Collaborative Drug Discovery with Blockchain Technology

Christoffer Olsson

Department of Electrical and Information Technology
Lund University

Supervisor: Mohsen Toorani

Examiner: Thomas Johansson

September 3, 2020

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

The cost of developing and researching novel molecular compounds in the pharmaceutical industry is very high today. If an actor accidentally releases too much information about a novel molecule to the public the molecule becomes prior art, and as an effect, can not be patented. This has resulted in a fragmented space where collaboration across organizations is virtually non-existent.
We propose a blockchain-based platform where participants can partially upload information of molecules while maintaining full ownership of the asset. This way novel molecules do not become prior art and gives the developer of the molecule the ability to reveal to the world that they have a molecule with a certain property, without actually revealing the molecule itself. Larger organizations could use this to auction out molecules that they are not interested in bringing to clinical trials or market, smaller organizations could use it to attract funding for a molecule that they have developed, and researchers could use it to timestamp that they knew the properties or a structure of a molecule at a certain date.
A prototype is implemented using the blockchain technology *Hyperledger Fabric* and is analyzed from security and performance perspectives. Given the right circumstances, Fabric provides a set of functionalities that can be used to make sure that ownership is maintained, integrity is protected, and critical information remains confidential. From a performance perspective, Fabric provides a good throughput and latency in the order of milliseconds. However, the number of participants that can maintain the network seems to be limited in the prototype. These results should be seen as preliminary as no fine-grained optimization was conducted. Further work required to make the prototype production-ready and open problems are also discussed.

Popular Science Summary

Developing pharmaceutical molecules is expensive. Organizations that develop pharmaceutical molecules pour a vast amount of resources into developing even a single molecule. In order to protect the rights of the molecule that was developed *patents* are used. However, to receive a patent the organization has to reveal exactly how their new molecule functions. This means that if an organization reveals what they are working on to others there is a high chance that they can not patent their work. This means that organizations risk being punished if they try to collaborate. Since it is so expensive to develop new molecules several actors in the space would like to collaborate.

Blockchain technologies aim to solve these kinds of problems. Namely, how can actors that do not trust each other collaborate? We expect actors that do not trust each other to try to cheat the system. To prevent them from cheating *smart contracts*, *consensus protocols* and the *blockchain data structure* are used. A smart contract is a piece of code that participants can use in order to transact value amongst themselves. In the case of this thesis, they transact representations of molecules. Consensus protocols are protocols that make sure that all participants agree on the result of the smart contracts. Finally, the blockchain data structure is a chain of blocks. Each block points to the preceding block. It is used to verify that no data has been corrupted. *Hyperledger Fabric* is such a blockchain technology.

A prototype is developed using Fabric. The prototype allows participants to upload partial information of molecules that they are developing. This way, they can reveal to the world what they are working on without exactly revealing the structure of the molecule. Using this, they can find other people that are interested in collaboration without losing their ability to claim a patent later. Individuals that are not interested in patents, e.g. scientists, can upload the full molecule in order to prove that they knew a molecule at a certain date. The developed prototype is analyzed from a security and performance perspective and open problems that need to be solved before the prototype is production-ready are also discussed in the thesis.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1
Introduction

Research and development in the pharmaceutical is a highly closed and siloed endeavor. Actors have a hard time collaborating, research and development suffer from inefficiencies and money is wasted. As a response to this, several actors are calling for a more transparent and open collaboration. Larger companies should ideally want to pool resources and share the fruits of their research and smaller companies want an easier time receiving funding for research. The problem so far, however, has been that there is no easy way to keep track of contributions and ownership of propriety. This has resulted in a somewhat paranoid space where it is very hard for competing actors to trust each other. As a result collaboration for the benefit of all is virtually non-existent [31].
We propose a solution where rights for molecules are tokenized and stored using the blockchain platform *Hyperledger Fabric* [2]. This is a vast undertaking and this thesis has no ambition to solve the problem in its entirety. What is proposed, however, is a prototype that can be used as a stepping stone to fully realize a system where a multitude of pharmaceutical actors can share resources *while providing strong reassurance that the correct actors receive and retain their intellectual propriety.*

1.1 History of Decentralized Computing

Tannenbaum & Van Steen [45] define a distributed system as *a collection of independent computers that appears to its users as a single coherent system.* As the cost of computers has fallen dramatically during the last decades, combined with the fact that computer networking has become increasingly faster and more reliable, more people have moved toward distributed computing. Distributed computing has enabled humankind to build software systems at scale. No single computer could process the amount of data that flows through a medium to large-sized organization today. As society becomes further digitized, more value from our physical lives are moved onto computer systems. In the end, there is nothing that physically prevents the owner of one of these systems to misuse the data that flows through it. This is problematic where several actors want to share a computing system as equals. Historically, someone owned the master-key to the databases and this actor had to be trusted not to abuse their position.
As a reaction to this, there has been a surge in the interest of decentralized com-

puting systems. Namely, systems where no single entity controls the entirety of the computing stack. Even though this concept has been around for a while, e.g. Bitgold [44], peer-to-peer torrenting [11], it could be argued that the dream of decentralized systems took off for real with the invention of Bitcoin [28]. Bitcoin proved that it is possible to build a distributed system where trust has been disintermediated, at scale.
In essence, Bitcoin is a distributed timestamp server with a built-in stack-based virtual machine [28]. Users of the network can pass scripts to the virtual machine. The purpose of these scripts is to sign over ownership of coins to other users. By design, the scripting language is simple in order to decrease the attack surface of the network. Using consensus mechanisms, the results of script executions will eventually be consistent across all well-behaved nodes. This begs the question, what if the virtual machine inside of Bitcoin is extended to a fully-featured Turing machine? This idea is what first spawned Ethereum [7] and subsequently, other smart contract platforms. A smart contract is a concept proposed by Nick Szabo [43], where the specification of a contract is formalized so that it can be executed by a computer. Essentially, a smart contract is a piece of code that moves value from one entity to another depending on its execution path. Code that moves value has existed for several decades. One does not need smart contracts to operate e.g. a bank or a digital market. The idea is that smart contracts should be able to create markets where there is little of or no need for an intermediary party. Even though it is theoretically possible to build any program, e.g a bank, using Ethereum it has proved difficult to do so in practice. The public nature of Ethereum means that anyone can join the network and participate. This is not optimal for enterprise use if confidentiality is of importance. Using Ethereum, all data is copied to every node in order to be verified. Also, the push-only nature of public blockchains means that it is hard to revert bugged smart contract implementations. Finally, it has proven to be hard to scale throughput in public smart contract platforms, while maintaining security.
In order to answer the shortcomings of public smart contract platforms, the Linux Foundation initiated a project called Hyperledger [23]. Hyperledger is a collection of smart contract platforms, with the key difference that they are permissioned. That is, all participants of the network are invited, as opposed to a public network, where anyone can participate. This way a consortium of organizations can establish a network amongst themselves, keeping unwanted participants out of the network, while providing trust disintermediation. Fabric is thoroughly discussed in Chapter 3 and is the smart contract platform used to implement the prototype proposed in this thesis. Another Platform called *Corda* was also considered. However, Corda is more suited toward financial services as of today, whereas Fabric is a general-purpose smart contract platform.

1.2 The Pharmaceutical Industry

Open collaboration is on the rise in the pharmaceutical industry [1]. The idea is that this will accelerate the development of new compounds benefiting both parties in such an agreement. Astra Zeneca, a major pharmaceutical actor in Sweden, identified that as more collaborations are established it gets increasingly difficult to keep track of ownership rights, origin, and usage rights. This is due to the fact that ownership rights to compounds used to mainly belong to a single company. Today rewards and risks of molecule development need to be shared across several organizations [1]. This means that most compound tracking tools are not designed to enable cross-organization collaboration in a painless way. They propose a system called *Compound Passport Service*. A *Passport*, according to their definition, is the digital rights to a compound. This way, they could less painfully keep track of which actor owns what right to compounds, even when working with external collaborators. This also means that they could automate usage rights to assets. Furthermore, they assess that open collaboration will probably continue to increase and express an interest in other systems that can further increase cross-organization collaboration in a painless way [1].

1.3 Problem Statement

This thesis proposes a prototype that aims to partially solve the problem that Astra Zeneca identified using Hyperledger Fabric. The difference is that the proposed prototype aims to work across several pharmaceutical organizations both large and small in a trust disintermediated way using blockchain technology. We were looking for answers to the following questions:

1. *How does Hyperledger Fabric work?* How is its architecture designed? How does one use it? - This is answered in Chapter 3.
2. *Problem Analysis* - What problems can be identified and how can they be solved? Discussed in Chapter 4.
3. *How will compounds be modelled?* - This is discussed in Chapter 4.
4. *What does an implementation of the Data Model look like?* - Discussed in Chapter 5.
5. *How secure is the prototype?* - This is discussed in the Chapter 6.
6. *Will the prototype scale to a high number of participants?* - It is important that many actors can join the network. Therefore a performance analysis was conducted. This is discussed in Chapter 7.
7. *What needs to be considered to bring the prototype to a production-ready version?* - This is discussed in Chapter 8.

1.4 Contributions

This thesis mainly contributes the following:

- *An analysis and a proposed solution to Astra Zeneca's problem* [1]: Even though there exist other platforms for molecule tracking [1], building such a platform using Fabric is novel as far as we are aware.
- *A high-level security analysis of the proposed prototype*: There exists no overarching discussion on the security properties of Fabric. The paper that introduces Hyperledger Fabric [2] briefly mentions it, and official documentation [19] of Fabric is still marked as work in progress.
- *A performance evaluation of the proposed prototype*: There exists literature that evaluates the performance of Fabric [46, 30, 29, 17, 41]. However, they focus on either a fixed network topology or a smaller amount of entities and do not investigate how Fabric scales when one adds a larger amount of entities to the network.
- *A discussion on open problems identified while working on the prototype*: The open problems can be separated into two categories: What it means to tokenize molecules, and technical problems that are needed to be solved in order to make Fabric scale.

1.5 Outline

The thesis is structured as follows:

- Chapter 2 provides a background on important topics needed in order to understand this thesis and smart contract platforms in general.
- Chapter 3 provides a thorough description how *Hyperledger Fabric* works on a high level.
- Chapter 4 provides an analysis on the problem of open collaboration of molecules, why collaboration is needed, and how one can translate the problem and its solution into the blockchain domain.
- Chapter 5 provides a discussion of the prototype implemented for this thesis.
- Chapter 6 provides a high-level security analysis of the implemented prototype.
- Chapter 7 provides a performance evaluation for the implemented prototype using various network topologies.
- The thesis is concluded with Chapter 8, which provides a discussion on open problems that need to be solved before the prototype can be brought to a production-ready state.

Chapter 2
Theoretical Background

2.1 Tokenization

Tokenization is the minting of an asset that represents some other phenomenon [3]. A token can represent one's right to park a car in a parking lot, the rights to a physical painting, or the right to a revenue stream from some other asset. In the case of this thesis, a token is a digital representation of a chemical compound. In short, pretty much anything can be tokenized. The challenge lies in keeping the tokenized representation in accordance with the physical asset. This is known as the *Oracle Problem* [13]. Namely, how can one safely convert physical assets to digital tokens when there exist incentives to cheat?

Fungible and Non-Fungible Tokens

Fungibility is when one unit of a good is interchangeable with another [3]. For example, national currencies are fungible. One dollar is considered equal to another in a transaction as they are both worth the same. Paintings, however, are highly non-fungible. The value of one painting varies wildly from the value of another.
Some goods can be partially fungible. Bitcoin is an example of this. Even though one bitcoin is interchangeable to another, one can track the exact history of all transactions that lead to the current state of the coin [28]. As an effect one coin could be evaluated differently to another, depending on the histories of the two coins.

2.2 Intellectual Property

Intellectual property (IP) are property rights regarding intangible creations of the human mind [37]. Intangible creations are abstract constructions. In essence, they are the opposite of concrete properties such as physical goods, or owned land. Examples of intangible creations are computer software, music, art, and business plans. Intangible products can have a physical manifestation once they are written to a physical medium. For example, software can be stored on a physical hard-drive. The reason a society needs rights for intangible creations is because they are *inexhaustible*. An inexhaustible good is a good that has an infinite number of uses. For example, once a certain business plan is public knowledge, everyone can

use it. This is a stark contrast to physical goods like apples. Each apple can only be eaten once. IP can be categorized as follows [37]:

- *Copyright law* - Laws relating to the rights of copying the physical manifestation of an intellectual property. For example, who has the right to copy and commercialize a book.
- *Trademarks* - Laws regarding recognizable brands. For example, who has the right to use a certain picture to promote their products.
- *Design Rights* - Laws regarding design of products. That is, who has the right to use a certain design for a physical product.
- *Patents* - Laws regarding technical innovations. That is, who has the right to use certain technical innovations.
- *Trade Secrets* - Laws regarding secrets that one can possess. Certain secrets give competitive advantages, and trade secret laws aim to help protect these secrets in a fair way.

Given that intangible products are virtually free to duplicate, IP law aims to protect the rights of, and provide incentives to people that innovate [37]. For example, the moment someone writes a novel, the novel is protected by copyright laws. Anyone that copies the book unlawfully can be taken to court. IP laws vary across jurisdictions. For this thesis mainly *Patents* and *Trade Secrets* are of interest.

2.2.1 Patents

A patent is an exclusive right to a technical innovation [37]. A patent does not give the right to produce the technical innovation, but it does give the holder the right to exclude anyone else from producing the innovation. In order to receive a patent, the recipient has to disclose the technical details of the innovation to the public. The idea is that the inventor will be incentivized to reveal their innovation to the public, thus benefiting humankind as a whole, while receiving a monopoly for a limited time. The monopoly, in turn, can be used to regain the money that was invested into the invention. For an invention to be patentable the following must be fulfilled [37]:

1. *Novelty*: The invention has to be novel. *Prior Art* is a term used to describe what knowledge humankind as a whole possesses about technical products. For an innovation to be novel it has to be outside of the prior art.
2. *Inventive Step*: The innovation has to have an innovative step. That is, it has to be an actual technical innovation and solve a technical problem. Moreover, this technical innovation must not be obvious to skilled people in whatever field the innovation came from.
3. *Industrial Applicability*: There must be reasonable belief that the product can be produced industrially.

2.2.2 Trade Secrets

When one applies for a patent one has to reveal the technical details of the innovation. In some situations, this is not ideal, for example, if one expects the product to live longer than the duration of the patent or if one believes that it is hard to enforce the patent due to the nature of the technical product. In this situation it might be better to rely on *Trade Secrets* [37]. A product can be protected by trade secret laws if one believes that the disclosure of the secrets will harm the owner of the secrets. That is, something can be considered a trade secret if it gives the owner a competitive edge.
Protection around trade secrets is not as strong as it is for patents. For example, if a competitor discovers one's trade secret via research, the competitor is free to use the discovery commercially. However, one can explicitly mark certain IP as a trade secret. Any unlawful disclosure of the secret can then be taken to court [37].

2.3 Decentralized vs. Distributed

The terms *decentralized* and *distributed* are ubiquitous when discussing blockchain. *Distributed* is a physical characteristic of a system. It means that the system is distributed across processes. *Decentralized* systems are a subset of distributed systems. They are also dispersed across processes. The difference lies in governance. A decentralized system is a system where decisions are not centrally organized.
Decentralization is not an absolute but a scale. The most radically decentralized system is something like a true peer-to-peer network, for example, *Bittorent* where there is no leader at all. However, a system where a handful of actors make decisions could also be considered decentralized.

2.4 CAP-Theorem

A distributed system can not be both consistent and available during a network partition [15]. A distributed system can only have two out of three of the following properties at all times.

1. *Consistency*: Components of a network return the correct value for each request. That is, if one writes $x = 5$ to a distributed system, all queries for x will return 5.

2. *Availability*: When sending a message to a node, it will respond.

3. *Partition Tolerance*: The network functions even when one node can not send any message to another node.

This is known as the *CAP-theorem* (Consistency, Availability and Partition Tolerance) [15]. This has major implications for distributed systems, as network partitions will happen due to the distribution of computers. In practice one has to choose *consistency* & *partition tolerance*, or *availability* & *partition tolerance*.

2.5 Hyperledger Caliper

Hyperledger Caliper is a tool that can be used to benchmark blockchains [24]. One defines either connections information to an existing Fabric network or a new Fabric network. Moreover, one also defines benchmark tests that are run by Caliper. Caliper collects performance metrics and generates reports when it is run. Caliper was used in the course of the thesis to run performance evaluations. This is further discussed in Chapter 7.

2.6 Consensus

A core issue of distributed systems is to reach consensus among processes. Processes that share state have to somehow agree on how the state is updated. A consensus protocol is a protocol where there are at least 2 processes [14]. The processes communicate via sending messages. The processes need to reach consensus on values proposed by one or more of these processes. The following properties can be defined for the consensus problem [26]:

1. *Safety*: Can be defined as: *Nothing bad will ever happen.* That is, the execution of the algorithm will not produce faulty results.

2. *Liveness*: Can be defined as: *Computation will progress.* That is, the algorithm will not halt.

2.7 Crash Fault-Tolerant Consensus Protocols

A class of consensus protocols are considered crash fault-tolerant. *Safety* and *liveness* of the protocol are not threatened if a certain number of processes crash. Crash fault-tolerant protocols are a subset of *Byzantine fault-tolerant protocols*, which can handle arbitrary failures. Byzantine fault-tolerant (BFT) protocols are discussed below.

2.7.1 Raft

Raft [32] is a crash fault-tolerant consensus protocol. Previous crash fault-tolerant protocols were criticized for being too complicated. As a response to this, Raft was constructed to be an easy to understand protocol. In theory, this should make it more reliable as it is easier for humans to understand the implementation [32]. Raft is a consensus protocol that manages a replicated log by forming a cluster of nodes. A leader is elected by the set of nodes. The leader proposes new values and commits the value to the other nodes that are called followers. If a leader is not present elections are held. A potential new leader is called a candidate. Each node has a local value called the *term number*. Nodes always follow leaders with the highest term number [32].

Leader Election

If a leader exists, it periodically sends heartbeats to the followers. If a node misses a heartbeat, it holds an election. The node enters the candidate state and increments its term number. It votes for itself and sends a message to all other nodes that it is holding an election. The message contains the term number. Three outcomes are possible [32]:

1. The followers receiving the election proposal recognizes that the candidate node has the highest term number. They respond that they will now follow the candidate node. Once it receives a majority of votes it considers itself leader and starts sending heartbeats to the followers.
2. The other nodes know a leader with a higher term number. They reject the candidate node. It could also be the case that they receive a message from another leader node with a higher term number during the election. Nodes always follow nodes with the highest term number. The candidate will eventually become a follower once it receives a message from the leader that had the higher term number.
3. If two nodes hold an election at the same time, there could be a deadlock if neither of them can receive a majority of votes. In this case, the election will time out and nodes are free to become new candidates in a new election round.

The nodes rely on random timers to avoid deadlocks. For example, if two nodes have the same term number, they are both eligible to become leaders. Therefore, when the election times out they wait a short but random amount of time. This way it becomes less likely that they will reach an election deadlock.

Log Replication

A client sends values it wants to commit to the log to the leader peer. Once a leader peer receives a request to commit a value it sends the value to all followers along with an index and its term number. The term number is used so that followers can verify that the received value is from an eligible leader. The index is a number starting from one. Each committed value increases the index by one. The index is used to keep track of values. The followers respond that they have committed the value to their log. Once the leader has received a confirmation message from a majority of the followers it considers the value properly committed and responds that the commit was successful to the client. The leader has total authority over the log. If logs become inconsistent among nodes, followers always replicate the leader's log.

Safety and Liveness of Raft

Safety is maintained by the leader. As long as the leader is correct, all followers will eventually replicate its log. Moreover, when elections are held or nodes time out, there can be some downtime when the cluster decides on its next leader. In this situation, followers ignore messages from leaders whose term number is lower

than the highest term number they know of until a leader emerges with a high enough term number.
Liveness is maintained during process crashes by the leader election: if a leader times out a new leader will emerge eventually. However, this only holds if at least a majority of nodes can hold an election. If more than half of the cluster crashes, the remaining nodes can not elect a new leader and thus the log replication is halted.

