University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Computer Science	Spring/ Autumn semester, 20..... Open
Author: Simen André Jakobsen	
Programme coordinator: Hein Meling Supervisor(s): Hein Meling and Rodrigo Q. Saramago	
Title of master's thesis: Design, Implementation, and Evaluation of Academic Credentials on the Hyperledger Fabric Blockchain	
Credits: 30	
Keywords: Distributed Systems, Blockchain, Private Blockchain Verification, Consensus, Academic Records Verification, Hyperledger Fabric	Number of pages: ……117…… + supplemental material/other: …14… Stavanger, 22.06/2020

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Design, Implementation, and Evaluation of Academic Credentials on the Hyperledger Fabric Blockchain

Master's Thesis in Computer Science

by

Simen André Jakobsen

Internal Supervisors

Hein Meling

Rodrigo Q. Saramago

June 23, 2020

"Certification from one source or another seems to be the most important thing to people all over the world. A piece of paper from a school that says you're smart, a pat on the head from your parents that says you're good or some reinforcement from your peers that makes you think what you're doing is worthwhile. People are just waiting around to get certified."

"A Conversation With Frank Zappa" by Dave Rothman, in Oui (April 1979)

Abstract

Hyperledger Fabric is a novel blockchain technology platform that is modular and adaptable in terms of applying it to use-cases: The architecture is modular, which makes it accommodate a variety of use cases such as consensus, privacy, and memberships. The smart contract is adaptable to the architecture too, making the data-modeling very flexible. The benefit of this framework relies on its ability to remain private by both separating and merging transaction ledgers from different network participants. Furthermore, we aim to create a solution for verifying academic records in a decentralized manner. With the use of blockchain technology for use cases like this, we establish a new way of gaining trust, transparency, and accountability for those entities that issue academic records. Though not as transparent as permissionless blockchain systems, we seek to see the benefits of choosing a system like Fabric over systems such as Ethereum. Our smart contracts are designed to be modular and has support for the modularity of the architecture in Hyperledger Fabric. Nevertheless, the modularity comes at a cost of an increased amount of configuration required to make the network work. We share some of our experiences on how working with Hyperledger Fabric is like, which we believe is important for future decisions to decide to use Fabric or not. Moreover, we attempt to address this by assessing different common issues with frameworks like these - namely, the configuration needed to provision blockchain systems to those who want to adopt it. However, we also experiment with the framework where we measure the impact of the different features of the Fabric chaincode package, and how they affect the execution times.

Acknowledgements

I would like to thank my girlfriend Camilla for holding out with me during times of Corona and the great support she gives me in what I do in life. I then dedicate a big thanks to my parents for always giving a second thought in finding the motivation to get back on track in what I do in life. Further, I am greatful for the useful discussions with my supervisors Hein M. Meling and Rodrigo Q. Saramago.

Contents

Abbreviations

HLF	**H**yper**L**edger **F**abric
MSP	**M**embership **S**ervice **P**rovider
POQ	**P**roof of **O**f **Q**ualification
PKI	**P**ublic **K**ey **Infrastructure**
PTS	**P**roposal **T**ime**S**tamp
CA	**C**ertificate **A**uthority
SAN	**S**ubject **A**lternative **N**ame
PDF	**P**rivate **D**ata **C**ollection
VM	**V**irtual **M**achine

Symbols

symbol	name	description
$H(X)$	Hash function	indicates hashing of X data to hash value Y
$\|\|$	Concatenation	$A\|\|B$ A concatenated with B
Δ	delta	Signifies a change.
$\Rightarrow$	leads to	Doing A leads to B (A $\Rightarrow$ B)
{ }	HashSet	Unordered set of items
[]	List	Ordered set of items
$< somevalue >'$	Arbitrarily value	Some value T'
α	Hashing power of dishonest node	
θ	Certainty of block commit	$\theta = 1 - \alpha$

Chapter 1

Introduction

1.1 Motivation

Today certified documents are mostly paper-based. The validity of a physical document is usually provided through a process of certification. This certification typically involves humans working in a trusted centralized entity that vouches for the validity of the material itself. However, these processes are time-consuming and are prone to human errors and fraud. Many organizations today provide digital signatures to facilitate signing of documents in the digital world in a centralized manner. *Durumeric, Zakir, et al* [1] showed that this centralization has in the passed been shown to be unreliable. Moreover, centralized systems also makes it hard to recover and verify these documents - especially in between computer systems as they would need to adapt to each other. With the help of blockchain, one can address this problem by imposing a global state and execution of storing documents as records. The immediate benefit of this is the reduction of time-usage, costs, and a comprehensive agreement on how to distribute academic documents. Hyperledger fabric utilizes the concept of a permissioned blockchain, unlike other systems such as Ethereum, which is open for anyone. Together with encryption, access control, and chain code one can create closed systems for academic entities to communicate authorized updates to records, which will be stored globally while providing a tamper-proof transaction history for all actions done in the system.

1.2 Problem Definition

Rather than relying on the centralization of information systems and integration across different entities, one can use the blockchain as a global ledger to perform integrity checks. This technology can be applied to the process of verifying documents used in real-world

processes that are fundamentally transactional and adheres to trust and ownership. More specifically, the *BBchain* project aims to model such a system in regards to academic records verification. Currently, some countries have strict processes (and costly for the subject), which applicants will have to go through to certify their educational history [2]. The lack of a global consensus of how academic records[3, section VII] are processed and verified does indeed impose several problems. One of them is concerning trust - how can an educational entity trust domestic entities when there is no universal protocol to provide Proof of Qualification (POQ)? As a result, the process of verifying such documents require human interactions accross different organizations to trace the issuing and validation of academic records. As more people are involved in the verification process it does require time and human resources, which will impose costs to the issuer of the document and the verifier of it. Moreover, the process of checking integrity relies on the systems of those who issue the academic record. What if a student was able to tamper with the registered grades, and no repudiation system was implemented? Then the verification part on the other side will fail as there will be no record of it. As a result, this is where the POQ would fail indeed. If the POQ was tampered with from its original source - how would the intermediate organizations know this? Consequently, situations where these verifiying entites will accept a forged academic record without any guarantees of it being in its original form.

With the help of Hyperledger[4] we can design a closed system where educational instances all over the world can join and run a standardized set of processes to handle academic records. Fabric utilizes the Public Key Infrastructure (PKI) for identifying authorized entities that can do operations in the system on the blockchain shared would indeed help in solving the problem in regards to attestation of academic records across different organizations. Preferably than having many centralized servers that hold documents for each region, we propose a system developed with a private blockchain infrastructure with policies and processes (code) that will be shared by all participating universities in the network. We argue that this makes the process of verifying documents easier: participants of the system will be able to prove a *claim* of grades, certificates or other qualifications made by students. Moreover, other universities does not need to reveal the original the document itself to other universities as we represent them using cryptographic has functions. These functions allows for representing a document uniquley by its content and yet verifiable if some other entity put the same document through this function.
Nevertheless, as everyone uses the same system: we argue that all universities participating in the network can verify any document claimed by some student reduces the time needed for verifying the original and its source. Therefore, the required buracreacy by having trusted third party entities that vouches the validity of documents is no longer required. Thereby also reducing the time needed for proving academic records.

In this thesis we aim to reduce the process of ensuring that an academic record is verified without all the bureaucracy of intermediate organizations involved. However, we are not solving the probability of external corruption. Such as dishonest and fraudulent behavior of users of a computer system, this is also known as the oracle problem. In short, the oracle is some real-world entity outside the system which inputs data into the system. The oracle is the entity that empowers the blockchain's view of the external world [5]. So if the oracle gets compromised, so does the system. We therefore assume that this must be controlled by external systems, and not the blockchain itself. The scope of lost or stolen identities is therefore not up to the blockchain system to solve as it will only be a distributed and immutable transaction log.

1.3 Blockchain Usecases and Examples

Commonly, when hearing about Blockchain, words like cryptocurrency and bitcoin comes to mind. Though, we would like to introduce the concept of Blockchain as a Service (Baas). Primarily, the principle builds upon creating applications for real-world scenarios using the strength of the blockchain technology. Then deploy this to a server-group which is distributed all around the world, like in the cloud. The most important thing about the BaaS principle is that it gives the strength and the completeness of the blockchain technology - namely, the consensus part where all nodes should have the same value, and an immutable log of events. These two combinations give the participants in the network: *security, transparency, and agreement*; the first one being that *none can change* the log without having to convince the majority of nodes, secondly - *everyone sees* the changes done, lastly - everyone should have the *same* view of what happened [6]. These three attributes are essential for many real-world cases, as we will describe in this section.

A South Korean University created a distributed ledger using blockchain for students to verify their academic records as a result of the Coronavirus. This mitigated the need to attend physically to an office to interact with other humans with the potential of being infected by the Coronavirus [7]. Furthermore, in current times where climate change is an emerging threat to humanity and infrastructure. One can decouple the need for centralized data and rather store it into the blockchain. The idea behind a blockchain is that it contains a chain of events that occurred to the state, e.g., a change of bank account balances. All transactions are put into blocks. If any of these blocks are changed, then the next event will be invalid as blocks reference eachother through unique addresses that are represented by the transactions the block contains. As a result, blockchain

makes it possible to have a common perception of the truth in an open and distributed log that can be used to verify claimed truths from un-trusted sources.

This kind of distributed and immutable record log will help us protect essential data such as academic records, health journals, drivings licenses, and more from disasters or corruption. Moreover, refugees who come to new places might only have a paper-like scheme that describes their history from their home country which makes it hard to trust. A blockchain system to verify their attributes would indeed give them more credibility in terms of applying for a visa, working career, and establishment.

In terms of private and public sector, there are challenges in terms of reaching consensus with the contracting of shared assets and agreements. In this context, a common approach for contracting is that lawyers/diplomats/politicians from both parties are joining together in meetings in order to define the content of the contract. Once done, the task can start, and upon finalizing of some project one will need to file in that the contract is finished. Upon finalization, there is a significant amount of bureaucracy involved to ensure that all involved parties have met the scope of the agreement. This includes verification of all contracts to ensure share the same definitions of the scope. E.g. requirements, finalizing date, budget, etc. This would indeed impose extra time and cost to all involved parties. Having a shared ledger where all participants have clearly stated their role and ability to change the state of the contract would indeed help in lower costs and time spent to agree on the final result.

1.4 Contributions and Outline

This thesis will in summary make the following contributions:

- We propose a general purpose document verification using a modular data model which allows for cooperation between organizations, using the Hyperledger Fabric framework. We also attempted to fulfil the Decentralized Identifiers (DIDs) specification.
- We give a thorough insight on how Hyperleder Fabric Works and how it compares to unpermissioned blockchain technologies.
- We address one of the bigger issues with distributed systems - which is configuration of large networks. We surveyed multiple implementations of how to provision private blokchain networks (like Hyperledger Fabric) - and found better ways to improve the configuration Fabric proposes in their examples.

- We analyse our implementation by deploying or architecture on powerful cluster to assess different parts of how the Fabric system works.
- Sharing our experience working with Fabric - the amount of required technologies to use the framework makes it extremely coupled to work with.

The remainder of the this thesis will be like the following.
Chapter 2. Gives the relevant background and comparision in between permissioned and unpermissoned blockchain systems.
Chapter 3. Gives insight in the basic technology stack and how it is like to get and running with Hyperledger Fabric.

Chapter 4. Provides relevant information about the BBchain project and other relevant litterature studies to further emphasize the difference in between the permissioned and unpermissioned networks.

Chapter 5. Will cover the design and overview of decided system architecture.

Chapter 6. Provides an experimental evaluation of the Fabric framework using our implementation.

Chapter 7. Describes lessons and experiences with learned while working with fabric.

Chapter 8. Concludes our work and gives future directions.

Chapter 2

About the blockchain

This section gives an introduction to the concept of blockchain. Then we will discuss two different blockchain system models. Namely, permissioned and unpermissioned systems. On the latter, we'll provide insight into the theory to establish a difference amongst these two schemes. We will define the concept of how unpermissioned systems work in general and will not deep dive into the subject as this is out of the scope of this thesis. Then, we will define the permissioned model and mention how Hyperledger Fabric work, and provide theory into core concepts which *Hyperledger Fabric* utilizes in the system. This section will define the theory needed to understand the concepts which this paper will address during its evaluation.

Unpermissioned(General) vs. Permissioned (Fabric)		
Aspect	**Permissionless (General)**	**Permissioned (Hyperledger Fabric)**
Fault Tolerance	Tolerates $2f + 1$ Byzantine Fault Tolerance or PBFT	Fabric Tolerates only Crashes (at least half of the network must be up), some systems provides PBFT.
Consensus	Incentive-based "race" to propose blocks	Specific nodes handles transactions through traditional consensus protocols
Trust	Majority need to be honest and agree on proposals	Only authorized entities can do operations
Architectural model	Inherently Distributed (anyone can join)	Authorized nodes with specific role to complete the underlying consensus model
Other	Smart Contracts is executed on all nodes - majority must agree on execution results	Policy-centric - all nodes does not have to execute the same transaction for it to be included in a block, only a subset of peers defined in a policy

Table 2.1: Brief description of differences.

2.1 The blockchain

Blockchain is known for being inherently distributed in terms of its architecture. The main idea is to run program executions on all peers in the network and not rely on a centralized entity to do the job. Once execution finishes, all nodes will commit their view of the current world state. This world state can be seen as a global transaction log. The purpose is to maintain a record where the majority agrees on all transaction proposals. In this chapter we want to emphasize on why this concept works and how it can be utilized in a *distributed* manner and for both permissioned and unpermissoned system model assumptions. However, they are both supposed to be distributed. In computer science one will have to preserve that network nodes are always doing the same (Consistent), always ready (Available), and tolerate the system gets segmented (partitioned). Though ensuring all of these properties is a fundamentally hard problem to solve, which leads us to the CAP-theorem.

Theorem 2.1. *The* CAP *theorem [8] states that it is impossible for a distributed system to guarantee (C)onsistency, (A)vailabity, and (P)artition Tolerance. Distributed systems can give following combinations of guarantee CP, CA, or AP.*

Following the theorem above, the *Consistency* in the context of the blockchain is that it will avoid to have two views of the world-database. The *Availability* is when transactions submitted by users is permanently stored in the chain and available to everyone. *Partition Tolerance* in a blockchain context refers to the ability for the system to continue operate by committing blocks despite a fraction of crashed/malicious nodes. Permissioned and permissionless Systems focuses on preserving Availability and Partition Tolerance (AP) and eventual Consistency, but cannot guarantee all. Nevertheless, unpermissioned assumes any entity can join the network, whereas permissioned only allow for a certain set of defined identities to join. Unpermissioned systems must carefully consider that malicious behavior can happen and break the properties of CAP, hence they tolerate that only a certain threshold of dishonest nodes. On the other hand, permissioned models (such as Fabric) only allows for nodes to crash as dishonest behavior will be identified through digital identities.

2.2 The Hash Chain

A hash chain [9, Section 3.2] is a data structure that can be viewed as a linked list. Each node is referred to as a block. Each block B has a pointer $B.prev$ to a preceding block and has a variable $B.prev.Hash$. The node also hold data, $B.Data$ which contains

some representation of an object. Each node will have a hashing address of its own concatenated with the previous block. A hash function has the following properties [10, Chapter 11]:

1. Given some data D, it is fast to compute $H(D) \Rightarrow X$.
2. It is deterministic, two objects holding the exact same values results in the same hash, e.g. if D = D', then $H(D) = H(D')$
3. Given a set of distinct data objects S, where $s \subseteq S, s = \{D', D''\}$, then $\forall D'D''$ there exists no solution s.t $H(D') = H(D'')$.
4. A small change in D should yield a new uncorrelated hash value. I.e. given $D, H(D) \Rightarrow X \rightarrow \Delta D \Rightarrow H(\Delta D) \Rightarrow Y$, where $X \neq Y$.

The hash pointers will represent an address to a certain block in memory. The definition of a block address is given as follows:

1. A genesis block is the initial block in the chain. It's $B_0.prev = nil$. Its $B_0.Hash$ field will therefore be defined as $H(B_0.Data)$
2. Any subsequent blocks B_n of the genesis block will have $B_n.Hash$ defined as $H(H(B_n.Prev.Hash||B_n.Data))$ [1].

Theorem 2.2. *If hash functions with properties* $1-4$ *is used in a hash chain. And all the blocks in the chain points to the previous block through hashing addresses. Then tampering with any block* B_n *which is already referred in some* $B_n.Prev.Hash$ *would require to change all subsequent blocks.*

Proof. Let's assume that we have a hash chain $C = B_4 \rightarrow B_3 \rightarrow B_2 \rightarrow B_1 \rightarrow B_{Genesis}$ and $B_1.Data$ was changed by some entity. Then, according to property 4 the $B_1.Hash$ would change significantly. As a result, the block B_2 would no longer hold a valid pointer address to the previous known block. Therefore, the hash addresses for subsequent blocks $H(H(B_n.Prev.Hash||B_n.Data))$ would also be invalid and blocks must be changed accordingly. □

With these properties we can maintain a log of transactions that are included in blocks that are pointed to each other using the Merkle tree root hash. Changing any transactions will break the reference of the block(s), as a result these would indeed require some entity to touch all subsequent blocks in order to produce a valid hash chain.

[1] Some implementations do include more fields concatenated with each other. Such as timestamp etc.

2.2.1 The Merkle Tree

The Block.Data field is usually structured as a binary tree. This is to ensure that each block can hold several transactional data points which can later be retrieved in an efficient way. Each intermediate node in the Merkle tree is calculated with $H(hash_{leftChild}||hash_{rightChild})$ whereas the leaf nodes contains the data.

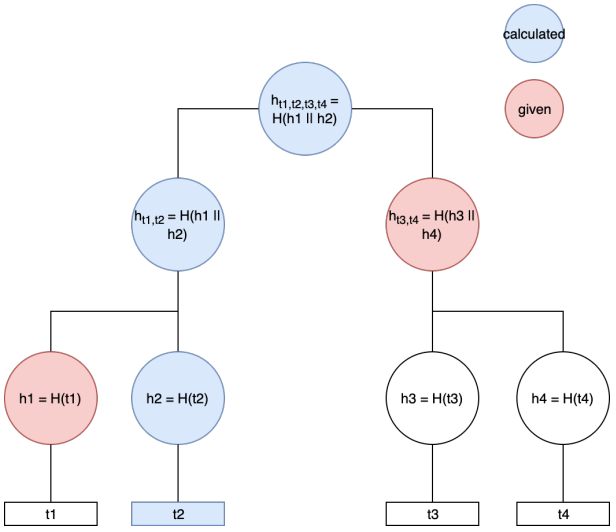


Figure 2.1: Verification of transactions in Merkle tree.

The Merkle tree [11, Chapter 1.2] allows for committing of multiple values to a block. It can be viewed as an optimization for multiple reasons. The first one being that rather than publishing $[H(t_1), H(t_2), H(t_3), H(t_4)]$ one can publish a Merkle root hash $H_{merkleroot}$. Furthermore, coupling transactions together into one root opens for a quicker verification of data. As showed in in figure 2.1. To verify t2, one have to re-compute the blue nodes and retrieve the red nodes. The new root node would then be compared with the old one. If they're equal we know that no transactions has changed.

Lemma 2.3. *Let H_r be the root of the Merkle tree T_{merkle}. The Block $B_n.Hash$ will point to H_r, such that if any nodes in T_{merkle} changes, then also the value of H_r changes.*

Proof. Let's assume a hash chain where each block B_n has a field $B_n.Data$ which holds several transactions in a Merkle tree. The $B_n.Hash$ points to the root node of the $B_n.Data.Root$. A small change in any of the nodes would indeed change the $B_n.Data.root$ significantly. Resulting in invalid hash addresses of subsequent blocks. □

Definition 2.4. From Lemma 2.3 we can use the tree as an optimization to the Hash Chain as any changes will be reflected in the root hash. Theorem 2.2 still holds.

The other benefit of the Merkle Tree is that it allows for *Proof Of Membership*. By that we mean that if some transaction T_x is claimed to be in some block. Then the system

can verify this by only assessing those nodes on the path from the leaf of T_1 itself up to the root. Which is approximately O $(log(N))$. Ethereum and Hyperledger uses Merkle Tree in order to achieve multiple transactions into one block. The implementation of it is not a pure Merkle Tree, but a *Merkle Patricia Trie*. However, the basic concept is still the same.

2.2.2 Reaching Consensus (Agreement)

The blockchain is essentially a distributed hash chain of states which cannot be modified once committed. However, just distributing a data structure is not enough to enforce that the state *cannot* be modified by some entity. There needs to be some consensus amongst the peer nodes in order to achieve the same agreement.

Definition 2.5. *Consensus protocol.* There exists N nodes in the network. They want to propose some value that all nodes should agree on. The consensus protocol must enforce the following interface and properties [12].

Module 2.1 Consensus

Module:

Name: Consensus, **instance** c.

Events:

Request: ($\prec$ PROPOSE, r | value v $\succ$): Attempt to write a value.

Indication: ($\prec$ DECIDE, r | value v $\succ$): Commit value.

Properties:

Termination. All behaving nodes should agree upon the same value to commit.

Validity. All proposed values must source from behaving nodes.

Agreement. No nodes should decide differently.

Integrity. No node commits the same proposed value twice.

The *Agreement* property of the Consensus protocol is vital for keeping the integrity of the blockchain. Together with *Termination* and *Integrity* ensures that every honest node should only commit some proposed value once and no nodes should commit a different value at some time T. The *Validity* will enforce that no node will suddenly commit a value invented by itself [12]. However, in a open system one must carefully consider Byzantine Faulting nodes. These are nodes that might not behave according to their specification. As a result, the network must ensure that they agree on identifying these nodes to agree on proposals only from honest nodes. A system model like this assumes that $3f + 1$ nodes are behaving according to their specification. The term "time" in Consensus algorithm is abstracted to *epochs*. An epoch is a round of *proposals* and *Decide* messages. The reason for this is that distributed systems are usually all around

the world. Due to the nature of CPU clocks and the lack of synchronization of clocks amongst computers, the abstraction of time into epochs is needed.

Enforcing these properties is where the permissioned and the unpermissioned schemes diverges in terms of reaching Consensus.

2.2.3 Identifying Clients That Interacts Within The Consensus

Both System Models adapt a mechanism for proving the ownership of some identity. For unpermissioned systems these are identified through public hex-addresses whereas for Fabric, there exists an address tied to a name, more specifically the X.509 standard. Though at its core, they both utilize concept of having a Private $Key_{private}$ and a Public Key_{public} key. These two pairs of keys are pre-selected before the clients issue transactions towards the blockchain. For fabric this is done through Certificate Authorities, whereas for unpermissioned systems they have client applications which does so [13]. The transformations of these is dependent on the underlying encryption algorithm. Furthermore, the Key_{public} can be known to everyone and $Key_{private}$ has to stay secret to the entity which holds it. The $Key_{private}$ is used for decryption($D(PrivateKey, Ciphertext)$ and Key_{public} is used for encryption($E(PublicKey, Text)$) that produces a ciphertext C. A ciphertext is essentially a scrambled text hidden by the encryption algorithm. In order for it to be secret there needs to be some assumptions. First, the private key is kept secret. Secondly, it must impractical to deduce the original text T from C without knowing $K_{private}$. Lastly, even though knowing the encryption algorithm one cannot determine the original text T.

Scenario 2.1. Let us assume that entity A wants to forward a secret value X to entity B. A knows the $Key_{publicB}$ and does the following operation $C = E(Key_{publicB}, X)$. A then forwards C to entity B. Then B can get X by following operation $X = D(Key_{privateB}, C)$. However, an adversary can try to deduce T' with $Key_{PublicB}$ and attempt to find $Key'_{PrivateB}$ such that $T' = D(Key'_{PrivateB}, C)$ where $T' == T$.

Given the Scenario 2.1, even if the adversary tries to find the private key, the best attempt is to search through the whole possible key-space and attempt to find the right private key. In practice, with current computer systems - this would require that the adversary spend significant amount of time to attempt to find the right key that upon finding the right private key $Key_{PrivateB}$, it will already be deprecated and no longer used.

Since we are now able to hide information, we still cannot prove its authenticity, i.e., that it has not been tampered with by some adversary. As a result, it is possible to reverse the operation. By reversing we mean that entity A can *sign* the value by hashing the

value X and then encrypt it with $Key_{privateA}$, s.t. $M = S(Key_{privateA}, H(X))$. Signing here is when A uses its private key to encrypt the message and others can reverse the operation using $A's$ public key. Upon receiving the message M, entity B can verify if entity A really sent the message by calculating $H(X) = V(K_{privateA}, M)$. Since A keeps the private key secretly, no one else can create this proof. B can then verify the original message $H(X_{received}) == H(X)$, if true,then B can be sure that A sent the message and that it was not tampered with. This is called *Digital Signatures* [10]. This is what Fabric is relying on in its trust model of the system together with encryption of traffic. Unpermissioned systems also do this, but only to identify client users with the help of digital signatures.

2.3 Unpermissioned Blockchain

This scheme runs a public ledger where anyone is eligible to join and participate in the network. The main challenge is that there is no specific identity tied to the different entities in the system, e.g. name, address, birth date etc. Which could be used for auditing purposes or other system-like decisions to detect non-behaving nodes. Typically, each node and client is only represented by some address. Creating a consensus protocol which follows Definition 2.5 is challenging as one cannot trust the network since it is open for anyone to join it. More specifically, how does the network prevent faulting/arbitrary nodes in proposing values?

2.3.1 The Architecture

As mentioned, the architecture of unpermissioned systems is open for anyone to join and participate regardless. As a result, the system is inherently distributed. However, there is typically roles amongst the nodes which defines their behavior.

- *Validators (Peers).* Enforcing the *Validity, Integrity, Agreement* properties of the Consenus protocol.
- *Proposers (Miners).* Proposes values whenever they are eligible to do so.

The typical pattern [14] [15] of interaction with the unpermissioned blockchain is depicted in the figure below. Transactions are usually issued from the outside world and is typically a broadcast. The Peer nodes will then store this transaction and will work out the consensus protocol. There are nodes which *Proposes* values and *Validators* which verifies if they agree on the proposals (Can act as both). A proposal in this case would be a

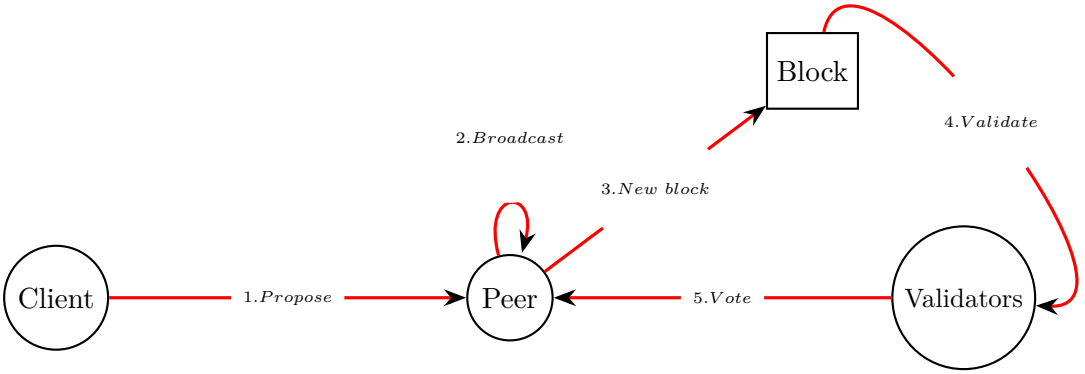


Figure 2.2: Some entity issues a transaction in a unpermissioned system. A vote is considered valid once 2/3 of the network agrees on the final solution. In step 2 nodes broadcasts to their neighbouring nodes such that everyone in the network finally sees the transaction. Neighbouring nodes is also peers, but becomes validators.

block which holds transactional data (using a Merkle Tree) and references to the previous block.

2.3.2 Consensus Protocol

Unpermissioned systems such as Ethereum infer a consensus protocol called *Proof Of Work* [16]. This algorithm exploits the properties of cryptographic hash functions in order for nodes to be allowed to propose *blocks*. The focus here is to impose an incentive on the peers before they are allowed to propose a value. The basic concept is to brute-force a cryptographic hash function in order to achieve an acceptance by the network.

Definition 2.6. Let H be some cryptographic function used in a Proof-Of-Work scheme, $Hashfunction(Nonce, []data, Block)$ which calculates $H(nonce||B_{n-1}.Hash||D_1||D_2||..||D_n)$. We then define a hexadecimal puzzle $P = Hashfunction(Nonce', []data', Block')$. P must satisfy the target hexadecimal value d. If P falls into the target space $< d$, and $Nonce', []data', block'$ are valid. We then allow a peer to propose a value. Else, reject proposal and wait for a valid P. The target value d can be adjusted accordingly (larger or smaller search space).

The nonce is a randomly (and independently) value picked for each try, we then have independent trials for success which gives $1/p$ probability. Based on the definition above a node will require to take all the data it has collected for a particular block b and try to compute the puzzle with the help of a nonce. The process of making a proposal can be viewed as a race between nodes. The node that is able to find the right hash-value with some nonce can propose a block. If too many nodes gets *too strong* there is possibility

to increase the puzzle spaces such that the nodes require more time to produce blocks. Reducing the likelihood of getting block proposals at the same time. This is common for unpermissioned systems whenever they observe that the frequency of stale blocks etc is occurring at faster rates as this would signify that more nodes are getting stronger.

All other participating nodes can then verify different attributes of that particular block. So if any node changes that data, it will eventually be detected as the hashes in a Merkle tree would not match. The participating nodes will also verify whether the nonce hashes to the same provided result. If everything verifies, it can be committed. The takeaways from the Proof of Work scheme in short is:

- *Free of Censorship.* Another node cannot prevent others in calculating the Proof Of Work.
- *Failure resilient* Failing nodes won't affect others in calculating a PoW.
- An adjustable difficulty to number of Proposals.
- Calculating a PoW requires resources (in terms of Hashing Power).
- Each participant has equal likelihood of proposal for each epoch.
- A progress-free puzzle, the previous result doesn't help in finding the final solution.

The key take away here is the incentives that the Proof Of Work gives. Hashing Power is proportional to the computing power of a machine. This means that resources are spent in order for a miner to consider it profitable. Altering transactions and providing a fake nonce, would indeed result in an invalid proposal. As a consequence, the miner has lost resources (in terms of energy consumption). Most blockchain systems impose*Block Rewards* which is typically a fixed value and transaction fees. As a result, it will form an encouragement for the nodes to behave because if not, it wouldn't be profitable. See algorithm below.

Algorithm 2.2 Miner Decision Algorithm

Result: Propose? True/False

propose = nil;

if *(Block.Reward + Block.Transactions.Fee) > (hardware.Cost + cooling + electricity)* **then**

| propose = True;

else

| propose = False;

end

2.3.3 Chain Rules

Systems such as Ethereum defines some basic rules that each proposer should follow.

- Once a Proof-of-Work solution is found, publish the block immediately.
- Always extend the longest known chain in the network.

These rules can be exploited. An attacker that breaks the first rule is performing a *Selfish mining attack*. In essence, the attacker doesn't publish his blocks and hopes to generate a chain that is longer than the longest public known chain. The attacker will therefore *fork* out from the public chain.

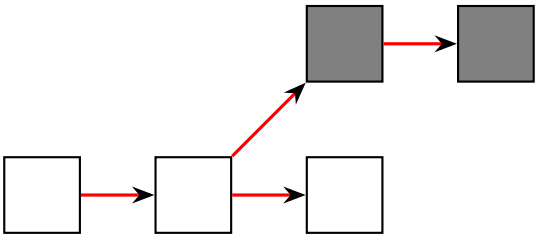


Figure 2.3: Selfish miner keeps a secret chain without broadcasting newly mined blocks.

Once the Selfish miner's private chain is either equal or longer than the public chain, he publishes it and hopes that others will adapt the secret chain as the new longest chain. The immediate benefit of this attack is that selfish miner will get a larger proportion of block rewards. According to [17] a selfish miner can profit with this attack by having a hash power $\alpha \geq 30\%$.

Definition 2.7. We say that an entity with α hashing power gets proportional amount of blocks according to its α. Given that each committed block represents an epoch, then the expected amount of blocks after X epochs is given as $E_{blocks}(X) = \alpha * X$, where $0 \leq \alpha \leq 1$ and X is strictly increasing.

Moreover, when an attacker achieves $\alpha >= 51\%$ power in the network it is possible to perform a *double spend attack*. This allows the attacker to changes values of transactions in the block and then ensure that peering nodes accepts this (since the attacker controls majority of the voting).

