the different scenarios involving chaincode to chaincode call:

1. **Same channel read operation**: It is possible to call a chaincode from another chaincode on the same channel to read the world state. This will be included in a single transaction and will respect the atomicity guarantee.
2. **Different channel read operation**: Reads from a chaincode on a different channel are also possible. However, there is no MVCC check of the read value and the caller has to maintain the state in its own transaction context. If the caller has no access to the callee channel, the read will fail.
3. **Same channel write operation**: The called chaincode has its own individual read write sets within the caller's transaction context. Therefor, committing this transaction will involve multiple chaincode's read-write sets on a single MVCC check.If there exists any key-version check failure from any involved chaincodes, the whole transaction will fail. This provides **atomicity** and **consistency** to the transactions [Ngu17].
4. **Different channel write operation**: The same transaction context cannot be used when different channels are involved. There are different ledgers involved with different key sets. In order to atomically co-ordinate the sub-transactions, the caller has to create an intermediate implementation like a 2-PC transaction monitor that handles all the sub-transactions in a single transaction context [Ngu17].

4.9 Effect of Endorsement Policy on Transactional Properties

Endorsement policies are defined at the time of instantiation of chaincode on a channel. A strong endorsement policy will guarantee **consistency** and **atomicity** of transactions. For example, if there are two organizations *Org1* and *Org2*, endorsement policy should be such that it enforces agreement on execution results from both the organizations. This can be achieved by using the policy *AND(Org1.member, Org2.member)*. This will force the client to collect endorsements from both the organizations. The endorsement will be valid if and only if the peers sign the same bytes which will be different if the transaction results are different and will be easily caught. Thus a strong endorsement policy guarantees **consistency** and **atomicity** in the network.

4.10 Other Miscellaneous Transaction Terms

In this section we will see some of the RDBMS transaction concepts in relation to the permissioned blockchain transactions and how they can be implemented.

4.10.1 Peer Transaction Manager

The traditional RDBMS have a *resource manager* that is responsible for affecting the state changes in the databases to maintain data consistency. In Hyperledger Fabric, the world state is maintained and managed by the Peer Transaction Manager (PTM) [ABB+18]. The PTM has a key-value store to manage the latest state in the form as (*key, value, ver*) for every unique key of the chaincode

state. The *ver* in the store provides information about the sequence number of the block and also the transaction sequence number in the block. The version is unique and increases every time there is a state change.

When a transaction is executed, its *readset* in the form (key, ver) and writeset in the form (key, value) is maintained by the PTM. The PTM also supports rich range queries by computing a hash of the query execution results and then adding the query string and hash in the *readset*.

The PTM also perform the validation phase to ensure consistency of data in the ledger. The validation of transactions is done sequentially in a block by comparing the version in the readset for a key to the latest version present in the blockchain. If the version differs, the PTM aborts the update of world state by marking the transaction as invalid. To avoid phantom reads for range queries, PTM executes the query again and compares the new hash with the one present in readset. In case of successful transactions, the PTM updates the latest state with the *writeset* that it has. Thus, PTM has the responsibility to maintain the integrity and consistency of transactions on all the peers respectively.

4.10.2 Savepoint

Savepoint is a way of defining labels or markers in a transaction so that if a failure occur, the transaction can be resumed from these labels [Ora17]. It is a way of dividing a large complex transaction into smaller manageable transactions. In Hyperledger Fabric Ledger, the PTM is responsible for managing the *savepoint* for the blockchain [ABB+18]. After the PTM validates the transactions and applies the state changes, it calculates the savepoint value that defines the largest successfully committed block number in the ledger. This savepoint is very useful in times of crashes when the world state and the indices have to be recovered from the persisted blocks in the ledger.

4.10.3 Rollback Transactions in Hyperledger Fabric

Rollback [Micb] is the process by which the transaction effects are undone in the context of transaction execution. As stated before, the blockchain is an append only data structure and the world state contains the latest value for a key. It is always possible to generate a world state from the blockchain with the help of *savepoint* in case of crashes and with the help of mutual consensus in case the world state has to be regenerated for all the peers. However, rollback will not decrease the version number for a key in the world state and it will only increase with the rollback transaction.

4.10.4 Compensating Transactions

In RDBMS, a compensating transaction [Mica] is used when the transaction results are committed and have to be undone with a new transaction that overwrites it to the previous value or to some predefined required value. As discussed before, the ledger contains an append-only blockchain data structure and to create a compensation action a new transaction has to be submitted. This transaction will also be recorded in the blockchain and the world state can be updated with the required value.

5 Concept, Specification and Design of a Permissioned Blockchain Network

In this chapter, we will create a blockchain network that works with a web application and an Android application. We will see the creation of different artifacts like the chaincodes, channels, endorsement policy, ledger, peer and client web application and android application that work with this blockchain network. Some of the experiments in Chapter 6 are based on the extension of the application discussed in this chapter.

5.1 System Overview

We will create a simple supply chain application using Hyperledger Fabric that will help us to understand the transaction processing functions in permissioned blockchains. As a product passes hands through different parties in a supply chain, traceability of product regarding its authenticity and quality becomes troublesome and time-consuming. With the help of a blockchain application, all the interested parties are included in a single network and every participant maintains a copy of data. As a result, traceability becomes fast and it also provides transparency in the whole network. Moreover, it provides a trust environment for business transactions that was hitherto managed by third parties like banks.

The supply chain application can be used to trace any product from source to destination. For simplicity, we will not categorize the products and consider it as a general food product. The products, however, can be identified through their product id as we will show in our Android application. The supply chain involves participants like **supplier**, **producer**, **buyer** and **consumer** that we will depict in the application. The functions on a product will include *query a product with its id*, *query all the products*, *update a product with the new owner* and *create a new owner*. The different functions will be based upon the type of user permissions granted for the participant. For example, a supplier will have the permission to update the owner, a producer will have the permission to view all products, create a new product, view a product by its id, a buyer will have query permissions, update privileges whereas a consumer will have the permission to query a product with its id to trace the product.

Initially, we will create a single channel for the supply chain and for the experimentation we will further extend it by creating multiple channels and simulating a 2-PC transaction monitor. The Android application is useful for the **consumer** who can scan a product's QR code and get the information about the product.

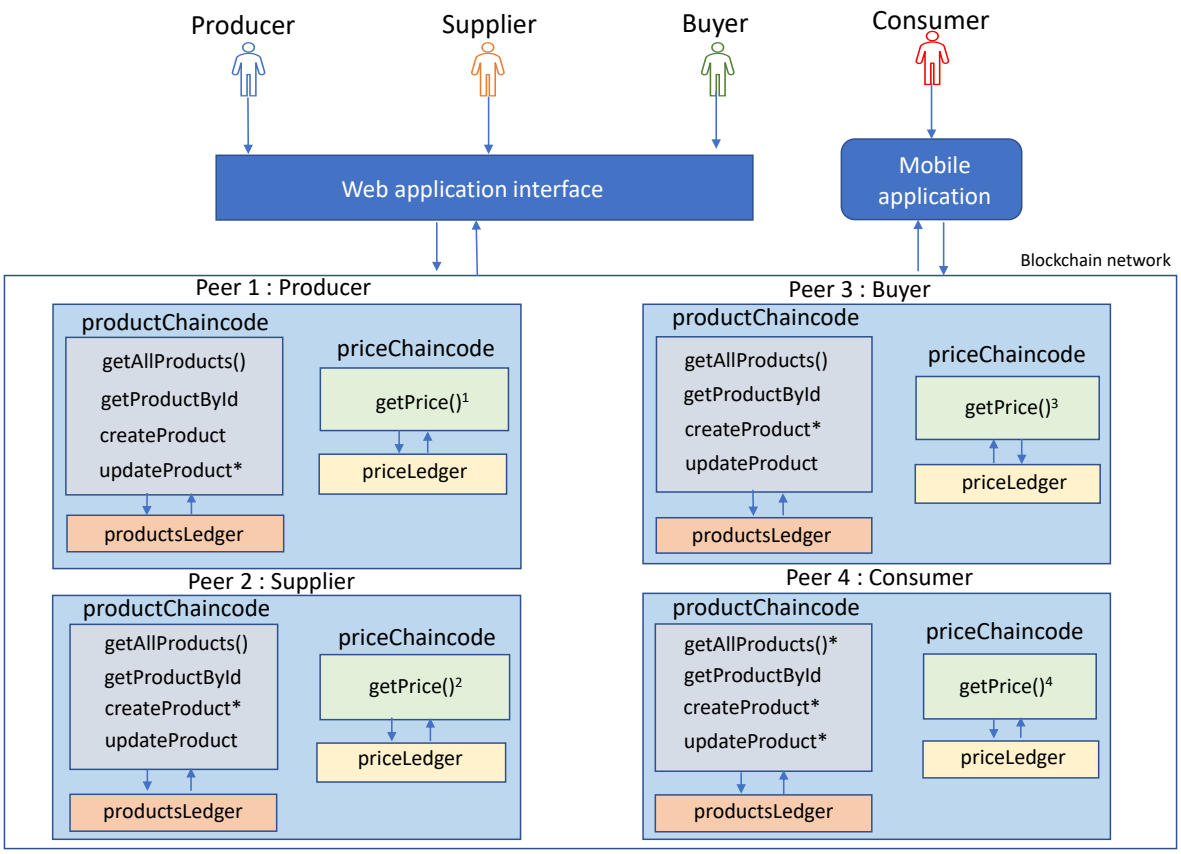

Figure 5.1: The architecture of the developed application. Methods marked with (*) will not be accessible to the corresponding peers. The method $getPrice()^1$ will fetch price for channel between Producer and Supplier, $getPrice()^3$ will fetch price for channel betwwen Supplier and Buyer, $getPrice()^2$ will fetch prices for both the channels, $getPrice()^4$ will fetch price for channel between buyer and consumer

.

5.2 System Architecture

The participants of a blockchain network will be accessing the blockchain network from a web application interface and a mobile application. As shown in Figure 5.1, the different peers are associated with different participants of the network. This ensures that only the authorized peers can access the functions of the chaincode and can submit transactions on the blockchain network. Initially for the application, we created a single channel where all the participants are included in the channel and the use case is further extended with the experimentation in the next chapter where we use multiple channels.

Depending upon the login credentials, the user identity is mapped to the login details and the peer and the view related to the user's permissions is shown on the screen. For example, the user **producer** will have access to three functions that include *getting all the products details*, *get product detail by id* and *create a new product*. The view for **producer** will only show these functions and it

will be handled programatically. Additionally, there is an android application created that can be used to *get a product by id* or *get a product by scanning QR-Code*. Also the app can be used by different users like **producer**, **supplier**, **buyer** or **consumer** to perform the associated functions.

As it is seen in Figure 5.1, the peer for the **producer** will have two chaincodes installed on it. The **productChaincode** contains all the functions related to the products like querying, updating and creating a new product. However, the producer has no access to the **updateProduct** method of the chaincode. Similarly, supplier and buyer has no access to **createProduct**, consumer has no access to **getAllProducts**, **createProduct** and **updateProduct**. The chaincodes maintain their own state and the **priceChaincode** will have its own state containing price information. Both the chaincodes maintain their separate ledger that is distributed over the network.

5.3 User Roles

The blockchain application will have four different type of participants with each participant having its own functions in the blockchain application. Figure 5.2 shows the different actors in the developed blockchain application. The description of the participants is as follows:

Figure 5.2: Figure showing different participants in the blockchain application.

- **Producer**: The producer of product can be any manufacturer that creates products into finished goods or as raw products. For example, a farmer who produces wheat in the field or a company that manufactures biscuits. The different functions associated with the producer are as follows:
    1. **getAllProducts()**: Fetches the details about all the products.
    2. **getProductById()**: Fetches the details of a specific product identified by its id.
    3. **createProduct()**: Creates a new product with all the necessary and required fields.
- **Supplier**: The supplier is the link between the producer and buyer and plays an important role in the supply chain application. The supplier can be an organization or a company that delivers raw or finished goods to a buyer. The different functions associated with the supplier are as follows:
    1. **getAllProducts()**: Fetches the details about all the products.
    2. **getProductById()**: Fetches the details of a specific product identified by its id.
    3. **updateProduct()**: Updates the owner of the product as it moves in the supply chain.

