Distributed System Project: Hyperledger Fabric
Final Report

Francesco Montelli
francesco.montelli@studio.unibo.it

March 26, 2020

Contents

Chapter 1

Abstract

Hyperledger Fabric is a permissioned Distributed Ledger Technology (DLT) designed for use in enterprise contexts.

Fabric differs from other blockchain technologies for its peculiar architecture which provides for the segmentation of the network into different "channels" that allow a subset of the nodes to communicate privately. Other saving points can be identified in the introduction of the "execute-order-validate" architecture for the management of transactions and in the rich policy system that allows you to efficiently model the requirements of organizations that intend to manage their relationships with the support of blockchain technology

The purpose of this document is to be an introduction to Fabric. The aim is to present its basic architecture and allow the development of the first applications with this technology.

Chapter 2

Introduction

2.1 Blockchain

2.1.1 What is a blockchain?

A Blockchain is distributed ledger composed by list of *blocks* containing *transactions* linked together using cryptography. A transaction can be defined as a change made by a user on some shared data.

The ledger is described as distributed because it is replicated across many nodes linked in a peer-to-peer network, each node collaborates to ledger's maintenance. Since the various nodes must have the same version of the blockchain, consensus algorithms are used to ensure that everyone agrees on the legitimacy of transactions and their order.

Blockchains can be used as a backbone for a wide range of applications like proprietary management, voting system, and supply chain management. Despite the different use cases the main goal is to have a decentralized way for securely storing data, enabling a group of mutually untrusted parties to cooperate.

2.1.2 Data structure

In a blockchain, each block stores a hash pointer to the previous block in the chain, figure 2.1 shows an example of this structure. Thanks to those pointers we can always reach the previous block and be sure that it's content haven't tampered because any data alteration will result in a hash variation.

The most recent block in the chain is called "head", to append new data is necessary to create a new block that contains a hash pointer to the most recent block.

To tamper the blockchain, an attacker should modify every block in the chain to keep the hash pointers consistent. An operation of this kind would be noticed by other users who would deny consent to that blockchain

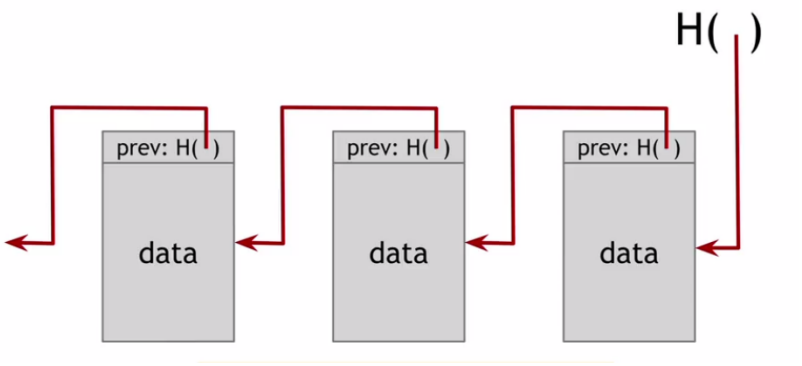

Figure 2.1: In a blockchain each block is linked to the previous using an hash pointer.

2.1.3 Machine Status Replication (MSR)

BTCs are a good example of machine status replication. The main idea behind MSR is executing the same state machine over multiple independent processors, in parallel, to achieve:

- Fault tolerance
- Availability and reactivity
- data and software replication

In this way, a network of replicas is built.

A solution like MRS necessarly implies working with a Distributed System, this introduces some challenges:

- **Network problems**: messages may be lost, corrupted, reordered or duplicated while in transit. Each replica may perceive a different view on the system events. It is necessary to use robust transport protocols, like TCP, and hash functions to verify data integrity
- **Time related problems**: there is no global time shared by the peers
- **CAP theorem**: some replica may fail and won't be able to communicate with the others. One common solution in the BTC context is to choose Availability and Partition-resistance and, as in distributed databases, relay on *eventual consistency*[1]
- **Byzantine faults**: a Byzantine fault is any fault presenting different symptoms to different observers. To avoid catastrophic failure of the system, the system's actors must agree on a "strategy" using a consensus algorithm.

[1] In a fine time things will be consistent.

2.1.4 Consensus

Due to the value possibly stored in the ledger, bad actors may have economic incentives to try to cause faults, for example, a replica may try to alter the blocks' order or exclude particular events. The presence of a replica that operates in this way may prevent to reach consistency, indefinitely prevent agreements.

Distributed DBs and blockchains usually employ some type of BFT algorithm to prevent Byzantine failures. A BFT consensus algorithm aims to guarantee eventual consistency even if some replicas that behave incorrectly.

Reach a distributed consensus is **impossible** under **any** of the following conditions:

- the underling network is asynchronous causing messages to be arbitrarily re-order or delayed.
- the amount f of Byzantine-faulty replicas is $f \geq \frac{1}{3}N$, where N is the total number of replicas.

2.1.5 Blockchain and identity

Based on how identities are managed, two main types of BYC can be identified

- **Public or permission*less* blockchains**: in this kind of blockchains there isn't a trusted third party which regulates the access, no restrictions are made on who can interact. Examples are Bitcoin and Ethereum
- **Private or permission*ed* blockchains**: these blockchains are controlled by a third-party who regulates the access. Access can also be managed via a fine-grained policy system, for example by defining permissions for single users or groups. Examples are Hyperledger Fabric and Corda.

Like many other distributed systems BTCs assign an identity to entities for accountability purposes. In permissionless BTCs, everyone can generate identity whereas in permissioned BTCs there is a Certification Authority that has total control over the entities and their interaction capability with the system.

Most BTCs use asymmetric cryptographic keys to face up identity and trust issues. Each member of the network has a pair of keys, a public one that can be shared with everyone, and a private one that must be kept safe.

Before writing any data to the blockchain a user must sign it with its private key, every node/user that receives the signed block can verify the signature with the corresponding public key verifying both integrity and authenticity. In private blockchains, the public key is a digital certificate issued by the CA.

2.2 Smart Contract enabled BTCs

2.2.1 Introduction

A Smart Contract (SC) is the digitized version of the classical contract, but hosted on a blockchain that:

- regulates the interaction between parties while enforcing the terms of the contract
- clarify the implication for the transactions
- minimize exceptions
- minimize the presence of intermediaries

2.2.2 Functioning

Usually at least two operation can be performed by clients:

- **deploy**: the SC code is installed on a network peer via a *deployment transaction*. Once the installation is confirmed, you can assume that an SC instance is running on *all* network peers. A contract on a network is identified by a unique identifier.
- **invoke**: a user can trigger a SC by submitting an *invocation transaction* to the SC's address. Eventually, every replica will receive the transaction and execute it. If the computation terminates without exceptions any produced side effect becomes part of the new intermediate state[2]. If the consensus algorithm confirms the block containing the new intermediate state the invoked computation can be considered executed.

2.2.3 Issues

The most common implementations of contracts on blockchain suffer from some problems:

- **No privacy or secrets**: every information ever published on the blockchain stays on the blockchain. The private state of a smart contract is visible to everyone.
- **Poor randomness**: real randomness cannot be employed because the replica would diverge, preventing consensus to be reached
- **SC inter-communication**: SCs are objects communicating using method calls, the control flow originating from a user may traverse more than one SC
- **Immutable code**: since the code is stored on the blockchain it can't be changed, buggy/fraudulent contracts cannot be fixed/removed
- **Lack of pro-activeness**: SC is a purely reactive computational event, they cannot schedule or postpone computations
- **Resource usage**: every node of the network must execute every transaction

[2]Otherwise side effect are dropped and the new intermediate state coincides with the previous one

2.2.4 Order-Execute architecture

Almost all BTCs use an architecture called *order-execute*, this means that transaction is first ordered using the consensus algorithm and then executed on all peer sequentially. This solution is simple but limits the throughput of the network and does not permit transactions to be confidential.

2.3 Hyperledger Fabric (HLF)

Hyperledger Fabric is an open-source enterprise-grade permissioned distributed ledger technology (DLT) platform designed to be used in an enterprise context.

Fabric allows components, such as consensus and membership services, to be plug-and-play. Another important difference from other BTCs is that smart contracts are written in general-purpose programming languages, this is a strong advantage for enterprises since they do not have to acquire new resources to develop their project. Contracts run on isolated Docker containers, establishing a further security layer since this permits to segregate execution from other peers functionality.

Chapter 3

HLF's architecture

In this chapter we introduce the HLF architecture and the *execute-order-validate* that regulates its transaction flow.

3.1 Overview

3.1.1 Application Structure

Fabric can execute distributed applications written in general-purpose programming languages leveraging a distributed ledger to keep track of transaction's order.

A distributed application for Fabric consists of two parts:

- **A smart contract**: describe the transaction logic and its executed during the execution phase. Special chaincodes exists for managing the blockchain system, those are collectively called *system chaincodes*
- **An endorsement policy**: define the smallest set of organizations that are required to endorse a transaction for it to be valid. To endorse, an organization's endorsing peer needs to run the smart contract associated with the transaction and sign its outcome.

3.1.2 Network architecture

A Fabric network consists of a set of *nodes* connected together, see figure 3.1. Since Fabric is a permissioned system every node that forms the network must have an identity issued by a trustworthy authority, in this case by a modular *membership service provides*.

In Fabric a peer can fill one of these roles:

- **Clients**: submit transactions proposal for execution, help orchestrate the execution phase and broadcasts transaction for ordering.

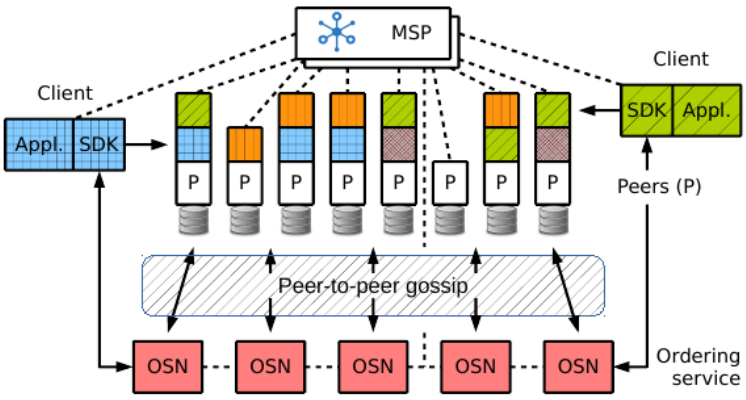

Figure 3.1: A Fabrics network running multiple chaincodes, selectively installed on peers.

- **Peers**: execute and validate transactions. All peers maintain the blockchain ledger, an append-only hash chain, and the *world state*, a succinct representation of the latest ledger state. Note that not all peers need to execute instructions, only the peers indicated in the endorsement policy need to execute the transaction.
- **Ordering Service Node (OSN)**: are the nodes that collectively form the ordering service. The ordering service establishes the *total order* of all transactions. Orderers are entirely unaware of the application state and do not participate in execution nor the validation of transactions.
- **Membership Service Provider (MSP)**: abstracts away all cryptographic mechanisms and protocols behind issuing certificates, validating certificates and user authentication. An MSP may define the notion of identity, and the rules by which those identities are governed (identity validation) and authenticated (signature generation and verification). Multiple MSPs can be defined in the same netwtork.

Fabric supports multiple blockchains over the same network, each blockchain is called *channel* and can have different peers as its members. Also notice how it's possible to install different chaincodes on different peers.

3.2 execute-order-validate

Fabric introduces the *execute-order-validate* architecture, illustrated in 3.2.

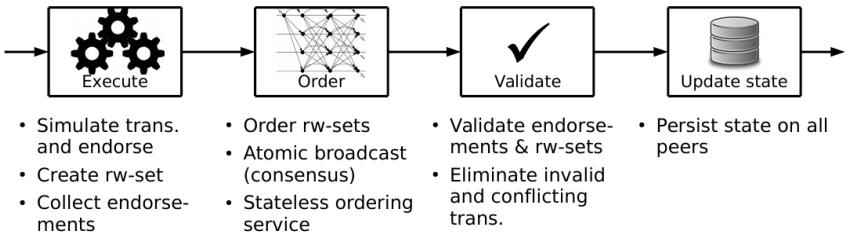

Figure 3.2: The execute-order-validate architecture of Fabric.

3.2.1 Execute

In this phase clients sign and send the *transaction proposal* to one ore more endorser as specified in the chaincode's endorsement policy. The transaction proposal contains:

- operation's name
- operation's parameter
- chaincode's identifier
- a nonce, used only once by each peer
- transaction identifier derived from client identifier and nonce

The endorses *simulates* the proposal by executing the operation specified in the chaincode inside a Docker container isolated from the endorser's main process. This simulation is run against the endorser's current state which is maintained as a versioned key-value store. As a result of the simulation, each endorser produces a *proposal response* which is sent back to the client. This response is signed by the endorser and contains the *readset* and the *writeset* which are, respectively, the list of keys that have been read and written. Each key is associated with a version number which will then be used in the verification phase. The client collects these responses until the endorsement policy is satisfied. After this step, the transaction is built and sent to the ordering service.