Theorem 2.8. *A unpermissioned blockchain can only give probabilistic guarantees that a block is eventually committed.*

Proof. Let's assume a selfish miner with hashing power $\alpha = 51\%$ starts to create a secret chain c'. Let's assume the number of epochs $\gamma = 1000$. Then the expected amount of

blocks for the selfish miner is 510. At some point, the selfish miner will be in the lead and is able to publish c'. For all received transactions T_X the miner has altered the transactions such that ΔT_X. We then have a series of blocks that breaks the integrity of the original data. As a result, we can only rely on honest nodes $\theta = 1 - \alpha = 0.49$. We then say that we are θ certain that the block actually gets committed in its original state. □

Even though this attack is possible it is yet hard to perform in practice. The reason being that it is simply not profitable enough for any entity to do so, being honest is more profitable in the long run, see Algorithm 2.2. This means that once we say a block is committed in a PoW-scheme, it will only be in a probabilistic manner as there is some chance that the integrity of the chain will be broken. We can say that the properties defined in definition 2.5 is ensured through incentives for BFT-nodes by imposing a dilemma on the network. The PoW scheme enforces *Validity* by letting a node propose when it claims that a solution to the puzzle is found. If the network doesn't validate the proposal, then the proposer has wasted resources. *Integrity*, if a node tries to issue the same transaction twice, then it would be detected through the Merkle tree, such that the participating nodes can reject the proposal. The *Termination* property is enforced by the longest chain rule. If a block b' is contained in a chain c' and there is sufficient subsequent blocks committed to the chain, we can be confident that honest nodes will eventually agree on that particular block. The *agreement* property will only hold for the majority of honest nodes (over 50 percent). Furthermore, the also PoW-scheme ensures that the likelihood of concurrent proposal of blocks is small. However, this will no longer hold at the moment there is a powerful selfish miner attempting to rewrite the hash chain history.

2.3.4 Storing sensitive data in a blockchain

As mentioned earlier, anyone can run and participate in the public blockchain network. In order to join the network one would require a private and a public key $K = [K_{private}, K_{public}]$ such that the user can identify itself through the public key infrastructure that the underlying blockchain technology requires. Furthermore, any business logic coded into these systems will be available to anyone. Meaning that any transactional information that is stored in the chain will be reflected at all participating nodes in the network. Consequently the common practice is therefore to store such data off-chain and rather use the public blockchain for integrity checks. This means that the process of forwarding documents occurs off-chain. Furthermore, since there exists no central authority that vouches keys it is also harder to manage those cases where

the client loses the key combination. A question that arises then is: How is the keys recovered and if not, how to ensure that the old key combination tied to specific data can be reassigned to the new keys.

2.4 Permissioned Blockchain

The permissioned blockchain maintains a list of identities which can participate within the system. Unlike the unpermissioned system, there is no block propagation with a nonce which is expected to match a certain (PoW) difficulty. In the beginning of this chapter we abstract the core concept of committing blocks without the help of a Proof of Work-scheme.

2.4.1 Consensus Protocol

The system model in the permissioned blockchain assumes that there is N participants within the network. Typically these nodes can propose and decide values with the help of digital signatures. We therefore abstract this scheme as *Proof of Certification*. This means that for each proposal of blocks, in order to decide, there must be a majority of nodes that agree (with digital signatures) to the state change to the system. Depending on the system model we can define a certain threshold on how many nodes that need to agree on some proposed value. For a Crash fault tolerant model (CFT) we can tolerate that N/2 of the network has crashed and the required amount of votes cast by the network to be:

$$S_{cft_{min}} = \left\lceil \frac{N+1}{2} \right\rceil \tag{2.1}$$

Whereas in the Byzantine Fault Tolerant (BFT) model there can be f faulty nodes and the network must consist of n nodes where $N = 3f + 1$ to tolerate f. This model require S_{bft_min} of signatures in order to decide and commit a block. We define $S_{bft_{min}}$ as an equation [18]:

$$S_{bft_{min}} = \left\lceil \frac{2N+1}{3} \right\rceil \tag{2.2}$$

This means that at least 2/3 (in BFT) or 1/2 (in CFT) of N nodes in the network will have to agree on the same value before deciding to accept the new block. Upon agreement (either following $S_{cft_{min}}$ or $S_{bft_{min}}$), we say that a block is *confirmed*. The block will then contain a field which holds a set of signatures to prove that nodes in the network agreed upon that particular value. Furthermore, a node cannot sign any arbitrary blocks as the validating nodes will eventually detect this.

Definition 2.9. Each block b' must point to a previous block, $b'.previousBlock$. A node will only consider a valid block if $b'.depth$ is greater than previously signed depth d' and $b'.previousBlock$ is a valid block in the ledger L. A node therefore requires $b'.depth > d'$ and $b'.previousblock \in L$.

Following definition 2.9 a node will not sign any blocks which aren't apart of the previous known history or more than one block at the same depth. Furthermore, trying to extend a depth lower than the highest known depth is invalid, this can be seen as the *longest chain rule* in the unpermissioned system.

Definition 2.10. The signatures S collected in the blocks is only valid if S' is a known identity in the system. Upon majority of S_{min} signatures, a block has a *Certification* and is now eligible to be committed to the ledger.

Compared to the unpermissioned model, there cannot exists a fork. The reason being that the underlying model of systems like Hyperledger works differently as they rely on deterministic consensus algorithms like RAFT which helps in determining the order of which transactions should be put in a block. Though blocks need to be valid Hyperledger allows for a fine-grained definition on how different transactions should be considered to be valid before appended to the block. We refer to this as a rule-set. It is common to write a rule-set which only impose execution on a subset of nodes that needs to execute the transaction, and finally send their result to validation nodes. The validation nodes then decides whether the transaction fulfils the rule set, and that the executing nodes agree on the outcome of the transaction. These rule-sets can be defined by a sequence of AND and OR operators. The AND operations in this case are more strict as it would require more nodes to see the transaction and then execute it on their view of the world state. The ownership of execution are proven through digital signatures such that the validation can see whether the requests originated from a node within the rule set. An immediate benefit of this is that it allows for scaling transactions by allowing for Horizontal scaling in the system and eliminate bottlenecks such as letting many nodes execute duplicate transactions. Refer to the transaction flow in the figure below.

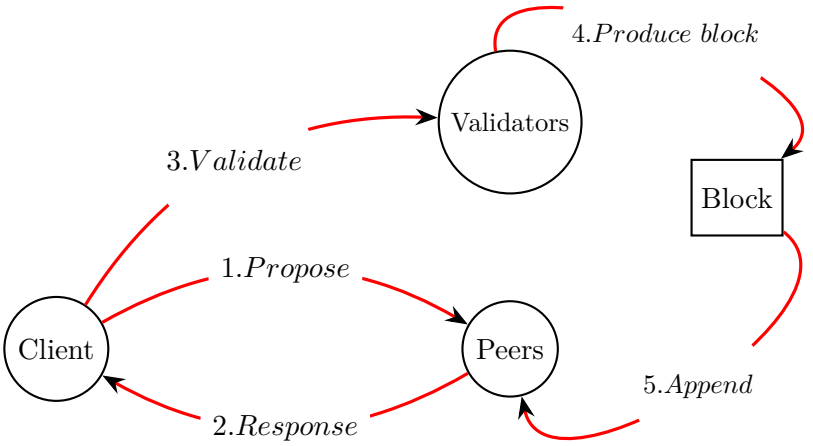


Figure 2.4: Issuing a transaction to executing nodes (peers) and then validate and finally produce a block.

Definition 2.11. Assume a network with N nodes, though only a subset of these nodes needs to execute a transaction. We can therefore define a rule-set $R = A \wedge B \wedge C$ which requires nodes with identity A, B, and C to execute Transaction T'. Once done, the final result can be forwarded the validating nodes of the network which checks the result of the different executions $T = [T'_A == T'_B == T'_C] = TRUE$. Furthermore, the validators checks the signature field of each transaction in the rule R to verify with the respective public keys of the identities. A rule may be relaxed by using $R' = T'_A \vee T'_B \vee T'_C$ which then require that one of the A, B or C should execute and sign that transaction to consider it to be valid before appending it to the next block.

Following the definition above, once the transaction is considered to be valid it can be prepared to be included in a future block which will eventually be forwarded to the rest of the nodes in the network.
Comparing the permissioned model transaction flow in fig. 2.4 with the unpermissioned in fig. 2.2 shows that the permissioned model has a different way of producing blocks. In the permissionless system, all nodes which takes part in the consensus are allowed to propose a block, and then other nodes (including block proposers) validates the Proof Of Work puzzle and the included Merkle tree within the block. On the other hand, in the permissioned model, there are exists a distinguished set of nodes that performs these tasks. The peers executes the transaction and signs it, but cannot produce blocks, and the validators cannot produce transaction results, but may only validate transactions and create new blocks which is maintained by the peers too.

Two blocks cannot contain the same transaction as this is controlled by the validators. This means that any attempt to execute a transaction on the world-state twice in the same epoch will not be possible.

Proof. Let's assume the majority of the network is behaving with a current state of the ledger $L = [A, B, C, D]$. At some time t' there can be two proposals of blocks to the network. Then, according to theorem 2.9 and a block cannot get majority of vote and no split-votes. As a result, one of the blocks will be discarded and a fork won't happen. □

From above, the block creation under the CFT will always ensure that the ledger will be in a consistent state as there cannot exist two different views of the ledger. Though, the proof above doesn't hold for Byzantine Fault model, the ledger may be corrupted if peers in the network behaves maliciously. Such an attack on Hyperledger Fabric would cause severe outcomes as the official release of the framework currently only supports the CFT model. Hyperledger supports pluggable Consensus protocols [19] amongst the validating nodes. Some of these are *Raft*, which has the following specifications [20]:

Module 2.3 RAFT

Module:

Name: Raft, **instance** c implements Consensus 2.1

Events:

ElectLeaderEpoch: (≺ VOTE | *addr node ≻): Casts a vote for a candidate in epoch.

Follower: (≺ COMMIT | value v ≻): Listen for decisions to replicate log.

Candidate: (≺ NEWLEADER | *addr node ≻): Enroll for leadership.

Properties:

Strict Ordering. All nodes agree on order of a set of transactions stored in one replicated log.

Election Safety. For each epoch, there can only exist one leader

Leader Append-Only. A leader will only append values to the log; It will never delete an entry.

Log Matching. If two or more logs contain the same epoch, value and index, then all the entries are identical.

Leader Completeness. Any given log entry committed in some epoch, will always be present in logs of leaders in higher epoch rounds.

State Machine. A entry in the log in some index will always be the same for each server.

The protocol doesn't handle Byzantine failures, but will handle the common failures such as network partitions, delays, packet loss, duplication and re-ordering [21].

Raft must be initiated by a proposal which initiates an epoch. For each epoch there should only be one leader. The rest of the servers are then acting as *followers* which is passive. By passive, we mean that they don't perform any IO-operations than what the leader tells them to. By this, we mean that the followers never issues request other than replying to both candidates and leader requests. External clients communicates commands (transaction proposals responses from peers) to the leader of the epoch. The leader will the ensure that all followers obeys the strict ordering of commands. It does so by detecting whether the followers are in an inconsistent state (by logs comparison), there are multiple scenarios which RAFT must handle in the followers.

- *Inconsistency.* A follower log is incomplete. I.e. not all epochs are included.
- *Missing Entries.* A follower doesn't have a complete view of the log.
- *Non-existing values.* A follower has values which doesn't exist in the leader's log.

In each epoch, there can only be one new entry with an index which is only provided by the leader. Further, the leader will also perform a Remote Procedure Call known as *AppendEntries*. This is where the leader includes the current epoch and its latest index in log. Upon receival, the follower will return either the corresponding value or nil. The leader can therefore (during normal operation) determine whether the followers are consistent in the logs or not. If the *AppendEntry* message detects anomalities in the log, then the leader will attempt to find the part in log which both the leader and follower agree on. Then overwrite the subsequent log entries at the follower and update it accordingly to the leader's view of the log state. As a consequence, *Log Mathcing* property is preserved. However, if a follower cannot reach a leader it might end up electing itself as a leader, subsequently executing commands which is not consistent with the *actual* leader of the epoch. As a result of this, RAFT imposes a leader election restriction. Lets assume three servers $N = [N_1, N_2, N_3]$ with logs $L' = [L_1, L_2, L_3]$ with respective log entries $l' = [\{1, 3, 4\}, \{1\}, \{1, 2\}]$. During the candidate state, nodes will have to reach out to the majority of network in order to gain votes. All nodes will prefer those candidates which are the most *up-to-date* and ignore those candidates which have lower log entries than itself. Therefore, in this case, nodes N_1, N_3 will silence N_2 during leader election (due to more updated log), and N_1 will precede over N_2. This way, the candidate with *highest* log will be elected as the next leader. This means that candidates with incomplete logs never gets elected, preserving the *Leader Completeness* property. Furthermore, if two or more candidates tries to be the leader at the same time, then they have a *Split Vote*. RAFT solves by letting nodes back off at random times, reducing the likelihood of conflicting leader elections. A follower will only cast one vote for each epoch,

which means that there can only be one leader elected with majority which ensures that *Election Safety* property.

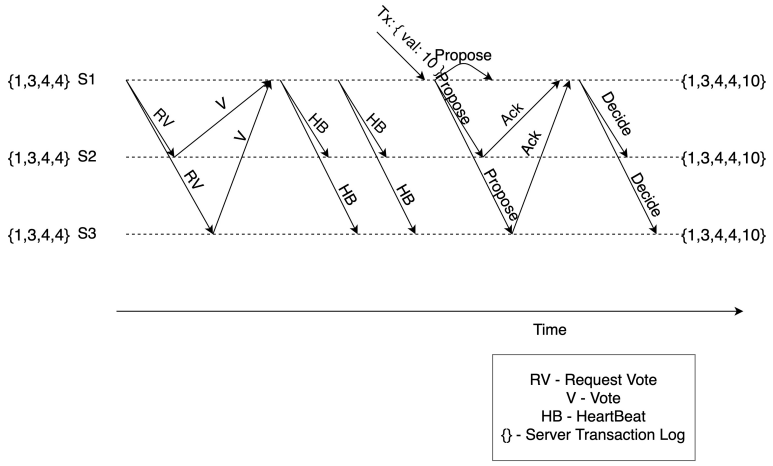


Figure 2.5: A normal epoch of RAFT. S1 requests for a vote, gets a majority and forwards Heartbeats to indicate it is alive. External client proposes a value and network decides on appending new value to the log.

In Hyperledger the ordering nodes (previously known as Validators) runs the RAFT protocol. The proposals are received by the clients which have previously run their transaction according to their respective rule-sets (See 2.7). The leader of the network will then run the validation of the rule-set to check the validity of the transaction T to see whether the $T_{signatures}$ fulfils the rule-set. If the rule-set is fulfilled, and all transaction proposals in $T_{proposal_{Peers}}$ are consistent in their execution, then the orderer will forward that transaction in to a so-called Ordered blockcutter method. Once the blockcutter is ready to form a block then the leader of the network will propose the block to the logs of the follower of validation nodes in the network. When done, the new block will be forwarded to each leading peer in the peer organizations which will append this to their ledger and then finally gossip the block to other peers within the peer organization [22]. Since the RAFT protocol is CFT, any anomalies will not necessarily be detected by the orderers. Nevertheless, each peer will go through the transactions in the block and validate it.

There are radical differences in between the consensus approach for private and public blockchains. As discussed, the public blockchains are giving probabilistic determination versus the private blockchains where the block commitment is done through a voting system, giving deterministic commitment of block sequences. Moreover, each of the models have different assumptions in terms of CFT (for Hyperledger) and BFT for public blockchains. Given that the Proof-of-Stake scheme is emerging in some permissionless blockchain systems (such as Ethereum) does indeed impose a faster consensus protocol and the protocol will still work given that only a fraction of the nodes misbehave. For

public blockchain systems there will always be a risk of Sybil attacks, meaning that there must exists an mechanism to give incentives on those nodes which tries to cheat the system. In contrast, private systems has white-listed the participants (i.e. known set of identities). These identities can be held responsible and held legally accountable by their actions. As a result, a common way of arguing of the system model in permissioned blockchains is that running a costly consensus process (such as Proof-Of-Work) isn't necessary as each network participant obey contractual obligations to behave correct. Consequently facing incentives from the outside world (in terms of fines, legal processes etc.) [23].

2.4.2 The Hyperledger Fabric Architecture

Hyperledger separates the network in to roles for all participants in the network to handle the consensus procedure [21].

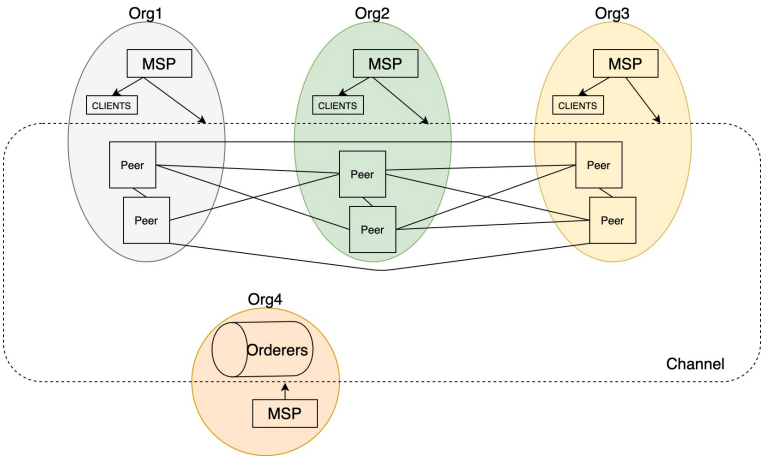


Figure 2.6: Simple overview of Architecture.

- *Peers.* Peers holds the chain code (smart contract) and maintains the ledger. Some peers might act as *Endorsers* which means they are a part of a *rule set*, these nodes verifies a transaction by assessing whether the transaction holds the necessary and sufficient conditions, like provisioning of required signatures around some database asset. A peer must also be connected to a NoSQL database system which will hold a snapshot of the state(s) stored in ledger. A peer can also be assigned the *Anchor Peer* role, an anchor must exist for each organization to enable gossiping to discover other peers that are part of another organization.

- *Orderers.* Controls the channel abstractions and access control attached to them. All endorsed transactions are batched amongst the orderers to form blocks and eventually push these to the peers. Peers can reach other peers through the Orderers. Without these nodes Peers across organizations (consortiums) cannot reach each other. The orderers can be seen as the controllers and viewers of the network.

- *Clients.* Authorized clients initializes transactions towards the Peers and ordering service(s).

- *Membership Service Provider (MSP).* Ensures that all nodes are tied to a certain identity. The MSP is a white-list approach to define allowed entity in the network. Each node must acquire an MSP. Though this Service doesn't exists as an entity it doesn't in the architecture it will be located at each node. If the MSP did not exist, the network would not function as this carries the trust in between different nodes.

Hyperledger Fabric is organization-centric. Meaning that all network participants must contain a certain sub-set which in this context is abstracted as an organization. For each organization there exists multiple deployed peers or orderers.

An *Endorsing* peer is a peer running a smart contract. Its task is to simulate the transaction proposal in order to validate the corresponding result of that transaction. Though, a peer which is not an endorser is also known as a *Committers*. They receive orderered state updates, in this case in form of blocks from the ordering service. Whenever a new block is received, the peer will validate the transaction and commits the changes on its local copy of the ledger. The ordereres runs the Consensus protocol to ensure that the transactions are strictly orderered to make sure that peers agree on the block and that the system is in deterministic state. In the context of Hyperledger the RAFT consensus protocol will work like the following. The client initiates a proposal for the network peers to simulate the transaction. Upon finishing the transaction the peers will return an endorsement to the client. Once the correct amount of endorsements is received (defined by a policy), then the client will forward the transaction with endorsements to the leader of the network, namely one of the orderers. They must decide whether the signed endorsement from the peers is valid or not and agree that there exists a *strict ordering* of the transactions such that any inconsistencies in the blocks does not occur. If the required signatures is valid, the orderer will create a block and forward it to the peers to append a new block to their ledgers. An architecture like this can be defined as a *Execute-Order-Validate model*.

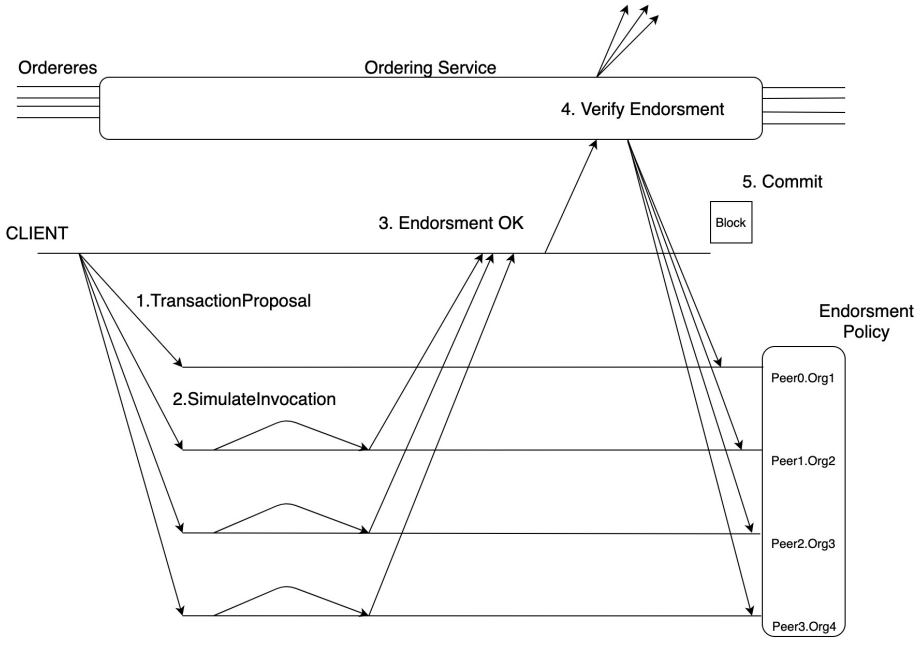


Figure 2.7: Transaction flow. Endorsing peers is required to execute the transaction and agree on producing the same value before the orderers can validate and produce a block.

2.5 Transaction types

In Hyperledger Fabric, there exist two types of transactions that can be sent towards the network, namely Invoke and Query. The invoke function executes a specified function in conjunction with parameters to pass. Usually, these parameters manipulate some asset which is stored in the ledger. The asset modified is usually controlled by a so call smart contract. A smart contract is essentially a piece of code that has functions which that does read and write operations on the peers' database. Once retrieved the contract can modify the asset and then commit again. A smart contract in Hyperledger Fabric is also known as chaincode. The invoke function may involve modification/query of the ledger, and the corresponding contract produces a success or failure response. Query, on the other hand, may only view the state and works similarly as the invoke function. Any attempt to modify the state here won't be committed as it will be executed only as a read operation on a specific peer. Back to the invoke function, there must exist a subset of peers that handles the transaction proposal that is required to endorse (simulate) the transaction on the relevant contract, which is set during setup/update. The client must have a protocol which allows for which peers to communicate with to execute a particular transaction and is responsible for collecting the endorsements too, before forwarding it to the ordering service such that the defined policy is fulfilled. Any transaction without a fulfilled endorsement would fail [24].

2.6 Endorsement Policy

The Endorsement Policy (previously known as rule-set) reflects the business logic of which organization that participates with the agreement on the current chaincode (i.e. the simulation of the transaction). These policies are defined as simple expressions that should evaluate to either true or false. An endorsement policy in this case would be defined as **"OR('Org1.member', 'Org2.member')"** where one or both of peers within the organization one and two must endorse the transaction before deciding the block. The client that invokes a certain transaction will go through the Endorsement System Chaincode that checks whether the client can invoke that transaction on the channel. After validation, and if successful, then the endorsing peer can execute the relevant chaincode to simulate the transaction. Once the client has got enough responses, it will forward all the endorsement to the ordering service. If the transaction(s) are equal (in terms of output result) and have valid signatures, then the block is forwarded back to the peers. As a final check, the final validation of the endorsement is performed. The checks happens in the Validation System Chaincode (VSCC), which is associated with a particular chaincode. The main job of the VSCC is to ensure that there are

valid signatures from valid certificates for the message - i.e., that the policy fulfills the requirements set by the system administrator. These policies are kept and maintained by the admin-role of the network [25]. The endorsement policy is also pluggable which means specific logic can be put here [26].

Double Spending and sybil attacks in Hyperledger are handled like the following. Sybil attacks cannot accrue without adding new nodes to the network. These have to be white-listed by the system administrator(s) and gets incentives (externally) for not behaving. One cannot simply add new nodes to the network - they have to be approved by an authorized person. Furthermore, an issue that arises with the transact-order-validate model is the following:

Scenario 2.2. Assume that an client issues n transaction proposals $P = [P_1, P_2, P_3]$ where all P_n applies changes to some chaincode asset A in the respective order. Then let's assume that an endorsing peer N experiences significant amount of requests resulting in a Transaction proposal endorsement time delay $T' >> T_{mean}$ such that in the validation phase at the peers that Proposal P_1 is delayed and the subsequent Proposals are executed and manipulates the ledger. Then we risk updating an object in the ledger to an inconsistent state. Given the scenario from above, there can exist a possible double spend attacks in the context of permissioned blockchains running a transact-order-execute model. However, Hypereledger fixes this through the concept of Multiversion Concurrency Control (MVCC) which is called **before the VSCC**. This is an mechanism which doesn't allow a peer to write a proposal if there exists a set of outstanding transactions with an earlier timestamp in the stored proposal-set (from the transaction phase) [27].

Definition 2.12. We can formulate the MVCC as the following. A transaction proposal T' cannot be applied as long as their exists a Transaction set $T = \{T_1, T_2, T_3, ..., T_n\}$ where Proposal Time Stamp (PTS) $PTS(T_i) \leq PTS(T')$.

The endorsement policy can be abstracted in to the following model:

Module 2.4 Endorsement Policy (Rule-set)

Module:

Name: *instance* Endorsement Policy extends Chaincode;

Events:

Invoke: ($\prec$ Transaction, $T \mid Txvalue \succ$): Execute transaction and produce result.

Sign: ($\prec$ Transaction, $T \mid SigningKey \succ$): Sign Transaction.

Detection: ($\prec$ Transactions, $T \mid []Transaction, *pemFile \succ$): Check the integrity and consistency of the Transaction Executions.

Properties:

Scoped. According to the policy a set of peers must execute the transaction for it to be considered to be valid.

Verifiable. Other nodes can verify the transaction result and that it was produced by the correct peer(s).

Strict execution. Any unauthorized peer which is not part of the policy will not be able to take part of the voting of transaction.

To summarize the module 2.4 - the endorsement policy is tied to the chaincode. A chaincode will then impose a requirement that a subset of peers *must* execute the transaction proposal issued by the client - which is the *Scoped* property defined above. Once the client has gotten enough Proposal Responses it can be sent to the ordering service. The orderers will then verify whether the policy is fulfilled - by inspecting whether there is an inconsistency in the transaction results and that the signatures are belonging to the set of nodes which the policy requires - utilizing the *Verifiable* property. Moreover, any unauthorized peer executing the transaction which in the proposals sent to the ordering service will be denied - thereby the *Strict Execution* property.

2.7 The Message Servicing Provider

As Hyperledger Fabric is dependent on user management, there exists a so-called Message Servicing Provider. Without this entity in the network, the assumptions of the Permissioned Model breaks. The MSP is the main component of Hyperledger that issue certificates, validation of certificates, and authentication of users. An MSP might define its notion of identities and rules that the identity validation/authentication should obey. However, the default implementation of issuing identities in HLF is through x509 certificates. For the MSP to work, it expects a set of predefined folders that reflect the organization internally. More specifically, the MSP defines the inbound definition as who

is the admins, users, clients, etc. And a validation part which allows for verification of that MSP's identities. Furthermore, the MSP also states which Certificate Authorities (CA) are accepted to establish a trusted domain through a listing of their members, or which CAs that can issue valid identities for adding new users.
With the help of these certificates, the MSP can translate a particular identity to a specific role that determines what rights a specific identity has within the system/context it resides. More specifically, all peers can be assigned the "peer" role, all admins can have the "admin" role, and all application users can be assigned the "client" role. Through these roles, Hyperledger allows for extensive and modular policy definitions that can set accordingly for the business application.
For two or more organizations to be able to authenticate each other through their respective MSPs, they most form a common communication channel. When the channel is configured and initialized, the two organizations are aware of each other's MSPs allowing for easy user management across the different businesses. This way, it's possible to define channel policies that build upon the MSPs that have joined the channel, giving a more fine-grained policy definition. As an example, there could be two organizations, one and two that have formed a channel. By default, both Admin from **Org1MSP** and **Org2MSP** will be administrators of the channel. However, let's assume Organization 2 is a subsidiary of Organization 1. Given the context, maybe the admin of Org1 is the "true" admin of the channel, such the channel policy reflects that only the administrator of Organization 1's MSP can perform admin-like tasks. Also, each component that resides within the network must have an MSP defined, and they all have to have their unique ID (especially if organizations is joining the channel) meaning that faking another MSP by having the same ID as one another wouldn't work unless the genesis block of the current blockchain located at the joined peers needs to be changed.

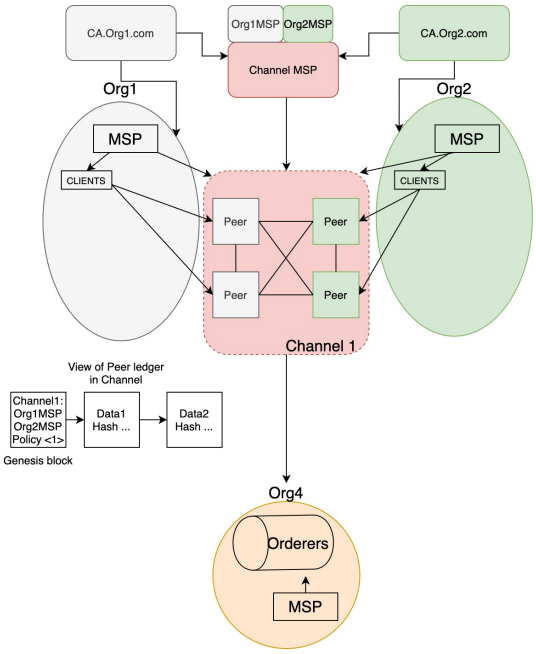


Figure 2.8: MSPs in channel 1 are "Merged" such that all nodes are aware of each others identities. Policies in the channel definition can then be built on top of these MSPs.

To summarize the capabilities of the MSP we can abstract its core functionality in to a interface module.

Module 2.5 MSP

Module:

Name: *instance* MSP;

Certificate Authority: MSP uses CA extends PKI interface; **Identity:** Uses the X.509 standard.

Events:

Verify: ($\prec$ Identity, $I \mid *pemFile \succ$): Verify Membership.

Revoke: ($\prec$ Identity, $I \mid *pemFile \succ$): Revoke Access of user.

GetRights: ($\prec$ Identity, $I \mid *v, *pemFile \succ$): Retrieve the role and affiliation of the user.

Add: ($\prec$ Identity, $I \mid *UserDefinition, *admimKey \succ$): Add a new user to the MSP.

Properties:

Unique. There cannot coexist two or more MSPs with the same ID in a channel/network.

Verifiable. All identities in the network can be verified at the CA.

Shared. All Organizations participating in the same channel will have a view of the roles located at the MSP.

Trusted. Only a subset of entities can generate and verify the content of the MSP

As a general rule, there should exist an equal amount of MSPs as there are organizations. The exclusive relationship between an identity and the MSP makes it logical to define the MSP as the Organizational name. Furthermore, we say that MSP is *Shared*. Through sharing doesn't mean that other MSPs have direct access to that MSP's secret keys etc. However, they can establish a relationship between the identity (pemFile) and the corresponding role such that policies on the channel level can be defined and enforced.

2.8 Channels

Hyperledger Fabric (HLF) seems somewhat to be a single blockchain network. However, in more extensive networks, some peers might want to execute private transactions that must stay confidential from the rest of the world state of other peers. HLF introduces the idea that these transactions can be stored on the blockchain, albeit as a separate ledger and a snapshot database. These nodes would request the ordering service to install a new channel for them. We can abstract a channel into the following module.

Module 2.6 Channel

Module:

Name: Channel, **instance** C, Uses RAFT instance;

Variables: Ledger, **instance** L

Events:

Request: ($\prec$ CREATE, $channelConfig$ | *peerOrganization[], *ordererOrganization[], *adminKey $\succ$): Order a new channel to the network.

Request: ($\prec$ JOIN, *channel* | *peerAddress $\succ$): Join peer to Channel.

Propose: ($\prec$ DECIDE, *value* | *v, *peerAddress[] $\succ$): Commit a value for the channel.