- **Buyer**: A buyer is the one who gets the product produced by the producer and can further process it or sell it the way its delivered. The buyer can be a supermarket store, vehicle manufacturer or other such manufacturing units. The different functions associated with the buyer are as follows:
  1. **getAllProducts()**: Fetches the details about all the products.
  2. **getProductById()**: Fetches the details of a specific product identified by its id.
  3. **updateProduct()**: Updates the owner of the product as it moves in the supply chain.
- **Consumer**: The consumer is the last point in the supply chain and is the one who enjoys the services of the final finished good after purchasing it from the manufacturer or from a supermarket. The different functions associated with the consumer are as follows:
  1. **getProductById()**: Fetches the details about a particular product specified with the Id.

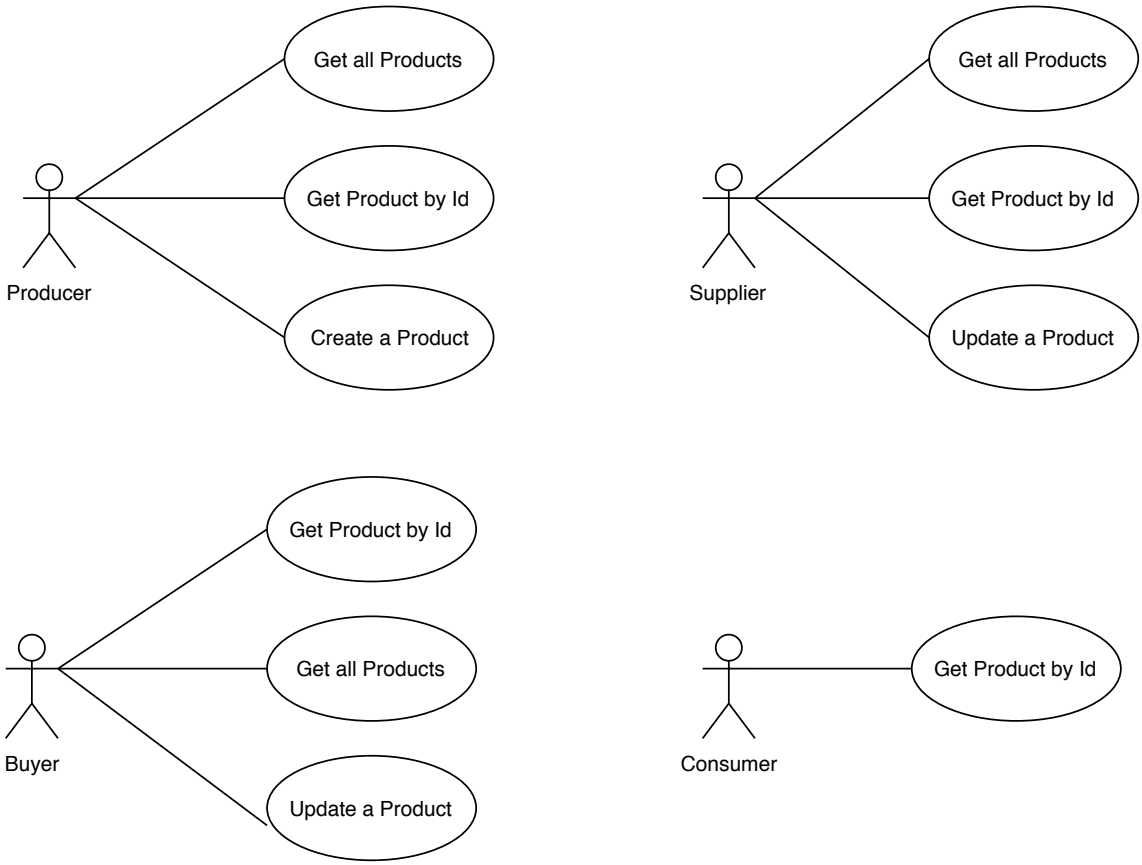

Figure 5.3: Use case diagram for the different participants. The first figure shows the use case diagram for producer, second figure shows the use case diagram for supplier, third figure for the buyer and fourth figure for the consumer.

5.4 Use Cases

As discussed before, there are four different participants in the network where each participant has a specific set of functions. The use case diagram helps to understand the different functions a participant can perform and the constraints and possible outcomes are also discussed. Figure 5.3 shows the different participants in the network and their functions with the help of a use-case diagram.

The details about the different use-cases is described further:

Name	**Get all Products**
Goal	Get all the details about all the products on the blockchain network.
Actor	Producer, Supplier, Buyer
Pre-condition	The logged user should have proper authorization to perform the function.
Post-condition	The details about the product like owner, location, datetime and product Id are shown in a table.
Post-condition in Special Case	The user is not authorized to perform the action and hence, no results are shown.
Normal Case	• The user clicks on the button that fetches all the products. • A chaincode query is executed against the world state on the local peer and response is returned in a json format. • The response from the blockchain is handled by the front-end application that parses the data and shows it in a tabular format.
Special Cases	Request to get the product details failed because the user now is no longer a part of the network or its membership has been revoked.

Table 5.1: Description for the use case *Get all Products* for the participants *Producer*, *Supplier*, *Buyer*.

Name	**Get Product by Id**
Goal	Get all the details about a product by querying it with its Id on the blockchain network.
Actor	Producer, Supplier, Buyer, Consumer
Pre-condition	The logged user should have proper authorization to perform the function.
Post-condition	The details about the product, specified with the Id, like owner, location, datetime and product Id are shown in a table.
Post-condition in Special Case	• The user is not authorized to perform the action and hence, no results are shown. • The product with the specified Id does not exist in the blockchain network.
Normal Case	• The user enter the Product Id and clicks on the button that the details about a specific product. • A chaincode query is executed against the world state on the local peer and response is returned in a json format. • The response from the blockchain is handled by the front-end application that parses the data and shows it in a tabular format.
Special Cases	• Request to get the product details failed because the user now is no longer a part of the network or its membership has been revoked. • No response returned as the product with the specified Id is not there in the network.

Table 5.2: Description for the use case *Get Product by Id* for the participant *Producer*, *Buyer*, *Supplier*, *Consumer*.

Name	**Create a Product**
Goal	Create a new product in the blockchain network by providing all the details.
Actor	Producer
Pre-condition	The logged user should have proper authorization to perform the function.
Post-condition	A new product with the supplied values for owner, location, datetime and product Id fields is created.
Post-condition in Special Case	The user is not authorized to perform the action and hence, transaction fails.
Normal Case	• The user enter product Id, owner, location, datetime for the product. • A chaincode invoke transaction is executed on the peer and the results after validation are submitted to all the peers and transaction is committed. • The response from the blockchain contains the transaction Id of the transaction and it is displayed on the web and mobile application.
Special Cases	• Request to get the product details failed because the user now is no longer a part of the network or its membership has been revoked. • The ordering service is down and as a result the transaction cannot be put in blocks and cannot be committed. Transactions failure notification is shown in the application.

Table 5.3: Description for the use case *Create a Product* for the participant *Producer*.

Name	**Update a Product**
Goal	Update the **Owner** field of the product as it flows through the supply chain.
Actor	Supplier, Buyer
Pre-condition	The logged user should have proper authorization to perform the function.
Post-condition	The product is updated with the new owner and result can be verified by quering.
Post-condition in Special Case	• The user is not authorized to perform the action and hence, transaction fails. • The Product with specified Id does not exist resulting in error.
Normal Case	• The user enter the Product Id and owner for which the update is required. • A chaincode invoke transaction is executed on the peer and the transaction is sent to the ordering service. The ordering service puts the transaction in a block and broadcasts it to all peers that eventually commit it. • The response contains the transaction Id of the invoke transaction and acknowledgement about transaction status that is shown in the application.
Special Cases	• Request to get the product details failed because the user now is no longer a part of the network or its membership has been revoked. • The ordering service is down and as a result the transaction cannot be put in blocks and cannot be committed. Transactions failure notification is shown in the application.

Table 5.4: Description for the use case *Update a Product* for the participant *Buyer*, *Supplier*.

5.5 Data Model

The data model defines the schema of the data that is going to be used in the application. The product has information like **ProductId**, **Location**, **Owner**, **Datetime** which is maintained in a separate ledger called **productLedger** on every peer. Similarly, the Price schema defines the information required to get a product's price. It has fields like **ProductId** and **Price**. The Price schema is maintained by the priceLedger on every peer. Figure 5.4 shows the data model for the application developed. The importance of the various properties can be explained as:

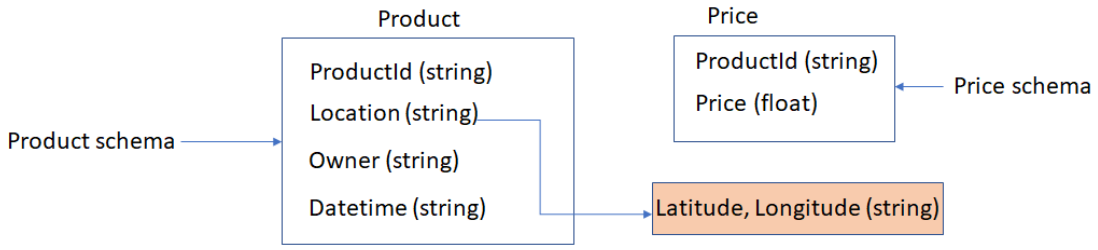

Figure 5.4: Figure showing the schema of product and price that are maintained in two different ledgers.

1. **ProductId**: Every product is unique and therefore, ProductId is the primary key for the product that can be used to query a product.
2. **Location**: The Location property defines the current location of the product in the supply chain. The location field is composed of latitude and longitudes for exact geographical location.
3. **Owner**: The Owner fields defines the current owner of the product in the supply chain. This helps to understand the flow of product through different people and can be used for traceability and quality analysis.
4. **Datetime**: This field defines the processing time of the product at that specific location with the owner mentioned in the Owner field.
5. **Price**: This field shows the price of the product. The price is fetched from different channels [1] and one product may have different prices as it is on different channels.

All these properties help in properly understanding the flow of products in supply chain as the product moves and reaches to the end user. It helps in traceability, quality analysis, dispute resolution and creation of trust environment between the participants.

[1] multiple channels are created in the next chapter and this example deals with a single channel

5.6 Chaincode functions

Chaincode contain the actual business logic for the blockchain network [Fab18b]. It deals with the transaction execution on the peer on which it is attached and the results are then sent to all the peer for committing so that every peer maintains a shared copy of the same data. There are two types of chaincode methods:

- **Init** method: The Init method is called whenever the chaincode is instantiated or upgraded on a channel [Fab18b]. The initialization is accompanied with initialization of the world state by creating keys and their corresponding values.
- **Invoke** method: The Invoke method is called whenever an invoke transaction is explicitly specified on the peer. In other words, Invoke method creates a transaction proposal on the peer on which it is executed.

Let us examine the different chaincode methods [Com17d] developed in the application.