2.8 Byzantine Generals' Problem

Crash fault-tolerant protocols work under the assumption that nodes behave according to the protocol. Crashes can be handled, but what happens if one or more nodes behave arbitrarily? This is what is known as the *Byzantine Generals' Problem* [27]. Consider an army consisting of one general and his lieutenants surrounding a city. They have to coordinate if they are to attack or retreat from the city. They communicate via messages. However, one or more of the commanders might be traitors and behave in a way that prevents the well-behaved commander from reaching the same plan. Regarding *safety*, we consider protocol execution successful if all lieutenants reach the same conclusion because the general that proposes can be a traitor. Moreover, we define *retreat* as a default value. This way one does not have to consider messages that are lost, or if the general simply refuses to deliver any messages [27]. A general is equivalent to an elected leader in Raft. Lieutenants are equivalent to followers. For consistency, we will refer to commanders as nodes. A node that diverges from protocol is considered *byzantine*. Moreover, we consider nodes sending messages containing a decision on 0/1 instead of *retreat/attack*. A byzantine node potentially threatens all desired properties of a consensus protocol that does not take byzantine faults into account.
An example demonstrates how severe byzantine faults can be. A byzantine node in Raft could set its *term number* to a very high number and then hold an election. It is likely that this node would become the new leader. The node could then alter the established log in a malicious way. This destroys the safety of the protocol. Moreover, the node could refuse to deliver commit messages from honest clients. This would destroy the liveness of the protocol. The core issue of byzantine faults is that it removes all restrictions on participating nodes, rendering them free to behave in an arbitrary way. Lamport et al. [27] showed that any deterministic byzantine fault-tolerant protocols maintain safety and liveness given that at least $3m + 1$ nodes are well-behaved, where m is the number of byzantine nodes.

2.8.1 Naive Byzantine Fault-Tolerant Protocol

The $OM(m)$ algorithm solves the byzantine generals' problem for m traitors [27]. Let m be the number of traitors and $3m + 1$ be the number of nodes. A naive approach to solving the byzantine generals' problem is to have all followers relay the message they received from the leader to all other followers. This way followers select the value that a majority of other nodes have selected. The $OM(m)$ algorithm is described in algorithm 1. To further illustrate the algorithm an example

is provided. Two situations have to be considered, when the leader is byzantine and when a follower is byzantine.

Algorithm 1: OM(m): Simple algorithm that solves byzantine generals' problem

Result: All followers reach same conclusion
if *m=0* **then**
  Leader sends value to followers;
  Followers accept value;
else
  Leader sends value to followers;
  Each follower sends value to other followers;
  Accepted value for follower$_i$ ← Majority of values from other nodes;
  OM(m-1), where new leader ← follower$_i$;
end

Byzantine Follower

Consider one leader (L) and three followers (F1, F2, B) of which one is byzantine (B). The leader sends 1 to every follower. The byzantine node sends 0 to the other followers. The $OM(m)$ algorithm is illustrated as a tree in Figure 2.1. Table 2.1 contains the values that each follower node received. Each row depicts the message that the follower received from the other followers. This way the nodes can safely select 1 as the result by simply choosing the value that they received in a majority of the messages.

Byzantine Leader

The other situation is when the leader itself is byzantine. The leader sends 0 to the first follower and 1 to the other followers. The algorithm execution is depicted in Figure 2.2. In Table 2.2 the values each follower receives from other nodes are depicted. By taking the majority of each row, each follower can safely elect 1 as the result.

Name	L	F1	F2	B
F1	1	-	1	0
F2	1	1	-	0
B	1	1	1	-

Table 2.1: $OM(1)$ Byzantine follower

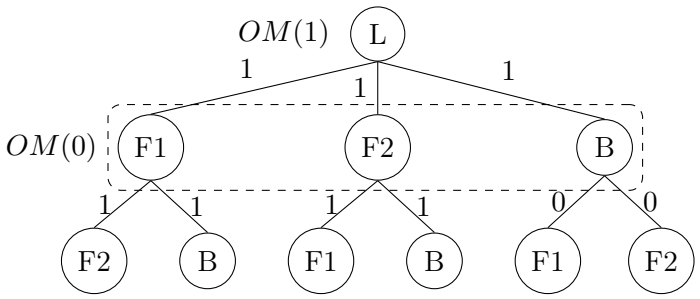


Figure 2.1: $OM(1)$ Byzantine Follower

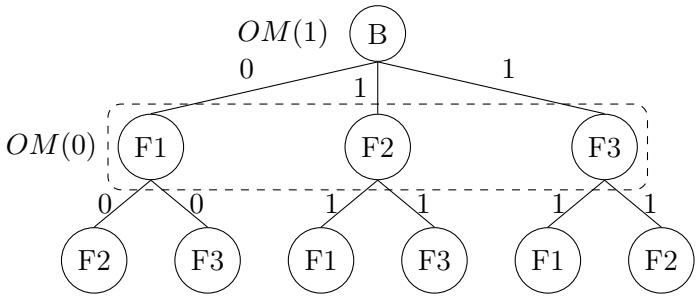


Figure 2.2: $OM(1)$ Byzantine Leader

Name	B	F1	F2	F3
F1	0	-	1	1
F2	1	0	-	1
F3	1	0	1	-

Table 2.2: $OM(1)$ Byzantine leader

2.8.2 Practical Byzantine Fault Tolerant Protocol

Practical Byzantine Fault Tolerant Protocol (PBFT) [8] is another byzantine fault tolerant protocol. There are plans to implement this algorithm in near future releases of Fabric [19]. Let m be the number of byzantine nodes and $R = \{0, \ldots, 3m\}$ denote the set of nodes. We have $|R| = 3m + 1$. All messages passed between nodes are signed. $v \in R$ is called *view* which defines the current leader. Leader election is done via *Round Robin*: the current leader is $v \ mod \ |R|$. The next leader is $v + 1 \ mod \ |R|$. The algorithm is described in algorithm 2 and contains three steps: *pre-prepare*, *prepare*, and *commit*. The purpose of *pre-prepare* is for the current leader to propose a unique sequence number n for the request to the followers. If any of the requirements fail the follower will reject the proposed n. *Pre-prepare* is described in algorithm 3, in which m is number of byzantine nodes, n is a *sequence number*, σ is a signature, v is the current view and d is *Hash(m)*. The purpose of *prepare* is for the followers to agree on the sequence number n.

Once a follower has received $2m$ valid prepare messages from other followers with the same n the follower accepts the sequence number. *Prepare* is described in algorithm 4. Finally, the purpose of *commit* is to commit the value. Once all nodes have established a unique and shared sequence number n for the request they will try to commit the request from the client. *Commit* is described in algorithm 5. Once the client receives $m + 1$ confirmation messages from the nodes it considers the request committed and is free to make new requests to the consensus consortium.

Algorithm 2: Practical Byzantine Fault Tolerant Protocol

```
Result: Committed value to shared log
Client sends request to Leader;
if valid request then
    1. Pre-prepare;
    2. Prepare;
    3. Commit;
end
Nodes send response to client;
if Number of identical responses > m then
    Client accepts;
end
```

Algorithm 3: Pre-Prepare

```
Result: Accepted or rejected n
Leader sends (d, v, n, σ) to all followers;
foreach follower_i do
    if d, v, n, σ are valid then
        if (v,n) has not been processed for another d then
            follower Accepts n;
        end
    end
end
```

Algorithm 4: Prepare

```
Result: Accepted or rejected n
foreach follower_i do
    Send (n, d, i, v, σ) to other followers;
    if Received values: n, d, i, v, σ are valid then
        if 2m > Valid prepare messages received then
            follower_i, Accepts n;
        end
    end
end
```

Algorithm 5: Commit

```
Result: Accepted or rejected commit
foreach node_i do
    Send commit message containing (d, v, n, σ) to other nodes;
    if Received values:(d, v, n, σ) are valid then
        if 2m > Valid commit messages received then
            node_i Accepts commit;
        end
    end
end
```

Safety and Liveness of PBFT

Safety and *Liveness* of *PBFT* is guaranteed if there are no more than m byzantine nodes for a network of $3m + 1$ nodes [8]. If m byzantine nodes are followers, they can not threaten liveness nor safety as the protocol is designed to reach consensus no matter what value they propose. If they refuse to respond to requests the algorithm is designed to reach a safe consensus by ignoring their messages using time-outs. If the leader is byzantine, the network holds an election called a *view-change*. A view-change occurs if a follower suspects that the leader is faulty. The suspecting follower sends a view change request to other followers. The new leader is $v + 1 \ mod \ |R|$. Once the new leader has collected $2m + 1$ confirmations, the network can progress [8].

2.8.3 Nakamoto Consensus

Nakamoto Consensus, also known as *proof-of-work*, is the consensus protocol used in Bitcoin [28]. It moved the concept of consensus from a deterministic framework to a probabilistic. Leader election works via a cryptographic game that is hard to compute, but easy to verify, namely finding partial pre-image collisions for the latest state of the ledger. When a participant finds a collision they can claim to be the leader of the next state update. There can exist several potential leaders at the same time. A solution to the cryptographic puzzle is also called a block discovery. All nodes accept all solutions but none of them are considered final. We also define difficulty: the more competitors that are participating a number called the difficulty is raised which determines how hard it is to solve the cryptographic game.
The moment a competitor hears of a solution, he stops working on the current problem and starts using the latest solution as the input for the next pre-image hash collision search. This way solutions to the cryptographic game are chained and built on top of one another. The state that is deemed the true one is always the one with the most accumulated difficulty, which could also be phrased as the longest chain of solutions to the cryptographic game. This way the consensus on state is probabilistic as one chain of solutions could be overtaken by another. This can happen when competitors that have better hardware or are just luckier out-compete a chain. However, if one waits, the probability that a certain block in the chain gets reverted falls sharply as more solutions are stacked on top of it.

Safety and Liveness

Safety in Nakamoto consensus could be defined as *only correct blocks are included*. Safety is maintained if a majority of nodes only accept valid blocks. Proof-of-works decreases the likelihood that competitors produce faulty blocks, as they have to sacrifice electricity to solve the computational puzzle. Moreover, an attacker could produce an alternate chain consisting of valid blocks that is longer than the current chain. This would destroy safety as the attacker could overwrite blocks that are considered committed. The probability that an attacker can revert a specific block decreases as the chain grows. Nakamoto [28] argues that blocks will arrive according to a *Possion Process*. In effect, this means that block discovery is a non-stateful process. The more competitors that participate in the block discovery process the harder it will be for the malicious actor to discover a block. As the chain grows from the block the attacker wants to revert, they have to solve an ever longer sequence of puzzles to arrive at a chain that can replace the legitimate chain. One can consider a block to be valid once enough blocks have been stacked on top of it.
Liveness is maintained because any entity can participate in the puzzle. If one competitor halts, the network can rely on other competitors to progress the chain of blocks.

2.9 Blockchain Datastructure

The blockchain is the core data structure for distributed systems where immutability is of importance [28]. It is called blockchain because blocks of data are chained using cryptographic hashes. The blockchain can be illustrated as in Figure 2.3. We can compute the hash of the last block N which contains a hash of block $(N-1)$, which in turn contains a hash of the block $(N-2)$ and so on until we reach the block $(N-N)$, called the genesis block. If a single bit has changed in any of the blocks the whole hashchain will diverge, making it easy to detect that a block has been corrupted.

2.10 Bitcoin

Bitcoin is probably the most well-known blockchain-based distributed system [28]. Its application is money. That is, the distributed system facilitates a token that users can transact. Its consensus mechanism along with the blockchain data struc-

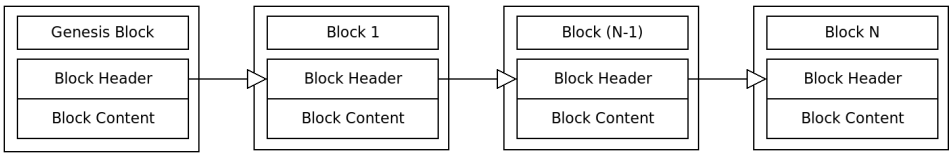


Figure 2.3: Blockchain Structure

ture is designed to solve the *double-spend problem* [28]. A double-spend is when some entity spends a resource more than once. Double-spends breaks a monetary system as one can spend one unit of money several times.
Bitcoin uses a blockchain to track the history of its tokens, making it tractable to maintain immutability of transactions. To protect the integrity of the blockchain, *Nakamoto consensus* is used. The competitors in the game are usually called miners. In order to incentivize people to compete in the game, a reward is given for each solution that is found to the winner. The winner of the competition gets to write the next block in the blockchain. This way, the winning miner gets to select the transactions that are added to each block. The idea is that the reward one can receive from competing in the Nakamoto consensus game will incentivize people to partake in the competition. This means two things for the stability of the network.

- People who can profit from attacking the network might as well join the mining pool to receive rewards.
- As more computers join the global mining pool, difficulty gets higher and it gets harder to out-compete honest miners.

These two steps work in a virtuous cycle. The more miners join, the harder it is to attack the network. The harder it is to attack the network, the more confidence people will have in it. So far this cycle has worked and Bitcoin has been running with virtually no down-time for more than 10 years. It remains to be seen if this consensus protocol will be as stable in the future.

2.11 Cryptography

This section discusses the cryptographic components that Fabric utilizes.

2.11.1 Elliptic Curve Digital Signatures

A digital signature is a scheme that provides authenticity and integrity of a message. Given a private key, an actor can sign a message. If the message is tampered with, the signature will be rendered invalid. Moreover, the signature serves as proof that the message was created by the owner of the private key. Digital signatures can be constructed and verified using *elliptic curve cryptography* (ECC) more specifically, *elliptic curve digital signature algorithms* (ECDSA) [25].

Elliptic Curve Cryptography

An elliptic curve can be used to construct a commutative group over a field. A cryptographic elliptical curve is defined as follows:

$$y^2 = x^3 + ax + b \; mod \; p \tag{2.1}$$

where $y, x \in \mathbb{F}_p$, $\mathbb{F}_p$ denotes a finite field, and p is prime. We define the group E as:

$$\{(x, y) : y^2 = x^3 + ax + b \cup \{\mathcal{O}\}\} \tag{2.2}$$

where $\mathcal{O}$ denotes a point at infinity. If the parameters for the curve are selected appropriately we can define a scalar multiplication for the group as follows:

$$nP = R \tag{2.3}$$

where P and $R \in E$. Moreover, if the parameters have been selected appropriately it becomes intractable to find n such that $nP = R$. This is what is known as the *elliptic curve discrete logarithm problem* (ECDLP). This property is used to construct the signature scheme [25].

Elliptic Curve Digital Signature Algorithm

Let d_a be the secret key of Alice. Let H_a be her public key. Alice wishes to send a message m and have it signed by her private key and verified by her public key. The message itself is not part of the signature. Instead a truncated *hash* is calculated from the message m. The hash is truncated by taking leftmost bits to bitlength n, where n is the order of the subgroup. We denote the truncated hash z. Finally, G is the base point of the elliptic curve that generates the cyclic subgroup. To construct a signature Alice uses algorithm 6. To verify Alice's signature, Bob uses algorithm 7. Note that $P = v_1G + v_2H_a$ is based on the fact that the inverse for the scalar multiplication is intractable given appropriate parameters [25].

Algorithm 6: ECDSA Signature Generation

Result: Signature: (r, s)
Select random $k \in \{1, ..., n\}$
$P \leftarrow kG$
$r \leftarrow x_p \ mod \ n$ where x_p is x-coordinate of P
if $r = 0$ **then**
  Start over
end
$s = k^{-1}(z + rd_a) \ mod \ n$
if $s = 0$ **then**
  Start over
end
return (r, s)

Algorithm 7: ECDSA Signature Verification

Result: Valid Signature?
$v_1 \leftarrow s^{-1}z$
$v_2 \leftarrow s^{-1}r$
$P \leftarrow v_1G + v_2H_a$
if $r = x_p \ mod \ n$ **then**
  return *True*
else
  return *False*
end

2.11.2 Hash Functions

A cryptographic hash function is a function whose image looks random. Using hashes one can construct a fingerprint without revealing the data itself. Formally, they can be defined as follows:

$$h : D \to M \tag{2.4}$$

where $D = \{0,1\}^*$ and $M = \{0,1\}^n$. "$*$" Indicates that the string can be of arbitrary, countable length [40].
For a hash function to be secure it needs to be a *highly non-invertible mapping*. *Highly* in this case is subjective, but it should be of such a degree that one will have a hard time finding collisions even if one has access to powerful computers. These security properties can be further refined as [40]:

1. *Pre-image Resistance* - Given an output: $y \in M$ from h, it should be hard to find corresponding input $x \in D$ such that $h(x) = y$.

2. *Second Pre-image Resistance* - Given an input: $x_1 \in D$, it should be hard to find another input $x_2 \in D$ such that $h(x_1) = h(x_2)$.

3. *Collision Resistance* - It should be hard to find inputs x_1 and x_2 such that $h(x_1) = h(x_2)$.

A function that has these properties can be considered a cryptographically secure hash function. As of version 1.4, Fabric uses *Secure Hash Algorithm 256* (Sha256) [19]. *Sha256* takes an arbitrary lengthed sequence of bytes and outputs a sequence of length 256 bits.

2.12 Digital Certificates

Digital certificates aim to solve the problem of identity when using computers. Even though digital signatures can prove that a message originated from a certain actor, there is no way of knowing who this actor is. Digital certificates solves this problem by binding a public key to a digital document. This document, in turn, is signed by an entity called a *Certificate Authority* (CA), which is known to give out credible certificates [19].

2.12.1 Certificate Authorities and Public Key Infrastructure

A *Public Key Infrastructure* (PKI) is a set of roles, policies, hardware, software and procedures needed to create, manage, distribute, use, store and revoke digital certificates and manage public-key encryption. CAs issue new certificates that are added to the PKI. Moreover, they also keep track of revoked certificates. Certificates can be revoked if some actor misbehaves, or if they lose a corresponding private key to a public key [19].

2.12.2 Root CAs and Intermediate CAs

A root CA is an entity that issues self-signed certificates. Participants that use the PKI trust certificates that are signed by root CAs. This makes them security-critical. If a root CA gets compromised it will be impossible for any participant to trust any signed certificate. An intermediate CA is a CA that has received a certificate signed by a root CA or another intermediate CA. They can in turn issue certificates signed by themselves. This way one can establish a chain of trust over several CAs, making it easier to maintain the PKI. For example, some organization could facilitate a root CA that has provided certificates to an intermediate CA whose only responsibility is to issue certificates to a particular subdivision of the organization. This way certificate handling can be partitioned [19].

2.12.3 X.509 Certificates

Hyperledger Fabric uses *X.509* certificates. In short, a certificate is nothing more than information about an entity, a public key and some meta-data that is signed by some entity. An example certificate that Fabric could use is given in Listing 2.1. This certificate contains information about an entity named *orderer.example.com*, belonging to an organization called *example.com*. The certificate has a public key, that serves as an identifier for a message sent by *orderer.example.com*. The certificate is signed by another entity found under the tag *Authority Key Identifier*. If one trusts this identity to be a legitimate authority of certificates, one can trust that messages signed by *orderer.example.com* to originate from this actor. The certificate is tamper-proof thanks to the signature at the bottom of the document. If anyone attempts to change the certificate the signature would be rendered invalid.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            0b:64:c4:5d:fa:9f:a0:f1:d0:d3:98:9e:10:c7:e7:2b
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: C = US, ST = California, L = San Francisco, O = example.com, CN
    = ca.example.com
        Validity
            Not Before: Nov 20 10:16:00 2019 GMT
            Not After : Nov 17 10:16:00 2029 GMT
        Subject: C = US, ST = California, L = San Francisco, CN = orderer.
    example.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    Omitted
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Authority Key Identifier:
                keyid:AD:B8:42:BE:F7:5F:03:D9:C6:DD:AF:89:AC:1D:60:40:15:38:80:
    F0:C7:49:45:C6:EC:FB:1E:89:6C:C9:2C:11