3.2.2 Order

The ordering phase establishes a total order on all submitted transactions per channel. Multiple transactions are batched into *blocks* that are broadcasted over the channel. The delivery is supported by a gossip protocol. The ordering service mustn't maintain any state of the blockchain and neither validates nor execute transactions, this is paramount for separate execution and consensus. Each block contains a set of transactions; each transaction is associated with the endorsement message produced by the peers that performed that transaction.

3.2.3 Validate

When a block enters the validation phase, each transaction it contains undergoes two checks:

1. **Endorsement policy validation**: the transaction responses associated with each transaction are examined. If the endorsement policy is not met the transaction is marked as invalid.

2. **Read-write conflict check**: checks if the read-state of the transaction is compatible with the current world state. If the versions associated with the keys in the read set do not match the current state value the transaction is marked as invalid

Every transaction that passes these checks is marked as valid and its effects are applied, the block is then inserted into the peer's ledger. Note that this procedure will create blocks containing both valid and invalid transactions, this peculiarity can be useful when an audit is performed. Those checks are performed in parallel on every peer.

3.3 Transaction Flow

This section explains in detail the execute-order-validate architecture introduced in 3.2. In this scenario, we will work with two clients, A and B, who want to exchange goods. Each involved party maintains a peer in the network through which it send transactions and interact with the ledger.

3.3.1 Assumptions

This flow assumes that the channel which hosts the smart contract is set up and running. The smart contract contains the logic defining a set on transaction instructions and the agreed-upon price. An endorsement policy stating that both `peerA` and `peerB` must endorse any transaction has been set.

3.3.2 Step 1: Client A initiates a transaction

The SDK takes care of prepare a signed transaction proposal using ClientA's private key, this proposal is then sent to `peerA` and `peerB` who represent ClientA and ClientB. This forward is based on the chaincode endorsement policy which requires both organizations to endorse the transactions.

3.3.3 Step 2: Endorsing peers verify the signature and execute the transaction

Endorsers verify:

1. that the transaction proposal is well-formed

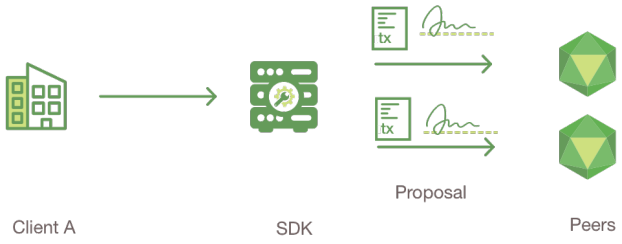

Figure 3.3: Client A initiates a transaction

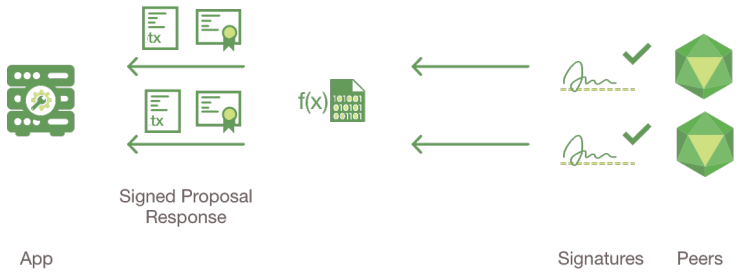

Figure 3.4: Endorsing peers verify the signature and execute the transaction

2. it has not been submitted already in the past
3. the signature is valid
4. the submitter is authorized to perform the requested operation

The endorsing peers take the transaction proposal input as an argument to invoke the chaincode's function which is executed against the current state database producing a transaction result that includes:

1. response value
2. read set
3. write set

No updates are made to the ledger at this point, read and write sets are only a list of accessed/modified keys. Those values are packaged along with the endorsing peer's signature as a "proposal response" and passed back to the SDK which parses the payload for the application.

3.3.4 Step 3: Proposal responses are inspected

The application verifies the endorsing peer signature and compares the responses to determine if the proposed responses are the same. If the transaction is a

Figure 3.5: Proposal responses are inspected

`query`, a transaction that only reads data, peers would typically not submit the transaction to the ordering service. If the client application wants to insert the transaction into the ledger it will verify if the endorsement policy has been fulfilled before submitting the transaction to the ordering services. More detail in the following subsections.

3.3.5 Step 4: Client assembles endorsement into a transaction

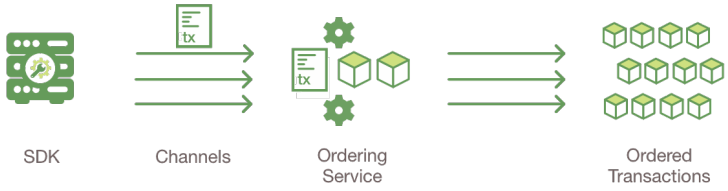

Figure 3.6: Client assembles endorsement into a transaction

The application broadcasts the transaction proposal and response to the ordering services. The transaction contains the read/write sets, the endorsing peers' signature, and the channel ID. The ordering service does not need to inspect the content of the transaction to perform the ordering, it's job is only to create blocks and distribute them to the peers. Peers also use a gossip protocol to facilitate block spread.

3.3.6 Step 5: Transaction is validated and committed

The blocks are delivered to all peers on the channel. The transactions within the block are validated to ensure endorsement policy is fulfilled and that there have been no changes to ledger state for read-set variables. Transactions in the block are tagged as being valid or invalid.

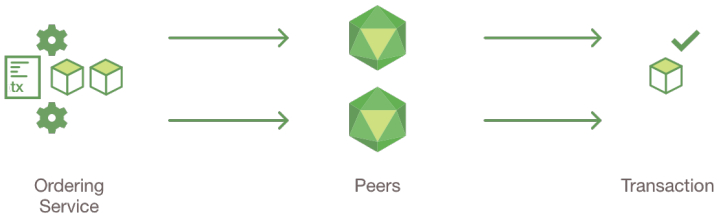

Figure 3.7: Transaction is validated and committed

3.3.7 Step 6: Ledger update

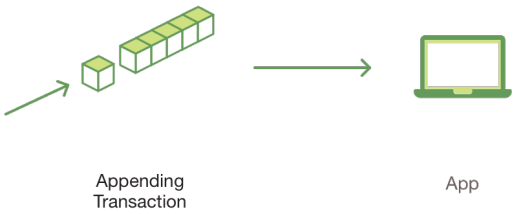

Figure 3.8: Ledger update

Each peer appends the new block to the channel's chain, for each valid transaction the write set are committed to current state database. An event is emitted by each peer to notify the client application that the transaction has been append to the chain.

Chapter 4

HLF's Main Components

4.1 Overview

In this section, HLF's main components are introduced. For more specific analysis refer to specific sections.

4.1.1 Assets

Assets can range from the tangible (real estate and hardware) to the intangible (contracts and intellectual property). Hyperledger Fabric provides the ability to modify assets using chaincode transactions.

4.1.2 Smart contracts

Smart contracts are the business model, they define common terms, data, rules, concept definitions, and processes. A smart contract can implement the governance rules for any type of business object, so that they can be automatically enforced when the smart contract is executed. Smart contracts functions execute against the ledger's current state database and are initiated through a transaction proposal.

4.1.3 Chaincode

Chaincode is a package used for install one ore more contracts. The terms "chaincode" and "smart contract" are often used interchangeably by the community

4.1.4 Ledger

The ledger is the sequenced, tamper-resistant record of all state transitions in Fabric. State transitions are the result of chaincode invocations submitted by participating parties. The ledger is composed of a blockchain that stores the

immutable sequence of old transaction records in blocks and a state database that keeps the current state. There is a ledger for each channel and each peer helps its maintained by storing a full copy of it.

4.1.5 Security and Membership Services

Every participant must have a known identity. PKI [1] is used to generate cryptographic certificates that are tied to organizations, network components, and end-user applications.

4.1.6 Channels

Each channel is defined by:

- members
- shared ledger
- applications
- ordering nodes

A channel can be shared across the entire network or can include only a specific set of participants. In the latter scenario, the participants would create a separate channel and thereby isolate their transactions and ledger. To handle scenarios that want to bridge the gap between total transparency and privacy, chaincode can be installed only on peers that need to access the asset states to perform reads and writes. In other words, if a chaincode is not installed on a peer, it will not be able to execute transactions but would sill maintain a copy of the ledger. If a subset of organization in a channel needs to keep some aspect of their transaction confidential a private data collection can be used to create a private database accessible only to the authorized subset. To further obfuscate the data values within the chaincode can be encrypted using AES before sending the transaction to the ordering service and append blocks to the ledger

4.1.7 Certificate Authority (CA)

Certificate Authority dispenses certificates to different actors. These certificates are digitally signed by the CA and bind together the actor's information with its public key. As a result, if one trusts the CA (and knows its public key), it can trust that the specific actor is bound to the public key included in the certificate.

4.2 Ledger

The ledger stores information about business objects; both current value and the history of the transaction that resulted in these current values. In HLF the

[1] Public Key Infrastructure

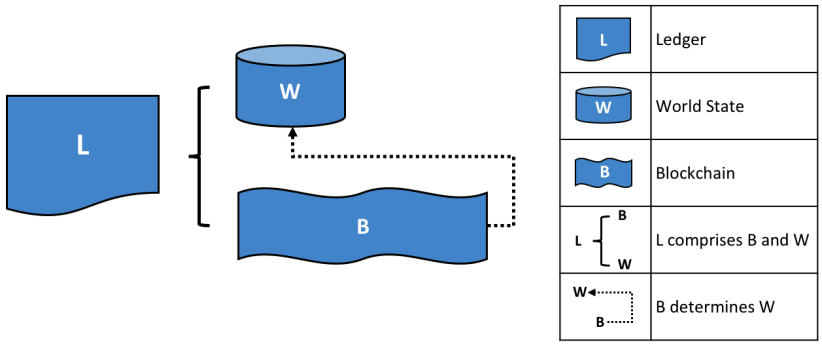

Figure 4.1: A Ledger comprises the blockchain and the world state

ledger is composed by two entity: **world state** and **blockchain**.

A full copy of the ledger is maintained on every peer inside the network, those copies are kept consistent through a process called *consensus*.

4.2.1 World state

The world state contains the current value of the attribute of a business object, this eliminates the need to traverse the blockchain to calculate the current value. World state is implemented as a database, this makes possible to use a rich set of operator for the storage and retrieval of states; different kind of DBs can be used to address different needs.

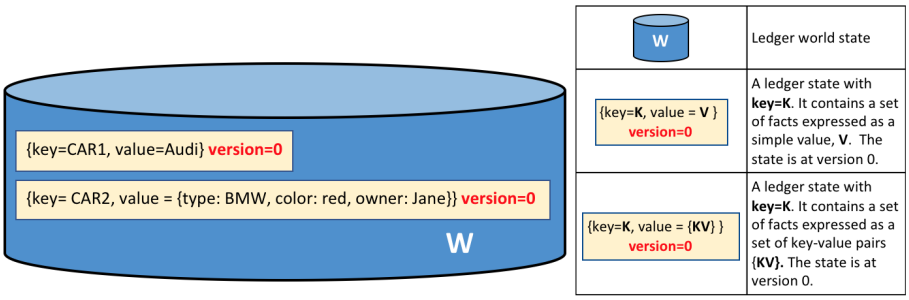

Figure 4.2: A world ledger with two state, each identified by is key. Notice how the value format is not required to the same for every state

SCs can interact with world state with a simple API that mainly consists

of three methods: **get**, **put**, and **delete**. Every entry also maintains a version number which is changed every time its modified. This value is used in the validation validation step, more detail in 3.2.3

Finally is worth noticing that the world state can be regenerated any time from the blockchain. This is very useful when a new peer is created or fails abnormally and the world state needs to be regenerated.

4.2.2 Blockchain

The blockchain is a historical record of the facts about the objects.

The blockchain is structured as a sequential log of interlinked blocks where each block contains a sequence of transactions determined by the ordering service.

Each block's header includes a hash of the block's transaction and a hash of the prior block's header, in this way all the blocks are cryptographically linked.

The blockchain is implemented as a file to facilitate the append operation

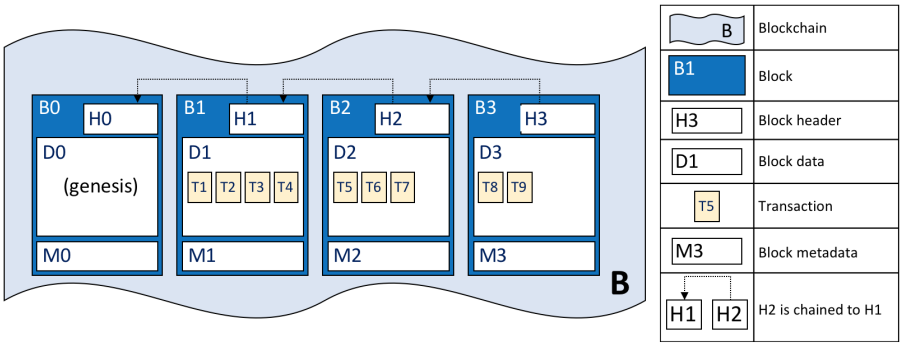

Figure 4.3: A blockchain B containing block from B0 to B3.

