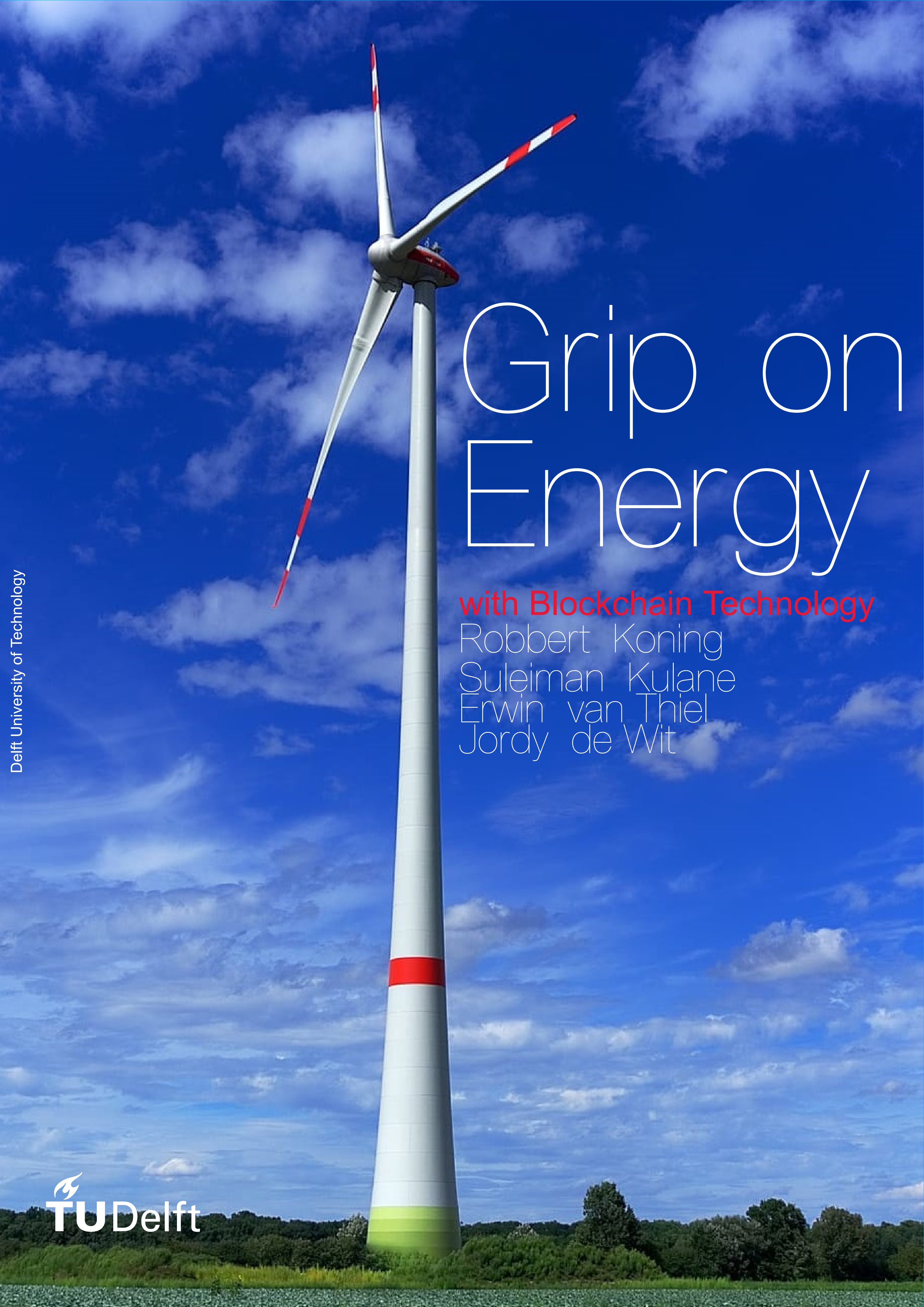

Grip on Energy

with Blockchain Technology

Robbert Koning
Suleiman Kulane
Erwin van Thiel
Jordy de Wit

Delft University of Technology

TU Delft

Grip on Energy

with Blockchain Technology

by

Robbert Koning
Suleiman Kulane
Erwin van Thiel
Jordy de Wit

Bachelor's Thesis
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Project duration:	November 11, 2019 – February 16, 2020	
Thesis committee:	Dr. S. Roos,	TU Delft, supervisor
	Ir. O.W. Visser,	TU Delft, coordinator
	Ir. S. Hijgenaar,	CGI Nederland, Senior Business Consultant
	Ir. H. Heine,	CGI Nederland, Director Consulting Services

TU Delft

Preface

This report portrays what we, the authors, have accomplished and learned during our bachelor's thesis and we would like to present it to you proudly. We would like to thank and acknowledge Stefanie Roos for her guidance and consultation, Otto Visser for helping us with legal concerns, Henry Heine for being our contact at CGI and answering all our questions with respect to contractual matters, and finally Sjors Hijgenaar for his consultation, for his encouragement, for his enthusiasm despite the setbacks we experienced, and for giving us the opportunity to do this project in collaboration with CGI Nederland in the first place.

Robbert Koning
Suleiman Kulane
Erwin van Thiel
Jordy de Wit
Delft, February 2020

Summary

Switching energy suppliers can be a time consuming process and the manner in which permissions regarding consumer data are stored lacks transparency. To overcome these issues, a solution was proposed in the form of a mandate register. Said register keeps track of which consumer gave what permission, regarding energy data, to whom. In this project a prototype of such a register was built.

The register is required to be designed in such a way that it is expandable and secure. From these characteristics, the conclusion was drawn that a permissioned blockchain network was the most suitable option for storing mandates in a decentralised and immutable fashion.

The most fitting consensus algorithm for the blockchain network was determined to be the Raft algorithm. For implementation of the blockchain network, a widely-documented and advanced framework called Hyperledger Fabric was used, which was be configured to use Raft. A network in Hyperledger Fabric is a set of organisations of which subsets can form channels together. Each organisation consists of multiple peer nodes, each with a corresponding ledger, database and smart contracts. The mandate register network consists of two channels, the first containing seven organisations, with two peer nodes per organisation. Apart from these seven organisations, the network contains seven orderers, which work in the second channel and which are responsible for managing transactions made by an application.

On top of the network, an application was built that connects to the network and functions as a simple web server. The web server allows consumers to submit their mandates as input to the network and query mandates from the network.

Performance evaluation of the network shows that it requires much optimisation before being ready for deployment in the real world.

Apart from optimisation, there are various tasks related to security, authentication, deployment, and the GDPR which have to be completed before the register can be used in production.

Contents

1 Introduction

For consumers, switching from energy supplier within the Dutch energy market is quite a straightforward process; a consumer contacts their preferred new energy supplier and waits a few weeks. During these weeks, the new supplier checks the Contract Einde Register (CER) to verify the consumer is allowed to switch according to their current contract and their cancellation period. If the consumer is allowed to switch, the new supplier contacts a distribution grid operator (DGO) with the switch request.

There are seven distribution grid operators in the Netherlands which deal with the electricity network. These DGOs each have their own territory, meaning all estates in a certain area are managed by a certain DGO. Therefore, consumers cannot choose their DGO like they can choose their energy supplier; the location of their home determines which DGO is responsible for them. Among other tasks, DGOs handle supplier switching and maintain the energy grid.

One step of supplier switching is communicating a consumer's energy measurement data between energy suppliers. This data is collected using the smart meter installed at a consumer's estate, which is numbered by a unique EAN code. The collected data is used to determine at what usage value a certain contract ends and a new one is started. For the new supplier to be allowed to collect this data, they need permission from the consumer to read their smart meter. Permission is given implicitly when a consumer initiates a switch request.

1.1. Problem Definition

From the context defined above, a few main problems become apparent:

1. Lack of transparency: In the current system, consumers have no clear overview of the parties that are or have been able to see and use their data, because currently no record of these permissions is being kept.

2. Time consumption: The process of switching suppliers can be time consuming for consumers. It can take up to six weeks for the switch to go into effect [21]. When switching, there is much communication between parties, for instance for verifying each others' data. Verification can be even more time consuming when there is conflict between parties. Moreover, the new supplier has to wait on permission to read the client's smart meter data.

The project's aim is to solve these problems by building a mandate register that stores the consumer's permission (or mandate) for the data to be used. Such a register would allow consumers to digitally specify which energy suppliers are allowed to use their data. This means that consumers become able to change which energy suppliers are able to use their data with a simple update in the register.

A register accessible to both consumers and suppliers could overcome data transparency issues from the perspective of consumers while keeping consumer energy data usable from the perspective of suppliers. The register can be used to implement an interface in which consumers can see a proper overview of the parties that are or were allowed to use their data, and when.

Moreover, the register could speed up the process of switching from suppliers, because it could perform the necessary checks itself so that suppliers do not have to trust each other.

Lastly, many other use cases become possible with an efficiently functioning mandate register. This is because consumers can make specific data available for use by any type of party, such as an academic institution. Though the main focus of this project is switching to another energy supplier, a large emphasis is placed on keeping the door open for other use cases.

1.2. Outline

First, a conceptualisation of the solution is given, by specifying design goals, supporting the choice to use blockchain technology, and specifying a list of requirements. The conceptualisation ends with a set of success criteria. Then, a mandate register is designed by choosing a suitable consensus algorithm, choosing a framework, and analysing its workflow and concepts. Then, the implementation of software regarding logic of the network and the application that connects to it are explained. Then, the performance of the register is evaluated by means of specifying the metrics, procedures and topology of an evaluation network. The evaluation is concluded by listing the results in suitable tables and figures. Finally, the resulting implementation and evaluation are reflected upon. This is done by discussing the evaluation results, the success criteria, the ethical implications, the process and issues that were involved, and lastly the work that has yet to be done.

2 Conceptualisation

In this chapter the project's design goals are specified, the design choices with respect to data storage are elaborated upon, and the requirements that were set in consultation with the client and supervisor are listed. The chapter is concluded by stating the success criteria.

2.1. Design Goals

2.1.1. Expandability

Expandability is crucial to the success of the project. All major design choices were made with the idea that the system should support many different use cases and many users.

Support for various Use Cases

It is critical that the system can be extended to support many different use cases. While the project's specific use case is allowing consumers to switch between energy providers, the challenge is to facilitate as many other use cases for mandates as possible. In order to facilitate other use cases, mandates should adhere to a generic format, intuitively formatted as: 'Party X grants/revokes permission for party Y to access X's data concerning Z, for the purpose of W.' Z in this project would be energy and W could for instance be 'contract', 'research', etc. Therefore, instead of implementing a single use case, a system is implemented, which is then proven to work by implementing a use case on top of it.

Scalability

Another important aspect of expandability is that the system should be able to handle thousands of consumers simultaneously granting and revoking mandates and hundreds of parties simultaneously querying this data.

2.1.2. Security and Privacy

Another design goal is 'security and privacy'. This means that data read from the system is accurate, that data written to the system is verified, that parties with access to the system are authenticated, that the system does not have a central authority, and that the system handles sensitive data with care. These statements are elaborated upon below.

Verified Data Reading

Data queried from the system must be accurate and untampered with. Parties must be assured that they can retrieve all mandates that belong to them from the system, and that these mandates are untampered with and complete. Therefore, all mandates that are accepted by the system, should be immutable once stored.

Verified Data Writing
All data that is written to the system should be verified to adhere to the mandate format defined in Section 2.1.1, before it is written to the system.

Authentication
In order to guarantee that mandates cannot be granted, revoked or read by unauthorized parties, all parties with access to the system should have to authenticate themselves before being able to read or write mandates.

No Central Authority
In order to prevent improperly functioning parties from modifying consumers' mandates, there should not be a central, corruptible authority for adding data to the system.

GDPR Compliance
The system is credible if it handles consumers' data with care. Therefore, it should follow the guidelines laid out by the General Data Protection Regulation (GDPR) [7]. The concept of a mandate register is in line with the GDPR's core principles, for it gives consumers more control over their own data.

However, mandates themselves are personal data because they can be traced back to individuals. Storing these mandates raises GDPR concerns and these concerns should be taken care of. However, it is important to realise that solving all problems might not be possible within the scope and duration of the project.

2.2. Blockchain

Distributed database systems were considered for the storage of mandates. After discussion with the client, it became apparent that they had a preference for using blockchain, so blockchain was considered an option. Another option would be any other type of distributed database. Both systems can have similar properties such as decentralisation and immutability. Decentralisation is important because the authority to change data in the network should not be centralised. Decentralisation is only possible in distributed systems. Immutability guarantees that the data cannot be altered after addition. These properties are required when considering the design goals of expandability and credibility.

Blockchain in general was found to have a large collection of available documentation and tutorials. Also, nodes in a blockchain network can all be mutually mistrusting entities that do not want to have to rely on a trusted third party for exchanging data [22]. Taking into account these properties of blockchain, it is apparent that blockchain does not have any considerable disadvantages compared to any other distributed databases for this project. Therefore, after thorough research and discussion, it was agreed upon to use blockchain for the project.

Having decided upon using blockchain, two types of blockchain can be considered. Namely permission-less and permissioned blockchains. A permission-less blockchain is a blockchain that everybody is free to join at any time. There is no central authority that can decide to ban or disallow any peer. This means that everybody can host a node in the network [22]. A permissioned blockchain, however, does work with an authority which decides who can or cannot be part of the network and which participant gets what rights. An example could be the right to validate blocks [22]. Note that the central authority in a permissioned blockchain cannot hinder data transactions between nodes.

In this project, there was need for a third party to decide who is allowed to host a server, because not everybody should be able to do so. Consequently, a permissioned blockchain with a central authority determining its participants was determined to be the option best suited for this project. The blockchain would also be private rather than public because not everybody should be able to access the stored data. The parties participating in the blockchain network are the seven DGO's. They do not have a financial stake in the switching of energy suppliers, but are however part of the process.

2.3. Requirements

After discussing with the client, the original project description, as found in Appendix C, was altered to create a MoSCoW requirements document. The enumeration below specifies what exactly was changed.

- Not all use cases listed in the original project description document were to be be implemented. Instead, the focus was put on switching energy suppliers.
- The original project description stated that multiple data sources should be interconnected. A consequence of shifting the focus to only the aforementioned use case was that only one data type would be considered, namely energy consumption data.
- DigiD was meant to be used for authentication, and the client had tried to get a development environment for this. However, the client could not get this done before the project started thus authentication would be implemented in another way.