    Signature Algorithm: ecdsa-with-SHA256
         Omitted
```

Listing 2.1: Example X.509 Certificate

2.13 Chemical Identifiers

A chemical identifier is a string that denotes some chemical substance [18]. These chemical identifiers are essential in providing a standardized way of describing molecules. A chemical identifier has to have the following properties [18]:

1. The identifier must not be ambiguous.
2. Different compounds must have different labels.

A single compound can have several identifiers. The labels discussed here are *linear notations* [18]. That is, the structure of the labels do not have geometric components and can be described on a single line. This makes them ideal for digital representation. Mainly, there are three identifiers we consider: *InChi*, *InChiKey* and *Smiles*. The reason these three are considered is because they are a common way to represent molecules and are ubiquitous in the pharmaceutical industry [31].

2.13.1 InChi

The *International Chemical Identifier* (InChi) is a chemical identifier developed by *International Union of Pure and Applied Chemistry* (IUPAC) [18]. It is a linear string representation of substances. An InChi starts with an identifier *InChi=*, followed by a version identifier *1S*, in which *S* denotes that it is a standard representation. We only consider standard InChi. Inchi labels consist of several layers and sub-layers which are separated by slashes. Each layer, except for the first one are identified by a single character identifier. The layers are the following:

1. *Main Layer* - Parent Structure
   (a) *Chemical Formula*: The only sublayer without an identifier. It is mandatory. It denotes the atoms of the substance. The ordering of the atoms are carbon, hydrogen, followed by remaining atoms in alphabetical order.
   (b) *Skeletal Connections Layer (with prefix c)*: Denotes how non-hydrogens are connected with bonds.
   (c) *Hydrogen Layer (with prefix h)*: Denotes how hydrogens are connected to other atoms.
2. *Charge Layer*: Denotes charge information
   (a) *Protonation Sublayer (with prefix f)*: Denotes which atoms have been protonized or de-protonized. That is, which atoms have protons removed or added.
   (b) *Charge Sublayer (with prefix q)*: Denotes the total charge of the parent structure.
3. *Stereochemical Layer*: Denotes stereo information i.e. geometric information about the compound.
4. *Isotopic Layer*: Denotes isotopic information i.e. atoms that differ in number of neutrons.

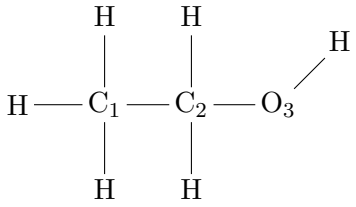


Figure 2.4: Ethanol. Main layer atoms are enumerated 1 through 3.

We only consider the main layer in this thesis.
In order to illustrate how InChi works an example is provided. InChi=1S/C2H60 is the most basic way to represent the *Ethanol* molecule. Ethanol consists of two carbon, six hydrogen and one oxygen atom. The string does not reveal anything about the structure of the molecule. To annotate the structure, the molecule is labeled InChI=1S/C2H6O/c1-2-3/h3H,2H2,1H3. The two carbon atoms are enumerated 1 and 2 respectively, and the oxygen is enumerated 3. "c" represents the skeleton structure. *1-2-3* indicates that atom 1 has a connection to atom 2, and atom 2 has a connection to atom three, respectively. *h3H,2H2,1H3* denotes that atom 3 has one hydrogen, atom 2 has 2 hydrogen and atom 1 has three hydrogen. Figure 2.4 depicts a two-dimensional rendering of the molecule where the atoms have been enumerated. There exist procedures to canonically enumerate the atoms and construct the InChi string for molecules [18]. They are not included in this thesis.

2.13.2 InChiKey

An *InChiKey* is a hash of an *InChi* string. InChiKeys are helpful when indexing molecules in a database. If one knows an InChi string or a specific InChi representation of a sublayer, an InChiKey can be derived. This InChiKey can be used to search in the database [18]. It is a 27 long character string constructed by hashing the InChi string, using *Sha256*. The InChiKey consists of three sections separated by dashes. The first 14 characters are derived by hashing the *main layer*. The first 8 characters in the second section are derived by hashing all layers, except for the main layer and charge layer. The last 2 characters of the second section denote if the InChi string was standard (S) or normal (N) and the version of the InChiKey. *A* denotes it is version 1. Finally, the last character denotes protonation, which is not considered in this thesis. The InChiKey structure can be depicted as follows:

$$\underbrace{XXXXXXXXXXXXXX}_{\text{Hash of main layer}}-\underbrace{YYYYYYYY}_{\text{Hash of other layers}}\ \underbrace{S/N}_{\text{Standard}}\ \underbrace{A/B...}_{\text{Version}}-\underbrace{Z}_{\text{Protonation}}$$

2.13.3 SMILES

SMILES is another linear representation of molecules. *SMILES* is easier for humans to read than InChi, whereas InChi is easier for computer programs to interpret. *SMILES* works as following [48]:

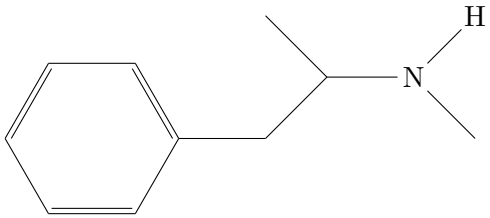


Figure 2.5: Methamphetamine. The representation is hydrogen repressed. All unmarked nodes are carbon atoms.

1. Atoms are represented by their atomic symbols. Capital letters refer to non-aromatic atoms and lower-case letters refer to aromatic atoms. Bonds are represented by - (single bond), = (double bond), # (triple bond), * (aromatic bond) and . represents a disconnected structure. If no bond is specified, single bonds are interpreted as default. For example, *c#c* represents two carbon atoms that are connected by a triple bond.
2. Atoms that can be interpreted ambiguously are surrounded by *[]*. For example, *Sc* denotes a Sulfur connected with a single bond to an aromatic carbon and *[Sc]* denotes scandium.
3. *SMILES* is hydrogen-repressed, meaning that, hydrogens are omitted when there is no risk of ambiguity. For example, *c* represents a single carbon that is connected to four hydrogens.
4. Parentheses are used to construct branches. The bond of the branch is placed leftmost inside of the parentheses. For example, *CC(=O)C* represents a carbon chain where the second carbon has a double bond to an oxygen.
5. Ring structures are constructed by surrounding the ring with 1. For example, *c1ccccc1* represents *Benzene*. If rings are nested the following rings are constructed by enumerating the number. For example, *c1cc2ccccc2cc1* represents a carbon structure with two nested rings.
6. Charge is denoted by surrounding atoms with -/+. For example, C- denotes a carbon with one electron missing.

An example SMILES string for the methamphetamine molecule is provided. It looks as follows: *CC(CC1=CC=CC=C1)NC*. It is depicted in Figure 2.5. It consists of two carbon atoms connected to a carbon ring on one side and a nitrogen on the other side. The carbon atoms in the ring are connected by double bonds. All other bonds are single bonds. Here one also sees that the representation is hydrogen repressed.

Chapter 3
Hyperledger Fabric

3.1 Overview

A Fabric network consists of organizations that contain entities. Each entity has a unique identity. The following entities and artifacts are defined for Fabric [2]:

Entities

- *Peer*: Maintains the ledgers. There exist two types of peers: endorsing and non-endorsing peers. Endorsing peers maintain ledgers and have capabilities to endorse transactions on behalf of clients. Non-endorsing peers only maintain ledgers.
- *Orderer*: Maintains the integrity of the network. They build blocks from endorsed transactions, establish consensus amongst themselves, and send blocks to peers.
- *Client*: Any program that connects to the Fabric network to execute transactions or query information.
- *Certificate Authority*: Any entity that can issue certificates for a Fabric network.

Artifacts

- *Ledger*: Each channel contains one ledger. Ledger is a synonym for a blockchain. It contains the state of every chaincode that is installed on that channel.
- *Chaincode*: Chaincode is known as smart contracts in other blockchain systems. Chaincode has the ability to update and read data from the ledgers.
- *Channel*: A channel is a partition of the state of the network. Different entities can establish different channels amongst themselves in order to hide data from entities that are not part of that channel.
- *Membership Service Provider (MSP)*: An interface that abstracts away details of identity handling and cryptography. An MSP is a module that lives inside peers and orderers. There will be at least one MSP in each peer and orderer.

3.1.1 Transactions

Transactions are client-invoked commands that update the state of the chaincode. This section gives an overview on how they work. They are described in detail in section 3.5. Fabric uses a transaction execution paradigm called *Execute-Order-Validate*, depicted in Figure 3.1. On a high level, it works as follows [2]:

1. A client sends a transaction proposal to one or more endorsing peers.
2. The endorsing peers simulate, verify and sign the transaction proposal, but do not update any state. The client collects the signed results from the endorsing peers.
3. The client sends the endorsed result to the ordering service. The ordering service takes any transactions it receives and orders them in a block. This establishes a total order of transactions.
4. The orderer sends the block to all peers of that channel. The peers validate the transactions and append the block to the ledger.

Execute-Order-Validate solves two problems identified in the so-called *Order-Execute* execution paradigm [2]. *Order-Execute* is a transaction execution paradigm where transactions are totally ordered before they are executed. This paradigm is used in e.g. Ethereum. First, if transactions have to be totally ordered before they are executed, they can not be executed in parallel. Using *Execute-Order-Validate*, transactions can be executed in parallel as they are executed before they are ordered. Any transactions that are invalid due to invalid signatures or double spends are marked as invalid and do not update state. Second, because orderers are agnostic about the validity of transactions, the execution does not have to be totally deterministic. This way one can use any programming language as is when writing chaincode for Fabric as long as a proper interface is defined for the chaincode environment.

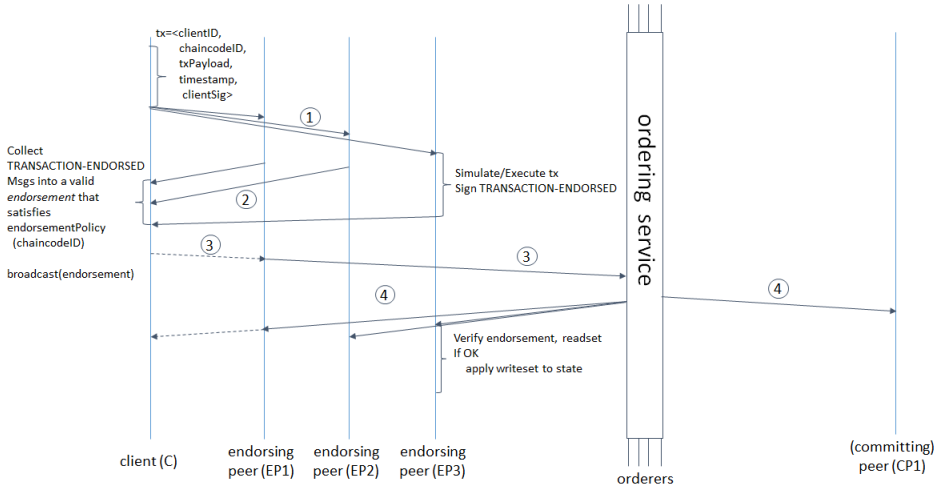


Figure 3.1: Transaction Flow [2]

3.2 State modeling

The *world state* is a versioned key/value abstraction that chaincode uses to reason about state further defined below. The world state is determined by the blockchain that lives inside peers. Figure 3.2 depicts how the world state and blockchain are tied together. There exist several states of the world: one for each chaincode that is deployed. Formally, state in Fabric is modeled as a *versioned key/value store* [19]. The state s is defined as the mapping $K \rightarrow (V \times N)$, where K is a set of keys, V is a set of values, N is a countably infinite set of ordered values, and $s(k) = (v, n)$. We define the following variables: $k \in K$, $v \in V$ and $n \in N$. Two empty types, $\perp \in N$ and $\perp \in V$ are also defined. When a *world state* is initiated, all k are mapped to $(\perp \times \perp)$. We also denote $s(k).value = v$ and $s(k).version = n$. The operations that are defined on the *key/value store* are defined as follows:

$$\begin{aligned} &put_s(k, v) \text{ updates state } s \rightarrow s' \text{ such that } s'(k) = (v, next(n)) \\ &get_s(k) = s(k) \end{aligned} \tag{3.1}$$

Only the key referenced in $put_s(k, v)$ is updated: for all $k' \neq k$ we will have $s'(k') = s(k')$.

3.2.1 Read/Write-Sets

When transactions are simulated a so called *read/write-set* is generated [19]. The read-set contains all keys referenced by the transaction with corresponding version numbers. The write-set contains all referenced keys that are to be updated in the world state [19]. The read-set prevents double spending: If a key is updated using the write-set in a transaction, any following reference to that key of the same version will be invalid. The read-set is defined as follows:

$$\{(k_i, n_i) \mid k_i \in K,\ n_i \in N\}. \tag{3.2}$$

An example *read-set* is $\{(k_1, n_1), (k_2, n_3), (k_3, n_1)\}$ that contains three keys. Two of the keys are of version n_1 and one key is of version n_3. The write-set is defined as follows:

$$\{(k_i, v_i) \mid k_i \in K,\ v_i \in V\} \tag{3.3}$$

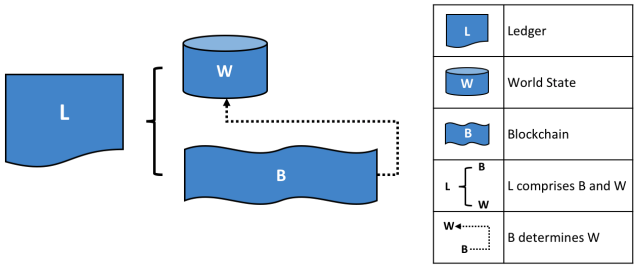


Figure 3.2: World State [19]

An example write-set could look like $\{(k_1, v_1), (k_2, v_2), (k_3, v_3)\}$ which contains the keys k_1, k_2, k_3 and aims to update them to v_1, v_2 and v_3, respectively.
An example is provided to illustrate how read/write sets prevent double spending. Define a world state as following: $\{(k_1, n_1, v_0), (k_2, n_3, v_0)\}$. There exist two keys, one of version n_1 and another of version n_3. They both map to the value v_0. Consider the two transactions T_1 and T_2 defined as follows:

$$T_1 = get_s(k_1);\ put_s(k_2, v_1)$$
$$T_2 = put_s(k_1, v_1)$$

Both of these transactions would succeed. T_1 reads k_1 and then writes to k_2. Since different keys are referenced there is no conflict. T_2 does not read any keys, and consequently, succeeds. Consider the transactions T_3 and T_4 defined as follows:

$$T_3 = put_s(k_1, v_1)$$
$$T_4 = get_s(k_1);\ put_s(k_2, v_1)$$

Since T_3 updates the value at k_1, its version will now be $next(n_1)$. T_4 has a read-set that stated that k_1 is of version n_1. If this key was to be updated that would be akin to a double spend and thus the transaction is invalid [19]. The client that proposed the transaction is free to make a new transaction proposal where the *read-set* has been updated to reflect the current state of the world.

3.3 Peer Gossip

A *gossip protocol* is used to disseminate data between peers. Peers gossip data to a random set of peers connected to the same channel. The number of peers they broadcast messages to is configurable. The gossip protocol provides the following primary functions [19]:

- It manages peer discovery and channel memberships. By using gossip, peers can keep track of which other peers are online and their identities.
- It disseminates data across the network. Any peer that is in a bad state can request data from other peers to synchronize itself with the network.
- It enables newly connected peers to retrieve configuration and state from other peers that are up to date.

3.3.1 Leader Election

In each organization, a leader peer is elected. The leader peer will be responsible for maintaining a connection with the ordering service. It retrieves new blocks that are created by the ordering service and disseminates them to other peers in their organization. This way, one conserves bandwidth to the ordering service as not all peers have to maintain a connection to it [19]. Leaders can be elected in two ways:

- *Dynamically* - The peers in the organization vote on which peer should be leader. With dynamic election, an elected leader periodically sends heartbeat-messages to its followers.

- *Statically* - A peer is manually configured to be leader. Static election is done via configuration.

If there is a network partition, the two partitions can each elect a leader which both maintain a connection to the ordering service. When the partition is mended, one of the leaders revokes their leadership and becomes a follower.

3.3.2 Anchor Peers

Peers normally need to send messages to peers in other organizations. Anchor peers allow peers to gossip across organizations. Other peers can then connect to the anchor peer to learn about other peers in the network [19]. It is recommended that each organization has at least one anchor peer. The anchor peers should be configured to be highly available. If a new peer joins the network it needs to learn about the other peers in the network in order to participate. If anchor peers are offline this can hinder discoverability of other peers [19].

3.4 Orderers

Orderers have two operations defined on them [19]:

- $Broadcast(blob)$: Broadcast *blob* to a channel. Invoked by clients.
- $Deliver(SeqNo, prevhash, blob)$: Sends *SeqNo*, *prevhash* and *blob* to peers on channel. Invoked by orderers when they have a blob to deliver to a peer.

The parameters are defined as follows: *blob* is an arbitrary message, *SeqNo* is a monotonically increasing sequence number and *prevhash* is defined as: $hash(SeqNo_{i-1}||prevhash_{i-1}||blob_{i-1})$. *Deliver* invocations are ordered. We define the current *Deliver* as $Deliver_i$ and the previous as $Deliver_{(i-1)}$ If $i = 1$ there does not exist a corresponding *Deliver* for $(i - 1)$. Therefore, each channel has to be bootstrapped with a genesis block. In the case of *Broadcast*, it is common for *blob* to be an endorsed transaction proposal, and for *Deliver*, a block of ordered transactions. When *Broadcast* is invoked, a subsequent *Deliver* will be invoked in order to deliver a block to relevant peers. The process can be illustrated as:

$$\begin{aligned} &Client \rightarrow Broadcast \rightarrow Orderer \\ &Orderer \rightarrow Deliver \rightarrow Peer \end{aligned} \tag{3.4}$$

Properties of the Ordering Service

Using *Broadcast* and *Deliver* the two following properties of the ordering service can be defined [19]:

1. *Safety*: Orderers are the entities that maintain the integrity of the network. The following properties are enabled by a correct ordering service:
   (a) *Agreement*: For all $Deliver_i$ and all corresponding $SeqNo_i$, all correct peers will eventually retrieve the same $prevhash_i$ and $blob_i$.

(b) *Hashchain Integrity*: For all pairs of $Deliver_i$ and $Deliver_{(i-1)}$, a hashchain can be constructed using *prevhash*. The hashchain can be used to verify integrity.

(c) *No skipping*: For all pairs of $Deliver_i$ and $Deliver_{(i-1)}$, if $Deliver_i$ was invoked then $Deliver_{(i-1)}$ was already invoked.

(d) *No creation*: Any *Deliver* that is delivered at a correct peer is preceded by a *Broadcast*.

Combining these properties, correct peers that communicate with the ordering service will retrieve the same information. Safety is dependent on the correctness of the orderers, if they lose their ability to maintain integrity, the whole network loses it.

2. *Liveness*: Liveness depends on the consensus protocol implemented for the orderers. Using a byzantine fault tolerant protocol liveness can be guaranteed if no more than $\frac{n-1}{3}$ Orderers are byzantine [27]. If Raft is used then liveness can be maintained if at least half of the orderers are available and correct [32].

3.5 Transactions

Transactions change the state of the ledgers. A client proposes a transaction to endorsing peers as determined by the chaincode policy. The endorsing peers endorse the transaction, given that the client has the appropriate authorization. When the client has collected enough endorsed transaction proposals, it sends the endorsed transaction to the ordering service which orders transactions into blocks. When a block is ready it is delivered to peers on that channel which in turn update their ledgers accordingly [2]. This section describes the transaction flow in detail.

Transaction Proposal

Propose is a transaction proposal defined as:

$$Propose = (PROPOSE, tx, [anchor]) \tag{3.5}$$

in which $PROPOSE$ is a fixed string, *Anchor* is an optional element that specifies version numbers of read keys, i.e. the transaction gets anchored to a specific version of keys and tx is defined as:

$$tx = (clientID, chaincodeID, txPayload, timestamp, clientSig) \tag{3.6}$$

where *clientID* is the ID of the client, *chaincodeID* is the ID of the chaincode that is to be invoked, *txPayload* is the content of the transaction as defined further below, *timestamp* is a positive integer that denotes the time when the transaction was created, and *clientSig* is the signature from the client.
A transaction proposal can be of two types: *Invoke-transaction* or *Deploy-Transaction*. An invoke-transaction invokes an existing chaincode. For an invoke-transaction, the $txPayload$ is defined as $(operation, metadata)$. *Operation* is the name of

the function inside of the chaincode, as well as corresponding parameters to that function. A deploy-transaction proposes a deployment of a new chaincode, where $txPayload$ is defined as $(source, metadata, policies)$. $Source$ is the source-code of the chaincode and policies specifies the endorsement policy for that chaincode [19]. Transactions are identified by $txID = hash(tx)$ [19].
The client sends the transaction to one or more endorsing peers, depending on the policy of the chaincode. If the client aims to install new chaincode, it needs to send the transaction proposal to at least the number of peers that are required to sign the installation, defined by the channel configuration [19].

Transaction Simulation

When the endorsing peers receive the transaction proposal they simulate it. This means that they execute the transaction *but no state is updated* [2]. Specifically, when an endorsing peer receives a $Propose$, it first verifies the signature. If the signature is valid and the client has the correct authorization, the peer simulates the transaction by executing the requested chaincode specified by $chaincodeID$ using what was defined in $txPayload$. The result of this simulation is a *read/write-set* that defines which keys in the chaincode world state were affected [19]. Then, the peer endorses the transaction and constructs a message defined as [19]:

$$(ENDORSED, txID, txProposal, epSig) \tag{3.7}$$

in which: *ENDORSED* is a fixed string, *txID* is the identifier of the transaction, and *epSig* is the signature from the endorser. Finally, *txProposal* is defined as:

$$txProposal = (epID, txID, chaincodeID, txContentBlob, readset, writeset) \tag{3.8}$$

where: *epID* is the identifier of the *Endorsing Peer*, *txID* is the identifier of the transaction, *chaincodeId* is the identifier of the chaincode, *txContentBlob* is a representation of the transaction proposal, *readset* is the readset from transaction execution, and *writeset* is the writeset from transaction execution. The endorsement message is sent back to the client that proposed the transaction.

Broadcast of Endorsed Transaction

The client collects enough endorsed transaction proposals. *Enough* is defined by the chaincode's endorsement policy. The client invokes $Broadcast(blob)$ on the ordering service, where *blob* is the set of endorsed transaction proposals that the client has collected from endorsing peers. It can invoke $Broadcast$ in two ways: either by directly sending it to an orderer or by proxying it via some other entity such as a Peer [19].

Delivery of Blocks

The orderers order transaction proposals into blocks to establish a total ordering of transactions and then deliver the blocks to relevant peers [2]. When a peer receives a block of transactions via $Deliver(SeqNo, prevhash, blob = Block)$, it executes the following to validate the transactions in the block:

1. If state updates have been applied for all $SeqNo_i$ up to $i-1$, the peer applies a state update for $SeqNo_i$. If not, the peer queries the ordering service or other peers for any missing *Deliver* [19].
2. Once a peer is ready to update its state for $SeqNo_i$, it validates the endorsement field of the transactions in the block. It compares the endorsement policy of the referenced chaincode and the signed endorsements of each transaction. If the endorsement is not valid, the transaction is added to the ledger but marked as invalid and no further actions are taken.
3. If the endorsement is valid the peer validates the read-set of the transaction proposal. If a conflict is detected the transaction is marked as invalid and no further steps are taken. If there are no conflicts in the read-set the transaction is marked as valid and the state of the world is updated using the write-set [19].

In order to keep track of valid and invalid transactions in a block, a bit-mask is utilized. An invalid transaction is marked with 0 and a valid one is marked as 1 [19].

3.6 Chaincode

Chaincode is a program with a pre-defined interface that is run inside a secured docker container [2]. A chaincode program reads and updates the world state that is determined by a ledger that resides in some channel. Chaincode drives all business logic of a Fabric network. Chaincode is usually known as smart contracts in other blockchain contexts. The difference is that chaincode is a program that can contain multiple smart contracts. In most cases, the terms can be used interchangeably.
A chaincode developer writes chaincode for some business application and deploys it to peers. Endorsement policies are determined on a chaincode by chaincode basis. Note that all peers of the channel in question still verify all transactions. Only endorsing peers of a chaincode should have the chaincode itself installed onto them [19].

3.6.1 Special Types of Chaincode

There exist five pre-defined chaincodes that are used by the network to handle low-level interactions [19]:

- *Lifecycle System Chaincode (LSCC)* - This chaincode is installed on all peers and handles packaging, signing, installing, instantiation and upgrading of chaincodes on that peer.
- *Configuration System Chaincode (CSCC)* - Installed on all peers, handles configuration transactions related to a channel.
- *Query System Chaincode (QSCC)* - Installed on all peers, handles all queries that a transaction can invoke e.g. block querying, transaction querying.

- *Endorsement System Chaincode (ESCC)* - Runs only on endorsing peers. Handles transaction-proposal endorsements.
- *Validation System Chaincode (VSCC)* - Runs on all peers, handles transaction validations, i.e. verifies endorsements on a transaction and performs read/write-set versioning.

3.6.2 Developing Chaincode using a SDK

Any programming language that has access to a *Source Development Kit* (SDK) that exposes an interface to interact with Fabric smart contracts can be used to develop Fabric chaincode [2]. In this thesis, the SDK for Golang was used. Listing 3.1 defines the most basic smart contract that one could write using *Golang* [21]. Using this structure one binds functions to the *Chaincode* struct. *Init* is called once when the contract is instantiated. *Invoke* is called every time the chaincode is invoked.