The first block of a blockchain is called **genesis block**. This block does not contain any user transaction but is used to store the initial channel configuration.

4.2.3 Blocks

A block consists in three sections:

- Block header
  - Block Number: integer starting at 0, incremented by 1 after a new block is appended to the chain.
  - Current Block Hash: the hash of all the transaction contained in the current block

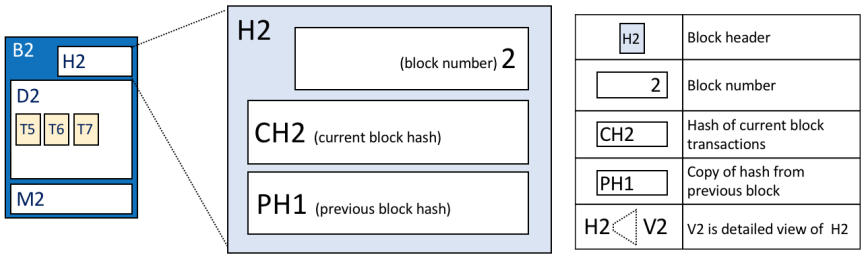

Figure 4.4: Block's structure.

  - Previous Block Header Hash: the hash from the previous block header.

- Block Data: contains the transaction arranged in order.
- Block metadata: contains the certificate and signature of the block creator and a valid/invalid indicator for each transaction.

4.2.4 Transactions

In figure 4.5 we can see the main components of a transaction:

- The header captures the essential metadata of the transaction such as relevant chaincode and version information.
- Signature (S4) contains the signature created by the client application. This field is used to check transaction's legitimacy and integrity. This filed is generated with the issuer's private key.
- Proposal (P4) encodes the input parameters supplied by an application to the SC. When the SC runs this proposal provides a set of input which, in combination with the current world state, determines the new world state.
- Response (R4) captures the before and after values of the world state. It's the output of a SC, if the transaction is validated will be applied to the current world state.
- Endorsements (E4) is a list of signed transaction responses from each required organization. Each response includes a complete response.

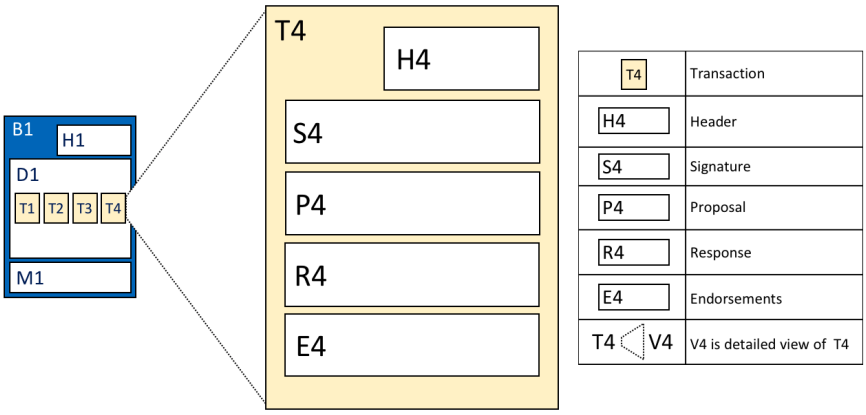

Figure 4.5: Block's structure.

4.2.5 Notes on namespaces

Each chaincode has its world state that is separate from all other chaincodes. A blockchain contains transactions from many different namespaces.

4.2.6 Notes on channels

Each channel has a completely separate ledger but, given the correct policy in place, chaincode on different channels can interact.

4.3 Smart Contract and Chaincode

SCs represent the business model that governs all the interactions between the network's components.

In this section we'll model the relation between two organizations, `ORG1`(Seller) and `ORG2`(buyer), that wants to manage a distribute register of cars. In order to do so they have defined a SC whom models the definition of the `car` object and the transactions, `query`, `transfer` and `update`, for its manipulation.

Figure 4.6 will illustrate the complete scenario.

4.3.1 Terminology

HLF uses the terms "smart contract" and "chaincode" interchangeably but, to be more precise, those terms are slightly different: SCs defines the transaction logic that controls the lifecycle of a business object while a chaincode is a package for ore or more SC and is used to deploy the contracts on the blockchain. Chaincodes commonly contain only one contract.

In figure 4.7 we can see some example of chaincodes containing different SCs. We can see a SC as a domain-specific program and a chaincode as a technical

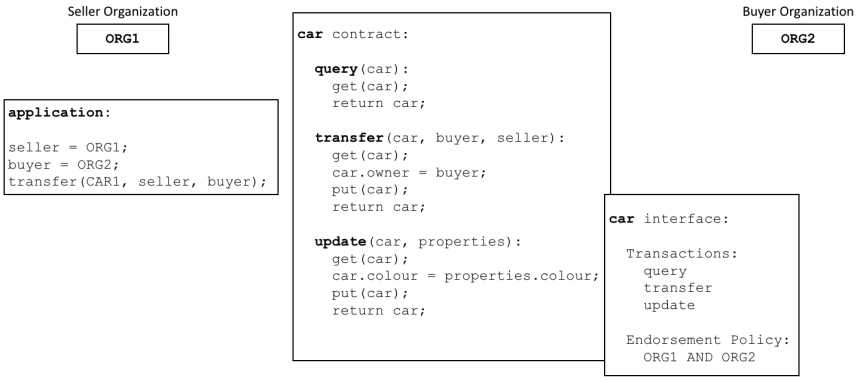

Figure 4.6: Smart contract example

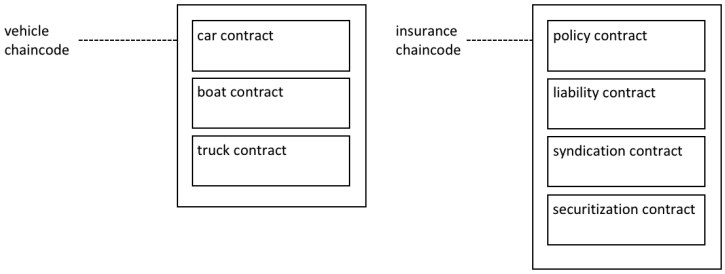

Figure 4.7: Example of chaincodes defining multiple contracts

container of a group of related smart contracts. Deploying a chaincode in a network makes all its SCs available to the organizations in that network, only administrators need to think about chaincode; everyone else can think in terms of SCs.

4.3.2 Endorsement policies

Associated with every chaincode there is an endorsement policy that applies to all the SCs defined within it. The endorsement policy are expressed as boolean expressions and are used to indicate which organizations must sign a transaction proposal to make it valid.

In figure 4.6 we can see that attached to the SC there is an interface that defines the structure of the contract and the endorsement policy. In the example a sign from both organizations is required, this means that both organizations need to execute the contract and produce a result.

Endorsement policies are what make HLF different from other BTCs like Ethereum or Bitcoin where the valid transactions can be generated from every

node of the network.Policies are designed to model real-world interactions and should be agreed before the chaincode is deployed on the channel but they can change over time.

4.3.3 Valid transactions

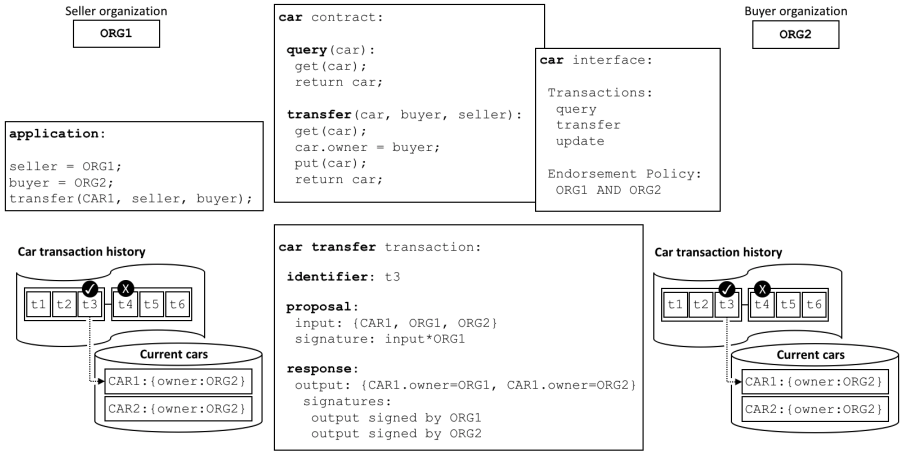

Figure 4.8: Note how different peers process transactions in the same way

Every transaction contained in blocks received by peers is validated in two steps:

1. Every peer checks if the transaction has been signed by a sufficient number of peers, as specified in the endorsement policy.

2. It is checked that the current value of the world state matches the read set of the transaction when it was signed by the endorsing peer nodes. This is required to ensure that there has been no intermediate update.

If a transaction passes both checks is marked as valid. All transactions are added to the blockchain history but only valid transaction results in an update in the ledger.

4.3.4 Install and define a chaincode

The fabric chaincode lifecycle is a process that allows multiple organizations to agree on how a chaincode will be operated before it becomes available on the channel. The lifecycle consists of four steps:

1. Package the chaincode

2. Install the chaincode on peers

3. Approve a chaincode definition for your organization. To enable the use of chaincode in a channel the majority of organization in that channel must approve the definition of said chaincode

4. Commit the chaincode definition to the channel. The commit transaction can be submitted by one organization once the required number of organizations have approved the definition

The chaincode definition is a struct that contains the parameters that govern how the chaincode operates. These parameters include the chaincode name, version, and the endorsement policy.

Each organization can make changes to a smart contract as long as it respects the definition and the overall behavior. This can be useful for implementing additional controls

4.4 Ordering Service and Consensus

4.4.1 Definition of ordering

In HLF transactions execution and ordering are performed by two different sets of peers to favor greater modularity of the system. Transactions ordering is handled by a set of peers (Orderers) called Ordering Service.

4.4.2 Ordering nodes and channel configuration

Orderers, in addition to their ordering role, also enforce basic access control for channels, restricting who can read/write data or configure them.

Transactions that have the purpose of configuring the channel must be performed by the peers that make up the ordering service since they are the ones who keep the system ledger. As for the other transactions, a block is issued and then forwarded to other peers, enabling them to verify if the work of the orderers has respected the channel's policy when performing their job.

4.4.3 Orderes and the transaction flow

Phase one: Proposal

In this phase, the ordering service will receive from a client application an endorsed transaction proposal. For a more in deep look into this phase refer to 3.3.2.

Phase two: Ordering and packaging transactions into blocks

In this phase received transactions are ordered and packaged into blocks that will then be distributed over the network.

The order of transactions within a block does not coincide with the order in which they arrived at the orderer since different orderers can receive the

transactions at a different time. Their ordering is strict and this order will then used by peers in the validation phase.

This strict ordering of transactions within blocks makes HLF different from other blockchains where the same transaction can be packaged into different blocks that compete to form a chain. In HFL blocks generated by the ordering service are final, once a transaction has been written to a block, its position in the ledger is immutably assured.

Phase three: Validation and Commit

This last phase involves the distribution and the subsequent validation of blocks. Each peer will validate distributed blocks independently, but in a deterministic fashion, ensuring that ledgers remain consistent.

4.4.4 Raft

Raft, the recommended ordering service implementation in Fabric 2.0, follows a "leader and followers" model where a leader node is elected and its decisions are replicated by the followers. This system can sustain a loss of nodes, including the leader nodes, as long there is a majority of ordering nodes. Raft is said to be "crash fault-tolerant", in other words, it can sustain the loss of some nodes.

Every channel runs on a separate instance of the raft protocol, this enables different channels to elect a different leader and allows further decentralization of the service in use cases where clusters are made up of ordering nodes controlled by different organizations.

Transactions received are automatically routed by the ordering node that receives the transaction to the current leader of that channel. This means that peers and applications can communicate with the ordering service without the need to know who is the leader.

4.5 Identity and MSP

Each actor on HLF has a digital identity issued by a trusted authority and encapsulated in an X.509 digital certificate, these identities determine the exact set of permissions over resources and access to information.

4.5.1 Authentication, Public keys, and Private Keys

Authentication and message integrity are important concepts in secure communications. Integrity means that the message's content cannot have been modified during transmission, authentication means that the creator's identity is assured.

Traditional authentication mechanisms rely on digital signatures that, as the name suggests, allow a party to digitally sign its messages. Digital signatures also provide guarantees on the integrity of the signed message.