Listing 5.1 Code snippet for the Invoke method in the product-chaincode.go

```
// product-chaincode.go
/*
 This code is based on code written by the Hyperledger Fabric community.
 Original code can be found here: https://github.com/hyperledger/fabric-samples/blob/release/
chaincode/fabcar/fabcar.go
*/

/*
 * The Invoke method *
 called when an application requests to run the chaincode "product-chaincode"
 The app also specifies the specific chaincode function to call with args
*/
func (s *SmartContract) Invoke(APIstub shim.ChaincodeStubInterface) sc.Response {

  // Retrieve the requested chaincode function and arguments
  function, args := APIstub.GetFunctionAndParameters()
  // Route to the appropriate handler function to interact with the ledger
  if function == "queryProduct" {
    return s.queryProduct(APIstub, args)
  } else if function == "initLedger" {
    return s.initLedger(APIstub)
  } else if function == "createProduct" {
    return s.createProduct(APIstub, args)
  } else if function == "queryAllProducts" {
    return s.queryAllProducts(APIstub)
  } else if function == "changeProductOwner" {
    return s.changeProductOwner(APIstub, args)
  }

  return shim.Error("Invalid chaincode function name.")
}
```

Listing 5.2 Code snippet for the *queryProduct* method in the product-chaincode.go

```
/*
 * The queryProduct method *
Used to view the records of one particular product
It takes one argument -- the productId for the product to be searched
*/
func (s *SmartContract) queryProduct(APIstub shim.ChaincodeStubInterface, args []string) sc.
Response {

  if len(args) != 1 {
    return shim.Error("Incorrect number of arguments. Expecting 1")
  }

  productAsBytes, _ := APIstub.GetState(args[0])
  if productAsBytes == nil {
    return shim.Error("Could not locate product")
  }
  return shim.Success(productAsBytes)
}
```

Listing 5.3 Code snippet for the *createProduct* method in the product-chaincode.go

```
/*
 * The createProduct method *
This method will be used by the Producer to create a record of a new product that becomes part
 of the supply chain.
*/
func (s *SmartContract) createProduct(APIstub shim.ChaincodeStubInterface, args []string) sc.
Response {

  if len(args) != 5 {
    return shim.Error("Incorrect number of arguments. Expecting 5")
  }

  var product = Product{Price: args[1], Location: args[2], Timestamp: args[3], Owner: args[4]}

  productAsBytes, _ := json.Marshal(product)
  err := APIstub.PutState(args[0], productAsBytes)
  if err != nil {
    return shim.Error(fmt.Sprintf("Failed to record product: %s", args[0]))
  }

  return shim.Success(nil)
}
```

Listing 5.4 Code snippet for the *queryAllProducts* method in the product-chaincode.go

```
/*
 * The queryAllProducts method *
allows for assessing all the records added to the ledger(all the products in the ledger)
This method will return an array of JSON response that can be parsed by the front-end
applications to show the results.
*/
func (s *SmartContract) queryAllProducts(APIstub shim.ChaincodeStubInterface) sc.Response {

  startKey := "0"
  endKey := "999"

  resultsIterator, err := APIstub.GetStateByRange(startKey, endKey)
  if err != nil {
    return shim.Error(err.Error())
  }
  defer resultsIterator.Close()

  // buffer is a JSON array containing QueryResults
  var buffer bytes.Buffer
  buffer.WriteString("[")

  bArrayMemberAlreadyWritten := false
  for resultsIterator.HasNext() {
    queryResponse, err := resultsIterator.Next()
    if err != nil {
      return shim.Error(err.Error())
    }
    // Add comma before array members,suppress it for the first array member
    if bArrayMemberAlreadyWritten == true {
      buffer.WriteString(",")
    }
    buffer.WriteString("{\"Key\":")
    buffer.WriteString("\"")
    buffer.WriteString(queryResponse.Key)
    buffer.WriteString("\"")

    buffer.WriteString(", \"Record\":")
    // Record is a JSON object, so we write as-is
    buffer.WriteString(string(queryResponse.Value))
    buffer.WriteString("}")
    bArrayMemberAlreadyWritten = true
  }
  buffer.WriteString("]")

  fmt.Printf("- queryAllProducts:\n%s\n", buffer.String())

  return shim.Success(buffer.Bytes())
}
```

Listing 5.5 Code snippet for the *changeProductOwner* method in the product-chaincode.go

```
/*
 * The changeProductOwner method *
The data in the world state can be updated with currently has the product.
This function takes in 2 arguments, product id and new owner name.
*/
func (s *SmartContract) changeProductOwner(APIstub shim.ChaincodeStubInterface, args []string)
 sc.Response {

  if len(args) != 2 {
    return shim.Error("Incorrect number of arguments. Expecting 2")
  }

  productAsBytes, _ := APIstub.GetState(args[0])
  if productAsBytes == nil {
    return shim.Error("Could not locate product")
  }
  product := Product{}

  json.Unmarshal(productAsBytes, &product)
  // Normally check that the specified argument is a valid owner of the product
  // we are skipping this check for this example
  product.Owner = args[1]

  productAsBytes, _ = json.Marshal(product)
  err := APIstub.PutState(args[0], productAsBytes)
  if err != nil {
    return shim.Error(fmt.Sprintf("Failed to change product owner: %s", args[0]))
  }

  return shim.Success(nil)
}
```

Listing 5.6 Code snippet for the *main* method in the product-chaincode.go

```
/*
 * main function *
calls the Start function
The main function starts the chaincode in the container during instantiation.
*/
func main() {

  // Create a new Smart Contract
  err := shim.Start(new(SmartContract))
  if err != nil {
    fmt.Printf("Error creating new Smart Contract: %s", err)
  }
}
```

5.7 Web Front-End Application

In this section, we will see the structure and flow of different actions that a user can perform with the front-end application. The user is authenticated with the login credentials and depending upon the type of user, different actions are performed by the user. The application uses the Hyperledger Fabric Client SDK to interact with the blockchain ledger.

5.7.1 Get All Products

Here we will see the front-end application part where we get all the details about all the products. We will also see how the blockchain network chaincode function **queryAllProducts** is called from the front-end application.

Get All Products

Get All

Product ID	DateTime	Owner	Product Location (Longitude, Latitude)	Price (€)
1	12/10/2018 18:30:01	Vikas	67.0006, -70.5476	12
2	11/10/2018 14:30:01	Ghareeb	91.2395, -49.4594	13
3	11/10/2018 11:30:01	Jason	58.0148, 59.01391	74
4	12/10/2018 18:30:00	Romeet	-45.0945, 0.7949	12
5	12/10/2018 12:30:01	Raja	-107.6043, 19.5003	12
6	12/08/2018 18:30:01	Bhupendra	-155.2304, -15.8723	16
7	12/10/2018 15:30:01	Mehul	103.8842, 22.1277	12.89
8	10/10/2018 18:30:01	Desi	-132.3207, -34.0983	14.7
9	12/10/2018 18:35:01	Mishri	153.0054, 12.6429	12.8
10	06/10/2018 18:30:01	Janice	51.9435, 8.2735	17.9

Figure 5.5: The view for displaying all the products in the blockchain.

Figure 5.5 shows the product detail view as the product flows in the supply chain. The information about products get updated depending upon the user functions.

Listing 5.7 Code snippet for the connecting to the Fabric Client SDK.

```
//define the object for the Fabric_Client SDK connection
var fabric_client = new Fabric_Client();

// setup the fabric network
//defines the name of the channel on which peer has to execute transaction
var channel = fabric_client.newChannel('mychannel');
//creates a peer object by passing its unique docker url
var peer = fabric_client.newPeer('grpc://192.168.99.100:7051');
//adds the peer object to the channel object for transaction execution
channel.addPeer(peer);
```

Listing 5.8 Code snippet for the *queryAllProducts* method in the controller function that calls the chaincode with the chaincodeId

```
// queryAllProducts - requires no arguments , ex: args: [''],
    const request = {                  //request defines the request object created
    chaincodeId: 'product-chaincode', //the chaincode name on which transaction is executed
    txId: tx_id,                       //transaction id assigned to this action
    fcn: 'queryAllProducts',           //chaincode function name
    args: ['']                         //args represent the function parameters
};
```

5.7.2 Get Product by Id

This view represents the specific product that can be viewed by passing its product id. It requires one parameter, the name of the key which has to be searched in the world state. In our scenario, the key is the product id that was created by the user or during initialization.

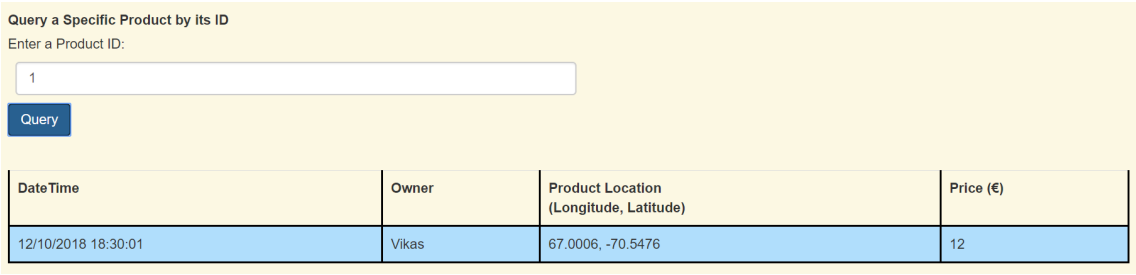

Figure 5.6: The view for fetching a product by its id in the blockchain.

Listing 5.9 Code snippet for the *queryProduct* method in the controller function that calls the chaincode with the chaincodeId

```
// queryProduct - requires 1 argument, ex: args: ['4'],
    //the request object created to pass to the client SDK
    const request = {
    //name of the installed chaincode for transaction execution
    chaincodeId: 'product-chaincode',
    //the generated transaction id
    txId: tx_id,
    //name of the method in the chaincode
    fcn: 'queryProduct',
    //the value of the id passed to the controller
    args: [id]
};
```

5.7.3 Create a Product

Here we will see the view designed for creating a new product in the blockchain network. The producer has the permissions to create a new product that becomes part of the supply chain.

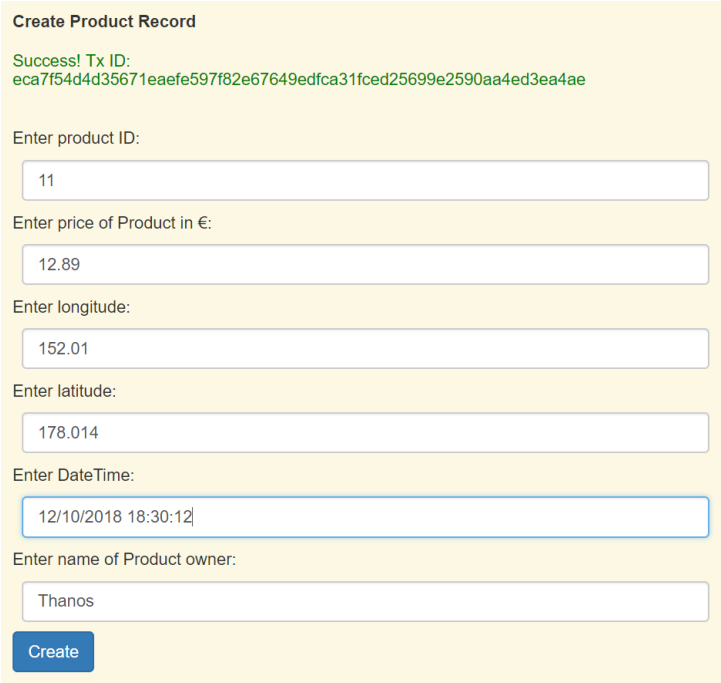

Figure 5.7: The view for creating a new product and the acknowledgement of the transaction with transaction id

Listing 5.10 Code snippet for the *createProduct* method in the controller function that calls the chaincode with the chaincodeId

```
 // the request object is sent to the endorser as a proposal
    const request = {
    //name of the installed chaincode for transaction execution
    chaincodeId: 'product-chaincode',
    //name of the method in the chaincode
    fcn: 'createProduct',
    //the required fields passed to the controller
    args: [id, price, location, timestamp, owner],
    //name of the channel
    chainId: 'mychannel',
    //the generated transaction id
    txId: tx_id
};

// after the request is created, the transaction proposal is sent to the peers
```

5.7.4 Update a Product

As the product flows in the supply chain, it constantly changes hands. It is therefore, essential to update the recent owner of a product so that any future issues can be easily resolved. The Supplier and Buyer are the ones who are between the Consumer and Producer. The product is updated whenever it flows from one user to the other.

Figure 5.8: The view for updating a product with the new owner.

Listing 5.11 Code snippet for the *changeProductOwner* method in the controller function that calls the chaincode with the chaincodeId

```
// the transaction proposal is created here and sent to endorser
    var request = {
    //name of the installed chaincode for transaction execution
    chaincodeId: 'product-chaincode',
    //name of the method in the chaincode
    fcn: 'changeProductOwner',
    //the required fields passed to the controller
    args: [id, owner],
    //name of the channel
    chainId: 'mychannel',
    //the transaction id generated in the request
    txId: tx_id
};

// after the request is created, the transaction proposal is sent to the peers
```

5.8 Android Application

The Android application is designed keeping the **Consumer** in mind. The consumer has no direct access to the web application. Thus, in order to check if a product has gone through all the required tests or to check if the product is illegitimate, the Consumer can use the app, scan the QR-Code on the product and verify the product details.