```
type Chaincode struct {
}

// Called when chaincode is instantiated
func (cc *Chaincode) Init(stub shim.ChaincodeStubInterface) sc.Response {
  fcn, params := stub.GetFunctionAndParameters()
  return shim.Success(nil)
}

// Called when chaincode is invoked
func (cc *Chaincode) Invoke(stub shim.ChaincodeStubInterface) sc.Response {
  fcn, params := stub.GetFunctionAndParameters()
  return shim.Success(nil)
}
```

Listing 3.1: Basic Chaincode

3.7 Endorsement Policies

An endorsement policy is *a condition on what endorses a transaction*. Peers have access to a set of endorsement policies that can be referenced by a *deploy-transaction*. These endorsement policies are parameterized through parameters that are defined by the *deploy-transaction*. Although possible in principle, it is forbidden to add custom endorsement policies via *deploy-transactions*. An endorsement policy has to be *bounded*, *deterministic*, *secure* and *performant* [19].

Endorsement Policies in the CLI-implementation

In order to understand Endorsement Policies, it is helpful to see how they are implemented in the *Command Line Interface* (CLI) that Fabric provides. The syntax of an endorsement policy, P, is defined as [19]:

$$P = EXPR(E[,\ E \dots]) \tag{3.9}$$

where $EXPR$ is one of the predicates: *And*, *Or*, or *n-outOf*, and E is $EXPR$ or a *principal*. A *principal* is the union of a participant's *ID* and attributes [19]. For

example, assume the following participants:

$$\{ID : Hal\ Finney,\ Attributes : (Admin, Member)\}$$
$$\{ID : Satoshi\ Nakamoto,\ Attributes : (Member)\}$$

The two principals would be:

$$\{Hal\ Finney, (Admin, Member)\}$$
$$\{Satoshi\ Nakamoto, (Member)\}$$

Two example endorsement policies P_1 and P_2 could be:

$$P_1 = Or(Admin, Satoshi\ Nakamoto)$$
$$P_2 = Or(Admin, AND(Hal\ Finney, Satoshi\ Nakamoto))$$

In P_1, either an admin or Nakamoto would have to endorse a transaction to be valid. In P_2, either an admin would have to endorse the transaction or both Finney and Nakamoto would have to endorse the transaction.

3.8 Identification

All entities in a Fabric network have identity. The identity is defined by some digital certificate and must contain two fields: *ID* and *Attributes*. ID is an identifier that uniquely identifies an entity and the attributes define what roles that entity has and to which groups it belongs to. The union of the ID and the attributes is called a *principal*. The principal defines permissions and access restrictions in a Fabric network [19].

Organizational Units

The attributes defined above defines to which organization an identity belongs. Organizations can be further divided into organizational sub-units, i.e. one could have an organization called *Org1* which has organizational sub-units *Org1-Factory* and *Org1-Sales*. This gives one the ability to partition access rights [19].

3.8.1 Membership Service Providers

The MSP is a component of Fabric that abstracts away the details of identification in the network [2]. It is possible to have more than one MSP per network. An MSP identifies which Root CAs and Intermediate CAs are trusted to define members of an organization [19]. MSPs can be of two different types, local and channel. Each peer and orderer have a locally defined MSP. A local MSP allows an entity to verify the message from the network and defines access rights when directly communicating with the entity. Channel MSPs are established when a channel is created. Each entity that is part of a channel will establish a channel MSP for that channel. The channel MSP defines identities and access rights for all members that are part of the channel [19]. An instance of the MSP contains the following items [19]:

- *Root Certificates*: A set of self-signed X.509 certificates from root CAs.
- *Intermediate Certificates*: A set of X.509 certificates from intermediary CAs.
- *Organizational Units (OU)*: A set of organizational units. Used to partition members of an organization into sub-groups, e.g. members and admins.
- *Administrators*: Identities that have elevated access.
- *Revoked Certificates*: Certificates that have been revoked.
- *Node Identity*: Contains the identity of the node that is hosting the MSP in the form of an X.509 certificate.
- *KeyStore*: Contains keys that the entity uses to sign messages.
- *TLS Root Certificate*: Set of TLS root certificates.
- *TLS Intermediate Certificate*: Set of TLS intermediate certificates.

All messages that are sent in Fabric contains the identifier of the sender. Moreover, all messages are signed by the sender. Any message that does not follow this format is invalid. Since all messages are signed, the MSP can also verify if the origin of the message is from the alleged sender of the message. Finally, the MSP can also verify if the sender of the message has the proper level of authorization by comparing the principals of the sender with whatever action the sender requested in the message. When another entity in the network receives a message, it can verify the sender of the message by passing the identifier of the sender to the MSP at the appropriate level. That is, if a message is communicated across a channel, the channel-MSP can verify the identifier of the sender. If the message is sent directly to an entity, the entity can use its local MSP to verify the id of the sender [19, 2].

3.9 TLS communication

Transport Layer Security (TLS) is a protocol that enables secure communication between applications. The protocol considers two actors: *client* and *server*. The client initiates a TLS session by contacting the server. Asymmetric cryptography, e.g. ECC, is used to establish a session key. This session key is then used to encrypt all data for the duration of the session using symmetric encryption. When the session is being set up either the server transacts its certificate to the client, or both the client and the server transact their respective certificates to each other. Using a TLS certificate, they can authenticate the other party.
All communication in Fabric can be done via TLS [19]. Fabric uses TLS version 1.2. This enables secure communication between entities. In Fabric, all entities are both TLS-servers and TLS-clients. An entity is a TLS-server when other entities connect to it and a TLS-client when it connects to other entities. If an entity has been configured to use TLS, it will reject any communication to it that is not done via TLS.

Chapter 4
Problem Analysis and Modelling

4.1 Patents and Trade Secrets in Industry

The pharmaceutical industry relies heavily on both *Patents* and *Trade Secrets* to generate value [5]. Intellectual property (IP) management deeply affects the everyday activities of businesses, as it affects their ability to protect and derive value from research [5]. A miss in the search of the prior art can mean that a product is either un-patentable or that some other company already owns a patent that will block commercialization of the developed product. As of 2011 the cost of bringing a product to market was estimated to be around $300 million to $1000 million and that it takes some 8-10 years from project inception to delivery [39]. This means that there is a substantial monetary risk to confidentiality breaches of ongoing compound development. Companies risk that their research gets unlawfully disclosed if they collaborate across organizations. If the compound becomes public knowledge it is not patentable. Furthermore, pharmaceutical companies can not rely on trade secrets alone as the world is global and highly connected. The probability that competitors reverse-engineer, or independently discover a product is non-negligible [39]. Patents are needed to protect IP. Finally, technical solutions to share information of ongoing compound development has been lacking so far [1].
These factors make cross-organizational collaboration hard. The risk of sharing information is too large. At the same time, there is a need to do so. Since the space is siloed it is likely that several actors work on the same problems. A technical solution that could enable a partial reveal of trade secrets, while maintaining a strong guarantee of confidentiality and auditability could dramatically reduce development time and costs [31].

4.2 High-Level Solution to the problems

The solution to the problem is a platform that allows partial or full reveals of intellectual property. Actors should be able to reveal properties of their intellectual property without making the intellectual property prior art. This way other actors can search for assets with certain properties. Once an actor finds an asset with certain attributes they can acquire rights to the asset. Using this asset new assets can be developed which can be partially uploaded to the platform. Moreover, if

the newly developed asset is not part of the prior art it can potentially be patented or be used for clinical trials. The rights of the asset can be acquired in potentially two ways. Either the rights can be bought outright, giving the new owner full control over the rights of the asset. Otherwise, rights can be shared. In this scenario, the original creator of the asset could be eligible to royalties if the new owner commercializes the asset. The idea is to create incentives for participants in the network to share information with each other by providing accreditation to actors that contribute molecules to the network.
Figure 4.1 provides an overview of this cycle. Consider two organizations that develop molecules. *Organization 1* has a compound that they tokenize. It is now considered an asset. They upload either the full information of the molecule or parts of it to the Fabric network (1). *Organization 2* acquires shared rights or buys the full rights to the asset (2). They use the newly acquired asset to develop a new compound which they tokenize into a new asset. *Organization 2* uploads the new asset in order to prove that they are the discoverers of the molecule (3). *Organization 1* proves that they are contributors to the original molecule via the network (4). Note that an organization does not have to upload molecules to the network. They are free to do whatever they want with their assets. However, if the accreditation mechanism is constructed correctly, actors will be incentivized to share whenever they can as there exist potential monetary gains if other organizations pay for their IP.

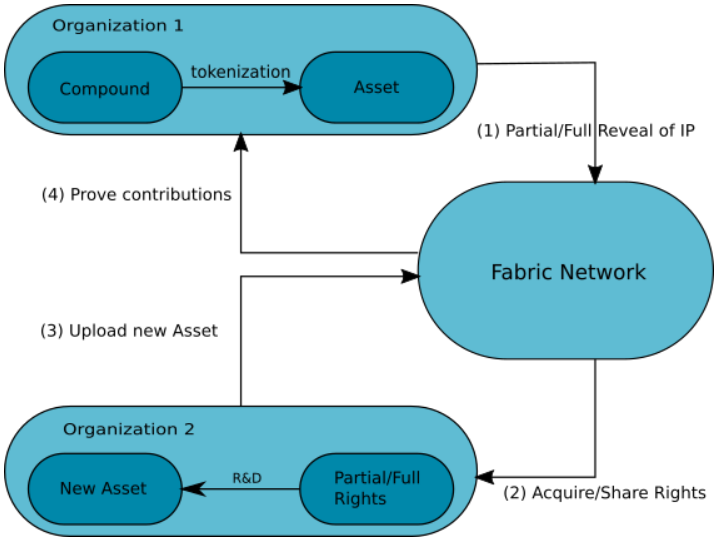


Figure 4.1: High Level Overview of Proposed Solution

4.3 Modelling

The scope of a decentralized globally available IP system for the entirety of the pharmaceutical industry is too wide for this thesis. In order to move towards

this potential solution, an agile approach has to be adopted. A minimum viable product that incorporates bare-bones functionality will be modeled in this thesis that can be used as a stepping stone to analyze how Fabric could potentially be used to build the final solution. In Chapter 8 it is discussed which future decisions have to be taken in order to fully realize a global IP system for the pharmaceutical industry. In the rest of this chapter, we discuss how molecules and entities can be modeled, and which operations the platform should provide.

4.4 Entities and Assets

We define a Fabric network as a collection of organizations. Entities are the participants of the network. An organization contains members. A member is either a peer, orderer, CA, client or a user. Each entity has an identity and a set of roles. The union of the identity and the roles is defined as the principal. Moreover, organizations establish channels amongst themselves. Each channel contains a ledger that is used to store assets. We define tokenized molecules as assets. We consider the ledger a key/value store. Each entity, channel, and network has an MSP defined at that level. At each channel, we define chaincode which contains smart contracts that drive the business behaviour of the network. For each chaincode or channel, we define an endorsement policy. The MSP takes a principal and compares it to the endorsement policy at that level to verify identity and if the identity has proper authorization. A user is anyone using the application that the platform offers via a client. Finally, an admin is a user that has access to elevated authorization for any of the members. Figure 4.2 describes pictographically how a network looks.

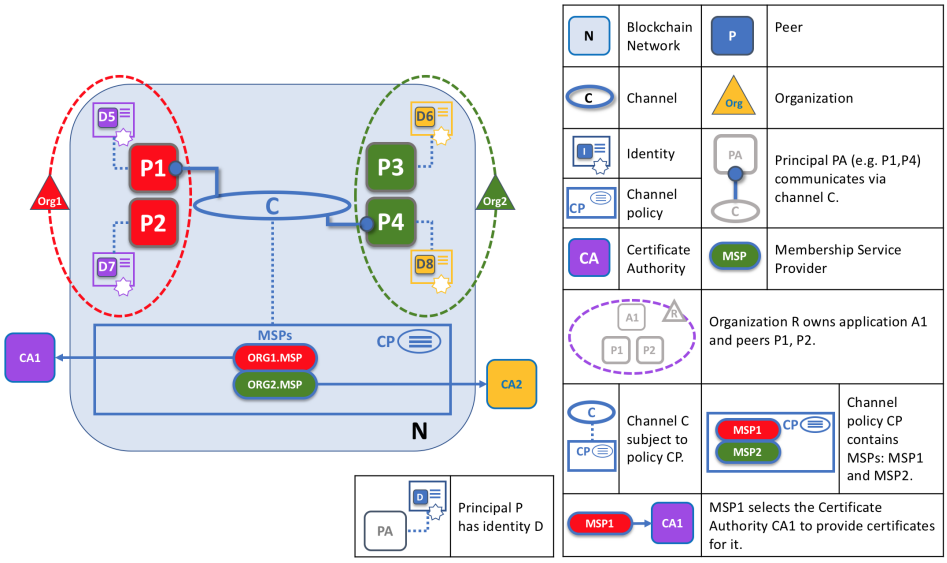


Figure 4.2: Reference Model [19]

4.5 Data format

Tokenized molecules consists of the following fields:

1. *Version (Mandatory)* - Version number of this data format.
2. *Identifier (Mandatory)* - A unique identifier for an asset. Mandatory so that each asset has an unique index.
3. *Name (Mandatory)* - A text string that represents a natural name for the asset.
4. *Synonyms (Optional)* - Other names for the same asset.
5. *Other Search Terms (Optional)* - Other terms that can be used when searching for the asset. For example, InChi key.
6. *Timestamp (Mandatory)* - A timestamp of when the asset was first uploaded. Mandatory, for non-repudiation and accountability.
7. *Asset Owner (Mandatory)* - The owner of the asset stored in the blockchain.
8. *IP Owner (Optional)* - Claimed ownership for the IP of molecule. If the owner is not known this field could be set to disputed, public domain, or unknown.
9. *Value (Optional)* - A monetary value assigned to the molecule, e.g. potential selling price in dollars.
10. *Sales Price IP (Optional)* - Selling price for the IP of the asset.
11. *Bids (Optional)* - Offers from potential buyers of rights to asset.
12. *Biological Target (Optional)* - Identifies biological functions of the molecule.
13. *Structure (Optional)* - Representation of molecule structure, i.e. *SMILES* or *InChi*.
14. *Other Data (Optional)* - Any other important data
15. *Mother Molecule (Optional)* - Index of tokenized molecule that served as inspiration for this molecule.
16. *Physical Information (Optional)* - Physical location of molecule, quantity, shipping information, and other information used to enable physical sharing of molecule.
17. *Contact Information (Optional)* - Information on whom to contact for this asset.

4.6 Operations

The following operations are defined for the platform. Operations can be added and modified if necessary.

1. *Register New User* - A user of the network with admin privilege can register a new user in the network.
2. *Update User Information* - An admin user of the network can update user information.
3. *Upload Molecule* - Tokenize a molecule according to the data format described in section 4.5 and save it in the blockchain.
4. *Search Molecule* - Search for molecules that match some query, e.g. structure, list of names, and biological targets.
5. *Update Molecule* - Update information of a molecule. Any owner of an asset can change any information. In future versions there should be configurable rights to change information of assets, e.g. multi-signature from several stake-holders.
6. *List History of Asset* - List complete history of an asset. In future versions it should be possible to retrieve only a subset of history.
7. *Transfer Ownership* - Move ownership of an asset from one user to another. In this version the owner of the asset can do this. In future versions more complex logic should be available on whom is allowed to transfer ownership. For example, several employees of a company that owns some asset might have to sign for the transfer of the ownership.

4.7 Use cases

This section describes some possible scenarios where actors use the network to handle assets.

Academic Molecule Tokenization for Educational Purposes

A student in some course could design and synthesize a possibly novel molecule. It is not known, however, if the molecule is novel. The ledger can be used to store the structure of the molecule, the creator's ID, as well as the current owner. In this case the IP could be waived, as the molecule was discovered in a course and not in an enterprise. The state of the IP for molecules is also stored in the ledger. This means that anyone can re-use the molecule that the student discovered and build upon their discovery. However, the ID of the student that discovered the molecule is forever stored in the ledger, meaning that they could prove that they were the discoverer of the molecule. Naturally, there could exist evidence of the same molecule outside of the ledger. In this case, disputes of the inventor will have to be taken off-chain.
Two operations are mainly of interest here; *Search* and *Upload Molecule*. *Search* is used by the student to verify that their discovery is indeed novel, as far as the

ledger is concerned. If they believes it is, *Upload Molecule* is used to store the molecule data on the ledger. Future actors interested in the molecule could use *Search* to retrieve the molecule discovered by the student in order to build upon her discoveries. The student themselves could also retrieve the molecule via *Search* in order to prove that they were the original discoverer of the molecule.

Organization with Unique Molecules Looking for Buyers of Intellectual Property

A confidential asset has been uploaded to the ledger by some organization. They are looking for a potential buyer of the IP of this molecule. In this scenario, the exact structure of the molecule is not uploaded to the ledger. The organization that upload the molecule sets the *IP owner* field to reflect that they own the IP. They also upload physical and chemical properties of the molecule using *biological target* and *other data*. However they do not reveal anything about the structure. The benefit of using this model lies in the fact that one can be transparent about ownership and properties of molecules without revealing the exact nature of the molecules. This also leads to increased exposure of assets to potential buyers. A potential buyer can use *Search Molecule* to find a molecule that fits their needs. Once they find a suitable molecule the owner can use *Transfer Ownership* to transfer the ownership of the asset.

Researcher or Company Seeking Funding to Further Develop an Asset

A researcher or a company uploads a molecule to the ledger similar to the previous scenario but the intent is to attract investors. Once investors are attracted, the IP of the asset could be modified to reflect whatever deal the researchers and investors agree upon. This way researchers get funding and can prove that they were indeed the researchers of the asset. The investors get provable evidence that they have a stake in the ownership of the molecule as a smart contract could update the ownership rights of the molecule to reflect the deal that researchers and investors agreed upon.
Update Molecule can be used to update the asset to reflect that there exist several owners of it by changing the *IP Owner* field. Later, the inventors of the molecule can use *List History of Asset* to prove that the original asset was created by them.

Open Source like collaboration of molecules development

A researcher that has an interest in collaborating openly on molecules could upload the full molecule to the ledger. This way anyone could retrieve the full information of the molecule in order to help research it and potentially related molecules. The owner of the asset loses the confidentiality of the molecule data and any potential patent claims. However, they can use *List History of Asset* to prove that they were the original uploader of the asset.

Ownership Transfer of Assets

The owner of the tokenized asset could be updated via a smart contract. This way IP rights could be transacted like any token. The benefit of using a blockchain for

this is that the history of ownership is always stored. This way a researcher could sell an asset, without losing the claim that they were the discoverer or researcher of the asset. In this scenario the operation *Transfer Ownership* is of interest as it can be used to transfer the owner of an asset.

Chapter 5
Implementation

An implementation of the proposed solution is provided. It is currently deployed on a cloud-based machine. The source code for the prototype is hosted on a set of private Github repositories. The prototype consists of a single peer, a single orderer, a single channel, a single chaincode and a rest-server that enables the end-user to interact with the network through a web browser. The prototype leverages *Swagger* [42] to specify an *application protocol interface* (API) for the prototype. The prototype is used to test what chaincode functionality makes sense and to analyze what future features are required to enable a platform of molecule assets. Graphically the system is depicted in Figure 5.1.

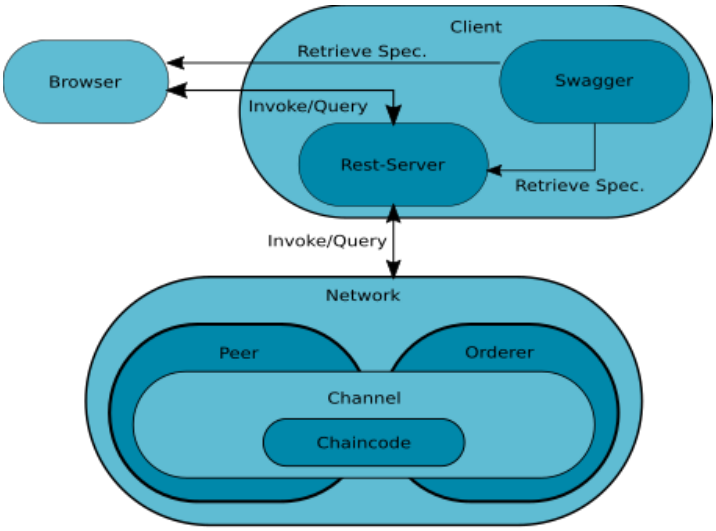


Figure 5.1: Implementation of the Prototype

5.1 Entities

The code that defines the peer, orderer and channel consist of *.yaml* and *.sh* files. They closely follow the samples that the official *fabric-samples* repository

[20] offers. These files are used to generate appropriate certificates, key pairs, configuration and docker files used to start the network. Two organizations are defined for the network. An orderer organization and a Peer organization. The Peer organization has two users defined for it: one normal user and one admin user.

5.1.1 Access Control

The access control of each member is defined by the configuration transaction that creates the network. The identity of each member is authenticated using x.509 certificates. The certificates are pre-generated before the network starts. As an effect, no CA is needed for the prototype.

5.1.2 User Model Implementation

Users of the network are partitioned into two separate models. One user type that is recognized natively by Fabric, and a user that belongs to the chaincode. From Fabric's perspective, a user is an organization member with a certificate from a legitimate source. That is, a Fabric user can invoke chaincode if they submit a transaction that is signed with the corresponding public key of the user. The user-defined in chaincode is defined as a struct containing the fields *name* and *ID*. ID is a unique identifier, and name is a natural name that is not necessarily unique. Using this model one does not need to issue a certificate for everyone that wishes to interact with the platform. An organization could elect an individual who is enrolled as a proper Fabric user. This individual could manage assets on behalf of other members of their organization by creating users in the chaincode model.

5.2 Data Format Implementation

One channel is used for the application. This way the business data is not partitioned in any way. Moreover, the underlying database-software that is used by the peer is *CouchDB* [4]. The reason CouchDB is used is because CouchDB has sophisticated search functionalities. The molecule model is implemented as a Golang struct and is defined in Listing 5.1. There also exists another database that can be used with Fabric called *LevelDB* [16]. LevelDB is more suitable if one does not need to do complex searches, as the instance of LevelDB lives inside the peers as a module. A CouchDB instance is its own separate container and as a result, the peer and the database instance have to communicate across containers.