Each party holds two cryptographically connected keys: a public key that is made widely available, and a private key that is used to produce digital

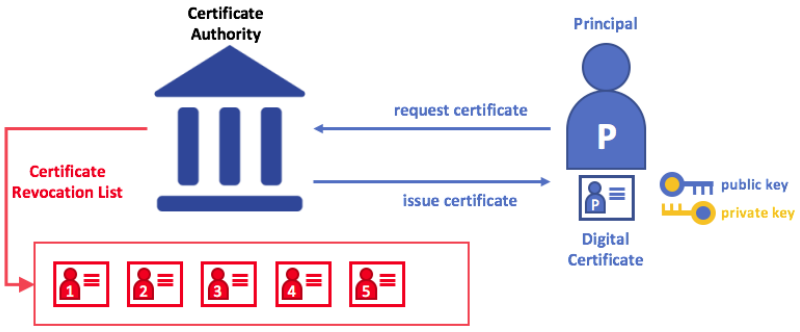

Figure 4.9: Main components and interaction

signatures on messages. Recipient of the messages can use the sender's public key to verify the origin and integrity of the message.

4.5.2 Digital Certificates

A digital certificate is a document which holds a set of attributes relating to the holder, in figure 4.11 we can see an example of such structure.

In addition to information about the user's identity and public key, digital certificates also contain a signature from the CA; this signature permits everyone to verify the correctness of the certificate using the public key of the CA.

4.5.3 Certificate Authorities

A CA dispenses certificates to different actors, these certificates are signed by the CA. Certificates can be widely disseminated as they do not include any private key.

In a blockchain setting, every actor who wishes to interact with the network needs an identity. In this setting CAs are used to define the members of an organization by providing actors with a verifiable identity.

CAs also maintain a list of revoked certificates. A revoked certificate is a certificate marked as invalid before its natural expiration. Certificates are revoked, for example, where a private key is leaked.

4.6 Applications

Applications can interact with a network by submitting a transaction to the ledger or query its content. The example contained in this section are written in Javascript but the logic is language independent.

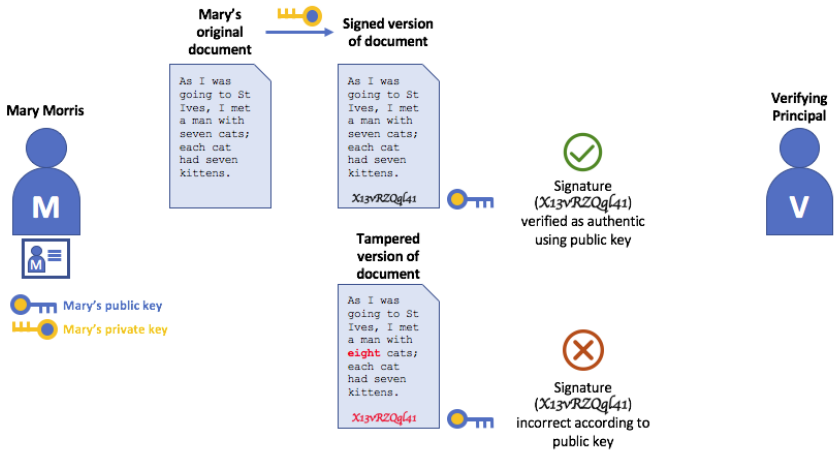

Figure 4.10: Signature process

4.6.1 Example scenario

PaperNet is a network that allows authorized participants to issue, buy and redeem commercial papers[2].

In figure 4.12 we can see the composition of the network. Currently, the network is used by six organizations that have a different role: MagnetoCorp can issue and redeem commercial paper, RateM can rate papers and the remaining organization can buy, sell and redeem papers.

4.6.2 Basic flow

In figure 4.13 we can see a simplified diagram of how an application invokes a smart contract.

An application has to follow six steps to submit a transaction:

1. Select an identity from a wallet
2. Connect to a gateway
3. Access the desired network
4. Construct a transaction request for a smart contract
5. Submit a transaction to the network
6. Process the response

[2]A Commercial paper is a money-market security issued (sold) by large corporations to obtain funds to meet short-term debt obligations (for example, payroll) and is backed only by an issuing bank or company promise to pay the face amount on the maturity date specified on the note.

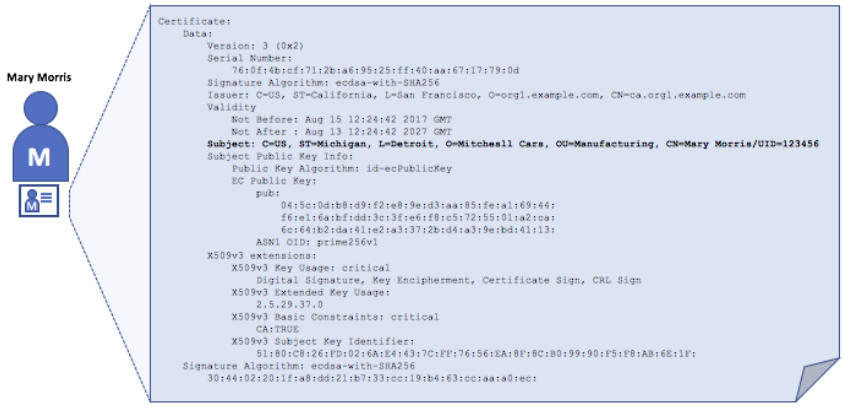

Figure 4.11: Example of digital certificate

4.6.3 Wallet

A wallet is a set of user identities that an application can use to connect to a channel. Permissions are determined by the channel's MSP and the selected identity.

In figure 4.14 we can see that the channel uses its MSP to determine the permissions associated with the selected identity. The same identity can have different permission on different channels.

Figure 4.15 shows that a wallet can contain identities provided by different CAs.

HLF supports wallet implemented with different technologies, like file or database. Hardware Security Modules are also supported

4.6.4 Gateway

A gateway manages the network interaction on behalf of an application, allowing the developers to focus only on the business logic. Applications connect to a gateway to submit a transaction to the network.

When an application needs to connect to a gateway needs to provide:

- `connectionProfile`: a configuration file that describes the network topology
- `connectionOptions`: additional parameters for the connection

A gateway can also be configured as `dynamic`. In this case, network changes will be discovered and leveraged to bring extra resilience and scalability. This dynamic behavior is the default one.

```
let connectionProfile = yaml.safeLoad(file.readFileSync('./gateway/
    connectionProfile.yaml', 'utf8'));
```

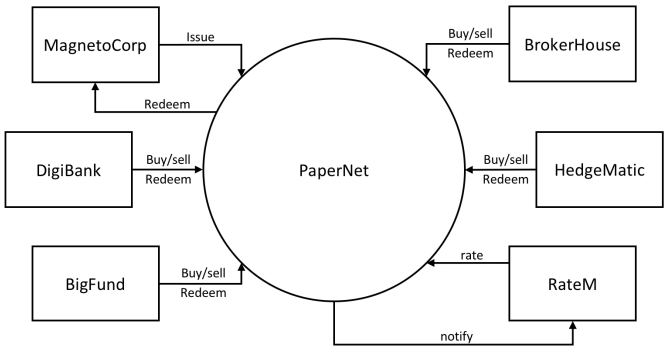

Figure 4.12: PaperNet scenario

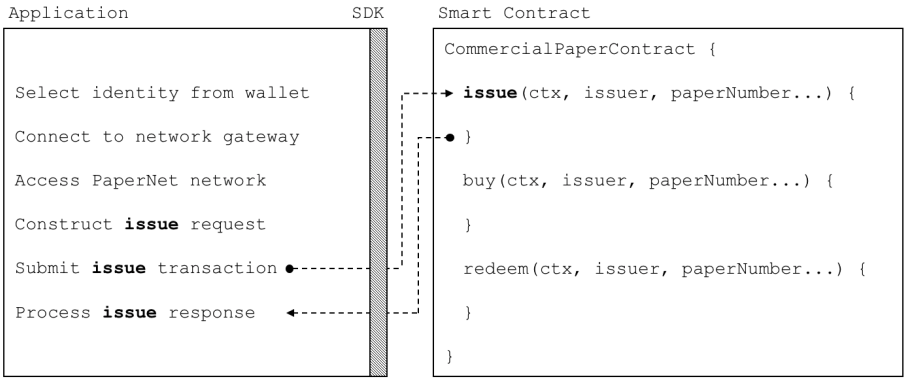

Figure 4.13: An application invokes the SC to submit a transaction request

```
let connectionOptions = {
  identity: userName,
  wallet: wallet
}
await gateway.connect(connectionProfile, connectionOptions);
```

4.6.5 Submit transaction

After the identity has been loaded from the wallet and the gateway created the application can interact with the desired channel and submit transactions to the network. The first thing to do is connect to a network using the gateway and specify the contract to use.

Notice that a gateway can be used to access multiple networks.

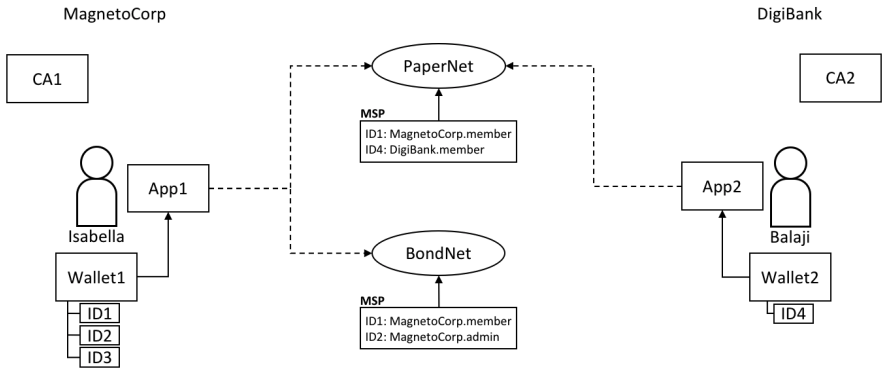

Figure 4.14: Two users, Isabella and Balaji, have wallets containing different identities they can use to connect to different network channels, PaperNet and BondNet.

```
//Connecteing to the nework
const network = await gateway.getNetwork('PaperNet');
const network2 = await gateway.getNetwork('BondNet');

//Specify the channles
const contract = await network.getContract('papercontract', 'org.
    papernet.commercialpaper');
const bondContract = await network2.getContract('BondContract');

//Submit transaction
const issueResponse = await contract.submitTransaction('issue', '
    MagnetoCorp', '00001', '2020-05-31', '2020-11-30', '5000000');
```

The parameters of `submitTransaction` are the parameters of the SC's function which corresponds to the first parameter.

4.6.6 Process transaction response

```
let paper = CommercialPaper.fromBuffer(issueResponse);
```

The application does not receive control back as soon as the smart contract completes its execution. Under the covers, the SDK manages the entire consensus process,

4.7 Putting it all together

In this section, we'll set up a network step-by-step, this scenario three organizations want to set up an HLF network. The main purpose is to show how the various components interact with each other.

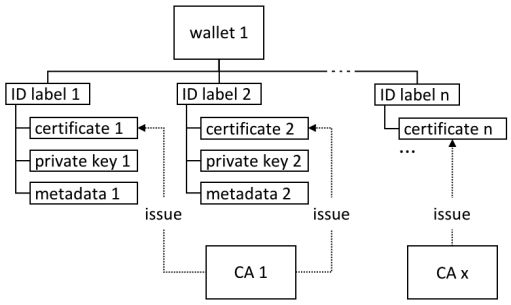

Figure 4.15: A Fabric wallet can hold multiple identities with certificates issued by a different Certificate Authority. Identities comprise certificate, private key and Fabric metadata.

4.7.1 Abbreviations

- **O**: ordering service
- **NC**: network configuration
- **CC**: channel configuration
- **R**: organization
- **CA**: certificate authorities
- **C**: channel
- **P**: peer
- **L**: ledger
- **S**: smart contract
- **X**: consortium

Each of this letters will be followed by a number which indicates the organization.

4.7.2 Creating the network

in figure 4.16 we can see the initial state of the network. In the beginning, the network contains only an ordering service, O4, that acts as the initial administration point of the network. NC4 contains the policies that describe the starting set of administrative authoritiess of the network, initially only R4 has rights over the network

CA4 issues certificates to administrators and network nodes from organization 4. Every organization needs a CA to map identities to resources.

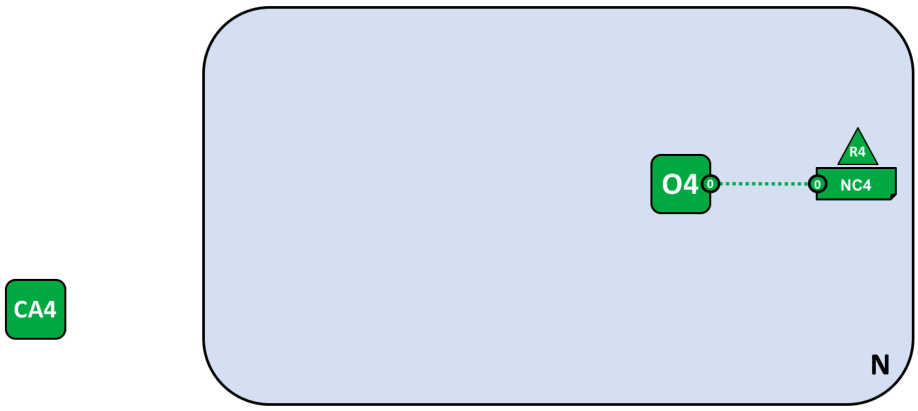

Figure 4.16: Initial state of the network

In this example, the ordering service is composed of a single node owned by a single organization, in a real deployment scenario the ordering service should be formed by multiple nodes owned by different organizations.