Update: ($\prec$ CONFIG, *channel* | *configBlock, *adminKey $\succ$): Update the current view of the channel.

Properties:

Isolated. Only permitted nodes in the network can process the transactions within the channel definition.

Immutable. Channel definitions are permanent.

Immutable Transaction History. A channel maintains an immutable transaction history.

Controlled Partition. All unauthorized identities are blocked from joining the channel.

Considering the abstraction above. The channel is maintained by the ordering service O which runs the RAFT consensus mechanism. Note that for each channel there will be one RAFT instance. The *Isolation* property ensures that only peers can view and

and append blocks to their channel ledger L. Channels $C = [C_1, C_2]$ at O will work in isolation such that $C_1 \cup C_2$ where transaction queue $C_1 = [T_1, T_2, T_2]$ and $C_2 = [T_3, T_4]$. The configuration of the channel is *immutable* in such case that it is stored as the genesis block of those peers participating the channel. We say it is *immutable* as a result of this. M However, there is possibility to issue a change to the current view of the network by changing the participating organizations that will join. Not altering and updating the channel view, there cannot exists arbitrarily organizations to a channel without issuing an update on behalf of the system administrator [28] [29]. Moreover, an ordering service may serve several channels and a peer organization may join multiple channels to act as an endorser or committer.

2.9 Private Data Collections

Channels does indeed partition the network in a such way that transactions and ledgers are isolated from each other. However, creating a channel for each set of organizations that requires so do create overhead in terms of administrative tasks. As a result, Hyperledger Fabric introduces the concept of Private Data Collections (PDC). A PDC allows for hiding sensitive information within a channel from entities which are not supposed to write/read it. As a result, the sensitive information itself are not disseminated in the transaction flow. More specifically, the peer invoking a private transaction plays an important role that involves in gossiping that private data to other authorized peers in the network as the blocks disseminated by the ordering nodes will only contain a SHA256-digest of the transaction. The PDC is associated to a specific contract and will be referenced to in the particular contract through a key-value approach. Note that one contract may have multiple PDC-definitions such that many organizations can use the same contract but call with a different collection definition. There are multiple benefits of having a Private Data Collection:

- Sensitive Data is not stored on the blockchain.
- Required Configuration is reduced.
- Though data is kept private, all other peer nodes in the channel will hold a digest of the original document.
- Multi-tenancy private data sharing.

The PDC are strong for organizations that wants to share a common asset but doesn't want to share all details with other organizations that also requires to know about the asset.

Scenario 2.3. Assume a channel C with three organizations $Orgs = [Org_1, Org_2, Org_3]$. Furthermore, Org_1 is selling important products P in some industry. Then, let's assume that Org_1 and Org_2 have done a deal where Org_2 wants N amount of product P at some lower price compared to what Org_3 gets. Though, Org_1 wants this to be a deal with only Org_2. Then, rather than storing that deal in a separate channel they can define a set of PDCs $PDC_{orgs} = [PDC_{privatedeal_{org_2}}, PDC_{privatedeal_{org_3}}]$. Where the constraints are defined as $PDC_{privatedeal_{org_2}} = Org_1 \cup Org_2$ and $PDC_{privatedeal_{org_3}} = Org_1 \cup Org_3$. As a result, both Org_2 and Org_3 gets their respective deals - yet without knowing the details of each other.

From the scenario above, the details about the respective deals are kept private. Though if there would be a dispute between some of the organizations about the integrity of the respective deals, then they can use each others peer nodes to validate the integrity by comparing digests stored on the ledger.

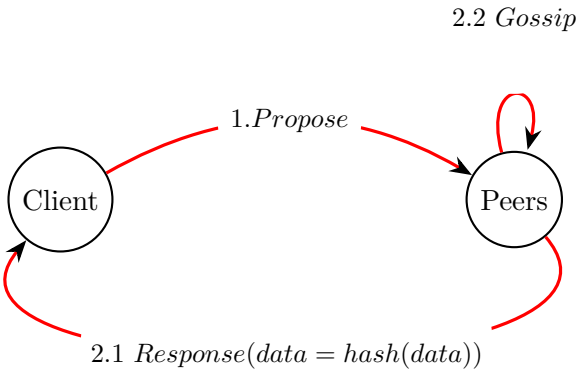


Figure 2.9: Proposal responses from Peers using PDC is hashed. Sensitive information are gossiped to other authorized peers

The sensitive information are stored in the peers' private data store i.e., its local Database. A PDC policy can also specify whether the information can be read/written by other members that may not be included in the policy. Furthermore, some sensitive information may only be retained for some period as a consequence of government regulations or policies like GDPR that requires the system to do so. PDC allows for defining for how long the data should be persisted in the private data store depending on the amount of blocks that have been added after the transaction digest was committed in to a block in the respective ledger.

Module 2.7 Private Data Collection

Module:

Name: Private Data Collection, **instance** PDC, Uses Channel instance;

Variables: PrivateDataStore, **instance** PDS;

Events:

PutPrivateData: ($\prec$ PUT, $TransientValue$ | Txvalue, PDCKey, Identity, PDS $\succ$): Put private data in a collection.

GetPrivateData: ($\prec$ GET, $PrivateValue$ | PDCKey, Identity, PDS $\succ$): Get private data from collection.

Purge: ($\prec$ NewBlockEvent, PDC_{data} | PDC[], $Block_{number}$ $\succ$): Purges the PDS for information which requires so.

Properties:

Private. Unauthorized entities attempting to access the PDC will be denied.

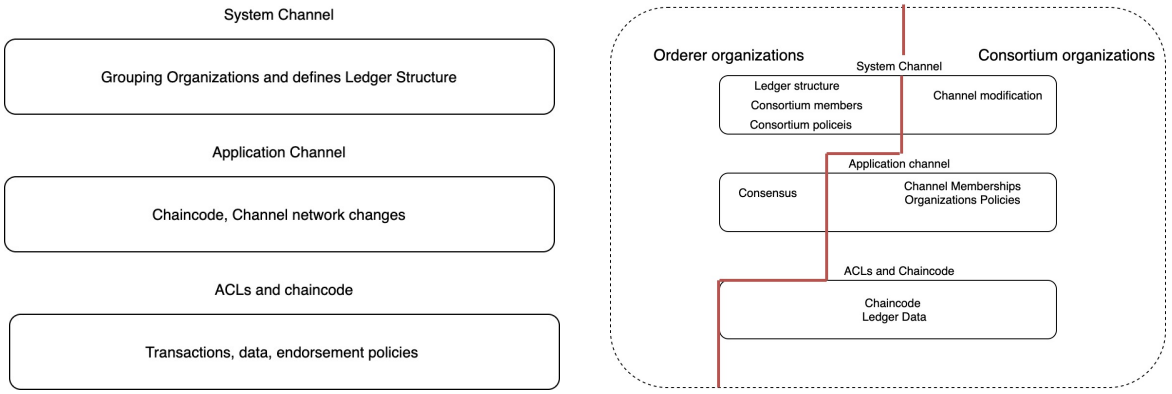
TimeScoped. Private details may be time scoped and eventually purged.

Verifiable. Ledger at non-participating peers may verify the integrity of the data.

The PDC lifecycle is per channel and is defined on per chaincode instance. Upon initializing of the chaincode the administrator of the network may define which organizations that should share the private data and also whether the private data should be purged after N blocks in the ledger L. Also, the definition of a PDC requires to specify an upper and lower bound of the amount of peers which should get the data for the first time the network sees a private value. This is for replicating the data in the case of a peer-crash. Peers that may not have the private data yet may lazely pull this upon endorsing some transaction [30].

2.10 Policies in general

The policies in Hyperledger can be abstracted in to a hierarchy. Different types of policies affects different parts of the application cycle including administrative tasks, access rights, transactional rights and so on.



(a) Overview (b) Hierarchical policy structure

Figure 2.10: Hierarchical structure of policies.

The *system channel* is the most top definition of an organized group in Hyperledger. The system channels define which ordering organizations that enforce the memberships of the channel and dissemination of blocks and which organizations (consortiums) that can transact within the channel. The ordering service runs the consensus protocol defined through configuration and decides how blocks are issued. Organizations that are part of the consortium can issue new channels and add new organizations to the channel itself, note that not all organizations need to be part of the consortium. Furthermore, the *Application channel* defines the possibility to add and remove members from the channel. The *ACLs and chaincode* gives the ability to define *Endorsement Policies* for chaincode. There is also a way of defining access control on specific functions that map to a certain policy defined on the application/channel level. Undefined policies in a given system level would either precede to a default policy or inherited from the level above. Managing different kinds of policies is done by different participants of the network (ref figure above). The organizations manage any channel modifications in the consortium (System Channel level). Subsequently, the management of memberships and organizational policies are also maintained by the organizations together with maintaining the ledger data and chaincodes. On the other hand, blockchain structure, consensus protocols, and consortium definitions remain maintained by the orderer organizations. The "partition" of which organizations that maintain the policies is natural in terms of the delegated responsibility. The orderers manage which organizations that are part of a channel (i.e., the blockchain structure), whereas everything else that is business agnostic

such as policy definitions and memberships are of concern for those organizations that require it.

2.11 Final words on differences

We've now established the differences between permissioned and permissionless systems. More specifically, the permissionless systems do, in general, assume that the peering nodes maintain an abstracted business process in terms of code which is open for anyone. In contrast, the permissioned system only a subset of nodes that is eligible to run and execute the process. The appending of blocks in the permissionless model is done by a hard cryptographic puzzle that is solved by nodes that act as validators. Any attempt to cheat the system results in costly financial penalties for the byzantine fault node. In this case, Hyperledger Fabric doesn't have any incentives for such behavior. Nevertheless, HLF expects that all members are predefined before performing tasks within the network. Consequently, the authorized list of identities of such systems allows for defining incentives like laws, policies, and financial penalties as everything is traceable and tied to a specific entity in the real world. As a result, the consensus protocol used is a leader-follower model that doesn't rely on substantial computing power such as PoW to produce blocks. As discussed, the downside of the PoW-scheme is that it doesn't produce a final deterministic view of the chain. More like probabilistic guarantees. Consequently always having risks of potentially getting a skew hashing-power distribution in the network that can lead to two or more versions of the original chain if a miner chose to go byzantine. Contrary to the permissionless model, the permissioned system collects a majority of signatures and disallows split votes meaning that only one view of the chain can exist (given that all nodes behaves).

Another benefit of the permissioned model is that the expected performance in terms of transactions that can it can handle per second is greater than what permissionless systems would produce. However, the system architecture (especially Hyperledger) does get more advanced and would require more configuration to run properly. Furthermore, the key distribution and defining the white-lists (MSP) in HLF is harder than the permissionless systems as anyone with a public-key value pair (in the right format) can join. The decoupling of centralized distribution means that the business logic itself can define a allowed list of members with right key pairs that are eligible to execute the business logic. Despite the key distribution, as everything is publicly available on the ledger, storing sensitive data is more laborious.

Chapter 3

Working with the Hyperledger Environment

3.1 Configuration

This chapter gives an introduction to the different aspects of configuring a Fabric network. However, this chapter is not necessary to understand the rest of the paper. However, the chapter aims to provide the necessary information required to understand and setup a testing network to explore with the capabilities Hyperledger provides. We also give suggestions based on experiences while configuring the framework. The reason for including the configuration section is because the framework is heavily dependent on the configuration to work according to requirements of the project. Specifications include everything from access control lists, distribution of cryptographic material, channels, consensus protocols to run, installing and instantiating chaincodes.

Summary of working with Fabric	
Aspects	**Comment**
Architecture	Docker images need to be configured with scripting files.
Deploying to Production	Hard to maintain as all nodes require configuration which is specific for each organization.
Updating the network	Manual process, needs scripting files and admin privileges to change configurations.
Consensus	Pluggable, ability to chose between Solo, KAFKA, or RAFT.
Performance	Adjusted in the configtx.yaml file, more specifically - block parameters.
Chaincode (smart contract)	ChaincodeStub API provided for more popular developing platforms such as Java, GO and NodeJs.
Business requirements	Modeled in channel configurations and policies such as endorsements, private data collection and channel policies.
Application Development	Additional SDK requried for external communication must be added. Available for platforms such as Java, GO, NodeJS, and Python.

Table 3.1: Brief description of how to get started.

3.1.1 Testing and development environment

To test and develop the algorithms, we needed a proper testing environment to execute and deploy code while observing the output results. One could set up the required architecture manually by installing Hyperledger binaries to different hosts and connect them. However, this is cumbersome and time-consuming. The solution was Docker. Docker is a technology that works like a virtual machine (VM). Unlike traditional VMs, Docker aims to share the same kernel on the host machine rather than having one kernel for each Operating system; this provides less overhead and better utilization of the host machine's resources. Hyperledger has docker images for all the required components in the architecture that can be pulled and used both for testing and distribution (distribute docker images).

Docker

The Docker engine is essentially an infrastructure which plumbs a set of binaries into images abstracts them to *containers* when run. The Docker engine consists of many small specialized parts which enables the engine to run almost all installable/downloadable software within the containers. Everything from Network Interface, Storage, and more. Due to its low overhead and ease of use, it makes the process of installing software easy [31, Chapter 2].

Docker images – can be seen as classes for those who are used to the OOP-paradigm. Images can be stacked upon each other and can be represented as a single object (container). This way, all images(classes) are "loosely coupled", which means that the "scalability" of images is significant, making it possible to create containers with a lot of features from different branches of software on top of each other.

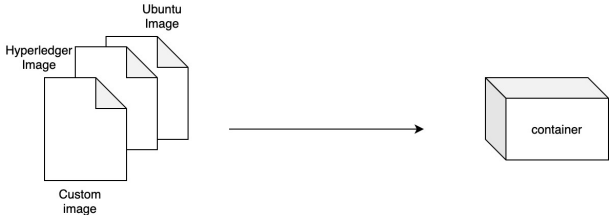


Figure 3.1: A docker image becomes a container.

How is the images built? This can be done by providing a "Dockerfile" in the desired project folder. These files does not have any file extension. It holds a set of instructions used by docker to form a new image. This could be everything from installing a certain program or pulling other images to build on top of. This way its possible to reuse existing images to build a custom image. A way of utilizing this is that a software provider can pull the original Hyperledger image(s), provide the custom software and configuration and then distribute it to external organizations as an image to be built as a container that eventually can exist as a pod in Kubernetes. Hyperledger is dependent on running on a predefined private network of nodes. *Docker-compose* is a way of creating a local network on the host machine to facilitate multi-host applications using containers.

A yaml file (refer to listing A.1 for details) consists of a key-value specification. For docker-compose one would requiretThe *networks* key which allows for defining one or more networks with properties such as IP-addresses, network drivers and more. Secondly, docker-compose must have *Services* which refers to the containers/hosts that will be part of the network. A service could point to an existing image in the Docker repository or to a Dockerfile which that container should build and run. In terms of Hyperledger, configuration of environment variables is required under the *environment* section, refer to Hyperledger's github code base for more configuration parameters [32].

Since Docker runs on a single host machine, it is wrong to assume that the performance is equivalent to a real world multi-host environment. This is due to the fact that the containers we are running is actually using the same RAM, CPU power and disk. Moreover, most Operating Systems work with context switching which also creates overhead, meaning that the containers isn't necessarily being run in parallel. Therefore, we only used Docker as a testing environment for developing and testing of the chaincode.

3.2 Deploy Hyperledger to Production Using Docker Images

A possible way to deploy hyperledger into production is to use the containerized applications within Kubernetes. In essence, Kubernetes is a orchestration tool that manages containerized applications across several cluster machines. The idea here is to allow for up-scaling and down-scaling of applications without having to worry about configuring addresses, ports, and more for each time a container dies. Kubernetes therefore provides predictability, scalability and high availability. The *master* server acts as a main gateway for the cluster creation and generation of resources. The master will issue commands through its API to nodes that isolates different applications in to docker containers known as a pods. A pod represents multiple instances or a single instance of an application. As pods are created and deleted continuously(not persisted) they end up with different addresses each time. Consequently it would be hard to manage during run-time. The Kubernetes services will then act as a selector that routes requests to containers that are running. The Services is essentially a group abstraction of pods and routes requests to pods by discovering them through their service name. However, as Kubernetes runs in isolation there cannot exists external requests to communicate with the internal cluster. A common way to handle this problem is to put a load-balancer or an ingress in front of the cluster. Essentially, a load balancer will be exposed to the public network, through NAT one can map an external port to an internal port related to a certain service [33]. It is also common to use an ingress, which essentially can be seen as another type of *Load-balancer*. What distinguishes the ingress from the traditional LB is that it creates a rule set that maps rules towards backend services. A way of doing this is to create a rule-set based on hostname of the internal service, such that the user(s) can write hostnames to access the services they require (as in this case that could be the peer.<org>.no, orderer.<org>.no, etc.)

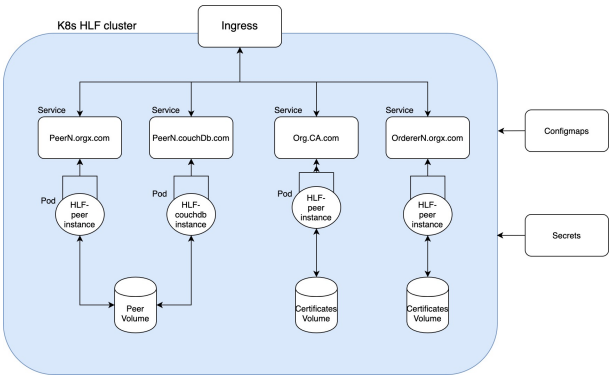


Figure 3.2: Simple Kubernetes Cluster for hyperledger fabric.

As depicted in the figure above, volumes are persisting data as containers are mortal, i.e. by mortal we say that the containers lifecycle is essentially is getting started and

killed without leaving anything behind. Kubernetes solves this by a so called persistent volume. Volumes can in the context of Hyperledger persist ledger data, certificate data for users, and will also be helpful during the update strategy of the cluster. Furthermore, as discussed, Hyperledger requires to a certain extent a lot of configuration. In this case, to persist configuration files can be put into a Kubernetes configmap resource. A configmap is basically an entity located in the cluster that holds configuration data that the pod will consume during provisioning of the container through a volume [34]. With this, the cluster can grow with minimal configuration changes if the organization would require to get more resources in terms of peers and ordering services. Furthermore, keeping sensitive data such as certificates, passwords for the CA can be stored in secrets [35].

3.2.1 Helm Charts for provisioning Kubernetes Resources

Even though Kubernetes is made for scaling applications with ease - it will still require coding of Kubernetes resources - and will probably be duplicate configuration files as many peers require the same.

- Secrets. One for each peer/orderer/org.
- Services. One for each peer/orderer/org.
- Configmaps. One for each peer/orderer/org.
- Pods. One for each network participant.
- Volumes, depending on architectural design decisions.

In the long run this wouldn't scale for operators of the blockchain network. There needs to be done automation of these configurations in order to save costs, time and human errors. This is where the concept Helm Charts comes to play. Helm Charts allows for defining scripts in YAML files known as templates. The templates are configured and reflected through what we call the *metadata files*, these files include everything from required host names, crypto material and so on. Helm charts reads this properties and translates them accordingly into their respective Kubernetes resource definitions. This way an operator may only write meta data in a subset of files and run a few helm commands in order install/upgrade the desired resources for his/her network. As a result deploying and maintaining the network is much easier.

3.3 Configuring the Framework

The images built and distributed by Hyperledger contains the required software to utilize the framework. These core components of the network includes:

- *Certificate Authority.* The Authority of the network vouching and validating X.509 Certificates.
- *Peers.* The Nodes of the network which maintains the ledgers and maintains the lifecycle of the Chaincodes (Smart Contracts).
- *KeyValue Data store.* A Key Value Database which maintains a snapshot version of the Ledger located at the peer.
- *Orderers.* Runs the consensus protocol to be configured.

However, these images are not pre-configured and provides the software-components for each participant in the network. The configuration must be done per node. A recommendation here is that the network administrator should have an idea of how the project scope is defined in order to create a proper topology for Hyperledger to run on according to the application. The framework is heavily dependent on policies and the configuration of the network to work properly. There are multiple steps such as defining the scope of Messaging Service Provider (MSP), configuring the Hyperledger Fabric Key material (Public Key Infrasctruture), channels, endorsement policies and so on. Due to the amount of steps required it was discovered that this process were prone to human errors in terms of configuration mistakes.

3.3.1 Configuring Initial Certificates with cryptogen

Before configuring and running the Hyperledger Fabric network one will need to create certificates and signing keys that is used for the MSP of all peer and ordering organizations. Two peers cannot initate communication without these keys as they wouldn't recognize their respective MSPs. *Cryptogen* is a tool provided by Hyperledger to make the process of generating Hyperledger fabric key material easier. The tool looks for the *crypto-config.yaml* (See listing A.2 for details) file in the same directory the command is called from. This file will contain all the entities in the network, such as those organizations that manages orderer and peer nodes. For each organization, one can also specify the amount of users in each organization.

```
cryptogen generate --config=./crypto-config.yaml
```

Listing 3.1: Command for generating all required keys for the network.

The cryptogen [1] tool will read the YAML file and generate a corresponding *crypto-config* folder in the current directory. The created folder will contain information such as cryptographic keys. For organizations, there will be a corresponding folder with the necessary material, which will be added to the corresponding MSP of different nodes during the bootstrapping of the network. Properties for the organization's Certificate Authority can also be put here. Such as Subject Alternative Name (SAN), which can hold parameters such as IP addresses, DNS names, and more.

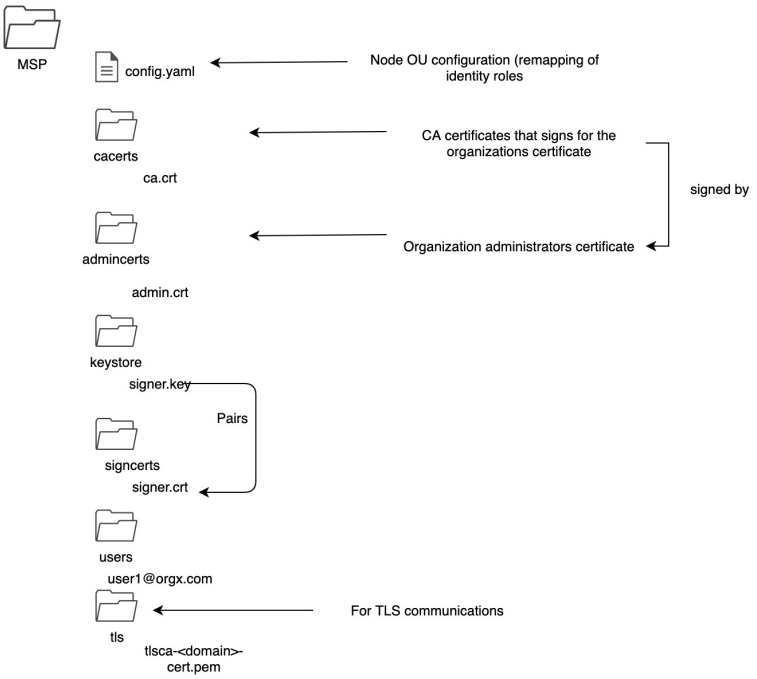


Figure 3.3: Generated MSP structure.

The generated structure of the MSP must then be mapped accordingly to the different containers in docker such that the endorsement policies and more should work as expected. Failing in doing so may result in multiple errors when starting the network.

3.3.2 Generating Channel Artifacts

In order to create the topology of the network one would need the following:

- *The Genesis block.* Contains specific information of which certificates (from which organization) that can serve the initial fabric blockchain for the channel. There will be one genesis block for each channel created.
- *Channel config.*

[1] Cryptogen should only be used for testing purposes and not for production environments.

- *Anchor peers.* Anchor peers need to be configured for each organization. These nodes acts as a supporting node within the organization. It syncs with other organizations ledger's using the builtin gossip protocol.

All these are configured in the file *configtx.yaml* which is read and used by the *configtxgen* tool. A channel can be defined as shown in listing A.3. Essentially, a channel is defined as a profile. Within the profile there exists a consortiun. The consortium is a collection of non-ordering organizations of the network which the orderer is going to serve and allow for administrative tasks within that channel (like adding new members etc). These organizations will be the initial organzations that form and join the channel. Furthermore, a channel may support different versions of running nodes, known as Capabilities. This needs to be specified in the configtx file to ensure that client applications and channel nodes can run on different binary versions which may produce inconsistent results as the framework changes over time. Defining capabilities allows for cross-version compatibility which allows for different nodes running different versions to communicate with each other regardless.[2]. Furthermore, other than the configuration block of the channel definition the ordering service also need to know about the Anchor peers for each organization. Each organization has set their respective MSP information as well as crypto material including certificates, keys, hostnames, port information in regards to anchor peers as shown in listing A.4.

Here the organization refers to the MSP directory which is where the cryptographic material (represented in a folder) generated with *cryptogen* is stored. The ID parameter of the MSP must be unique for each channel as the channel abstraction would shared all participants local MSPDir. Furthermore, anchor peers are defined here as well. A good practice would be to have redundant anchor peers for high availability, since these nodes are critical in synchronizing with other organizations when it comes to configuration changes in the channel - like newly added organizations or other changes to the genesis block. An organization without anchor peers will not cause any crashes. But in a case of a partition where a peer organization loses connection with the orderering service, then other peers in the same channel will receive updates from neighbouring peer organizations. The same applies if a project uses private data collections extensively. If a node doesn't have the latest private data - then the anchor peer(s) will need to get this data from another organization through the gossip protocol.
To generate the required genesis block, one can use the following command in the same directory where the configtx.yaml file is located.

```
// create the genesis block for channel definition.
 configtxgen -profile UniversityChannel -outputBlock ./channel-artifacts/genesisunichannel.block
 -channelID unichannel
```

[2]https://hyperledger-fabric.readthedocs.io/en/release-1.4/capabilities_concept.html

```
// create anchor peer config
configtxgen -profile UniversityChannel -outputAnchorPeersUpdate
-/channel-artifacts/UniAnchors.tx -channelID unichannel -asOrg <OrgID here>
```

Listing 3.2: Creating a genesis block.

The *UniversityChannel* keyword corresponds to the same profile defined in the Channel profile listing. Furthermore, to generate the anchor peer configuration used by the network one will need to execute the anchor peer command for each organization while changing the name for the *asOrg* parameter.
It can also be defined Access control List for each channel. In Hyperledger these are defined as policies and will be defined on the entity type. An entity type can be defined as a specific peer, member, client or admin. There exists two types of Policies [36]:

- **Signature Policy.** Evaluates the access control based upon the MSP principals. It supports combinations such as *AND, OR and NOutOf* which makes it possible to create rules like : "Admin of UiS and UiO or 8 of 15 organizational admins approve this".
- **ImplicitMeta policy.** Allows for referring to other policies (utlimately signature policies) to be evaluated up against. E.g. "Majority of admins in the channel". However, these policies seems to be less flexible in comparison to evaluating the signatures.

If a channel has two Organizations joined in, then the "Any Writers" policy would fallback to the Organizational policy definition. This means that users from both organizations have the same rights as within their own organization unit. However, this can be overriden by specifying the new policies under the channel profile. A policy would can be defined for applications (clients), channels and organizations, same structure applies for all policy definitions as depicted in listing A.5.

An important note while developing with Hyperledger Fabric is that identities defined in the Rule section can be users, whereas in Endorsement policies these identities must be Peers in the network to sign and validate transactions. **Readers** in this case will be allowed to read the state of the ledger. **Writers** will be allowed to propose new values that are to be appended to the ledger. **Admins** are allowed for more administrative operations such as updating channels, initiate chaincodes etc. Though, the format given above is the general structure for defining System, channel and transaction policies in Hyperledger.

The orderer also has a set of parameters that can be defined before bootstrapping the network. These parameters include which consensus protocol to run, adresses of other

orderers, and list of organizations which are defined as participants on the orderer side of the network. Moreover, adjusting block frequency is possible. The block frequency can be set by variables such as:

- **Number of messages.** This is the upper limit of number of messages to include in a block.
- **Absolute Max Bytes.** Maximum size of messages to include in a block.
- **Preferred bytes.** A preferred value for message sizes from a message. A message size larger than this value will result in a larger batch size to form a block of.
- **Batch timeout.** How long to wait for cutting the block after receving a transaction.

These parameters need to be configured according to the application the ordering service will run in. More specifically, they tell when the ordering service when to stop, create a block and send it before creating a new one (refer to the configuration of these in listing A.6).
Furthermore, the configuration of the consensus is done under the orderer profile. As discussed, there are multiple ways of configuring which consensus type that can be used (SOLO, KAFKA or CFT RAFT) though KAFKA and SOLO are deprecated as of version 2.0 in Fabric. There has been research on providing a Practical Byzantine Fault Tolerant protocols for the project, though this is still not included in their official distribution files yet. Hyperledger is flexible in the terms of which consensus protocols it can run. There are projects which have defined their own software module which can be used together with HLF. This modular concept allows organizations to chose which of these consensus-providers to use and works best for their business need. E.g. organizations that already have expertise on KAFKA or Requirements of high performance CFT (RAFT), or lastly requirement of Byzantine fault tolerant ordering service. We utilize RAFT in our project which can be configured specifically for the project. The *TickInterval* is the time in between two HeartBeatTicks sent by the RAFT-leader. Moreover, the *ElectionTick* property tells followers in a current term that, if a node has not gotten 10 tick messages from the leader - it will set itself to candidate. The *HeartbeatTick* means that a leader maintains its leadership by every HeartbeatTicks. The *MaxInflightBlocks* is the number of appendBlocks messages the ordering node will allow for during an optimistic replication phase. Essentially, an optimistic replication phase allows for how much two or more nodes may diverge in their current replication log. This allows for a *lazy commit* of blocks. Lastly, the *SnapshotIntervalSize* tells RAFT when to take a snapshot of the log and save it to its persistent storage [37]. Though, changing these parameters such as attempting to set the ElectionTick and the TickInterval to low values

(for performance gains) may lead to a state of the system where a leader cannot be elected. We therefore propose that the system administrator should thoroughly test this before moving to production.

3.3.3 Defining channels

Orderers maintain the channel configuration and holds the information of which organizations that participates in the consortium. Note that the consortia of the channel defines which organization that are joining the network initially but are able to add other organizations to the channel at a later point. The purpose of the Consortia is that the organizations that joins initially are able to govern the channel with policies and more. This is powerful for those scenarios where there is only a subset of organizations should have the ability to define application policies and more. Though, not including organizations into the consortia does indeed pose constraints such as; not allowing for channel creation, not allowed to add new organizations, and MSP structure needs to be included. More specifically, an organization that is not part of the consortia must include their entire MSP structure to the channel configuration. As a result, organizations that joins afterwards needs to consult the system administrator of the consortium which then have to include the new channel artifacts (MSP) and issue a new reconfiguration block to the orderers and the rest of the network. This is where the *configtxlator* is used to create a new block - by using the previous one. The general idea here is that some of the organizations in the Consortia needs to fetch the latest config block from the ledger and then use the configtxlator to convert it to a JSON-format. With the help of tools like jq one can specify where in the produced JSON the new configuration data should be included and then create a new block-file again.

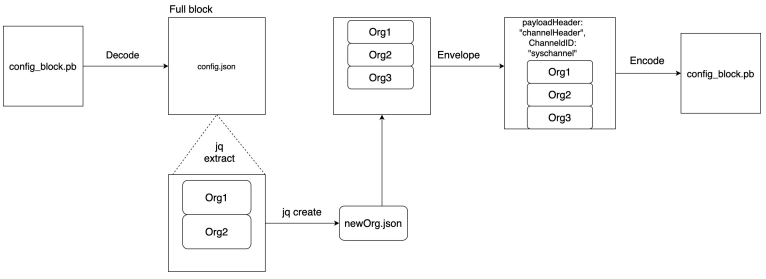


Figure 3.4: Flow of updating channel configuration.

The process is depicted in the figure above. Once the new block file is ready for submit, one of the authorized peers needs to issue that block the ordering services in order to take effect. If sucessful, then the new organization may include their peers.

3.3.4 Starting The Network

When crypto-material and required channel artifacts are generated the project should be able to be started. The general steps to get the network started is to utilize the *fabric tools* docker image to configure the cluster. Essentially, this image the commands required for managing the network. It can hold certificate data such that admin-like tasks can be done in the cluster.

- *Create the Genesis block.* All channel definitions will be contained here.
- *Create Anchor Peers.* Defines which peers that acts as anchor peers.
- *Create channel.* Create channel and instantiate it (once per channel definition). Update anchor peer configuration.
- *For All Peers.* Join channel(s), install and instantiate(once per channel) chaincode.