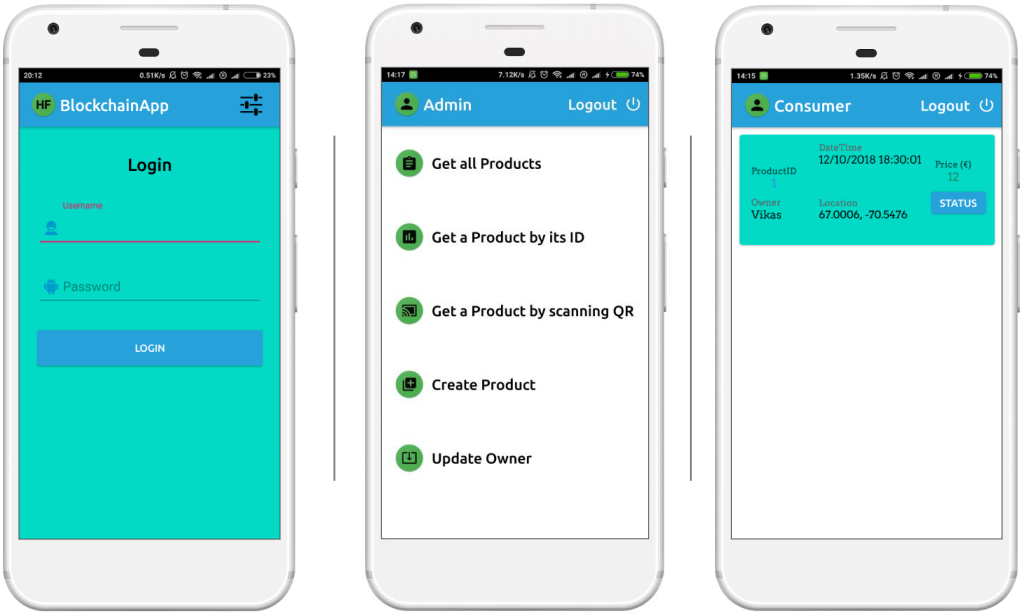

Figure 5.9: The mobile app screens designed for the blockchain network

Figure 5.9 shows the different screens designed for the Android app. Authentication is required for the user to login to the blockchain network and to use the app. With the correct credentials, the view with the designated functions is shown to the user. For example, **Admin** user sees all the functions in the app. The **Consumer** can see the two relevant functions like *Get a Product by its ID*, where the numeric id can be entered on the screen by the consumer, and *Get a Product by scanning QR* where a QR can be scanned for the Product ID and the result will be shown as displayed in the third screen.

The app also works in the same way as the front-end web application works. The app is designed in JAVA using the Android SDK and is compatible with the Android devices running Android 3.0 or higher.

Figure 5.10 shows the architecture of the app that uses the backend developed in Node.js. The request proposal is sent to the backend which then forwards it to the respective chaincode method. The chaincode method is executed on the peer and the transaction proposal response is sent back to the Node.js backend. The response is returned as a JSON response to the app which then parses the object and shows the result back to the user. The use cases in the app are similar to the web frontend application with additional feature of QR scanning for the product id.

Figure 5.11 shows the extension of the mobile application that can be designed to provide the important functionality of traceability and quality assurance. The extension is not discussed here and just shows the idea but can be implemented as a future work for the app.

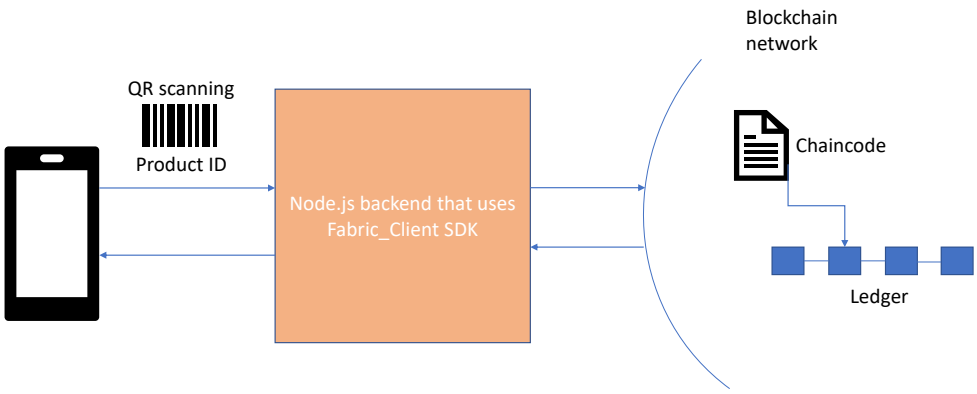

Figure 5.10: The architecture of the mobile application interacting with the blockchain network.

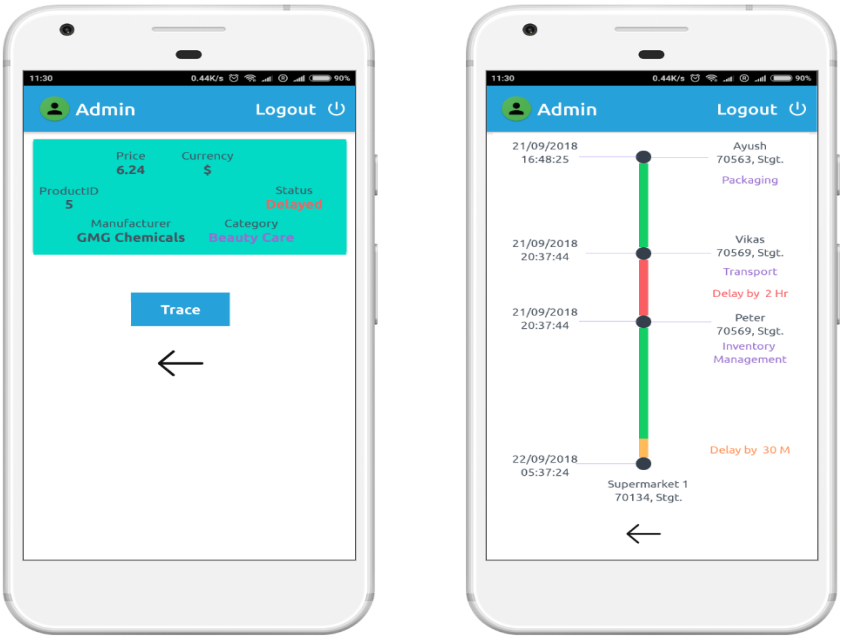

Figure 5.11: The extension of the app that can be used to view the status and other information about a product.

6 Experimental Design and Results

In this chapter, we will discuss the different experiments performed with the blockchain network to study various important transaction characteristics of permissioned blockchain networks. The business logic is written in chaincodes and therefore, the focus will be on chaincodes and the ledger for the experiments.

6.1 Experiment 1: Chaincode to Chaincode Call on Same Channel

In this section we will see how a chaincode transaction is scoped when one chaincode calls another chaincode on the same channel. The chaincode call can be a read transaction on the world state or a write transaction on the world state and the blockchain. Therefore, the experiment will be divided into two subsections for read and write. We will verify and put forward the results.

6.1.1 Read from Same Channel

There are various business transactions that require data from different applications in the network. Chaincodes contain the business logic for the blockchain applications and transactions can only be executed with the chaincodes method on the ledger. It is therefore important to study the behaviour of chaincode in a read operation requiring data from a different chaincode.

Experimental Setup

In this approach, we will use two different chaincodes, *chaincode1* and *chaincode2*. These chaincodes will be installed on the peer (considering a single peer) and will be on the same channel. In this setup, we will use a chaincode2 method to call chaincode1 method to read the world state of chaincode1.

We will first create two keys **a** and **b** with some initial value by instantiating the chaincode chaincode1. Then we will instantiate chaincode2 with a key **sum** and initial value as **0**. Then with the help of chaincode2, we will call a method that reads the values of **a** and **b** and puts the values in the key **sum** that is scoped to chaincode2. The chaincode1 is based on the code at [Com17b] and chaincode2 at [Com17c].

Listing 6.1 shows the *invoke* method defined in chaincode1 that deducts value from key **a** and adds the same value to key **b**.

Similarly, Listing 6.2 shows the invoke method for the chaincode2 that reads the keys **a** and **b** from chaincode1 and shows its sum in the key **sum** of chaincode2.

Listing 6.1 Code snippet for the *invoke* method in the chaincode1

```
/* the experiment is based on the code at https://github.com/hyperledger/fabric/blob/release
-1.3/examples/chaincode/go/example02/chaincode.go */
// Transaction makes payment of X units from A to B
func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string) pb.Response
{

    //A and B are the arguments or keys initialized
  A = args[0]
  B = args[1]

  // Get the state from the ledger
  Avalbytes, err := stub.GetState(A)
  if err != nil {
    return shim.Error("Failed to get state")
  }
  Aval, _ = strconv.Atoi(string(Avalbytes))

  Bvalbytes, err := stub.GetState(B)
  if err != nil {
    return shim.Error("Failed to get state")
  }
  Bval, _ = strconv.Atoi(string(Bvalbytes))

  // Perform the execution
  X, err = strconv.Atoi(args[2])
  if err != nil {
    return shim.Error("Invalid transaction amount, expecting a integer value")
  }
  //sends a value from A to B
  Aval = Aval - X
  Bval = Bval + X
  fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)

  // Write the state back to the ledger
  err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
  if err != nil {
    return shim.Error(err.Error())
  }

  err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
  if err != nil {
    return shim.Error(err.Error())
  }

  return shim.Success(nil)
}
```

Listing 6.2 Code snippet for the *invoke* method in the chaincode2

```
/* the experiment is based on the code at https://github.com/hyperledger/fabric/blob/release
-1.3/examples/chaincode/go/example05/chaincode.go */
// Invoke queries another chaincode and updates its own state
func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string) pb.Response
{
  var sum, channelName string // Sum entity
  var Aval, Bval, sumVal int  // value of sum entity - to be computed
  var err error
  chaincodeName := args[0] // Expecting name of the chaincode you would like to call, this
name would be given during chaincode install time
  sum = args[1]

  if len(args) > 2 {
    channelName = args[2]
  } else {
    channelName = ""
  }

  // Query chaincode1
  f := "query"
  queryArgs := toChaincodeArgs(f, "a")

  //   if chaincode being invoked is on the same channel,
  //   then channel defaults to the current channel and args[2] can be "".
  //   If the chaincode being called is on a different channel,
  //   then you must specify the channel name in args[2]

  response := stub.InvokeChaincode(chaincodeName, queryArgs, channelName)
  Aval, err = strconv.Atoi(string(response.Payload))

  queryArgs = toChaincodeArgs(f, "b")
  response = stub.InvokeChaincode(chaincodeName, queryArgs, channelName)

  Bval, err = strconv.Atoi(string(response.Payload))
  // Compute sum
  sumVal = Aval + Bval

  // Write sumVal back to the ledger
  err = stub.PutState(sum, []byte(strconv.Itoa(sumVal)))
  fmt.Printf("Invoke chaincode successful. Got sum %d\n", sumVal)
  return shim.Success([]byte(strconv.Itoa(sumVal)))
}
```

The various steps followed to install the chaincode and perform the operations can be described as follows:

Install Chaincode

Listing 6.3 CLI command for chaincode *install* on same channel for read operation.

```
//installs the chaincode1 with name as chaincode1 on channel mychannel and the chaincode file
is located at the path specified by -p
peer chaincode install -n chaincode1 -C mychannel -v 1.0 -p github.com/chaincode/chaincode1/
//installs the chaincode2 with name as chaincode2 on channel mychannel and the chaincode file
is located at the path specified by -p
peer chaincode install -n chaincode2 -v -C mychannel 1.0 -p github.com/chaincode/chaincode2/
```

Instantiate Chaincode

Listing 6.4 CLI command for chaincode *instantiate* on same channel for read operation.

```
//instantiates chaincode1 with key a=100, b=200 and the endorsement policy is specified with -
P, -o specifies the orderer endpoint
peer chaincode instantiate -o 127.0.0.1:7050 -C $CHANNEL_NAME -n chaincode1 -v 1.0 -c '{"Args
":["init","a", "100", "b","200"]}' -P "OR ('Org1MSP.member','Org2MSP.member')"

//instantiates chaincode2 with key sum=0 and the endorsement policy is specified with -P, -o
specifies the orderer endpoint
peer chaincode instantiate -o 127.0.0.1:7050 -C $CHANNEL_NAME -n chaincode2 -v 1.0 -c '{"Args
":["init","sum", "0"]}' -P "OR ('Org1MSP.member','Org2MSP.member')"
```

Query Chaincode

Listing 6.5 CLI command for chaincode *query* on same channel for read operation.

```
peer chaincode query -C $CHANNEL_NAME -n chaincode1 -c '{"Args":["query","a"]}' //response is
a=100
peer chaincode query -C $CHANNEL_NAME -n chaincode1 -c '{"Args":["query","b"]}' //response is
b=200
//the args specifies in order the method in chaincode2 to call, the target callee chaincode
name as chaincode1, the key in which value is stored and the channel name on which chaincode1
is installed.
peer chaincode query -n chaincode2 -c '{"Args":["query","chaincode1","sum","mychannel"]}' //
response is sum=300
```