4.7.3 Adding Network Administrators

In this phase we'll give administration rights to R1, allowing it's users to administer the network.

R1 and R4 now have equal rights over the network. Although the orderer node, O4, is running on R4's infrastructure, R1 has shared administrative rights over it, as long as it can gain network access. This means that both R1 and R4 could update the network configuration NC4.

4.7.4 Defining a consortium

At this point very little can be done with the network. The first thing to do is create a *consortium*, a group of organizations that wants to work together to reach a common goal.

The diagram 4.18 shows the addition of consortium X1, which defines R1 and R2 as its constituting organizations. A consortium is a set of organizations in the network that share the need to transact with one another.

4.7.5 Creating a channel for a consortium

A channel is the primary communication mechanism for consortium's components. A network can contain multiple channels. In this example, the channel has only two components but there is not an upper limit.

in figure 4.19 we can notice that C1 has a completely separate configuration (CC1) that governs the right that R1 and R2 have over the channel. It is worth

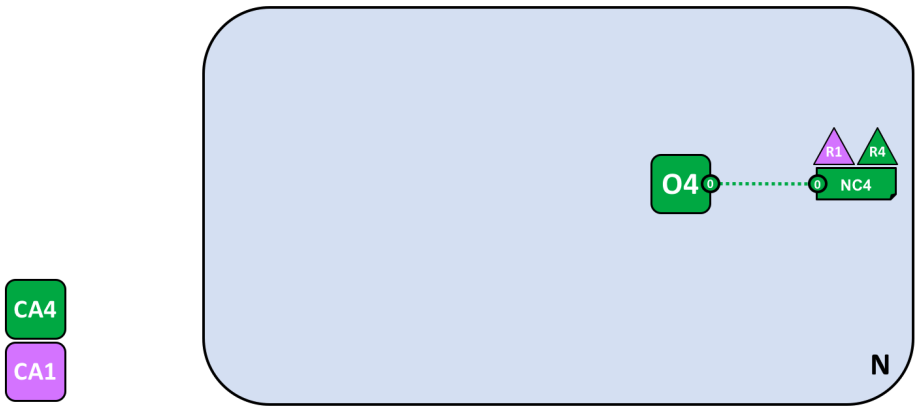

Figure 4.17: Organization R4 updates the network configuration to make organization R1 an administrator too. After this point R1 and R4 have equal rights over the network configuration.

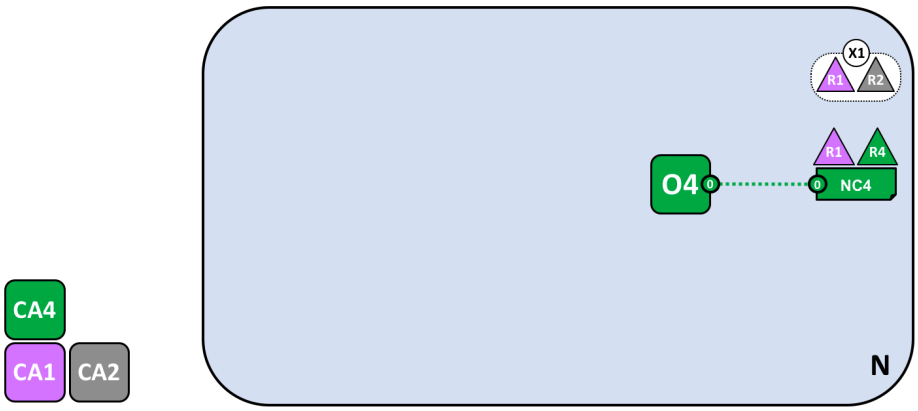

Figure 4.18: Definition of consortium X1

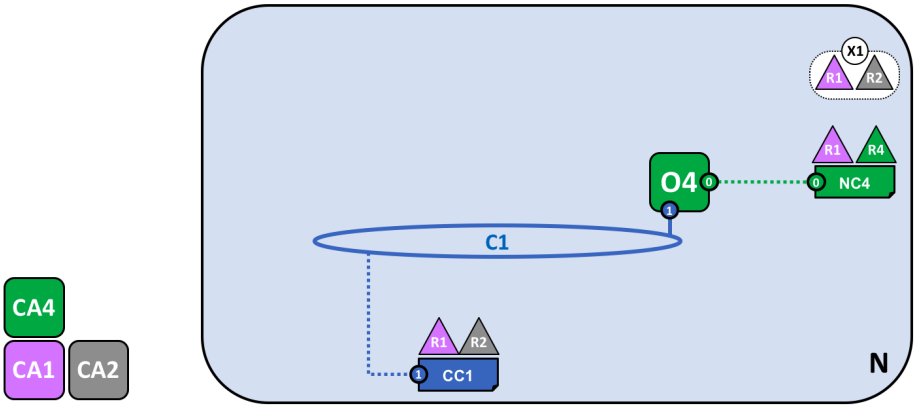

Figure 4.19: A channel C1 has been created for R1 and R2 using the consortium definition X1

notice that R4 has no right over C1 even if it's an administrator for the network; for example, it cannot add himself to the channel without authorization.

Channel created in this way are usually called *application channels*. There is also a special *system channel* dedicated to the ordering service.

4.7.6 Peers and Ledgers

In figure 4.20 P1 joins the channel. Peer nodes are the network components where copies of the blockchain ledger are hosted. We can think that L1 as *physically hosted* on P1 and *logically hosted* on C1.

P1 has an X.509 identity issued by CA1 which associates P1 to R1. P1 can join C1 using O4 which uses CC1 to determine P1's permission on the channel.

4.7.7 Applications and Smart Contract chaincode

In this step we'll connect a client application that will interact with the data stored on L1, using the smart contract S5 installed on P1.

Notice how the application is outside the network but can still interact with it.

Figure 4.21 shows that client application A1 can use C1 to connect to network resources, in this case, P1 and O4.

The client application also has an identity that associates it with an organization. In this case, A1 is associated with organization R1.

Smart contracts are created to implement business processes shared by channel members, the client application uses SCs to interact with the ledger. SC can be viewed as a set of access patterns to the ledger.

After S5 has been developed an administrator in R1 must create a chaincode package and install it into P1, after this step is completed P1 has full knowledge

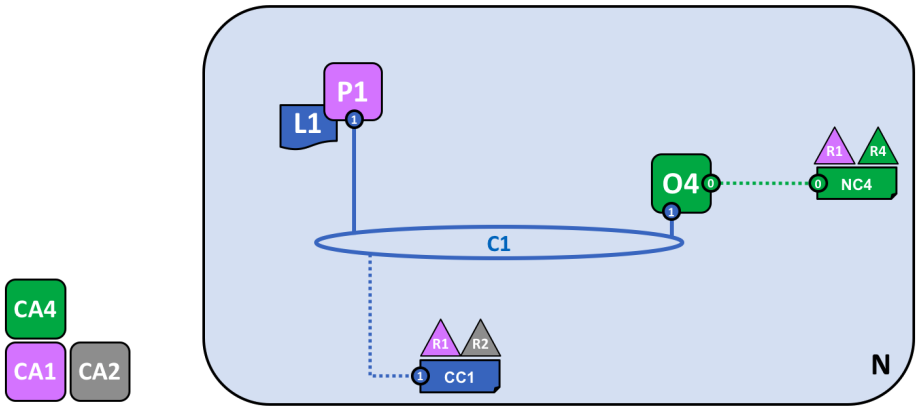

Figure 4.20: A peer node P1 has joined the channel C1. P1 physically hosts a copy of the ledger L1. P1 and O4 can communicate with each other using channel C1.

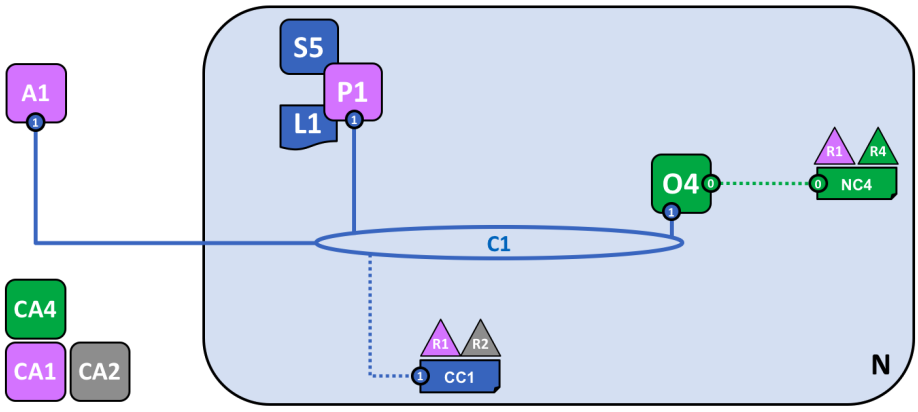

Figure 4.21: A smart contract S5 has been installed onto P1. Client application A1 in organization R1 can use S5 to access the ledger via peer node P1

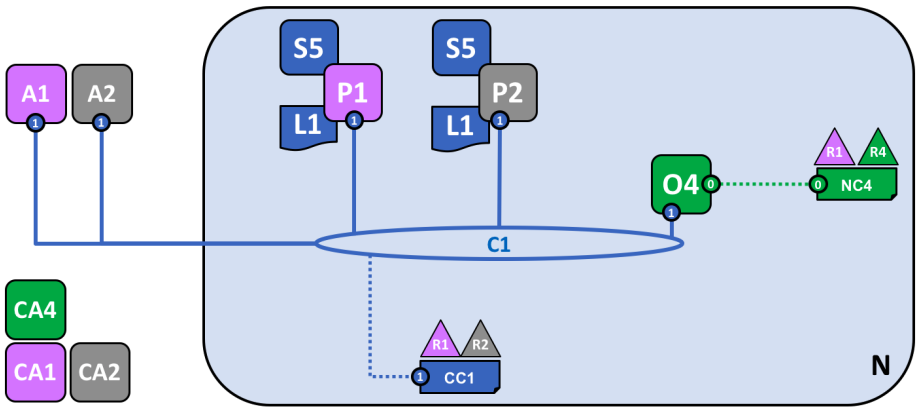

Figure 4.22: The network has grown through the addition of infrastructure from organization R2.

of S5, i.e.: can see it's implementation.

When an organization has multiple nodes in a channel it can choose the peers upon which it installs smart contracts, it is not required to install a copy of an SC on every peer on the network.

Although a chaincode is installed on the peer of an organization it is governed and operated inside a channel. To use an SC inside a channel each organization needs to approve a chaincode definition, a set of parameters that describes how the SC works.[3] The most important part of the chaincode definition is the endorsement policy, the list of organization that needs to approve a transaction before committing to the ledger.

Once an SC has been installed on a peer and approved on a channel can be invoked by external applications.

Only peers with an installed SC can execute it and participate in the endorsement phase but every node on the channel knows the chaincode definition and can flag a transaction as valid or invalid.

4.7.8 Network completed

In this phase, R2 will contribute to the network's resources by adding a peer to C1.

P2 has been added to C1. P2 hosts a copy of L1 and S5, to do so an administrator of R2 has approved the same chaincode definition of R1.

[3] Custom policies for definition approval can be enforced, the default one requires a majority vote from the channel members.

Chapter 5

The first application

In this section, we will show how to develop contracts and applications and how to make them available within a network.

We'll work with the scenario introduced in 4.6.1 using the material contained in the `commercial-paper` directory of the attached repository. Inside said folder there is the `organization` folder which contains a dedicated directory for each organization.

5.1 Prerequisites

Follow the official setup guide[1] but:

- **don't** download the sample repository
- when installing binaries and images follow the instructions for download the latest production releases

If you have executed this code before please run `./network-clean.sh` for start from a clean state.

5.2 Create the network

In this section, we'll use Fabric's sample network configuration:`test-network`. This network composed of two organizations, operating a node each, and an ordering service composed by a single Raft node operated by a third organization. A channel named `mychannel` will be configured over the network and both organizations will be members of this channel.

The network can be started with a simple script, `./network-starter.sh`. If the launch is successful log messages will appear on the screen. The resulting docker containers can be seen with `docker ps -a`, see figure 5.1

[1] https://hyperledger-fabric.readthedocs.io/en/latest/getting_started.html

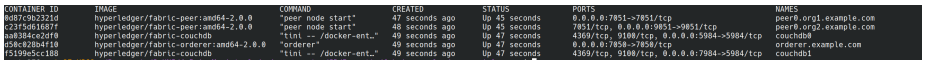

Figure 5.1: Running container after `./network-starter.sh`

By launching `docker network inspect net_test` the network configuration will be displayed, see figure 5.1

In the following experiment `peer0.org1` will belong to DigiBank, the organization that wants to buy and redeem commercial paper, whereas `peer0.org2` will belong to MagnetoCorp, the organization who wants to issue commercial papers.