MoSCoW

Must Haves

- Mandates must be stored on a blockchain
- The blockchain must be permissioned, such that validators will only be added on a permissioned basis.
- A consumer must be able to see an overview of their mandate(s).
- A consumer must be able to grant mandates for specific parties to access specific data.
- A consumer must be able to revoke mandates for specific parties to access specific data
- A consumer must be able to switch between energy providers.
- A producer, meaning a party that produces energy, must be able to query the network to verify if a consumer gave them a mandate to access specific data.
- Validator nodes must be able to validate users' requests to append data. A request is considered valid if:
  - The user making the request is authorized and authenticated.
  - The request is properly formatted according to the protocol that is specified.
- Validator nodes must be able to append valid requests to the blockchain using a consensus algorithm.

Should Haves

- The protocol should allow granting/revoking access to different types of data.
- A user should be able to authenticate using DigiD or another authentication method.
- A user should have a graphical user interface to give and revoke mandates.
- A user should have a graphical user interface to show a history of their mandates.
- A user should be able to automatically revoke mandates for a previous energy provider when switching to a new one.
- The network should be able to add and remove validators.

Could Haves

- A user could be asked for confirmation when they initiate a mandate modification.

- The consumers' mandates are categorized under their EAN-code.

Won't Haves

- The blockchain will not support prosumers (consumers that produce energy as well).
- The blockchain will not support any other type of asset beside energy meters.

Non-Functional Requirements

- The system will be properly tested. Tested means that the written code will be:
  - Software tested and measured using code and branch coverage.
  - System tested for a small mocked network.
- The blockchain will be set up using frameworks.
- The system will be developed using version control Git.
- The code will be sent to the Software Improvement Group (SIG) for evaluation.

2.4. Success Criteria

Based on the problem definition of Section 1.1 and the design goals and requirements of this chapter, it is possible to state the criteria that must be met in order to consider the project a success. These success criteria form a useful guide during the implementation phase of the project and will give a rough indication of the project's progress. The criteria are rather intuitive:

1. The system must be more transparent and less time consuming than in the current situation.
2. The design goals stated in this section must be met.
3. From the MoSCoW at least the must haves need to be implemented.

3
Design

This chapter highlights the choices made for the design of the product. Several consensus algorithms are discussed and compared. This comparison is then used to make a choice for the framework to be used. Hyperledger Fabric and its way of modelling a network is discussed. This is followed by the design choices made for the mandate register network. Finally, the way Fabric puts such a model into practice is explained.

3.1. Consensus Algorithms

When choosing an algorithm that is going to determine in what fashion the blocks are added to the ledger, several characteristics have to be taken into consideration. To elaborate, these specific characteristics are speed, scalability and fault tolerance:

- The network should be fast enough to let people switch providers or query their mandate information within a matter of seconds.
- The network should be relatively robust to scaling, meaning that it should maintain reasonable performance with respect to the number of network participants as this number grows. Scalability is only a factor up to a certain degree because the network is not expected to become very large nor grow multiple orders of magnitude.
- Fault tolerance is also an important attribute because network nodes can always fail. If this happens, the network should maintain a consistent and correct state, when it comes to stored data.

In the field in which the network will be deployed, no concern is needed for malicious nodes since it is a permissioned system. Therefore, resistance against maliciousness is not taken into consideration.

Proof of Work

In a Proof of Work algorithm (PoW), as used in for example Bitcoin, each node in the network competes for adding a new block by solving a complex mathematical puzzle. When a node finds the desired, easily verifiable solution, it broadcasts the block to all other nodes. The other nodes must validate the block and if it is correct, they all append it to their own copy of the chain. The essence of this algorithm is that a node has to do a tremendous amount of work before being able to append a new block, to prove that a node is not likely to be malicious [23]. All this work costs a lot of energy and resources, therefore this approach does not seem appropriate for an application in the energy market. Besides being very energy consuming, a PoW protocol also has a poor transaction rate [3], and speed is a factor that should be taken into account.

Proof of Stake

Unlike Proof of Work, a Proof of Stake algorithm (PoS) does not need an enormous amount of energy to append a new block. The node which appends a block to the chain will be pseudo-randomly chosen out of the stakeholders [18]. A 'stakeholder' refers to a party with a monetary incentive in the network. In the network that is being built, the entities participating in the network do not have such an incentive. Therefore, Proof of Stake is unsuitable for application in the energy sector.

Practical Byzantine Fault Tolerance
Practical Byzantine Fault Tolerance (pBFT) is an algorithm in which all of the parties are known, but they may be faulty or malicious. The network will reach consensus by means of voting [5]. Voting can be done fast, but there will be overhead in order to reach consensus after voting. The overhead is not a problem within a small network, but once the network becomes larger, it will become a bottleneck [3]. This type of consensus algorithm works well in a permissioned system.

Proof of Elapsed Time
Proof of Elapsed Time (PoET) is a consensus algorithm that elects a random leader to append a block to the chain. All nodes in the network are assigned a random waiting time, and the first node to wake up finishes the block and broadcasts it to the other nodes. To avoid manipulation by malicious nodes, all nodes need to run in a Trusted Execution Environment (TEE), for example Intel's Software Guard Extensions. This TEE guarantees that the random waiting timer cannot be tampered with by malicious software [3]. PoET overcomes the drawbacks of PoW and PoS but requires the nodes to run on specific hardware.

Raft
The Raft consensus algorithm is built up to solve three problems, namely leader election (Figure 3.1), log replication and safety.

Leader election is the process of network participants choosing a 'leader' which determines what data to add to the chain. A leader uses an `AppendEntry Remote Procedure Call` (RPC) for this. All requests to add data must go through the leader. In the case of Raft, all nodes start a self-specified pseudo-random timeout, and the node whose timer ends first becomes a candidate node. This candidate starts an election and asks all other nodes to vote for him, by means of a `RequestVote Remote Procedure Call`. Nodes will vote for the first candidate which contacts them, except if they are a candidate themselves. If a majority of the nodes vote for a candidate, it becomes the leader. After an arbitrary, self-specified amount of time or when the leader fails to send a heartbeat, a new election is started.

Log replication is the process of ensuring that all network participants share the leader's log. In Raft, the leader sends the new entries from its own log to all its followers. If a majority approves this entry and adds it to their logs, the entry is committed. Raft ensures that all previously committed entries are also agreed upon in the network.

The last problem which Raft attempts to tackle is safety. For instance, Raft imposes restrictions on becoming a leader, such as demanding that all leaders have an up-to-date log. However, these safety measures do not guarantee that malicious leaders cannot take over the network. For this project it is not a problem, because malicious nodes are not taken into account because the network is permissioned system.

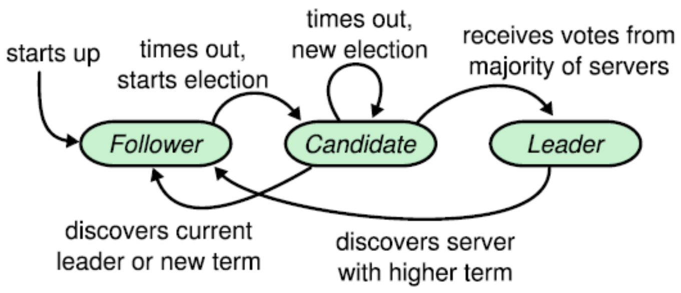
starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Figure 3.1: The states of the nodes can be represented by this state machine [17]

The Raft algorithm seems well-suited for a permissioned blockchain in which all parties are treated equally. Every node has an equal probability of becoming leader. This is also the case for the PoET algorithm, but Raft does not depend on specific hardware. The algorithm is also robust to failing nodes due to the heartbeat mechanism; it can tolerate up to 49% of nodes failing - which is called Crash Fault Tolerance - and is much faster than PoW or PoS algorithms. A drawback of Raft is however that the network is limited in scalability due to its architecture and cannot handle malicious nodes.

The information in the Raft section was based on [17] and [15].

Conclusion

After evaluating the above consensus algorithms, Raft seems to be most appropriate for this project. The two drawbacks, being lack of scalability and incapability of handling malicious nodes are lesser problems than hardware dependencies, lack of speed and lack of fault tolerance. This is because in this project the possibility of nodes being malicious is not taken into account and the number of network nodes will not be very large. An overview of the comparison of the previously discussed algorithms can be found in Table 3.1.

	PoW	PoS	PoET	BFT and variants	Raft
Blockchain type	permission-less	both	both	permissioned	permissioned
Transaction rate	low	high	medium	high	high
Scalability	high	high	high	low	limited
Adversary tolerance	<=25%	algorithm dependent	unknown	<=33%	none

Table 3.1: Consensus comparative analysis based on [3], the raft properties are based on [15]

3.2. Frameworks

For the development of the blockchain network, multiple frameworks were considered. The requirements and design choices discussed in earlier sections will be used to evaluate the applicability of the frameworks. Other important factors to consider are how well the frameworks are documented and how well they are maintained.

Hyperledger Fabric

Hyperledger Fabric is an open-source platform designed for the development of permissioned blockchain networks. It has substantial documentation which contains theoretical explanation and code tutorials. Fabric implements a consensus algorithm based on Raft [2], which means it is resistant against a considerable percentage of node failure and can achieve high transaction rates. Fabric also supports querying data through CouchDB instead of querying the blockchain directly. The information in this paragraph was gathered from [10].

+ Widely used, well-documented and maintained;
+ Raft-based consensus algorithm has most of the desired characteristics: it is fast, relatively scalable and Crash Fault Tolerant.
+ Querying capabilities powered by CouchDB.

Hyperledger Sawtooth

Hyperledger Sawtooth is an open-source platform designed for the development of blockchain networks [13]. Sawtooth provides an elaborate documentation and is also widely used. It is also built in such a way that most application logic can be written in a variety of common programming languages, such as Python, JavaScript, Go, C++ and Java. Finally, Sawtooth allows plugging in different consensus algorithms, but by default it only offers a form of Practical Byzantine Fault Tolerance or Proof of Elapsed Time. This is Sawtooth's main drawback with regards to the project; neither consensus algorithm is suitable, and implementing another one is not feasible considering the project's limited duration.

+ Widely used, well-documented and maintained;

+ Application logic can be written in a wide variety of languages;
- Consensus algorithms are not suitable due to specific hardware dependencies or scalability limitations.

Tendermint
Tendermint is an open-source platform for blockchain application development. Although Tendermint provides various well-documented examples, general documentation is limited. A benefit of Tendermint is that it satisfies the Byzantine Fault Tolerance (BFT) property. This provides various safety guarantees. For example, up to $1/3$ of the nodes in the network can be explicitly malicious without the network being compromised [20]. However, BFT is achieved through Tendermint Core - Tendermint's consensus algorithm - which cannot scale enough to suit this project. The information in this paragraph was gathered from [20].

+ Satisfies Byzantine Fault Tolerance;
+/- Development documentation is available with examples but general documentation is limited;
- Consensus algorithm is not suitable due to scalability limitations.

BigchainDB
BigchainDB is an open-source platform that offers tools for building both public and private networks with the characteristics of a (MongoDB) database and a blockchain together. It is properly documented. It provides decentralisation, Byzantine Fault Tolerance and immutability like a blockchain, but it also provides high transaction rates and querying functionality like a database [4]. However, BigchainDB achieves BFT by using a Tendermint consensus algorithm that cannot scale enough [4].

+ Widely used, well-documented and maintained;
+ Querying capabilities powered by MongoDB;
- Consensus algorithm is not suitable due to scalability limitations.

Conclusion
After considering the frameworks above, it appears that Hyperledger Fabric is the most suitable framework for this project.

Although Hyperledger Sawtooth seems to be a valid option as well, its built-in consensus algorithms are not suitable for the project. Implementing another consensus algorithm might be a possibility but seems unreasonable given the duration and other objectives of the project.

3.3. Hyperledger Fabric Framework

The following section will explain various technical terms necessary to understand a Hyperledger Fabric network (version 1.4). For a more in-depth and detailed explanation, please look at the Hyperledger Fabric documentation[1].

3.3.1. Network Concepts

Channels
All communication within a Hyperledger Fabric network is done through channels. Parties join a channel to communicate with other parties in the same channel. It is possible to have a network that contains multiple channels with different purposes alongside each other but every channel does have its own ledger. Every channel's ledger contains at least one configuration block with information about the channel; for example, the organisations participating in it. Consequently, when a new party joins a channel, a new and updated configuration block needs to be appended to the ledger of the channel.

[1]"https://hyperledger-fabric.readthedocs.io/en/release-1.4/

Organisations
The term ‘organisation’ in a Fabric network is used to define a real-world organisation that participates in the network. Every organisation has its own set of peer nodes, and multiple organisations together can belong to a consortium, which simply is a group of organisations.

Peer Nodes
One of the key concepts in Hyperledger Fabric is a peer. A peer, or peer node, is a participant in the network that hosts its own copy of the ledger, world state (see below), and chaincode (see Section 4.1). A peer can be part of multiple channels simultaneously, and as such can also have multiple ledgers. Access to the chaincode and the ledger(s) can be obtained through peer nodes. This interaction happens through an API that is accessed via a peer.

Peer nodes have many responsibilities that are crucial for operating on the blockchain. An example of this is proposal endorsement. A transaction proposal, which can be a request to append data to the ledger, should be endorsed by peers. The importance of endorsements will be explained in more detail in Section 3.3.3.

Anchor Peer Nodes
In Hyperledger Fabric, there is a special type of peer called ‘anchor peer’. Anchor peers are the only peers that can communicate with peers from other organisations. They communicate with other anchor peers to exchange information about all peers they know about. The anchor peers then relate this information back to the peers in their own organisation, which assures that also those peers know about all the other peers active in the channel. As such, all peers contact their respective organisation’s anchor peers to learn about the other peers in the network. Communication is done by means of a gossip protocol, described in Section 3.6.

Orderer Nodes
Orderers are responsible for ordering transactions and putting them into a block, which can then be added to the ledger. The specific moment when orderers play a role in the network is explained in Section 3.3.3.

There are several methods of managing transaction ordering. Though the results may be identical, the underlying methods of reaching consensus vary. Hyperledger Fabric supports three different implementations for reaching consensus in the ordering service (Figure 3.2), namely Solo, Raft and Kafka. Solo is mostly used during development, however due to its single orderer point of failure, it is in production replaced by Raft or Kafka. Apart from ordering transactions, orderers also maintain a list of all organisations, and enforce read, write, and configuration access to channels.

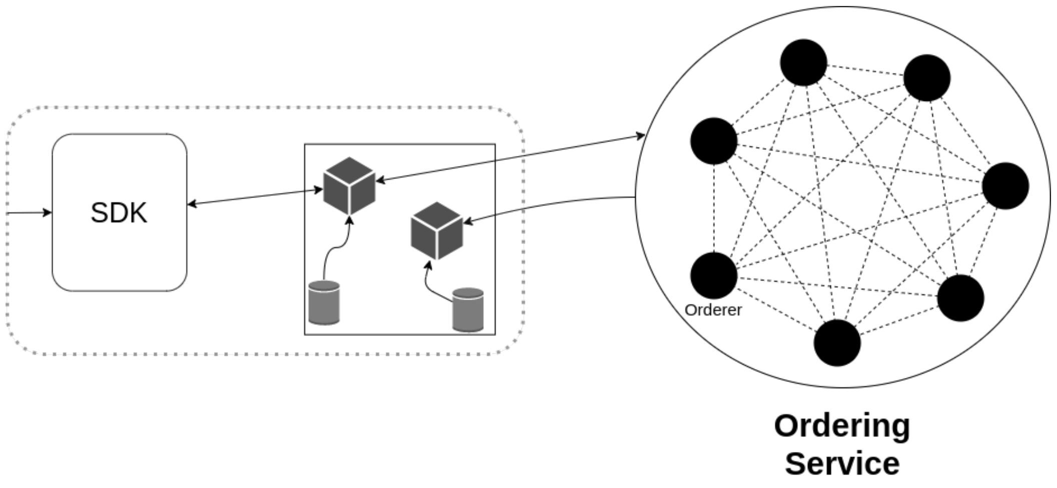

Figure 3.2: An orderer receives the proposal from a peer. A block is sent back from the ordering service to the peers in the channel.

The World State
The world state is a database which stores the current state of all transactions. Note that this is different from the ledger, which stores the entire history of all transactions. Like ledgers, a copy of the world state is stored on every peer contributing to the channel. By using the world state for querying transactions, it is not necessary to traverse the entire ledger.

3.3.2. Network Rules

Membership Service Provider
A Membership Service Provider (MSP) is a component that manages user authentication and certificate validation. The MSP allows organisations to specify different identity classes. These classes are client, admin, peer, and orderer. Users that submit transactions on the network are classified as clients, and users that manage administrative tasks, such as letting peers join a channel or configuring channel updates, are classified as admins [8].

Policies
Policies determine what kind of classification an identity needs to perform a certain action. The MSP uses the policy as a function to determine whether a certain identity has the right certificate for performing a certain operation. Fabric implements two types of policies:

- SignaturePolicy allows for specification of rights, using AND, OR, and NOutOf constructs. For example:

```
Writers:
    Type: Signature
    Rule: ”OR(’DGO0MSP.admin’, ’DGO0MSP.peer’, ’DGO0MSP.client’)”
```

- Implicit Meta Policy allows for specification of rights, deeper in the configuration hierarchy. This means that it is constructed implicitly based on the current configuration, which is defined by the SignaturePolicy. For example:

```
Writers:
    Type: ImplicitMeta
    Rule: ”ANY Writers”
```

Policies make distinctions between three types of operations, namely reading-, writing- and admin operations.

3.3.3. Transaction Endorsement Process

Hyperledger Fabric has designed and implemented a process which must be gone through when a transaction is submitted. First, a new transaction is created by a user, for instance by filling out a form on a web page. This form is processed by an application, which is connected to the network using Fabric’s SDK. The application decides which peers to send the processed form data to, in the form of a transaction. This is explained in detail in Section 4.1.

The peers that are configured to be endorsing peers, will receive the proposal and process it by executing chaincode. If the chaincode executes successfully, the peer endorses the proposal. If the ‘endorsement policy’ is satisfied, meaning a specific combination of peers endorses a proposal, the proposal becomes a transaction. If not, the proposal is rejected. This process is displayed schematically in Figure 3.3.

After fulfilling the endorsement policy, the transaction is sent to an orderer; together with the other orderers, they form the ordering service. The ordering service orders transactions into blocks per channel. Once a block is finalised by the service, it is sent to all peers in the respective channel. Peers receiving a new block validate its contents and add it to their copy of the ledger. Once this is done, a confirmation message will be returned to the application that proposed the transaction.

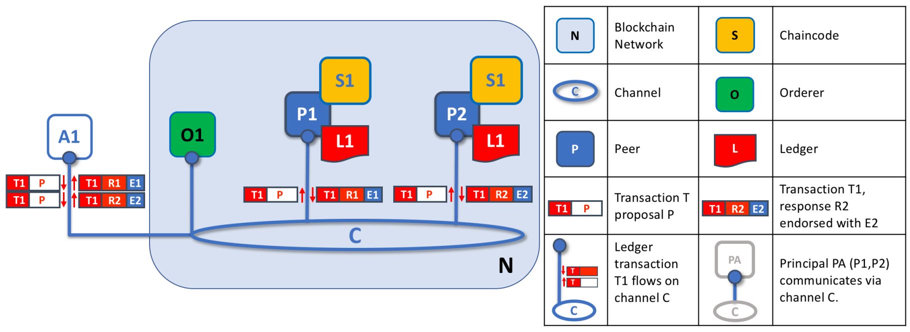
A1
T1 P
T1 R1 E1
T1 R2 E2
O1
P1
S1
L1
P2
C
N
Blockchain Network
Chaincode
Channel
Orderer
Peer
Ledger
Transaction T proposal P
Transaction T1, response R2 endorsed with E2
Ledger transaction T1 flows on channel C
Principal PA (P1,P2) communicates via channel C.

Figure 3.3: Transaction proposal of peers: This figure displays the process through which application A1 submits a transaction proposal to endorsing peers P1 and P2 and these peers return a transaction response i.e. endorsement or rejection [10].

3.4. Mandate Register Model

3.4.1. Components

The mandate's components are displayed schematically in Figure 3.4, and elaborated upon below.

Channels

The mandate register network consists of two channels. One of the channels, `syschannel`, is only for the orderers. The other channel is used for interaction of the network members, which are the distribution grid operators. Hence, this channel is named `dgochannel`.

Organisations

In the mandate register network there are seven participating peer organisations. The organisations are named DGO0 up to DGO6 during the development. In production they could be renamed to respective distribution grid operator names such as Liander or Stedin. Each organisation has two peers defined. In each organisation, one of the two peers is designated as the anchorpeer.

Orderers

The orderers belong to their own orderer organisation and this organisation runs on the separate `syschannel`. The mandate register network has a total of seven orderers defined. The reason for seven orderers is to allow each of the seven distribution grid operators to host an orderer.

Databases

Hyperledger Fabric supports LevelDB and CouchDB for an organisation's peers' databases [10]. For the register's world state, CouchDB was chosen because it supports richer queries on data than LevelDB does, since it allows for modelling the data as JSON objects instead of key-value pairs. CouchDB also supports indexing, which means querying on attributes other than the key can be done quickly.

To view a peer's world state in a web browser, CouchDB's Fauxton Interface can be used. The interface shows all databases and records stored on the peer and allows for editing them as well. For usage, see Appendix B.

3.4.2. Policies

The mandate register network has three types of policies defined in the configuration. These are the organisation policy, application policy and orderer policy.

Organisation Policy

This policy allows for boolean operations such as `AND` or `OR`. There are policies defined for readers, writers and admins. For all DGOs, admin operations can only be performed if the identity carries an admin certificate. Currently, the rule is an `OR`, therefore the signature of one admin from DGO0 is enough.

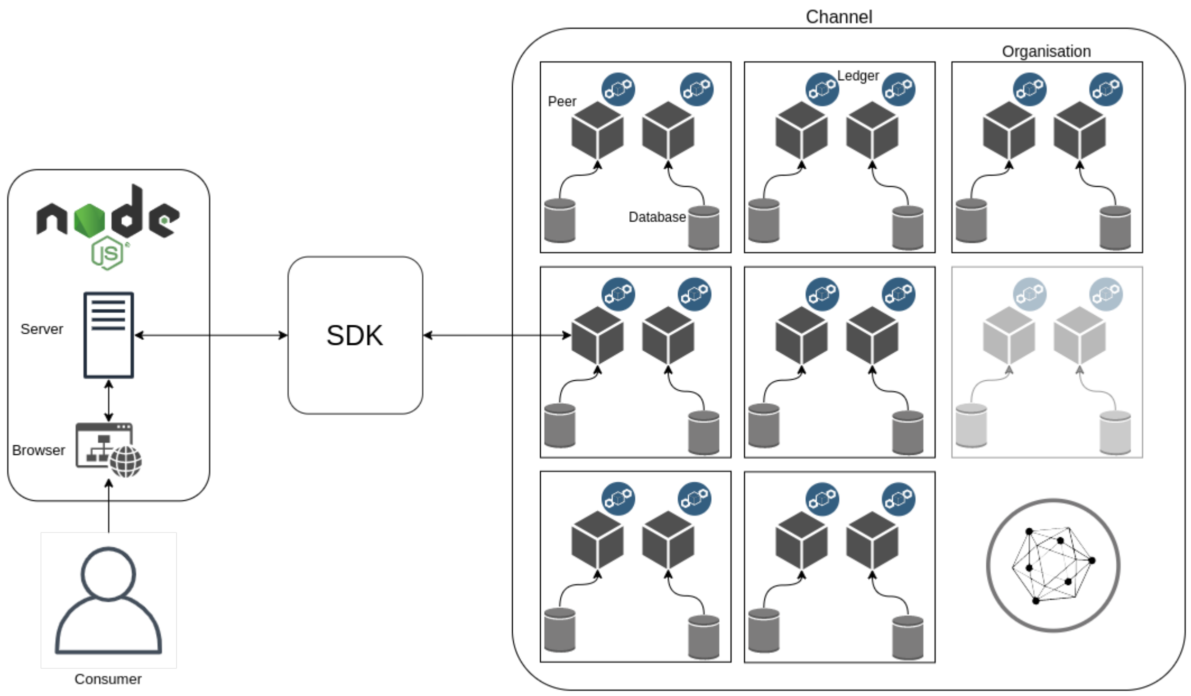

Figure 3.4: This image shows the topology of the mandate register model. Left is the GUI and the server, which connect to the network using the SDK. The network consists of seven organisations. An extra organisation is greyed out, signifying that it can be added later.

Application Policy
This policy has the type `ImplicitMeta`. The policy for reader, writer, or admin are satisfied if `ANY` reader, writer or admin signs it. The readers, writers, and admins rely on the signature policy since the reader, writer and admin are defined there.

Endorsement Policy
As mentioned earlier in Section 3.3.3 every transaction to the network, using the chaincode functions, has to fulfil the endorsement policy. The endorsement policy describes which of the organisation's peers has to endorse it. Furthermore the policy also defines the set of endorsers needed for the specified chaincode. This set can be specified using `ANY`, `OR` or `NOutOf` operators and consists of peers part of the organisations.

Orderer Policy
In the configuration of the network, one of the policies for orderers includes `BlockValidation`, which is required before sending blocks to the peers. The `BlockValidation` has to be signed by `ANY` reader from the orderer organisation. The readers for the orderer organisation are defined under the organisation policy.

3.5. Network Binaries

Running a network means running nodes in Docker containers, which is done via Docker Compose[2]. The containers have to be specified in the Docker Compose configuration files. Docker uses these configuration files to launch the containers, which will be executing the Docker images that are provided by the framework.

Before a network can be started, it is necessary to generate both cryptographic material and 'channel artifacts'. Channel artifacts are binary files that configure the channel. The model described in Section 3.4.1 is written down in the configuration files `crypto-config.yaml` and `configtx.yaml`. In `configtx.yaml` the profile section describes the mandate register model.

[2]Docker Compose is a tool for defining and running multi-container Docker applications [6]

Fabric provides a binary executable called `cryptogen` that uses `crypto-config.yaml` to create cryptographic keys for all identities in the organisations. Note that `cryptogen` should not be used during production; see Section 6.4. Fabric also provides a binary executable called `configtxgen`, which uses `configtx.yaml` to create channel artifacts. These include the following files:

- `genesis.block`: The genesis block for the orderer channel `syschannel`.
- `channel.tx`: A transaction that configures the `channel`.

3.6. Container Communication

As mentioned before in Section 3.5, HyperledgeFabric uses Docker containers to run each network component. Every orderer, peer, peer database, and chaincode runs in separate containers. Therefore, every peer in the register requires at least three containers; the peer itself, a database container, and a chaincode container. Within the network there are also certificate authority (CA) containers. A single CA container is created for every organisation. Furthermore there is the command line interface (CLI) container. This CLI can be used to execute commands from other peers' containers. Communication between containers is done using Transport Layer Security (TLS), which Hyperledger Fabric supports natively. For a detailed explanation on setting up TLS communication in Fabric, refer to [9].

The containers running Fabric can run on physically separated host machines as well. This is necessary because the purpose of the framework is to maintain a blockchain network, which should be able to be distributed. Although this can be achieved in different ways, the chosen approach for this project was Docker Swarm. A swarm is first created on a single machine, which other machines can then connect to using a generated token. This way, each machine runs a part of the network.

Communication between peers is done using Fabric's gossip protocol [11]. Note that messages sent with the gossip protocol are also encrypted with TLS. The gossip protocol allows for discovery of online peers and detecting when a peer becomes unreachable. Also, if a peer has an outdated ledger, for instance upon joining a channel or after going offline, other peers use the protocol to send the updated ledger. The gossip protocol ensures peers are in sync, and maintain the same ledger and world state. The concept of the 'world state' was explained in Section 3.3.1.

4 Implementation

In this chapter, the implementation of the code written for the product is elaborated upon. This includes the functionality of the chaincode and the application. Furthermore, testing of the code is discussed.

4.1. Fabric Background

Chaincode

Chaincode is the code that describes the business logic of the network. It is Fabric's way to refer to what in common blockchain terms is called a 'smart contract'. Chaincode can be invoked from a client application external to the blockchain [10]. It accesses world state data or ledger data by means of a query or modifies it by means of an invoke.

Before chaincode can be executed on a network, it needs to be instantiated. To instantiate chaincode, it needs to be installed on at least one peer per organisation that wants to use said chaincode. After installation, the chaincode can be instantiated on the channel along with a specified endorsement policy. This policy describes how many and which endorsers are needed for the chaincode. After instantiation, the chaincode can be executed by any peer that has it installed.

Fabric Application Model

Applications can submit proposals to the ledger or query data by interacting with the network. Hyperledger Fabric provides SDKs for connecting with the network in Java, Go and Node.js. For all SDKs, the workflow is as follows:

1. The SDK registers a user for an organisation. The CA creates the certificates for the designated user and stores them in the file system. These certificates are JSON objects containing cryptographic material.
2. The SDK creates a client object for the registered user. It does so by parsing the network profile. In this configuration file, the network components that the SDK needs to be aware of are listed. These components are channels, organisations, orderers, peers, and CAs. The client object is then used to submit transactions to the blockchain.

4.2. Mandate Register Logic

4.2.1. Chaincode

Writing chaincode in Go

The register's chaincode is written in Go. By default, Hyperledger Fabric offers chaincode support for Go, Node.js and Java. Initially, all three were tried, though the most progress was made with Go. This was due to the fact that Fabric is written in Go, has supported Go since its early stages, and its documentation contains elaborate examples written in Go. Both Node.js and Java were supported later and were featured less in the Fabric samples repository[1].

[1]Fabric samples repository is a collection of example Fabric projects [14].

Mandate Structure
The mandates in the register are formatted according to the specifications in the chaincode. The mandate object is still quite primitive due to the fact that it is only a prototype. Enhancing the object by adding more attributes is possible.

A mandate is formatted as a Go ‘struct’ containing the following fields:

1. ObjectType: For every mandate, this is equal to ”mandate”. In the future, this text could be used to indicate other types of mandates, such as for example ”finance” for financial data.
2. ClientID: The ID of the consumer who grants or revokes the mandate. In the current implementation, this can be any valid integer. See Section 6.4 for a better alternative using EAN codes.
3. PartyID: The ID of the party to whom the mandate is addressed. In the current implementation, this can be any valid integer.
4. MandateType: The purpose for which this mandate was given. The intention was to have standardised types relating to, for example, energy contracts or research. Currently, any string is accepted.
5. Permission: A boolean indicating whether the consumer grants or revokes permission. Currently, ”true” and ”false” are accepted respectively.
6. Time: The time this mandate was created, formatted as ”DD Mon YYYY HO ZON”. For example, ”02 Jan 2006 15 MST”. The date within the field gets created upon creation of a mandate.

Mandate Register Chaincode Functionality
The mandate register makes use of various chaincode functions in order to add or retrieve mandates. The following functions are implemented:

- createMandate: Add a mandate to the register.
- init: Initialise the chaincode.
- queryMandateByClient: Query all mandates granted by a certain client.
- queryMandateByParty: Query all mandates granted to a certain party.
- queryMandateByClientAndParty: Query all mandates granted by a certain client to a certain party.
- getHistory: Query all (previous) values of a mandate corresponding to a certain key. The construction of a key is explained in the next paragraph.
- getClientHistory: Query all (previous) mandates corresponding to a certain client.
- switchProviders: Add a mandate granting permission from a client to a party, and if present, revoke the client’s previous mandate (see Figure 4.1).

These functions were implemented using the Fabric Shim interface. Shim was used to create a ‘chaincode stub’ which is used to interact with the state of the ledger. It is also used to format the functions’ return values. These return values are either of the type `shim.Error` in case of an error - such as not passing enough arguments to a function call - or of the type `shim.Success` if a function executed without any errors.

Key Generation
Mandates are stored in a key-value store in which the key is a unique identifier and the value is a mandate object. Unique keys are generated based on the mandate itself, by converting `ClientID`, `PartyID`, and `MandateType` to Base64. When a new mandate with an existing key is added to the key-value store, for instance when revoking permission that was previously granted, the value of the old mandate is overwritten with the new value.

`Permission` and `Time` are not used to generate a key. If they were, mandates revoking access that was previously granted would have a different key and would therefore show up as different mandates in the world state. That would be problematic, as it would break the history functions described previously.

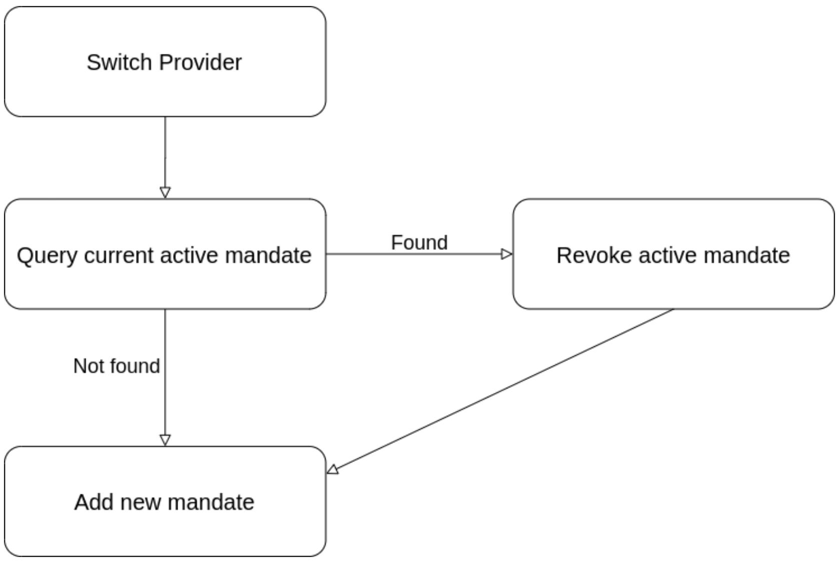

Figure 4.1: Flow diagram of the switch operation.

4.2.2. Mandate Register Application

Server and GUI

To allow consumers to easily submit and query mandates, the application needs a GUI. To provide such an interface, a basic Node.js server was set up. The server displays a web page with various HTML forms. These forms execute HTTP POST requests that are handled by the server's starting point, using Node.js' 'Express' library.

Network Interaction

After the HTML forms sent by the GUI are handled by Express, the Node.js server makes calls to one of Fabric's SDKs functions directly, except when adding mandates. For adding mandates, the server calls another function which invokes the chaincode but also attempts to contact different orderers when one becomes unavailable. This function is elaborated upon below.

Managing Orderer Failure

When an orderer becomes unreachable (Figure 3.2), the application needs to connect to another orderer to maintain network interaction. To realise this, a function was implemented with a variable orderer endpoint included in its parameters. This function attempts to invoke the chaincode and, if it fails, is executed again with another orderer endpoint. If this is successful, the function will keep using the last successful orderer. If it is not, the function will keep attempting to invoke the chaincode until every orderer in the channel is tried.

4.3. Code Testing

4.3.1. Chaincode Unit Tests

The `mandate_contract` chaincode, written in Go, is unit tested for 78.1% statement coverage. In order to test the chaincode, Testify Toolkit and Fabric Shimtest were used. Testify Toolkit provides the option to use detailed assertions within the test code and also provides a mocking tool. Fabric Shimtest provides a 'mockstub' used to mock the functions that would normally communicate with the chain or CouchDB. These unit tests are not run using continuous integration. Whether continuous integration can be used for these tests is unsure as currently an import in the code needs to switched in order to run the tests, see Section 6.3.

4.3.2. Node.js Testing

The Node.js server and its back-end have been thoroughly end-to-end tested. These tests were done during the verification of every merge request regarding the GUI. During these merge requests the code was also extensively reviewed by the developers. The back-end has not been unit tested as most of the functionalities within in the back-end rely on the Fabric Client API.

5 Evaluation

In this chapter, the performance of the network is evaluated. First the metrics to be evaluated are listed and the metric computation methods are explained. Then the topology of the physical network is specified. Finally, the results are displayed in the included tables and figures and they are discussed afterwards.

5.1. Performance Metrics

It is important to assess latency and throughput to measure the performance of a blockchain network. Latency is important because it clarifies how fast the network handles a single operation. Throughput is important because it clarifies how fast the network handles operations if they come in a consistent stream. In the implemented prototype, three types of operations can be distinguished: read, query and invoke. Read operations concern reading the blockchain, not querying external CouchDB databases. Consequently, query operations concern CouchDB and are evaluated as well. Finally, invoke operations concern adding data to the blockchain. These three types of operations are evaluated on both their latency and throughput.

To calculate latency and throughput, the following equations are used [12]:

```
Throughput = number of operations / total time in seconds
Latency = confirmation time - submit time
```

5.2. Procedures

For determining the throughput, a Node.js script is utilised. The script invokes the chaincode in a batch of 50 asynchronous transactions and calculates the throughput in transactions per second. This number was chosen because a larger batch of asynchronous transactions caused the orderer to overload.

For determining the latency, the calculated time difference is determined over 50 runs. For transactions, the `createMandate` chaincode is invoked; for queries `queryByClient`; for reads `getHistory`.

The difference between the `queryByClient` and `getHistory` functions is that `queryByClient` queries the world state of a peer (stored with CouchDB), whereas `getHistory` traverses the ledger of a peer. For `getHistory` this is necessary because the mandate history is not stored in the world state but in the ledger.

5.3. Evaluation Setup

The evaluation network consists of seven organisations distributed over three machines, as can be seen in Figure 5.1. Each organisation consists of two peers with their CouchDB instances and a CA server. To simulate the intended usage scenario, every machine hosts as many orderers as it hosts organisations. The three machines are all connected to the TU Delft Eduroam WIFI network.

The Fabric blockchain makes use of batches. This means that transactions are committed to the ledger by means of a buffer called a batch. Hence, the Raft-based ordering service requires configuration of `batchSize` and `batchTimeout`. The `batchsize` determines how many transactions are buffered before the orderer commits. The `batchTimeout` determines how long the orderer waits for new transactions before it commits a batch, regardless of its size.

The aforementioned parameters, namely `batchSize` and `batchTimeout`, remain untweaked and thus correspond to the configuration of the Fabric samples repository [1]. More elaboration on improving performance can be found in Section 6.4.

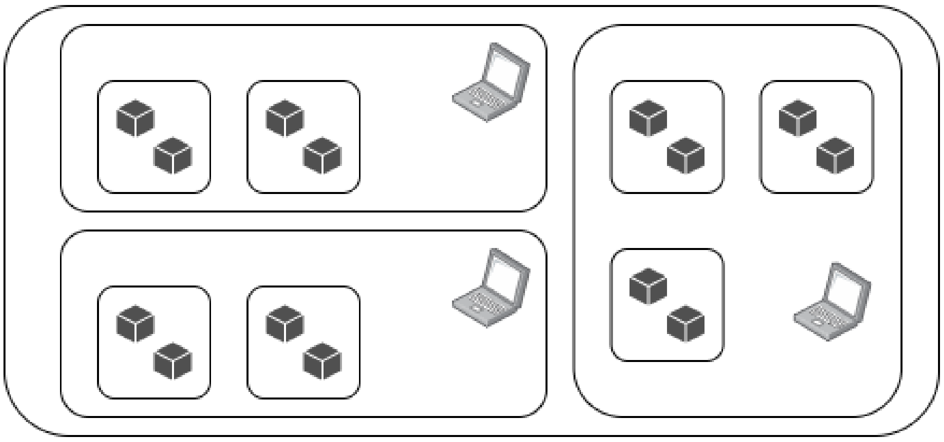

Figure 5.1: The mandate register network with seven DGOs is distributed over three machines. This setup was used for testing the network.

5.4. Results and Discussion

This section contains the results of the evaluation. The variance (σ^2), mean (μ) and 95% confidence intervals (CI_{95}) of the results are displayed in tables 5.1 and 5.2. All result sets consist of 50 samples. The histograms of the latency and throughput are displayed as well, in figures 5.2, 5.3, and 5.4.

	σ^2	μ	CI_{95}
Invoke	596	474	[467;480]
Read	121	396	[392;399]
Query	228	400	[396;404]

Table 5.1: Latency statistics in milliseconds

	σ^2	μ	CI_{95}
Invoke	0.55	16.86	[16.66;17.07]
Read	5.73	55.32	[54.66;55.99]
Query	3.81	48.44	[47.90;48.98]

Table 5.2: Throughput statistics in operations per second

[1] https://github.com/hyperledger/fabric-samples

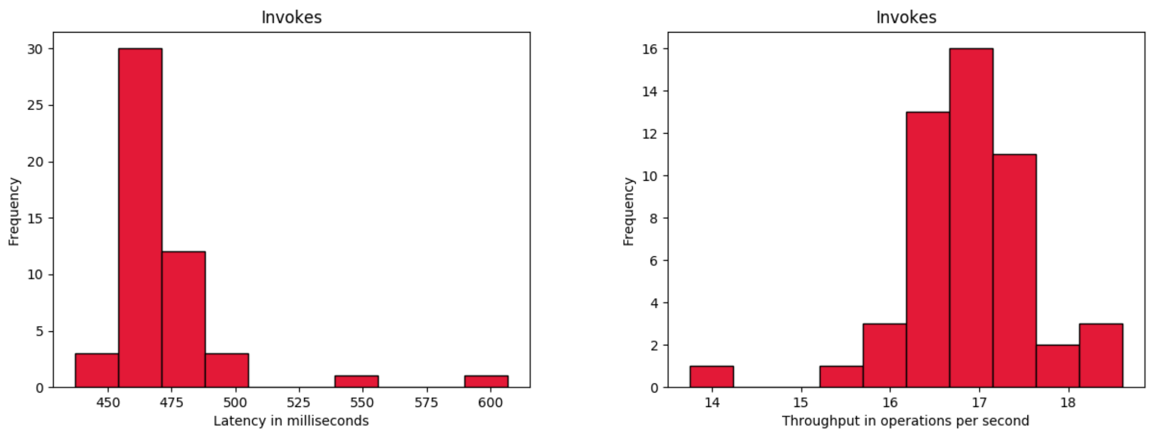

Figure 5.2: Histograms for invoke performance data

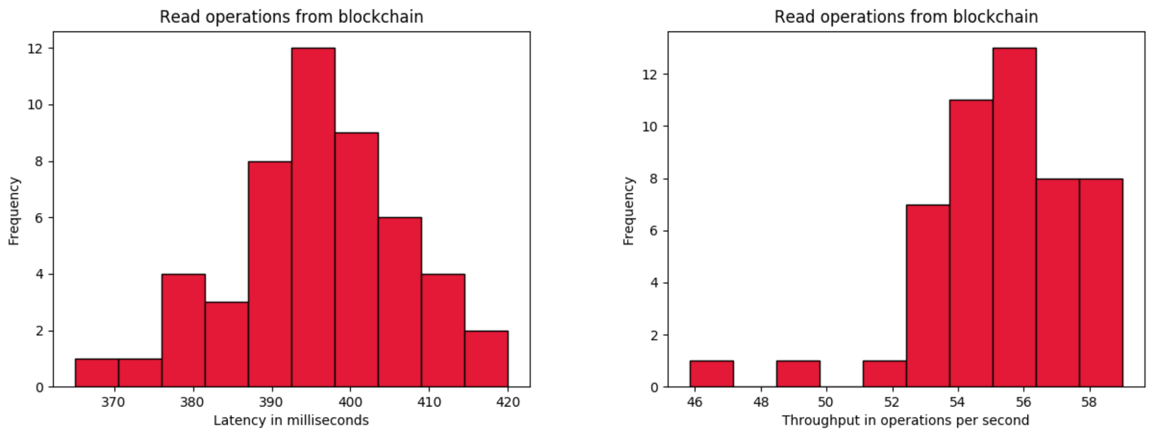

Figure 5.3: Histograms for read performance data

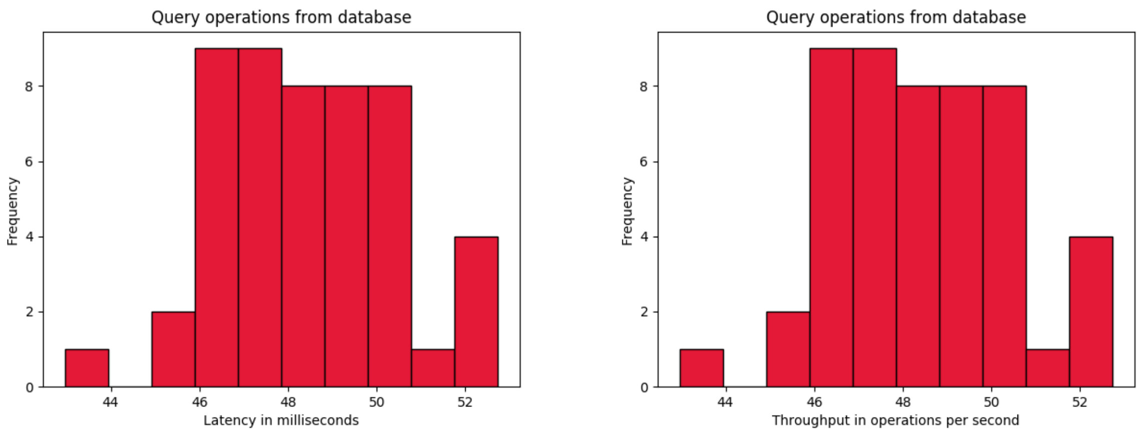

Figure 5.4: Histograms for query performance data

Discussion
From the fact that the 95% confidence intervals are narrow, and the look of the histograms, the con-

clusion can be drawn that the results are all relatively stable. Besides this, the means in tables 5.1 and 5.2 respectively also tell us that latency is relatively high and throughput is relatively low. The mean throughput value of an invoke tells us that the network processes 16.86 invokes per second on average, where Hyperledger Fabric networks are claimed to be able to reach a transaction throughput of 3000 per second [1]. The read and query operations reach a higher throughput because these do not alter the world state nor append data to the ledger, but they are still not as fast as they could be. The reason for this lack of performance has yet to be determined, however some presuppositions can be endowed.

To find out whether the Node.js script, which was utilised to obtain the results, caused a bottleneck, a script was run that consisted of Linux bash commands in order to avoid making use of SDK logic. This batch script gave very comparable results, thus the conclusion was drawn that the Node.js script does not cause a bottleneck and the performance obstructing factor must be somewhere in the network.

The `BatchSize` and `BatchTimeout` are parameters that concern the transaction ordering. `BatchSize` corresponds to an amount of transactions before a transactionblock is submitted. `BatchTimeout` is specified as the time the orderer waits before submitting the block. A larger `BatchSize` and `BatchTimeout` would allow for ordering more transactions per batch, since a batch then fits more transactions. This would increase throughput in a consistent stream of transactions due to less communication per transaction. On a single transaction, a higher `batchtimeout` does however cause a higher latency as well. A lower `BatchTime` and `BatchSize` would provoke the orderer to send the block to the peers earlier. The consequences of this are a lower latency but also a lower throughput due to the communication overhead. Finding an optimum between these parameters should increase performance significantly, but is dependent on the network demand.

The topology of the network used in evaluation could also be the bottleneck. The 14 peers and 7 orderers are distributed over only 3 machines. Using more machines to parallelise the workload could also improve performance.

6 Discussion & Future Work

In this chapter the success criteria are reviewed and ethical implications are considered. Then the process of development along with the experienced issues is laid out. The chapter is concluded with a section containing future work and according recommendations.

6.1. Project Evaluation

In order to evaluate the project, the success criteria defined in Section 2.4 are looked at. Three main success criteria were established; solving the problems defined in Section 1.1, meeting the design goals defined in Section 2.1 and completing the MoSCoW defined in Section 2.3.

6.1.1. Solving the Defined Problems

The first criterion has been met by implementing the mandate register and chaincode functions to retrieve a client's history and switch from provider.

Lack of transparency has been resolved through the implementation of the mandate register in combination with the functionality to retrieve one's mandates. This functionality can give a user an overview of all their granted mandates over time.

Besides having more transparency, the system also provides a quicker way to switch from energy provider. Parties involved in the switch will need less communication due to the fact the mandates are stored on the blockchain. Instead of having to verify each other's data they can consult the blockchain. The system also provides a function to have consumer's mandates switched from their last provider to a selected new provider.

6.1.2. Meeting the Design Goals

The second criterion can be split up in individual design goals; expandability, security and privacy, and GDPR Compliance.

In order to accomplish expandability, the register should support various use cases and should be scalable. Through the channel functionality of Hyperledger Fabric, multiple channels can be created, each with a different purpose. Different use cases can be executed on the same network, yet entirely separated through channels. Also, a mandate object - as described in Section 4.2.1 - offers flexibility in its formatting and can be extended easily. As for scalability, performance measurements were done and described in Chapter 5. Apart from this, invoking and querying the register can be done with every peer in the network, meaning that for example multiple instances of a Node.js application can function simultaneously.

The second design goal is security and privacy. Security and privacy were split up in four parts; 'verified data reading', 'verified data writing', 'authentication' and 'no central authority', as specified in Section 2.1. In order to guarantee that mandates should be immutable, the register was set up as a blockchain. A majority of the endorsing peers verify all mandates before they are added to the ledger. All parties in the network communicate securely through TLS, meaning no mandates can be read or written by unauthenticated parties. Unfortunately no authentication of parties has been implemented, thus it can not be verified whether a party is retrieving data belonging to itself. Lastly, mandates are added on a majority basis, meaning there is no central authority that can be corrupted.

The third design goal is GDPR Compliance. This goal was not fully met. It is still possible for grid operators to read mandates from other operators, and 'deleting' mandates is still not possible. Although the entire ledger is encrypted and inaccessible to unauthorised parties, and the effect of a mandate can be undone by revoking it, the current implementation is strictly speaking not GDPR compliant. A further elaboration will be provided in Section 6.4.

6.1.3. Completing the MoSCoW

MoSCoW

Must Haves

- Mandates must be stored on a blockchain
- The blockchain must be permissioned, such that validators will only be added on a permissioned basis.
- A consumer must be able to see an overview of their mandate(s).
- A consumer must be able to grant mandates for specific parties to access specific data.
- A consumer must be able to revoke mandates for specific parties to access specific data
- A consumer must be able to switch between energy providers.
- A producer, meaning a party that produces energy, must be able to query the network to verify if a consumer gave them a mandate to access specific data.
- Validator nodes must be able to validate users' requests to append data. A request is considered valid if:
  - The user making the request is authorized and authenticated.
  - The request is properly formatted according to the protocol that we specify.
- Validator nodes must be able to append valid requests to the blockchain using a consensus algorithm.

Should Haves

- The protocol should allow granting/revoking access to different types of data.
- A user should be able to authenticate using DigiD or another authentication method.
- A user should have a graphical user interface to give and revoke mandates.
- A user should have a graphical user interface to show a history of their mandates.
- A user should be able to automatically revoke mandates for a previous energy provider when switching to a new one.
- The network should be able to add and remove validators.

Could Haves

- A user could be asked for confirmation when they initiate a mandate modification.
- The consumers' mandates are categorized under their EAN-code.

Won't Haves

- The blockchain will not support prosumers (consumers that produce energy as well).

- The blockchain will not support any other type of asset beside energy meters.

Non-Functional Requirements

- The system will be properly tested. Tested means that our own code will be:
  - Software tested and measured using code and branch coverage.
  - System tested for a small mocked network.
- The blockchain will be set up using frameworks.
- The system will be developed using version control Git.
- The code will be sent to the Software Improvement Group (SIG) for evaluation.

Explanation of accomplished requirements
Mandates are stored on a permissioned blockchain through implementation of Hyperledger Fabric. An overview of the consumers' mandates is visible through the application UI, which allows for granting, revoking and switching as well. Parties can also see all of their mandates through the UI. The MoSCoW states that validator nodes must validate a request by means of checking request format (and authentication, see paragraph below). In the current prototype, endorsing peers enforce correct request formatting through chaincode endorsement. Validator nodes must append requests by means of a consensus algorithm. This is implemented in the prototype through the ordering service which functions with a Raft consensus protocol.

Different data types are supported through a mandate type variable. As mentioned before, the UI in the application allows for granting and revoking mandates and it also allows for viewing mandate history. Switching energy suppliers through the UI automatically revokes mandates for a previous provider. Through a batch script, which is discussed in Appendix B, organisations, and thus validator nodes can be added to the network dynamically. Removing organisations and thus validators is made possible through Fabric binaries included in the repository, this is however not automated.

The network was setup using a framework called Hyperledger Fabric as mentioned before. The code was developed using Git via Gitlab. The code was sent to SIG indeed.

Explanation of missed requirements
After discussing with the client and the supervisor of the project, the conclusion was drawn that implementing authentication was not achievable within the project duration. Consequently, authentication is not featured in the prototype. Besides authentication, confirmation before a mandate is submitted is not implemented in the project and neither is consumer categorisation by means of EAN code. The latter requirements were also missed due to time constraints. For an elaboration on these topics, please examine Section 6.4. The requirements in the 'won't have' section are not implemented for obvious reasons. In the non-functional requirements it is stated that the code will be tested using branch coverage. Unfortunately, Go's testing library does not support this, thus statement coverage was used for.

6.2. Ethical Implications

6.2.1. Authentication

As may have become clear, no secure authentication was developed during the project period. This leads to obvious security flaws within the prototype. Anyone with access to the system could abuse the obtained authority to change or even add new mandates. For example a new contract could be added which gives a party X access to all the data of consumer Y. The only restraint right now, is that this newly added transaction is visible to all of the parties participating in the network. This is not much of a constraint for a system that can have financial consequences. Not to mention the usage data of consumer Y being accessed without their consent. However, after the implementation of an authentication method this should be no longer an issue, as every transaction on the network will have to be signed by both party X and consumer Y in order to accept the transaction.

6.2.2. GDPR

With correct usage of the prototype consumers can control those who have access to their private data. Since the implemented system is blockchain based, consumers can see all previous mutations related to their mandates. Unfortunately it also means that the mandates are immutable, thus no deletion of their mandate is possible. This is required under the right to erasure which is article 17 of the GDPR [7]. Not abiding by the regulation is illegal and should therefore be resolved before using the prototype.

6.3. Process

Fabric Configuration

After the research phase of the project, the development of the network started. The first few days were spent installing the requirements needed for running the Hyperledger Fabric framework. Installing the requirements on Windows was slightly more difficult than expected. Once this was done, sample projects were used to get familiar with the framework. The goal was to start with building a simple network within the same week, but it soon became clear that this would not be feasible. To build a network using the framework, several files need to be configured. These configurations are complex because they are interconnected and depend on each other. While trying to run the network, during the first few tries, many errors were encountered. Most of the errors were similar in the sense that they related to the configuration files. The true difficulty lied in that the same error was detected, yet the errors related to different configurations in different configuration files. Therefore, in order to build the network more in-depth knowledge of the framework was needed. While this took more time than expected, it turned out to be very useful. Later in the process, several other errors relating to the network were solved quicker due to the understanding gained of the configuration files in the starting phase of the process.

Chaincode

Once the network was configured, the logic of the network - the chaincode - was to be implemented. Hyperledger Fabric supports three languages for the implementation of chaincode, namely Java, JavaScript, and Go. In the process of understanding Fabric's chaincode, all three languages were tried, and eventually the decision was made to use Go. The process of setting up a sample chaincode proved to be quite difficult and took a couple of days. After said chaincode could be executed successfully, development of the mandate register's chaincode was done throughout the rest of the project's duration.

Networking

After the initial network was able to function on a single machine, it needed to be possible to distribute the containers to other machines. It was decided to attempt creating a network over two machines, for simplicity. In order to implement this, the documentation of Hyperledger Fabric was consulted. Surprisingly, there was no mention of using the framework on an external network, meaning on multiple machines. This implied that, in order to finish this crucial part of the implementation, other unofficial sources were to be consulted. While some sources pointed to using paid services, eventually a few were found which pointed to Docker Swarm (see Section 3.6).

Unfortunately, the configuration files which were used to run a network on a single machine did not work with Docker Swarm. The solution was to change the network configuration in the docker-compose files. The network had to be specified as an external network. This network had to be created before the containers were launched. This was tested to run on a single device. After this, the network needed to be distributed over a network of multiple devices. For Linux, this was rather straightforward, apart from the minor problem that Docker does not yet support the latest version of Linux at the time of writing. However, networking on Windows was non-trivial. On Windows, Docker uses Windows Subsystem for Linux (WSL). Therefore, Docker did not run directly on Windows which seemed to cause a problem with ports. Docker Swarm needed certain ports to be open in order to communicate. However, opening Windows's ports lead to nothing, and eventually, it was decided to drop Windows networking support. By the time the decision was made, Linux networking functioned fully, and the pressure of implementing additional features increased. This was not a problem since it was not a requirement to have networking enabled on Windows. This did mean that some machines used during the development needed to switch to Linux.

In hindsight, configuring Fabric to work on a distributed network does not seem like a particularly difficult task. However, the lack of documentation and decent tutorials made it a challenge and a source of frustration. Most explanations were useful to some extent, although none solved the problem entirely. The solution was based upon information which was scattered over a range of different articles and technologies. Therefore, implementing networking took a long time, considering the duration of the project.

Application
It was decided that interacting with the network from the outside was unfeasible through the command line. Therefore, the application was created. Initially, the application used the `fabric-network` [1] API to communicate. However, this API was found to be too limited in its options, in particular because it did not allow specification of the orderer that would be contacted for proposals. Therefore, a switch was made to the `fabric-client` API, which provided more fine-grained control and did allow orderer specification. Eventually, the application was connected to a Node.js server so that the register could be accessed through a web interface.

Testing
Besides implementing features and configuring the network, the written code had to be tested.

Initially the chaincode's code was tested and this seemed to come with no troubles beyond the fact that there was no experience yet with testing Go code. However, once started with testing it was discovered that the imports used for Shim and Peer, `github.com/hyperledger/fabric/core/chaincode/shim` and `github.com/hyperledger/fabric/protos/peer`, no longer exist in Fabric's Github repository. This meant that the Go code could not be tested, for these imports need to be present in the `GOPATH` of the system. After searching for the imports, new locations of both dependencies were found, namely `github.com/hyperledger/fabric-chaincode-go/shim` and `github.com/hyperledger/fabric-protos-go/peer`. Replacing the old, unavailable imports with the new ones fixed the problem of testing.

Sadly, using the new imports caused the network to throw an error indicating that the new imports could not be found when installing chaincode on the peers. Attempts to get either the tests working using the old imports or the network using the new imports did not succeed. Thus, a question about the matter was posted on Hyperledger's Rocket Chat. The replies indicated that the new Shim import paths were not supported by Fabric yet. Upon discovery, it was decided that both old and new imports were to be included in the respective Go files, keeping the appropriate imports enabled and commenting the others, depending on the task.

During the testing of the chaincode it was also discovered that the mock implementation of a stub provided by Fabric was incomplete. The two following functions were necessary to test the code and were not implemented: `getQueryResult` and `getHistoryForKey`. These functions were used particularly often in the chaincode. The lack of implementation meant that querying and history functions could not be tested fully, for mock functions were needed to do so.

In an attempt to make unit tests work, a workaround for `getQueryResult` was found. In a non-approved merge request, FAB-5015[2], an implementation of a `getQueryResult` mock was found. FAB-5015 was refused by the Fabric development team because it recreated CouchDB functionality within Go, which to them seemed like a 'slippery-slope'.

However, by recommendation of the project's supervisor, FAB-5015's implementation was added to `mockstub.go` located in `\$GOPATH/src/github.com/hyperledger/fabric-chaincode-go/shimtest`. Adding this dependency made it possible to test querying functionality fully.

For `getHistoryForKey`, no mock implementation could be found.

Once the application was created, it had to be figured out how to test its code. The conclusion was drawn that much of the code relied on the Fabric Client API which would have to be mocked out. Figuring out how to mock these functionalities would take quite some time, which was not available because the GUI was implemented last. As it is untested, there could be bugs present in the code. In the case a bug is present in the code it would result in not being able to use the application layer on top of the network. The network itself will still function correctly and can be used through the command

[1] https://hyperledger.github.io/fabric-sdk-node/release-1.4/module-fabric-network.html
[2] https://jira.hyperledger.org/browse/FAB-5015

line interface. Thus it was decided to not test this code as it had already been end-to-end tested during the merge requests of the code to verify it behaves the way it should.

6.4. Future Work & Recommendations

Performance

Performance evaluation of the network (Chapter 5) shows that throughput and latency must be improved greatly to be able to support a large number of users and organisations. Having more organisations - and therefore more peers - results in a higher latency.

The cause of these low transaction rates and high latencies has yet to be determined. Comparisons between invoking through the application and the CLI indicate that the application is not the bottleneck, since both methods have an approximately equal performance. A first step in improving performance could be optimising batch parameters such as `batchSize` and `batchTimeout`. Another measure that could be tried to improve performance is hosting every node on a separate machine. As seen in the evaluation topology in Chapter 5, the evaluation was done on 3 machines instead of 21; 7 times 2 peers plus 7 orderers.

Replacing Binaries

All cryptographic material is currently generated using a binary executable provided by Hyperledger Fabric. The Hyperledger documentation[3] indicates that this binary, among others, should only be used during development. Therefore, it is suggested to generate cryptographic material in a different manner as well as replacing other binaries. Fabric supports generating certificates using its Certificate Authority (CA), which it recommends to use in a production environment, instead of binaries.

EAN Code Support

Currently, consumers in the register are identified by a random integer. In the current market, a client is identified by the EAN code of their smart meter. To reflect the market, it would be preferable to identify consumers by an EAN code instead of a random integer. This feature can be added once authentication has been added; after a consumers has been authenticated, their EAN code can be fetched from a database available to DGOs.

Use Milliseconds for Mandates

Currently, the time stored in a mandate object is specified in hours. This was done because executing chaincode on different machines resulted in slightly different results when using milliseconds. However, this is undesirable because it should be possible to update mandates in a shorter time frame. Therefore, the underlying problem of using time within chaincode should be fixed such that mandates can be ordered by milliseconds instead of hours. A possible solution would be to pass the time along with the proposal, and have peers verify that this time is within reasonable bounds.

Adding an Orderer Dynamically

The network supports adding an organisation while it is running. Unfortunately, orderers cannot be added during runtime. As the orderers have been divided over the seven DGOs, it would be best that adding an orderer is supported along with adding an organisation. When implementing this, the Raft protocol configuration needs to be updated as well. A starting point for implementing this functionality would be to find out how to modify the `syschannel` during runtime, as this is essential to update the ordering service.

GUI

The application's GUI currently has a basic design that could be improved upon significantly. Enhancing the server, such that it hosts multiple web pages instead of one, would make the interface more intuitive. Moreover, the server's responses to submitting data are formatted in plain JSON; they should be displayed in a more readable fashion.

[3]https://hyperledger-fabric.readthedocs.io/en/release-1.4/

Authentication
In order to deploy the mandate register in production, consumers need to be authenticated. Without authentication, it cannot be verified that a mandate was submitted by the consumer it refers to. In the initial project description, authentication was projected to be done using DigiD. In the first week of research, it was decided that IRMA[4] would be used instead. A few weeks later, it was determined that implementing authentication was beyond the scope of the project due to time constraints.

During the project, IRMA was considered to be used for authentication. However, ideally DigiD would be used for authentication.

Platform Independent Networking
Networking currently works on Linux systems only. Trying to connect with a network on a Windows system does not work yet (see Section 6.3). Networking has not been tested on MacOS. The network should be more platform independent to be a more complete product. The development team suspects that networking on Windows does not work due to Docker Swarm failing to connect.

Hyperledger Fabric Version
The register was developed using Hyperledger Fabric 1.4.4. During development, version 2.0 of Fabric entered its Beta phase. Version 2.0 adds potentially useful functionality. Examples of such features are a new chaincode lifecycle and private data enhancements. Not upgrading could also lead to features losing support, such as imports that are moved. Thus looking into version 2.0 of Hyperledger Fabric could be quite fruitful. The Fabric 2.0 documentation[5] provides explanation on how to upgrade to a newer version.

GDPR
It was established early that mandates are personal data and therefore fall under the GDPR. Research was done on making a blockchain implementation GDPR compliant. This meant it should be possible to remove mandates upon request. Different solutions to this problem were theorised, though none were deemed feasible for implementation within the project's duration. One of these solutions would be to use Fabric's private data implementation to send private keys off chain. If a client would then issue a deletion request, parties involved would delete said private key. Once a new mandate would be added by that client, a new key for encryption could be created and distributed. After various consultations, it was decided to minimise the risk of comprising the GDPR. However, complete GDPR compliance should still be implemented.

[4]https://privacybydesign.foundation/irma-en/
[5]https://hyperledger-fabric.readthedocs.io/en/release-2.0/

7 Conclusion

A mandate register was built using the Hyperledger Fabric blockchain framework. It allows individuals to grant or revoke third parties' permission to inspect specific data of the individual. The register was designed in such a way that it solves the lack of transparency regarding data access, and makes switching energy suppliers quicker for consumers.

While developing the mandate register, three design goals were kept in mind; expandability, security and privacy, and GDPR compliance. Although the register can be used for different use cases, and is scalable in that sense, its lack of performance hinders deployment. As for security and privacy, the immutability property of blockchain and endorsement policies of Fabric ensure that data can be read and written reliably. However, authentication of consumers was not implemented due to time constraints. Finally, combining GDPR compliance with blockchain technology appeared to warrant its own project and was not completed due to time constraints either.

To conclude, the current mandate register fulfills a majority of its design goals and requirements but requires additional development in order to be deployed. A fully developed mandate register will, without doubt, grant a better grip on energy.

A

Info Sheet

Grip on Energy

Presentation date: 6 February 2020

Project Description

Challenge:
The challenge of this project was to create a blockchain register which holds consumers' mandates. Mandates are authorisations, signed by individuals, that give third parties permission to inspect their data. The main goal was to make switching between energy providers possible through the register.

The project was commissioned by CGI, a globally active IT consultancy company.

Research:
During the research phase our focus was on blockchain, different consensus protocols, blockchain frameworks and the way the energy market is setup.

Process:
We tried using agile but unfortunately the setup proved quite hard. The troubles setting up the framework made it so agile did not work, thus during set up we worked together on fixing the bugs. After setup we started using agile again.

Product:
In the end we created a mandate register with the use of blockchain. The blockchain code has been unit tested. On top of the blockchain an application was built. This was end-to-end tested thoroughly.

Outlook:
The product is not finished. Features like authentication still need to be implemented and the overall performance needs to be increased.

Team Members

Robbert Koning:
Node.js, Application

Suleiman Kulane:
Networking, Automatisation, Smart Contract

Erwin van Thiel:
Application, Networking, Smart Contract, Evaluation

Jordy de Wit:
Automatisation, Testing, Smart Contract

All members contributed to writing the report and to the network configuration

Client & Coach

Client:
S. Hijgenaar
Senior Business Consultant, CGI Nederland

H. Heine
Director Consulting Services, CGI Nederland

Coach:
S. Roos
Assistant Professor Distributed Systems, TU Delft

Contact Information

Robbert Koning:
robbert.m.koning@gmail.com

Erwin van Thiel:
vanthiel.erwin@gmail.com

Suleiman Kulane:
suleimankulane@gmail.com

Jordy de Wit:
jordy1998@live.nl

CGI

The final report for this project can be found at: *http://repository.tudelft.nl*

TUDelft

B Usage Manual

In this section, an explanation is given on how to use the register. This is done by explaining the function of the project's script files, and providing an overview on how to start the register and GUI.

B.1. Scripts

In order to automate tasks needed for development, various scripts were created. An explanation of these scripts is given below. However, it is advised to study the scripts' contents before execution, as to prevent any unwanted side effects.

B.1.1. docker-swarm-cleanup.sh

The `docker-swarm-cleanup.sh` script removes the machine executing the script from the Docker swarm the machine is connected to, and removes Docker containers and networks.

Note that the script removes all Docker containers, networks, and volumes; also ones that do not necessarily have to do with Fabric.

B.1.2. generate-artifacts.sh

The `generate-artifacts.sh` script removes and re-generates files that are necessary for development on a single machine. It does not start any containers or servers. First, it removes and regenerates cryptographic material necessary for the MSP (Membership Service Providers) to function. Then it creates a genesis block, a channel configuration, and anchor peers. Lastly, it updates the private keys in `application/gateway/network-config.yaml` so that the Node.js server can send transactions, if need be. Starting the Node.js server is explained in Appendix B.2.4.

B.1.3. network_init.sh

In the CLI, the script makes all peers join the `dgochannel` and install the chaincode on them. Finally, anchor peers are updated and the `mandate_contract` chaincode is instantiated and initialized.

B.1.4. start.sh

The `start.sh` script removes and restarts all Docker containers necessary for development on a single machine. Next, it initializes a docker swarm and creates a network called `net_bep`. After that, it uses Docker Compose to start containers specified in `docker-compose-cli.yaml`, `docker-compose-ca.yaml` and `docker-compose-couch.yaml`. Finally, it executes `network_init.sh`. When the script has executed, there is a network online that is ready to handle chaincode invocations and queries.

Note that the script removes all Docker containers, networks, and volumes; also ones that do not necessarily have to do with Fabric.

B.1.5. addOrg.sh

The `addOrg.sh` script adds an additional organisation to a running Fabric network. First, it fetches the newest configuration block from the channel using a peer that is already in the channel. Second, the configuration of the new organisation is added to the configuration block and a new block is committed to the ledger. Afterwards, from within the newly created CLI dedicated to the peers of the new organisation, the peers are added to the channel and the chaincode is installed on them. Finally, a new anchor peer is added to the channel. The process is similar to joining the channel, the configuration block is updated to include the new anchor peer. This is then updated once again, so that the entire channel knows who and where the new anchor peer is; this allows for cross-organisation gossip with the new organisation. As parameters, `AddOrg.sh` takes:

1. `CHANNEL_NAME`: The name of the channel the organisation should be added to.
2. `ORG_NAME`: The name of the organisation that needs to be added to the network.
3. `CONTRACT`: The name of the chaincode that needs to be installed on the peers of the new organisation.
4. `VERSION`: The version of the chaincode that needs to be installed. This version will usually be one newer than the currently installed version on the channel. Specifying the version is necessary because peers from the new organisation likely will have to participate in the endorsement process. To do so, the endorsement policy of the chaincode needs to be updated to include peers from the new organisation and thus a new version is specified.
5. `PORT`: The port used for the first peer of the new organisation.

For example, adding organisation ‘dgo7’ to the `dgochannel`, with version 2.0 of the chaincode, can be done as follows:

```
./addOrg.sh dgochannel dgo7 mandate_contract 2.0 21051.
```

In order to add a new organisation to a running network, various configuration files in specific directories have to be created. The following directories have to be created:

- `\$[ORG_NAME]Artifacts`, e.g. `dgo7Artifacts`
- `docker-compose-files/dynamic-orgs/\$[ORG_NAME]`, e.g. `docker-compose-files/dynamic-orgs/dgo7`

The following files have to be created:

- `\$[ORG_NAME]Artifacts/configtx.yaml`, e.g. `dgo7Artifacts/configtx.yaml`. In this file the organisation’s configuration is specified.
- `\$[ORG_NAME]Artifacts/\$[ORG_NAME]-crypto.yaml`, e.g. `dgo7Artifacts/dgo7-crypto.yaml`. In this file the configuration needed to generate the crypto files for the new organisation are specified.
- `docker-compose-files/dynamic-orgs/\$[ORG_NAME]/docker-compose-ca-\$[ORG_NAME].yaml`, e.g. `docker-compose-files/dynamic-orgs/dgo7/docker-compose-ca-dgo7.yaml`. This is the docker compose file needed to create the ca container for the new organisation.
- `docker-compose-files/dynamic-orgs/\$ [ORG_NAME]/docker-compose-couch-\$[ORG_NAME].yaml`, e.g. `docker-compose-files/dynamic-orgs/dgo7/docker-compose-couch-dgo7.yaml`. This is the docker compose file needed to create the couchdb container for the new organisation.
- `docker-compose-files/dynamic-orgs/\$[ORG_NAME]/docker-compose-\$[ORG_NAME].yaml`, e.g. `docker-compose-files/dynamic-orgs/dgo7/docker-compose-dgo7.yaml`. This is the docker compose file needed to create the peer containers and a dedicated cli container for the new organisation.

B.2. Running the network

In this section, concrete steps are given on how to run the network on a single host and how to run it on multiple hosts.

B.2.1. Prerequisites

To run the network, the used machine must have the prerequisites, as defined in the Fabric Documentation [10], installed.

B.2.2. Single Host Machine Network

To launch a network on a machine locally, run the following command:

```
./start.sh
```

B.2.3. Multi Host Machine Network

Firstly, ensure that every machine which joins the network has the same generated cryptographic material in its file system. Secondly, kill and prune the containers that are still active with the following command:

```
./docker-swarm-cleanup.sh
```

To launch a multi-machine network, a Docker swarm and swarm network are needed. The following commands provide these, respectively:

```
docker swarm init
docker network create --driver overlay --attachable net_bep
docker swarm join-token manager
```

The last command returns a text with a manager token, for example:

```
docker swarm join --token SWMTKN-1-4
   f2q8ewzgsw7qy486f4kzqj1rzd1rwslyzpl93yqq46ij90q4o-0
   pcsziwvl9yat8fh5wd5wp6yk 145.94.198.150:2377
```

To join the swarm network on another machine, simply append the following to the command above:

```
--advertise-addr <ip address of joining machine>
```

Execute this

Next, run the containers for the desired organisation on the machine with the corresponding script. Running organisation dgo0 for example ought be done in the following fashion:

```
./dgo0.sh
```

Finally, run the following command for adding the organisations to the channel and installing and instantiating the chaincode on it.

```
./network_init.sh
```

B.2.4. Application

In order for the Fabric Node SDK to know what parts of the network to address, the connection profile should be adapted to the network setup. On a single machine network the URL of all peers and orderers must be configured to localhost. On a multi machine setup however, the URLs should contain the ip of the machine hosting the container. Moreover, the adminPrivateKeys must correspond to the ones in the crypto-config folder. The update-admin-keys.py script, which is also called in generate-artifacts.sh, takes care of this but it can also be done manually ofcourse. To get the application up and running, navigate to the application folder and run the following consecutive commands:

```
node register-user.js
node app.js
```

This registers the user and runs the server on `localhost:3000`, where the GUI will speak for itself. `localhost:5984` hosts the fauxton interface which displays the peer's world state.

B.2.5. Testing

To run the unit tests, the following steps need to be taken:

- Change the import path of Shim in `mandate_contract.go`. See Section 6.3 for the reason behind this.
- Run the `setupGoTest.sh` script located in `/chaincode`. This script will get all dependencies, like Testify Toolkit and Shimtest, needed to run the tests.
- Copy `mockStub.go` from `chaincode/testDependency` to the Shimtest dependency located at `GOPATH/src/github/hyperledger/fabric-chaincode-go/shimtest`. Again, see Section 6.3 for the reason behind this.
- Navigate to `/chaincode` and execute `go test`. This command will return either 'PASS' or 'FAIL' depending on the outcome of the test. Execute `go test -cover` to see the statement coverage besides 'PASS' and 'FAIL'. In order to generate an HTML file showing which statements and lines are covered, run `go test -cover -coverprofile cp.out`, then `go tool cover -html cp.out -o coverage.html`. Note that `cp.out` can be renamed to `<name>.out`, but must be the same in both commands. Also, `coverage.html` can be renamed at will.

B.2.6. Evaluation

For evaluating throughput and latency, simply run the following commands respectively.

```
node test_throughput.js
node test_latency.js
```

C Original Project Description

Een toestemmingenregister voor de Nederlandse energiemarkt Met de toenemende druk op energietransitie als gevolg van klimaatverandering, zien we dat het beheer van onze energienetten een steeds complexer proces wordt. Door grootschalige intrede van duurzame opwek, zoals uit zon en wind, en de elektrificatie van onze samenleving, denk aan mobiliteit en verwarming/verkoeling, neemt de dynamiek toe en de voorspelbaarheid af. Slimme serviceproviders kunnen hier alleen op inspelen met veel, accurate data. Echter, (nieuwe) privacywetgeving bemoeilijkt de toegang tot die data.

Koppelen van verschillende bronnen
De afgelopen decennia worden gekenmerkt door de digitalisering van onze energiesystemen. Op grote schaal wordt data verzameld en gebruikt voor facilitatie op macroniveau. Echter, die data wordt nog onvoldoende aan elkaar wordt gekoppeld en gebruikt voor het managen van het systeem op microniveau. Denk hierbij aan slimmere netten, accurate voorspellingen en het vermijden van netwerkproblemen. Vaak is privacygevoeligheid een show-stopper, omdat veel data in deze systemen aan een persoon gebonden zijn. Het gebruik van die data voor een specifiek doel zou dus op basis van toestemming van de eigenaar moeten gebeuren.

Toestemmingen in de energiemarkt
Toestemmingen (soms: attestaties of mandaten) voor datatoegang komen al overal voor in de energiemarkt. Wie een contract afsluit met een energieleverancier geeft, eigenlijk impliciet, toestemming om consumptiedata te ontsluiten en te gebruiken voor bijvoorbeeld factureren. Bij een verhuizing krijgt de netbeheerder toegang tot je slimme meter die vervolgens gedeeld wordt met een hele keten aan marktpartijen. Helaas zijn de meeste van deze processen niet inzichtelijk voor alle betrokken partijen en berusten ze vaak op veel handmatig werk, wat ze foutgevoelig en tijdrovend maakt. Dat kan slimmer.

Een slim toestemmingenregister
Het toekomstbeeld is een register waarin consumenten en prosumenten op een overzichtelijke manier slimme afspraken met marktpartijen kunnen maken. Blockchaintechnologie kan hier een rol in spelen door het onveranderlijk bijhouden van alle autorisaties die men uitdeelt op hun persoonlijke data. Zo kan met een toestemming voor het koppelen van jouw BRP-registratie en je slimme meter een verhuizing voortaan automatisch verlopen en in gang gezet worden door een simpel belletje met de gemeente.

De opdracht
Bouw een decentraal toestemmingenregister waarin het mogelijk is om op basis van identificatie en authenticatie met DigiD verschillende datatoegangsrechten te verlenen. De focus ligt op het koppelen van verschillende (bestaande) databronnen, die in verschillende combinaties andere use cases mogelijk maken:

- Switchen van energieleverancier;
- Diensten afnemen van aggregators of MSPs

- Verhuizen;
- Energie verkopen aan je buurman;
- Gebruik van je data door kennisinstellingen;
- Etc.

Essentieel is het bedenken van een veilige manier voor het opslaan van de authenticatie en het slim uitwisselen van de toestemmingen tussen een veelheid van partijen met behulp van blockchaintechnologie.

D Research

D.1. Introduction

In this chapter, research that was conducted during the project is reported.

First, an analysis of the problem is presented in Section D.2 by evaluating how the current situation causes problems, after which a proposal for a solution is given. Then, the design goals of the project and its success criteria are determined in Section D.3. After this, various critical decisions that were taken with respect to system design are elaborated upon in Section D.4. This elaboration consists of design choices concerning the manner in which data is stored and design choices specific to the preferred technology used to solve the problems at hand. Finally, the frameworks available for implementing this technology are discussed in Section D.5.

D.2. Problem Analysis

The first step of the research process is an analysis of the problem. The current situation in the energy market is discussed in Section D.2.1, issues to be resolved are then stated in Section D.2.2 and finally a possible solution to these issues is proposed in Section D.2.3.

D.2.1. Current Situation

Currently, many processes in the energy market are convoluted and tedious. One such example is the process of consumers switching their energy supplier. Whenever a consumer wants to switch from their supplier, they go to a new supplier's website or a switching service and issue a request to switch to the new supplier. When the new supplier receives this request, it checks the Contract Einde Register (CER) to verify the consumer is allowed to switch according to their current contract and their cancellation period. If the consumer is allowed to switch, the new supplier contacts the Distribution Grid Operator (DGO) with the switch request. After the DGO receives the request they will undergo the following steps [16]:

1. The DGO processes the request and does seven checks to see whether the request is valid or not.
2. In case one of the seven checks fails, a rejection is sent to the new supplier and the new supplier processes that rejection.
3. The new supplier receives and processes a 'GAIN' message.
4. The old supplier receives and processes a 'LOSS' message. It also checks the meter value.
5. The old Program Responsible Party (PRP) - the party responsible for the purchase of electricity - receives and processes a 'LOSS' message and the new PRP receives and processes a 'GAIN' message.
6. The DGO plans the mutation into the 'Aansluitingsregister'.

Besides the actual switch request, the measurement data of the consumer also need to be communicated between these parties, but the consumer is not involved in this process. This data is collected via the uniquely numbered EAN code of the consumer's smart meter. The collected data is used to determine at what usage value a certain contract ends and a new one is started. For the new supplier to be allowed to collect the data, they need permission from the client to read their smart meter.

D.2.2. Problem Definition

Looking at the current situation, a few main problems become apparent:

1. Lack of transparency: in the current system, consumers have no clear overview of the parties that are or have been able to see and use their data, because currently no record of the permissions is being kept.
2. Time consuming process: the process of switching suppliers can be time consuming for consumers. It can take up to six weeks for the switch to go into effect[21]. When switching, there is much communication between parties, for instance for verifying each others' data. Verification can be even more time consuming when there is conflict between parties. The new supplier also has to wait on the permission to read the client's smart meter data.

D.2.3. Proposed Solution

To solve the problems stated in Section D.2.2, a register could be built that stores the consumer's permission (or mandate) for the data to be used. Such a register would allow consumers to digitally specify which energy suppliers are allowed to use their data. This means that consumers become able to change which energy suppliers are able to use their data with a simple update in the register.

A register accessible to both consumers and suppliers could overcome data insight issues from the perspective of consumers while keeping consumer energy data usable from the perspective of suppliers. This register can be used to implement an interface in which consumers can see a proper overview of the parties that are or were allowed to use their data, and when.

Moreover, the register could speed up the process of switching from suppliers, because it could perform the necessary checks itself so that suppliers do not have to trust each other.

Lastly, many other use cases become possible with an efficiently functioning mandate register. This is because consumers can make specific data available for use by any type of party, such as an academic institution.

Though the main focus of this project is switching to another energy supplier, a large emphasis is placed on keeping the door open for other use cases.

D.3. Design Goals

In this section, the most design goals of the project are elaborated upon. These goals are considered when making influential decisions. The essential design goals for this project are expandability, security and privacy, and GDPR Compliance. The section ends with criteria that should be met in order to consider the project a success.

D.3.1. Expandability

Expandability is crucial to the success of the project. All major design choices were made with the idea that the system should support many different use cases and many users.

Support for various Use Cases

It is critical that the system can be extended to support many different use cases. While the project's specific use case is allowing consumers to switch between energy providers, the challenge is to facilitate as many other use cases for mandates as possible. In order to facilitate other use cases, mandates should be as generic as possible, formatted intuitively as: 'Party X grants/revokes permission for party Y to access X's data concerning Z, for the purpose of W.' Z in this project would be energy and W could be 'a contract', 'science', etc. Therefore, instead of implementing a single use case, a system is implemented, which is then proven to work by implementing a use case on top of it.

Scalability

Another important aspect of expandability is that the system should be able to handle thousands of consumers simultaneously granting and revoking mandates and hundreds of parties simultaneously querying this data.

D.3.2. Security and Privacy

The second design goal is 'security and privacy'. This means that data read from the system is accurate, that data written to the system is verified, that parties with access to the system are authenticated, that the system does not have a central authority, and that the system handles sensitive data with care. These statements are elaborated upon below.

Verified Data Reading

Data queried from the system must be accurate and untampered with. Parties must be assured that they can retrieve all mandates that belong to them from the system, and that these mandates are untampered with and complete. Therefore, all mandates that are accepted by the system, should be immutable once stored.

Verified Data Writing

All data that is written to the system should be verified to adhere to the protocol, before it is written to the system.

Authentication

In order to guarantee that mandates cannot be granted, revoked or read by unauthorized parties, all parties with access to the system should have to authenticate themselves before being able to read or write mandates.

No Central Authority

In order to prevent improperly functioning parties from modifying consumers' mandates, there should not be a central, corruptible authority for adding data to the system.

GDPR Compliance

The system is credible if it handles consumers' data with care. Therefore, it should follow the guidelines laid out by the General Data Protection Regulation (GDPR)[7]. The concept of a mandate register is in line with the GDPR's core principles, for it gives consumers more control over their own data.

However, mandates themselves are personal data because they can be traced back to individuals. Storing these mandates raises GDPR concerns. These concerns should be taken care of. However, it is important to realise that solving all problems might not be possible within the scope and duration of the project.

D.3.3. Success Criteria

Based on the proposal of Section D.2.3 and the design goals of this section, it is possible to state the criteria that must be met in order to consider the project a success. These success criteria form a useful guide during the implementation phase of the project and will give a rough indication of the project's progress. The criteria are rather intuitive:

1. The system must be more transparent and less time consuming than in the current situation.
2. The design goals stated in this section must be met.

D.4. Design Choices

In this section, the design choice that was made with respect to the type of data storage is elaborated upon, together with design choices specifically for that type of storage.

D.4.1. Data Storage

For the storage of mandates, two main options are considered. One of the options is blockchain. The other option is a distributed database. Both systems can have similar properties such as decentralisation and immutability. These properties are required when considering expandabilty and credibility.

Blockchain in general was found to have a larger collection of available documentation and tutorials. Also, nodes in a blockchain network can all be mutually mistrusting entities that do not want to have to rely on a trusted third party for exchanging data [22]. Taking into account the aforementioned properties of blockchain, it is apparent that blockchain does not have any considerable disadvantages compared to distributed databases. After discussion with the client, it became apparent that they have a preference for using blockchain. Therefore, after thorough research and discussion, it was agreed upon to use blockchain for the project.

D.4.2. Blockchain

Having decided on using blockchain technology, this section will contain an elaboration of design choices that follow accordingly. First, a distinction will be made between a permissioned and permission-less blockchain. Finally, the concept of a consensus protocol is introduced and several different consensus protocols are considered, along with their applicability for this project.

Permissioned vs. Permission-less

A permission-less blockchain is a blockchain that everybody is free to join at any time. There is no central authority that can decide to ban or disallow any peer. This means that everybody can host a node in the network [22].

A permissioned blockchain, however, does work with an authority that decides who can or cannot be part of the network and which participant gets which rights. An example could be the right to validate blocks [22]. Note that the central authority in a permissioned blockchain cannot hinder data transactions between nodes.

In this project, there is need for a third party to decide who is allowed to host a server, because not everybody should be able to do so. Consequently, a permissioned blockchain with a central authority determining its participants will be the option best suited for this project. The blockchain will also be private rather than public because not everybody should be able to access the stored data.

Consensus

In this section, the concept of a consensus protocol is brought to light and different known consensus protocols will be discussed. First, the desired characteristics of a consensus protocol suitable for this project will be determined. Consensus protocols help a distributed or decentralized network to unanimously take a decision whenever needed[19].

Desired Characteristics

When choosing a protocol that is going to determine in what fashion the blockchain's ledgers are expanded, several characteristics have to be taken into consideration. To elaborate, these specific characteristics are speed, fault tolerance and scalability.

- The network should be fast enough to let people switch providers or query their mandate information within a matter of seconds.
- The network should be relatively robust to scaling, meaning that it should maintain reasonable performance with respect to the number of network participants as this number grows. Scalability is only a factor up to a certain degree because the network is not expected to become very large nor grow multiple orders of magnitude.
- Fault tolerance is also an important attribute because network nodes can always fail. When this happens, the network should maintain a consistent and correct state when it comes to stored data.
- In the field in which the network will be deployed, no concern is needed for malicious nodes. Therefore, resistance against maliciousness is not taken into consideration.

	PoW	PoS	PoET	BFT and variants
Blockchain type	permissionless	Both	Both	permissioned
Transaction rate	low	high	medium	high
Scalability of peer network	high	high	high	low
Adversary tolerance	<=25%	Depends on specific algorithm	Unknown	<=33%

Table D.1: consensus comparative analysis[3]

Proof of Work
In a Proof of Work protocol (PoW), as used in for example Bitcoin, each node in the network competes for adding a new block by solving a complex mathematical puzzle. When a node finds the desired, easily verifiable solution, it broadcasts the block to all other nodes. The other nodes must validate the block and if it is correct, they all append it to their own copy of the chain. The essence of this algorithm is that a node has to do a tremendous amount of work before being able to append a new block, to prove that a node is not likely to be malicious [23]. All this work costs a lot of energy and resources, therefore this approach does not seem appropriate for an application in the energy market. Besides being very energy consuming, a PoW protocol also has a poor transaction rate[3], and speed is a factor that should be taken into account.

Proof of Stake
Unlike Proof of Work, Proof of Stake does not need an enormous amount of energy to append a new block. The node which appends a block to the chain will be pseudo-randomly chosen out of the stakeholders [18]. A 'stakeholder' refers to a party with a monetary incentive in the network. To be clear, the use case described earlier does not concern monetary transactions; defining a 'stake' is not appropriate for this use case. Therefore, Proof of Stake is unsuitable for application in the energy sector.

Practical Byzantine Fault Tolerance
Practical Byzantine Fault Tolerance (pBFT) is an algorithm in which all of the parties are known, but they may be faulty or malicious. The network will reach consensus by means of voting [5]. Voting can be done fast, but there will be overhead in order to reach consensus after voting. The overhead is not a problem within a small network, but once the network becomes larger, it will become a bottleneck [3]. This type of consensus algorithm works well in a permissioned system.

Proof of Elapsed Time
Proof of Elapsed Time is a consensus algorithm used mainly by the Hyperledger Sawtooth framework. In this section, only the algorithm is explained; the framework is elaborated upon in Section D.5.

In short, PoET elects a random leader to append a block to the chain. All nodes in the network are assigned a random waiting time, and the first node to wake up finishes the block and broadcasts it to the other nodes. To avoid manipulation by malicious nodes, all nodes need to run in a Trusted Execution Environment (TEE), for example Intel's Software Guard Extensions. This TEE guarantees that the random waiting timer cannot be tampered with by malicious software [3]. PoET overcomes the drawbacks of PoW and PoS but requires the nodes to run on specific hardware.

Raft
Raft is a consensus algorithm that is used by for instance the Hyperledger Fabric framework. In this section, only the algorithm is explained; the framework is elaborated upon in Section D.5.

The Raft consensus protocol is built up to solve three problems, namely leader election, log replication and safety.

Leader election is the process of network participants choosing a 'leader' which determines what data to add to the chain. A leader uses an 'AppendEntry Remote Procedure Call' (RPC) for this. All requests to add data must go through the leader. In the case of Raft, all nodes start a self-specified pseudo-random timeout, and the node whose timer ends first becomes a candidate node. This candidate starts an election and asks all other nodes to vote for him, by means of a 'RequestVote Remote Procedure Call'. Nodes will vote for the first candidate which contacts them, except if they are a candidate themselves. If a majority of the nodes vote for a candidate, it becomes the leader. After an arbitrary, self-specified amount of time or when the leader fails to send a heartbeat, a new election is started.

Log replication is the process of ensuring that all network participants share the leader's log. In Raft, the leader sends the new entries from its own log to all its followers. If a majority approves this entry and adds it to their logs, the entry is committed. Raft ensures that all previously committed entries are also agreed upon in the network.

The last problem which Raft attempts to tackle is safety. For instance, Raft imposes restrictions on becoming a leader, such as demanding that all leaders have an up-to-date log. However, these safety measures do not guarantee that malicious leaders cannot take over the network. For this project it is not a problem, because malicious nodes are not taken into account.

The Raft protocol seems well-suited for a permissioned blockchain in which all parties are treated equally. Every node has an equal probability of becoming leader. This is also the case for the PoET protocol, but Raft does not depend on specific hardware. The protocol is also robust to failing nodes due to the heartbeat mechanism; it can tolerate up to 49% of nodes failing - which is called Crash Fault Tolerance - and is much faster than PoW or PoS protocols. A drawback of Raft is however that the network is limited in scalability due to its architecture and cannot handle malicious nodes.

The information in the Raft section was based on [17] and [15].

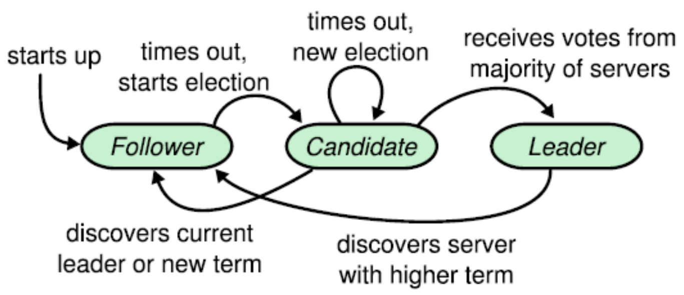
starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Figure D.1: The states of the nodes can be represented by this state machine [17]

Conclusion

After evaluating the above consensus protocols, Raft seems to be most appropriate for this project. The two drawbacks, being lack of scalability and incapability of handling malicious nodes are lesser problems than hardware dependencies, lack of speed and fault tolerance. This is because in this project the possibility of nodes being malicious is not taken into account and the number of network nodes will not be very large.

D.5. Blockchain Frameworks

For the development of the blockchain network, multiple frameworks were considered. The requirements and design choices discussed in earlier sections will be used to evaluate the applicability of the frameworks. Other important factors to consider are how well the frameworks are documented and how well they are maintained.

Hyperledger Fabric

Hyperledger Fabric is an open-source platform designed for the development of permissioned blockchain networks. It has substantial documentation which contains theoretical explanation and code tutorials. Fabric implements a consensus protocol based on Raft [2], which means it is resistant against a considerable percentage of node failure and can achieve high transaction rates. Fabric also supports querying data through CouchDB instead of querying the blockchain directly. The information in this paragraph was gathered from [10].

- + Widely used, well-documented and maintained;
- + Raft-based consensus protocol has most of the desired characteristics: it is fast, relatively scalable and Crash Fault Tolerant.

- + Querying capabilities powered by CouchDB.

Hyperledger Sawtooth

Hyperledger Sawtooth is an open-source platform designed for the development of blockchain networks [13]. Sawtooth provides an elaborate documentation and is also widely used. It is also built in such a way that most application logic can be written in a variety of common programming languages, such as Python, JavaScript, Go, C++ and Java. Finally, Sawtooth allows plugging in different consensus protocols, but by default it only offers a form of Practical Byzantine Fault Tolerance or Proof of Elapsed Time. This is Sawtooth's main drawback with regards to the project; neither consensus algorithm is suitable, and implementing another one is not feasible considering the project's limited duration.

- + Widely used, well-documented and maintained;
- + Application logic can be written in a wide variety of languages;
- − Consensus algorithms are not suitable due to specific hardware dependencies or scalability limitations.

Tendermint

Tendermint is an open-source platform for blockchain application development. Although Tendermint provides various well-documented examples, general documentation is limited. A benefit of Tendermint is that it satisfies the Byzantine Fault Tolerance (BFT) property. This provides various safety guarantees. For example, up to $1/3$ of the nodes in the network can be explicitly malicious without the network being compromised [20]. However, BFT is achieved through Tendermint Core - Tendermint's consensus algorithm - which cannot scale enough to suit this project. The information in this paragraph was gathered from [20].

- + Satisfies Byzantine Fault Tolerance;
- +/− Development documentation is available with examples but general documentation is limited;
- − Consensus protocol is not suitable due to scalability limitations.

BigchainDB

BigchainDB is an open-source platform that offers tools for building both public and private networks with the characteristics of a (MongoDB) database and a blockchain together. It is properly documented. It provides decentralisation, Byzantine Fault Tolerance and immutability like a blockchain, but it also provides high transaction rates and querying functionality like a database [4]. However, BigchainDB achieves BFT by using a Tendermint consensus algorithm that cannot scale enough [4].

- + Widely used, well-documented and maintained;
- + Querying capabilities powered by MongoDB;
- − Consensus protocol is not suitable due to scalability limitations.

Conclusion

After considering the frameworks above, it appears that Hyperledger Fabric is the most suitable framework for this project.

Although Hyperledger Sawtooth seems to be a valid option as well, its built-in consensus algorithms are not suitable for the project. Implementing another consensus algorithm might be a possibility but seems unreasonable given the duration and other objectives of the project.

Bibliography

[1] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. *CoRR*, abs/1801.10228, 2018. URL `http://arxiv.org/abs/1801.10228`.

[2] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[3] Arati Baliga. Understanding blockchain consensus models. In *Persistent*. 2017.

[4] BigchainDB. Bigchaindb whitepaper. URL `https://www.bigchaindb.com/whitepaper/`.

[5] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[6] Docker. Docker documentation. URL `https://docs.docker.com/`.

[7] EU. Gdpr. URL `https://gdpr-info.eu/`.

[8] Hyperledger. Hyperledger fabric msp, . URL `https://hyperledger-fabric.readthedocs.io/en/release-1.4/msp.html`.

[9] Hyperledger. Hyperledger fabric documentation ; securing communication with transport layer security, . URL `https://hyperledger-fabric.readthedocs.io/en/release-1.4/enable_tls.html`.

[10] Hyperledger. Hyperledger fabric documentation, . URL `https://hyperledger-fabric.readthedocs.io/en/release-1.4/`.

[11] Hyperledger. Hyperledger fabric gossip protocol, . URL `https://hyperledger-fabric.readthedocs.io/en/release-1.4/gossip.html`.

[12] Hyperledger. Hyperledger blockchain performance metrics, . URL `https://www.hyperledger.org/wp-content/uploads/2018/10/HL_Whitepaper_Metrics_PDF_V1.01.pdf`.

[13] Hyperledger. Hyperledger sawtooth documentation, . URL `https://sawtooth.hyperledger.org/docs/`.

[14] Hyperledger. Fabric samples. `https://github.com/hyperledger/fabric-samples`, 2019.

[15] Du Mingxiao, Ma Xiaofeng, Zhang Zhe, Wang Xiangwei, and Chen Qijun. A review on consensus algorithm of blockchain. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2567–2572. IEEE, 2017.

[16] Vereniging Nederlandse Energie-Data Uitwisseling (NEDU). Detailprocesmodellen mutatiemeetprocessen kleinverbruik. Technical report, mar 2018.

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

[18] Fahad Saleh. Blockchain without waste: Proof-of-stake. *Available at SSRN 3183935*, 2019.

[19] Lakshmi Siva Sankar, M Sindhu, and M Sethumadhavan. Survey of consensus protocols on blockchain applications. In *2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS)*, pages 1–5. IEEE, 2017.

[20] Tendermint. Tendermint documentation. URL `https://docs.tendermint.com/master/`.

[21] Peter van der Wilt. Overstappen naar een andere energieleverancier. URL `https://www.consumentenbond.nl/energie-vergelijken/overstappen-energieleverancier`.

[22] Karl Wüst and Arthur Gervais. Do you need a blockchain? In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 45–54. IEEE, 2018.

[23] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 557–564. IEEE, 2017.