```
type Molecule struct {
  ObjectType       string    `json:"docType"`
  Version          int       `json:"version"`
  Index            string    `json:"index"`
  Name             string    `json:"name"`
  Synonymes        []string  `json:"synonymes"`
  Submitter        string    `json:"submitter"`
  Timestamp        string    `json:"timestamp"`
  Owner            string    `json:"owner"`
  Value            int       `json:"value"`
  SalesPriceIPR    int       `json:"salesPriceIPR"`
  Bids             int       `json:"bid"`
  BiologicalTarget string    `json:"biologicalTarget"`
  Structure        string    `json:"structure"`
```

```
    Data              []string  `json:"data"`
16  MotherMolecule    string    `json:"motherMolecule"`
    PhysicalStorage   string    `json:"physicalStorage"`
18  Contact           string    `json:"contact"`
    AssetType         string    `json:"assetType"`
20  }
```

Listing 5.1: Molecule Model

5.2.1 The Chaincode

The operations themselves are defined as a chaincode that is deployed on the single peer. The chaincode is a Golang-program that leverages a *source development kit* (SDK) [19] that Fabric provides for the language. The code for the chaincode is defined in Listing A.1, available in appendix A.1.

5.3 Interacting with the application

The operations defined in Chapter 4 are implemented as an API. A user connects to the network via a web browser-based front-end generated by Swagger. Behind the front-end, a rest-server sits that proxies user requests to the peer and orderer of the network. The rest-server is a NodeJS [34] application. It leverages Express [33] to drive API calls and the Fabric SDK [22] to connect to the Fabric network. In practice, the rest server is the actual member of the peer organization. The reason it is implemented like this is due to simplicity. This way one does not need authentication mechanisms between the front-end and the rest server. During the course of the thesis, the implementation has only been run locally and thus the machine running the rest server and the front-end has been the same. The API is defined in Figure 5.2.

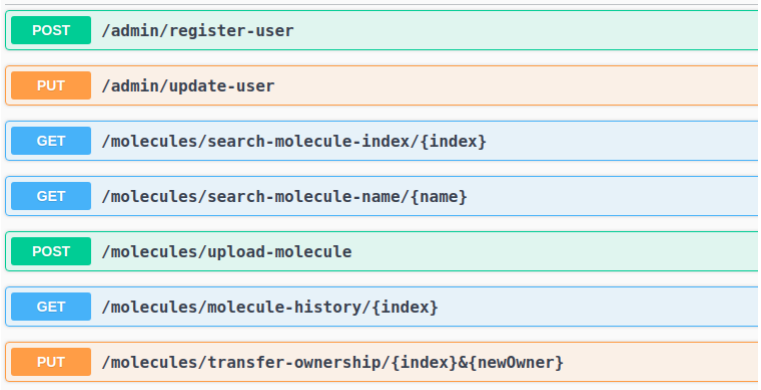


Figure 5.2: API for the Platform

Chapter 6
Security Analysis

This chapter provides an overview of security aspects of Fabric. The reference model is equivalent to the prototype model.

6.1 Information Taxonomy

Information Taxonomy defines the nature of data that is to be protected [10]. We consider tokenized molecules that are uploaded to the network as the information that is to be protected. The four information taxonomy attributes for assets are defined as follows [10]:

- *Form*: We only consider molecules in a tokenized format.
- *Sensitivity*: A molecule has two levels of sensitivity. Molecules that have been made public property need integrity protection, but only confidentiality protection from non-members of the network. Importantly, auditability has to be preserved so that the correct owner of the molecule can claim credit for it. For assets that could be patented or commercialized in the future, both confidentiality and integrity of the asset has to be protected even from other members of the network.
- *Location*: All assets can be in one of the following places: In clients, in peers and orderers. When entities send information to one another this is done so over the internet. Depending on circumstance information can be found in peers, orderers, and clients from other organizations.
- *State*: Information can be in one of the following five states: *creation, transmission, storage, processing or destruction.* Molecules can only be created once and never be destroyed. Once they are created they will be stored forever. Molecules are transmitted to peers and orderers after creation. Peers that are synchronizing their state will require that molecules are transmitted. Molecules are processed inside of peers, using chaincode.

Fabric uses the following cryptographic primitives [19]:

1. In Fabric version 1.4, *SHA256* is the only available hashing function.
2. Fabric uses *ECDSA* to create signatures.
3. The default MSP implementation uses X.509 certificates.

6.1.1 Assumptions

We make two assumptions when analyzing the security of the prototype:

1. *Participants of the network consider reputational damage*: The cost to produce faulty messages for participants is virtually zero in Fabric. In public blockchains there exists a cost to produce blocks, e.g. miners have to sacrifice computing power to generate blocks. The only cost of diverging from the Fabric protocol for an adversary that is part of the network is potential reputational damage. Therefore one has to consider whom they invite to the network. An actor that is not afraid to damage their reputation could cheat the protocol for their own gain. However, ill-behaved actors can be booted from the network.
2. *The CAs and PKI are secured*: Since all of Fabric's security is built on identities and signatures, the CAs and PKI have to be secure. How to secure PKIs and CAs can be found in other literature and is not considered in this thesis.
3. *Perfect Cryptography*: Fabric uses well known cryptographic primitives. In the security analysis, we assume that they can not be attacked.

6.2 Security Goals

This section describes the *Security Goals* for the prototype. The definitions of the goals are defined in the *Reference Model for Information Assurance & Security* (RMIAS) [10]. The *CIA* (Confidentiality, Integrity, Availability) triad is a way to define security aspects of a system [47]. Cherdantseva and Hilton [10] expand the *CIA* triad to provide a more fine-grained description of security services. Apart from already described security services they also include: *Accountability, Auditability, Authenticity, Non-repudiation* and *Privacy*. Fabric enables them in the following way:

- *Confidentiality* - Fabric is designed to be a permissioned blockchain, only invited members can participate. Given that participants protect their networked devices properly this provides a high level of confidentiality against non-members of the network. When communicating between entities, TLS can be enabled which protects the confidentiality of messages, especially when sent over insecure communication channels like the Internet [19]. Furthermore, the Fabric network can be further partitioned using channels. Any entity that is not part of the channel will be denied access. Using channels, one can hide data from entities that are part of the same network [2]. The prototype only utilizes one channel, however. Finally, Fabric has private data. Using private data one only commits the hash of the private data to the blockchain, while saving the data at a set of predefined, presumably trusted peers [19]. This way one can even hide data from entities that are part of a channel. Private data is not part of the prototype, however.
- *Integrity* - Integrity is mainly protected using the blockchain [2]. If a single byte is changed in the chain of hashes, the final hash of the chain will diverge

from its true value [28]. Assuming that the ordering service is correct, one can depend on it to only deliver legitimate blocks. Moreover, peers can verify that their state of the blockchain is the same as the rest of the network by comparing the hash of the latest block with other peers. This is best done with entities that are trusted to be correct, e.g. using peers from one's own organization. By replaying the content of the blockchain one can reconstruct the state of a chaincode. This way a peer can reconstruct state independently, only needing to trust the final hash of the blockchain [2]. Integrity is further enhanced by the need to sign every transaction because an actor that sends malicious data to the network can be tracked down using their identity. The actor could face legal consequences or be banned from the network if they sends malicious messages to the network. Clients can verify the integrity of executed chaincode by requesting that their transaction be executed at several peers. The chaincode execution produces a read/write-set for referenced keys [19]. If this set contains inconsistencies across peer executions this might point to the fact that either the chaincode is non-deterministic, or that some peers are cheating. This feature can be enforced by chaincode execution policies. Moreover, peers validate each transaction in the final step of the transaction lifecycle to make sure that the state is updated correctly, and that the read/write set has not been tampered with [2].

- *Availability* - Fabric is designed to be a distributed system [2]. This means that availability is protected by having several entities that provide the same services. One organization can deploy several peers that all can endorse transactions. If one peer crashes or gets attacked the other peers could potentially still provide the same services.

- *Accountability* - Accountability is when one can hold some actor accountable for his actions [10]. Since all transactions in Fabric are signed it is easy to hold the appropriate actor accountable given that their keys have not been compromised [2]. Transactions that lack the correct level of authorization will be blocked by the entities of the network and be logged. Consequences of malicious actions are not defined by Fabric. The owners of the network will have to determine what happens to users that are acting maliciously.

- *Auditability* - Auditability is the ability to monitor the events that have taken place inside a system. For a system to be auditable the auditability must be mandatory. Otherwise, ill-behaved actors would avoid the monitoring [10]. Fabric has strong auditability thanks to the blockchain. All transactions are stored in the blockchain, even invalid ones [2]. One can track transaction requests of authenticated users. The peers and orderers also provide logging that could optionally be saved and used to enhance auditability. The maintainers of the entities have to explicitly save and share these as they are not part of the transaction life-cycle.

- *Authenticity* - Authenticity is the ability to verify identities and how trustworthy these actors are and how trustworthy the data they present are [10]. Authenticity of data presented to the network is not a feature of Fabric.

This is due to the fact chaincode is a general computer program. Data presented to Fabric can be arbitrary. Therefore, it is up to the developers of the network to find ways to guarantee that data that gets included in the network is authentic. This is a general problem of tokenization. *How does one know that data included in a blockchain from the external world is valid?* This is further discussed in Chapter 8, as it is one of the biggest challenges identified during this thesis.

- *Non-repudiation* - Non-repudiation is the ability to prove that some actor did or did not do some action [10]. Fabric has non-repudiation. Any valid message in the network contains identity with a corresponding signature [2]. Given that the key that is used to sign for the identity has not been compromised, no actor can deny that a message did not originate from them. Moreover, an actor can prove that a message did originate from them by signing another message using the same public identity of the disputed message.
- *Privacy* - Fabric has some privacy features. First, it is a permissioned blockchain, meaning that only invited actors can participate. This means that the Fabric network can be run in a secured network. Second, each organization is responsible for enrolling members. This means that an organization can protect identities by enrolling users under whatever name they choose. However, they can not hide the fact that a member is part of their organization. Privacy of data itself is also protected by the fact that the network is permissioned. If a transaction contains personal information, the permissioned nature of the network means that privacy of the data itself can be protected from external actors. This can be further enhanced by using channels. Lastly, private data can be used to hide data on a channel. This way only a select number of peers will be able to see the private data [19].

Security countermeasures are measures taken to enforce the security goals. They can be classified into four categories according to RMIAS [10].

1. *Organizational*: Policies that organizations employ to increase the likelihood that a product is secure.
2. *Human-Oriented*: Relates to organizational culture, training, and ethics that decreases the likelihood of human faults when developing and operating security-sensitive products.
3. *Technical*: Technical countermeasures are technical mechanisms that increase the security of a product. For example, cryptography, authentication, or access control.
4. *Legal*: Legal countermeasures are countermeasures where legislation is utilized to increase the security of a product. For example, non-disclosure agreements can decrease the likelihood of confidentiality breaches. In the case of the prototype, a participant could be forced to sign a contract that they will not misuse the platform and risk facing legal measures if they do.

We only consider technical countermeasures in the security analysis.

6.2.1 Authentication and Access Control

Entities of the network are divided into organizations. An organization can contain sub-organizations. Each organization forms a trust domain. Entities trust other entities from their trust domain. It is assumed that members of the same organization will not attack each other [2]. The prototype uses the standard *MSP* implementation provided by Fabric. It uses *X.509* certificates for identification [19]. One should handle certificates as in any other software project. Any certificate authority can be used to generate certificates, given that they follow the specifications discussed in section 3.8. The *MSP* uses certificates that have been installed on it in combination with public keys and signatures included in messages to authenticate entities. Certificates of peers, orderers, and CAs have to be installed *out-of-band*. That is, one generates certificates for them using some CA and then distributes them outside of the Fabric network. Certificates of clients can be installed out-of-band or in an *online mode* [19]. Online mode, in this case, means that one can enroll clients programmatically, using the Fabric network. In order to revoke a certificate one adds the certificate to be revoked to the appropriate MSP's certificate revocation list.
Members of organizations have roles as defined in section 3.8 which define authorization of a member. An MSP uses a known channel configuration in combination with an authenticated identity to define authorization for a member on that level. Resources in Fabric are protected via access control [19]. To enable access control at the channel level, Fabric uses an *Access Control List* (ACL) to define access to resources for entities. The ACL is defined on a channel by channel basis and is configured using a *channel-configuration transaction*. A resource from the network's perspective is an endpoint that clients can interact with. For example, a smart contract is a resource. Policies are used to define which authorization is needed to access a resource. For example, $OR(Org1.member, Org2.admin)$ defines that a member of $Org1$ or an admin of $Org2$ is needed to access a resource. Moreover, access control can be defined inside chaincode by using the *GetCreator* function defined in the SDK used to develop chaincode [21]. *GetCreator* returns information on the creator of the transaction. This way one can define access control on the chaincode level.

6.3 Chaincode Security

The security analysis of chaincode can be considered separate from the underlying Fabric network. When analyzing the chaincode security we assume the underlying network to be secure. There must be a high probability that chaincode is correct and secure. If chaincode is exploitable, a potential attacker could steal from other participants. Smart contract security is an ongoing research topic. Most papers focus on Ethereum as it is the second largest blockchain measured in market capitalization, as well as it being the oldest smart contract platform.
Praitheeshan et al. [36] provided a survey on Ethereum smart contract vulnerabilities. Considering that smart contracts move value, there exists an incentive to attack them. Furthermore, the public nature of Ethereum means that anyone can participate in the network. This is mitigated by Fabric due to its permissioned

nature. Moreover, there exists no easy way to upgrade Ethereum smart contracts due to the decentralized nature of the network. Using Fabric one needs to coordinate smart contract upgrades with a smaller set of participants. They identify two key vulnerabilities that are of interest:

- *Timestamp dependent contracts*: A timestamp-dependent contract is a smart contract that uses timestamps in its execution. An entity could change time to manipulate code execution. For example, if time is an input parameter to a chaincode, a client could pass an incorrect timestamp to exploit it. One way to solve this is to avoid timestamps altogether. If timestamps are required one could rely on trusted entities to provide time, e.g. members from one's own organization. Potentially, a third party could be responsible for providing legitimate timestamps.
- *Transaction ordering problems*: Smart contract platforms are asynchronous systems. A smart contract that is dependent on another smart contract and assumes an order of execution is vulnerable to race conditions. For example, if one contract is dependent on a second contract, the state of the second contract could be updated before the first contract execution is finished. In the case of Fabric, this could be two peers that execute the first contract separately. The first peer retrieves one result from the second contract and the other peer retrieves another result. This way they have retrieved different results from the second contract and might reach different conclusions on the final state of the first contract.

There could also be general software vulnerabilities such as buffer overflows, integer overflows, command injections, etc. [36]. These are not unique to smart contracts and can exist in chaincode, as it is a general-purpose program. In order to decrease the likelihood that software contains faults *static analysis* and *formal analysis* can be utilized. Static analysis is an analysis of source code without executing the code. Static analysis can help with finding code patterns that are known to cause issues. Formal analysis is a form of static analysis that provides some proof that a program is correct according to some specification. Several papers propose rigorous development processes in order to minimize the risk of these vulnerabilities using static and formal analysis [36, 38, 6]. No formal verification tool for Golang chaincode was found. Beckert et al. [6] modified *KeY*, a formal verification tool for Java to verify correctness of Java chaincode. There also exists a suite of verification tools for Ethereum: *Oyente*, *ZEUS*, etc. [36]. A static analysis tool for Golang chaincode was found, namely, ChainSecurity's *Chaincode Scanner* [9]. It analyzes chaincode for the following potential critical issues:

1. Goroutines - Concurrency is discouraged in chaincode as it is easy to get concurrency wrong.
2. The chaincode object should not contain fields.
3. When operating on a ledger, it should not rely on global variables.
4. Certain non-deterministic libraries should not be utilized.
5. One should not iterate over maps using ranges as this operation is non-deterministic.

Non-determinism is something that could break the integrity of a blockchain, so it is helpful that the analyzer finds occurrences where non-determinism in the chaincode is likely. Moreover, the analyzer also presents less critical issues as warnings:

1. Number of arguments should be validated in each function. This way one can stop function execution if the end-user passes too few or too many arguments.
2. Read after writes introduces performance overhead. This is, one writes to a key in the ledger, only to read the same key from the ledger after. This invokes the underlying database an unnecessary amount of times.
3. Error variables should always be handled
4. Any variable that is read from the ledger should be used.

The implement chaincode found in section A.1 was scanned using ChainSecurity's chaincode scanner. None of the above-mentioned issues were found which provides some degree of certainty that the chaincode is correct.

6.3.1 Security Development Life Cycle

Since chaincode has a large attack surface, *Security Development Life Cycle* can be utilized to reduce the probability of introducing exploits when developing chaincode. *Security Development Life Cycle* is when security measures are integrated into the development of software [10], and spans all components of an organization. We only consider technical implementations that increase the likelihood of correct chaincode. The following components were developed as part of the development lifecycle to the prototype and are hosted in a set of private repositories:

1. *Automatic Build Tool* - *Buildbot*, a build tool developed in pure Python, is part of the development life cycle. Each commit to the codebase triggers a build. Each build automatically builds the code, runs tests, does static analysis and deploys the new commit. This speeds up the development process and minimizes the risk of human errors when deploying code. Moreover, since tests are run automatically commits that break the system are quickly identified.
2. *Static Code Analysis* - At each build, the chaincode is statically analyzed using *golangci-lint*. *golangci-lint* is a code linter for Golang that can uncover common errors in code. For example, it can help identify errors that are not handled.
3. *Unit Testing* - At each build, the code is unit tested using Golang's built-in test tool. If tests are developed correctly they can help detect if bugs are introduced into newly committed code.
4. *End to End Tests* - At each build, a mock network is set up locally. Using this mock network transactions are executed in order to observe if the new commit introduces faults using Hyperledger Caliper.

5. *Automatic Deployment to Test Environment* - The newly committed code is automatically deployed to a network that is designed to be as close as possible to a production environment. This network is available around the clock and provides an ideal environment where functionality can be tested.

6.4 Security Attacks

This section describes some common attacks and how they are prevented in Fabric.

6.4.1 Impersonation attacks

Assuming that no private key is compromised, any message that is sent will be signed by its creator. One can trust that messages received was not created by an adversary. If TLS is not used, it is possible to send a message to an adversary as authentication is one-way. Using TLS, the authentication becomes two-way. Moreover, all entities maintain a list of outgoing connections defined by configuration [19]. One can specify that entities only communicate with entities that are known to be part of the network. Human errors can be mitigated by using automated node discovery. Using this, connections to other entities are discovered by asking known peers for information. Malicious peers could provide connections to malicious entities. If TLS communication is enabled, then any communication with an unauthorized entity will be rejected.

6.4.2 Byzantine Peers

A substantial portion of the communication in Fabric is gossip between peers. A compromised peer has the potential to attack the network. They threaten the main security consideration in the following way:

1. *Integrity* - A byzantine peer could alter transactions that flow through it. Since transactions are signed, this would make them invalid and thus byzantine peers do not threaten the integrity of messages that are passing through them. However, the peer could sign its own message that contains data where integrity has been broken. For example, a byzantine peer could send the wrong transaction results to a client and sign it with its own keys. This can be mitigated if important transactions require signatures from several peers.