5.3 Examine the SC

In this section, we'll analyze the smart contract developed by the two organizations to manage their business relationship. This contract provides applications developed by organizations with 3 functions, `issue`, `buy` and `redeem`, which allow them to manage the commercial paper's lifecycle. Each organization has a copy of the SC.

SC's core is `lib/paper contract.js` while `lib/paper.js` is used to model a single paper and `lib/paperlist` to model a list of papers. The latter two are based on `ledger-api/state.js` and `ledger-api/statelist.js` respectively.

5.3.1 Analysis of `ledger-api/state.js`

This class models a generic object stored on the blockchain.

```
1  class State {
2      /**
3      * @param {String|Object} class  An indentifiable class of the
       instance
4      * @param {keyParts[]} elements to pull together to make a key
       for the objects
5      */
6      constructor(stateClass, keyParts) {
7          this.class = stateClass;
8          this.key = State.makeKey(keyParts);
9          this.currentState = null;
10     }
11     /**
12     * Join the keyParts to make a unififed string
13     * @param (String[]) keyParts
14     */
15     static makeKey(keyParts) {
16         return keyParts
17             .map(part => JSON.stringify(part))
18             .join(':');
19     }
20
21     static splitKey(key){
```

```
        return key.split(':');
    }
}
```

In this snippet of `ledger-api/state.js` we can see that the constructor takes the class name and `keyParts`, an array of objects that will be used in `makeKey` to build the key, a unique identifier for this object in the ledger.

5.3.2 Analysis of `lib/paper.js`

Extends `ledger-api/state.js` to model the specific operations that can be performed on a paper. Includes basic getters and setters and methods of encapsulating access to state information.

5.3.3 Analysis of `ledger-api/statelist.js`

`ledger-api/statelist.js` provides a named virtual container for a set of ledger states. Each state has a unique key which associates it with the container.

```
class StateList {

    constructor(ctx, listName) {
        this.ctx = ctx;
        this.name = listName;
        this.supportedClasses = {};

    }

    async addState(state) {
        let key = this.ctx.stub.createCompositeKey(this.name, state
    .getSplitKey());
        let data = State.serialize(state);
        await this.ctx.stub.putState(key, data);
    }

    async getState(key) {
        let ledgerKey = this.ctx.stub.createCompositeKey(this.name,
     State.splitKey(key));
        let data = await this.ctx.stub.getState(ledgerKey);
        if (data && data.toString('utf8')) {
            let state = State.deserialize(data, this.
    supportedClasses);
            return state;
        } else {
            return null;
        }
    }

    async updateState(state) {
        let key = this.ctx.stub.createCompositeKey(this.name, state
    .getSplitKey());
        let data = State.serialize(state);
        await this.ctx.stub.putState(key, data);
    }
```

```
32
33      use(stateClass) {
34          this.supportedClasses[stateClass.getClass()] = stateClass;
35      }
36  }
```

This class exposed method to add, get and update states. A list of supported classes is also managed. One of the most important parts of this snipped is `ctx`, the transaction's context. The transaction's context performs two functions:

1. define and maintain user variables across transaction invocations within a smart contract
2. provides access to a wide range of Fabric APIs that allow smart contract developers to perform a wide range of operations like querying or updating the ledger to retrieving the transaction-submitting application's digital identity.

HLF's APIs can be access via `ctx.stub`, more detailed information can be found on the official documentation[2]. Notice how `ctx.stub.createCompositeKey` is used in combination with `State.splitKey` to create unique keys across multiple state list. With this solution, it is possible to have two states with the same key if inserted in a different state list.

5.3.4 Analysis of `lib/paperlist.js`

Simple wrapper for the latter class models the paper's specific operations.

5.3.5 Analysis of `lib/papercontract.js`

Let's start by looking at the setup part of the class:

```
1  **
2  * A custom context provides easy access to list of all commercial
       papers
3  */
4  class CommercialPaperContext extends Context {
5
6      constructor() {
7          super();
8          // All papers are held in a list of papers
9          this.paperList = new PaperList(this);
10     }
11
12 }
13
14 /**
15  * Define commercial paper smart contract by extending Fabric
       Contract class
16  *
17  */
18 class CommercialPaperContract extends Contract {
```

[2] https://hyperledger-fabric.readthedocs.io/en/latest/developapps/transactioncontext.html#stub

```
19
20      constructor() {
21          // Unique namespace when multiple contracts per chaincode
        file
22          super('org.papernet.commercialpaper');
23      }
24
25      /**
26       * Define a custom context for commercial paper
27      */
28      createContext() {
29          return new CommercialPaperContext();
30      }
31
32  }
```

The first thing that is possible to notice is that the `Contract`[3] class is extended. The `Contract` class defines all the code needed to run the chaincode inside a Docker container.
A custom `Context`[4] is defined in order to use an instance of `PaperList` to store papers on the ledger.
The remaining part of the contract contains the three transactions: issue, buy and redeem.

```
1   /**
2    * Buy commercial paper
3    *
4    * @param {Context} ctx the transaction context
5    * @param {String} issuer commercial paper issuer
6    * @param {Integer} paperNumber paper number for this issuer
7    * @param {String} currentOwner current owner of paper
8    * @param {String} newOwner new owner of paper
9    * @param {Integer} price price paid for this paper
10   * @param {String} purchaseDateTime time paper was purchased (i.e.
       traded)
11  */
12  async buy(ctx, issuer, paperNumber, currentOwner, newOwner, price,
       purchaseDateTime) {
13
14      // Retrieve the current paper using key fields provided
15      let paperKey = CommercialPaper.makeKey([issuer, paperNumber]);
16      let paper = await ctx.paperList.getPaper(paperKey);
17
18      // Validate current owner
19      if (paper.getOwner() !== currentOwner) {
20          throw new Error('Paper ' + issuer + paperNumber + ' is not
        owned by ' + currentOwner);
21      }
22
23      // First buy moves state from ISSUED to TRADING
24      if (paper.isIssued()) {
25          paper.setTrading();
```

[3] https://hyperledger.github.io/fabric-chaincode-node/release-2.0/api/fabric-contract-api.Contract.html
[4] https://hyperledger.github.io/fabric-chaincode-node/release-2.0/api/fabric-contract-api.Context.html

```
    }
    // Check paper is not already REDEEMED
    if (paper.isTrading()) {
        paper.setOwner(newOwner);
    } else {
        throw new Error('Paper ' + issuer + paperNumber + ' is not trading. Current state = ' +paper.getCurrentState());
    }
    // Update the paper
    await ctx.paperList.updatePaper(paper);
    return paper;
}
```

This snippet shows a detail of the `buy` transaction but the considerations that can be made also apply to the other transactions. The goal of each transaction is to ensure that all the operations are legitimate, for this reason, it is important to insert all the checks that organizations have agreed upon when they have defined the smart contract. In this case, checks are made on the status of the paper, specifically that it has not already been redeemed. If the code executes without exceptions the peer will return a signed endorsement to the application.

5.4 Examine the application

Figure 5.3 shows how applications interact with the HLF's network to perform operations.

5.4.1 Get the identity

We can see that the first step requires to retrieve an identity from a wallet. Identities are X.509 digital certificates issued by CA to authenticate users. Identities, in this case, can be created with `addToWallet.js`; let's examine how this utility works.

```
const wallet = await Wallets.newFileSystemWallet('../identity/user/isabella/wallet');

// Identity to credentials to be stored in the wallet
const credPath = path.join(fixtures, '/organizations/peerOrganizations/org2.example.com/users/User1@org2.example.com');
const certificate = fs.readFileSync(path.join(credPath, '/msp/signcerts/User1@org2.example.com-cert.pem')).toString();
const privateKey = fs.readFileSync(path.join(credPath, '/msp/keystore/priv_sk')).toString();

// Load credentials into wallet
const identityLabel = 'isabella';

const identity = {
    credentials: {
        certificate,
        privateKey
    },
```

```
    mspId: 'Org2MSP',
    type: 'X.509'
}
await wallet.put(identityLabel,identity);
```

The utility class `wallets` made available by HLF is used for managing wallets stored on the filesystem. Identities inside wallets are managed like a key-value pair where the user's name acts as the key and a data structure, containing both certificate and private key, acts as a value.

5.4.2 Connect to the gateway

Gateways are used by applications to connect with the network using a configuration file that describes its topology, in particular, which are the peers to contact to submit transactions.

```
const gateway = new Gateway();

try {

    // Specify userName for network access
    // const userName = 'isabella.issuer@magnetocorp.com';
    const userName = 'isabella';

    // Load connection profile; will be used to locate a gateway
    let connectionProfile = yaml.safeLoad(fs.readFileSync('../
    gateway/connection-org2.yaml', 'utf8'));

    // Set connection options; identity and wallet
    let connectionOptions = {
        identity: userName,
        wallet: wallet,
        discovery: { enabled:true, asLocalhost: true }
    };

    // Connect to gateway using application specified parameters
    console.log('Connect to Fabric gateway.');

    await gateway.connect(connectionProfile, connectionOptions);
}
```

To connect to the network via the `gateway.connect()` method the user must provide a file that describes the network topology and a set of additional settings for the connection. Once the connection is established the gateway object can be used for access multiple channels.

5.4.3 Main interactions with chaincode

```
const network = await gateway.getNetwork('mychannel');
const contract = await network.getContract('papercontract');
const issueResponse = await contract.submitTransaction('issue', '
    MagnetoCorp', '00003', '2020-05-31', '2020-11-30', '50000');
let paper = CommercialPaper.fromBuffer(issueResponse);
```

When we want to execute a transaction the first step to do is connect to the desired channel via the gateway and then get the contract from the channel. HLF's API `contract.submitTransaction` is used to execute transaction on the specified contract. This functions takes a list of values as input where the first one is the desired function name whereas the ramaining are passed as arguments to said function.

The same considerations are valid for all the remaining applications.

5.5 Deploy the smart contract to the channel

Before the contract can be invoked by the organizations it must be installed on an appropriate number of nodes and its definition must be approved within the channel. SCs are contained inside an artifact called *chaincode*. Multiple contracts can be defined within a single chaincode. The number of nodes on which the chaincode must be installed depends on the chaincode approval policy defined within the network. In this case, we use the default policy which requires approval from the majority of the organizations that make up the channel.

5.5.1 Install and approve for MagnetoCorp

We'll start by installing and approve the contract for MagnetoCorp, be sure to be inside the `organization/magnetocorp` folder. The first thing to do is open a new terminal and set some environment variable needed to interact with MagnetoCorp's peer by running this command:

```
source magnetocorp.sh
```

Now the package for the chaincode can be created using the `peer`[5] CLI.

```
peer lifecycle chaincode package cp.tar.gz \
    --lang node \
    --path ./contract \
    --label cp_0
```

Now we can install the chaincode on MagnetoCorp's peer:

```
peer lifecycle chaincode install cp.tar.gz
```

Logs will appear on the screen showing the result of the installation process.

Notice what happens if we try to launch now the `issue` application: In figure 5.4 we can see that the execution fails because no endorsement plan can be found for the specified chaincode, this occurs because it is installed on the peer but not defined in the channel.

After the chaincode's installation we need to approve its definition, the first thing to do is get the chaincode ID:

```
peer lifecycle chaincode queryinstalled
```

In figure 5.5 we can see the chaincode ID, now we can approve it's definition for this organization.

[5] https://hyperledger-fabric.readthedocs.io/en/latest/commands/peercommand.html

```
export PACKAGE_ID=#chacode id

peer lifecycle chaincode approveformyorg \
    --orderer localhost:7050 \
    --ordererTLSHostnameOverride orderer.example.com \
    --channelID mychannel \
    --name papercontract \
    -v 0 \
    --package-id $PACKAGE_ID \
    --sequence 1 \
    --tls \
    --cafile $ORDERER_CA
```

Using `approveformyorg` we can approve the chaincode definition for this organization. DigiBank will need to approve the same definition.

One of the most important chaincode parameters that channel members need to agree to use the chaincode definition is the chaincode endorsement policy. The endorsement policy describes the set of organizations that must endorse (execute and sign) a transaction before it can be approved. By approving the `papercontract` chaincode without the `--policy` flag, the MagnetoCorp admin agrees to use the default endorsement policy, which requires a majority of organizations on the channel to endorse a transaction. All transactions, whether valid or invalid, will be recorded on the ledger blockchain, but only valid transactions will update the world state.

If we try to execute now the `issue` application we'll the same error occurred before, see figure 5.4. The chaincode is approved for this organization but not yet defined in the channel.

5.5.2 Install and approve for DigiBank

By default, the Fabric Chaincode lifecycle requires a majority of organizations on the channel to successfully commit the chaincode definition to the channel. This implies that we need to approve the papernet chaincode as both MagnetoCorp and DigiBank to get the required majority of 2 out of 2. Open a new terminal window and navigate to the folder that contains the DigiBank's smart contract and application files located in `commercial-paper/organization/digibank`.