Invoke Chaincode

Listing 6.6 CLI command for chaincode *invoke* on same channel for read operation.

```
peer chaincode invoke -o 127.0.0.1:7050 -C $CHANNEL_NAME -n chaincode1 -c '{"Args":["invoke","
a","b","10"]}' //transfers 10 from a to b

peer chaincode invoke -o 127.0.0.1:7050 -C $CHANNEL_NAME -n chaincode2 -c '{"Args":["invoke","
chaincode1","sum","mychannel"]}' //adds the value of a and b in chaincode1 and saves sum=300
```

Final Query

Listing 6.7 CLI command for chaincode *query* on same channel for read operation.

```
peer chaincode query -C $CHANNEL_NAME -n chaincode1 -c '{"Args":["query","a"]}' //response is
a=90
peer chaincode query -C $CHANNEL_NAME -n chaincode1 -c '{"Args":["query","b"]}' //response is
b=210
peer chaincode query -n chaincode2 -c '{"Args":["query","chaincode1","sum","mychannel"]}' //
response is sum=300
```

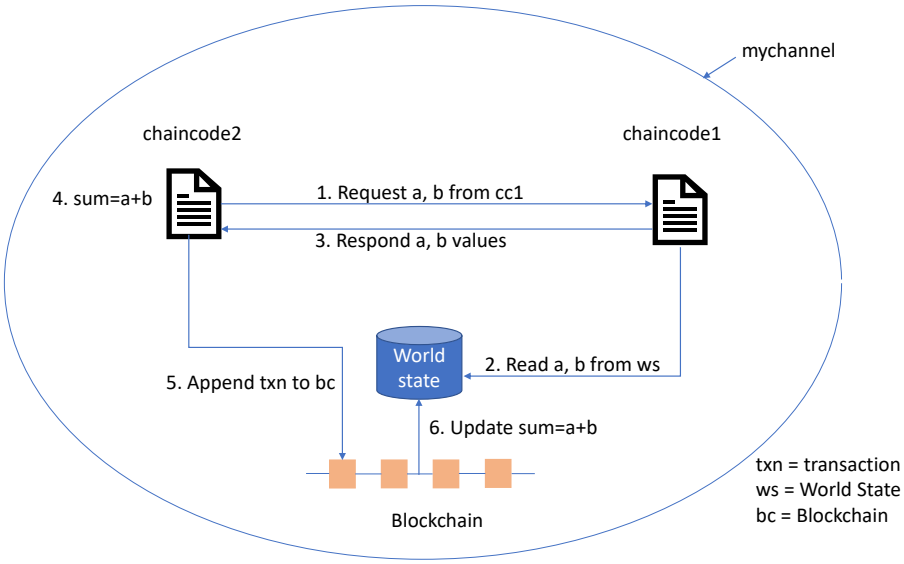

Figure 6.1: The overall flow of read operation from a different chaincode on the same channel.

The overall flow of the experiment can be seen in the Figure 6.1. The figure shows the different operations that reads values from chaincode1 and returns those values to chaincode2 for further calculations.

Results

The above experiment to read the chaincode state from different chaincode can be summarized in Table 6.1. The table shows different values of the chaincode keys after different operations were performed. The key **sum** that belongs to **chaincode2** contains the sum of the values read for key **a** and key **b** from **chaincode1**.

Keys	Chaincode Operations				
	Install	**Instantiate**	**Query**	**Invoke**	**Final Query**
a	-	100	100	90	90
b	-	200	200	210	210
sum	-	0	300	300	300

Table 6.1: Different values for the keys during different chaincode operations for read from same channel.

Conclusion of Results

The experiment helps us to conclude that

1. A chaincode can read states of other chaincodes in the same channel with proper authorization.
2. The caller chaincode, **chaincode2** interprets the object in the chaincode logic, and this object is not part of the MVCC for chaincode2 unless the caller chaincode writes it back to the ledger (using putState).
3. Any error in the called method of a different chaincode can be handled in the business logic of the caller chaincode that helps to create nested transactions characterized by read operations across different chaincodes that are both **atomic** and **consistent**.

6.1.2 Write on same channel

In the previous experiment, we have seen that chaincode can read state of another chaincode within the same channel. In this experiment, we will try to write to a chaincode state by executing a method of a different chaincode on the same channel.

Experimental setup

For the experiment, we will use the same two chaincodes, **chaincode1** and **chaincode2** discussed in the previous subsection. However, we will create a new method in chaincode2 that writes a value to the keys belonging to chaincode1.

The method in chaincode2 will deduct value of key **a** by amount **x** and increase the value of key **b** by the same amount **x**. It is important to note that the operation will be called on **chaincode2** and the keys belong to **chaincode1**.

Listing 6.8 Code snippet for the *write* method in the chaincode2

```
/*the experiment is based on the code at https://github.com/hyperledger/fabric/blob/release
-1.3/examples/chaincode/go/example02/chaincode.go*/
//write method for writing to chaincode1
func (t *SimpleChaincode) write(stub shim.ChaincodeStubInterface, args []string) pb.Response {
  var channelName string // channel Name
  var A, B string        // Entities
  var X int              // Transaction value
  var err1 error

  chaincodeName := args[0] // Expecting name of the chaincode you would like to call, this
name would be given during chaincode install time
  A = args[1]
  B = args[2]
  X, err1 = strconv.Atoi(args[3])

  if len(args) > 2 {
    channelName = args[4]
  } else {
    channelName = ""
  }

  // invoke function of chaincode1 is called that puts A=A-X and B=B+X
  f := "invoke"
  queryArgs := toChaincodeArgs(f, A, B, strconv.Itoa(X))
  response := stub.InvokeChaincode(chaincodeName, queryArgs, channelName)
  if response.Status != shim.OK {
    errStr := fmt.Sprintf("Failed to query chaincode. Got error: %s", response.Payload)
    fmt.Printf(errStr)
    return shim.Error(errStr)
  }
  fmt.Printf("Query Response:%s\n", X)
  return shim.Success([]byte(strconv.Itoa(X)))
}
```

In section 6.1.1, we have already seen how the chaincodes are installed in listing 6.3, instantiated in listing 6.4 and queried in listing 6.5. The new operation introduced in this section is invoking a method **write** that will write to chaincode1 state.

Invoke Chaincode

Listing 6.9 CLI command for chaincode *invoke* with write operation on same channel.

```
peer chaincode invoke -o 127.0.0.1:7050 -C $CHANNEL_NAME -n chaincode1 -c '{"Args":["invoke","
a","b","10"]}' //transfers 10 from a to b, a=90, b=210
peer chaincode invoke -o 127.0.0.1:7050 -C $CHANNEL_NAME -n chaincode2 -c '{"Args":["write","
chaincode1","a","b","27","mychannel"]}' //calls the invoke method of chaincode1 that transfers
 27 from a to b, a=63, b=237
```

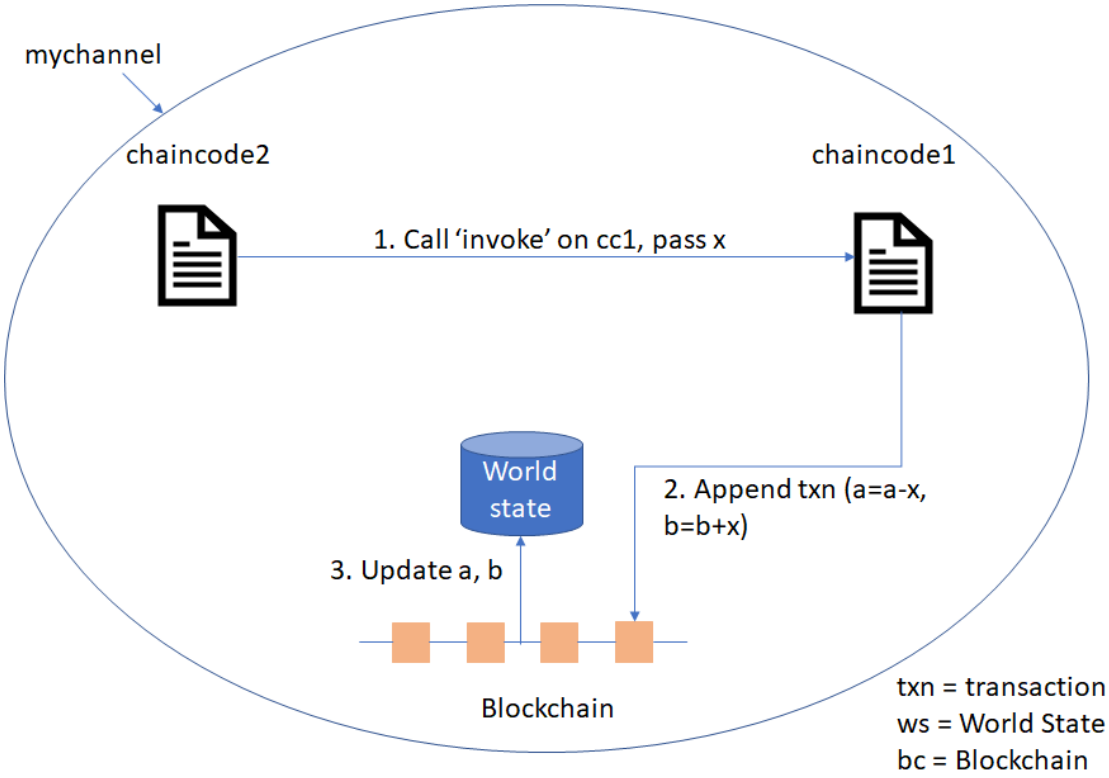

Figure 6.2: The overall flow of write operation from a different chaincode on the same channel.

The overall flow of the experiment can be seen in the Figure 6.2. The figure shows the different operations that invokes method on chaincode1 which in turn writes the passed values from chaincode2 to the keys of chaincode1.

Results

The above experiment to write to the chaincode state from a different chaincode can be summarized in Table 6.2. The table shows the different values of the chaincode keys after different operations were performed. The keys **a** and **b** belong to **chaincode1** and their values are modified by calling a write operation from **chaincode2**.

Keys	**Chaincode Operations**				
	Install	**Instantiate**	**Query**	**Invoke_cc1**	**Invoke_cc2**
a	-	100	100	90	63
b	-	200	200	210	237

Table 6.2: Different values for the keys during different chaincode operations during write on same channel.

Conclusion of Results

The experiment helps us to conclude that

1. A chaincode can write states to other chaincodes in the same channel with proper authorization.
2. The key version check will take place for both the chaincodes in a single transaction context and if any check fails, the whole transaction will be termed as invalid.
3. Nested transactions spanning two different chaincodes can be considered as **atomic** and **consistent**.

6.2 Experiment 2: Chaincode to Chaincode Call on Different Channels

In this section, we will perform the same experiment as in section 6.1 but the two chaincodes will be installed on different channels on a single peer.

6.2.1 Read from Different Channels

We will use the same two chaincodes, **chaincode1** and **chaincode2** discussed in the previous section. The goal will be to call a chaincode method to read its state that is present on a different channel but on the same peer.

Experimental Setup

The steps are similar to what is described in section 6.1.1 with a major difference in the chaincode path. The **chaincode1** will be installed on **mychannel1** and **chaincode2** on **mychannel2** with the code in the two chaincodes remaining the same.

The **chaincode2** will invoke a method on **chaincode1** that queries and reads the states for keys, **a** and **b** belonging to chaincode1. These read values will be added and saved in the key **sum** belonging to chaincode2.