2. *Availability* - Since gossip is used, the refusal to propagate information of a byzantine peer can be mitigated by communicating with other peers. If the byzantine peer happens to be the sole leader peer in an organization, i.e. it is the only peer with a connection to the ordering service, then an election needs to be held before the availability of the attacked organization can be resumed.
3. *Confidentiality* - The worst security issue regarding a byzantine peer is probably related to confidentiality. A compromised peer could be well-behaved but disclose confidential data. Fabric provides no mechanism to detect if an entity leaks information unlawfully.

In the end, one has to ask if byzantine peers are likely. It could be the case that it is entirely reasonable to fully trust peers inside one's own organization. However, if communication happens across organizations there could be more cause for concern as peers from other organizations are potentially from a competing organization.

6.4.3 Byzantine Orderers

As of Fabric version 1.4, the network breaks if orderers are byzantine. Currently, there exists no byzantine fault-tolerant consensus protocol for the orderers [19]. Byzantine behavior of orderers could break the whole network. This is potentially a fatal security flaw for production systems. The Hyperledger Foundation is working to have a byzantine fault-tolerant protocol available as soon as possible. However, since all transactions are signed in Fabric, one could make a case that the probability of an orderer diverging from the consensus protocol is small. If an orderer is caught being malicious the owner could face legal actions. This is the case if they send malformed messages that attack liveness or safety of the network. It might be easier to hide attacks where orderers delay or refuse to deliver messages. In this case, the attacker could blame slow internet connection or disconnections while attacking the liveness of the system.

6.4.4 Replay Attacks

A replay attack is when someone replays a message that was previously sent [47]. In Fabric, transactions contain a nonce. This way each transaction will only be handled once by each entity. Moreover, since each transaction contains a versioned write-set a replayed transaction will not update any state of chaincode [19].

6.4.5 Man in the Middle Attacks

A man in the middle (MITM) attack is when a malicious actor gets hold of a message before it arrives at its destination [47]. Since Fabric messages are signed active attacks are not possible. If TLS is enabled attackers can not read the content of the message. Moreover, if an entity has been configured to use TLS it will reject all non-TLS based communication. However, traffic analysis is possible. A MITM attacker could deduce some information by analyzing which entity is sending messages even if they can not see the content of the messages.

6.4.6 Denial-of-Service (DoS) Attacks

A single node could be spammed by an attacker that has access to the network. A healthy network should have a reasonable amount of peers and orderers. This way it becomes harder to perform DoS attacks on a network as the attacker would need to attack several entities. If one entity is being attacked, honest actors could turn to other entities in order to transact their messages. A Fabric network is susceptible to denial of service attacks from clients that are part of the network. When running evaluations a client sending more transactions than the peers could commit to a ledger affected the availability of the network. One should ask if clients are likely to spam the network as their actions are logged by all peers and

orderers. However, depending on the application it could be hard to differentiate between attacks and a flood of honest transactions. A potential mitigation is to introduce some rate limit. It could be implemented in the clients, or by placing network entities behind some load balancing mechanism.

DoS mitigation using early authentication

Denial of service attacks can be mitigated by authenticating early in the communication protocols [47]. If an attacker has to reveal identity early when establishing communications this makes it easier to detect that a flood of messages is coming from the same identity. The following holds for the transaction life-cycle discussed in Chapter 3:

1. Transaction proposals are sent from a client to endorsing peers and have the following format: $(clientId, chaincodeId, txPayload, timestamp, clientSig)$. The endorsing peer verifies the $clientID$ with respect to the $clientSig$, before processing the transaction [19].
2. Endorsed transaction proposals are sent back to the client from one or more endorsing peers. Fabric documentation does not specify if the client verifies the signatures.
3. The client sends the endorsed transaction to the ordering service. According to Fabric documentation endorsed transaction messages only contain the signatures from the endorsing peers [19].
4. When the transaction is delivered to peers from the ordering service, the full transaction will be visible to the peers. This way they can always disregard any malformed transaction [19].

In step 3 of the transaction life-cycle, there is weakened authentication and could potentially suffer from a weakened denial of service protection. However, if TLS is enabled for all entities in the network early authentication is enforced throughout the network.

Other types of DoS Defenses

Other types of DoS attack protections includes: *client puzzles* which is akin to proof-of-work described in Chapter 2, protocol-fail-stop which stops the communication when bogus messages are detected, or increasingly strong authentication as communication progresses [47]. None of these strategies are implemented in Fabric version 1.4. However, since Fabric is intended to be run as a permissioned network, denial of service attacks could be mitigated via other common defense mechanisms, e.g. firewall protection where any communication from potentially malicious actors can be discarded quickly.

6.4.7 Transaction malleability

A client proxying messages to the ordering service via peers is subject to a malleability attack. This can happen in step 3 of the transaction life-cycle discussed

in Chapter 3. In this step, the client needs to send an endorsed transaction to the ordering service. If the client does not have access to a direct connection to the ordering service it must proxy it to the ordering service via a peer. The peer can not fabricate endorsements and thus non-repudiation is not at risk. However, the proxying peer can remove endorsements from the transactions, potentially making it invalid [19].

6.4.8 Sybil Attacks

A *sybil attack* is when one participant of a network generates a large number of identities in order to gain a disproportionate influence over the network [12]. In the case of Fabric that would be one malicious actor possessing several certificates that they can use to drive malicious behavior. Assuming that the certificate authority for an organization is not compromised Sybil attacks are not possible as the malicious actor can not generate their own identities. However, the actors controlling a certificate authority could generate as many identities as they like to. This could open up an attack vector potentially. Imagine a chaincode where the result of some transaction is based on votes from users. An organization could generate a disproportionate amount of certificates in order to gain an advantage in the voting process.

6.4.9 Eclipse Attacks

An *eclipse attack* happens when all incoming and outgoing connections of a victim entity are to malicious actors [12]. Consider a client that connects to three peers in order to increase the likelihood that transaction results are executed correctly. If all three of these connections happen to be to malicious peers the client has been subject to an eclipse attack. These attacks are of limited use in Fabric as all communication is done via signed messages. That is, an attacker can not fabricate and send messages to the eclipsed entity as it would reject them. Moreover, if TLS is used, the entity being attacked will not establish a communication with the attacker as the attacker's identity will not be recognized. Eclipse attacks are mitigated using configuration. One should always configure an entity to communicate with at least one trusted entity.

6.4.10 Double Spend Protection

A double spend is a set of two transactions that are in conflict with one another that both get accepted by the network [12]. It is an integrity issue. In the case of Fabric, a double spend would be two transactions that get accepted into a ledger, where the first transaction updates a key-version pair, followed by a transaction that updates the value for the same version of the key. How state is modeled was discussed in Chapter 3. In the case of molecules, this would be akin to someone transferring ownership of a molecule ($T1$), followed by another transfer of ownership of the same molecule, by the same user ($T2$). Namely:

$$T1 = put(k_1, v_1)$$
$$T2 = get(k_1);\ put(k_1, v_2)$$

This would break integrity of the network, as one user could transfer ownership of a single molecule to several users. Given a secure network, double spending is stopped by the validation step in the transaction flow. If a key in the world state is referenced after it has been updated there will exist a conflict in the read/write-set. As a result, the transaction will be marked as invalid by the peer [2].

Other Double Spend Attacks

Both Raft [32] and PBFT [8] are deterministic consensus protocols. That is, each round of consensus is final. This means that once consensus has been established for a round it will not be reverted. This prevents a class of double spend attacks found in other blockchains such as Bitcoin. Due to the probabilistic nature of *nakamoto consensus*, there exists a set of potential double spend attacks that rely on the unfinality of the consensus protocol, e.g. *finney attack*, *race attack*, *brute force attack*. [12]. In Nakamoto consensus a block that was discovered recently has a probability to be reverted as one chain could outgrow another [28]. None of these attacks are possible using Raft or PBFT due to their finality. However, one could possibly persuade peers to revert blocks, and thus enable double spending that rely on unfinality of blocks.

Chapter 7
Performance Analysis

This chapter provides a performance analysis of the prototype described in Chapter 5. We used Caliper [24] to run a performance analysis on the developed chaincode using varying network topologies. The Hyperledger Foundation put together a document describing key points when analyzing the performance of blockchain networks [35]. The key metrics defined by the document are:

1. *Read Latency*: Time from read request submission to response from the network.
2. *Read throughput*: How many reads are completed in a time period. Reads in Fabric also leverage the transaction lifecycle discussed in Chapter 3. These transactions are referred to as query transactions with the difference that they are stopped after the execute step [19].
3. *Transaction Latency*: The time from submission of a transaction to a response from the network that the transaction succeeded.
4. *Transaction throughput*: Number of committed transactions per second (tps). Note that even invalid transactions are added to the ledger in Fabric [19].

7.1 Previous Work

This section summarizes some previous work on the analysis of Fabric. The subjects discussed here can be used to guide future decisions as the prototype is developed further.

7.1.1 Performance Evaluation

Thakkar et al. [46] investigated how Fabric version 1 performed under various configurations. Following is a selection of their findings that are relevant to the future development of the prototype proposed in this thesis.

1. Transaction latency increased linearly as transaction arrival rate increased until a certain threshold was reached, namely around 140 tps. At this threshold, more transactions arrived than the verification system chaincode could

handle and thus it became the bottleneck. This means that transaction latency started increasing.

2. If transactions arrived slower than blocks were filled, an increase in block size increased the transaction latency linearly. This is due to the fact that as blocks are larger they can include more transactions. This means that transactions are held longer on average in the ordering service. However, if transactions per second are larger than the block saturation level, it is preferred to have larger blocks as they can hold more transactions, and thus more transactions are committed at a higher rate.

3. Including more organizations in a $n - outOf$ endorsement policy increased transaction latency linearly. For 175 tps, *1-outOf* had a transaction delay of around 250ms, and *4-outOf* had a transaction delay of around 300ms. However, nesting endorsement policies had an exploding effect on the transaction delay. Using a $OR[AND(a, b, c), AND(a, c, d), AND(b, c, d), AND(a, b, d)]$ endorsement policy increased transaction delay to 30 seconds for 175 tps. This indicates that one should avoid nesting endorsement policies and favor $n - outOf$ when possible.

4. CouchDB performs significantly worse than LevelDB. This is due to the fact that LevelDB lives inside of the peers, whereas the peers have to make HTTP calls to interact with couchDB. The more reads and writes that were included in a single transaction the worse CouchDB performed. This is due to the fact that shared locks are used in CouchDB and thus the more reads and writes that are made means that more agents will compete for lock time. Therefore the amount of reads and writes that a transaction makes should be minimized or batched when possible.

There exist other literature that analyze Fabric performance [29, 17, 41]. However, most of them are focused on a somewhat fixed topology, i.e. no literature was found where someone analyzed how a Fabric network performed under a larger amount of peers and orderers, e.g. in order of hundreds of orderers. The paper that introduces Fabric [2] claim good evaluation results for one hundred peers. Their results are discussed further below.

7.1.2 Impact of Network Delays on Performance

Nguyen et al. [30] set up a Fabric network over an area network between France and Germany. In their experiments, they used Fabric version 1.2.1. They artificially introduced delays between the two sites in order to analyze how Fabric behaves under different network delays. By increasing the artificial delay by 3.5 seconds, they had up to 135 seconds of delay between the write of the last blocks on the different sites. This threatens the consistency of the network as one partition of it is several blocks ahead of the other one. The consistency threat arose due to a specific topology that they created. Namely, the part of the network that created new transactions had a delay-free connection to the ordering service. As the orderers are responsible for liveness of the whole network [2], that partition of the network could progress and continuously wrote new blocks. The other partition

of the network that was subject to the artificial delay had to wait to receive the blocks. This way the healthy partition of the network kept on progressing while the delayed partition fell further behind. This highlights some important considerations. First, one has to assess the likelihood of network delays in an established Fabric network. If delays are likely, and more importantly potentially large, one should not make strong consistency assumptions, as a partition of the network could fall behind and thus hold an outdated state of the world. As the ordering service, in this case, was healthy the state was updated correctly but only eventually. Second, the problems identified when introducing network delays potentially puts a constraint on who should maintain orderers. The consistency issue, in this case, can be mitigated if one of the orderers would be part of the partition that was under delay. This way the ordering service would have to wait for the delayed orderer before progressing, given that a consensus protocol with stronger consistency guarantees is used, e.g. Raft with strong consistency enabled. However, this will only work up until a certain level of delay, as eventually the delayed orderer will be considered disconnected by the consensus protocol.
Nguyen et al. [30] also found that orderers could be crashed by introducing delay. When an orderer sends a block, it waits for a confirmation from the peer before sending the next block. When delays were severe enough, the block buffer in the orderer would be overfilled and as a result, it halted. They mitigated this by increasing the buffer size of the orderer. This could be further mitigated by using disk space to store blocks that do not fit in the buffer or by introducing a dynamically resizable buffer during network delays. However, it is unknown if this has been addressed in later Fabric versions.

7.2 Discussion of Results

This section provides a discussion on the observed results from the performance evaluation. We tested transaction latency, throughput, and success rate for the operations *Upload Molecule*, *List History of Asset*, and *Transfer Ownership* using the full transaction lifecycle. Latency, throughput, and success rate when doing query transactions for *List History of Asset* was also analyzed. The reason these three operations were selected is because *Register New User* is very similar to *Upload Molecule* and *Update Molecule* is very similar to *Transfer Ownership*. Moreover, *Search Molecule* requires a large set of already existing molecules to be interesting. The testing environment can be described as following [35]:

1. *Consensus Protocol*: Raft [32] was used as consensus protocol for all tests. Ideally, a BFT protocol should have been tested as well. Since no official implementation did exist only Raft was used for the evaluation.

2. *Geographic Distribution*: All of the tests were done on a single machine. Ideally, several machines should be used for testing but only one machine was available to us at the time.

3. *Hardware*: The single machine had 128GB of RAM, and a Ryzen Threadripper 1950X (3.4GHz base, 4GHz boost with 16 cores).

4. *Network Model*: Different network topologies were tested. Their sizes varied from one up to 26 organizations. Each organization contained two peers. Two peers seems to be a reasonable starting point for analysis as each organization has access to one backup peer. Moreover, for each network, the number of orderers varied from one up to 31 orderers. A higher number of orderers were tested. However, at higher orderer numbers, the tests either started having a consistent success rate of zero, or the entire test suite started crashing. These test results are not included in the thesis.
   The block configuration for the ordering service were configured as follows: *AbsoluteMaxBytes* (how large a block can be) was 99MB, *MaxMessageCount* (maximum number of messages in a block) was 10, *PreferredMaxBytes* (how large blocks should be preferably) was 512KB and the *BatchTimeout* (how long the orderers should wait before packaging transactions into a block) was 2 seconds. All other parameters used default values.

5. *Software Components*: The chaincode was written in Golang. Moreover, for the tests, LevelDB was used as the state database in peers instead of CouchDB. The two different databases were discussed in Chapter 5.

6. *Test Tools and Frameworks*: Caliper was used to run the tests. Caliper was selected as it was the only benchmarking tool identified for Fabric.

7. *Type of Data*: The data being used was simple mock data. The size of the data was in the order of kilobytes. Transactions in the order size of megabytes were also tested. However, these tests quickly started to fail. As a result, those results were deemed unsatisfactory, and are not included in the thesis.

8. *Workload*: The workload was generated by Caliper. A *fixed rate* was used, i.e. transactions were sent at a fixed rate. For each operation, transactions were sent at the rates 1 tps, 2 tps, and 5 tps. The duration of each round was 30 seconds. For each operation, each round was run 5 times. After each of these 5 rounds, Caliper paused its transaction sending for 60 seconds. This was used to allow the network to stabilize after each test round.
   The following operations defined in Chapter 4 were tested: *Upload Molecule*, *Transfer Ownership*, and *List History of Asset*. *List History of Asset* was run two times. One time as transaction invocations, and another as transaction queries. Transaction invocations go through the full transaction lifecycle and transaction queries stop once the client receives a result from the execution step. In theory, a query should be faster than a full invocation. In summary, a total of 12 test suites were run: 4 operations (*Upload Molecule*, *List History of Asset*, *List History of Asset Query*, *Transfer Ownership*), 3 arrival rates (1 tps, 2 tps, 5 tps). Each round was repeated 5 times and the final result for each test suite was retrieved by averaging the results from the 5 repeated rounds.

Upload Molecule

Figure 7.1 shows the results from the evaluation of *Upload Molecule*. The size of the molecules uploaded was in the order of kilobytes. For the transaction arrival rates of 1 and 2 tps, every network seems to work fine up until 26 orderers and 16 organizations. After that, the success rate quickly goes to zero. However, for the arrival rate of 2 tps for 26 organizations and 6 orderers, the success rate increased, whereas it was lower for 21 organizations. For the arrival rate of 5 tps, every network seems to work fine up until 11 organizations. After that, the rates trend to zero with varying speed depending on the number of orderers. For 31 orderers, the success rate went to zero after 11 organizations for all transaction rates.
The latency has, in general, a positive trend and the throughput a negative. However, in the case of 1 orderer for *Upload Molecule* at the rate 5 tps, one sees that latency is lower for 16 organizations than for 11 organizations. This could be because the success rate is lower. Transactions that fail quickly provide a lower latency. For 21 organizations with 1 and 6 orderers the latency is higher when the transaction arrival rate was 1 tps than it was for 2 tps. In this case, the difference can not be explained by the 2 tps arrival rate having a lower success rate as it is higher than for 1 tps. It is unknown at this stage why this occurred as one would expect a lower transaction rate to yield better results.
We expect that it is acceptable for end-users of the platform to accept a lower throughput and higher latency when uploading molecules. A latency in the order of minutes or even hours could be acceptable as long as the platform provides strong confidentiality and integrity of the data that is uploaded. However, it is concerning that the network halts completely at 31 orderers. This is more of an availability problem, but if it is severe enough then end-users will become too frustrated as they wait for the network to stabilize. It should be noted that there might still exist querying capabilities even if the transaction lifecycle has halted. End-users could query peers directly for blockchain data. If the peer is reasonably up to date, then they can at least retrieve information, even if they can not upload new molecules.

List History of Asset

Figure 7.2 depicts how *List History of Asset* behaved using full transaction invocations, and Figure 7.3 depicts how *List History of Asset* behaved when the transactions were query transactions. As expected, query transactions are much faster than full transaction invocations. The highest average latency for the query transaction was around 0.08 ms and the highest average latency for the full transaction was around 30 ms. This is beneficial for the prototype as we expect there to be orders of magnitude more reads than writes to the platform. The success rate for *List History of Asset* full transactions was around 100 percent until 21 orderers and 21 organizations. For 31 orderers, the success rate went to zero from 6 organizations and onward. *List History of Asset* query transactions had high success rates up until 21 organizations and 21 orderers. *List History of Asset* query transactions could sustain a success rate of at least 50 percent for 31 orderers up until 11 organizations, and up until 21 organizations for 1 through 26 orderers. 6 orderers could sustain up to even 26 organizations at a success rate of 50 percent.