The network is then up and running. The examples of Fabric configures docker images using bash-scripts which configures the cluster through the fabric-tools images. Note that some configurations also require to set specific environment variables for both peers, orderers, and the CA. For example extended timeout values and CouchDB addresses. The fabric tool image must have set the right MSP configuration parameters in order for it to apply changes to any node in the fabric-network, as shown in [38].

```
peer channel join -b <channel_name>.block
peer chaincode install -l java -p your/path -n <name>
peer chaincode instantiate -n <chaincode_name> -C <channel_name> -P 'AND(A,B,C)' --collections config '
```

Listing 3.3: Fabric Tools commands, executed with 'docker exec <ENV> ...

The commands in listing 3.3 is run from the fabric-tools container towards the peer/orderer defined in the environment variables within the container. In this case, a peer joins a channel by fetching the genesis block from the orderer. Then it will install some Java chaincode (installation needs to be done on all endorsing peers), and then instantiate with an Endorsement Policy and a Private Data Collections. This is how fabric proposes to configure their containers.

3.3.5 Concluding the Configuration of Fabric

Fabric requires configurations on all parts of the application. From the point where chaincode is ready for deployment one need to configure an endorsement policy (optionally a PDC). Moreover, once the chaincode is ready one will have to find out which channel

that particular chaincode may run in. For a channel to work one must have an ordering organization that will hold the configuration of the consortia and corresponding policies for channels, applications and organizations. When a channel is defined then the channel-artifact need to be issued to the relevant ordering nodes. Then, rather than automatically allowing peers for the organizations which participates in a channel to join - e.g. through a discovery protocol (with the help of signatures to prove ownership) they will have to manually join themselves. Fabric shows that doing so requires a bash-script that uses the fabric-tool docker image for peers to join a new channel definition. Furthermore, updating to a newer version of fabric requires that the content of the ledger(s) etc. are copied manually and persisted on some other disk. This is more manual work - and will indeed be a candidate for adding a new script for such behavior. Moreover, updating the channel requires to fetch the current block, decode the block-definition to a JSON format using a new tool - namely, the configtxlator [29]. Then apply the changes to the JSON-file and finally encode it back the .block format. Following the examples of Fabric - this is also a new bash-file. All-in-all a lot of YAML files and bash-scripts is required for running a Fabric network locally.

3.4 Programming in Hyperledger Fabric

Configurations as previously discussed are a declarative way of working with the environment. The commands only enable functions/logic for the framework. These parts of the systems are typically used by operators/service people. In this section we look at the possibilities of creating a programmable logic - which is more aimed for developers.

3.4.1 The Chaincode API

Chaincode is an API available for Java, NodeJS, and Go. Chaincode allows for putting transactions on the ledger in any format. The API itself has the the following definition which all languages follows.

Module 3.1 Chaincode

Module:

Name: Chaincode, **instance** *cc* uses Private Data Collections 2.7, Endorsement Policy 2.4, MSP 2.5, X.509Interface

Events:

Init: ($\prec$ INIT | values[] parameters $\succ$): Initialized once the chaincode is instantiate and after upgrades.

Invoke: ($\prec$ INVOKE | value v $\succ$): Invoke a function to commit a value v.

PutData: ($\prec$ COMMIT | value v $\succ$): Commit value v to the state database.

GetData: ($\prec$ READ | key K $\succ$): Read a value associated to K from the state database.

Properties:

Strict Execution. All endorsing nodes that is required to execute must produce the same results.

Privacy. Upon using PDC, all information is only propagated to a subset of authorized peers. All other data is protected by the channel policy.

Locked. A state database key cannot be changed within the same block.

Identifiable. All callers are identified through their X.509 certificates.

To invoke and commit a value, one will have to call the Invoke function. A typical pattern here is a *switch(case..)* that passes the arguments in the call to a corresponding method. The chaincode only contains a set of APIs to update the underlying state-database, namely the CouchDB instance (no direct commit to the ledger itself). Once a chaincode execution finishes, the peer will create a read-write-set. A read-set is essentially the old value of the updated key, and the write-set is where the actual updated value(s) are. The read-write set will be used by the other peers in conjunction with the digital

signature of the endorsing (or executing peers) node(s) to verify the transaction result. If any disagreement exists, then the validation phase would fail, and the transaction will be discarded in the ledger, thereby the *Strict Execution* property. The updating of the ledgers only happens when the peers see a consistent read-write set. Deterministic code is important for chaincode to work. Though, all these processes are abstracted away from the chaincode developer. The primary concern of the developer is to model, develop, and test the code. To test the code without a running cluster, a developer will have to rely on mocking the chaincode API to replicate the Fabric's behavior. Nevertheless, to identify users, one can use the *ClientIdentity* [39] library, which allows for identifying different users from each MSP (not supported in the testing framework for v.1.4.5). This allows for fine-graining of Access Control down to the data-base key level. Furthermore, when using private data collections in chaincode one will have to refer the collections name through the *PutPrivateData(PDCKey, Key, data)* method. The data-field in PutData/PutPrivateData is typically in JSON-format. PDCs presevere the *Privacy* property as previosuly described in Chapter 2. Chaincode also has a capability of calling other chaincodes located both on the same channel and different channels (given that the channel policy allows the chaincode-caller to access). Essentially this capability is an *invoke* call from the chaincode. Note that a peer may have N chaincodes. From Fabric v2.0 there exists a new lifecycle of how the chaincode is installed. More specifically, organizations may vote in order to *agree on the chaincode defition*. This allows organziations to view the definition, sign it, and then vote. Upon majority, the code will be ready to be committed and used on the channel [40].

3.4.2 External Communication With Hyperledger Fabric

Fabric provides several SDKs for external communication through gRPC calls, some of them include the *Hyperledger Java SDK* [41]. The SDK is pulled as a dependency to a Gradle/Maven project. It is a low-level API that allows for controlling the network externally, some of these operations include installing chaincode, instantiating chaincode, and invoking & querying chaincode. Furthermore, the SDK allows for communication with the fabric CA, meaning that users can be registered with the client - as long as the admin/other authorized entity approves it.

Moreover, they also allow for subscription to block events on the channel. Using the Block Events makes it possible to even write tests on top of the SDK. As chaincode lacks testing possibilities during the Software Development Life Cycle (SDLC) one can use tools like Maven to write tests that issue transactions towards a deployed chaincode using Docker and assert whether the transaction was committed or not. Maven is essentially a way of building, managing, and testing Java code.

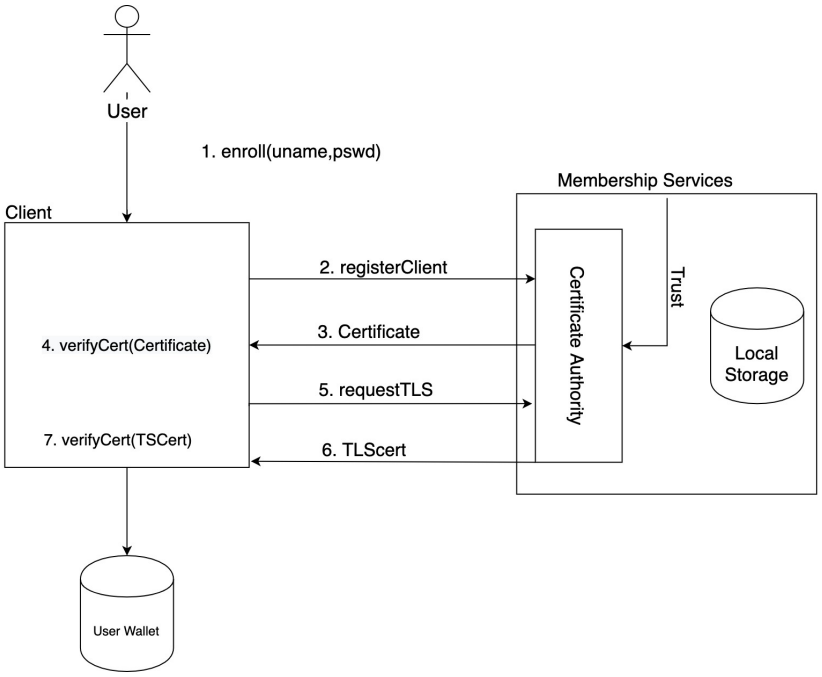


Figure 3.5: Fabric SDK allows for defining user-wallets. TLS is optional (but recommended).

Nevertheless, for the SDK to work, one must statically configure the network architecture through connection profiles. Essentially it is yet another YAML-file that needs to be defined for the SDK to read s.t. It knows the addresses of the ordering-, CA-, and peer-addresses. Moreover, a REST-API like approach can be created for external applications to communicate with the cluster so they can talk to the cluster through a rest-service. Alternatively, a strategy (for decentralization purposes) is to create a java client, or a JavaScript client (using the NodeJS API) for clients to issue transactions towards the cluster. The SDK has many possibilities and works like an external toolbox that connects the external world to a Fabric network.

Chapter 4

Related Works

This chapter covers the material studied in order to get ideas for the design approach, more information about the hyperledger framework, such as improvements, guidelines on how to develop secure chaincode. The basic idea is to establish an relationship between Hyperledger and the Bbchain project and create a similar solution approach as Ethereum.

4.1 The BBChain Project - Verifying Academic records

The Bbchain project at the University Of Stavanger uses Ethereum to develop a platform for academic institutions to verify academic records [42]. In essence, Ethereum is a permissionless system that allows for running *Smart Contracts* that runs on a public ledger. The Contracts provides for a limited set of executable commands to produce a final result. Depending on the resource consumption of the set of commands will consume something called *Gas*. The gas in Ethereum is a way of constraining the invoker to execute arbitrarily amount commands towards the system. Depending on the market situation, the client obtains gas by paying for *Ether* (which is the currency of the system) in exchange of value of a real-world currency (e.g. Dollars).
Consequently, if a user in the system decides to transact expensive transactions, then the gas usage will increase accordingly. Whenever the Etherum Virtual Machine detects that the transaction gas limit attains, and if the transaction execution-set still got commands to execute, then the EVM will undo all operations if the funds are not sufficient. Furthermore, similar (yet more expensive) incentives are given to the system's miners as a result of the PoW-scheme. As the system is a child of the permissionless model they deploy a P2P architecture where the role distribution for the backend service includes [43]:

- *Light Nodes.* Requests the current state of the ledger and validates block headers. The required resources in terms of CPU power is minimal from the peer.
- *Fast nodes.* Retrieves all the block headers but does not validate any previous transactions until the current one.
- *Full Nodes.* Maintains all history, and all intermediate states are computed on the fly. Unneeded transitions are pruned from the system.
- *History Nodes.* Sames as full nodes, but stores all state transitions.

The nodes in the network differ in terms of how they reflect the ledger history. Compared to hyperledger, these are committers (passive nodes) and endorsers (acting nodes). The acting nodes check whether the state-change is valid, whereas committers only listen for blocks and trusts that the endorsers did the right thing. Additionally, Hyperledger deploys nodes with a separate set of tasks to perform to form a consensus within the network (orderers and peers). As a result of the different roles, Hyperledger's transaction flow is more complex (in terms of steps) compared to Ethereum where a transaction engages the Proof-Of-Work algorithm at the moment where the network sees the transaction.

4.1.1 Attacks in Ethereum

The longest chainrule described (LCR) in Chapter 2 applies to Ethereum, however, they use a protocol known as Greediest Heaviest Subtree (GHOST). GHOST is a way of optimizing the most extended chain rule to increase the scalability and performance of the blockchain protocol. The nodes will continuously extend the chain, which has the highest amount of leaves. One can optimize throughput as one doesn't care about the length of a single chain, but the chain with the most leaves. The rationale behind this is that one can increase the block frequency and increasing the likelihood of disagreement (from the LCR view) though, as the new rule applies, these "disagreements" will not be significant. Further, as previously discussed in Chapter 2, the hashing power required to create two versions of the chains in Ethereum is working at under 30 % of the total mining power, given that the selfish miner holds the majority of the network capacity. As described in [44], the selfish mining attack is successful when the selfish miner has a significant influence on the network capacity γ. With γ we mean that the broadcasted blocks produced by the selfish miner are disseminated and received by honest miners before they receive any of honest miners blocks'. Nevertheless, they concluded that this could be leveraged to some extent if they consider both frequencies of blocks on the main chain and uncle blocks yielding better security than other systems such as bitcoin. Additionally, as the likelihood of successfully performing a selfish mining attack

increases with the network capacity γ of the selfish miner, one can perform this attack by silencing the network's nodes. As described in [45] by Xu *et al.* the Ethereum node's 13 connections can be controlled by an adversary which lures the target node, ultimately the nodes' view of the world state can be manipulated resulting in a disagreement of the network. Alternatively, the adversary can only choose to forward blocks by some mining pool. Consequently making the selfish mining attack easier.

Smart contracts in Ethereum is compiled and interpreted by the Ethereum Virtual Machine. The purpose of the EVM and the solidity language is to ensure that code produced is Turing-complete yet not time-consuming in its operations. I.e., complex data structures could significantly impose slow execution times, especially for larger distributed contracts. As a result the programming structure are simpler than it would become in languages like Python, Java etc. In [46] Nicola Atzei *et al.* concluded that the solidity language itself doesn't have any vulnerability that can be exposed though code patterns produced by developers might lead to exploits. An example of such is the DAO attack, where execution patterns resulted in an undesirable state of the ledger, such that the program's desired logic fails in updating accordingly. They surveyed ways to mitigate this and concluded that tools like Static Analysis Tools (SAST) that analyses code flows of smart contracts could greatly reduce the likelihood of deploying vulnerable smart contracts to the public.

4.1.2 Ethereum Data Storage

TThe smart contract technology of Ethereum is known as Solidity. In short, the Solidity contract shares concepts with the Object Oriented Programming model. The contract has a set of public fields controlled by methods that can be called on externally. These fields are the state that will be distributed to the network and eventually committed to the ledger. Whenever the system has completed the change, then that change will be publicly available to anyone participating in the ledger. As a result, the BBchain project cannot store sensitive information within the smart contract fields. Their current strategy is to implement the Ethereum SWARM to their technology stack [47]. The storage system deploys a decentralized Peer-to-Peer network architecture where all files that are stored have the following properties.

- Files are referenced through its hash representation.
- Chunks, represents a limited size of data unit.

- Manifest, abstracts a collection of files into a manifest. Referenced through root hash.

The storage nodes will coexists together with the ethereum nodes, and will require that the client bridges the connection in between them. Furthermore, the storage is collision-free (for two different blobs), and identifiers are deterministic such that data with the same content will always get equal value. Whenever a swarm node receives a blob, it will split that blob into chunks and distribute them to different nodes. The Distribute Pre-image-Archive (DPA) will then calculate which nodes can store which pieces. Which nodes (or bins) will be collocated based on their distance in terms of their respective addresses. Related chunk addresses are stored in bins close to each other (none of the nodes decides which chunk they will get). The system extends the idea of incentives for the storage nodes. The nodes deposits a unit of a particular economic value, upon request for delivery, the database will give credit to itself, and whenever requesting a chunk, it will debit itself. The basic idea here is that the storage provider deposits money to provide security for the chunk. An attacker cannot corrupt these chunks and attempt to send it as the original blob, making it harder to fake the system as the root hash will be different. If the storage provider fails to provide the fragment, then the incentives mechanism applies - failing in transmitting the right values means that the providing node has lost its "bet". Consequently, nodes should decide to disconnect from these nodes too [48].
Moreover, there are challenges with the system as data can get corrupted, the different storage nodes replicate different chunks (based on the hash), if one node fails to give a part of the file, another one must be able to assist. This impediment results in a trade-off. If using high replication, then the system requires more nodes with the data, which means retrieval is faster. The consequence is that the system will consume significantly more space. If less replication, then the system depends on particular nodes not to fail, which imposes a higher risk of losing data. As a result, there is no guarantee for the chunks of getting persisted at nodes if the nodes go malicious. Hyperledger, on the other hand, does not have an approach like this, the data is stored on the ledger tied to the respective channel. With this, all transactions, intermediate states, etc. are stored in the ledger whereas the CouchDB only contains the snapshot of the last block, I.e., the CouchDB can equal to Ethereum's *Light nodes*

4.1.3 BBChain System Design

The BBChain project [49] utilizes the concept smart contracts to represent different entities of a system. They want to aggregate the data to verify based on the affiliation of

the entities in the system, which means that the entities with higher affiliations such as teachers and faculty leaders will have the authority to aggregate the data produced in the lower parts of the hierarchy. The idea is to utilize the Ethereum Public Key cryptography to register users in the system. Each student, teacher, and faculty management identifies themselves with their Public/Private Key combination. The smart contracts will maintain these keys so they can control information flow, entities' ability to invoke functions, and information association. The idea is to create a data model s.t. Teachers can issue transactions containing information about a document. Once a document is registered, other roles such as assistants can verify the state of that document. This way, the issuing of information requires an endorsement of stakeholders that is authorized to verify a particular claim about another entity, i.e., a student's grade/diploma. Once a set of courses are finalized, in terms of time, and they all form a degree. Then the responsible entity that issues diplomas can call an aggregation function that reads the state of all the sub-contracts and calculate a unique identifier using digest-values (or hash values) to commit a token-value (aggregated digests). Once done, and after some time T the token will be stored in a block in the Ethereum ledger. At this point, a student has a token-like value, which is representing a digest of the diploma in the ledger. Third-party entities can verify this token by using real-world documents to re-calculate a hash value that can be verified up against the ledger. Note that this token-value is distributed all over the Ethereum nodes, which is currently 7500 nodes distributed across the globe [1]. As previously described, the BBChain project planned to store these academic documents off-chain as depicted in the figure below and then have the client to relay these hashes to the ledger.

[1] https://ethernodes.org/

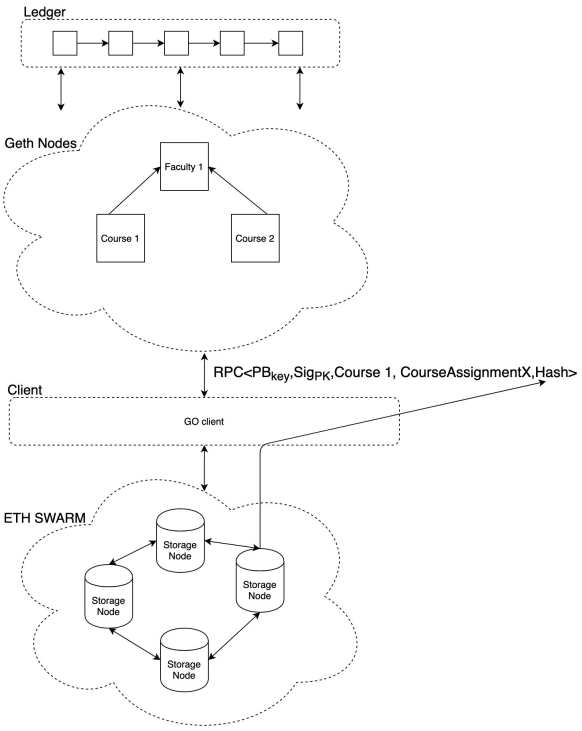


Figure 4.1: Clients is relaying data from distributed storage to smart contracts.

To verify the final diploma external clients may use the client application to invoke query transactions on the relevant smart contracts to do the following assertions:

- *Timestamp.* When was the last time someone changed the state of the course digest.
- *Integrity.* Through comparison of hashes a third party may verify the integrity of a claim.
- *N-eye-approval.* A set of N stakeholders that verify whether a subject was qualified for a certification.

The Faculty in Figure 4.1 will contain a final hash of the diploma after the faculty leader has aggregated all the digests for the sub-courses. Compared to a single records system, an adversary/corrupt node will need to convince most GETH nodes to accept value-changes. On the other hand, using a single (and maybe a replicated) system, a change of a value in a database is much easier. Even though its harder to convince the ETH nodes, it is still hard to avoid external factors such as human corruption. However, this issue is arguably not up to the blockchain to solve.
Nevertheless, while running these contracts on the public ledger means that the consensus will happen not only for these smart contracts but also for all other contracts. As

Ethereum relies on the Pow-scheme each invocation needs to include gas and a transaction fee. The higher the fee the faster the expected commit time would be as this would give incentives to the miners (or Validators) to include that transaction in the next block. The Profit algorithm 2.2 describes this behavior in Chapter 2. *Spain et al.* [50] describes that transactions with higher transaction fees requires less time tbefore committed to a block compared to those transactions that have lower fees. Furthermore, they also discovered that the average transaction throughput was around 161 transactions per minute. They also found that Ethereum produces approximately 4 blocks per minute. There is a median value of 22 seconds to wait for the inclusion of transactions in a block. Together with a slow block propagation, a client would likely need to wait for a while for a confirmation that the transaction is included in the ledger. However, there exist techniques which let uncorrelated transactions to execute in parallel ledgers, commonly known as *Sharding*.

4.2 Decentralized Identifiers

DID definitions [51] [52]	
Name	**Comment**
Subject	The owner of the data of which a set of claims are made.
Claim	A statement about a subject, e.g. a property.
Credential	Is a set of one or more claims by some entity. May include metadata such as information about the issuer and proof to verify the data content.
Issuer	Asserts claims about one or more subjects.
Verifier	An entity which receives one or more credentials for processing.
Zero-knowledge proof	Proving an attribute about a subject without revealing it.

Table 4.1: Decentralized Identifiers Definitions.

Decentralized Identifiers (DIDs) is a specification that is getting more attention as the rise as decentralized systems are emerging in the market. Essentially, a DID identifies a certain *subject*, which could be anything represented in the real world. The basic idea is to let the subject have control over the DID independently of any centralized registries such as identity providers, certificate authorities, and more. Each DID can be located at a certain URL allowing for trustable interactions between verifiers and the subject. A DID can represent a cryptographic representation of some documents. Each document can have multiple methods such as verification, revocations, or other controls. With this approach, subjects do not have to worry about how information is stored and handled in different parts of the web—hence losing control over how the information is stored, used, and accessed by third parties. By accessibility, we mean that some service stores

information about a subject but relays it to third parties for business purposes. When the information is scattered around, it is hard to control it. Currently this is governed by the GDPR. Furthermore, not all actors take proper security measures to protect the confidentiality, integrity, and availability of information about the subject. As a result, DIDs aim to solve this by decoupling the need for giving some information in pure text to third-party actors. The basic idea is that the subject (i.e. controller of information) may define who can access and verify a certain *claim* about a person. We give an example in the Scenario 4.1.

Scenario 4.1. Let us assume that a student supposedly wants to buy a book from an online bookstore. The book store claims that all students gets 40% discounts on books if they login through a identity provider tied to the university.

Traditionally, the way of solving scenarios like these is to let the Identity Provider of the University share such information. Though only proving that the university knows about the entity. However, the bookstore ends up with information about the subject in its database. What if the bookstore could relate to a decentralized system to verify information about the subject? Rather than *giving information about the subject* we can *verify the subject's claims* about a certain set of attributes - which in this case is the student's enrollment. Rather than exposing the full identity in terms of email, grades, an other information but rather a zero-knowledge proof. According to the DID definitions [53], in Scenario 4.1 the university will be a *Issuer*. The Issuer here is some organization that vouches for a statement around a subject, which could be anything from enrollment, participation in courses, or diplomas. These types of information are known to be *Credentials*. A credential is a claim made by an issuer - and a verifiable credential is tamper-evident credential that can be cryptographically verified. On behalf of the student's permission, the bookstore may ask the relevant URL to verify enrollment. Once the student decides that the bookstore does not need to read these types of information anymore - then a *Revocation* may be done. Revocation is stronger in Decentralized Identifiers. DIDs aims to protect the subject in their decision to revoke access of personal information to other parties using it. After revocation the bookstore will only know that the enrollment status of the student at the university. This does not only change the idea of information sharing - but it also provides minimal information disclosure to other parties while also knowing what was shared.
Furthermore, with approaches like these, one can also mitigate the idea of 'Self-asserted' information - like, for example, a student providing an academic record to an employer. The employer will then have to assume that the grades are correct - and not tampered. With an underlying Decentralized Ledger like a blockchain the student can still give that

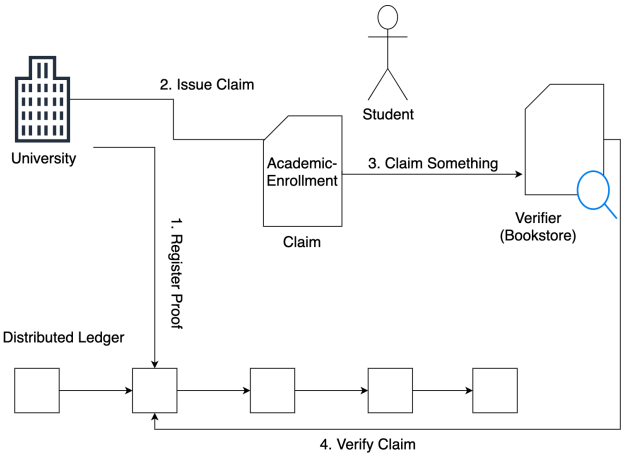


Figure 4.2: Idea of Decentralized Identifiers.

employer the diploma - but the employer may verify the integrity of this diploma with the blockchain as depicted in Figure 4.2.

4.3 Sharding the Blockchain

As described in the previous section, permissionless systems are known to be slow in their throughput of transactions. As a consequence, this is one reason why the technology isn't deployed to large-scale systems in the market yet. Still, there are attempts to make such systems more scalable in terms of transactions throughput. Some of these attempts are known as *Sharding*. The concept of the *Sharding* lies in the way the ledger-structure is distributed. Rather than maintain one ledger for all transactions, one node can have many ledgers for uncorrelated transactions. Sharding can be defined as follows [54]:

Definition 4.1. A Protocol which outputs some result R based on some result of a subset of independent processes P_i.

As Loi Luu *et al.* [54] describes, each process within the subsets of X_i needs to have a constraint function which validates transactions in X_i s.t , $\forall x_i \in \{1...k\}$, VALIDATE($x_i$)=> $TRUE$. The basic idea here is that whenever a set of disjoint transactions hits the system, then one can forward them to their respective shard for processing. Thus, rather than processing them sequentially, they can be parallelized and processed in isolation. Upon finishing, the validation function can assess the final result(s) of each shard and determine whether the outcome can be committed for an example in the main ledger or syncing with another ledger. Their protocol *Elastico* segments the available computing power in to *Committees*. Each committee contains a static number of members, which runs a byzantine fault-tolerant protocol; these partitions

are their shards. On top of these shards, a final committee merges the results from the shards, validates it, and then commits. With this approach, blockchain systems can scale horizontally and linearly of the shards. Although scalability is important, it imposes some challenges for permissionless models as they lack trust in the system (i.e., a PKI to trust). An adversary might simulate many processing units, thereby creating a Sybil attack. These attacks would indeed impact the security as there exists fewer nodes per chain affecting the *Safety* property of the blockchain, as an adversary can silence honest miner nodes and form its pool to create blocks within the shard. However, Sharding is not yet implemented in the current version of the EVM, though they plan to roll the feature of sharding into different phases. According to their road map [55], they will initially attempt to decouple the need of the PoW-consensus algorithm and move to a Proof Of Stake Scheme that has significantly better throughput, yet maintains the incentives that PoW-scheme gives. Essentially Proof-Of-Stake is when miners put their stakes to create a block - failing in doing so will indeed result in a lost stake, thereby an *incentive*. What PoS essentially do is to replace the Resource usage in PoW and shift it over to a betting system. Furthermore, the Ethereum Developer team will focus on integrating a sharding feature for the EVM gradually to increase the scalability and usability of the framework such that it can cope up in performance with existing systems such as credit card processing.

Besides this, Hyperledger's channel abstraction 2.6 defined in Chapter 2 does provide a way of sharding the transaction ledger in Fabric's environment. The reason being that each channel will run its own set of rules, Consensus instances, and ledgers. To synchronize a set of ledgers into a single one they have the following function in their ChaincodeStub API, which allows for reading states from other channels using *stub.InvokeChaincode(ccName, channelName, args..)*. This function will not allow for writing values in other ledgers, however it can be used to aggregate results from N channels into one specific channel. Then deploying a chaincode that aggregates the results from other chaincodes located in different channels. This way, the external application can load-balance their traffic to different chaincodes located on different channels. Isolated instances of RAFT means that blocks will not get filled with other transactions created by other chaincodes during the RAFT term(s). Fitting this into definition 4.1 - the $VALIDATE(x_i)$ function is the endorsing peers and orderer's job, per channel instance. Upon reading a value V_c from chaincode(s) in different channels $C_{shards} = [C_{pull}]$ the *invokeChaincode* function will not perform any validations of the state read - it needs to be assumed that read value V_c is correct as it is reflected in the ledger of the peer. Though this is a weak assumption considering the *Safety* property in the case where the request $C_{current}[currChaincode] \xleftarrow{pullData} C_{pull}[chaincodeId]$ where $C_{current} \neq C_{pull}$ may be forged by the ledger. Though we argue that this is safe as described in our Experimental Chapter - any inconsistency in the peers' ledger is detected

and the chaincode container will crash. If malicious request forgery would occur, it would indeed be out of bound of what the system model assumption is - namely, Crash Fault Tolerance. Though what we cannot assume is that the ordering service for that channel is honest, meaning that the blocks produced by it might be conflicting with what the original proposal(s) was.

4.4 A Practical Byzantine Fault Tolerant Ordering Service Protocol

As previously described, Hyperledger Fabric supports pluggable consensus, and currently, there is no official consensus implementation to tolerate Byzantine Faulting ordering nodes. Though making a consensus protocol truly BFT is hard. Therefore, current solutions offer *practical* BFT, which means that the algorithm optimizes only a few aspects of this system assumption. Marko Vukolić *et al.* proposed a BFT ordering service for Hyperledger Fabric v1.3 that used a wrapper around their existing BFT-smart consensus protocol. They override the default implementation of the HLF protocol by specifying configuration paths in the relevant docker images s.t. their implementation can be run instead. Moreover, their protocol is implemented in Java, exploiting the netty framework to handle many concurrent requests simultaneously, which is arguably the benefit of the framework. They also allow for authenticated *view* changes during run-time, meaning that re-configuration can happen while the consensus runs. Furthermore, they adapt a transaction flow that let multiple clients *propose* at the same time. Within the same epoch, the Java Netty framework will forward the request(s) to a message processor that handles the message and puts in a *decided queue*, which is included in the replies upon finalizing the current consensus round. To decide a value in a round, the processing needs to issue a *Write* and *Accept* messages, which are cryptographic hash representations of the original batch of transactions sent from the leader node. They will validate the MAC to check the integrity of the message whenever a replica receives one of these messages. With digital signatures, they claim that its hard to forge these requests, which may trigger new leader changes as a result of not being able to decide in an epoch. This way, they are more resilient against malicious behavior in BFT-smart. So to decide a value the assumption is that when a node gets $2/3 + 1$ of correct *Write* messages, an *Accept* will be sent to indicate that the transaction(s) have been decided for the current consensus round [56]. To implement this in the HLF-environment they created a *Frontend* node, which utilizes the Hyperledger Fabric Java SDK to create a communication protocol that talks to the ordering nodes from the peer(s) trust domain. The purpose of these nodes is to relay the transactions on behalf of clients and ensures that relevant peers receives block for their channel. The frontends will collect these

blocks and check if there is $2f + 1$ matching blocks from the ordering nodes whenever the ordering services produce blocks. Comparing this to the standard solution with RAFT, what they do is essentially relaying transactions and blocks through a new node type (the frontend). This node type collects a majority of *Votes* from the ordering nodes to ensure that they all produce the same block. In the RAFT implementation, only a leader ordering node will forward blocks to the peers after it has been able to replicate a certain block in the logs of the follower nodes. BFT-smart does this differently, as all ordering nodes must return the same block as the majority is formed, making it more resilient for malicious behavior between the ordering nodes. Nonetheless, what BFT-smart doesn't protect against is a traditional DoS attack from the leader in the network to reduce the throughput of the system. Furthermore, the change of view in the network is supported amongst the ordering ndoes. But, the change of view between the frontends is not supported as Hyperledger's Go implementation doesn't have support for transfer ledgers between processes [57].