The installation and approval steps are the same seen for MagnetoCorp

```
source digibank.sh

peer lifecycle chaincode package cp.tar.gz \
    --lang node \
    --path ./contract \
    --label cp_0

peer lifecycle chaincode install cp.tar.gz

export PACKAGE_ID=#chaincode's ID

peer lifecycle chaincode approveformyorg \
    --orderer localhost:7050 \
    --ordererTLSHostnameOverride orderer.example.com \
```

```
    --channelID mychannel \
    --name papercontract \
    -v 0 \
    --package-id $PACKAGE_ID \
    --sequence 1 \
    --tls \
    --cafile $ORDERER_CA
```

5.5.3 Approve chaincode definition inside the channel

Now that DigiBank and MagnetoCorp have both approved the chaincode, we have the majority we need to commit the chaincode definition to the channel. Once the chaincode is successfully defined on the channel, the `CommercialPaper` smart contract can be invoked by client applications. Both organizations can commit the chaincode to the channel.

```
peer lifecycle chaincode commit
    -o localhost:7050 \
    --ordererTLSHostnameOverride orderer.example.com \
    --peerAddresses localhost:7051 \
    --tlsRootCertFiles ${PEER0_ORG1_CA} \
    --peerAddresses localhost:9051 \
    --tlsRootCertFiles ${PEER0_ORG2_CA} \
    --channelID mychannel \
    --name papercontract \
    -v 0 \
    --sequence 1 \
    --tls \
    --cafile $ORDERER_CA \
    --waitForEvent
```

The chaincode's container will start after the chaincode definition has been committed to the channel. You can use the the `docker ps` command to see `papercontract` containers running on both peers.

5.6 Using the applications

Application can be run with `node application.js`, remember to install the required dependencies with `nmp install`.

First of all all applications have been modified so that the id of the papers is obtained from the command line rather than defined within the code.

```
var myArgs = process.argv.slice(2);
const issueResponse = await contract.submitTransaction('issue', '
    MagnetoCorp', myArgs[0], '2020-05-31', '2020-11-30', '50000');
```

In figure 5.6 we can see what happens if we issue a paper with id set to 1 (`node issue.js 1`) whereas in figure 5.7 we can see what happens if an invalid transaction, like buying a non exixtent paper, is issued.

5.7 Some small extensions

Each organization can use a different *packageID* when they approve a chaincode definition. This allows channel members to install different chaincode binaries that use the same endorsement policy and read/write. Channel members can use this capability to install chaincode written in their preferred language or to implement additional checks.

5.7.1 Custom checks for DigiBank

Let's suppose that DigiBank wants to prevent MagnetoCorp to issue papers with a value lower than 10K$. DigiBank can introduce some additional controls over its chaincode definition and reinstall it over its peers. With this solution when MagnetoCorp tries to generate a paper with a value lower than 10K$ DigiBank will throw an exception and, as a result, there will not be enough signatures to comply with the endorsement policy and, consequently, the transaction will not be included in the blockchain.

This additional check can be implemented by altering the `issue` implementation for DigiBank:

```
1 if (intFaceValue < 10000){
2     throw new Error("Face value must be over 10K$");
3 }
```

Now we can install this new chaincode on `peer0.org2` following the same steps described in 5.5.2.

In figure 5.8 we can see what happens if MagnetoCorp tries to submit a paper with a face value lower than 10K$.

5.7.2 Custom checks for MagnetoCorp

With the current chaincode and channel configuration, DigiBank is also capable to issue papers. MagnetoCorp can use the `getCreator()` API to obtain information on who issued the request and preventing DigiBank to issue papers.

```
1 if (ctx.stub.getCreator().mspid != `Org2MSP`) {
2     throw new Error("Only MagnetoCorp. is authorized to issue
      papers")
3 }
```

The `mspid` field contains the organization's name contained in the transaction proposal sent by the application.

In figure 5.9 and figure 5.10 we can see, respectively, what happens when DigiBank issues a paper before and after the introduction of this new check.

5.7.3 Approve unilateral changes and restrictions

This kind of change requires approval from only the organizations that made them. The required instructions for the approval process are the same showed in 5.5.2.

This approach *cannot* be used when:

- contracts interface is modified
- read/write sets are modified

5.8 Chaincode upgrade

Unilateral changes, like those shown in 5.7, are useful for adding some simple controls but a chaincode upgrade is required when major changes are needed.

Suppose the two organizations agree to make sure that ids cannot be reused. To carry out this check it is necessary to verify in the ledger that the supplied ID is not already in use, this operation causes a modification of the read set and therefore requires a chaincode update.

```
let paperKey = CommercialPaper.makeKey([issuer, paperNumber]);
let p = await ctx.paperList.getPaper(paperKey);
if (p != null) {
    throw new Error("ID already in use")
}
```

After the chaincode has been modified we can follow the same procedure illustrated in 5.5 to install the chaincode and commit its definition on the channel, version and sequence numbers must be incremented by 1. Now if we try to issue a paper with the same id an error will be thrown, see figure 5.11.

Notice that even if an involved party changes it's chaincode implementation to avoid this new check the other party would still enforce it preventing the transaction to take place.

```
[
    {
        "Name": "net_test",
        "Id": "afa10b4a59877dfe30b14980aaf407c0e3c275b9873e9e700eb9791ebeb24b29",
        "Created": "2020-03-04T22:14:09.58618327+01:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "1d63e3333c3725671f1f344c948871aeb6657d5397acc5d51f490ab1a38020cc": {
                "Name": "orderer.example.com",
                "EndpointID": "b2b335153d3bff23b87de2d3695fe15ebc369b4081b15495778b12a4d2ae2619",
                "MacAddress": "02:42:ac:13:00:04",
                "IPv4Address": "172.19.0.4/16",
                "IPv6Address": ""
            },
            "4d80367667d0a6bb781aa96e32b8c7b7815cbd1c8d2e7859858b064ae2029688": {
                "Name": "peer0.org1.example.com",
                "EndpointID": "9907a95424a4c0a4617c55f3d12bd0d263c85d9e611603ab70c1b34c2ae2c462",
                "MacAddress": "02:42:ac:13:00:06",
                "IPv4Address": "172.19.0.6/16",
                "IPv6Address": ""
            },
            "5b1460aeeecb8eeab147ea8a081ac64341a852eddacbd592ed1777078090c3cf": {
                "Name": "peer0.org2.example.com",
                "EndpointID": "bae3573f8858ac9949c202c8847e19ea69e39a4161def1d0235c8e1a57877477",
                "MacAddress": "02:42:ac:13:00:05",
                "IPv4Address": "172.19.0.5/16",
                "IPv6Address": ""
            },
            "7effca0bd27f4bb7654f8a79e46ed01213fd2e37e81418f7083bc91ba1677a94": {
                "Name": "couchdb0",
                "EndpointID": "3cb4f0c47862d764624d220001bdd06b9c115ed812e28e860262011a85bc16fb",
                "MacAddress": "02:42:ac:13:00:03",
                "IPv4Address": "172.19.0.3/16",
                "IPv6Address": ""
            },
            "a420cc636ca6f23793b725a912772dd0d8d8dcc20b74ef00a574b9a54cae9100": {
                "Name": "couchdb1",
                "EndpointID": "c24ef84468d2a269a01ffec4c8d3266d2cb23d22ff17904b8d4b2ee979048b25",
                "MacAddress": "02:42:ac:13:00:02",
                "IPv4Address": "172.19.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```


Figure 5.2: Docker network

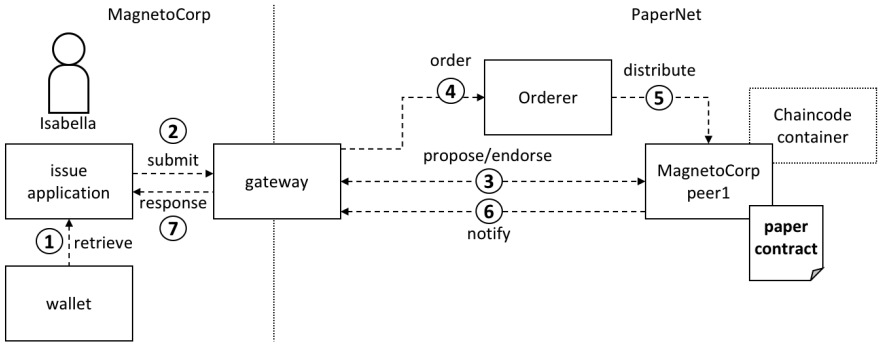

Figure 5.3: Application's main component and flow

```
Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper issue transaction.
2020-02-24T09:01:47.205Z - error: [Channel.js]: Channel:mychannel received discovery error:failed constructing descriptor for chaincodes:<name:"papercontract" >
2020-02-24T09:01:47.296Z - error: [DiscoveryEndorsementHandler]: endorse - no endorsement plan found for {"chaincodes":[{"name":"papercontract"}]}
Error processing transaction. Error: No endorsement plan available for {"chaincodes":[{"name":"papercontract"}]}
Error: No endorsement plan available for {"chaincodes":[{"name":"papercontract"}]}
```

Figure 5.4: Chaincode not found in network

```
Installed chaincodes on peer:
Package ID: cp_0:640f216fa696e10cde147eae9b6a81529ca7586a93544e83eeca0ffcf6c73a52, Label: cp_0
```

Figure 5.5: List on installed chaincode

```
Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper issue transaction.
Process issue transaction response.{"class":"org.papernet.commercialpaper","key":"\"MagnetoCorp\":\"1\"","currentState":1,"issuer":"MagnetoCorp","paperNumber":"1","issueDateTime":"2020-05-3
1","maturityDateTime":"2020-11-30","faceValue":"50000","owner":"MagnetoCorp"}
MagnetoCorp commercial paper : 1 successfully issued for value 50000
Transaction complete.
Disconnect from Fabric gateway.
Issue program complete.
```

Figure 5.6: Issue of paper. number 1

```
Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper buy transaction.
2020-02-24T12:45:13.447Z - error: [Transaction]: Error: No valid responses from any peers. Errors:
    peer=peer0.org2.example.com:9051, status=500, message=error in simulation: transaction returned with failure: TypeError: Cannot read property 'getOwner' of null
    peer=peer0.org1.example.com:7051, status=500, message=error in simulation: transaction returned with failure: TypeError: Cannot read property 'getOwner' of null
Error processing transaction. Error: No valid responses from any peers. Errors:
    peer=peer0.org2.example.com:9051, status=500, message=error in simulation: transaction returned with failure: TypeError: Cannot read property 'getOwner' of null
    peer=peer0.org1.example.com:7051, status=500, message=error in simulation: transaction returned with failure: TypeError: Cannot read property 'getOwner' of null
Error: No valid responses from any peers. Errors:
    peer=peer0.org2.example.com:9051, status=500, message=error in simulation: transaction returned with failure: TypeError: Cannot read property 'getOwner' of null
    peer=peer0.org1.example.com:7051, status=500, message=error in simulation: transaction returned with failure: TypeError: Cannot read property 'getOwner' of null
    at newEndorsementError (/home/monte97/Documents/2_UNI/4_PrimoMagistrale/primo_semestre/SD/Progetto/fabric-samples/commercial-paper/organization/digibank/applicatio
network/lib/transaction.js:49:12)
    at getResponsePayload (/home/monte97/Documents/2_UNI/4_PrimoMagistrale/primo_semestre/SD/Progetto/fabric-samples/commercial-paper/organization/digibank/applicatio
etwork/lib/transaction.js:21:23)
    at Transaction.submit (/home/monte97/Documents/2_UNI/4_PrimoMagistrale/primo_semestre/SD/Progetto/fabric-samples/commercial-paper/organization/digibank/applicatio
etwork/lib/transaction.js:218:28)
Disconnect from Fabric gateway.
Buy program complete.
```

Figure 5.7: Transaction failure due invalid response