Transfer Ownership

Figure 7.4 illustrates how *Transfer Ownership* behaved under various network topologies. As with *Upload Molecule* we expect that it is tolerable if this operation has a higher latency and slower throughput. An end-user can probably wait a couple of minutes to transfer the ownership of an asset as long as the platform is secure. For the arrival rates of 1 tps and 2 tps, and network sizes up to 16 organizations the latency was never higher than 30ms. For the arrival rate of 5 tps the highest latency was around 45ms, which occurred at 6 orderers and 16 organizations. As with the other operations, the transactions started to fail as the number of orderers and peers increased.

7.3 Conclusions on Evaluation Results

The results of the tests are somewhat underwhelming. We would like the network to support hundreds of organizations that maintain orderers and peers in the long run. At this point it is hard to know exactly what causes the transactions to fail. Better error reporting, logging, and documentation from both Fabric and Caliper are needed to understand where bottlenecks exist. The results look reasonable until 16 organizations and 26 orderers. After that we could conclude that the networks are too unstable to derive satisfactory conclusions. According to the paper that introduced Fabric [2], they managed to run a network with one hundred peers at transaction arrival rates that were orders of magnitude higher than what was used in this thesis. However, they do not specify exactly how they configured their networks. This makes it hard to replicate their results. Moreover, Caliper consistently crashed as the number of network entities increased. Caliper is a very young project, so its reliability will hopefully improve in the future. According to documentation the following could help make the network more scalable:

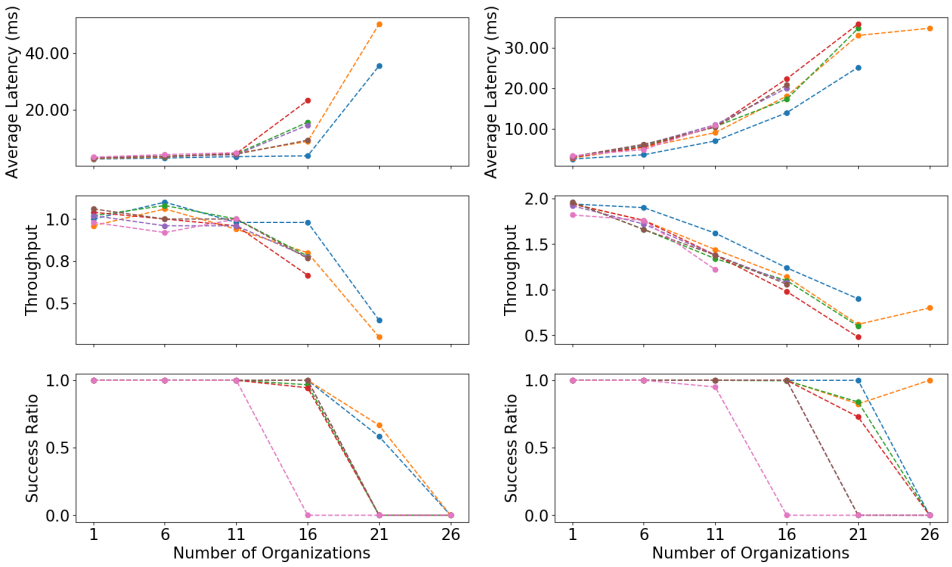
1. The election timeout for the ordering service could be set too low in the default configuration of Fabric [19]. If the ordering service is busy serving blocks, they could miss leader heartbeats and as a result, trigger an election. This way they could end up in a state where they are never able to catch up and as a result, the whole network halts.

2. In order to enable a high transaction throughput, a favorable way to design the chaincode is to commit changes to an asset instead of absolute changes [20]. This is because an absolute change requires a read of the asset. That is, one loads the asset from the ledger and makes a change to it. If there is a high transaction arrival rate the probability that the readset is out of date once it reaches the verification phase increases. Using a so-called delta approach one simply writes the difference in changes to an asset. This way there will be no read/write-conflict as all transactions are write-only.

3. Fabric is designed to be a distributed system. In the paper that introduces Fabric [2] it is claimed that entities rely heavily on CPU intensive cryptographic operations. It is possible that the results can be considered invalid as all of the tests were run on a single machine and that results would improve drastically if one would disperse entities across several machines.

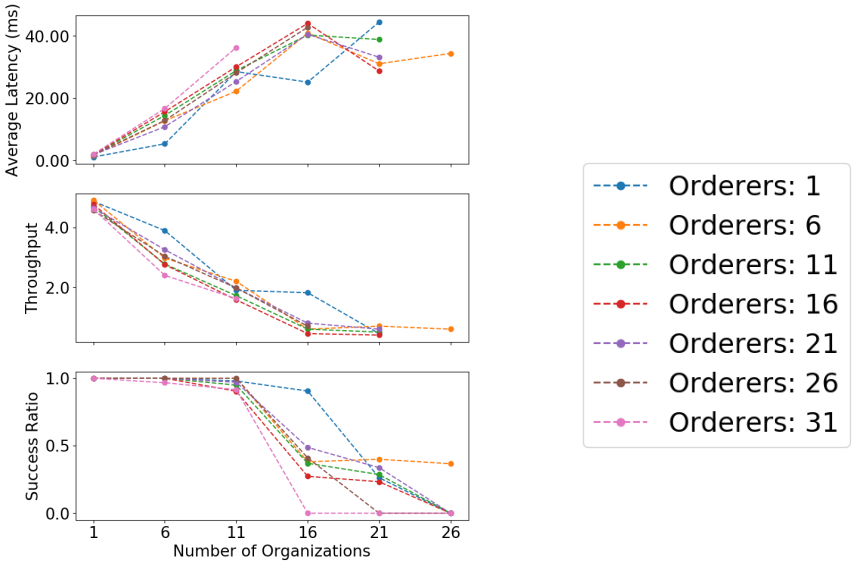
4. How block creation is configured could be bad. Due to a lack of time only one block creation configuration was tested in this thesis. In the paper that introduces Fabric [2] they claim that a block size of 2 MB was optimal for their evaluations. However, they do not state if this is the maximum block size or the preferred size. Moreover, nothing is stated about *MaxMessageCount*, or *BatchTimeout*. Varying these parameters could improve the performance of the prototype.

Scaling Fabric to a high number of participants seems to be non-trivial. An important lesson from the evaluations is that one can not simply use any configuration for a higher number of participants. One has to have a deep understanding of distributed systems and Fabric in order to understand how to configure the network. Also, it seems highly worthwhile to configure sophisticated logging mechanisms so that one can analyze where bottlenecks exist. Moreover, it could be argued that the documentation on how to configure Fabric is unsatisfactory at this moment. This is not surprising as Fabric is a new project. For example, Fabric v1.4 is the first long-term support release of the project.



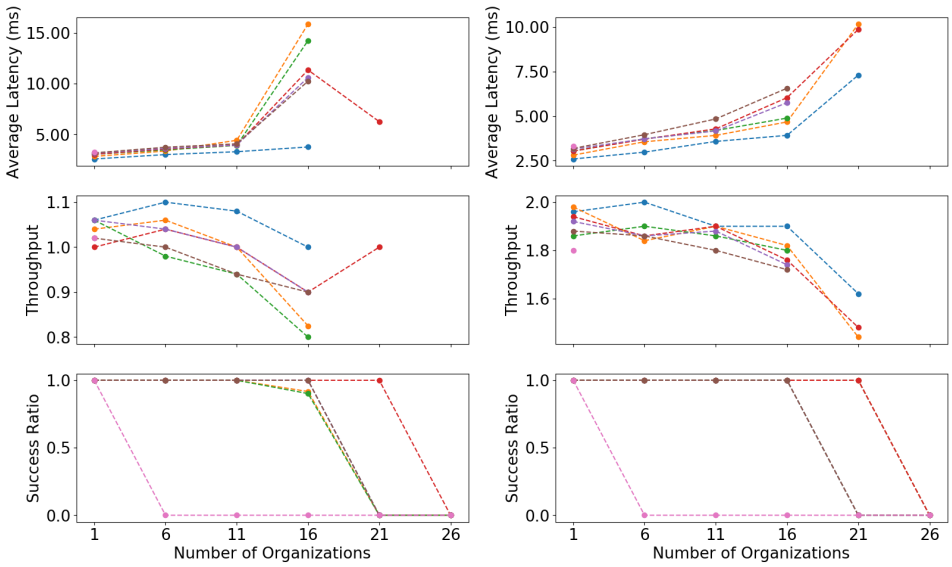
(a) Upload Molecule (1 tps)

(b) Upload Molecule (2 tps)



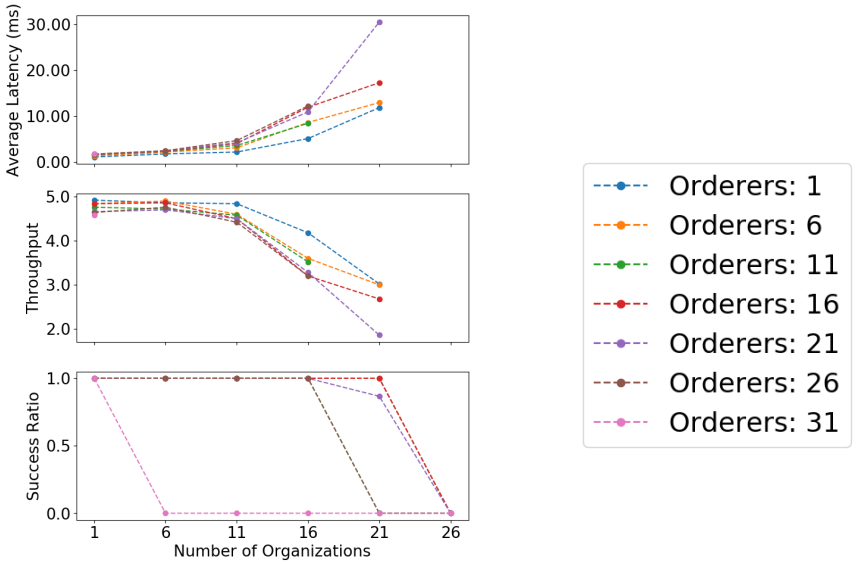
(c) Upload Molecule (5 tps)

Figure 7.1: Average latency, transaction throughput, and success ratio for the *Upload Molecule* operation for the arrival rate of 1,2, and 5 tps. The transactions arrive at a fixed rate for 30 seconds.



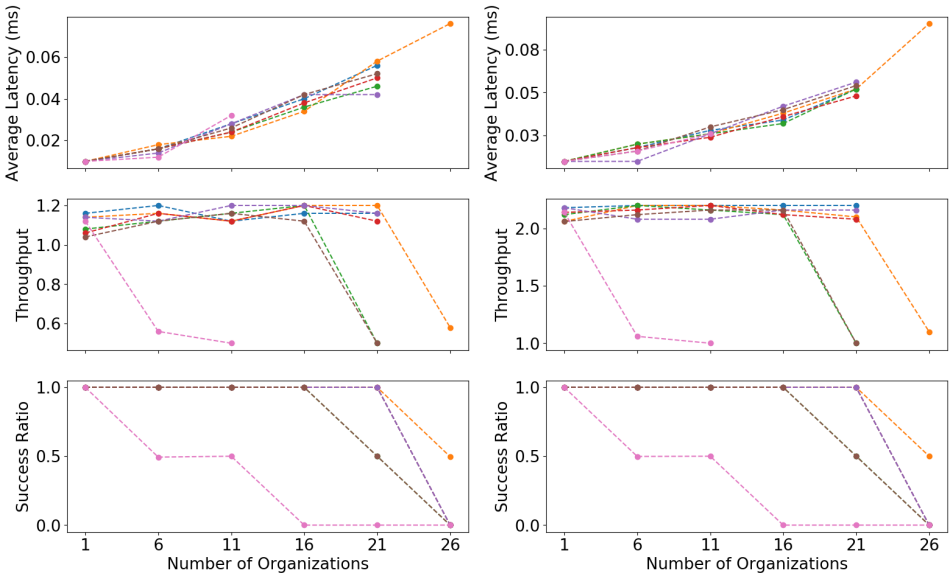
(a) List History of Asset (1 tps)

(b) List History of Asset (2 tps)



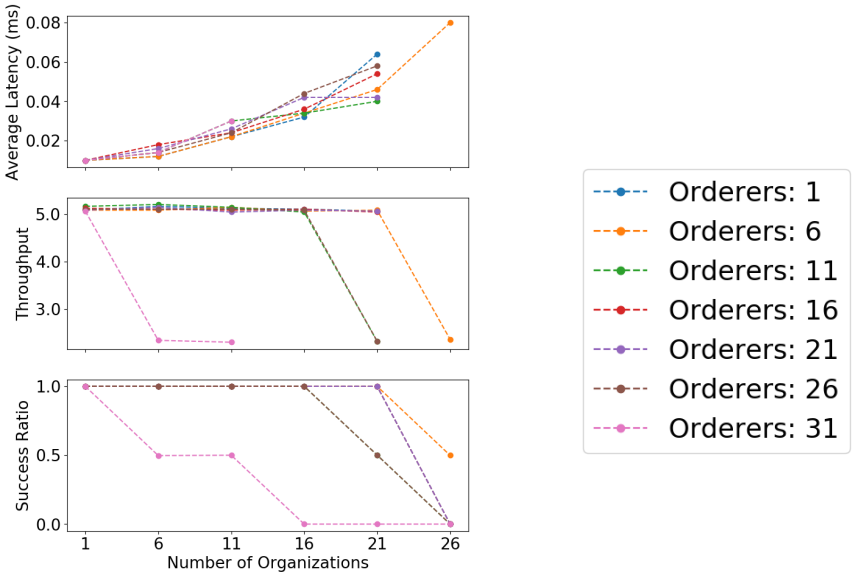
(c) List History of Asset (5 tps)

Figure 7.2: Average latency, transaction throughput, and success ratio for the *List History of Asset* full transaction operation for the arrival rate of 1,2, and 5 tps. The transactions arrive at a fixed rate for 30 seconds.



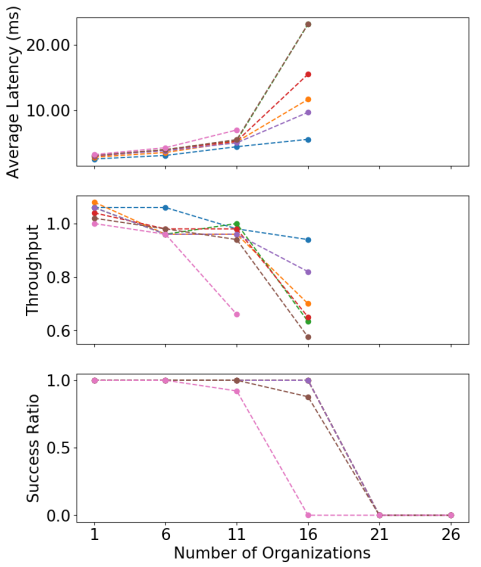
(a) List History of Asset Query (1 tps)

(b) List History of Asset Query (2 tps)

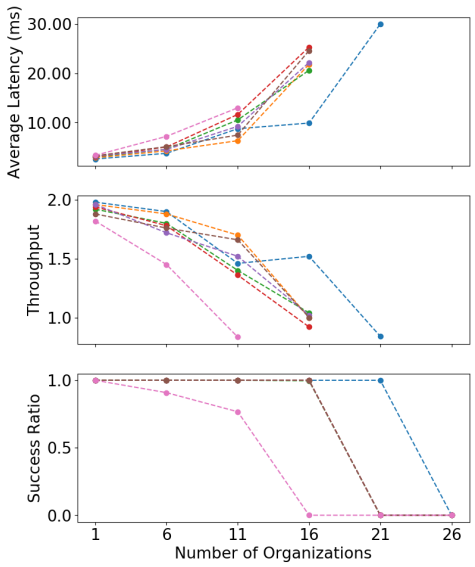


(c) List History of Asset Query (5 tps)

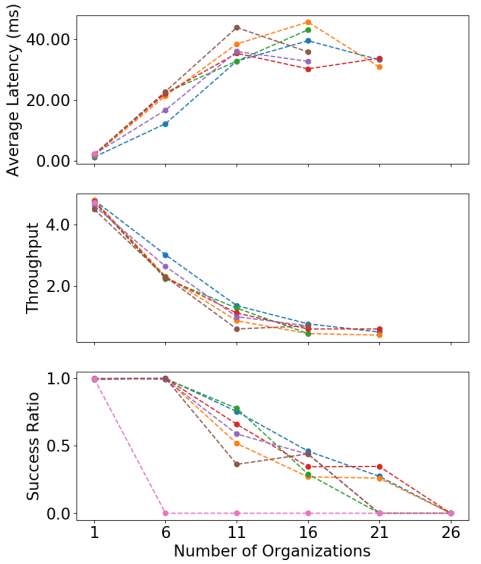
Figure 7.3: Average latency, read throughput, and success ratio for the *List History of Asset* query transaction operation for the arrival rate of 1,2, and 5 tps. The transactions arrive at a fixed rate for 30 seconds.



(a) Transfer Ownership (1 tps)



(b) Transfer Ownership (2 tps)



(c) Transfer Ownership (5 tps)

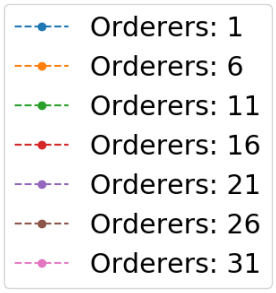


Figure 7.4: Average latency, transaction throughput, and success ratio for the *Transfer Ownership* operation for the arrival rate of 1,2, and 5 tps. The transactions arrive at a fixed rate for 30 seconds.

Chapter 8

Conclusions & Future Work

There exists a great need to revolutionize how intellectual property is handled in the pharmaceutical industry. The current way it is handled seems to stifle innovation greatly since different actors work separately on the same problems. Due to how prior art works there exists no incentive for anyone to collaborate as the monetary risk is too great. Bitcoin has shown that similar issues can be solved in a highly distributed manner. Our hope is to achieve something similar in the pharmaceutical industry. That is, building a distributed system with strong auditability and integrity where ownership of assets is truly in the hands of molecule developers and researchers.

When developing the prototype proposed in this thesis, Fabric was selected as the blockchain technology. The technology stack seems to provide a good enough throughput of transactions and very fast querying capabilities. We expect there to be orders of magnitudes of more reads than writes if the platform can attract users. This is promising for the prototype. However, it is very concerning that the amount of orderers seems to be capped around 26. It is likely that the number of orderers could become higher as the configuration of the network is improved. An open question for the future is if a byzantine fault-tolerant protocol is worth the performance penalty it imposes. Considering that it was non-trivial to scale Raft to a higher number of orderers, we should expect a byzantine fault-tolerant protocol to be even harder to scale.

Assuming that entities trust other entities from the same network there exist no obvious security vulnerabilities in Fabric. However, the security was only analyzed from a conceptual point of view and a very thorough security analysis will be needed in the future to make sure that the prototype provides the desired security properties.

We remain cautiously optimistic that the full vision proposed in this thesis can be realized in the future. However, a handful of open problems were identified during the course of the thesis that will have to be solved before the prototype can be deemed production-ready. In the case that they can not be solved, a different technological stack will be needed to enable an open collaboration on pharmaceutical molecules. The prototype is only a first step, however, and once this thesis is concluded the real work begins.

8.1 Future Work

The security of the proposed platform is completely essential. If security is broken everyone will lose trust in the system and abandon it. There exists a myriad of examples where poor development processes in companies have led to severe reductions in security. One notable example is a crash of the airplane *Boeing 737*. After analysis, it turned out that there was a software bug that likely caused it. A bug in the proposed solution could potentially mean enormous monetary damage to all participants of the network and completely destroy the trust that people have in the product. On the other hand, organizations like *NASA* manage to deploy software in unknown environments with success e.g. when sending Voyager 1 into deep space there were no rollbacks. If there is a bug in the software it will be there for the lifespan of the vehicle. As the project progresses one has to assess the correct development process in order to minimize the risk of catastrophic bugs. Most likely, if no attention is put toward building a security-oriented development process, security will suffer over the long run.

Tokenization - How to bridge the gap between the external world and the blockchain?

Let *bitcoin* be the token that is transacted on the Bitcoin blockchain. The token bitcoin has no externalities, it lives entirely inside of the blockchain and any value accrued to it comes from an external valuation. The only externality Bitcoin has is the miners which, through proof of work, convert physical energy into new blocks. Putting Bitcoin's controversial nature aside, it is hard to dispute that the tokens living inside of the blockchain make total sense. There is no external information needed for an end-user of Bitcoin to trust that a bitcoin is valid, provided that they trust the network.
The contrary is true for molecule tokenization. In this case, we move something from outside of the blockchain onto the blockchain. As far as the blockchain is concerned, if you can produce the keys for a token, then you are the legitimate owner. Using a naive implementation where any user of the network can upload anything, we should expect someone to tokenize a molecule that is legal according to the blockchain, but where it is obvious for other users that the tokenization does not make sense. An example of this could be a user uploading a molecule that is widely known to be a common property e.g. someone uploading the oxygen molecule and claiming full ownership of it. This situation would produce a divergence of the state of the blockchain and the perceived view of the external world. This illustrates the fact that some governance mechanism is needed and a lot of thought has to be put into what it means to tokenize something physical.

Governance and Legal Disputes, Arbiter or Voting Mechanisms

There could be disputes regarding molecules that simpler smart contracts can not resolve. Some mechanism will be needed to resolve these. These could be resolved in two ways or using a combination thereof. Either there could be actors in the network that serve as some sort of arbiter. This could, for example, be legitimated

lawyers that have access to special smart contracts that overrule certain actions e.g. unlawful transfer of ownership. Another approach could be to resolve issues using voting mechanisms where the participants of the network vote on which outcome they think is most reasonable.
How the network handles disputes is critical to how widely the platform will be adopted. On one hand, an arbiter is a point of centralization that could be persuaded to act unjustly. For example, maybe a judge from one country tends to favor companies from the same country. On the other hand, if disputes are resolved via voting, a set of actors may coordinate their votes to disadvantage other members. Therefore serious thought has to be put into how disputes are resolved. The best solution may be a combination of both arbiters and voting, and the best solution could even depend on the nature of the dispute. This is likely one of the hardest problems to solve in order to realize an open collaboration on molecules.

How should derived molecules be valued?

A use-case scenario for the proposed solution is for some entity to use an uploaded molecule to drive research. This begs the question, how should one value molecules that are derived from other molecules on the blockchain? The owner of the base-molecule will be well-defined. The same owner could make a case that they are obliged to part of the reward for the derived molecule. Most likely, the value of the derived molecule will be hard to define using a simple smart contract, and some mechanisms will need to be constructed that can solve disputes in this scenario.

Should the code of the platform be open-sourced?

The essence of this project is to enable trust between competing actors. This is done via using a blockchain where the owner of the private keys legitimately owns the associated token. Smart contracts are used to drive business logic. The ability to verify that the smart contracts follow specifications is made easier if the code is open-source. If the source is closed then the participants would have to trust the developers of the smart contracts not to cheat them. This could potentially make it harder to attract users to the platform. On the other hand, it could be important to protect the source code to prevent other actors from stealing technology. This decision most likely comes down to monetization and network effects. For example, if the platform is monetized via selling membership-subscriptions the money is not made directly from the code itself and could be potentially open-sourced. Also, if there is a substantial network-effect where a critical amount of users belong to the network, the probability that someone else could build the same platform might diminish. In the end, the developers of the platform need to make money in order to maintain it, so that should drive this decision. However, if it is possible, open-source code should certainly be considered.

Is a Byazntine Fault-Tolerant Consensus Procotol worth it?

The current implementation of the prototype uses the *Raft* consensus protocol for the orderers. As discussed Raft is *Crash Fault-Tolerant*. This means that the network survives situations where orderers crash, disconnect or are to slow

to respond. Since computer crashes and dropped internet connections happen a crash fault-tolerant protocol should be used. However, if one of the orderers deviates from the protocol they have the potential to stop the network in its entirety. As discussed previously *Byzantine Fault-Tolerant* protocols are used to mitigate this. The drawback of byzantine fault-tolerant protocols is that they scale significantly worse than crash fault-tolerant protocols. This begs the question, do we expect orderers to behave maliciously? If this is the case then a byzantine fault-tolerant protocol should indeed be used, but this could potentially significantly limit the number of participants that can maintain orderers in the network. Since identities are known in the network all ill-behaving actors could face legal action and banishment from the network, which might be enough to disincentive bad behavior. This comes down to the nature of the participants and how many that will want to be full maintainers of orderers.

Bootstrapping the network - Start with Non-commercial Molecules

One approach to realize the full vision presented in this thesis is to start building the first production version of the prototype only for non-commercial molecules. Non-commercial molecules, in this case, means molecules where no monetary gain is expected from the end-user of the platform. This means that larger risks can be taken on the developers side, meaning that, development can be more rapid and experimental. This is crucial in order to be able to discover good solutions for the problems discovered during the thesis. If no-one is expecting to make money of the molecules, critical errors in the Fabric network will not be as severe. An example of where these kinds of molecules could be found is in University classes. If the University and students agree to publish the molecules in the public domain they stand to gain from helping develop the Fabric network while minimizing the risk of monetary damage.

How much power should the developers of the platform have?

One issue that has to be discussed further is how much power the developers of the platform should have. Since the whole purpose of a blockchain technology is to minimize trust-reliance between participants, the developers can not have absolute control over the platform. If that would be the case a normal distributed database would be preferable as it is easier to scale on a technical level. On the other hand, the developers need some room to make decisions. They need some way to develop the network, fix bugs and handle ill-behaved actors. Crucially, the developers also need some way to make money off the platform, as otherwise, they could not keep maintaining it. This is a hard decision. On one end there is radical decentralization, e.g. bitcoin, where there exists no clear owner. On the other end, there exist proprietary systems, where the developers have control the full rights of the system. The essence of blockchain is to enable trust, meaning that, if one slides too far into proprietary territory the utility of using blockchain diminishes.

No static analysis tool for JavaScript chaincode

There also exist chaincode bindings for JavaScript. No static analysis tool for JavaScript chaincode was identified. This provides an opportunity for someone to implement it. This could be in the scope of another masters' thesis or for the project owners to contribute to the Fabric ecosystem. Yamashita et. al. [49] implemented a static analysis tool for Golang chaincode. They provide a list of steps that their tool goes through in order to detect potential errors in their chaincode. The same procedure could be used to implement a static analysis tool for JavaScript chaincode.

References

[1] D. M. Andrews, S. L. Degorce, D. J. Drake, M. Gustafsson, K. M. Higgins, and J. J. Winter. Compound passport service: supporting corporate collection owners in open innovation. *Drug discovery today*, 20(10):1250–1255, 2015.

[2] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[3] A. M. Antonopoulos and G. Wood. *Mastering ethereum: building smart contracts and dapps*. O'Reilly Media, 2018.

[4] Apache. Couchdb homepage. `https://couchdb.apache.org/`. Accessed: 2020-01-14.

[5] J. D. Atkinson and R. Jones. Intellectual property and its role in the pharmaceutical industry. *Future medicinal chemistry*, 1(9):1547–1550, 2009.

[6] B. Beckert, M. Herda, M. Kirsten, and J. Schiffl. Formal specification and verification of hyperledger fabric chaincode. In *Proc. Int. Conf. Formal Eng. Methods*, pages 44–48, 2018.

[7] V. Buterin et al. Ethereum white paper. *GitHub repository*, pages 22–23, 2013.

[8] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[9] chainsecurity. Chaincode Scanner. `https://chaincode.chainsecurity.com/`. Accessed: 2020-01-14.

[10] Y. Cherdantseva and J. Hilton. A reference model of information assurance & security. In *2013 International Conference on Availability, Reliability and Security*, pages 546–555. IEEE, 2013.

[11] B. Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[12] M. Conti, E. S. Kumar, C. Lal, and S. Ruj. A survey on security and privacy issues of bitcoin. *IEEE Communications Surveys & Tutorials*, 20(4):3416–3452, 2018.

[13] A. Egberts. The oracle problem-an analysis of how blockchain oracles undermine the advantages of decentralized ledger systems. *Available at SSRN 3382343*, 2017.

[14] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. Technical report, Massachusetts Inst of Tech Cambridge lab for Computer Science, 1982.

[15] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[16] Google. Leveldb github. `https://github.com/google/leveldb/blob/master/db/builder.cc`. Accessed: 2020-01-14.

[17] Y. Hao, Y. Li, X. Dong, L. Fang, and P. Chen. Performance analysis of consensus algorithm in private blockchain. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 280–285. IEEE, 2018.

[18] S. R. Heller, A. McNaught, I. Pletnev, S. Stein, and D. Tchekhovskoi. Inchi, the iupac international chemical identifier. *Journal of cheminformatics*, 7(1):23, 2015.

[19] Hyperledger. Architecture Deep Dive. `https://hyperledger-fabric.readthedocs.io/en/release-1.4/`. Accessed: 2020-01-14.

[20] Hyperledger. Github fabric samples. `https://github.com/hyperledger/fabric-samples/tree/release-1.4/`. Accessed: 2019-10-13.

[21] Hyperledger. SDK for Golang. `https://godoc.org/github.com/hyperledger/fabric-sdk-go`. Accessed: 2020-01-14.

[22] Hyperledger Foundation. Fabric sdk go. `https://github.com/hyperledger/fabric-sdk-go`. Accessed: 2020-01-14.

[23] Hyperledger Foundation. Hyperledger - Open Source Blockchain Technologies. `https://www.hyperledger.org/`. Accessed: 2020-01-14.

[24] Hyperledger Foundation. Hyperledger caliper. `https://www.hyperledger.org/projects/caliperl`. Accessed: 2020-01-14.

[25] Johannes Bauer. Ecc tutorial. `https://www.johannes-bauer.com/compsci/ecc/`. Accessed: 2020-01-14.

[26] L. Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (2):125–143, 1977.

[27] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[28] S. Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system, 2008.

[29] Q. Nasir, I. A. Qasse, M. Abu Talib, and A. B. Nassif. Performance analysis of hyperledger fabric platforms. *Security and Communication Networks*, 2018, 2018.

[30] T. S. L. Nguyen, G. Jourjon, M. Potop-Butucaru, and K. Thai. Impact of network delays on hyperledger fabric. *arXiv preprint arXiv:1903.08856*, 2019.

[31] N. Nilsson. personal communication. MSc, PhD, Head of Open Innovation, LEO Pharma.

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

[33] OpenJS Foundation. Expressjs homepage. `https://expressjs.com/`. Accessed: 2020-01-14.

[34] OpenJS Foundation. Nodejs homepage. `https://nodejs.org/en/`. Accessed: 2020-01-14.

[35] H. Performance and S. W. Group. Hyperledger blockchain performance metrics. Technical report, Hyperledger Foundation, 2018.

[36] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss. Security analysis methods on ethereum smart contract vulnerabilities: a survey. *arXiv preprint arXiv:1908.08605*, 2019.

[37] PRV. Patent och registreringsverket. `https://www.prv.se/en/`. Accessed: 2020-01-14.

[38] S. Rouhani and R. Deters. Security, performance, and applications of smart contracts: A systematic survey. *IEEE Access*, 7:50759–50779, 2019.

[39] C. N. Saha and S. Bhattacharya. Intellectual property rights: An overview and implications in pharmaceutical industry. *Journal of advanced pharmaceutical technology & research*, 2(2):88, 2011.

[40] D. R. Stinson. *Cryptography: theory and practice*. Chapman and Hall/CRC, 2005.

[41] H. Sukhwani. Performance modeling & analysis of hyperledger fabric (permissioned blockchain network). *Duke University: Duke, UK*, 2018.

[42] Swagger. Swagger homepage. `https://swagger.io/`. Accessed: 2020-01-14.

[43] N. Szabo. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16)*, 18:2, 1996.

[44] N. Szabo. Bit gold. *Website/Blog*, 2008.

[45] A. S. Tanenbaum and M. Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[46] P. Thakkar, S. Nathan, and B. Viswanathan. Performance benchmarking and optimizing hyperledger fabric blockchain platform. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 264–276. IEEE, 2018.

[47] M. Toorani. Security protocols in a nutshell. *arXiv preprint arXiv:1605.09771*, 2016.

[48] U.S Environmental Protection Agency. Smiles tutorial. `https://archive.epa.gov/med/med_archive_03/web/html/smiles.html`. Accessed: 2020-01-14.

[49] K. Yamashita, Y. Nomura, E. Zhou, B. Pi, and S. Jun. Potential risks of hyperledger fabric smart contracts. In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 1–10. IEEE, 2019.

Appendix A
Chaincode

A.1 Source Code