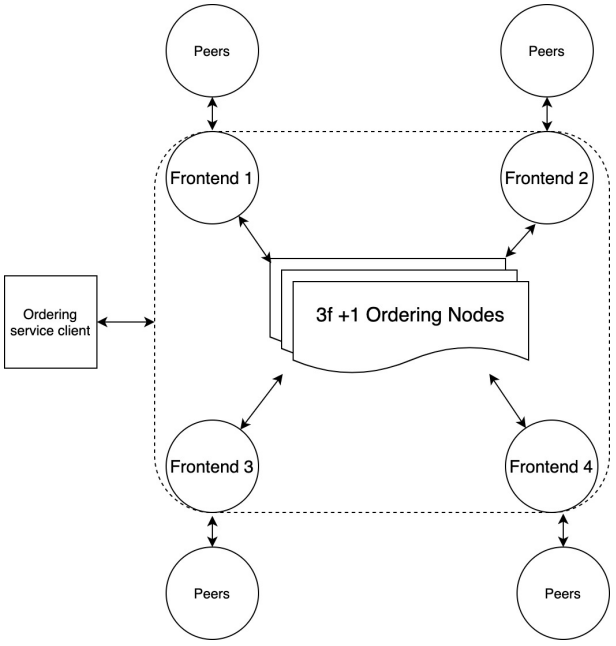


Figure 4.3: BFT-smart HLF architecture.

4.5 RAFT Practical Byzantine Fault Tolerance

Even though BFT-smart implements a beginning for a BFT protocol for Hyperledger, there also exists work on creating a BFT protocol for the RAFT consensus protocol. The current implementation only tolerates nodes that crashes, not nodes that *lie*. What if some orderer randomly sent vote in an epoch and to a node which doesn't have an updated replicated log. Believing it is the most updated node and elects itself as a leader.

These behaviors will cause a scenario where two nodes will be elected as leader in the same epoch. As a result of this, there exists significant research on making RAFT BFT. According to Fabric RAFT is the first step towards getting a decentralized and BFT ordering service. As proposed by Copeland *et al.* they created *Tangaroa* which is RAFT with Practical BFT properties. They use digital signatures extensively to authenticate messages exchanged by the protocol while integrity checking the messages. Digitial signatures are to ensure that the messages can't get forged nor sent by unauthorized entities. The digital signatures help in the verification of a leader that claims to be elected, upon the election event the followers will validate and count all the RequestVoteRespones. By validating all leader elections will preserve the *liveness* property as all new leaders need to prove that they got a majority of votes by the followers.
Furthermore, in the CFT Raft the leader will forward the current commit index during the AppendEntries. This can create problems where a malicious leader can tamper with the index, resulting in proposals not being included in the final replicated log. In Tangaroa, they move this logic to the followers. Upon committing values, the followers will maintain the latest committed term index. This means that the leader cannot increase the term index until all previous logs have been replicated and committed. The storing of the term index at the follower nodes ensures that *Safety* is preserved as followers can decide whether the quorum of nodes has adapted the value and that they follow the same ordering.

Furthermore, they also allow clients to interrupt the current leadership election if the election doesn't make any progress. They do this to enforce the *liveness* property as described in Chapter 2. The liveness is preserved because the external application can restore the availability of the system.
When it comes to the RAFT-logs they deploy an incremental hashing strategy, this means that each replica will create a linked-hash-list where each appended item is hashed together with the previous log entry. Other nodes can then verify the log quickly by assessing the hash and verify the corresponding signature of the message. This will preserve the *Safety* property as leaders may not forge, modify, delete or re-order the client requests issued to the system. Nevertheless, malicious nodes in the network may attempt to start new leader elections even though they're not necessary. This will influence the *Liveness* of the system as frequent leader elections may slow down the consensus. To deal with this, they infer the following restrictions to cast votes. The first is if followers receive heartbeats from a leader, then all new RequestVotes will be effectively ignored. Secondly, only eligible nodes may ask for a reply on RequestVotes, meaning that the node's term index, hash of log entry must be up to date, and that a vote from the node is sent in the current term. Whenever a node believes claims that it is a leader. Then the followers will expect that the candidate will forward heartbeat messages with all

the signed votes. Followers will then only listen to proposals from that leader, given that the signatures are valid and form a majority. They also claim that implementing a round-robin leader election algorithm will remove the possibility of getting split votes, which means that malicious nodes cannot be elected as the current leader. Using the round-robin approach, they will achieve a RAFT protocol where each term will circularly have an alternating leadership. Meaning that for each term a non-faulty node will be elected as leader. This means that the network can detect whether the elected node is faulty or not and ensure that a correct leader is always running the network as long as $2f + 1$ nodes are assumed to be correct [58].

4.6 Best way to deploy Hyperledger Fabric Project To a Development/Production environment

Each Hyperledger fabric (HLF) component requires a specific Membership Service Provider and certificates. With the help of Docker the process is somewhat error prone yet manageable. Once successfully configured its relatively easy to have an environment to develop on. The docker images retrieve their configuration through their mounted volumes on the host machine which allows for just providing external commands to start the network. However, shared storages is not necessarily always available. Furthermore, it is not a good practice to store all the MSP-information at one place which introduces a single point of failure. As a result, deploying the HLF project is challenging in terms of real-world systems in order to manage keys and more in a secure way and manageable way. In regards to pushing the configuration to the network such as channels and chaincodes through bash scripts is either not a proper solution as they would increase in size and become challenging to maintain. In this subsection we assess some of the existing work out there that addresses some of these problems. We summarize deployment issues to this table (based on the provided Hyperledger Fabric Samples).

Fabric Architectural Concerns Deployment		
Concern	*Comment*	*Affecting*
Bash Scripting	N scripts for N installations	Maintainability of Configuration gets exhausting
Docker compose files Port numbers	Hard to maintain port numbers at production site	Maintainability gets exhausting
Network maintenance using bash scripts	Lifecycle of chaincodes in bash scripts is hopeless as they require different names, same for channels and joining peers to a channel	Network Operations doesn't scale for distributed systems
MSP	Each node must have a specific MSP attached to it (including TLS certificates)	Increasingly difficult to scale MSP distribution for larger systems
Assuming access to containers	Kubernetes doesn't work like that, containers (pods) are abstracted away	Moving to production requires total refactoring of current setup.
Assuming usage of single volume in a VM	Only works for a single virtual machines. Such behavior doesn't work in Kubernetes	Affects scalability
Other	In general a lot of configuration required for network to operate. General solution is to extensively rely on static scripting files.	Understand-ability

Table 4.2: Issues with Fabrics Architectural assumptions.

4.6.1 Existing Work On Launching/Populating the Hyperledger Fabric Network

AidTechnology is a company which works on decentralized applications[2] that follows the W3C standards for Decentralized Identifiers, located in in Dublin, London and New Work. Last year they had a workshop on how to deploy the HLF network on Kubernetes. They exploit the *Helm* framework with the help of Charts. A chart (in context of Kubernetes) is a collection of files that describes a set of Kubernetes resources. It expects a certain file structure in order to work [59], see [60]. The charts are essentially bundles of templates defining Kubernetes components. Charts for HLF [3] are demonstrated using Kubernetes on Azure, however as demonstrated in the video workshop [61] part 1 and part 2[62] the process of manually generating the MSPs and the corresponding crypto material still persists. This includes manually creating secrets for TLS-communication

[2]https://www.aid.technology/
[3]https://hub.kubeapps.com/charts?q=hlf

and MSP certificates too. Moreover, one would also need to manually deploy and configure n copies of Peers, CouchDb, Orderers, CA separately. As a result the process of launching the HLF network are still time consuming and would be hard to maintain as the configtx.yaml and cryptoconfig are still used with additional files. The time required for provisioning a network is still not reduced and would likely to become a expensive and a time-consuming process in real-world project.

Another Contribution found on **Github, by feitnomore** [63] does not use the Helm framework and only plain kubernetes yaml manifest files. The solution to distribute configuration to launch the network is to have a shared file system with the help of a Network File System server setup. The shared storage will then contain the relevant scripts (same as with the docker-compose system) and configuration files needed. The NFS file would require to have the fabric-tools in order to run the configtxgen and crypto-config to generate the necessary data. Meaning that configuration and setting the right parameters are done outside of the server. However, for each node in the network there must exist a kubernetes yaml manifest file. This results in equal amount of configuration required as with the docker-compose network (in addition to Kubernetes service files + the shared file system). In order to populate the network with channels and chaincode one would still require the process of creating scripts to configure peers and orderers - which is the same as with docker-compose.

On the other hand, APG and Accenture have a Github Repository [64] for configuring Hyperledger Fabric network easily through Helm charts, with inspiration from [61] [62]. Though, they automate processes of configuring, updating, and backing up data through Argo workflows which uses the already existing crypto-config and configtx files. In their approach they extensively use these charts to read and build the network from the configtx.yaml and crypto-config files, only with an additional network definition file to create the required Kubernetes resources. The .yaml files in crypto-config and configtx works nicely with Helm as they rely on parsing .yaml files in their chart components. Compared to the other approaches discussed, it is much faster to get up and running with a Fabric network as all the requirements of knowing Kubernetes resources are abstracted away. As a result, they only introduce one file that the network administrator need to fill out for a kubernetes cluster. This file is a network file which specifies which channel each organization need to join, and where to install specific chaincodes. The helm charts will then create the Orderer- Peer-Organizations defined in the crypto-config accordingly. They also create all the crypto-material, pods, services, configures the resources, and launches the network. In this approach there is minimal requirement to create scripts to provision the required architecture for the project. A non-technical person would be able to start a Fabric-network on Kubernetes Cluster without any prior knowledge about Kubernetes, bash scripting, nor fabric concepts and configuration. Furthermore, Helm

also allows for updating the Helm Charts by updating the values file - meaning that one doesn't need to reinstall all the resources within the cluster.

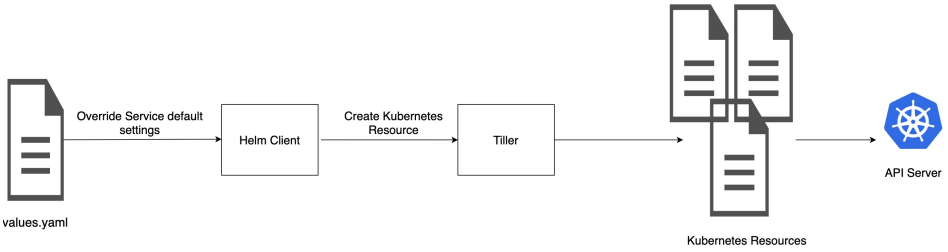


Figure 4.4: Helm abstracts away the need for configuring Kubernetes resources. A service here is essentially any software which can run in K8s.

To configure the network they use Kubernetes workflows. Workflows allows for running pods to configure the network. The standard way of defining workflows in Kubernetes doesn't allow for orchestrating sequential or parallel workflow jobs. As a consequence, they leverage this work to the Argo-framework. Argo allows for defining both sequential and parallel workflows which means that they can get rid of the bash-scripting and rely on the pods - which in turn only reads an existing .yaml file defined in previous steps. Furthermore, there is also minimal amount of addition configuration files to be added. As a result, the process of launching the network is less error prone, less time consuming and much easier to maintain as there is almost no required scripts in order to run and maintain the network. The *Argo* workflows makes the process of updating chaincode, configuration an easy process for the operator. PIVT proposes a important step for Hyperledger Fabric to become more close to a production ready system.

4.6.2 Cross Kubernetes Communication in PIVT

The MSP is created by the configtxgen tool located in the Fabric Binary files. Though this works fine for a local environment it also imposes some challenges. Recall that the MSP contains vital information such as secret keys, trusted certificates (like which CA should issue new users), and details of the organizations administrator. For a project running in different Kubernetes clusters one will have to run this tool for N clusters. Furthermore, the ordering/peer nodes will have to be configured with a either an ingress or a load-balancer to allow for external communication. Server technologies such as *Nginx* has helm-templates that can be overridden. Essentially, the service type and port need to be configured accordingly - making it easy to expose the fabric cluster to external applications. Furthermore, each of the peers/orderers need to be reachable by their respective service name and port number in order for cross-communication to work. This part of the PIVT project is where it can get cumbersome, as each of the

host-alias:ports combinations need to be merged in to each cluster which then needs a restart for changes to take effect. Note that nodes outside a certain cluster may run as bare-metal installations or docker containers - though, not recommended.

4.6.3 Backing Up Data In Kubernetes in PIVT

PIVT introduces a way of backing up data in the cloud using the Argo workflows. However, to back up data in the cluster a persistent storages needs to be initiated with a persisted storage. To do so, the Argo tool must be configured to use a local repository - the Minio tool is the recommended way of doing this. Argo will then use this repository to pull the data out from the cluster to its local repository which in this case will be Azure Blob Storage. The system then need to enable persistence on startup, the system administrator extends the values file included for the Fabric Helm charts to enable persistence respective to peer/orderer/CouchDB. The keys read by the chart assumes a false/true value. If enabled, the chart will enable the default Kubernetes storage-class resource. The workflow for backing up the data will then disable the peer/orderer/CouchDB containers and mount a Rsync container in the pods. The Argo Workflow will then have to traverse the different pods, get the data from the Rsync container, and push it to the Azure Blob Storage. During restoring of the state(s) the Argo-restore-workflow will pull the data from the Azure Storage and then re-traverse all the pods and write the respective content of peer/orderer/CouchDB. Note that the system needs to be down during these executions. If we compare this to other solutions - using the docker-compose solution, one will have to copy ledger data from peers, data files of the corresponding DB, and orderer files out from the container and in to a docker-volume. Then Docker containers must then be mapped to volumes. This means that respective data etc is directly written to a persistent storage meaning that no downtime is required. Same can be done using the NFS-approach where each of the node in the network has a persistent volume in the NFS though it would require more Kubernetes resource definitions. Nevertheless, as these approaches already extensively rely on N config-files for N organizations adding more configuration will not mitigate the issue of already having to rely on multiple files for getting a network up and running. In terms of ease and usage - the PIVT-way of doing it seems like the fastest and easiest solution to adapt for a Fabric Project backup. Though it will assume that Fabric architecture runs in a cloud environment.

Chapter 5

Design and Implementation

This chapter will give an insight on how we chose to replicate a similar system for vouching/verifying academic records as done in the BBchain project. Note that the system will deviate in some terms as HLF have other assumptions as the permissionless scheme(s) such as Etereum. We first give an overview and the reasons of taking the approach we did. Then we will provide a more detailed view of the implementation itself by describing how we can derive the features discussed in the overview.

5.1 Data model design

Academic institutions today typically have faculty, courses, and people who participate in a course. The general structure is that a student must finish a set of courses before the faculty can issue a diploma of certification. The BBchain project initially assumed this approach when vouching for academic records. Data modelling these hierarchies is found in fig. 5.1.

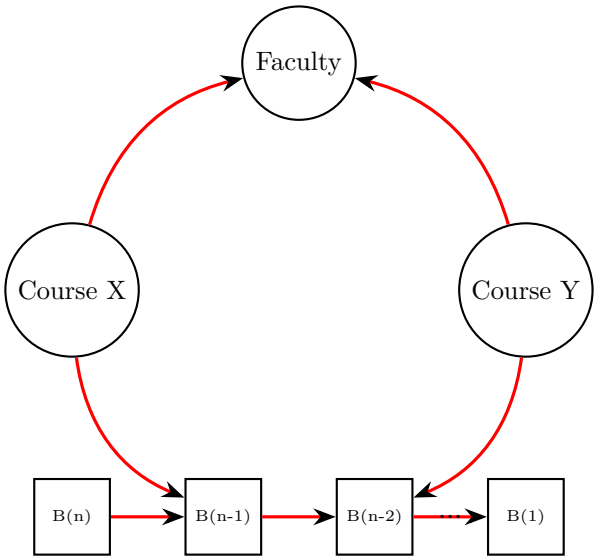


Figure 5.1: Initial Data model for BBchain project for academic verification.

Here the idea is to let the courses to handle the part of issuing information (or credentials) specifically tied to a particular course, e.g., the grades, assignments etc. Each course will then store N credentials on the ledger, where each of the credentials signifies a coursework. The meta-data stored on the ledger is timestamp, block number, issuer address, student address and a final concatenated digest of the coursework(s) representing the course proof. To finally commit the digest there need to be a set of owners that need to sign this information before it can be regarded as valid. Typically these are people which is related to the grades-process. There is three benefits of storing the meta-data this way.

Scenario 5.1. Assume some external entity that wants to ensure whether the credential of some finalized document D. Where D is formed by a hash-concatenation of $H_{D_{finalized}} = H(H(D_{course1})||H(D_{course2})||H(D_{course3})$. Then to verify that document one will need to retrieve digests for all three documents (in sequence) to check whether $T_{verify} = H_{D_{finalized}} == H'_{D_{finalized_{calculated}}}$

There are several assumptions in the scenario given. First one being that the digests are stored on an immutable ledger L s.t. no one can change the state of the final proof once signed. Secondly, the entity verifying the document has access to all the three documents externally so hashes can be calculated for checks. Moreover, those entities signing the credential cannot be impersonated by some other entity. Lastly, assuming that the hash-algorithm is not broken and that the Private keys used is kept secret then the verifying entity can be sure that the system reflects the true state of the document D. Moreover, using the timestamps and digests the entity can know *when* and *what* was issued at a certain point in time. Meaning that if T_{verify} is true, then the document is in its true

original form as when it was originally signed by the course administrators and the student.

Each of the nodes in the graph 5.1 are deployed as smart contracts. The idea of the faculty node is to aggregate all the course-digests to one single hash as described in Scenario 5.1. Though only an administrator of the network should be allowed to do so, i.e., the faculty leader. The same applies for the course contracts where entities such as teachers have the ability to give results of exams, teaching assistants gives results on assignments, and finally when student agrees, he/she can sign and the credential can be seen as valid. We can form an algorithm for the aggregation of digests as the following.

Algorithm 5.1 Aggregator

Result: A list of valid aggregated proofs.

```
contracts := [] ;                                   // Ordered set of contracts.
mapOfDigests;                                       // Aggregated set of proofs for each user.
owners[] ;                                          // Who may invoke the aggregate function.
if client.identity ∈ owners then
    for user ∈ mapOfDigest do
        aggregate(contracts, user, contractIdx, mapOfDigest) concatenatedSubDigest :=
          mapOfDigest.get(user) mapOfDigest.put(user, sha256.hash(tmpConcatenatedDigest))
    end
function aggregate(contracts, user, contractIdx, mapOfDigest):
    if contractIdx ≥ contracts.size then
        return ;          // All contracts are iterated through, end recursive call.
    else
        subDigest := contracts[contractIdx].call("readProof")) tmpDigest := mapOfDi-
          gest.get(user) mapOfDigest.put(user, tmpDigest.concatenate(subDigest))
    end
    contractIdx = contractIdx+1 return aggregate(contracts, user, contractIdx, linkedList);
     // Recursively concatenate credentials/credential proofs digest.
```

The Algorithm above will have a map of initialized users. The procedure will go through the keys of the map and call the *Aggregate* function. The aggregate function recursively go through all contracts and append the digest values s.t. $concatenatedSubDigest := (H_1||H_2||...||H_n)$, and finally creating a hash value that represents a credential. Both Ethereum Smart Contracts and Hyperledger Fabrics' chaincode do support operations like this.

Though, this data model fits the use cases of verifying academic records it will somewhat be limited for other uses cases where the organizational structure is more complex in terms of depth. As hyperledger is highly configurable in terms of architecture we want to create a data model which supports various configurations and interactions between organizations. Therefore, we abstract away the idea of Courses and Faculties and rename

these to *Issuers*. The task of these is to either collect a set of *Credentials* or a set of Proofs. A *Credential Proof* is essentially a way of proving that a subject has acquired the credentials needed to sign-off that they form some kind of qualification - which in this case could be a Academic Record like a diploma or a course. Assuming that most organizations follows a tree-like hierarchy one can build up a credential in a bottom-up manner.

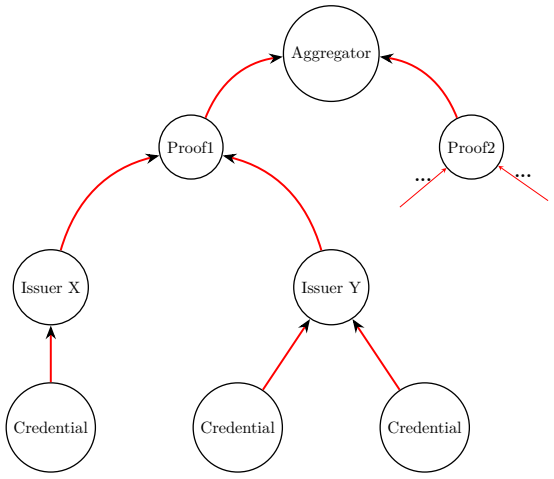


Figure 5.2: Generic Data Model. A issuer creates a proof based on N credential leafs.

In Figure 5.2 we propose the generic way of registering credential towards Issuers. Upon finalizing all the required Credentials (leaf nodes) there exists a proof. To form a proof, then an authorized entity need to *Aggregate* the credentials. Then all the intermediate nodes are aggregators of which authorized entities may only call (typically entities which is higher up in the hierarchy). At the top we have the final proof which will consists of all the child proofs. We can define our data model as a Graph $G = (V, E)$ where V stands for Vertices and Edges E are the connection between a Vertex of type P or type C where $P = ProofNode$ and $C = CredentialNode$. Graph G is a finite directed graph with no cycles. Meaning that all intermediate nodes cannot point to the same Vertex at the same height H or any $H - n$ where $H > n$ and $n > 0$. Following with defintions.

1. Each *Branch Vertex* C represents a credential. Where a credential can be any data provided by the outside world, e.g. a document, chaincode accepts N documents (credentials).

2. $\forall_V \notin \{C\}$ represents the $ProofNodes$ P, these are chaincodes.

The general idea is to allow the operations/service teams to define the overall hierarchy of the organization they are installing the system on. The P vertices and relation between them are important to correctly model the nature of how the organizational hierarchy. A

vertex of type P can collect credentials - but may also collect proofs from $[P_1, P_2, P_3, .., P_n]$ depending on the references defined in the chaincode. For each Aggregator in the graph, an aggregation needs to happen before the next one. The idea here is that authorized people will be able to pull data from P or C vertices. Note that both types may refer to multiple nodes in height $H + 1$ in cases where the Credentials/CredentialProofs might be used for several different Credential proofs at a higher height. The top level will then introduce a final proof signed off by the top-leader (ideally) for that final proof. The general idea is that we build up on a trust-chain at the issuer. The leaf nodes are produced by either teaching assistants, these credentials then form the base of a final *proof*. Then the final diploma (top level proof) consists of an aggregation from all of the nodes under it containing the faculty and everyone else that took part of the signing process. The aggregators are protected such that only a set of known identities can call this function and pull data.

5.1.1 The Issuer interface

Following the definitions from previous chapter in table 4.1 we need the Issuer instance to obey an interface which allows for creating verifiable credentials.

Module 5.2 Issuer Interface

Module:

Name: Issuer, **instance** I Uses ChaincodeInterface section 3.4.1

Events:

isRevoked: (≺ READ | Key k ≻): Assess whether the key k is revoked or not.

registerCredential: (≺ COMMIT | Subject key, Credential c ≻): Provides a credential for a subject identified by key.

revokeCredential: (≺ COMMIT | Subject key, Credential c ≻): Revoke the a credential for a subject.

confirmCredential: (≺ COMMIT | Subject key, Credential c, PrivateKey K ≻): Some authorized identitfy confirms that a credential conforms to the expected result.

verifyCredential: (≺ COMMIT | Subject key, Credential ≻): A verifier may verify the validity

aggregateProofs: (≺ COMMI | Subject[] keys, Chaincodes[] contracts ≻): Aggregate the refered chaincodes set upon initialization in the issuer definition.

Properties:

Zero-knowledge proofs. A claim can be verified without revealing the actual secret.

Verifiable. The origin and validity can be verified.

Signed. Only entities with a given private key may confirm and register credentials.

Controlled. The subject may control access by external parties.

Locked. Some malicious entity cannot change a proof at a node without convincing the rest of the network.

Module 5.2 can be implemented as a chaincode using the *init* function to set the definition of the issuer - such as registering users, owners (authorized) nodes, and alternatively all the chaincodes to pull data from. To mitigate the risk of recursive calling between chaincode all levels need to aggregate and store the proof before Algorithm 5.1 can be executed. Nevertheless, the events in Module 5.2 can be implemented in the *Invoke* function. Furthermore, before an issuer decides whether a subject's proof is valid a *quorum* property will be used to assess how many credentials have been asigned by the *owners* before setting a credential validity to true. Upon $\#credentials == quorum$ an issuer will update the proof-status, change its validity from false to true and commit the result on the ledger. Whenever the *aggregation* happens the issuer being called on will assess the value of the proof to check if its valid before pulling it.

5.1.2 The Recorder Interface

As we will discuss later. All the issuers will use Private Data Collections to keep the credentials secrets such that the unauthorized nodes in then Channel 2.6 cannot view the private details about subject. Furthermore, we'd like the meta-data to finally be exposed to all members of the channel. We also want the possibilities to not only verify a credential, but also the original content of a proof (i.e., the original assignment submission, course-certificate, etc.). We therefore introduce the *Recorder* which eventually will be the holder of the proofs for different subjects in a trie-like manner. Furthermore, a course also changes every year - meaning that we cannot deploy a chaincode for each of these as it would simply not scale in the long run. We therefore allow for modification of which students is participating in a certain course. Though this is only allowed once for each duration of the course, e.g., twice a year (depending on how long a course/degree duration is).

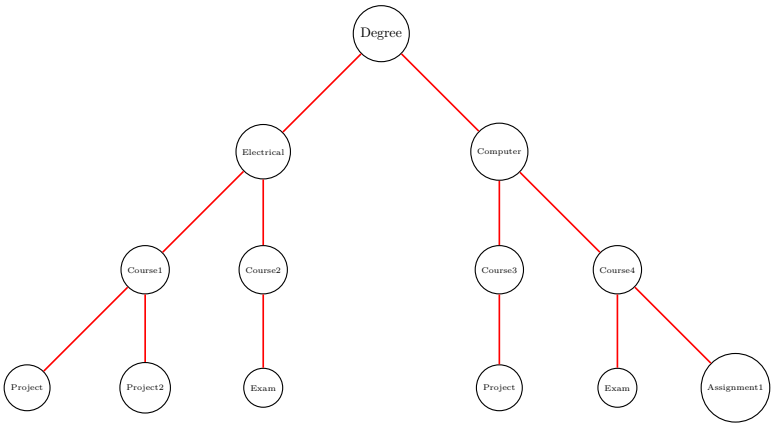


Figure 5.3: The recorder builds up a trie based on the proof-tree from an aggregator.

The idea in Figure 5.3 is to create a trie such that it is easy to verify a claim. E.g., a student claims that he or she finished a course in Web-programming. Then rather than verifying the whole degree (or some other document) one can provide the hash of the course and let the verifier check if this exists. The link between a recorder and the issuers is that the *root issuer has a link to the recorder* on the same Fabric channel to push the final generated proof. The recorder then resolves the credentialproof to a trie where each node will reference a hash of a proof / credential data.

Algorithm 5.3 Building Trie

Result: Trie

TrieNode root := null

function `insertTrie (`*Key,Node*`):`

  if *root == null* **then**
    root := new TrieNode()
  TrieNode current := root List[] keys = resolveKeys(key) `insertElement(`*keys, current*`)`

function `insertElement(`*TrieKeys, currentNode*`):`

  Trienode node subKey := TrieKeys.First() **if** *currentNode.children.hasKey(subKey)* **then**
    currentNode = *currentNode*.children.get(subKey) `insertElement(`*TrieKeys, currentNode*`)`
  else if *TrieKeys.isEmpty()* **then**
    node := new TrieNode(*currentNode*.Hash) current.children.push(subKey, node)
  else
    node := new TrieNode(*currentNode*.Hash) *currentNode*.children.push(subKey, node)
    `insertElement(`*TrieKeys, currentNode*`)`

Consolidating the Algorithm 5.3 - the idea with the *Keys* is to use the credential proof's descriptions to push an object to the Trie. E.g., "degree.computer.course1.project2". The algorithm *insertElement* will then traverse the tree to insert the element in right position. Similiar key-structure is needed while asserting claims about parts of the degree. Each node will reference the hash of the credential (leaf nodes) and credential proofs (intermediate nodes).

Whenever the aggregating Issuer has finished pulling the data it will commit the proof and forward it to the recorder - given that a Recorder's address is defined in initialization of that Issuer. Before the Issuer does so, it will need to consult all the proofs and validate whether the proof for each registered user is valid.

Algorithm 5.4 CredentialProof to Trie

```
Result: Trie which contains hash values of Credentialproofs in intermediate nodes and hash of
        actual document in leafs.
TrieNode root := null
function createTrie(CredentialProof root):
    Trie trie := new Trie() ;                                          // Initialize Trie
    Stack keys := new Stack()  ;                                  // "root.course.credential1"
    LinkedHashSet visitedNodes := new LinkedHashSet() ; // keep track of visited nodes.
    DFSProof(keys, root,visited, trie)  return trie
function DFSProof(proofKeys, node, visited, trie):
    currNode := node  if not currNode.key ∈ visited then
        if (type of currNode == 'proof') then
            keys.push(currNode.getDescription()) ;          // Keys used to traverse trie.
            visited.push(currNode) ;                           // Do not visit node again.
            trie.insert(kb.peekString(), currNode) ;   // A proof is not a leaf, don't pop
             key.
            for node ∈ currNode.children do
                DFSProof(keys, visited, node, trie) ;          // keep traversing children.
                keys.pop() ;                                 // Child node visited, pop key.
            end
        else
            keys.push(currNode.getDescription()) ;              // The node is a leaf node
            trie.insert(keys.pop(), curretNode);       // Pop current key as this is a leaf
```

The Algorithm 5.4 is executed by the Agggregator with a registered recorder chaincode-address. The Trie is only built whenever the quorum of credentialproofs collected are marked as *valid*. The stack of keys is used to build up the underlying Trie as showed in Algorithm 5.3. When the Trie is Committed it is ready for the *Verify* method in the Interface 5.2. A student can forward a Hash value to a external party together with the corresponding key - the verifier will then know if the student was enrolled in a course, finished some important assignment, or if a degree is finalized. Furthermore, the verifier can also forward the hash-representations of exams, diplomas, assignments, and projects s.t.validation may happen. Furthermore, if the verifier is unsure of the origin of information he can verify the creation of the chaincode (e.g. the metadata stored upon initialization the chaincode) - and check the identity which created it (E.g., the faculty leader).

5.2 Implementing Design in Hyperledger Fabric Environment

This chapter will dig more into the technical section of the implementation and will explain how fabric-concepts comes in to play for the chaincode-types Issuers, Aggregations, and Recorders. We will first give an overview of the architectural design choices in relation to channels. Then, we will emphasize on how the chaincode is implemented in Java.

5.2.1 Hyperledger Channel Setup

A solution to provide Academic records on different channels would be possible. However, as previously shown - this would significantly increase the amount of configuration required. As a result, we only define one channel where all the universities should interact with the same ledger. Each university will adapt our data model, deploy it to their endorsing peers. Additionally, we argue that Hyperledger scales well Horizontally as all peers does not have to execute the same chaincode, which is the consequence of Endorsement Policies 2.4. Furthermore, we also handle critical information and do not want the ordering nodes and unauthorized peers (other universities) to see this information. As a result, only the organizations defined in the Endorsement Policy and the same organizations reside in the Private Data Collections 2.7. Consequently, the non-authorized universities and the ordering nodes will only work with digests of the transacted values which reside within a university as depicted in Figure 5.4.

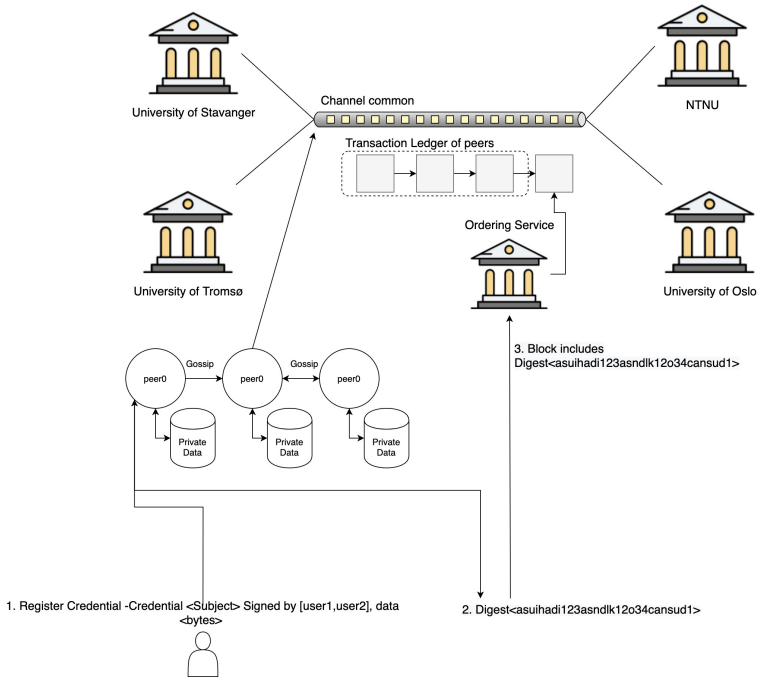


Figure 5.4: A university will never reveal its information to the rest of the network when interacting with clients.