```
Connect to Fabric gateway.
Use network channel: mychannel.
Use org.papernet.commercialpaper smart contract.
Submit commercial paper issue transaction.
2020-02-24T13:45:19.312Z - warn: [DiscoveryEndorsementHandler]: _build_endorse_group_member >> G1:1 - endorsement failed - Error: error in simulation: transactio
or: Face value must be over 10K$
2020-02-24T13:45:19.313Z - error: [DiscoveryEndorsementHandler]: _endorse - endorsement failed::Error: Endorsement has failed
    at DiscoveryEndorsementHandler._endorse (/home/monte97/Documents/2_UNI/4_PrimoMagistrale/primo_semestre/SD/Progetto/fabric-samples/commercial-paper/organizat
node_modules/fabric-client/lib/impl/DiscoveryEndorsementHandler.js:185:19)
Error processing transaction. Error: Endorsement has failed
Error: Endorsement has failed
    at DiscoveryEndorsementHandler._endorse (/home/monte97/Documents/2_UNI/4_PrimoMagistrale/primo_semestre/SD/Progetto/fabric-samples/commercial-paper/organizat
node_modules/fabric-client/lib/impl/DiscoveryEndorsementHandler.js:185:19)
Disconnect from Fabric gateway.
Issue program complete.
```

Figure 5.8: Failed endorsement thanks to the new policy

Figure 5.9: DigiBank issues a paper without *mspid* checks

Figure 5.10: DigiBank issues a paper with *mspid* checks

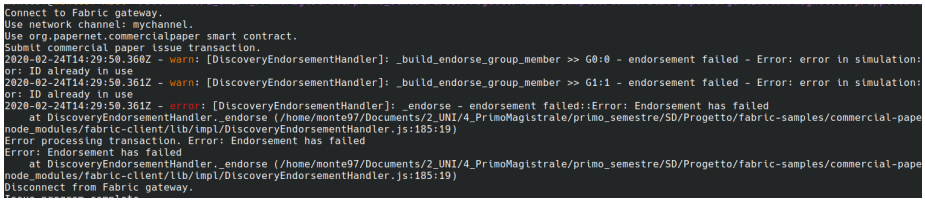

Figure 5.11: Fails when a ID is reissued

Chapter 6

Extension: reactive application

Experiments in this chapter will expand those in 5.

6.1 Scenario

Let's suppose that DigiBank would like to buy any paper with a face value of over 1M$ as soon as it's issued. To achieve this we will intervene both on SCs and applications to create a system capable of reacting to events that occur on the ledger.

6.2 Smart contract

The contracts have been modified so that the `issue` transaction is associated with a particular event when inserted into the block. In this way, by examining the blocks, it is possible to recognize this type of transactions more easily.

A new function capable of executing arbitrary queries has also been added.

6.2.1 Rich query and CouchDB

Fabric supports two types of databases: LevelDB and CouchDB.

LevelDB is the default state database embedded in the peer node. LevelDB stores chaincode data as simple key-value pairs and only supports key, key range, and composite key queries.

CouchDB is a state database that allows to model data on the ledger as JSON and issue rich queries against data values.

```
1 async richQuery(ctx, queryString) {
2
3     let iterator = await ctx.stub.getQueryResult(queryString);
4
```

```
    let allResults = [];
    while (true) {
        let res = await iterator.next();

        if (res.value && res.value.value.toString()) {
            let jsonRes = {};

            jsonRes.Key = res.value.key;
            try {
                jsonRes.Record = JSON.parse(res.value.value.
toString('utf8'));
            } catch (err) {
                console.log(err);
                jsonRes.Record = res.value.value.toString('utf8');
            }
            allResults.push(jsonRes);
        }
        if (res.done) {
            console.log('end of data');
            await iterator.close();
            break
        }
    }
    return Buffer.from(JSON.stringify(allResults));
}
```

Rich query have been issued with `getQueryResult(queryString)` which requires a query string with the following JSON format:

```
{
    "selector": {
        "key1":"value1",
        "key2":"value2",
          "key3": {
             "$gt":100000
        }
    }
}
```

Search key are comma separated. "Complex" queries are inserted inside a dedicated object, a full list of available condition can be found in the offical ChouchDB documentation[1]

6.2.2 Events

The issue `function` has been modified so that an event is placed in the block along with the transaction:

```
ctx.stub.setEvent('issueEvent', "");
```

In figure 6.1 and in figure 6.2 we can see, respectively, block's content before and after the introduction of this event.

Block's acquisition is discussed in 6.3.

[1] http://docs.couchdb.org/en/stable/api/database/find.html#condition-operators

Figure 6.1: Block's content *without* event

Figure 6.2: Block's content *with* event

6.3 Application

```
const listener = await network.addBlockListener((p1, p2, p3, p4) => {
    if (eventInBlock('issueEvent', p3)){
        let response = contract.evaluateTransaction(
            'richQuery',
                `{"selector": {
                    "owner": "MagnetoCorp",
                    "issuer" : "MagnetoCorp",
                    "intFaceValue" : {
                        "$gt": 100000
                    }
                }}`
        );
        response.then(content => {
            selectedPapers = parseResponse(content)
            selectedPapers.forEach(paper => {
                let number = paper.Record.paperNumber
                let price = paper.Record.intFaceValue
                let buyResponse = contract.submitTransaction('buy',
                    'MagnetoCorp',
                    number,
                    'MagnetoCorp',
                    'DigiBank',
                    price*0.9,
                    '2020-05-31'
                );
                buyResponse.then(
                    ok =>
                        {console.log(ok.toString())},
                    nok =>
                        {console.log("error: " + nok.toString())}
                )
            })
        }, error_msg => {
            console.log(msg)
        })
    }
}, 'name')
```

A new listener is added on the network via `addBlockListner`, this method allows us to define a callback that's called every time a block is added on the blockchain. When the callback is executed it is possible to access the block and inspect its transactions. In this case, we used the `eventInBlock` function to check if the specified event is contained in the specified block.

6.4 SDK's issues

Network's `listner` API have been temporally removed from Fabric's Node SDK, see this pull request[2]. What showed in 6.3 currently seems the only way to intercept a block event.

[2]https://github.com/hyperledger/fabric-sdk-node/pull/116

6.5 Update (22 March, 2020)

Due to an update to `evaluateTransaction`'s signature it is necessary to introduce a small modification to how the listener of the event is registered. Please, always refer to always refer to the latest available version of the official documentation.

```
const listener = async (event) => {

  let response = contract.evaluateTransaction('richQuery', `{"
    selector": {"owner": "MagnetoCorp", "issuer" : "MagnetoCorp", "
    intFaceValue" : {"$gt": 100000}}}`);
  if (response !== "undefined") {
    response.then(content => {
            selectedPapers = parseResponse(content)
            selectedPapers.forEach(paper => {
                let number = paper.Record.paperNumber
                let price = paper.Record.intFaceValue
                let buyResponse = contract.submitTransaction('buy',
     'MagnetoCorp', number, 'MagnetoCorp', 'DigiBank', price*0.9, '
    2020-05-31');
                buyResponse.then(ok => {console.log(ok.toString())
    }, nok => {console.log("error: " + nok.toString())})
            })
        }, error_msg => {
            console.log(msg)
        })
  }

}
const options = {
  startBlock: 1,
};
await network.addBlockListener(listener, options);
```

Chapter 7

Extension: communication between contracts

Experiments in this chapter will expand those in 5.

7.1 Scenario

In this scenario a new organization is introduced: RatingAgency. This new organization will develop a smart contract for some simple analytically purposes. The definition of the new contract is simple, the most interesting part concerns the deploy process.

7.2 Add an organization to the channel

A new startup script has been created: `network-starter-3org.sh`. This script makes use of another Fabric's sample located in the `test-network/addOrg3` folder.

The script provided by Fabric will take care of generating all the needed cryptographic material for the new organization and perform the required operations to make its peer join the channel.

7.3 Chaincode namespace

A chaincode namespace allows it to keep its world state separate from other chaincodes. This means that SCs defined in the same chaincode share direct access to the same world state whereas SCs defined in different chaincodes cannot directly access each other's world state. If a chaincode needs to access another world state it can do this via a chaincode-to-chaincode invocation. Namespaces allow users to operate different parts of a shared system without step on each

other feet, HLF uses namespaces to help smart contracts to keep their world state from separated other's SCs.

7.3.1 Cross chaincode access

Chaincode-to-chaincode interaction can be done using the `invokeChaincode` API but there are some restrictions:

- both chaincodes must be installed on the same peer
- If the interaction only requires to read some data then the invocation can be done across channels
- If the interaction requires to write on the world state then the invocation must be in the same channel

7.4 Query world state

In this sample we'll call the `richQuery` function defined in 6.2.1 from another chaincode using the `invokeChaincode` API. A simple client application has also been created.

```
let chaincodeName = "papercontract";

let fcn = "richQuery";

let fcnArgs = `{"selector": {"issuer": "${who}"}}`

let channel = "mychannel";

let queryResponse
    = await ctx.stub.invokeChaincode(chaincodeName, [fcn, fcnArgs],
     channel);
```

This example can be arbitrarily complicated, the main goal is to show how chaincode-to-chaincode works.

7.5 Deploy

7.5.1 Prerequiste

First of all, deploy the CommercialPaper chaincode on MagnetoCorp and DigiBank's peers following the instructions in 5.5, add user's identity to its wallet and then issue some papers.

7.5.2 Approve chaincode definition for the channel

Let's see what happens if we try to install this new chaincode only on RatingAgency's peer and then approve its definition on the channel. The following commands are executed on RatingAgency's peer:

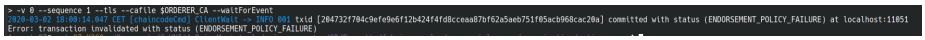

Figure 7.1: Commit proposal refused by the network.

```
peer lifecycle chaincode package evaluate.tar.gz \
    --lang node \
    --path ./contract \
    --label evaluate

peer lifecycle chaincode install evaluate.tar.gz

export PACKGE_ID=#


peer lifecycle chaincode approveformyorg \
    --orderer localhost:7050 \
    --ordererTLSHostnameOverride orderer.example.com \
    --channelID mychannel \
    --name evaluatecontract \
    -v 0 \
    --package-id $PACKAGE_ID \
    --sequence 1 \
    --tls \
    --cafile $ORDERER_CA
```

Up to now, the steps have not differed from those used previously but the commit phase requires a more detailed discussion.

```
peer lifecycle chaincode commit \
    -o localhost:7050 \
    --ordererTLSHostnameOverride orderer.example.com \
    --peerAddresses localhost:11051 \
    --tlsRootCertFiles ${PEER0_ORG3_CA} \
    --channelID mychannel \
    --name evaluatecontract \
    -v 0 --sequence 1 --tls --cafile $ORDERER_CA --waitForEvent
```

When issuing the `commit` command the `peerAddresses` and `tlsRootCertFiles` are used to specify peer's parameter. In summary, for now we have installed and approved the chaincode only on the new peer and then approved its definition on the channel.

Figure 7.1 shows that the channel's endorsement policy has not been satisfied so the chaincode definition has not been approved for the channel. The default policy used in this network requires the definition of a chaincode be approved by the majority of organizations participating in the channel, to satisfy this requirement we need to install the chaincode on one more peer, let's suppose DigiBank's peer. The steps are the same as before but instead of creating a package for the chaincode we can move the one just built on the RatingAgency's peer. Figure 7.2 shows the result.

The final commit command is:

```
peer lifecycle chaincode commit \
  -o localhost:7050 \
```

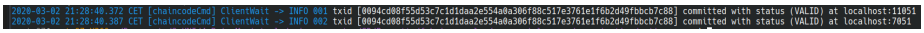

Figure 7.2: Commit proposal accepted by the network

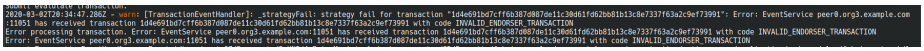

Figure 7.3: Execution proposal refused by the network

```
  --ordererTLSHostnameOverride orderer.example.com \
  --peerAddresses localhost:11051 \
  --tlsRootCertFiles ${PEER0_ORG3_CA} \
  --peerAddresses localhost:7051 \
  --tlsRootCertFiles ${PEER0_ORG1_CA} \
  --channelID mychannel \
  --name evaluatecontract \
  -v 0 --sequence 1 --tls --cafile $ORDERER_CA --waitForEvent
```

7.5.3 Execution

Now that the chaincode has been approved on the channel we can see what happens at this point if we try to launch an application that uses this new chaincode.

In figure 7.3 we can see how HLF does not allow a peer to perform a chaincode-to-chaincode invocation if the chaincode it wants to reach is not installed on itself. To have a successful execution it is necessary to install the `PaperContract` chaincode discussed in 5 also on the RatingAgency's peer. Follow the usual steps except commit on the channel as it has already been done.

Figure 7.4 shows the result.

Figure 7.4: Successful execution of a cross-chaincode invocation

Chapter 8

Final considerations

HLF is a good solution for businesses that want to manage their relationships with the help of blockchain and smart contract technology.

In addition to enabling the developing of contracts and applications using general-purpose languages, other great advantages of this technology can be identified in the management of privacy and the peculiarities of contracts.

First of all, the fact of being a permissioned blockchain allows administrators greater control over the participants, allowing to have a minimum trust relationship between the involved parties.

In addition to this is possible to update contracts after they have been published, thus enabling organizations to correct or eliminate incorrect contracts. A further peculiarity found in HLF contracts, starting from its second version, is the possibility of defining customized versions for the various organizations, provided that they respect a common interface; this possibility can be exploited to implement personalized controls or to allow different organizations to use different languages for the development of their projects.

One of the biggest disadvantages I faced was finding guides other than the official documentation. In many circumstances, the only way to solve the problem was to use the chat made available by the Hyperledger consortium. Another problem I encountered was the great complexity of creating and configuring a network, without the scripts made available in the example repository it would have been extremely complex to correctly configure the nodes; this problem could be even more exasperated for organizations wishing to introduce this tool into their business processes.