The different steps to perform the experiment can be highlighted as follows:

Install Chaincode

Listing 6.10 CLI command for chaincode *install* for read operation on different channels.

```
//installs the chaincode1 with name as chaincode1 on channel mychannel1 and the chaincode file
 is located at the path specified by -p
peer chaincode install -n chaincode1 -C mychannel1 -v 1.0 -p github.com/chaincode/chaincode1/
//installs the chaincode2 with name as chaincode2 on channel mychannel2 and the chaincode file
 is located at the path specified by -p
peer chaincode install -n chaincode2 -v -C mychannel2 1.0 -p github.com/chaincode/chaincode2/
```

Instantiate Chaincode

Listing 6.11 CLI command for chaincode *instantiate* for read operation on different channels.

```
//instantiates chaincode1 on channel mychannel1 with key a=100, b=200 and the endorsement
policy is specified with -P, -o specifies the orderer endpoint
peer chaincode instantiate -o 127.0.0.1:7050 -C mychannel1 -n chaincode1 -v 1.0 -c '{"Args":["
init","a", "100", "b","200"]}' -P "OR ('Org1MSP.member','Org2MSP.member')"

//instantiates chaincode2 on channel mychannel2 with key sum=0 and the endorsement policy is
specified with -P, -o specifies the orderer endpoint
peer chaincode instantiate -o 127.0.0.1:7050 -C mychannel2 -n chaincode2 -v 1.0 -c '{"Args":["
init","sum", "0"]}' -P "OR ('Org1MSP.member','Org2MSP.member')"
```

Query Chaincode

Listing 6.12 CLI command for chaincode *query* for read operation on different channels.

```
peer chaincode query -C mychannel1 -n chaincode1 -c '{"Args":["query","a"]}' //response is a
=100
peer chaincode query -C mychannel1 -n chaincode1 -c '{"Args":["query","b"]}' //response is b
=200
//the args specifies in order the method in chaincode2 to call, the target callee chaincode
name as chaincode1, the key in which value is stored and the channel name on which chaincode1
is installed which is mychannel1.
peer chaincode query -n chaincode2 -C mychannel2 -c '{"Args":["query","chaincode1","sum","
mychannel1"]}' //response is sum=300
```

Invoke Chaincode

Listing 6.13 CLI command for chaincode *invoke* for read operation on different channels.

```
peer chaincode invoke -o 127.0.0.1:7050 -C mychannel1 -n chaincode1 -c '{"Args":["invoke","a
","b","10"]}' //transfers 10 from a to b

peer chaincode invoke -o 127.0.0.1:7050 -C mychannel2 -n chaincode2 -c '{"Args":["invoke","
chaincode1","sum","mychannel1"]}' //adds the value of a and b scoped to chaincode1 and saves
sum=300 scoped to chaincode2
```

Final Query

Listing 6.14 CLI command for chaincode *query* for read operation on different channels.

```
peer chaincode query -C mychannel1 -n chaincode1 -c '{"Args":["query","a"]}' //response is a
=90
peer chaincode query -C mychannel1 -n chaincode1 -c '{"Args":["query","b"]}' //response is b
=210
peer chaincode query -n chaincode2 -C mychannel2 -c '{"Args":["query","chaincode1","sum","
mychannel1"]}' //response is sum=300
```

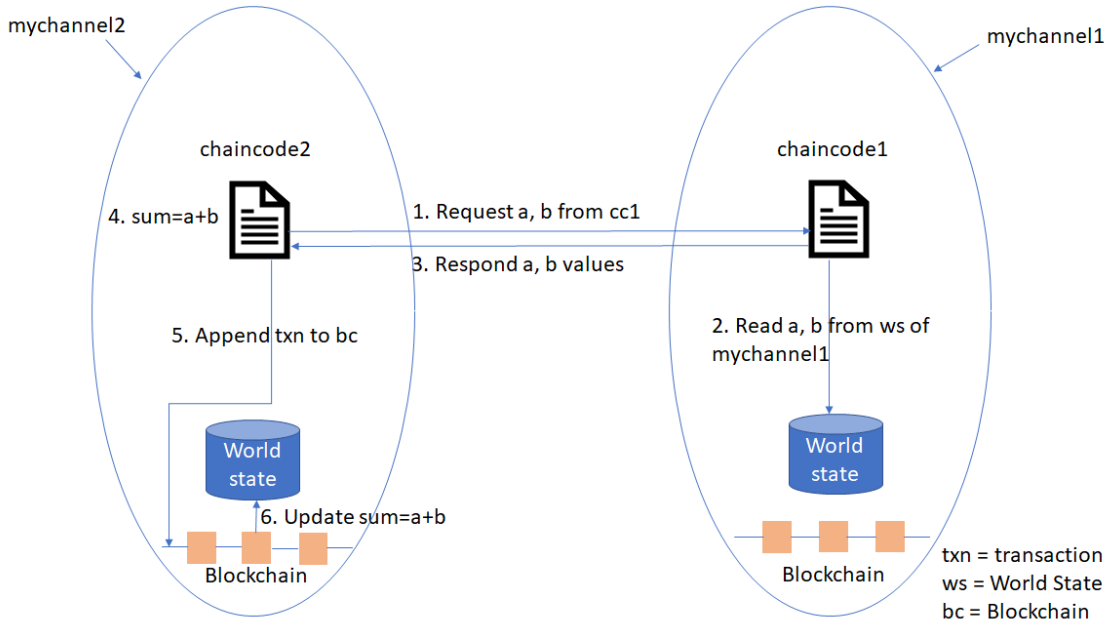

Figure 6.3: The overall flow of read operation from a different chaincode on different channels

The overall flow of the experiment can be seen in the Figure 6.3. The figure shows the different operations that reads values from chaincode1 and returns those values to chaincode2 for further calculations.

Results

The above experiment to read the chaincode state from different chaincodes on different channels can be summarized in Table 6.3. The table shows the different values of the chaincode keys after different operations were performed. The key **sum** that belongs to **chaincode2** on channel **mychannel2** contains the sum of the values read for key **a** and key **b** from **chaincode1** that belongs to channel **mychannel1**. The experiment was also performed for different values of **a** and **b** and it concurred with the previous results.

Keys	Chaincode Operations				
	Install	**Instantiate**	**Query**	**Invoke**	**Final Query**
a	-	100	100	90	90
b	-	200	200	210	210
sum	-	0	300	300	300

Table 6.3: Different values for the keys during different chaincode operations during read from different channels.

Conclusion of Results

The experiment helps us to conclude that

1. A chaincode can read states of other chaincodes on different channels with proper authorization.
2. The caller chaincode, **chaincode2** interprets the object in the chaincode logic, and this object is not part of the MVCC for chaincode2 unless the caller chaincode writes it back to the ledger (using putState).
3. Any error in the called method of a different chaincode can be handled in the business logic of the caller chaincode that can help to create nested transactions characterized by read operations across different chaincodes that are both **atomic** and **consistent**.
4. The check will be based on read-write set of the callee chaincode, **chaincode1**, and hence, **chaincode2** should have read access for channel **mychannel1**.

6.2.2 Write on Different Channels

In the previous experiment, we have seen that chaincode can read state of another chaincode present in a different channel. In this experiment, we will try to write to a chaincode state by executing a method of a different chaincode on a different channel.

Experimental Setup

For the experiment, we will use the same two chaincodes, **chaincode1** and **chaincode2** discussed in the previous section and the method **write** discussed in listing 6.8.

We will call a method **write** defined in chaincode2 on channel **mychannel2**. The entire setup is similar to subsection 6.1.2 with the only difference that chaincode1 will be on channel **mychannel1** and chaincode2 on **mychannel2**.

The steps include Install, Instantiate, Query, Invoke and Final Query as outlined in previous subsection. Lets see the results from Invoke method.

Invoke Chaincode

Listing 6.15 CLI command for chaincode *invoke* for write operation on different channels.

```
peer chaincode invoke -o 127.0.0.1:7050 -C mychannel1 -n chaincode1 -c '{"Args":["invoke","a
","b","10"]}' //transfers 10 from a to b, a=90, b=210
peer chaincode invoke -o 127.0.0.1:7050 -C mychannel2 -n chaincode2 -c '{"Args":["write","
chaincode1","a","b","27","mychannel1"]}' //the write results in no change to values of a and b
, a=90 and b=210
```

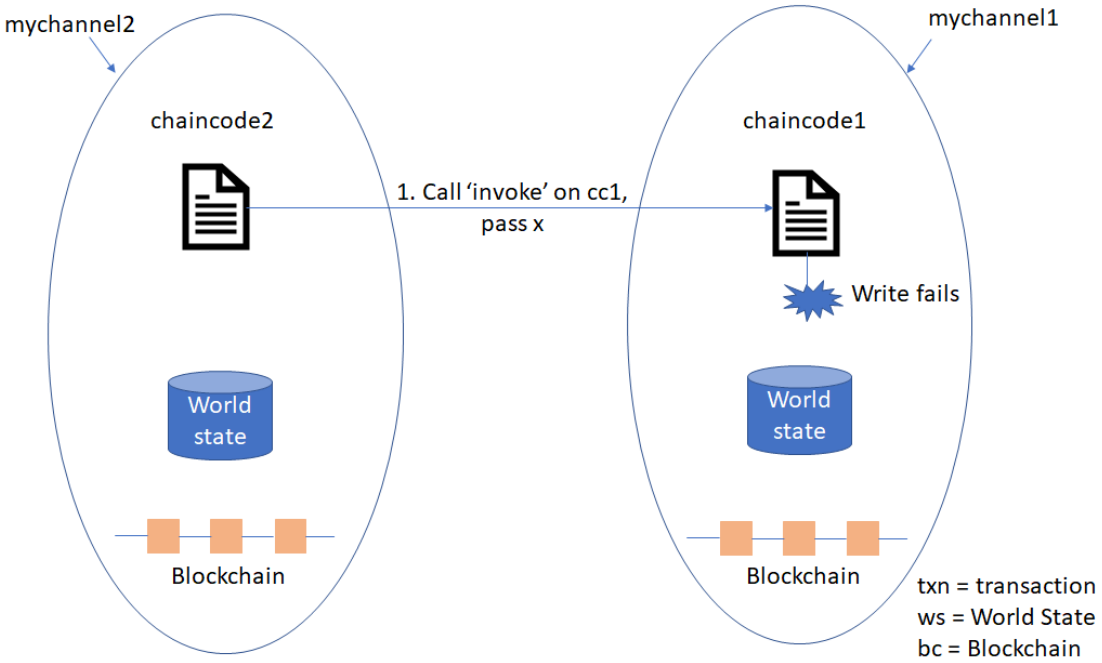

Figure 6.4: The overall flow of write operation from a different chaincode on a different channel.

The overall flow of the experiment can be seen in the Figure 6.4. The figure shows the different operations that invokes method on chaincode1 which in turn tries to update the state of chaincode1 on channel **mychannel1**.

Results

The above experiment to write to the chaincode state from a different chaincode on a different channel can be summarized in Table 6.4. The table shows different values of the chaincode keys after different operations were performed. The keys **a** and **b** belong to **chaincode1** on channel **mychannel1**.

Keys	Chaincode Operations				
	Install	**Instantiate**	**Query**	**Invoke_cc1**	**Invoke_cc2**
a	-	100	100	90	90
b	-	200	200	210	210

Table 6.4: Different values for the keys during different chaincode operations during write on different channels.

Conclusion of Results

The experiment helps us to conclude that

1. A chaincode on one channel cannot write to a chaincode state on a different channel.
2. Since every channel maintains its own ledger, it is not possible to update the state in a different channel.
3. In case a transaction consists of multiple sub-transactions, a 2-PC transaction monitor will be required that will manage the atomic commits and keep the blockchain and world state consistent.

6.3 Experiment 3: Implementation of an Intermediate Layer for Transaction Management

In the previous sections, we were able to conclude that chaincode to chaincode write is not possible if the two chaincodes are on separate channels. Channels in Hyperledger Fabric are created to provide data privacy to the member of that channel and hence, an external write request fails [Fab18c]. But a business transaction may include scenarios that require write on different channels and within the same transaction context. In short, a large business transaction may consist of smaller transactions that have write operations on different channels.

There are different alternatives with the use of channels discussed as follows :

1. Use of a single channel and private data collection to provide privacy for only the data that is sensitive and cannot be shared with other peers on the channel. It will serve the purpose of multiple channels for data confidentiality.
2. Use of single channel with data encryption.
3. Use of multiple channels with an intermediate implementation of a software layer like a 2-PC transaction monitor on the application side that has access to all the channel's data and manages the dependent transactions. This layer has to take care of **rollback**, **compensation action** and **commit** for all the transactions. However, this type of implementation will make the network slow.

We will focus on the third approach above and create a software implementation on the client application that handles multiple transactions within a single transaction context.