```
/*
 * SPDX-License-Identifier: Apache-2.0
 */

package main

import (
  "bytes"
  "encoding/json"
  "fmt"

  "github.com/hyperledger/fabric/core/chaincode/shim"
  sc "github.com/hyperledger/fabric/protos/peer"
)

var logger = shim.NewLogger("chaincode-logger")

// Chaincode is the definition of the chaincode structure.
type Chaincode struct {
}

/*
  Type Definitions
*/

type User struct {
  ID   string `json:"id"`
  Name string `json:"name"`
}

type Molecule struct {
  ObjectType        string   `json:"docType"`
  Version           int      `json:"version"`
  Index             string   `json:"index"`
  Name              string   `json:"name"`
  Synonymes         []string `json:"synonymes"`
  Submitter         string   `json:"submitter"`
  Timestamp         string   `json:"timestamp"`
  Owner             string   `json:"owner"`
  Value             int      `json:"value"`
  SalesPriceIPR     int      `json:"salesPriceIPR"`
  Bids              int      `json:"bid"`
  BiologicalTarget  string   `json:"biologicalTarget"`
  Structure         string   `json:"structure"`
  Data              []string `json:"data"`
  MotherMolecule    string   `json:"motherMolecule"`
  PhysicalStorage   string   `json:"physicalStorage"`
  Contact           string   `json:"contact"`
  AssetType         string   `json:"assetType"`
}

type AssetType int

const (
  ConfidentialAsset AssetType = iota
```

```
  NewAsset
  NewPublicAsset
  KnownAsset
  KnownPublicAsset
)

/*
  Chaincode
*/

// Init is called when the chaincode is instantiated by the blockchain network.
func (cc *Chaincode) Init(stub shim.ChaincodeStubInterface) sc.Response {
  fcn, params := stub.GetFunctionAndParameters()
  fmt.Println("Init()", fcn, params)
  return shim.Success(nil)
}

// Invoke is called as a result of an application request to run the chaincode.
func (cc *Chaincode) Invoke(stub shim.ChaincodeStubInterface) sc.Response {
  fcn, params := stub.GetFunctionAndParameters()

  switch fcn {
  case "TransferOwnership":
    return cc.TransferOwnership(stub, params)
  case "GetHistoryForAsset":
    return cc.GetHistoryForAsset(stub, params)
  case "UploadMolecule":
    return cc.UploadMolecule(stub, params)
  case "CreateUser":
    return cc.CreateUser(stub, params)
  case "UpdateUser":
    return cc.UpdateUser(stub, params)
  case "QueryMolecules":
    return cc.QueryMolecules(stub, params)
  default:
    return shim.Error("No match in function name")
  }
}

func (cc *Chaincode) CreateUser(stub shim.ChaincodeStubInterface, args []string)
     sc.Response {
  id := args[0]
  name := args[1]

  userExists, err := stub.GetState(id)
  if err != nil {
    return shim.Error("Failed to verify if user already exists")
  }
  if userExists != nil {
    return shim.Error("User already exists")
  }

  newUser := User{
    ID:   id,
    Name: name,
  }
  logger.Infof(fmt.Sprintf("New User: %+v", newUser))

  newUserAsJSONBytes, err := json.Marshal(newUser)
  if err != nil {
    return shim.Error("Failed to marshall new user to bytes")
  }

  err = stub.PutState(id, newUserAsJSONBytes)
  if err != nil {
    return shim.Error("Failed to update state for new user")
  }
  return shim.Success([]byte("New user was created"))
}

func (cc *Chaincode) UpdateUser(stub shim.ChaincodeStubInterface, args []string)
     sc.Response {
  id := args[0]
  name := args[1]

  userAsBytes, err := stub.GetState(id)
  if err != nil {
    return shim.Error("Failed to retrieve old user")
  }
  if userAsBytes == nil {
```

```
134       return shim.Error("User does not exist")
    }
136
    user := &User{}
138   err = json.Unmarshal(userAsBytes, user)
    if err != nil {
140     return shim.Error("Failed to unmarshall user")
    }
142
    user.Name = name
144
    userAsJSONBytes, err := json.Marshal(user)
146   if err != nil {
      return shim.Error("Failed to Marhsal user as Json bytes")
148   }

150   err = stub.PutState(id, userAsJSONBytes)
    if err != nil {
152     return shim.Error("Failed to update state for user")
    }
154   return shim.Success(userAsJSONBytes)
}
156
func (cc *Chaincode) QueryMolecules(stub shim.ChaincodeStubInterface, args []
      string) sc.Response {
158   if len(args) < 1 {
      return shim.Error("Incorrect number of arguments. Expecting 1")
160   }

162   queryString := args[0]

164   resultsIterator, err := stub.GetQueryResult(queryString)
    defer resultsIterator.Close()
166   if err != nil {
      return shim.Error(err.Error())
168   }

170   // buffer is a JSON array containing QueryRecords
    var buffer bytes.Buffer
172   buffer.WriteString("[")
    bArrayMemberAlreadyWritten := false
174   for resultsIterator.HasNext() {
      queryResponse,
176     err := resultsIterator.Next()
      if err != nil {
178     return shim.Error(err.Error())
      }
180   // Add a comma before array members, suppress it for the first array member
      if bArrayMemberAlreadyWritten == true {
182     buffer.WriteString(",")
      }
184   buffer.WriteString("{\"Key\":")
      buffer.WriteString("\"")
186   buffer.WriteString(queryResponse.Key)
      buffer.WriteString("\"")
188   buffer.WriteString(", \"Record\":")
      // Record is a JSON object, so we write as-is
190   buffer.WriteString(string(queryResponse.Value))
      buffer.WriteString("}")
192   bArrayMemberAlreadyWritten = true
    }
194   buffer.WriteString("]")
    fmt.Printf("- getQueryResultForQueryString queryResult:\n%s\n", buffer.String
      ())
196
    return shim.Success(buffer.Bytes())
198 }

200 func (cc *Chaincode) TransferOwnership(stub shim.ChaincodeStubInterface, args []
      string) sc.Response {
    //@TODO: make sure stub.GetCreator is equal to current owner
202   index := args[0]
    newOwnerID := args[1]
204
    assetAsBytes, err := stub.GetState(index)
206   if err != nil {
      return shim.Error("Error in retrieving asset")
208   }
    if assetAsBytes == nil {
210   return shim.Error("Asset does not exist")
```

```
  }

  newOwnerAsbytes, err := stub.GetState(newOwnerID)
  if err != nil {
    return shim.Error("Failed to retrieve new owner")
  }
  if newOwnerAsbytes == nil {
    return shim.Error("User does not exist")
  }

  asset := Molecule{}
  err = json.Unmarshal(assetAsBytes, &asset)
  if err != nil {
    return shim.Error("Failed to unmarshal assetAsBytes " + err.Error())
  }

  newOwner := User{}
  err = json.Unmarshal(newOwnerAsbytes, &newOwner)
  if err != nil {
    return shim.Error("Failed to unmarshal newOwnerAsBytes")
  }

  logger.Infof(fmt.Sprintf("New Owner: %+v", newOwner))

  asset.Owner = newOwner.ID

  assetAsJSONBytes, err := json.Marshal(asset)
  if err != nil {
    return shim.Error("failed to marshal asset as bytes")
  }

  err = stub.PutState(index, assetAsJSONBytes)
  if err != nil {
    return shim.Error("Failed to update asset state")
  }
  return shim.Success(assetAsJSONBytes)
}

func (cc *Chaincode) GetHistoryForAsset(stub shim.ChaincodeStubInterface, args
    []string) sc.Response {
  key := args[0]

  history, err := stub.GetHistoryForKey(key)
  defer history.Close()

  if err != nil {
    return shim.Error("Failed to retrieve History for key")
  }

  var result []byte
  for history.HasNext() {
    modification, err := history.Next()
    if err != nil {
      return shim.Error("error in iter of history")
    }
    result = append(result, modification.Value...)
  }
  return shim.Success(result)
}

func (cc *Chaincode) UploadMolecule(stub shim.ChaincodeStubInterface, args []
    string) sc.Response {
  //@TODO: make sure that index does not already exist!
  argumentMap := args[0]

  logger.Infof("Argument Map: " + argumentMap)

  newAsset := Molecule{}

  err := json.Unmarshal([]byte(argumentMap), &newAsset)
  if err != nil {
    logger.Errorf(err.Error())
    return shim.Error("Failed to unmarshall bytes: " + err.Error())
  }

  newAssetAsJSONBytes, err := json.Marshal(newAsset)
  if err != nil {
    logger.Errorf(err.Error())
    return shim.Error("Failed to marshall bytes: " + err.Error())
  }
```

```
290     err = stub.PutState(newAsset.Index, newAssetAsJSONBytes)
        if err != nil {
292       logger.Errorf(err.Error())
          return shim.Error("Failed to put state for new asset: " + err.Error())
294     }
        return shim.Success(newAssetAsJSONBytes)
296 }
```

Listing A.1: The chaincode