The flexibility of both the Endorsement Policy and Private Data Collections allows for collaboration between one or more organizations - say an exchange program. E.g., a Norwegian university deploys a chaincode which can interact and transact with courses defined for exchange students in other universities. The collaboration would require either an 'OR(A,B)' or 'AND(A,B)' policy for both the endorsement policy and private data collection. We also use *Transient fields* of chaincode to hide the parameters sent to the peer nodes as well, as these are normally included in the transaction proposals.

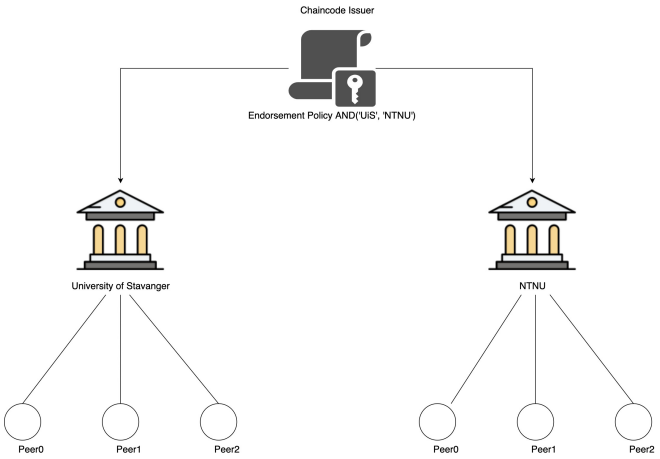


Figure 5.5: Collaborations can be modeled through endorsements.

In Figure 5.5 all peers in each organization will install the chaincode - the universities can then define a common set of credentials/credentialproofs to issue where entities from all universities need to sign the credential(s). The Application Channel must have relaxed policy registerd at the ordering service.

```
    Policies:
        Readers:
            Type: ImplicitMeta
            Rule: "ANY Readers"
        Writers:
            Type: ImplicitMeta
            Rule: "ANY Writers"
        Admins:
            Type: ImplicitMeta
            Rule: "MAJORITY Admins"
```

Listing 5.1: Application Channel definition.

This allows for a relaxed policy in terms of cross-organization collaboration within the same channel as these will default to the organizations policy definitions. However, a problem that occurs then is that users might read or write to other chaincodes. To cope with this problem Hyperledger introduces the concept of the *ClientIdentity* library [65]

that can be used in the chaincode. Essentially this allows for retrieving the transaction caller's ID which is retrieved from the caller's certificate. The *Authenticity* of caller is guaranteed by digital signatures as described in Chapter 2. As a result, at the chaincode-level, one can fine-grain access-control all up to specific users. We utilize this for both aggregating credentials, registering credentials, confirm credentials, and reading the credentials such that only authorized entities may be allowed to finish an invocation. This also opens for less configuration of the channel itself, and allows the operations which installs and instantiate the chaincode to define which users may be part of the administrative tasks.

5.2.2 The Java Implementation

We introduce the general concept of the implementation of our chaincode. We do only have one chaincode as this would make the deployment process easier as managing different types would be harder to manage. The type of chaincode is resolved automatically upon initialization. The initialized parameters will be committed to the ledger.

- The *Issuer* - must be initialized with participants (students), description (course name), time-scope (duration of course), quorum (i.e. how many credentials), organizational identifier (meta-data only), collectionsId (name of the Private Data Collection), and a set of sub-administrators (e.g teaching assistant, censor, etc.).
- The *Aggregator* - must have the same parameters as the issuer, but will require an array of chaincode ID to pull data from.
- The root *Aggregator* will require a Recorder-chaincode ID to publish the final credential proof.

The aggregator may exist for N levels depending on the Hierarchy of the data model and organization.

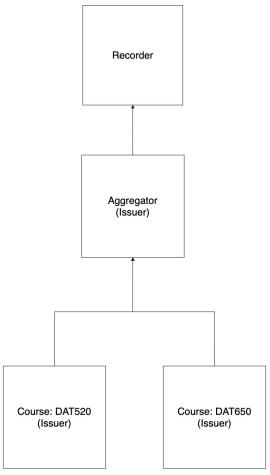


Figure 5.6: Simple structure of Chaincode Types.

Whenever a client initializes the *init* event, the Java code will attempt to understand the transaction context is wanted. We have implemented an abstract class known as a *Procedure*. A Procedure consists of two phases, namely *Validate* and *Apply*. Before creating the Procedure, the *Transaction Creator Hub* will parse the chaincodeStub parameters from the client such that the Procedure object can do a *get()* on those transactions parameters it expects. Essentially, the purpose of the Transaction Creator is to make the provided parameters easier accessible through methods when programming Procedures - and in the right type (string, integer, etc.). Rather than operating with a single list of parameters, which is how the Chaincode API injects the provided client Parameters. See Figure 5.7 to see the flow.

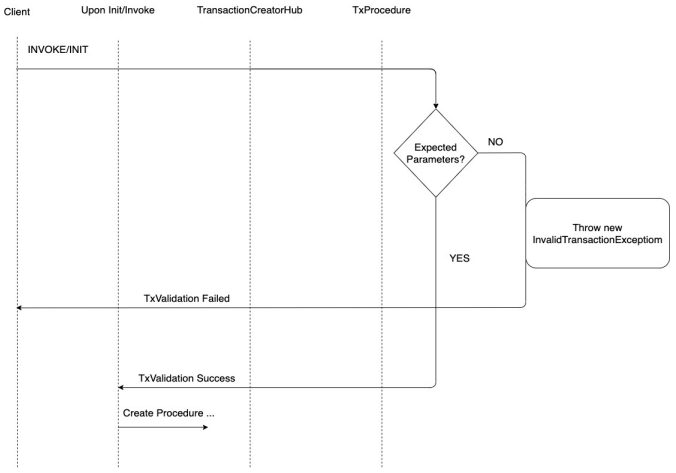


Figure 5.7: Transaction Creator Resolves Parameters which is persisted in a TxProcedure.

Upon finishing bundling the client parameters into the TransactionContext from the list returned by $Chaincode.getParameters()$, it will be injected into the corresponding

Procedure as shown in Figure 5.8.

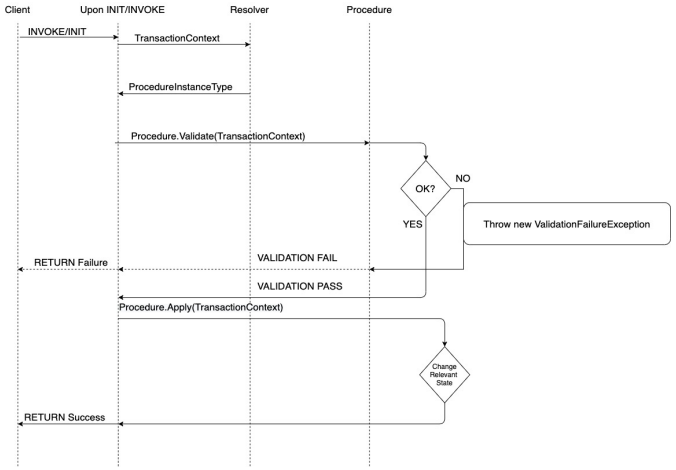


Figure 5.8: TransactionContext used inside Procedures.

The Validation phase goes through all the client parameters in the TransactionContext and ensures the following: the client is allowed to do the operation (ClientIdentity), the parameters provided are of right type, the operation brings to a legal state; we do so as the Chaincode API handles all client parameters as strings - which is not type-safe and to avoid non-determinismic data patterns. As a result, the structure needs to be enforced by some Java classes such that malformed documents cannot be submitted to the DB/Ledger. For each of the documents stored on the CouchDB we built a Data Access Object Class which all documents inherits. This abstracts away the need for how-to commit an object to the state database.

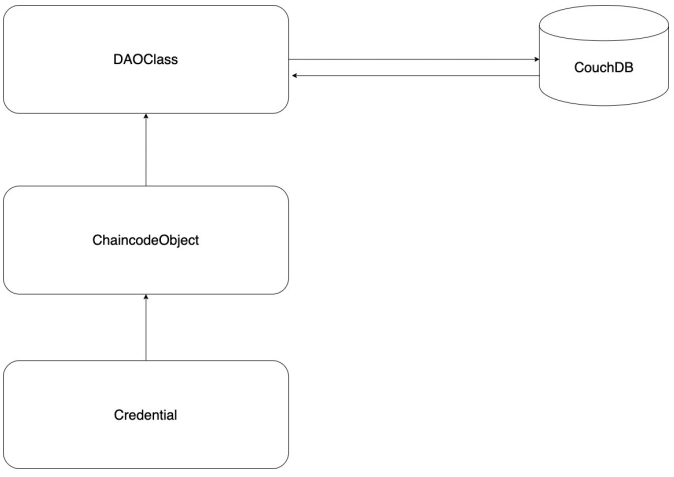


Figure 5.9: All Java Objects stored in CouchDB uses the DAO.

The DAO (fig. 5.9) translates the Java Object into a JSON format (which is what the CouchDB prefers). The idea of the DAO is that credentials is build up using *get*() and *set*() methods. Upon finishing the object.Commit() can be called. The committing

of objects is stored with <nameOfObject>.<Subject>.<issuer/recorder-Description>. Moreover, all the objects have fine-grained access control. This means only a subset of entities may view, edit, and commit changes to the corresponding ledger-object. This logic can be viewed as row-level security in Relational Database Systems. Furthermore, whenever a credential or credentialproof is finalized the transaction ID will be persisted to the object. This means that whenever the root-proof is finalized and the student has obtained the credential proof (in plaintext) is stored off-chain (in a local disk or DB) the client can then query the Hyperledger Fabric System with this transaction ID to get the block meta-data. This is a powerful way to check when a certain credential proof was committed to the ledger which will also give the verifier an idea of *when* a credentialproof was finalized.

5.2.3 Access Revocation and Purging Data

Following the Decentralized Identifiers definition there needs to be a possibility for the issuer to revoke a credential proof for cases where maybe a student has cheated the system. Therefore, the *Recorder* Chaincode will have the revocation method and not the Issuers. The Issuers will purge all the stored credentials in their corresponding data-store as these do not need to be retained for ever. This is done to reduce the storage consumption of the db instance (if we assume millions of users). We also assume that the final credential proof is replicated to an off-chain database, and only hash of the data remains in the ledger as an immutable evidence of the transaction. Thus, clear-text credentials/credentialproofs has has a limited lifespan, and will be automatically be purged after existing unchanged on the blockchain for a designated number of blocks (defined in the PDC). Hence why revocation is not implemented at the issuers and only at the recorder as information here will be persisted forever. Furthermore, as Hyperledger Fabric is a permissioned system external verifies must be granted access. Our idea here is that the Subject can request enrollment on behalf of a third-party. And when an admin allows for it, a temporary certificate is generated. Upon granting access, the third-party can then start to verify claims of a student. Either through pure hash-values provided by the student - or the student reveals his/her set of credentials to the thirdparty which can then assert the content of the credential proof as depicted in the Appendix listing B.1. Depending for how long the third-party needs access the lifespan of the certificate can be specified, or alternatively revoked based on the subject's request to the administrator of the CA. This way the Subject can control for how long the information should be read by external systems.

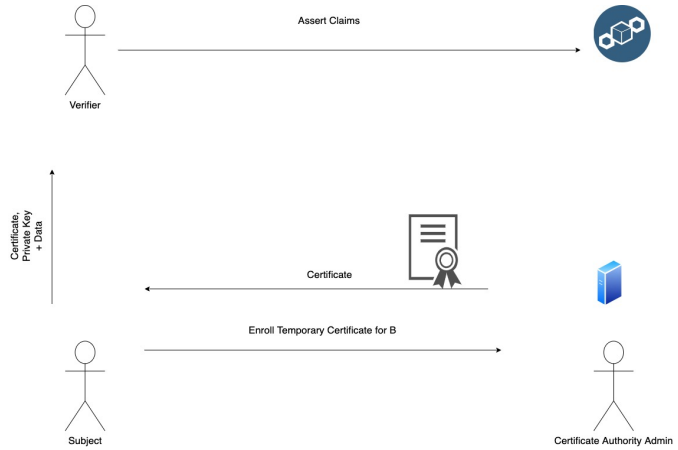


Figure 5.10: Subject may enroll temporary users for ledger access.

The *subject* for a university may enroll a temporary user. Upon granting the x509 Certificate and the corresponding secret key it can be forwarded to the verifier. The verifier may then invoke queries towards the relevant channels and organization identifier to assert some *claim* made by the student.

5.3 Architecture

Our Data model is also flexible in terms of how the operations/service team model the Fabric network. We propose that the different universities is abstracted into Availability Zones, e.g. North-, Middle-Europe, West-, and East-Europe. Each of these zones can be modeled as one channel or more. Moreover, we then propose that all universities also participates in a *global* channels where the *Recorder* instance will store the *Trie*.

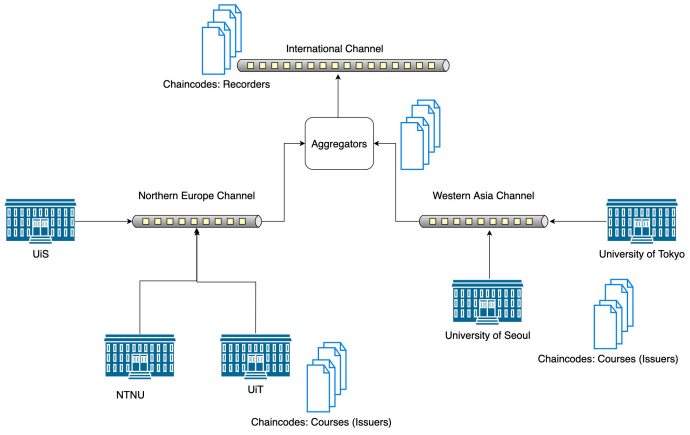


Figure 5.11: Multi-channel divided into availability Zones.

In fig. 5.11 all the aggregating chaincodes are located at the international channel. What we argue is a powerful utility here is that the aggregating chaincodes can pull data from

other availability zones. However, after tests we need to make the following assumptions for these patterns to work.

- A peer must have all the chaincodes it wants pull credential proofs from installed and instantiated.
- The peer node that has the aggregating chaincode must participate in the same channel as the other.

Furthermore, we also argue that there should be an international entity which governs the ordering service. More specifically, having multiple ordering nodes require more configuration of different nodes and only complicates the system set up. Hence, some trusted entity should manage the channel configurations in the different availability zones. We also want to see that the entity managing the ordering service also has root/intermediate Certificate Authorities that the universities need to obtain their trust relationship from. Furthermore, we also suggest that (if using cloud) to install the ordering nodes to different zones accross the world. Recall that for each channel the ledger and proposals are completely isolated - meaning that the availability zones will not be aware of the transactions that happens in the different channel configurations. The aggregator has the ability to be initialized with chaincodes resided in different channels through specifying *chaincodename:channelname* which in turn allows for viewing states between multiple channels, given that the assumptions in 5.3 are followed.

Chapter 6

Experimental Evaluation

This chapter will perform evaluations on the design previously described in Chapter 4. This aim is to experiment with the concepts used in the in the final design to see how Hyperledger Fabric behaves in different scenarios. As the framework is extremely policy-centric we aim to see what happens if unauthorized nodes attempts execute actions they're not allowed to. We'll also see how the proposed system works under a certain workload on a real cluster with N nodes. In the **test-setup** we utilize the PIVT project (with some modifications) in the BBChain cluster (N=30 hosts) to provision our Hyperledger Fabric nodes. Each organization is tied to a node, meaning that n peers, n couchdbs and 1 CA is collocated within a host (see Figure 6.1). Each of these hosts have 8 cores of CPU Power, 32 GB ram and more. Though they are powerful we cannot get a way of the Operating systems' context switching while running experiments. As a result, this might have impact to the final results we're collecting. Furthermore, we represent one client for each organization. We will refer to this as the client test-procedure throughout the text. This test-procedure is to register credentials, confirm them, aggregate them, verify and revoke. The clients are deployed as a maven test (for simplicity) running with the Hyperledger fabric SDK that deploys many requests with a thread-pool of significant size to attempt to mock concurrent users doing operations towards the system. While the client is running we utilize a service called Prometheus to scrape endpoints on all nodes in network to gather data from the hyperledger fabric cluster. These data are further sent through a push-gateway to a service called Grafana to display data with more advanced queries.

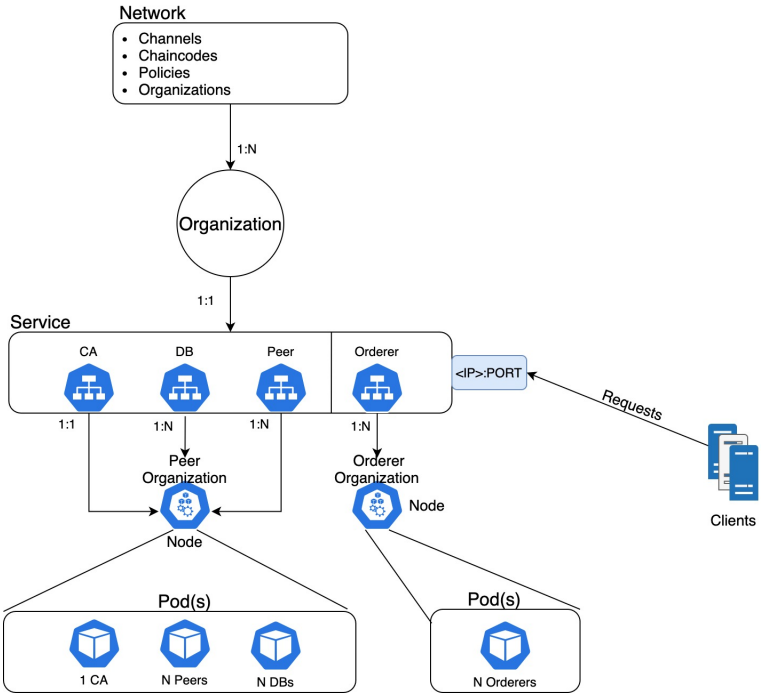


Figure 6.1: Kubernetes setup on the cluster. Collocating each organization to one host.

6.1 Same chaincode but with different Collections definition

The collections policies are defined per chaincode during instantiation and is key for keeping data private within the organization. It is therefore important in this design that the chaincodes which interacts with each other that the policies are consistent and that data is kept confidential. However, how does HLF cope with information leakage as described in the scenario below.

Scenario 6.1. Let's assume that three organizations are running an instance of the BBChain projects on the same channel. The organizations has the following running instances of chaincode with their respective collection policy on the same channel C $Chaincodes_{org1} = \{C_1, C_2, C_3\}$, $Chaincodes_{org2} = \{C_4, C_5, C_6\}$ and $Chaincodes_{org3} = \{C_7\}$. All of these Chaincodes contain sensitive information about the user's grade in the private data collection. Let's assume that the admin of Organization 3 is malicious and wants to initiate chaincode instances of $Chaincodes_{org1}$ s.t. Private Data Collection $PDC_{org3} = Chaincodes_{org3} \cup Chaincode_{org1}$.

With the scenario given above we wanted to see how Hyperledger would deal with instantiation of the same chaincode instance on the channel such that the unauthorized organization would gain access to the private data collection of the other organization, potentially getting access to the data through the gossiping in between the peers. The

results showed that this wasn't possible for the malicious organization as it would effectively break the Policy definition of the collection resulting in an endorsement failure when organization 3 attempted to synchronize their peers with the transaction history. Furthermore, when the client gets malicious and forwards the transaction to peers which are committers (and not part of the collection) but have the chaincode installed on it. Consequently the committing peers discovers that they haven't instantiated the chaincode. Upon the discovery the peers will issue a new container to build and run the chaincode, the installation finishes successfully though at the moment they attempt to perform the proposal they'll immediately fail the transaction.

6.2 Inconsistency in the Chaincode

Inconsistencies is dangerous for Distributed Ledger technologies and must be avoided at all costs to keep the integrity of the data stored. For HLF fabric we tested the network by manipulating the ledger resulting in a disagreement amongst the participants of the network. To do so, we altered the physical ledger through SSH into some peer container and change the underlying ledger of the channel. A common way of handling inconsistencies in Distributed Systems is to force the right order of transactions on those nodes that doesn't agree on the current world state, especially if there is majority of nodes agreeing on each other.

Scenario 6.2. Assume a network with N = 6 peers running on the same channel and all peers are endorsing nodes. The current state of the Ledger $L_{all} = [V_1, V_2, V_3, V_4]$. Then we assume that an attacker changes the ledger at peer P_0 s.t. $L_{p0} \neq L_{all}$. Consequently creating an inconsistency and disagreement whenever proposal are created. After doing so, we initiated a transaction proposal on the channel against a chaincode the attacked peer endorsed for. The system handled the inconsistency by terminating the running container of underlying chaincode. Consequently, the container went into exited state and required attention from the System Administrator through rebuilding the container and restart to retrieve the latest transaction history. This seemed somewhat like a limitation in terms of handling scenarios like this.

6.3 Java Chaincode Concurrency

In the initial testing of the performance of Hyperledger it was discovered that a high percentage of concurrently sent requests was rejected by the chaincode container, meaning

that multiple users will be denied service. From a security point of view this is a vulnerability that could be exploited. As an example, let's say that an external user invoked the some operation multiple times using concurrent pattern. Then, other users of the system might risk in getting denied service which could potentially stop the system from working as expected.

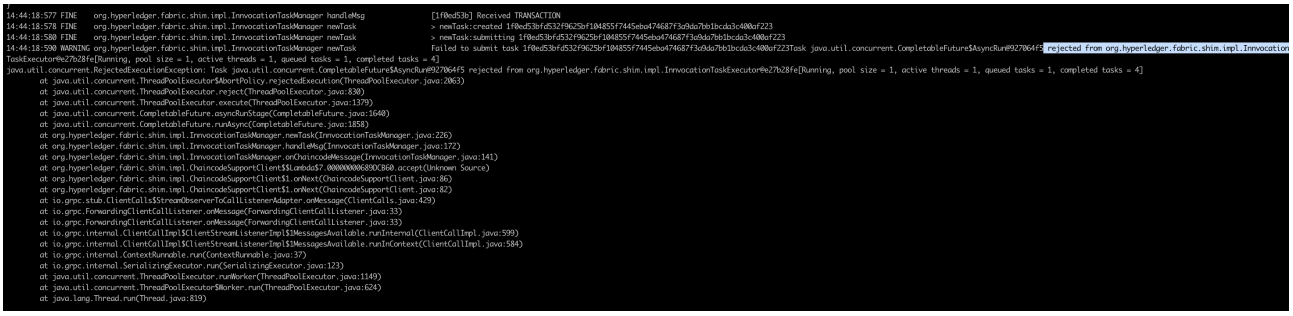


Figure 6.2: Transactions are rejected by the chaincode as a result of the pool size = 1, active threads = 1

As depicted in the figure above. The Java Chaincode runs on one single thread. Meaning that whenever there was someone else that invoked the chaincode it will collide with the current thread resulting in a *RejectedExecutionException*. The error was found in the message handler of the peer which makes sense why the concurrency wasn't handled well, see figure below. A proposed solution here is that the thread-pool should be adjusted according to the number of invocations. Rather than relying on a static thread-pool one can implement a dynamic way of setting this s.t. the needed resources are allocated to those containers which requires so. This would indeed give a better resource usage but also a better way of handling high concurrency situations.

```
public InnvocationTaskManager(ChaincodeBase chaincode, ChaincodeID chaincodeId) {
    this.chaincode = chaincode;
    this.chaincodeId = chaincodeId;

    // setup the thread pool here
    Properties props = chaincode.getChaincodeConfig();
    queueSize = Integer.parseInt((String) props.getOrDefault( key: "TP_QUEUE_SIZE",  defaultValue: "1"));
    maximumPoolSize = Integer.parseInt((String) props.getOrDefault( key: "TP_MAX_POOL_SIZE",  defaultValue: "1"));
    corePoolSize = Integer.parseInt((String) props.getOrDefault( key: "TP_CORE_POOL_SIZE",  defaultValue: "1"));
    keepAliveTime = Long.parseLong((String) props.getOrDefault( key: "TP_KEEP_ALIVE_MS",  defaultValue: "5000"));

    workQueue = new LinkedBlockingQueue<Runnable>(queueSize);

    logger.info(() -> "Max Pool Size" + maximumPoolSize);

    taskService = new InnvocationTaskExecutor(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
            threadFactory, handler);

    Metrics.getProvider().setTaskMetricsCollector(taskService);
```

Figure 6.3: Hard coding of threading parameters

6.4 What is sent to the orderers and other Peers during Proposal Phase when using PDC?

Even though Private Data Collections were utilized there was also raised questions whether the Proposal values that was sent to the orderers in clear text or not. To test for this, we utilized the HLF client API which has features for interacting with the blockchain. E.g. before the proposal are forwarded to the ordering services we can review the Read/Write-set that the client has received in the response. If data was visible in this part of the application then the purpose of the Private Data collection would be gone as information would have been disclosed.

The R/W-set is generated by the endorser and used by the committers to validate the transaction(s) The Read Set contains the values for a certain key K_{read} and versioning of the chaincode object before a commit whereas the write set will have $K_{read_{updated}}$ with the newest committed value without versioning. In general this can be viewed as the Simulation response created by the chaincode-executing peers (Note that this is what HLF utilizes in their MVCC mechanism as described in the scenario described in Chapter 2. 2.2). And will as a result of this be forwarded to the committing Anchor peers in each organization from the orderer to validated whether the update was valid or not. Tests indicates that all operations (read + writes) done in the private data collections are never forwarded in the read/write sets. However, all state-changes done to non-private collections are disseminated to all participating peers on the channel.

```
ns_rwset {
  namespace: "DAT520"
  rwset: "\n\024\n\016IssuerInstance\022\002\b\b\n\017\n\ttype_self\022\002\b\004"
  collection_hashed_rwset {
    collection_name: "issuerCollections"
    hashed_rwset: "\n\"\n \221S\305\375\333=\002\372\376K\256\365EX\201\313\271\224\000M\264Mb\3
    pvt_rwset_hash: "\254\372\276\214\206,\345\220,\v\320\205\004\220e\231\377OOL~\006j\201\020i
  }
}
```

Listing 6.1: read write operations on PDC are only forwarding hashed values of transaction results.

As a result, all other peers will commit this hash-value to their ledger and will act as a proof and a timestamp of some committed asset.

6.5 Adjusting the block parameters

Performance of the system could be adjusted according to block parameters such as batch size, block size, timeout before creating a block, and maximum transaction count per

block. We utilized the cluster resources to optimize for the best throughput and latency combination. To measure the impact of the reconfiguration we look at the number of transactions inside the ledger. Then we look at the ordering service to see how long it will take to process a proposal as this will indeed impact the response time for replying with an acknowledgement back to the client nodes. The table under shows different scenarios we've tested. We vary the batch timeout, transaction count and block size to get an idea of which of these will influence latency/throughput the most.

Scenario 6.3. We are assuming two of our test-clients in their respective organizations that fires transactions towards the cluster with a "Fire-and-forget" approach. Meaning they're not waiting for a confirmation. The approach here is to pack as many requests asynchronously within one time unit rather than focusing on a Poisson arrival rate. Metrics were then collected with Grafana. We measure the rate of transactions put in the ledger and compare it to the average time the orderer spend to complete a transaction proposal for four different chaincodes as "Issuer" types.

Block Parameters						
Case (ID)	**Timeout**	**Message Count**	**Max Bytes**	**Preferred Max Bytes**	**Throughput (Tx/sec)**	**Proposal latency (seconds)**
(1)	2 seconds	10	99 MB	512 KB	116	4.71
(2)	2 seconds	30	99 MB	512 KB	153	6.65
(3)	2 seconds	50	99 MB	4048KB	125	4.8
(4)	0.5 second	100	99 MB	512 KB	210	9.1
(5)	0.5 second	120	99MB	512 KB	190	8.7

Table 6.1: Adjusting block parameters scenarios.

The *block size* should be set to high if there is a constant flow of many transactions per seconds or if transactions are in general large, such that as many data points are packed into a block. The *Transaction count* should be set to high/low depending on the amount of requests client application(s) is producing per second. Whereas the *batch timeout* should be set depending on the latency of the transactions. Following equation can be used to get an idea of what the Timeout parameters should be set to.

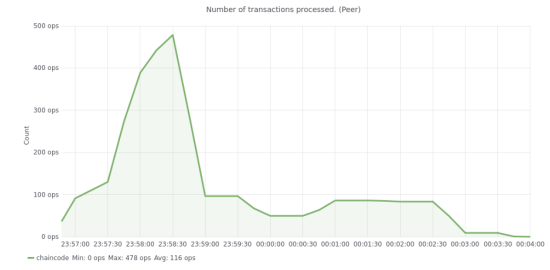
$$BatchTimeout = TS(Tx_0) + avg(T_{tx_{RTT}}) \tag{6.1}$$

Where TimeStamp(Tx_0) TS denotes the timestamp of the first transaction received and $T_{tx_{RTT}}$ is defined by the round-trip from issuing a transaction to the point it fulfils the

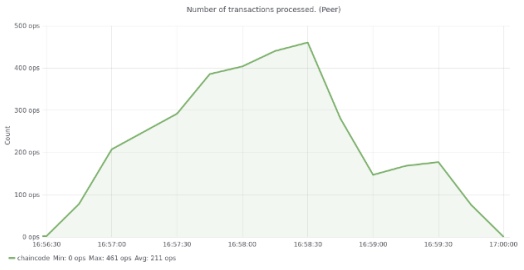
endorsement policy.

$$\begin{aligned} T_{tx_{RTT}} &= T_{endorsement} + T_{networkLatency} \quad where \\ T_{endorsement} &= T(Peers_{policy}) + T_{clientResponse} \end{aligned} \tag{6.2}$$

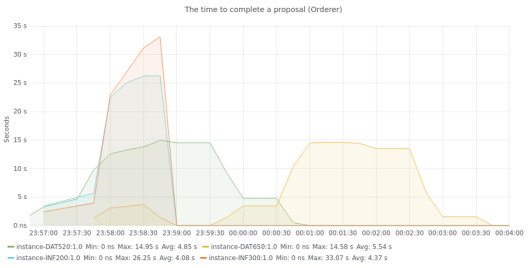
The time needed for endorsements is varying because $T(Peer_{policy})$ will be dependent on the endorsement definition of the respective chaincode, however, as a general rule, more complex policies such as 'AND()' and 'N-OutOf' policies are likely to spend longer time than 'OR()' policies to be processed. Furthermore, from equation 6.2 we can deduce that for blockchain networks that are distributed across the world might require $T_{networkLatency} >> T_{avg}$ meaning that the BatchTimeout should be adjusted accordingly. Furthermore, as our clients was on the same network the round-trip-time for forwarding endorsements, collect them, and send it to the ordering service was low, i.e. we assume a low $T_{networkLatency}$ value, we also utilize 'OR()' policies which means that the $T_{endorsements}$ time value was also low. As a result, the batch timing was set to 0.5 seconds. As a result, the block optimization can be done by adjusting the number of transactions and batch size (in bytes) in order to optimize the throughput.



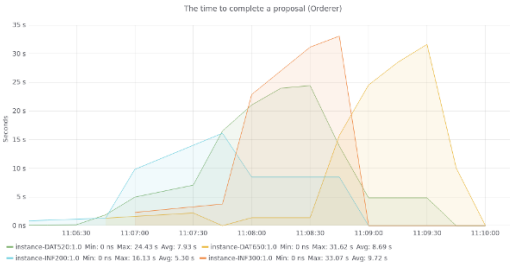
(a) 10 Transactions per block and 512KB block size



(b) 160 Transactions per block and 512KB block size



(c) 10 Transactions per block and 512KB block Proposal time



(d) 160 Transactions per block and 512KB orderer Proposal time

Figure 6.4: Comparison in throughput between the initial (a) setting and the new setting in (b)

By taking base in scenario 5 from table 6.1 we further adjusted the transaction count to 160 transactions per block which yielded a result of 211 transactions per second in the ledger during the client test-procedure with an average proposal time of 7.91 compared

to the Hyperledger Fabrics' standard setting which had a proposal time of 4.71 seconds. From our observations we have defined three cases:

- **Case 1.** By setting the number of transactions in one block to be around the expected transaction arrival rate gives a significant increase in throughput. Although, from our observations the time needed for completing the proposal at the ordering service increased compared to smaller sized blocks.
- **Case 2.** Setting a blocksize larger than the actual transaction arrival rate increases the time for getting confirmation of a block commit. Meaning the block-cutting parameters are not met which results in the orderer to hold the transaction for a longer time before issuing a new block to the peers. No significant change in throughput on the ledger.
- **Case 3.** Small amount of transactions per block decreases the time needed for completing the proposal at the ordering service. However, at a cost of the throughput.

Based on our findings we therefore guideline that applications which requires low latency, lower block size should be used, whereas for networks which require a high throughput and doesn't focus on latency should apply for larger blocks. For applications that require something in the middle should model their block size to be somewhere near the average transaction arrival rate.
Rather than statically settings these parameters is to let the ordering service optimize these during run-time. Some parameters to optimize for can be the following:

- Latency of clients and validation the Endorsement policy - batch time out.
- Average document size in transactions, e.g. the payload - preferred max bytes.
- Expected transactions per time unit - transaction count

At the moment these parameters requires a reconfiguration of the ordering nodes - meaning that for an existing system one would need acquire the configuration block from the ordering services and update it which needs to be done by the system administrator of the relevant ordering services. Then the administrator will have to conduct tests for which set of parameters that will influence the transaction throughput the most. Note that as the network grows - so does the number of transactions that can be processed. Which means that this should be evaluated upon reconfiguration of the view in the network.

6.6 Performance Metrics Chaincode

6.6.1 Endorsement Policies

The endorsement policies can as previously described be defined as 'AND()' or 'OR()' policies, stating the relationship between nodes to endorse a transaction. The 'OR' policies only requires a single peer to simulate the transaction proposal, whereas the 'AND()' will require that all the peers defined in the policy must execute and sign the transactions. There are two trade-offs here. First one being that more peers requires to execute a transaction, meaning that higher proposals time are expected as more nodes need to receive the request, perform it, and return to the client. Furthermore, the verification process of the transaction will require more time, as more signatures must be validated before including the transaction. Using the 'OR' policy, the client can load balance the traffic to its peering nodes, meaning that less of them need to be occupied for performing duplicate transactions. On the other hand, In the case of the 'AND' policies one would indeed waste the computing power of the peers as duplicate operations need to be performed in the network.

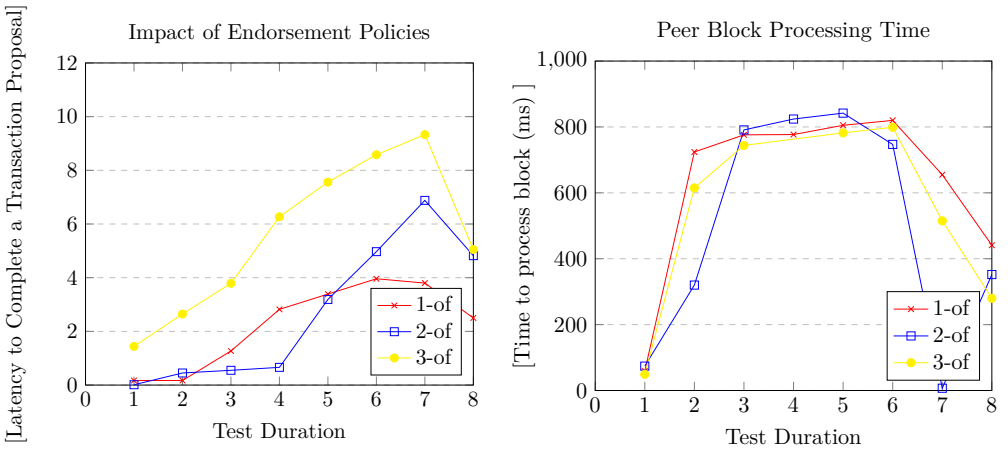
The test-case uses the test-procedure as previously described in the beginning of the chapter. In this section we show how the endorsement policies affects the throughput of the system. We first show a 'OR()' endorsement policy for our chaincodes, meaning that we only require one peer to execute the transaction. We utilized a load-balancing technique such as the following:

Algorithm 1: Simple Load Balancer

```
Config := init(NetworkConfig);
function handleRequest(Request, EventData):
    forwardToList := List();
    foreach org ⊂ EventData.org do
        node := RandomPickOrgNode(org, Request.getTransactionType());
        forwardToList.append(node) ;
    end
    return forwardTransaction(forwardToList, EventData);
```

Our test-clients utilized this approach for forwarding their requests to the network. The *RandomPickOrgNode* algorithm picks a random orderer or peer based on the transaction type. If the transaction needs to forward the request to multiple organizations then the algorithm will keep adding to the list of receivers of the transaction (picking one from each). However, if using cloud architecture we propose to create a peer/orderer load-balancer that balances requests towards the orderers and peers based on how occupied

the nodes are (in terms of pending proposals). The random picking of nodes to process transactions are seen as an optimization, at the moment the orderer produces a block, then all peers will finally get the changes done at the initial peer and reflect it through their world state.



(a) Time spent processing an endorsement at the ordering service.

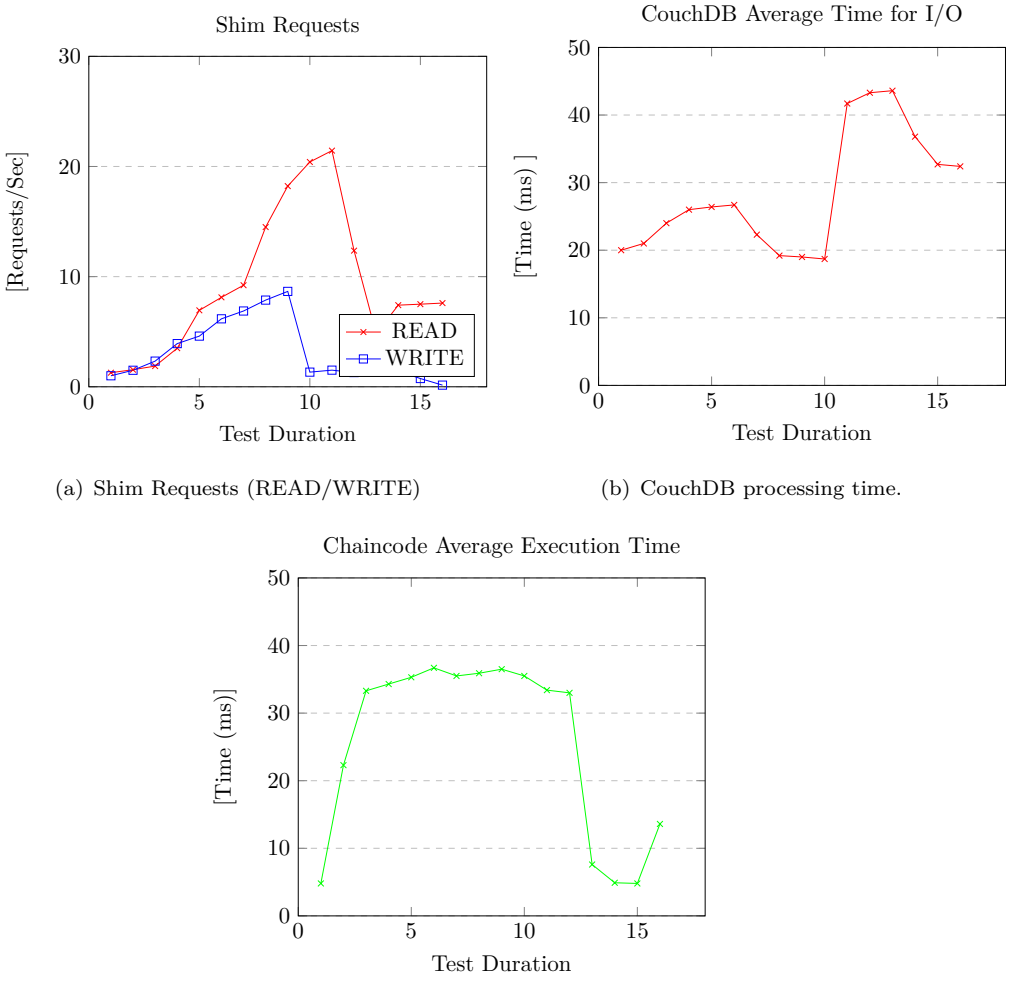
(b) Peer block processing time.

Figure 6.5: Ordering service and peer statistics.

As depicted in fig. 6.5 - the more restrictive policies for chaincodes the more time is required for the ordering service to process transactions. Recall that the validation of transactions includes: inspecting that all endorsing peers have consistent results, and all signatures are valid; As transactions takes longer to process for the transaction queue for the the ordering service will increase as more proposals arrives. Consequently clients will have to wait for longer time in order to wait for a block event. Moreover, we also observe that time needed for processing the block at the peer node does not show any significant impact as a result of the endorsement policies. Based on this we see that the bottleneck resides at the ordering service whenever transactions increase in terms of required endorsing nodes of the network.
One could measure the impact in terms of throughput of the system, however, with the metrics exposed by the Prometheus this was not be possible as the ledger transaction count is reflecting the number of transaction processed once the block is received - meaning that the block itself is delayed and the number of transactions put on the ledger would likely be equal anyway. We argue that in this case the bottleneck is not located at the peers, but at the ordering service as the time-delay for a block event is increased.

The limitation is therefore on the metrics Fabric exposes.
Furthermore, to get an idea of what the limited factor of Chaincode execution at the peers we collected statistics of other aspects such as gossiping of private blocks, CouchDB data,

and *Shim* requests. Essentially, shim is the underlying API package chaincode utilizes to connect to the world-state database. Every write and read done by the chaincode is counted in the test.



(a) Shim Requests (READ/WRITE)

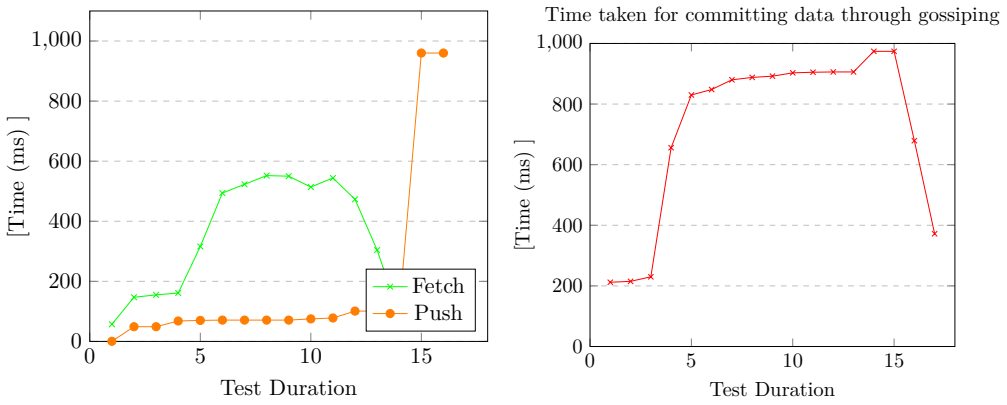
(b) CouchDB processing time.

(c) Average Time to Complete Shim (Chaincode) Request.

Figure 6.6: Chaincode related statistics.

From observations in fig. 6.6 we see that the CouchDB instance spends more time in executing its operations as the number of requests that the peer receives. However, we have reasons to believe that this effect amplifies as all peers for one organization is collocated at the same node in the cluster. Nevertheless, the amount of I/O-operations will directly impact the average time to execute a chaincode invocation. The Average Shim Request Time is lower than the DB-request processing time, which is explained that not all shim package requests interact directly with a database. Furthermore, clients waiting for block events will need to wait for $T_{wait} = T_{ChaincodeTimeExecution} +$

$T_{EndorsementValidation} + T_{BlockValidation}$. To measure the $T_{BlockValidation}$ we measured the time needed to validate private data blocks and how the gossiping impacts this.



(a) Private Data Dissemination Duration

(b) Peer block processing time.

Figure 6.7: Times for Committing private data.

As shown in fig. 6.7 we see that there is a strong correlation between commit time of a private block and the time needed for a peer to fetch it. PDCs are only gossiped once a peer finds out that its stored collection is outdated. Upon finding out, it will request a fetch of the most updated private collection. The *fetch time* directly impacts the *time taken to commit a private data block*. Which means that this will significant increase T_{Wait}. Nevertheless, the spike of pushing data in fig. 6.7 (a) is due to the pushing to the amount of *required peers* that need to hold the private data (defined in the policy).
As a result, our findings on chaincode metrics we propose a following guideline for future development of chaincode.

- Restrict the amount of I/O operations to state database in chaincode as this would increase the amount of *shim* requests towards the CouchDB. The higher amount of DB requests per invocation, the lesser the throughput, and higher block latency (see T_{wait}). Furthermore, concurrent requests gives higher average time for the CouchDB to finish its operations.
- Advanced endorsement policies significantly increases the wait time for clients to get confirmation that their transaction is included to a block.
- The gossiping of private data increases the time needed for peers to get a fresh *view* of some private state. Which directly impacts the chaincode execution time.

6.7 Failing Transactions

Hyperledger is complex in terms of communication between the different nodes in the network. However, this opens for possibilities to attack different kind of applications to deny service. In this section we will discuss different attack vectors that may lead to an exploit. The first one to discuss is the client SDK from Hyperledger. The client is the "brain" which drives the consensus and appending of new blocks. Recall the transaction lifecycle, a client must initiate the transaction, collect the required endorsements from the respective peers, and then forward the transaction to the ordering service. A possible way of denying the service here would be to perform a sybil attack in terms of exhausting the network interface of the client, meaning that the client won't be able to forward the ProposalResponses from peers to the orderers resulting in peers that will execute the simulation without achieving a block for it. Furthermore, if the chaincode updates ledger objects with the same key multiple times then the MVCC will kick in and fail the transaction, meaning that no peers will be able to commit their simulation proposal and no blocks will be produced. If the attacker knows any keys in the system that is frequently updated then that person can exhaust the system. The MVCC attack can be formulated as the following, let's assume a set of transactions included in a block proposal.

Scenario 6.4. $Worldstate$: (Credential1, Value : B, v1), (Credential2, Value: C, v1), (Credential3, Value: E, v1) (Proof1, value: Approved, v1)
T1 -> $write(credential1, v1', valueB)$, $write(credential2, valueA, v1')$
T2 -> $read(credential3, v1)$, $write(proof1, NotApproved, v1')$
T3 -> $read(credential1, v1, B)$, $write(credentual1, v1', D)$

Note that this scenario is an example of Definition 2.12. Before creation of the block, the MVCC will validate the read-write set to determine whether any reads/writes are happening on an updated version of the object. In this case T3 won't pass the validation, as a result of the write from the first transaction. Transactions one and two will pass the validation as there are no conflicts in between the keys updated so they are inlcuded in the block. However, Transaction T3 will be discarded resulting in a failure of T3 that will be denied by all the relevant peers in the cluster. This was discovered during development as some of the chaincode updated the same key several places within the *batchtimeout* of the block creation. We therefore propose that all keys should be unique and non-overlapping to avoid these kinds of problems. Furthermore, this attack is indeed possible for our application as the key creation is deterministic and related to the user. By knowing the issuer description and user name an attacker (that has the authorization to do it) might exhaust fake updates towards the system when updating credentials and proofs occur. As a result, we might risk in getting a denial of service attack as the MVCC

will either deny the attacker or the honest user of the network. It is therefore **important** block creation time is not set to too high as this would increase the likelihood of MVCC READ/WRITE failures. Adjusting parameters previously discussed is therefore vital to keep as low as possible. Furthermore, if we assume a man in the middle attack and that TLS is enabled, the MITM can malform the request to deny the client in forwarding its proposals. If this happens on the intra-net of the network where also the blockchain runs, it could potentially deny all known nodes to extend the ledger. The same type of attack is possible in the permissionless world, however, as any client can join the effectiveness of these types of attacks are debatable as users can be scattered around the world and not in a "centralized" manner as Hyperledger tends to do as it requires to know its peers. Furthermore, another limitation that can lead to a potential denial of service is to exhaust the leading peer of the business network. The immediate effect of this is that this will strict the block propagation towards the rest of the business network meaning that the MVCC could further be attacked as there will be no new commits unless a new leader is elected and not denied service.

6.8 Setting up Fabric In Azure

After using the Helm Charts and Argo we wanted to see how it was like to deploy a Fabric Network to a cloud-service such as *Azure*. We chose Azure as the provides free-credits for students at the University Of Stavanger. Then we utilized the *Kubernetes Services* [66] [67] - when setting up the cluster the Cloud Developer may chose to use multiple primary/master Kubernetes nodes. Though, for the sake of this project we chose one primary node. Then we allocated 10 nodes for our cluster. Once ready a primary node can be used to configure the cluster itself. Before starting the cluster the required tools such as Helm, Argo and other dependencies PIVT requires must be installed. Running the relevant commands for provisiong a fabric network using PIVT is the same as with the cluster at the universities - and requires no changes to it. Whenever this is done, and the project is up and running one must expose the cluster for applications. This is where the load-balancers comes to play.

```
apiVersion: v1
kind: Service
metadata:
  name: hlf-peer-external--uis--peer0-0
  labels:
    name: hlf-peer-external--uis--peer0-0
    component: hlf-peer
    fqdn: peer0.uis.no
    addToExternalHostAliases: "true"
```

```
spec:
  type: LoadBalancer
  selector:
    name: hlf-peer
    app: hlf-peer--uis--peer0-0
  ports:
  - protocol: TCP
    port: 7051
    name: grpc
---
```

Listing 6.2: Create a load balancer for a peer.

The load-balancer above will expose a public IP together with a port that can be used by the clients to talk to a specific peer in the cloud (here the peer0 for UiS). The loadbalancer will listen for grpc calls from the external applications. The provisioning of the load-balancer is done by Azure and may take some time. Therefore we can conclude that PIVT brings the scalability of configuring and provisioning fabric networks across platform really easy - as long as the platform has Kubernetes installed on it. Furthermore, we did not test our implementation here as Trial-accounts are restricted in terms of resources, e.g., 1 core CPU per node.

6.8.1 Deploying Fabric Network to different Availability Zones

Even though the PIVT project works nicely with a single Kubernetes cluster, we wanted to attempt to configure a truly distributed system by having two independent Kubernetes cluster as this would also reflect a truly distributed system. Moreover, in a multi-tenancy network, different nodes will likely be running as bare-metal nodes, docker containers, or located in a Kubernetes cluster distributed around the world.

Scenario 6.5. Assume a network where a university in North America (NA) wants to deploy a Fabric network to distribute and create credentials. Furthermore, a university in Asia would like to interact with this network. They both deploy their Kubernetes cluster using the PIVT project but need to establish a connection for communication. The university in NA decides that the initial ordering service will be provisioned in their cluster. For example 6.5 to work we replicated our initial Azure Kubernetes Service but provisioned in the West-Asia (WA) region. Though this time, for simplicity, we configured a single ordering service, single peer for three other organizations. In the NA region, we deploy one ordering service and peer organization, and we deployed two peer

organizations for the WA region. To allow for cross-cluster communication following needs to be done.

1. Create crypto-configuration for each organization.
2. For each communicating organization, copy the individual MSP file from both relevant peer and ordering organizations into the corresponding crypto-config file such that all nodes in the network will have the *same* $crypto-config/peerOrganization/MSP$ and $crypto-config/ordererOrganization/MSP$ file structure.
3. Create LoadBalancers/Ingress for all peers and ordering nodes.
4. In Kubernetes, launch the network without enabling pods - only the services that routes traffic to corresponding *docker containers* are enabled such that IP addresses can be collected.
5. Collect both internal and public (LoadBalancers and Ingresses) IP addresses.
6. Include all IP addresses into the Kubernetes resource files, more specifically to the Stateful-sets of both peer and ordering resources.
7. Upgrade the network using helm to initialize pods.

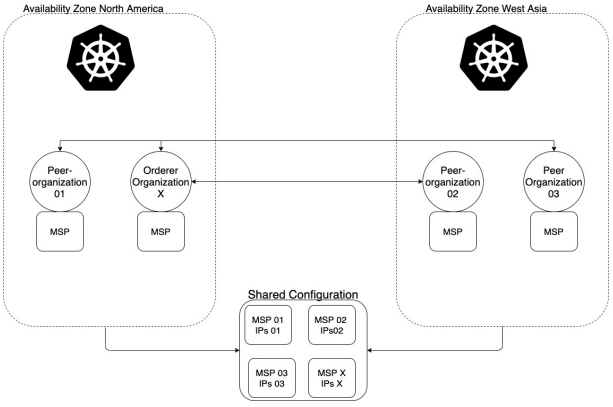


Figure 6.8: Overview of Configuration

This process does work for small clusters. Though, the scalability and maintainability of configuring larger systems are questionable. The amount of copying required is basically N where N is the amount of participating organizations. One will have to manually provide all the allowed public keys (from the MSP-file) to make both the peer and ordering nodes from NA- and WA-regions to allow for interaction in between them. An idea here is to utilize Ethereum's smart contract, where organizations can agree on which public keys (or x.509) certificates to trust. Then, upon startup of each node, the system administrators of the different organization(s) can build an interface that allows for

building up these MSP files dynamically based on a shared trusted resource - namely the public blockchain. Furthermore, the IP-addresses will be in the allowed-list of the smart contract such that these will be injected in the resource files upon startup of containers within the Kubernetes Cluster. The process we tried in Azure will require that for each time the *Kubernetes Services* are restarted, as new IP addresses will be allocated upon provisioning of new Kubernetes clusters.

Chapter 7

Lessons Learned

This chapter will provide some insights on how the experience with developing with Hyperledger was like. This involves everything from running a development environment, debugging, creating chaincode and other experiences such as tracing errors produced in the framework.

7.1 Golang vs. Java Chaincode

Hyperledger Fabric supports Javascript, Java and Golang to define the business process in chaincode. In this project it was an attempt to write in Golang and Java.
In terms of readability and compact code, Golang is superior to Java. However, in terms of unit-testing the code Java is stronger. The reason for this is that Hyperledger Fabric mocking framework has limited set of capabilities. It was discovered while writing the code that the mockstub object didn't support certificates while testing. This imposes a major issue in order to test application logic, especially in this project where most of the operations done in the chaincode requires identification of the users. In contrast, the same applies for the Java version, however, testing frameworks such as jUnit and mockito allows for easily mocking of real java objects. Making it easier to mock up HyperLedger Fabric behavior. However, in terms of developing with both the Languages it's up to the application developer to choose which language to go for. They both have the same API-definition meaning that none of the language imposes loss of application behavior.

When a peer in a channel instantiates the Java Chaincode, a new container is created by the peer to generate the jar-artifacts required for the chaincode, in this project this node will use maven and pull the required dependencies in order to build the main class. A important note here is that it is only possible to deploy/instantiate one chaincode at

the time. This means that for each chaincode there must exists a new project. This is the reason why the "ChaincodeInstance.java" was developed to be "one instance multiple types" as it would lower the need of installing multiple chaincodes on the different peers. Furthermore, using Java also increases the build time of the chaincode as a separate container is created for building the code within the Java Environment, furthermore this container allocates 1.8GB of disk space. As a result, the build process of Java chaincode does bring more overhead compared to GO.

7.2 Client Software Development Kit

The SDK provided by Fabric is available for Java, Go, NodeJS and Python. Even though there exists wide support for different languages we found it very hard to use them as there was lack of documentation and best-practices. The only way to grasp how to use these SDKs is to scan through many unit-tests written by different developers. Moreover, like the Python and JAVA SDK does not have a sufficient documentation in terms of examples compared to the NodeJS. As a result, the developer will have to attempt to translate the NodeJS examples over to the other languages. However, we find it strange that the naming conventions for all SDKs are different. E.g. for nodeJS a 'gateway' object need to be created, whereas in the Java World this is abstracted as a channel. Not only a bad practice, but also confusing for the application developer in the case where projects requires integration from two or more worlds.

7.3 Hyperledger Fabric Configuration Errors

What makes it hard and time consuming while developing with Hyperledger fabric is that some of the environment variables in the peers/orderers/cas could be wrongly set which results in a container that silently fails. The same applies if there is a small error within the configtx.yaml and crypto-config. As an example, providing channel names with capital letters will make the initialization of the channel to fail, however, no clear "ERROR" message are given and only a print statements are given somewhere in the log, making them hard to detect. This made the process of configuring the containers and the network time-consuming as a lot of investigation are required to find out what actually went wrong. The Docs of Hyperledger is helpful for understanding the theory, however, possible configuration parameters and other "tips and tricks" are scattered around in files within their git repository. This made it hard to grasp what the "best practices" are when setting up the fabric network, resulting in a lot of back and forth and usage of communities for support. The "Bringing Your first Network up" example were helpful and

is the example they refer the most to in the documentation. However, there is no clear example of how each component really works together. There are shell scripts which are approximately around 2000 lines + which sends commands to the docker cluster to make it ready for use. There are smaller examples to follow too, but they are essentially too small to even deploy a simple chaincode on to it. However, a fresh IT student wouldn't necessarily understand how the technology stack of creating a network would work making the process of understanding the core concepts of how to program and setting up the local environment. Also, the fabric-tools may change for each major version. Even though these tools change there is nothing that performs integrity checks in the hyperledger containers to ensure that the configuration is consistent in terms of versioning. The direct consequences here is that the network will produce errors that aren't related to the real problem (which may be the versioning). Throughout this project there was spent significant time on ensuring whether it was a configuration issue, versioning problems or environment issues. In regards to the environment issues we mean that volumes, containers, and networks might be reused by the docker environment potentially leading to a inconsistencies in the configuration read by the peers, orderers, and CA's. A mix of all of these problems makes it really hard to grasp the core problem of why the network isn't functioning properly. The consequence is that it leads to a "trial and error" process which may cause changes that cause new issues which masks the old ones. As an example, it was discovered was that chaincode images which are dangling and might contain errors will be reused when instantiating the chaincode from a peer resulting in not building the newest code changes. A way they could have solved this is to create new images with a new hash together with a salt value resulting in not using older images to build new chaincode-instance containers rather than always assuming the same name for the dev-<chaincodename.version>-peer(n), this would always trigger a new docker image build as it wouldn't find an existing one already, which has the old chaincode within it. This means that whenever there is a change of a chaincode, one *must* ensure that the docker environent doesnt use the cached images on the host while upgrading/changing of code. It will therefore be expected that maintaining and developing with Hyperledger as one developer is time-consuming (especially in the beginning) as there is a lot of files to take care of that may contain configuration mistakes resulting in one or multiple problems.

7.4 Chaincode vs Solidity

Ethereum has a high-level Object Oriented programming language which allows for contract development. Syntax is similar to other known programming languages, such as Java, Javascript, Python, etc. This makes it easier for a developer to abstract everything

that has to do with the blockchain and instead focus on application logic. As a result of this, getting into developing with Ethereum is quick and easy. Ethereum's solidity is a coding language created for open blockchain systems. In the contrary, Hyperledger only provides an API through different languages. The most common API to use is the **ChaincodeStub** which has methods such as **Init** and **Invoke**. In the beginning, this feels extremely limited and simple. However, since the API is available in Java it allows for a greater amount of advanced operations in the code than Solidity. However, this comes with a price. In Ethereum, application developers have incentives for writing memory consuming code through the concept of Gas. The incentive approach is a good practice for creating an application that should run on an immutable ledger allowing for fewer programming mistakes. However, less advanced data structures are available, making the programs relatively simpler than what it did in chaincode.

The freedom was the immediate effect and gives the ability to use third party tools. As a result, the code was more prone to errors then experienced with Solidity. When it comes to application testing Solidity can use *Ganache* [68] as a underlying ledger running on the Host machine. With the help of the NPM and the *Truffle* framework [69] to create unit tests and quickly test the smart contract while deployed to an actual instance of the Ethereum Virtual Machine running on a local ledger or Continuous Integration (CI) Pipeline. For fabric, this would require to configure the docker-compose files for a CI pipeline to work or to run test against a physical Fabric system. The immediate downside of not having a simple testing VM like ethereum is that while developing with chaincode one must mock up a backing ledger class/object that runs simple unit tests. Furthermore, some methods are missing from the included testing framework in the chaincode-shim package, resulting in assumptions that might not hold for a real-world system. Alternatively, the developer may run a docker-compose fabric network - which requires more time to setup than Ganache and Truffle. However, with the help of the fabric-java-sdk it is possible to write tests that run on the ledger. But this requires an already working network with keys to run correctly. Furthermore, the test needs to be aware of the crypto-material such as admin keys/certificates addresses of nodes, etc. As a result of the topology change, the tests itself need to be changed too to recognize the topology. A possible improvement here is that Hyperledger could have provided an update where the SDK reads the configtx.yaml and the crypto-materials directories to load the configuration into the run-time environment of the test itself. If the SDK utilized these files, that would have made the development of chaincode much more manageable as less configuration is required. Another issue with the SDK is that it requires the developer to learn yet another paradigm within the environment, making it more complex to use/interact with the system. Solidity doesn't require such as all nodes running the ledger has the same code/role, making it much easier to develop and test with during the

development life-cycle. Furthermore, the technology stack in Hyperledger is significantly larger than Solidity, meaning that debugging and reading stack-traces becomes more time consuming. Furthermore, requiring to rebuild and re-instantiate the chaincode to a fresh docker environment is time consuming. Especially if the environment gets more advanced in terms of business complexity. The reason being that the context switching of the Operating System must happen more often as the the concurrent processes increases, leading to a throttled system. Ethereum can use tools like Ganache [68] which is managed by its own in terms of handling overheads (clearing volumes, starting fresh environments, inspect results) making the Software Development Cycle much quicker compared to chaincode. However, in the context of the BBChain project HLF does have an advantage in terms of having a data storage within its environment. For permissionless systems, like Libra and Ethereum they'll need their own distributed storage in order to store the meta-data on-chain. As of this date [1] they would need to develop a distributed storage to hold the data making the Hyperledger Fabric more mature for the real world in terms of testing.

7.5 Collections vs. Channels

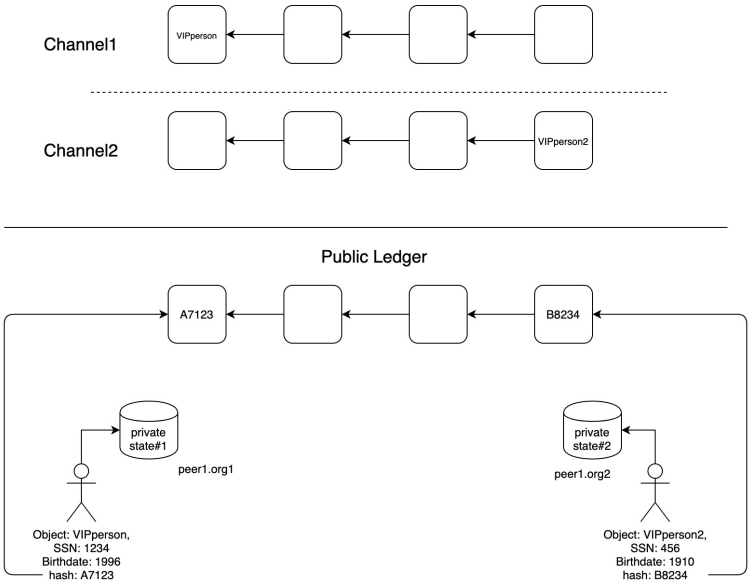


Figure 7.1: Channels vs Collections. Collections stores data on an authorized peer rather on the ledger. The hash is instead stored on the ledger.

From experience, evaluation, building, and operation Distributed Ledger solutions in an HLF environment (that scales) private data collections helped in scaling the operations a lot easier. Rather than defining policies for each channel, and for each participating

[1] 03.04.2020

organization, one can use collections. Multiple collections allow for one channel even though privacy is a requirement for assets - in this case *Credentials and CredentialProofs*. An important note here is that all proposals on the channel only contain hash values, and the nodes defined in the collection policy store the actual data itself. Essentially, collections allow for more fine-grained separation of organizations within one channel. Moreover, rather than having separate ledgers and potentially fewer peers to maintain that ledger, this allows for a multi-tenancy cooperation between organizations. As a consequence, the network also achieves a higher crash fault-tolerance (BFT in the future) as there are more participants on these types of channels rather than many small ones. Let's assume a scenario where there are many organizations on one channel, and two specific organizations want to create a shared state. Then, in the "classical" approach, one would require to update the network with a new channel which is specific to those two organizations. This specification would require to update the genesis block in the orderers to accept new channels and transactions within these. Afterward, one needs to install chaincode. In the "collections approach" one would still utilize the same channel but only install the required chaincode on the peers together with a collections policy. However, since they all participate in the consensus protocol, as a result it will affect the final throughput. It depends on the application that runs the HLF-framework. If we would assume an application that handles for an example credit card transactions, then channels would have been a better approach. The channels are better because one can use the concept of sharding to isolate the consensus epochs within each channel, ultimately increasing throughput. Even though transactions per time unit would increase, it will also have a significant impact on the required maintenance on the network, which would infer extra cost and time spent. Therefore there must be an evaluation between performance vs. costs - relative to the collections approach or even other projects such as Ethereum.

7.6 PIVT project: Chaincode and Helmcharts

Even though the PIVT project was helpful in reducing the amount of configuration needed we spent significant time debugging errors. Currently, Helm does not have any logs or transactions events which gives the developer insight in errors while provisioning resources. For our project, when integrating PIVT with our chaincode we got the error 'Helm relase not found' - which was not telling us anything about the error at all. Apparently, Helm Releases that creates Kubernetes ConfigMaps that exceeds 1MB makes the deployment to fail [70]. This issue was related to the size of the Chaincode included in the project structure folder consequently requiring refactoring of code to keep the chaincode files small in terms of size (less than a MB). Therefore, we suggest that

for larger projects that require larger chaincodes (in terms of size) to be written in Go as the language requires less code for the same logic with languages like Java - resulting in smaller files in general.

7.7 Other comments

During testing and development there was discovered that the default THREAD POOL size of the java chaincode container was set too low, where basic operations was time consuming. To resolve this issue we had to be add a resource directory in the same relative path as the main chaincode where the thread pool was adjusted up to 5. The size of this pool is naturally dependent on the application and what operations that are done. Having a too large number would result in competing between threads to allocate CPU and memory resources and if it was too small it could end up with a small throughput. As of [71] they recommend a thread pool size of $N_{CPU} + 1 threads$ whenever there is resource usage in terms of I/O operations and database connections. With the assumption that most modern computer systems today have a N_{CPU} of 4.

Chapter 8

Conclusion and Future Directions

8.1 Future Directions

We propose for future investigations on using Fabric as a distributed authentication scheme. This allows for a multi-organization way of synchronizing a set of user who should have access to different services. This allows for easier integration as many systems just allow for 'guest' users or creating new accounts for each data store.
If Fabric is to be matured and adopted by the industries, it needs to be easier to work with it. Furthermore, parts of the system, such as external communication are extremely coupled together. Current requirement is to develop a client application or some middle-ware using their SDK. We would like to see a REST-API built on top of the peers and orderers such that this would integrate more nicely with existing web-services. Furthermore, we also want to see a more decentralized way of configuring the network. Currently the configtxgen tool seem somewhat very limited in terms of scaling the system. More research of safely configuring and scaling large distributed systems using fabric should be conducted. A proposal here is to create an isolated configuration-service which only lives within the initialization of the Fabric network.
Nevertheless, we also suggest to setup a larger scale system in the cloud and conduct further assessments on how to create the architecture described in fig. 5.11. An idea here is to measure the impact of performance when the ordering services serves more than a channel, and how RAFT will perform when the ordering nodes are distributed across different availability zones.
Furthermore, as the consensus types are modular in Fabric we also want to see wrappers for frameworks such as *Quorums* to implement PBFT consensus protocols for Fabric. Moreover, an idea is to utilize the ability to write modular *Endorsement Policies* to allow for more powerful features of entities signing transactions.

8.2 Conclusion

Hyperledger Fabric has proven to be a highly configurable blockchain network that can adapt to any scenario which requires some secret consensus between two or more parties. We also believe that the doubt around "public" blockchains can be closed by the capabilities of Hyperledger. More specifically, Fabric abstracts away the *CryptoCurrency* part of the blockchain-paradigm and puts the trust on pure Public Key Infrastructure between entities. Moreover, as Fabric allows for secrecy, data modelling of real-world assets, and consensus between information systems, we believe that this would be something that the several industries would find interest in. In our project, we have done so by creating a system following the Decentralid Identifier Standard to allow subjects to handle more control of their data. As the subject controls access to the data, we also allow for issuers to ensure that the data's state and integrity are preserved without alteration and later verifiable by third parties. We also move the trust away from centralized database systems to a decentralized database. Furthermore, t the bureaucracy required to issue credentials accross borders do require significant time as this is often manually done by trusted centralized entities. We argue that the bound of the throughput of blockchain systems is slower than centralized, but as we remove the need for manual processes to vouch for the validity of document, we argue that this would save time and cost in the long run. Moreover, moving documents through different organizations that assert the validity of that document also increases the possibility of corrupt assertions. With a running ledger, where an academic diploma is stored and signed, we claim that changing the diploma's state is much harder as one would have to convince a majority of peer nodes in the Fabric Network to hold the same state.

Using the Public Blockchain would also be possible, though it would require off-chain storage to store assets and would indeed require some configuration/interface for such. Hyperledger Fabric is somehow easier to work on a data modeling as all executing peers have an actual instance of a database to manipulate assets. We argue that Fabric should be used for the following scenarios.

- Blockchain as a Service - if the idea is just to ensure some kind of consensus and not care about the possibility of BFT failure.
- All transacting entities must transact privately, and not everyone can view the blockchain.
- Performance - as long as Ethereum uses the Proof-Of-Work Algorithm - Fabric will be faster.

Furthermore, compared to blockchain frameworks such as Ethereum fabric allows for modular architecture - which means that it can be build depending on the requirement of the business. We argue that this is one of the most powerful aspects of the frameworks. This means that smart-contract developers may work with fewer constraints than the development process with unpermissioned blockchains where the scope of the requirements is constrained by the limited availability of commands to execute on the ledger. However, for Hyperledger's case, developing tests on top of a running network is hard as there exists no develop mode, which allows for either static set of available certificates - or assuming no certificates at all. To do so, one will have to run docker on a local host machine and ensure to configure a minuscule version of a more significant project.

Though, it comes at the cost of maintainability in terms of configuration. Throughout the project, significant time was spent on configuring, debugging, and figuring out issues related to the configuration that was not trivial. PIVT pushes the project in a new direction by significantly reducing the required configuration. Still, the backing services required for provisioning the Fabric resources assumes a available Kubernetes Cluster. For cloud environments, this works as many of the popular cloud providers already support Kubernetes [72]. However, PIVT does not scale for cases where the fabric network must be installed in different Kubernetes clusters and remains one of the bigger challenges ahead.

List of Figures

List of Tables

Listings

Appendix A

Appendix A

The Docker-compose YAML file is recommended to be used for developing with fabric on a local network.

```
# docker-compose.yaml
version: '3'
networks:
  basic:  #properties for network goes here
services:
  ca.example.com:
    image: hyperledger/fabric-ca:1.4
    environment: #container environment variables
      - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
    ports:
        - # Ports host_port:container_port
```

Listing A.1: YAML file for docker compose.

The crypto-config file is used to define the entities which participates in the network,

```
#--- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- #
# cryptogen showtemplate                                                                                  #
#--- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- #
OrdererOrgs:

  - Name: NorwegianGovernment # organization 1
    Domain: Government.no
    Specs:
      - Hostname: ordererGovernment

PeerOrgs:

  - Name: UiS
    Domain: UiS.sciencefaculty.com
```

```
    EnableNodeOUs: false
    Users: # amount of users in addition to admin
      Count: 1

  - Name: Ui0 # organization 2
    Domain: Ui0.sciencefaculty.com
     # EnableNodeOUs enables the identify classification based on OUs in X.509 certs.
    EnableNodeOUs: false

    Users:
      Count: 1
```

Listing A.2: crypto-config is used by cryptogen tool

A channel profile is defined within the configtx.yaml file. The asteriks points to the definitions located in the *Organizations* section in the same file.

```
UniversityChannel:
    Consortium: NorwegianUniversities
    Application:
     # point to definition somewhere else in the file.
        <<: *ApplicationDefaults
        Organizations:
            - *UiS
            - *Ui0
            - *NTNU
        Capabilties:
            <<: *ApplicationCapabilities
```

Listing A.3: Channel Profile

Organizations must be defined in the configtx.yaml.

```
 - &UiS
   Name: University Of Stavanger
   ID: UiS_Norway
   MSPDir: crypto-config/peerOrganizations/universityofstavanger.uni.com/msp
   AnchorPeers:
    - Host: peer0.universityofstavanger.uni.com
    Port: 7051
```

Listing A.4: University definition in configtx.yaml

Access control is also defined in each section specific to organizations.

```
  ...
  Policies:
    Readers: # Anyone can read
        Type: ImplicitMeta
```

```
        Rule: "ANY Writers"
    Writers: # Only admins can write
        Type: Signature
        Rule: "OR('UiS_Norway.member', 'NTNU_Norway.Member', ... )"
    Admins: # who are the admins
        Type: Signature
        Rule: "OR('UiS_Norway.admin', 'NTNU_Norway.admin', ... )"
...
```

Listing A.5: ACL Policy

Block parameters are adjusted in the orderer section of configtx.yaml.

```
...
  BatchTimeout: 2s
   BatchSize:
        MaxMessageCount: 10
        AbsoluteMaxBytes: 99 MB
        PreferredMaxBytes: 512 KB
...
```

Listing A.6: Defining block size in configtx.yaml.

Raft settings in under ordererdefaults in configtx.yaml.

```
...
    EtcdRaft:
        Consenters:
            - Host: orderer0.uis.no
              Port: 7059
              ClientTLSCert: crypto-config/ordererOrganizations/uis.no/orderers/orderer
              ServerTLSCert: crypto-config/ordererOrganizations/uis.no/orderers/orderer
            # - Host: ...
...
```

Listing A.7: Enabling RAFT in configtx.yaml.

These settings will define the defaults for the ordering service. The *Consenter* will define which ordering node that participates in the RAFT consensus as either follower, leader or candidate. The Client and Server TLS is essentially certificates that the ordering service require for gRPC communication between other ordering nodes. Furthermore, RAFT-specific parameters can be changed too as shown below.

```
...
    Options:
        TickInterval: 500ms
        ElectionTick: 10
        HeartbeatTick: 1
```

```
        MaxInflightBlocks: 5
        SnapshotIntervalSize: 20 MB
...
```

Listing A.8: Configuring the RAFT-protocol in configtx.yaml.

Appendix B

Appendix B

```
{
    "key": "CredentialProof.AaronSchulist.Degree",
     "doc": {
        "CreatedByOrg":"UiSMSP",
        "data": [
            {
                "CredentialData": "UmVzdWx0OiBB",
                "description": "SuperHardExam",
                "hash": "01fbd15e26b7a6033cc310ac5f4e73156e80994ab20e905bb5a3e5507a9ea73d",
                "key": "Credential.AaronSchulist.SuperHardExam",
                "type": "Credential",
                "parentProof": "CredentialProof.AaronSchulist.DAT520",
                "requiredAgreement": ["x509:Amalia Feest...", "x509: Anissa Bednar..."],
                "whoSigned": ["x509:Amalia Feest...", "x509: Anissa Bednar..."],
                "subject": "Aaron Schulist",
                "txid": "67a7s6d768asd78ee81991278d88a9e9h9h9c8axzczx8c9z90x90"
            },
            {
            "description": "DAT520",
            "hash": "8461bd4414ab9eaabbfd5065afd31317d7cb397910c7d6c6888915bc59cf0351",
            "key": "CredentialProof.AaronSchulist.DAT520",
            "type": "CredentialProof",
            "parentProof": "CredentialProof.Aaronschulist.Degree",
            "owners": "["x509:Amalia Feest...", "x509: Anissa Bednar..."]",
            "providedCredentials": "[Credential.AarondSchulist.SuperHardExam]"
            "quorum": "1",
            "subject":"Aaron Schulist",
            "valid": "true",
            "txid": "8461bd4414ab9eaabbfe0asjidug19239123b1b1ae2b951a0f81"
            }
        ],
        "description":"Degree",
        "hash":"2c7bddafa6f824cb0e682091aa1d9ca392883cb1f5bcff95389adc9feae77fcd", OU=client + OU=uis + OU=user::CN=ca.universit
        "objectType":"CredentialProof",
        "owners":[
            "x509::Paul smith.."
        ],
        "providedCredentials":[
            "CredentialProof.AaronSchulist.DAT520"
        ],
        "quorum":1,
        "subject":"Aaron Schulist",
        "txid":"2e5303380e69821bd199bdf54ae0792fe41e12c992e61906ebb1ae2b951a0f81",
        "type":"CredentialProof",
    }
}
```

Listing B.1: An example of how the System creates a final Credential proof

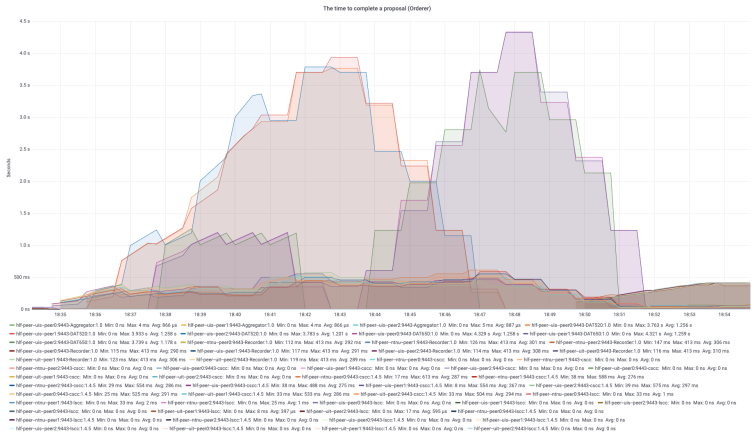


Figure B.3: Proposal times of transactions, with the old configuration the proposal time varies between 3-4 seconds.

B.1 Chapter 6 appendix.

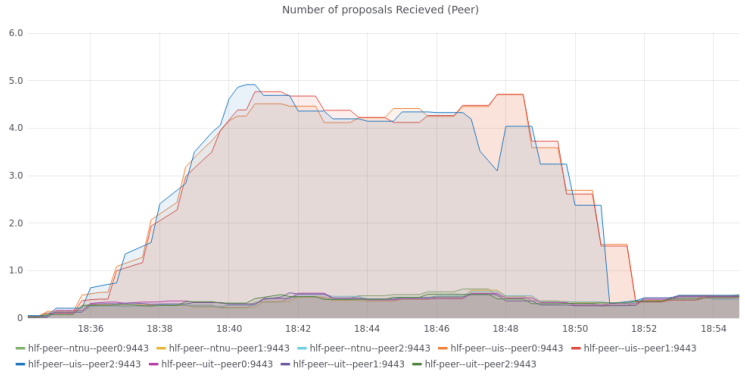


Figure B.1: Transactions Arrival for old configuration.

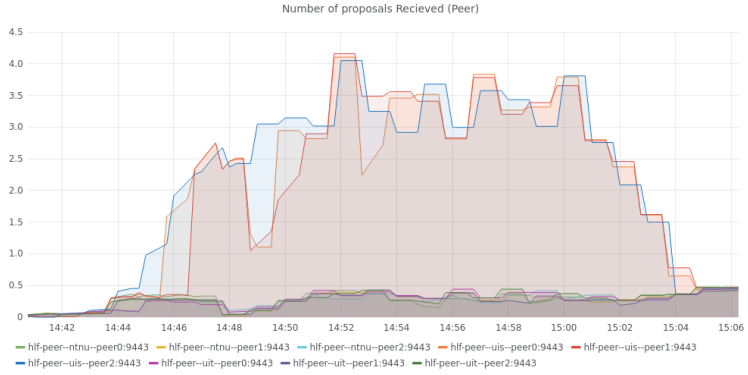


Figure B.2: Transactions Arrival for new configuration.

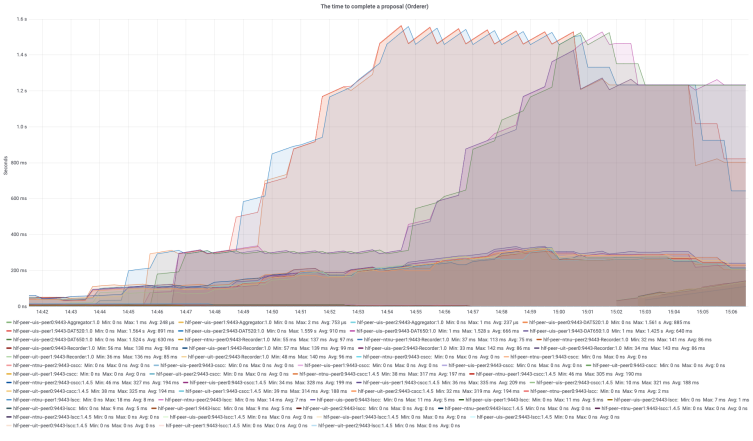


Figure B.4: Proposal times of transactions, new configuration shows improvements, time reduced down to 1.4s - 800ms.

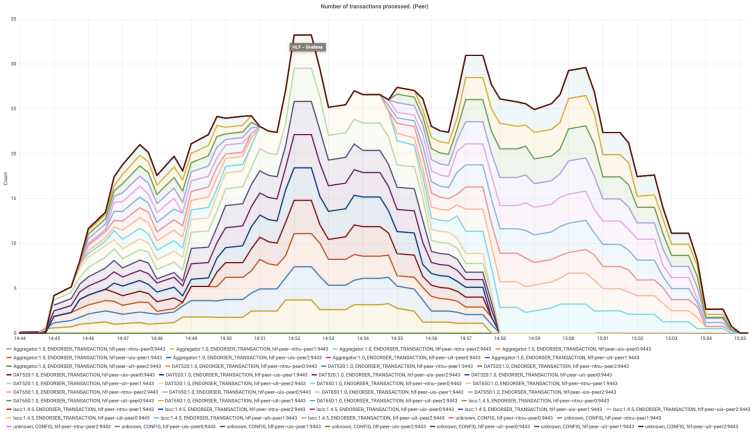


Figure B.5: Throughput of transactions to the ledger by forwarding one transaction to all peers in an org.

Appendix C

Appendix C

Unarchive the  embedded 7z file. To run any fabric samples get the Fabric Binaries from here: https://hyperledger-fabric.readthedocs.io/en/release-1.4/install.html . We use jq 2.6.x, helm 2.16 (3 is not supported), kubernetes (latest), docker (latest), Java 8, Maven 3.35 as dependencies.

- Client Directory - contains the client maven test under the /test directory (WorkFlowTest.java). This file uses /yamlconfig in the same directory. When building a new pivt/local developer environment make sure that private keys are copied in to the connection profile (testLocalKubernetes.yaml).
- Java directory - maven based project ('mvn clean install' the first time), chaincode is inside here. 'chaincodeinstance.java' is the brain of the chaincode. Trace it from here. We propose to use the startNetwork.sh to get a local development environment in the beginning - before moving to kubernetes. This starts one channel, 1 ordering service, and 3 organizations which we developed for local usage.
- localdockerenv directory allows for a simple fabric network that can be used in CI pipelines for chaincode or local development.
- k8s directory contains our modified version of the PIVT project. Refer to the ./makecluster.sh script file within fabric-cube folder to see what commands are required. To install Prometheus and Grafana use commands in listing below.
- When ready with Graphana, navigate to bbchain-scaled-raft-no-tls and import the json file to the project - this will be a statistics dashboard we created for this project.
- Note that crypto materials may become out of sync, so make sure that all parts (client and network) knows about the same certificates etc.

```
helm install --namespace hlf stable/prometheus --name my-prometheus  --set server
    .service.type=NodePort,server.service.nodePort=32333 -f samples/bbchain-
    scaled-raft-no-tls/values.yml --set  pushgateway.nodeSelector."kubernetes\\.
    io/hostname"=bbchain25 --set  server.nodeSelector."kubernetes\\.io/hostname"=
    bbchain25 --set kubeStateMetrics.enabled=true --set nodeExporter.enabled=
    false --set kubeStateMetrics.enabled=false --set server.persistentVolume.
    enabled=false # install prometheus server
helm install --name my-release stable/grafana  # configure prometheus in graphana
     dashboard afterwards
kubectl get secret my-grafana  -namespace hlf -o jsonpath='{.data.admin-password
    }'| base64 --decode
```

Bibliography

[1] Zakir Durumeric, James Kasten, Michael Bailey, and J Alex Halderman. Analysis of the https certificate ecosystem. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 291–304, 2013.

[2] International academic. https://www.verifiedcredentials.com/international-academic/, 2020. Online; Acceses Feb., 2020.

[3] James Crane Keith Ramsdell and Susan Bedil. Taking out the guesswork: A guide to internationalacademic records. https://nagap.org/sites/default/files/A-GUIDE-TO-INTERNATIONAL-ACADEMIC-RECORDS.pdf, 2020. Online; Jan., 2020.

[4] Open, proven, enterprise-grade dlt. https://www.hyperledger.org/wp-content/uploads/2020/03/hyperledger_fabric_whitepaper.pdf, 2020. Online; Accessed: Feb., 2020.

[5] MEDORA. The Oracle Problem. https://medium.com/@DelphiSystems/the-oracle-problem-856ccbdbd14f, 2017. Online; Accessed: March, 2020.

[6] Jake Frankenfield. Blockchain as a service. https://www.investopedia.com/terms/b/blockchainasaservice-baas.asp, 2020. Online; Accessed: Feb., 2020.

[7] BROOFKOREA. South Korean university issues blockchain-stored diplomas amid the spread of the coronavirus. https://www.theblockcrypto.com/linked/55176/south-korean-university-issues-blockchain-stored-diplomas-amid-the-spread-of-the-coronavirus?utm_source=rss&utm_medium=rss, 2020. Online; Accessed: Jan., 2020.

[8] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, page 7, Portland, Oregon, USA, July 2000. Association for Computing Machinery. ISBN 978-1-58113-183-3. doi: 10.1145/343477.343502. URL https://doi.org/10.1145/343477.343502.

[9] Matthias Bauer. Proofs of zero knowledge. *arXiv preprint cs/0406058*, 2004. Online; Accessed: Feb., 2020.

[10] Willians Stallings. *Cryptography and Network Security*. Pearson, sixth edition edition, 2014. ISBN 13: 978-0-13-335469-0.

[11] Edward Felten Andrew Miller Steven Goldfeder Arvind Narayanan, Joseph Bonneau. *Bitcoin and Cryptocurrency Technologies*. Princeton University Press, draft edition, 2016. ISBN Unknown.

[12] Rachid Guerraoui Christian Cachin and Luís Rodrigues. *Introduction to Secure and Reliable Systems*. Springer, 2011. ISBN 978-3-642-15259-7.

[13] unknown. Metamask. https://docs.kaleido.io/developers/smart-contracts/metamask/, unknown. Online; Accessed Feb., 2020.

[14] Life Cycle of an Ethereum Transaction. https://medium.com/blockchannel/life-cycle-of-an-ethereum-transaction-e5c66bae0f6e, 2017. Online; Accessed: May., 2020.

[15] Pulling the Blockchain apart.. The transaction life-cycle. https://medium.com/ignation/pulling-the-blockchain-apart-the-transaction-life-cycle-7a1465d75fa3, 2017. Online; Accessed: May., 2020.

[16] Unknown. What is proof of work (pow)? https://academy.binance.com/blockchain/proof-of-work-explained, 2020. Online; Accessed: May., 2020.

[17] Fabian Ritz and Alf Zugenmaier. The impact of uncle rewards on selfish mining in ethereum. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 50–57. IEEE, 2018.

[18] Harish Sukhwani, José M Martínez, Xiaolin Chang, Kishor S Trivedi, and Andy Rindos. Performance modeling of pbft consensus process for permissioned blockchain network (hyperledger fabric). In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 253–255. IEEE, 2017.

[19] HYPERLEDGER. Hyperledger, Pluggable Consensus. https://hyperledger-fabric.readthedocs.io/en/release-1.4/whatis.html, 2020. Online; Accessed: Feb., 2020.

[20] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX} {ATC} 14)*, pages 305–319, 2014.

[21] Christian Cachin et al. Architecture of the hyperledger blockchain fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers*, volume 310, page 4, 2016.

[22] HyperLedger. The ordering service, 2020. URL https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer/ordering_service.html#raft-concepts.

[23] ofnumbers. Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems. https://www.ofnumbers.com/wp-content/uploads/2015/04/Permissioned-distributed-ledgers.pdf, 2015. Online; Accessed: Mar., 2020.

[24] HYPERLEDGER. Hyperledger, Analysis of Transactions. https://hyperledger-fabric.readthedocs.io/en/release-1.4/developapps/analysis.html?highlight=transactions#transactions, 2020. Online; Accesed Mar., 2020.

[25] HYPERLEDGER. Hyperledger, Endorsement Policies. https://hyperledger-fabric.readthedocs.io/en/release-1.4/endorsement-policies.html, 2020. Online; Accesed Mar., 2020.

[26] Hyperledger. Pluggable endorsement and validation policies, 2020. URL https://hyperledger-fabric.readthedocs.io/en/release-2.0/pluggable_endorsement_and_validation.html. (accessed: 01.03.2020).

[27] Medium. System Chaincodes in Hyperledger Fabric — VSCC, ESCC, LSCC, ESCC. https://medium.com/coinmonks/system-chaincodes-in-hyperledger-fabric-vscc-escc-lscc-cscc-a48db4d24dc3, 2020. Online; Accessed May., 2020.

[28] HYPERLEDGER. Channels in Hyperledger Fabric. https://hyperledger-fabric.readthedocs.io/en/release-1.4/channels.html, 2020. Online; Accesed Mar., 2020.

[29] HYPERLEDGER. Updating a channel configuration. https://hyperledger-fabric.readthedocs.io/en/release-2.0/config_update.html, 2020.

[30] Hyperledger Fabric. Private data in hyperledger fabric, 2020. URL https://hyperledger-fabric.readthedocs.io/en/release-1.4/private-data/private-data.html. Online; Accesed: 01.03.2020.

[31] Nigel Poulton. *Docker Deep Dive*. Nigel Poulton, fifth edition edition, 2019. ISBN Unknown.

[32] Denyeart. Sample config file. https://github.com/hyperledger/fabric/blob/release-1.4/sampleconfig/core.yaml#L59, 2020. Online; Accessed: Feb., 2020.

[33] Kubernetes. Kubernetes Load Balancers. https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/, 2020. Online; Accesed Feb., 2020.

[34] Kubernetes. Kubernetes Volumes. https://kubernetes.io/docs/concepts/storage/volumes/, 2020. Online; Accesed Feb., 2020.

[35] Kubernetes. Secrets in kubernetes. https://kubernetes.io/docs/concepts/configuration/secret/, 2020. Online; Accesed Fev., 2020.

[36] HYPERLEDGER. Policies in Hyperledger Fabric. https://hyperledger-fabric.readthedocs.io/en/release-1.4/policies.html, 2020. Online; Accessed: Feb., 2020.

[37] HYPERLEDGER. Raft Configuration. https://hyperledger-fabric.readthedocs.io/en/release-1.4/raft_configuration.html, 2020. Online; Accessed: Feb., 2020.

[38] Unknown. Interacting with the network, 2020. URL https://hyperledger-fabric.readthedocs.io/en/release-2.0/test_network.html#interacting-with-the-network. (accessed: 05.03.2020).

[39] Hyperledger. Hyperledger fabric client identity, 2020. URL https://hyperledger.github.io/fabric-chaincode-java/master/api/org/hyperledger/fabric/contract/ClientIdentity.html. (accessed: 01.02.2020).

[40] Hyperledger, 2020. URL https://hyperledger-fabric.readthedocs.io/en/release-2.0/commands/peerlifecycle.html#peer-lifecycle-chaincode-commit. accessed: 05.06.2020.

[41] Unknown. Hyperledger fabric sdk, 2020. URL https://github.com/hyperledger/fabric-sdk-java. Online; Accessed: Mar.,2020).

[42] Rodrigo Q. Salamaco. Data model and solution for ethereum in bbchain project. Discussion throughout the semester, 2020. URL Papernotyetpublished.

[43] Steven McKie. What comprises an ethereum fullnode implementation?, 2019. URL https://medium.com/amentum/what-comprises-an-ethereum-fullnode-implementation-a9113ce3fe3a. (accessed: 02.02.2020).

[44] Jianyu Niu and Chen Feng. Selfish mining in ethereum. *arXiv preprint arXiv:1901.04620*, 2019.

[45] Guangquan Xu, Bingjiang Guo, Chunhua Su, Xi Zheng, Kaitai Liang, Duncan S Wong, and Hao Wang. Am i eclipsed? a smart detector of eclipse attacks for ethereum. *Computers & Security*, 88:101604, 2020.

[46] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. *IACR Cryptology ePrint archive*, 2016:1007, 2016.

[47] Unknown. Architectural overview, unknown. URL https://swarm-guide.readthedocs.io/en/latest/architecture.html. (accessed: 02.03.2020).

[48] Unknown. Incentives system, unknown. URL https://swarm-guide.readthedocs.io/en/latest/incentivization.html. (accessed: 02.03.2020).

[49] Rodrigo Q. Salamaco. Ethereum and persistent storage outside smart contracts. Discussion in February, 2020. URL Papernotyetpublished.

[50] Michael Spain, Sean Foley, and Vincent Gramoli. The Impact of Ethereum Throughput and Fees on Transaction Latency During ICOs, 2019. URL https://drops.dagstuhl.de/opus/volltexte/2020/11973/pdf/OASIcs-Tokenomics-2019-9.pdf.

[51] w3.org. Core Data Model. https://www.w3.org/TR/vc-data-model/#core-data-model, 2020. Online; Accesed Mar., 2020.

[52] w3.org. What is a verifiable Credential? https://www.w3.org/TR/vc-data-model/#what-is-a-verifiable-credential, 2020. Online; Accesed Mar., 2020.

[53] w3.org. Definition Credential. https://www.w3.org/TR/vc-data-model/#dfn-credential, 2020. Online; Accesed Mar., 2020.

[54] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30, 2016.

[55] James Ray (Last known edit). Sharding roadmap, 2018. URL https://github.com/ethereum/wiki/wiki/Sharding-roadmap#ethereum-20. Online; accessed: 04.03.2020.

[56] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[57] Joao Sousa, Alysson Bessani, and Marko Vukolic. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th annual*

IEEE/IFIP international conference on dependable systems and networks (DSN), pages 51–58. IEEE, 2018.

[58] Christopher Copeland and Hongxia Zhong. Tangaroa: a byzantine fault tolerant raft, 2016.

[59] HGF. Hyperledger Global Forum workshop . https://github.com/aidtechnology/hgf-k8s-workshop, 2020. Online; Accessed May., 2020.

[60] Helm. Helm charts, 2020. URL https://helm.sh/docs/topics/charts/. Online; Accessed: 04.08.2020.

[61] Alejandro (Sasha) Vicente Grabovetsky & Nicola Paoli. Deploying hyperledger fabric with kubernetes/helm part 1, 2018. URL https://www.youtube.com/watch?v=ubrA3W1JMk0.

[62] Alejandro (Sasha) Vicente Grabovetsky & Nicola Paoli. Deploying hyperledger fabric with kubernetes/helm part 2, 2018. URL https://www.youtube.com/watch?v=ubrA3W1JMk0.

[63] feitnomore. Blockchain Solution with Hyperledger Fabric + Hyperledger Explorer on Kubernetes. https://github.com/feitnomore/hyperledger-fabric-kubernetes, 2020. Online; Accessed May., 2020.

[64] pivt. The PIVT project. https://github.com/APGGroeiFabriek/PIVT, 2020. Online; Accessed: May., 2020.

[65] hyperledger. Clientidentity java documentation. https://hyperledger.github.io/fabric-chaincode-java/master/api/org/hyperledger/fabric/contract/ClientIdentity.html, 2020. Online; Accesed Mar., 2020.

[66] Quickstart: Deploy an Azure Kubernetes Service (AKS) cluster using the Azure portal. https://docs.microsoft.com/en-us/azure/aks/kubernetes-walkthrough-portal, 2020. Online; Accessed: Jun., 2020.

[67] Tutorial: Deploy an Azure Kubernetes Service (AKS) cluster. https://docs.microsoft.com/en-us/azure/aks/tutorial-kubernetes-deploy-cluster, 2020. Online; Accessed: Jun., 2020.

[68] GANACHE OVERVIEW. https://www.trufflesuite.com/docs/ganache/overview, 2020. Online; Accessed: Jun., 2020.

[69] Truffle overview. https://www.trufflesuite.com/docs/truffle/overview, 2020. Online; Accessed: Jun., 2020.

[70] Andrew Morsillo. Helm deployed my chart resources, but fails to record the release so it cannot be managed #4574, 2020. URL https://github.com/helm/helm/issues/4574. Online; Accesed May., 2020.

[71] GEEKSFORGEEKS. Thread Pools in Java. https://www.geeksforgeeks.org/thread-pools-java/, 2020. Online; Accessed: May., 2020.

[72] Platform9. Kubernetes cloud services: Comparing gke, eks and aks, 2020. Online; Accessed: 01.06.2020.