6.3.1 Use Case Scenario

Consider a scenario involving a large business transaction spanning 2 channels, **mychannel2** and **mychannel3**. Consider there is a **supplier1** and **company1** on **mychannel2** and **supplier2** and **company1** on **mychannel3**. Now consider a large business transaction such that the price on **mychannel2** for a product will be reduced if and only if the price of the product can be increased on **mychannel3**. These are 2 sub-transactions part of a large single business transaction. Since, we cannot write to a different channel, a software implementation that manages the transaction is developed. The developed software implementation is an extension of the application developed in chapter 5.

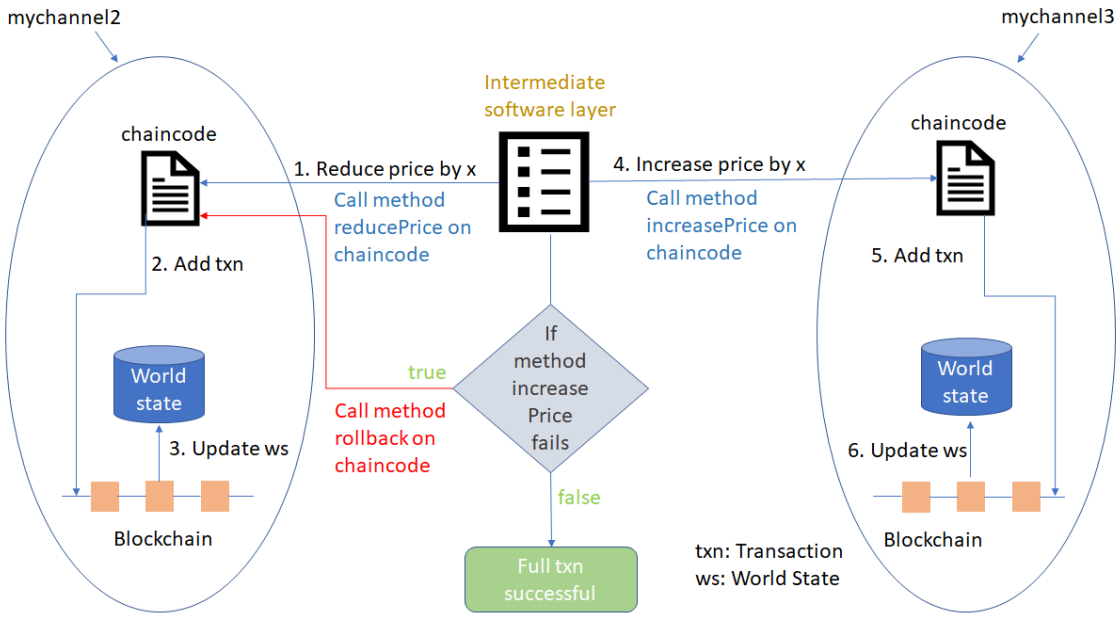

Figure 6.5: The overall flow of two separate transactions managed within a single transaction context.

Figure 6.5 shows the system with two different channels. Let us discuss the design of the software layer.

1. The entire transaction to reduce price on **mychannel2** and increase price on **mychannel3** is a single transaction that has to be managed as two sub-transactions by the intermediate software layer.

2. The transaction starts by reducing price on channel **mychannel2**. This is achieved by calling the chaincode method **reducePrice**. At the same time, the last value for price before reducing it on **mychannel2** is saved in a **localStorage** string variable and a boolean variable that stores if **increasePrice** was successful or not.

3. When the **increasePrice** method is called, the boolean variable **isPriceIncreased** is set to true. This marks the completion of entire transaction.

4. However, there are cases when the transaction for increasing price fails. In that scenario, the boolean variable **isPriceIncreased** is set to false. The software layer checks regularly if the **increasePrice** method was successfully executed. If its not successfully executed, a new transaction on **mychannel2** is submitted that puts the old price in the world state of **mychannel2**. This is mimicking the **rollback** transaction but with a **compensation action** as a blockchain is an append-only ledger.
5. Similar check can be carried out for **reducePrice** as the order of transaction does not matter and for the entire transaction to be successful, both the transactions should successfully be committed to their respective ledgers.
6. Thus the software layer works like a 2-PC transaction monitor for transaction management of multiple sub-transactions spanning different channels.

Figure 6.6: The view of the software layer that handles transaction on multiple channels.

Figure 6.6 shows the view that helped to conduct the experiment. The two actions, to reduce price and increase price, are put on respective button clicks as shown in the figure for experimentation. However, in real scenarios this will be a single event and can be managed by execution of one method after another with **callback**.

6.3.2 Results

With the help of this experiment, the price on both the channels was changed and this was part of a single transaction context. Write on a different channel is not allowed, but with an intermediate software layer we were able to manage transactions spanning multiple channels. It is also possible to rollback the transaction effects on the world state with proper implementation. However, a rollback transaction on the blockchain involves submitting a new transaction that is also recorded on the blockchain as it is an append-only ledger.

6.3.3 Conclusion of Results

It is possible to manage transactions spanning different channels. However, this makes the system slow due to an intermediate layer and should be implemented only when it is absolutely essential. It is possible to rollback the transaction effect by submitting a new transaction. In short, the rollback works on the world state and not on the blockchain.

6.4 Experiment 4: Test for Immutability of Ledger

The experiments that were performed till now were related to channels and chaincodes. In this experiment, we will focus on the immutability of blockchain component of ledger and also try to change the world state without invoking any chaincode transactions. This experiment will help us to understand how strong is the immutability property of blockchains compared to durability in ACID transactions.

6.4.1 Modifying the World State

In this experiment we will create a blockchain network with two peers, **PEER0** belonging to **Organization1** and **PEER2** belonging to **Organization2**. Both the peers will be part of a single channel **mychannel** and endorsement policy will be set to *OR ('Org1MSP.member','Org2MSP.member')*.

This experiment is based on the **marbles** example designed by the Hyperledger Fabric community [Com17a]. We will use the same setting to test the world state with couchDB container.

Steps to perform the experiment and the corresponding outcome for each step can be outlined as follows:

1. Joined both the peers, **PEER0** and **PEER1** to the channel **mychannel**.

   Outcome: The world state is populated in a separate container and can be accessed in the browser through the port specified in the docker-compose file. However, it is important to note that world state is only visible when the peers join the channel.

2. An invoke transaction is submitted from **PEER0** that creates a marble, **marble1** `{name:"marble1", size:"20", color:"blue", owner:"vikas"}`

   Outcome: The created marble can be seen for each of the peers in their respective containers. At this stage, all the peers have consistent copy of data.

3. The value for **color** field of **marble1** is changed to **green** by manually modifying the object in the couchDB host for **PEER0**.

   Outcome: A query for **marble1** on **PEER0** gave the manually modified value with **color** as **green** while a query on **PEER2** gave the value for **color** as **blue** that is the correct one. This shows that the peers are in inconsistent state.

4. There is a function in the **marbles** chaincode called as **transferMarblesBasedOnColor** that checks the color of the marble and changes its owner. Called this method from **PEER0** to change the owner to ”ghareeb”for marbles of color **green** (this is the modified color value on PEER0, and PEER2 holds the right value for color as blue).

   Outcome: There is no change in owner on any of the peers even when the **PEER0** had manually modified color value as **green**.

5. Called the same function **transferMarblesBasedOnColor** on **PEER0** but now the color value was blue (that is the right one) and new owner as ”ghareeb”.

   Outcome: Though the owner was changed to ”ghareeb”, the color field also got modified to green (this is wrong value) on both the peers. The data was consistent on both the peers but the color field was incorrect. So instead of `{name:"marble1", size:"20", color:"blue", owner:"ghareeb"}`, it showed `{name:"marble1", size:"20", color:"green", owner:"ghareeb"}`

Conclusion of Results

1. The above experiment showed that a peer’s world state can be accessed and modified leading to an inconsistent state on that peer. Authentication for the couchDB container should be provided so that unauthorized changes in world state can be avoided. This can be achieved by creating username and password for the admin account.

2. It was seen that the malicious peer’s data can affect the data of all the peers in the blockchain and lead to an inconsistent state. The solution for this would be to use a stronger endorsement policy requiring endorsements from multiple peers of all the organizations. Care should be taken to install chaincode on all the endorsing peers (endorsement is generated by executing a transaction and transactions can only be executed by the chaincode).

6.4.2 Modifying the Peer Blockchain File

In this experiment, we will try to modify the blockchain file present on the peer and see its effect on query and invoke transactions. The blockchain on a peer container is located at **var/hyperledger/production/ledgersData/chains/chains/mychannel**. Here mychannel is the name of the channel. We will modify the blockchain file for a peer and see its effect.

The same example of marbles [Com17a] can be used as discussed above. The steps for the experiment can be outlined as follows:

1. Joined **PEER0** to the channel, **mychannel** and created a marble object **marble1** with fields as `{name:"marble1", size:"20", color:"blue", owner:"vikas"}`

   Outcome: The **marble1** object was created and after querying fetched the right results. There is a method called **getHistoryForMarble** that queries the blockchain for the transaction ids related to the marble name. This method also returned the right transaction id.

2. The blockchain file on **PEER0** was located at `var/hyperledger/production/..` and was copied on the desktop to view its content. Then a new transaction was submitted for **marble2** on the same peer with fields as `{name:"marble2", size:"70", color:"red", owner:"ghareeb"}`.

   Outcome: The data was consistent and both the marbles object were visible in the couchDB container and the query also fetched right results.

3. Then copied the blockchain file with the new transaction for **marble2** on the desktop. Compared the contents of the two blockfiles. Removed the recently added block from new file that had the transaction for **marble2** and saved this file in the **PEER0** container. Queried the world state for the **marble2** and the blockchain with the method **getHistoryForMarble** for **marble2**.

   Outcome: It was found that a query on world state fetched the right result for **marble2**. However, **getHistoryForMarble** that queries the blockchain for marble2 history gave an endorsement failure error. This demonstrates that the blockchain on **PEER0** has been tampered with.

4. Joined a new peer, **PEER2** to the channel and executed the method **getHistoryForMarble**.

   Outcome: The query executed without errors and contained the transaction history for **marble1** and **marble2** executed separately. This indicates that when a new peer joins the channel, its blockchain contains the write non-tampered blocks.

Conclusion of Results

1. Modifying the blockchain on a peer can be detected and the malicious peer can be removed from the network.
2. The queries on keys are executed only on the world state and hence, it is decoupled from the blockchain for queries. Even if the blockchain has been tampered, the world state still contains the right values.
3. A new joining peer will never get the tampered blocks and will always contain consistent data.

6.4.3 Modifying the Peer and Orderer Blockchain and then Joining a New Peer

In this subsection, we will modify the blockchain for two peers, **PEER0** and **PEER2** on the channel, then we will modify the orderer blockchain and then join two new peers, **PEER1** belonging to **Org1** and **PEER3** belonging to **Org2**. The same example of **marbles** as in the previous subsections will be used [Com17a].

Listing 6.16 shows the different objects that will be created in separate transactions in the network. In the remaining part, we will refer to the objects with their **name** fields. In total five invoke transactions will be executed on the network at different times after modifying the blockchain file.

The different steps in the experiment and their corresponding analysis can be highlighted as follows :

Listing 6.16 The different objects created in the experiment to test the immutability of ledger.

```
marble1 {name:"marble1", color: "blue", owner: "vikas", size:"70"}
marble2 {name:"marble2", color: "green", owner: "ghareeb", size:"70"}
marble3 {name:"marble3", color: "red", owner: "jason", size:"70"}
marble4 {name:"marble4", color: "blue", owner: "bhuppi", size:"70"}
marble5 {name:"marble5", color: "blue", owner: "romeet", size:"70"}
```

1. Joined **PEER0** and **PEER2** to the channel **mychannel**. Submitted three transactions for **marble1**, **marble2** and **marble3** respectively.

   Outcome: The three transactions were visible on both the peers and query on the blockchain and world state yielded correct results.

2. Removed the block that contained the transaction for **marble3** from the blockchain file of **PEER0**. The blockchain file is located at `var/hyperledger/production/ledgersData/...`

   Outcome: Execution of method **getHistoryForMarble** on **PEER0** for **marble1** and **marble2** yielded correct results. However, execution of this method for **marble3** yielded endorsement failure error. Verified it again by running the method for all the three marbles on **PEER2** which yielded right results. Thus, the **PEER0** blockchain file has been tampered can be identified.

3. Created another transaction for a new marble, **marble4**.

   Outcome: The world state for the four marbles showed correct data in the couchDB container for both the peers. However, the **getHistoryForMarble** method on **PEER0** for **marble4** gave the endorsement failure error.

4. Removed the block for the transaction containing **marble4** from **PEER2**.

   Outcome:The same behaviour of endorsement failure was observed here on **PEER2** after the execution of the method **getHistoryForMarble** for **marble4**.

5. Submitted another transaction that created a new marble, **marble5** on the network.

   Outcome: The transactions was successful an the couchDB container for both the peers showed five different marble objects.

6. Removed the block containing the transaction for **marble5** from the orderer blockchain file. The orderer blockchain file is present at `var/hyperledger/production/orderer/...` Joined peer **PEER1** to the channel.

   Outcome Initially when the peer, **PEER1** joined the channel, it contained only three marble objects, **marble1**, **marble2** and **marble3**. However, the blockchain got synced after a considerable amount of time and showed all the five marble objects. Execution of the method **getHistoryForMarble** for all the five marbles yielded the right results.

7. Joined peer **PEER3** to the channel.

   Outcome: Initially it showed only four marble objects, **marble1**, **marble2**, **marble3** and **marble4** but eventually synced to all the five objects. Execution of the method **getHistory-ForMarble** for all the five marbles yielded the right results.

Conclusion of Results

1. It can be seen that initially the newly joining peers were not consistent, but eventually they synced their blockchain files and contained the right blocks even after the orderer and the peer blockchain files were modified.
2. This leads us to believe that any tampering with the blockchain files can be easily caught by the system and the system tries to reach a consistent state eventually.
3. The ledger can, thus, be considered as tamper-resistant.
4. The experiment can be further extended to test for immutability by adopting different approaches.

7 Conclusion

In this chapter we will finally conclude the entire thesis. In the first part we will discuss the summary of the tasks done and the second part will focus on the future work that can be done to further study the blockchain networks.

Summary

The aim of this thesis was to study the transaction characteristics of permissioned blockchain network. For the purpose of the study, we used Hyperledger Fabric that contains a highly modular architectural pattern. We have seen the various components like *chaincodes*, *channels*, *endorsement policies*, and the *consensus protocols* in a permissioned blockchain network. The transactional properties are affected by these components and hence, importance was given to create strong modular components.

The first part of the thesis mainly focused on the research aspects related to consensus algorithms, comparing the transactional properties of permissioned blockchain networks with that of the permissionless networks and also to ACID transactions. The later part was mainly focused on how a permissioned blockchain works with all the components and the experiments that helped us to understand certain important transactional aspects.

It can be concluded that a blockchain system has an tamper-proof ledger and any modifications in the blockchain does not affect newly added peers. However, a modification to the world state and without a strong endorsement policy can lead to unwanted data being committed to the blockchain and the world state. Also writes on different channels can be achieved in business transactions by implementing an intermediate software layer that mimics a 2-PC transaction monitor where a large transaction consisting of multiple sub-transactions is made **atomic**.

Future Work

A blockchain network can have multiple organizations adopting different blockchain frameworks and hence, it becomes necessary to adopt a dynamic solution for blockchain networks. Further study can be done to understand the interaction of different blockchain networks like Ethereum and Hyperledger Fabric that can make the blockchain environment more interesting. Work can be done to study the transactional properties of business transactions in mixed blockchain environment.

Also the immutability of the blockchain can be made more stronger by implementing compulsory strong endorsement policies and not leaving it to the disposal of the developer. A huge problem of networks like Hyperledger Fabric is the certificate authority which is currently a single node (therefore representing a single point of failure on the network) and this can be researched further.

Bibliography

[ABB+18] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevicha, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, J. Yellick. "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains". In: (Apr. 2018). DOI: arXiv:1801.10228v2[cs.DC] (cit. on pp. 18, 21, 22, 24, 27, 28, 32, 33, 38, 41–47, 52, 54, 55).

[CGR11] C. Cachin, R. Guerraoui, L. E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2011 (cit. on p. 44).

[CNS15] M. Crosby, P. Nachiappan, V. K. Sanjeev Verma. "BlockChain Technology Beyond Bitcoin". In: (Oct. 2015). DOI: 2015.10.16. URL: https://scet.berkeley.edu/wp-content/uploads/BlockchainPaper.pdf (cit. on p. 17).

[Com17a] H. F. Community. *Building your first network*. 2017. URL: https://hyperledger-fabric.readthedocs.io/en/v1.1.0-alpha/build_network.html (visited on 10/19/2018) (cit. on pp. 93–95).

[Com17b] H. F. Community. *chaincode_example02*. 2017. URL: https://github.com/hyperledger/fabric/blob/release-1.3/examples/chaincode/go/example02/chaincode.go (visited on 10/17/2018) (cit. on p. 77).

[Com17c] H. F. Community. *chaincode_example05*. 2017. URL: https://github.com/hyperledger/fabric/blob/release-1.3/examples/chaincode/go/example05/chaincode.go (visited on 10/17/2018) (cit. on p. 77).

[Com17d] H. F. Community. *fabcar sample chaincode*. 2017. URL: https://github.com/hyperledger/fabric-samples/blob/release/chaincode/fabcar/fabcar.go (visited on 10/17/2018) (cit. on p. 66).

[Com17e] H. S. Community. *PoET 1.0 Specification*. 2017. URL: https://sawtooth.hyperledger.org/docs/core/releases/1.0/architecture/poet.html (visited on 10/24/2018) (cit. on p. 22).

[Cou18] CouchDB. *Apache CouchDB*. 2018 (cit. on p. 33).

[CV17] C. Cachin, M. Vukolić. "Blockchain Consensus Protocols in the Wild". In: (2017). DOI: arXiv:1707.01873[cs.DC]. URL: https://arxiv.org/abs/1707.01873 (cit. on pp. 22, 23, 38, 39).

[Eth] Ethereum.org. *Proof of Stake FAQs*. URL: https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQs (visited on 10/24/2018) (cit. on p. 18).

[Fab18a] H. Fabric. *Hyperledger Fabric architecture reference*. 2018. URL: https://hyperledger-fabric.readthedocs.io/en/latest/arch-deep-dive.html#swimlane (visited on 10/08/2018) (cit. on p. 45).

[Fab18b] H. Fabric. *Hyperledger Fabric Chaincode Tutorials*. 2018. URL: https://hyperledger-fabric.readthedocs.io/en/latest/chaincode.html?highlight=chaincode (visited on 10/06/2018) (cit. on pp. 27, 53, 66).

[Fab18c] H. Fabric. *Hyperledger Fabric Channels*. 2018. URL: https://hyperledger-fabric.readthedocs.io/en/latest/channels.html?highlight=channels (visited on 10/06/2018) (cit. on pp. 30, 31, 53, 90).

[Fab18d] H. Fabric. *Hyperledger Fabric Endorsement Policies*. 2018. URL: https://hyperledger-fabric.readthedocs.io/en/latest/endorsement-policies.html?highlight=ENDORSEMENT%5C%20POLICY (visited on 10/06/2018) (cit. on pp. 28, 29).

[Fab18e] H. Fabric. *Hyperledger Fabric Ledgers*. 2018. URL: https://hyperledger-fabric.readthedocs.io/en/latest/ledger/ledger.html?highlight=LEDGER (visited on 10/06/2018) (cit. on pp. 31–33, 48).

[Fab18f] H. Fabric. *Key concepts: Identitty*. 2018. URL: https://hyperledger-fabric.readthedocs.io/en/latest/identity/identity.html (visited on 10/06/2018) (cit. on p. 25).

[Fab18g] H. Fabric. *Key concepts: Membership*. 2018. URL: https://hyperledger-fabric.readthedocs.io/en/latest/membership/membership.html (visited on 10/06/2018) (cit. on pp. 25–27, 30).

[FF18] T. M. FERNÁNDEZ-CARAMÉS, P. FRAGA-LAMAS. "A Review on the Use of Blockchain for the Internet of Things". In: (May 2018). DOI: 10.1109/ACCESS.2018.2842685. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8370027 (cit. on p. 18).

[Fou18] T. L. Foundation. *Hyperledger Fabric*. 2018. URL: https://www.hyperledger.org/projects/fabric (visited on 10/05/2018) (cit. on p. 21).

[IV17] R. G. IV. *Explaining How Proof of Stake, Proof of Work, Hashing and Blockchain Work Together*. 2017. URL: https://medium.com/@robertgreenfieldiv/explaining-proof-of-stake-f1eae6feb26f (visited on 10/24/2018) (cit. on p. 22).

[Kaf18] A. Kafka. *Apache Kafka: A distributed streaming platform*. 2018. URL: https://kafka.apache.org/ (visited on 10/05/2018) (cit. on p. 23).

[Lev18] LevelDB. *A light-weight, single-purpose library for persistence with bindings to many platforms*. 2018. URL: http://leveldb.org/ (visited on 10/06/2018) (cit. on p. 33).

[Mas16] N. Massart. *Why do smart contract languages need to be deterministic?* 2016. URL: https://ethereum.stackexchange.com/questions/3557/why-do-smart-contract-languages-need-to-be-deterministic (visited on 10/08/2018) (cit. on p. 42).

[Mica] Microsoft.com. *Compensating Transaction pattern*. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction (visited on 10/24/2018) (cit. on p. 55).

[Micb] Microsoft.com. *ROLLBACK TRANSACTION*. URL: https://docs.microsoft.com/en-us/sql/t-sql/language-elements/rollback-transaction-transact-sql?view=sql-server-2017 (visited on 10/24/2018) (cit. on p. 55).

[MOE09] B. Medjahed, M. Ouzzani, A. Elmagarmid. "Generalization of ACID Properties". In: (2009). Purdue University, Cyber Center Publications. URL: https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1107&context=ccpubs (cit. on pp. 17, 46–48).

[Mon17] Monegraph. *Monegraph*. 2017. URL: https://monegraph.com/ (visited on 10/10/2018) (cit. on p. 37).

[Nak] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. URL: https://bitcoin.org/bitcoin.pdf (visited on 10/24/2018) (cit. on p. 38).

[Ngu17] B. Nguyen. *Chaincode calling chaincode*. An issue rased on JIRA. 2017. URL: https://jira.hyperledger.org/browse/FAB-1788 (visited on 10/09/2018) (cit. on p. 54).

[Ora17] Oracle. *Transaction Management*. 2017. URL: https://docs.oracle.com/cd/B19306_01/server.102/b14220/transact.htm (visited on 10/09/2018) (cit. on p. 55).

[Org18] H. Org. *Introduction to Hyperledger Fabric*. 2018. URL: https://hyperledger-fabric.readthedocs.io/en/latest/whatis.html (visited on 10/05/2018) (cit. on p. 22).

[Pro17] Provenance. *Provenance*. 2017. URL: https://www.provenance.org/ (visited on 10/10/2018) (cit. on p. 37).

[Saw17] H. Sawtooth. *Hyperledger Sawtooth- Introduction*. 2017. URL: https://sawtooth.hyperledger.org/docs/core/releases/latest/introduction.html (visited on 10/10/2018) (cit. on pp. 33–35).

[Tai17] S. Tai. "Continuous, Trustless, and Fair: Changing Priorities in Services Computing". In: (2017). URL: http://www.ise.tu-berlin.de/fileadmin/fg308/publications/2017/2017-tai-continuous-trustless-fair.pdf (cit. on p. 49).

[TEK17] S. Tai, J. Eberhardt, M. Klems. "Not ACID, not BASE, but SALT A Transaction Processing Perspective on Blockchains". In: (2017). Information Systems Engineering, Technische Universitat Berlin. URL: http://www.ise.tu-berlin.de/fileadmin/fg308/publications/2017/2017-tai-eberhardt-klems-SALT.pdf (cit. on pp. 17, 18, 37, 49–51).

[WG18a] H. A. W. G. (WG). "Hyperledger Architecture, Volume II Smart Contracts". In: (2018). URL: https://www.hyperledger.org/wp-content/uploads/2018/04/Hyperledger_Arch_WG_Paper_2_SmartContracts.pdf (cit. on pp. 21, 28, 47, 48).

[WG18b] H. A. W. G. (WG). "Introduction to Hyperledger Business Blockchain Design Philosophy and Consensus". In: (2018). URL: https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf (cit. on pp. 21–25, 47).

[WOO16] D. G. WOOD. "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER". In: (2016). EIP-150 REVISION. URL: https://gavwood.com/paper.pdf (cit. on p. 47).

All links were last followed on October 25, 2018

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature