UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Sven Mitt

Blockchain Application - Case Study on Hyperledger Fabric

Master's Thesis (30 ECTS)

Supervisor(s): Luciano García-Bañuelos
Fredrik Milani

Tartu 2018

Blockchain Application - Case Study on Hyperledger Fabric

Abstract:

To enable software platform to be used without a third trusted party one of the possibilities is to use blockchain and smart contracts. One of the latest platform is open-source Hyperledger Fabric, a modular system that uses conventional programming languages for smart contracts. This opens up vast possibilities for using it product centric enterprise systems. In this paper we compare the platform to a conventional solution and study the challenges provided by the smart contract called chaincode. We implement a parking spot application for multisided market using smart contract and Go programming language. In the end we have a working prototype with solutions to technical problems, covering predetermined use cases.

Keywords:

Distributed Ledger Technology, Smart contract, Blockchain Technology, Hyperledger Fabric, chaincode, timestamps in distributed system, blockchain GIS support

CERCS: P170

Plokiahela rakendus – Hyperledger Fabric uuring

Lühikokkuvõte:

Usalduse keskkonna saamiseks kasutatakse kolmandaid osapooli ja nende tarkvara platvorme. Plokiahela tehnoloogia ja nutikaid lepingud on üks võimalus kuidas välistada kolmas osapool. Üks viimased turule tulnud vabatarkvara platvorme on Hyperledger Fabric, modulaarne süsteem, mis kasutab üldkasutavaid programmeerimskeeli nuitikate lepingute keelena. See avardab platvormi kasutamist ettevõtte tarkvara loomisel. Võrdleme platvormi tavapäraste lahendustega ning uurime väljakutseid, mida pakub uus plokiahela põhine süsteem ja selle jaoks loodud nutika leping nimega chaincode. Selle töö käigus realiseeriti parkimiseks mõeldud rakendus, mille nutikas leping on kirjutatud Go programmeerimiskeeles. Töö käigus realiseerisime prototüübi, leidsime lahendused tehnilistele probleemidele, realiseerisime kasutusjuhud.

Võtmesõnad:

Distributed Ledger Technology, Smart contract, Blockchain Technology, Hyperledger Fabric, chaincode, timestamps in distributed system, blockchain GIS support

CERCS: P170

Table of Contents

1 Introduction

1.1 Motivation

When creating an application that involves two or more parties exchanging monetary values, a third trusted party is used. The third trusted party will make sure that all transaction are rejected or accepted and finalized, leaving the system in consistent state. To enable platform to be used without a third trusted party, one of the possibilities is to use blockchain and smart contracts.

There are many blockchain enabled platforms that allow using smart contracts, Bitcoin, Ethereum, Hyperledger Fabric etc., but they are not equal.

Bitcoin smart contract use stack based language, which is not Turing complete, and supports only transactions for monetary value Bitcoin.

Ethereum has its own Turing complete language called Solidity. It is a new language designed especially for using in Ethereum and it is Turing complete [1]. It has support for loading code from another address and this enables creation of libraries, pieces of reusable software code.

These technologies have limited uses in Enterprise software, the first one is too limited and the other one has not had time to be mature and requires porting libraries from other systems to be used with Solidity.

This is where Hyperledger Fabric and its smart contract called chaincode shines. It is built from known and already proven Enterprise software pieces. Chaincode supports usage of Java, Go or Node.js as programming language for developing smart contracts. That also allows to use a vast amount already existing libraries created during the lifetime of the programming language. It also support complex queries that we are used to having in SQL and NoSQL databases. This is first blockchain based system that supports using conventional programming languages for building smart contracts [2] and only one enable complex data queries. This mechanism gives the edge to Hyperledger Fabric and opens up possibilities of using the smart contracts in systems like enterprise systems.

To compare the platform with a traditional soultion, we create a parking spot application prototype presenting an enterprise software and find solutions to problems that the new architecture calls for, evaluate if it is suitability for this software, and compare Hyperledger Fabric to conventional solution.

1.2 Scope

Scope of this thesis is to research blockchain suitability for parking spot type application and implement it using predetermined use cases. A network of nodes running Hyperledger Fabric is created and parking spot application is deployed into the network as a smart contract. This setup is used to study the new technology and compare it to conventional solutions.

1.3 Research problem

We define research questions accordingly:

1. What are the key similarities and differences between Hyperledger Fabric service paradigm and traditional database paradigm?
2. What are the benefits and disadvantages of Hyperledger Fabric compared to traditional database systems?

1.4 Summary of contribution

By the end of this thesis, we have implemented selected use cases for parking application and shown that it is possible to provide a viable system using Hyperledger Fabric as a blockchain platform. We have provided alternative solutions for nondeterministic functions in smart contract, time range queries and implemented GIS support. Built upon the experience of using the platform for creating parking application, we do a comparison between conventional systems and Hyperledger Fabric.

The thesis is structured as follows. Section 2 presents background about blockchain, smart contract and Hyperledger Fabric Section 3 describes the domain problem statement and implementation of the parking spot software. Section 4 contains the description comparison framework, details about conventional and Hyperledger Fabric platforms using the comparison framework Section 5 presents the evaluation of results Section 6 summarizes the paper and outlines future work to be done

2 Background

Distributed ledger (also called distributed ledger technology – DLT) is an digital ledger that is shared across multiple locations and all participating parties can verify the authenticity of the data. There is no central data store or 3rd trusted party. Ledgers use mathematical consensus algorithms to make sure that all or most participants have the same data. One of the technologies that enables this is called blockchain. In the context of this thesis, blockchain and distributed ledger, is used interchangeably.

2.1 Blockchain

The first business endeavor that used a blockchain-based ledger was presented in the white-paper published in 2008 by Satoshi Nakamoto describing a system for electronic cash [3]. This was the start of Bitcoin.

Blockchain is a decentralized ledger that is shared by all network participants. Because of its nature, modifying an existing ledger is not possible, because it is mathematically impossible. This is achieved by using cryptographic algorithms.

Blockchain data structure is list of data blocks that are **timestamped**, **immutable** and in **strict order**. Immutability is implemented by using a hash, a digital fingerprint of data. Every block has reference to previous blocks hash and this gives a strict order to blockchain. Following hashes from current block ends with block 0 – called genesis block. It is the first created block on specific blockchain. Blocks contain list of transactions. This type of data structure enables **provenance**, there is single place of origin for any transaction.

Blockchain are divided as **public** and **private** ledgers [4]. Public ledger are accessible and by anybody and everybody can add blocks to them. Private ledgers only include a selected group of people. To make a blockchain viable, a single chain is needed and this is achieved by **consensus** – all or most participants agree what is the next block in the chain and thus what is state of the whole blockchain. Public ledger consensus algorithms are Proof-of-Work and Proof-of-Stake.

Proof-of-Work make a use of computer CPU raw power. Whoever solves the mathematically difficult calculation for creating new block, will add the new block to chain and publish it to all other members. Algorithm is built so it is easy to verify that calculations are correct and others accept the new block. This consensus algorithm means that the longest chain will contain the highest amount of work. Proof-of-Stake tries to mitigate the high price of Proof-of-Work algorithm. The algorithm decides who will be the next one who creates a new block and must contain some randomness.

Private ledgers are **permissioned** and have much simpler consensus algorithms since the participants number is very limited and participants authenticated and authorized.

The General Data Protection Regulation (GDPR) will become enforceable in Europe by 25 May 2018. This has paradoxical effect on blockchain. It enables the right to be forgotten, but blockchain is immutable by its nature and data cannot be deleted.

Definitions used in blockchain context:

- **Asset** – any type of capital or durable goods, anything with monetary value.
- **Transaction** – move value from inputs to output. Input can be previous transaction output or in case of assignment of new Asset, a new issuance of unit.

- **Blocks** –multiple transactions are batched into blocks and those make up the blockchain. New block is written when a consensus program is executed successfully.
- **Consensus** – agreement of majority of participants in network.

2.2 Smart contract

A smart contract [5] adds functionality to blockchain as it is a computer program that can execute transactions, access blockchain blocks and the history. This functionality was added to second generation of blockchain.

It is a computer program that is stored in distributed database. Smart contract allows the addition of constraints, validation and business logic to transactions. They could be considered similar to database triggers. Quite a powerful tool for interacting with the blockchain, particularly if the smart contract implementation is Turing complete.

This program is executed on all nodes that are participating in the transactions. That means the contract itself has to be committed to the blockchain and all parties must accept it. This is also true if there is a need to change the currently existing contract. If the majority will not find consensus, then the contract cannot be updated.

Smart contract is not just a computer program, it is an agreement between parties. It contains parts for entering the contract, executing operations and exiting. Example using parking spot renting contract. Entering into a contract could mean that the parking spot provider signs the contract, stating the parking spot is available at given time and price. Interested renter signs the contract and enters to the agreement with the renter, accepting the time and price. During the execution of an agreement, the renter could exit the contract by payment of final amount or by canceling it. Cancelation could mean that renter has to pay a predetermined fee. These operations are executed on the smart contract and the transactions are written to the blockchain. Auditing the contract is simple by nature of blockchain.

There are pitfalls. Since smart contracts move digital values that also have real monetized value in the real world, a bug in a contract will be costly. Writing even a simple smart contract needs economic thinking and can lead to unexpected results [6].

Operations in a smart contract must be deterministic. It is being executed individually on each blockchain node and the result must be same. If the majority of executions fail, the transaction will fail.

Conventional contracts require that certified lawyer or lawyers by both parties come to an agreement of the contents. Reading a smart contract requires the knowledge of the programming language, that contract is written in. This provides more complexity.

2.3 Hyperledger fabric

Founded by Linux Foundation in 2015, Hyperledger project builds upon collaborative approach to develop blockchain technologies. Hyperledger Fabric is one of the projects, it is main purpose is to have modular, extensive architecture. This enables consensus and membership services to be plug-and-play. It is open source software licensed under Apache License, Version 2.0 [7].

Hyperledger Fabric is a platform for building you own blockchain applications. It differs from other known blockchain systems as it is private and permissioned. All participants must enrolled through a trusted Membership Service Provider (MSP), before they can be

part of the network. No transaction is allowed without verifying the participant. Since all participants are known, there is no need for proof-of-work or other protocols that are used in Bitcoin or Ethereum. A participant in permissioned network can be allowed invoke smart contract, but not allowed to deploy a new one. To enable private, confidential transactions, a separate channel can be created. Data is only visible to the participants of the channel.

Distributed ledger is shared between all nodes. It is composed of two data structures: transaction log and world state. The transaction log is not replaceable and is built in. After accepting new block with transactions, new world state is agreed and written. World state describes the end state of sequential transactions.

2.3.1 Nodes

The system contains three types of nodes: peers, client and orderers. The client is the node that represents the end-user. It connects to peers and orderers for updating the data. SDK is provided in Java, JavaScript (Node.js) and Go.

Peer manages digital ledger data, transactions and runs smart contract called chaincode. The ledger consists of two components:

- transaction log
- world state

The transactions change the world state by using chaincode. Also deploying a new chaincode is considered a transaction. The chaincode will be signed and system creates an immutable package of the chaincode. A separate Docker image is created with version tag and it is running as a separate machine. This will ensure that if something happens within the chaincode, the peer will not crash with it. Peers will run the chaincode on a channel, a separate ledger, and one peer can run multiple channels.

Hyperledger Fabric depends on certificates, the same ones used by HTTPS protocol. Every move is signed by certificates, so there are no users in perspective of system. These certificates can be generated in advance, but that would be too static for enterprise applications, so a separate service called Fabric CA is provided to dynamically generate certificates for users. Persistence is provided by MySQL, PostgreSQL or LDAP server.

Orderer is a consensus service that purpose is to quarantine that all transactions in the same order for all participants and it sends them as a block to all peers, which will persist the block to the ledger. There are multiple implementations supported, SOLO a single instance for developing and Apache Kafka, a well-known distributed streaming platform. Apache ZooKeeper, well-known coordination service, is used by Kafka for providing group services, distributed synchronizing and maintaining configuration information. The work of adding Simplified Byzantine Fault Tolerance is on the way. This is one of the strengths of the system, it is built modular and is possible can change the consensus service as needed.

Each peer has a world state database kept in a key-value store LevelDB or document-oriented database called Apache CouchDB. Latter enables chaincode to execute complex queries on blockchain data.

2.3.2 Chaincode

A smart contract in Hyperledger Fabric is called chaincode. It is a program, written in Go, Node.js or Java language. This enables the usage of vast libraries that have been developed during the lifetime of programming language.,

Chaincode runs in an isolated Docker container. This gives the ability to use existing API and make the migration easier. Chaincode purpose is to be the business layer in software development.

Deployment of new chaincode is two-step process. In the first step the code is deployed to all peers file system. Second step is called instantiate for new code and upgrade for upgrading existing chaincode. The second step is for actually deploying the code into production. Deployment of chaincode goes through the same process as transactions and requires that all peers sign the new chaincode. When deploying new chaincode, it is possible to assign policy, of which peers must sign transactions running this chaincode. Chaincode can be deployed via CLI or by using SDK.

To implement permissioned ledgers, platform offers channels. Chaincode is running on the channel – a separate ledger. Same Chaincode can run on different channels, similar how same server software can run on different client environments. It is possible to invoke other chaincode and even chaincode in another channel. Channels can be used to keep private data leaking to other members. When channel is set up between subset of members, the blockchain data is physically available only to the participants nodes. Hyperledger Fabric latest adds a possibility to encrypt part of data using built in functions. Use this built in functionality enables even querying the encrypted data. The safekeeping of encryption key is trusted to the client.

2.3.3 Processing of transactions

Transaction management is split between peers and orderers. This allows higher parallelism and concurrency for the network. Every transaction is executed in the peer using world state. If the transaction succeeds, it is signed with Peers certificate. Executing transactions prior ordering allows each node to process multiple transaction at the same time. The orderer will not re execute the transaction, just order them and do not maintain ledger. This also enables the peers to trust all orderers and vice versa, so they can run independently. Peers are divided into endorsing peers (peers that contain specific chaincode and are part of the policy) and peers without the chaincode. Peers without the chaincode can still validate and commit the transaction to their ledger after receiving it from the endorsing peer.

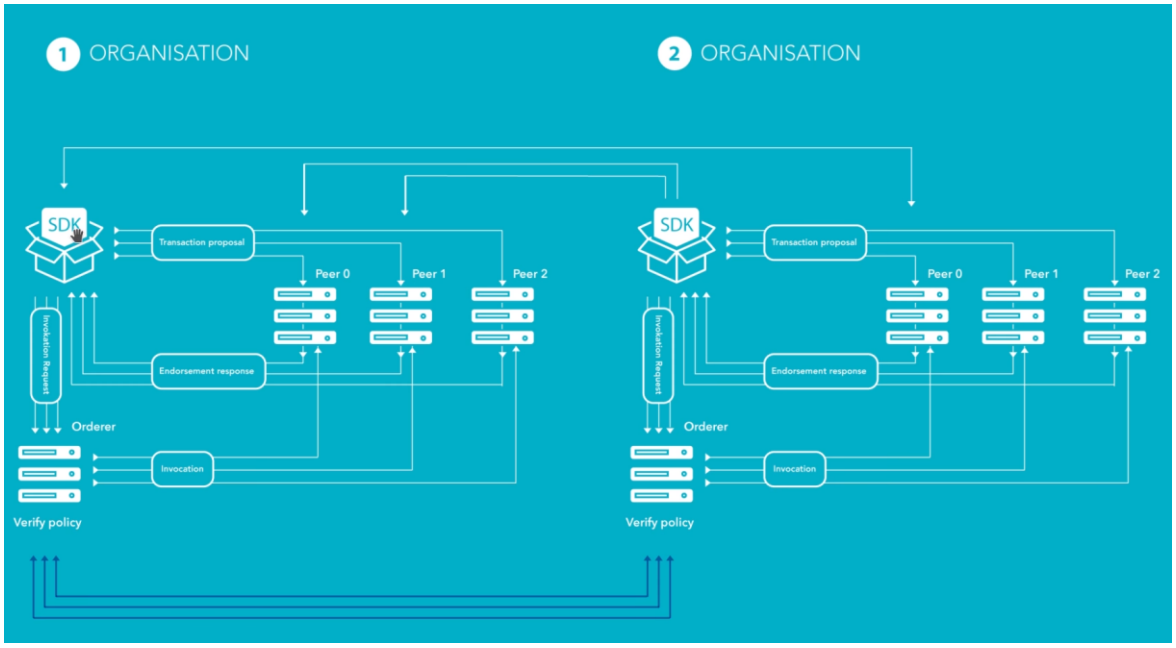

Figure 1: Hyperledger Fabric Network example [8]

In Figure 1: Hyperledger Fabric Network example Figure 1**Error! Reference source not found.** is a Hyperledger Fabric setup for two organizations, both using 3 peer groups and one orderer group.

1. **Transaction proposal** - client creates a transaction proposal using SDK, it sends it to all peers that need to sign it including the ones in another organization. This is set by endorsement policy when deploying smart contract called chaincode. Examples of policies:
    a. AND ('Organization1.member', 'Organization2.member') – both organizations are required to accept transaction
    b. OR ('Organization1.member', 'Organization2.member') – one is required to accept transaction
    c. OR('Organization1.member',
        AND('Organization2.member', 'Organization3.member')
) – logical operators can be combined

2. **Endorsement response** - If all needed peers return successful result called endorsement, a result of simulating the transaction (contains read-write set). Endorsement is cryptographically signed by all needed peers using their own copy of world state.
3. **Invocation request** - Client sends the transaction endorsement to orderer which verifies that all needed peers have signed the proposal and verifies policies. If all is correct, it creates a block consisting of transactions in order. This is logical order, not to be confused with strict timestamp based order.
4. **Verify policy** – The order of transactions are synchronized between orderers of both organizations
5. **Invocation**: Orderer sends transaction block back to Peers which apply the transactions from block and write to the ledger and update the state of world.

2.4 Blockchain for business

2.4.1 Technology for two- vs multi-sided markets

To understand how blockchain enables to provide better business value, we have to understand how it is produced currently [9].

Most business models today depend on two-sided platforms, where a trusted third party is involved for mediating the interaction between end-users. End users do not trust each other, so they need a third party that can be trusted by all. Transactions and resolving data left on the shoulders of the mediator and its platform. The platform must keep records and use architectural patterns to resolve transaction problems. All that involves cost since the third party must develop and maintain a platform for mediating and will charge end users for using it.

Multisided platforms enable end users trust each other since data is decentralized and all parties have a full copy of database. The problem is how to resolve simultaneous changes and how to synchronized data to all parties.

That is the reason why blockchain is multisided platform enabler [4]. It allows to create specific data store that contains immutable blocks and gives order to transactions. Mediator in this case will only monitor the trading and revenue is split according to contract and business rules.

2.4.2 Criteria for a platform to be blockchain enabled

With every technology, there are benefits and limitations. Blockchain based systems are no exception. Before committing to this technology, the current business model and requirements need to be analyzed for suitability to use blockchain [9].

If any of the next requirements do not apply, then it could mean that there are better alternatives than blockchain:

1. Durable or capital good – there is no business value for keeping an eternal ledger of non-durable goods
1. Need for trust for untrusted parties and permissions on items
2. Need for completely shared database
3. Need for multiple concurrent modifications
4. Need for single version database
5. Must depend on previous modifications – if it does not, then there is no consensus needed
6. Need to remove intermediation (someone who maintains joint database and enables trust and multiversion concurrency)
7. Technology can stand the transaction needs for system - blockchain can have limits to transaction performance as low as 3-20 transactions per second [4].
8. Data stored in block chain is limited. All data persisted to blockchain will remain there as long as the blockchain exists. Deleting it will not reduce the size of data.

There can be other technologies for implementing the solution for parking platform, but this paper concentrates on enabling platform on blockchain, specifically Hyperledger Fabric infrastructure and using chaincode for smart contracts.

3 Use case

The following chapter describes the parking application business models, architecture and implementation. Parking application contains all the major aspects of full solution and represents the traditional application, yet it is simple.

3.1 Currently available parking spot applications

Currently existing parking applications in Estonia are divided into two categories. In first type vehicle owner finds a parking spot and buys a ticket for predetermined time. This is done using separate machine, which accepts cash or card payment and prints out the parking ticket. It is possible to by using app on a smartphone. The cost of using parking app in Estonia is 32 euro cents. That is mediator fee collected by the parking app platform owner.

In the second type, you get a ticket before you enter through a barrier gate. Paying for parking time is done before leaving the parking lot. The fee for platform owner is included in the parking price. The standard fee is between 10-20%.

Both of those technical solutions is driven by a third party that has built a platform for managing the app and hardware and takes a hefty cut when you use their service.

We are interested in searching a cheaper solution. One that has smaller mediator fee and does not include vendor lock-in. That means the platform is shared between the companies that are interested providing parking spots.

What if you are in a hurry and need a parking spot and you do not have time to lose. There is no reservation possibility in current business models nor assigning specific parking spot per vehicle (premium spots could mean more revenue).

To get the cost down, we would like to remove the third party and make the community as the owner of data and software. Blockchain is technology that enables this.

3.1.1 Current business models in use

We provide a description of business model of parking model using Business Process Model and Notation (BPMN).

In conventional application, the platform owner takes the responsibility to mediate the transaction between the parking spot owner and renter.

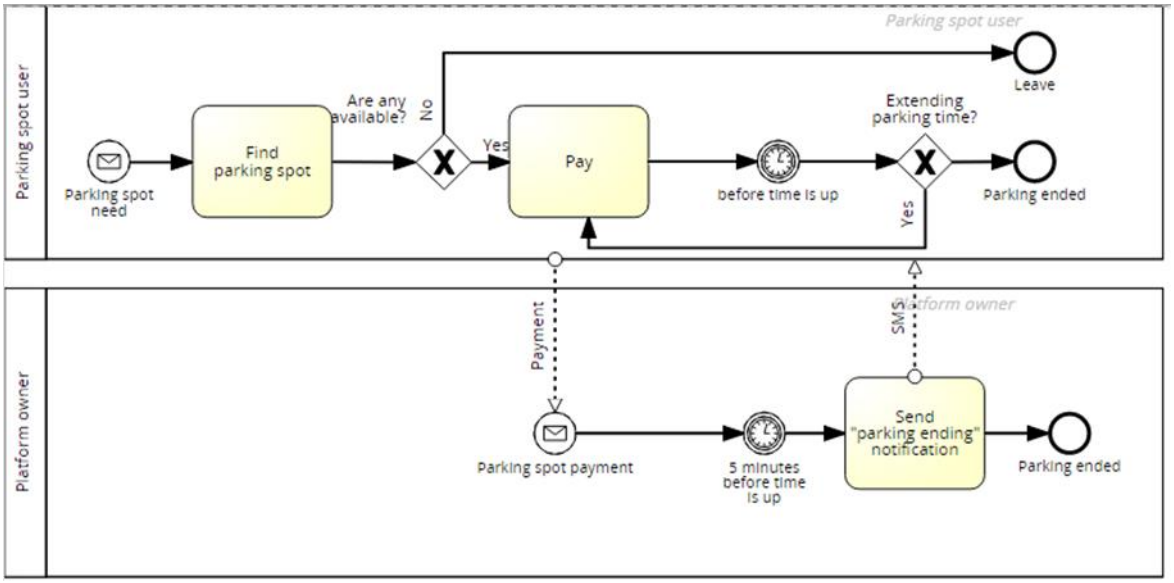

Figure 2: BPMN of conventional parking business model

3.2 Business process model for parking spot application

We provide a description of business model to be implemented using Business Process Model and Notation (BPMN).

We have three actors to business transactions: parking spot provider, parking spot renter and the software system that makes this possible. Parking spot provider can be any person or business who has business interest of creating revenue by renting out one or more parking spots. Parking spot renter is any person or business who has a need for parking spot and is willing to pay for it. Software system is blockchain enabled application that is implemented for this thesis.

3.2.1 Adding and removing parking spot from listing

Parking spot provider must be able to add the parking spot to list of available parking spots and remove it later. During business process “Rent parking spot”, the provider can assign daily periods when the parking spot is available.

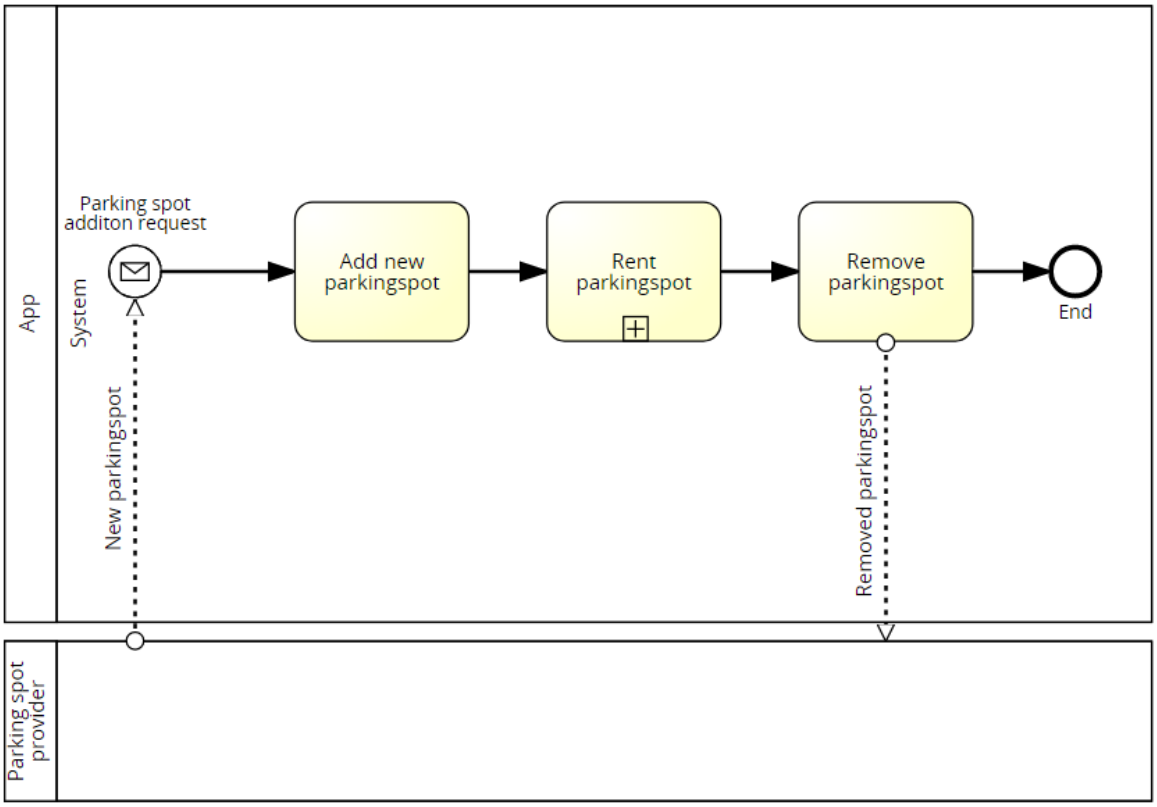

Figure 3: BPMN of adding and removing parking from listing

3.2.2 Reserving a parking spot

This is a use case that is not available in conventional parking spot business models. It gives the client the ability to reserve parking spot, for example a period of meeting, so he or she can be sure that upon arrival, the spot is free. The client provides a location and timespan. Multisignature smart contract I used with signatures from parking spot provider indicating it is available and another one from renter.

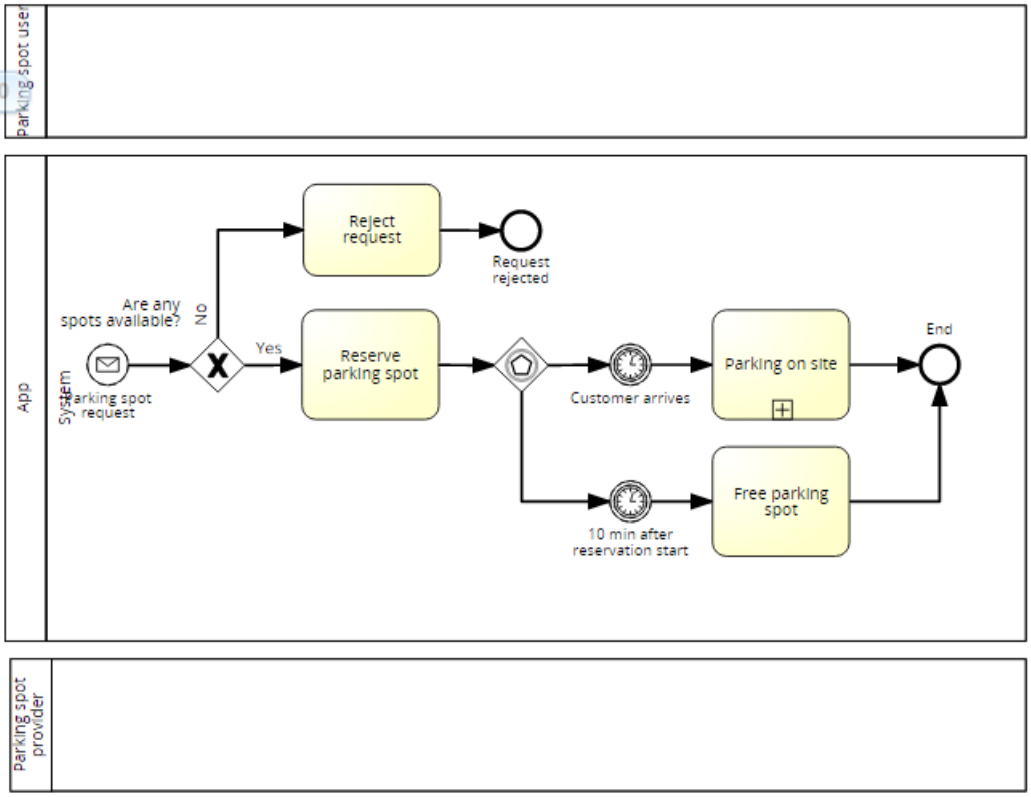

Figure 4: BPMN of parking spot reservation

3.2.3 Using parking spot: paying in advance

In this scenario, a person looking for a parking spot pays the full amount beforehand. The renter will specify the timeslot for using the parking spot. When the time is ending, the renter will receive a message from the system to create a new timeslot (and keep the same parking spot). If no notification is received from the renter, the parking spot will be available for usage after time slice has ended. Multisignature smart contract I used with signatures from parking spot provider indicating it is available and another one from renter.

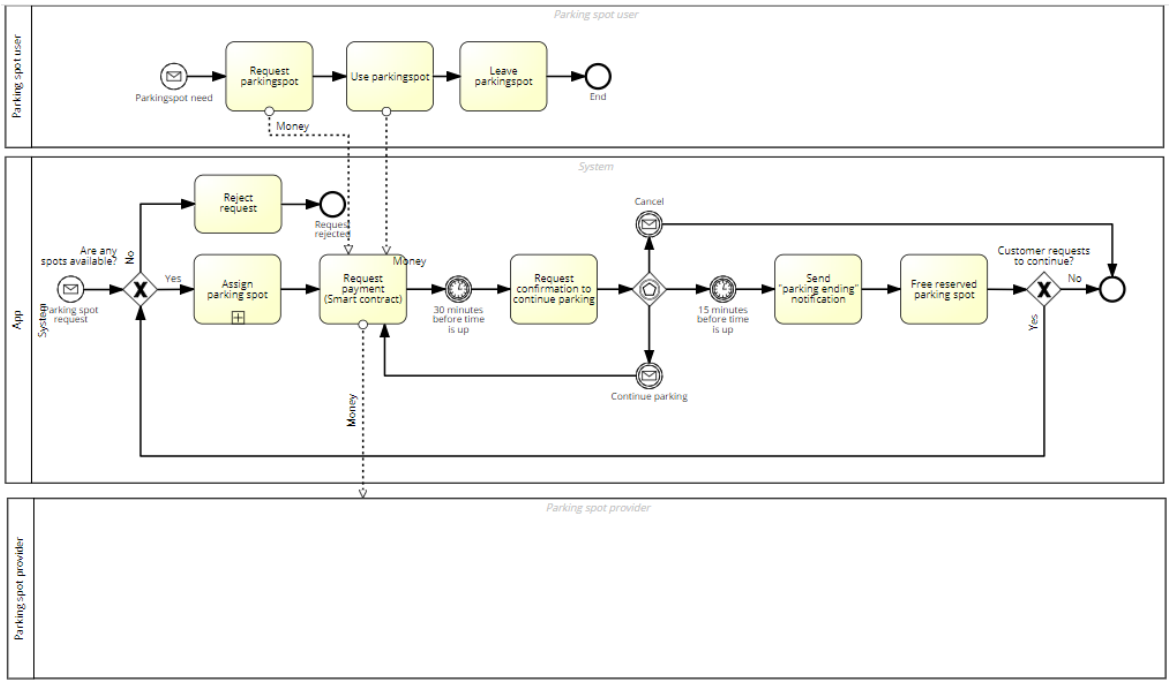

Figure 5: BPMN of paying for parking spot in advance

3.2.4 Using parking spot: paying after parking ended

Not always the renter knows how long he or she will be needing a parking spot. In that case it is more client friendly to pay after user has ended the contract and calculate the cost. This business model needs a predetermined maximum time limit, in case the customer forgets, clock keeps running and the final fee can be very expensive. Multisignature smart contract I used with signatures from parking spot provider indicating it is available and another one from renter.

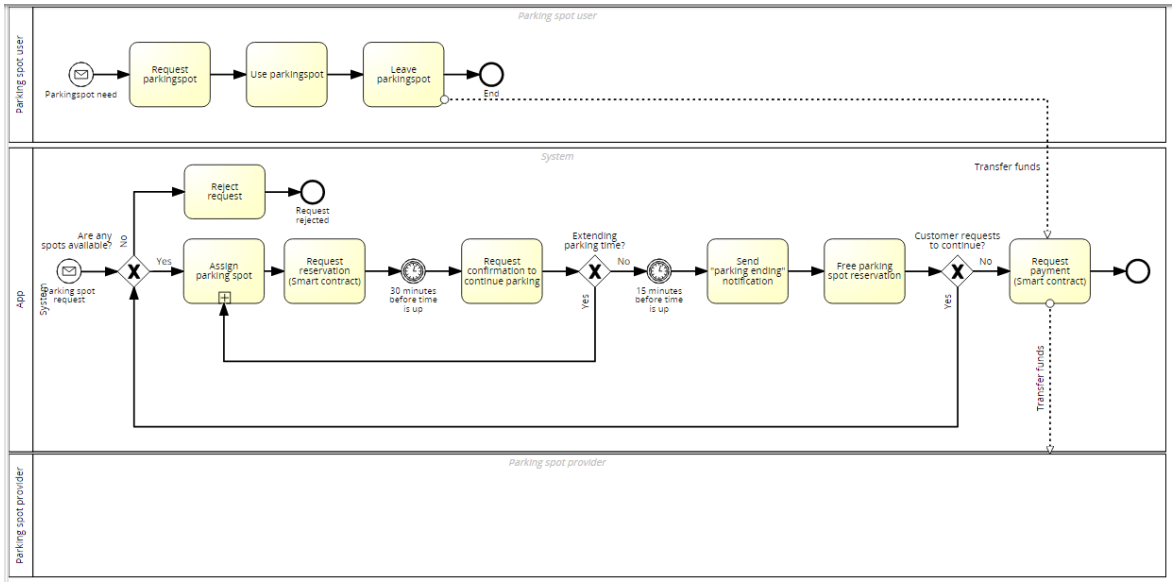

Figure 6: BPMN of paying for parking spot after using

3.3 Hyperledger Fabric prototype

3.3.1 Software architecture and implementation

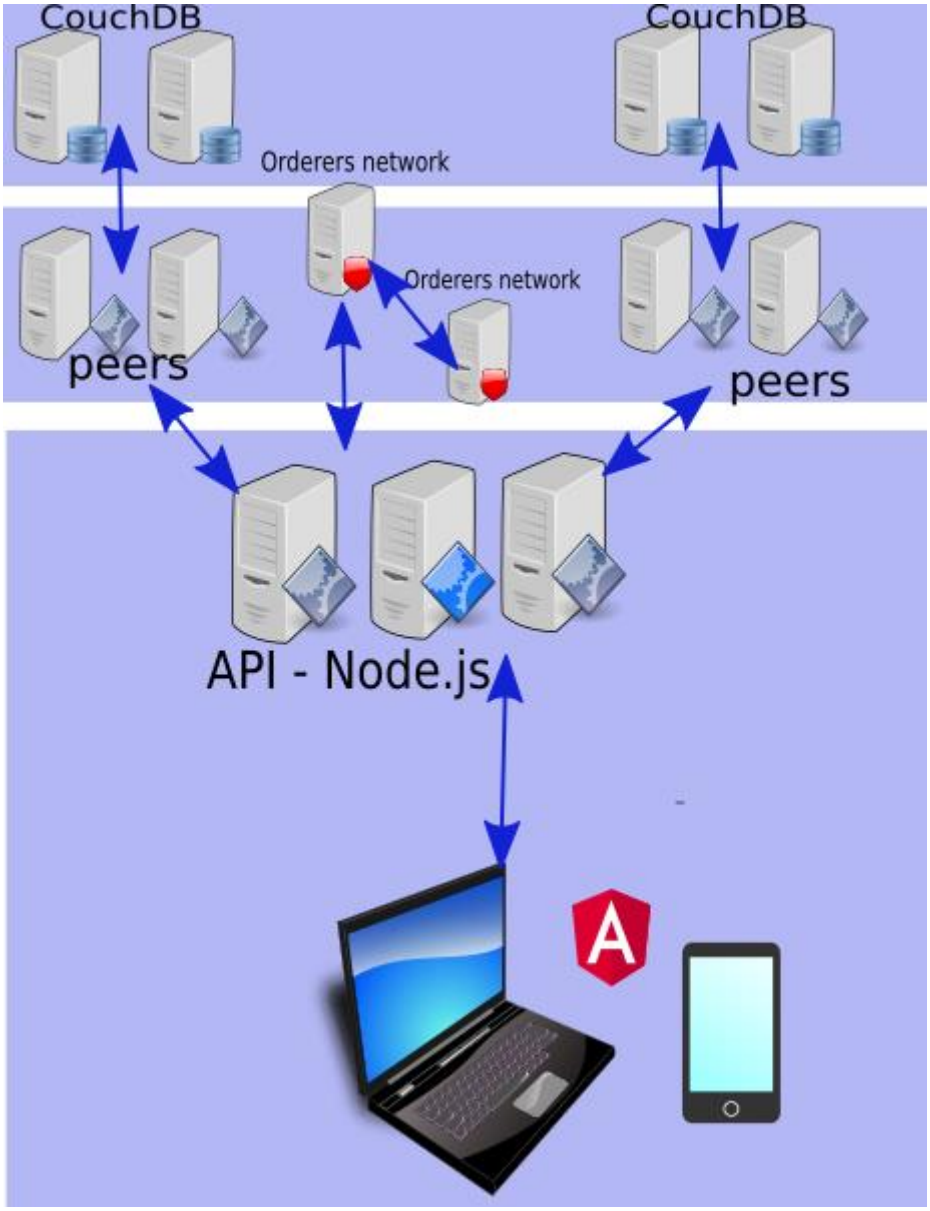

Figure 7 Hyperledger Fabric parking spot application architecture

Parking spot provider uses computer or smartphone to manage the information about the parking spot. Parking spot renter uses a computer or smartphone to search and pay for the parking spot. Smart contract running in the Hyperledger Fabric provides a contract between the two parties. Blockchain enables to implement these requirements without an third party.

Smart contract client side user interface is implemented using AngularJS and the API in Node.js. Package for the smartphone can be built from the same software using mobile app framework called Ionic. This enables the same codebase to be used for desktop and smartphone reducing cost. This is one possibility to implement the parking solution. System is built so that there can be multiple implementations and providers of software for client

side, enabling free market competition and lower prices. Since authentication and authorization is provided by smart contract, adding new client does not affect business processes are implemented in system.

The central implementation will be a smart contract framework Hyperledger Fabric. It will guarantee that transactions are accepted by all parties, are immutable after transaction has ended and allows access for interested parties, so the business transactions are transparent.

Implementation started by defining user stories and use cases. Use cases will be transformed into implementation code. In the end, this thesis the project code and underlying system available using version control system and Docker platform.

The source code is available in a public version control system: https://github.com/sven-zik/hyperledger-fabric-case-study-proto

3.3.2 Smart contract for parking spot

Smart contract is piece of software that runs the transactions, makes sure that the transactions is valid and calculates end results. This will make it the most valuable piece of the entire software where bugs may have expensive effect. This application uses multisignature contract with signature from owner of the parking spot and the renter of parking spot.

Smart contract must allow:

- Parking spot owner, to define
    1. a fee per hour, for the parking spot
    2. time for parking spot availability
    3. a fee for canceling the reservation
- Parking spot renter
    1. to search a parking spot in given location and time
    2. to make sure that parking spot is not double used (somebody has already using it)
    3. ability to cancel reservation and make sure only cancelation fee is taken

Example of one smart contract method for ending parking space rental and creating a payment:

```
func (s *SmartContract) EndParking(APIstub shim.ChaincodeStubInterface, args []string)
sc.Response {
          if len(args) != 1 {
                    return shim.Error("Incorrect number of arguments. Expecting 1, id of
parkingTime")
          }

          parkingTimeId := args[0]
          fmt.Printf("Ending parking for id: %s\n", parkingTimeId)
          //1
          // parkingTime, err := s.ParkingTimeService.Get(APIstub, parkingTimeId)

          //2
          // objectKeys :=  []string{parkingTimeId}
          // parkingTimeObject, err := s.ParkingCommonService.GetObject(APIstub, object-
Keys, ParkingTime{})
          // fmt.Printf("Got parking: %s\n", parkingTimeObject)
          // parkingTime, _ := parkingTimeObject.(ParkingTime)

          //3
          objectKeys := []string{parkingTimeId}
          compositeKey, _ := s.ParkingCommonService.CreateKey(APIstub, ParkingTime{}, ob-
jectKeys)
```

```
          oResultAsBytes, err := APIstub.GetState(compositeKey)
          parkingTime := ParkingTime{}
          err = json.Unmarshal(oResultAsBytes, &parkingTime)

          fmt.Printf("Got parking type: %s\n", parkingTime)
          if err != nil {
                    return shim.Error(fmt.Sprintf("Failed to find parkingTime(%s): %s",
parkingTimeId, err))
          }

          //TIME CALCULATIONS
          calculatedEndTime := CurrentTimestamp{TimeWindow: time.Minute * 5, Errors:
[]string{}}
          ts, _ := APIstub.GetTxTimestamp()

          //using current timestamp of server - will not work in multipeer envir
          // calculatedEndTime = time.Now()

          //Using transaction timestamp set by client, unsafe
          calculatedEndTime.TransactionTime = time.Unix(ts.Seconds, int64(ts.Nanos)).UTC()

          //Using transaction timestamp set by client with chekcing window (lets say 5min)
          endorsedEndTime := time.Unix(ts.Seconds, int64(ts.Nanos)).UTC()
          if math.Abs(time.Now().Sub(endorsedEndTime).Minutes()) < calculatedEndTime.Time-
Window.Minutes() {
                    calculatedEndTime.TimeWindowCurrentTime = time.Unix(ts.Seconds,
int64(ts.Nanos)).UTC()
          }

          // USING Timestamp Protocol
          // Time-Stamp request with nonce, to create with OpenSSL:
          //   $ openssl ts -query -data data.txt -cert -sha256 -out reqnonoce.tsq
          tspResponse :=

          timeServerSignedResponse, err := timestamp.ParseResponse(calculatedEndTime.
tspResponse)
          if err != nil {
                    calculatedEndTime.Errors = append(calculatedEndTime.Errors,
fmt.Sprintf("TSP: %s", err))
          } else {
                    calculatedEndTime.TimeServerCurrentTime = timeServerSignedResponse.Time
          }

          //from time chaincode
          timeChaincodeResponse := APIstub.InvokeChaincode("time-app", To-
ChaincodeArgs("GetCurrentTime"), "mychannel")
          if timeChaincodeResponse.Status != shim.OK {
                    errStr := fmt.Sprintf("Failed to query chaincode. Got error: %s", time-
ChaincodeResponse.Payload)
                    calculatedEndTime.Errors = append(calculatedEndTime.Errors,
fmt.Sprintf("CHAINCODE: %s", errStr))
                    // return shim.Error(errStr)
          } else {
                    chaincodeCurrentTime := HyperledgerFabricTimestamp{}
                    err = json.Unmarshal(timeChaincodeResponse.Payload, &chaincodeCur-
rentTime)
                    if err == nil {
                              calculatedEndTime.ChaincodeCurrentTime = chaincodeCur-
rentTime.CurrentTime
                    }
          }

          parkingTime.CurrentTimestamps = calculatedEndTime
          //END

          parkingTime.ParkingEnd = calculatedEndTime.TransactionTime

          delta := parkingTime.ParkingEnd.Sub(parkingTime.ParkingStart)
          totalCost := int(delta.Minutes()) * parkingTime.CostPerMinute
          parkingTime.Cost = totalCost

          s.ParkingTimeService.Save(APIstub, parkingTime)
          fmt.Printf("Saved parking type: %s\n", parkingTime)

          owner, err := s.UserService.Get(APIstub, parkingTime.Parkingspot.Owner.Id)
          if err != nil {
```

```
                    return shim.Error(fmt.Sprintf("Failed to change parkingspot owner bal-
ance: %s", err))
          }

          renter, err := s.UserService.Get(APIstub, parkingTime.Renter.Id)
          if err != nil {
                    return shim.Error(fmt.Sprintf("Failed to change renter balance: %s",
err))
          }

          owner.Balance.AddCents(totalCost)
          renter.Balance.SubtractCents(totalCost)

          owner, err = s.UserService.Save(APIstub, owner)
          if err != nil {
                    return shim.Error(fmt.Sprintf("Failed to add parkingspot owner balance:
%s", err))
          }
          renter, err = s.UserService.Save(APIstub, renter)
          if err != nil {
                    return shim.Error(fmt.Sprintf("Failed to reduce renter balance: %s",
err))
          }

          resultAsBytes, _ := json.Marshal(parkingTime)
          return shim.Success(resultAsBytes)
}
```

3.4 Executing nondeterministic functions

Smart contracts can only execute deterministic functions since the contract is execute in multiple nodes, the result for the same input state must be always the same output state. Otherwise, the transaction will be rejected. In the example application, current timestamp was needed to record the ending of parking time. Possible solutions were considered:

1. Providing timestamp from the client to all required peers
2. Getting timestamp from smart contract running in peer

First solution would make the system simple and smart contract executing nodes would get the same timestamp. Hyperledger Fabric protocol even provides the client timestamp as default to the smart contract. Unfortunately, the time from client is unreliable and cannot be used without some added trust.

Conventional solutions use the local time of the API, since transaction is performed at API level. With smart contracts, the business logic is inside blockchain network and cannot trust the API.

Here are two possible solutions for trusting the client timestamp.

3.4.1 Trusting client time stamp within tolerance

This is not a new idea, both Bitcoin and Ethereum use the notion of that the time is what most of the of the participants agree. Timestamp compared to previous block timestamps giving it a minimal value and not more than n hours into the future. This gives the timestamp very big tolerance, but these are public blockchain and with myriad of connecter time provided can have consensus algorithms to decide the time.

In the case of parting applications, hours is not acceptable accuracy. This needs accuracy in minutes. A more accurate solution must be found.

Proposed solution is using the peers internal time and comparing it with the clients timestamp. If the clients timestamp is within the tolerance of smart contract, the time of the client is accepted. Peers timestamp is kept accurate with the same methods as in conventional solutions – Network Time Protocol.

This solution can provide accuracy in minutes.

The time window must selected to cover latency from client to all peers. If a connection to one of the peers is down or lagging, then the client must try to deliver a new timestamp. Otherwise it will not be any more in the window of tolerance and will be rejected.

This will create a new problem, if the there is connection issues and the client connects later on, the lost time will cost the client money.

3.4.2 Cryptographically signed timestamp

If minutes is not precise, enough for a time widow requirement or we cannot accept network outages, we need a more accurate solution. The solution is to use trusted timestamping, using cryptography to sign piece of data and getting a time of the signature. This signature can be verified later on in the smart contract,, without connecting to network or using other nondeterministic functions. In this case the client timestamp as the data and what we got back was the timestamp on the service.

Since the protocol is well defined, the service can run on the same network as Hyperledger Fabric or by a trusted third party. Trusted timestamp request can be made by application running in the client hardware or by API. API is also considered to be untrusted client in the Hyperledger Fabric platform.

Validating correctness of the timestamp in a smart contract that is running in Hyperledger Fabric is not hard. Since smart contract are written programing languages that have been in use for a long time and all the necessary functionality is already packaged in reusable libraries, verifying the timestamp is not much work. Signed timestamp can be verified using deterministic functions. Timestamp is signed with timeserver certificate and all we need to do is to verify that we trust the signature using PKI. This is similar how software clients trust remote HTTPS servers.

If connection to all peers is not working right now, then signed timestamp can allow for eventual consistency – initial timestamp is serialized into immutable data structure at the time of creation and when the message is delivered later, the initial time can be trusted.

Verifying this data structure in Bitcoin is not possible at all and Ethereum does not support cryptographic algorithms in Solidity.

3.4.3 Keeping current timestamp in a another blockchain

Since each peer executes the smart contract and has its own state, it is can be a good idea to keep the current timestamp that is near the smart contract. The smart contract has read access to other chains that are running on the same peer and if correct access rights have given to it. So it is an easy deterministic read operation from, since all peers have the same state.

It must be used with time window to give time for the gossip protocol to guarantee that the new block will be delivered to all endorsing peers. The latency of this operation depends on the configuration, but is safe to consider it is less than 10 seconds.

This solution depends on another client that is committing the new current timestamps to this special channel after every predetermined period. Server, where the client is running, needs to be synchronized with timeservers to provide valid time, for example Network Time Protocol. This is with upmost importance, since if the client fails or its clock is not correct, it can jeopardize the rest of the system and make the transaction validations fail.

To raise the trust on committing client, the smart contract can also use the same solution of time tolerance, that is compared to peers clocks. If the transaction validation starts failing of peer with a bad clock, then the system could get into broken state with a stopped clock for parking application.

This is why in the prototype, a mix if these technique is used. If one of the services is down, then other replaces it. Order of executing is determined by the precision that it provides.

1. TSP protocol based time
2. Hyperledger Fabric blockchain based time
3. Client timestamp with time window

3.5 Time persistence and querying

Hyperledger Fabric allows data persistence of any byte stream, so anything that serializes into byte stream can be persisted. This is the part that requires the stream to be in a JSON format. It enables querying syntax called selectors. They are JSON object describing the document properties we are searching for.

To make time queries work we need to fulfill some of the requirements for time persistence. The JSON date is not a date type by a string. Comparing dates come down to comparing strings.

During persistence, convert all dates to ISO 8601 date format. This allows to compare dates as strings. Convert all dates to use Zulu time. Querying will not work if the time is in different time zones, date is actually text, not a date type like in SQL databases. The date type should be persisted using strict format, usage of precious should be same during writes. We encountered issues when Go persisted time library so that it lost the milliseconds off the ISO date when they were 0. Example: time 2018-05-09T04:37:57.000Z was persisted as 2018-05-09T04:37:57Z and that made the queries fail. So we recommend making a decision to persist timestamps with fixed precision and making sure that it is persisted into world state with same strict length.

3.5.1.1 Timestamps

Timestamps are a way to prove what order did the events and transactions happen.

Conventional systems timestamp can be generated by repository (database for instance) or middleware (micro services or server API) from current server time, that is kept synchronized with precise timeservers. Since there is one trusted process that generates the current time, then the result is trustworthy and can be propagated through the system.

Hyperledger Fabric uses multiple peer servers that all get the transaction from the client. The client is considered untrustworthy and thus the peers must be responsible for providing trusted timestamp. When the transaction is propagated to different peers, it is impossible for all the server have synchronized timestamps within the accuracy of millisecond. There has been suggestions on other blockchain enabled platforms that the timestamp should be the

one most participants, that validate the transaction, agree on. These time differences can be off by 10 of minutes. In case of parking spot application, that precision is not enough. Parking fees can be high and acceptable accuracy should be at least minute accurate. There are multiple possible solutions:

1. Trusting the client when the time is newer than the timestamp of last block in the chain. Unfortunately, Hyperledger Fabric does not persist timestamp into a block when Orderer sends it to peers. This might change in the future. The timestamps in transactions that are included in the block, contain only the timestamps from the client.
2. Trusting the client when the timestamp is within allowed tolerance. That gives us more accuracy. For example, the client sends a timestamp of 1.1.2018 00:00:53, if the server receives the transaction within the time of time window and time does not differ from client more than 1 minute, then the transaction is accepted and the time of client is used as current timestamp.
3. Using a validation oracle, mechanism for accessing external state
    a. Internal validation oracle – internal to the blockchain network. Since the smart contract can be written in programming languages that have existed for long time, there are multiple reusable pieces of code that can be used for accessing validation oracle – that means basically there are libraries that can be used to connect to such server within the smart contract execution environment – within Peer. The authenticity can be verified by TLS/SSL certificates or other similar cryptographic methods. Since this is internal to the network, then this is
        i. Fast, since it is internal connections and that can be controlled by the owner of the network
        ii. correctness of time in the oracle can be managed by the owner of the network
        iii. Peers are not responsible for ordering and generating the blocks. So if one connection blocks the transaction, then the execution of other smart contracts are not blocked and only this transaction will be part of the next block.
    b. External validation oracle – the timestamp must be generated by untrusted client and signed by external validation oracle that the internal Hyperledger Fabric network trusts (this trust is verified by Peers) using TLS/SSL or other similar cryptographic methods. It will be the responsibility of the client to make sure that the validated data is included into the transaction.

3.5.2 Queries

Capabilities for complex queries is enabled by the world state database called CouchDB. This enables searching data that no other smart contract platform enables. Example of time search is given in figure below.

```
{
  "selector": {
    "$and": [
      "parkingTime": {"$gt": “2018-03-15T05:25:59.000Z”},
      "parkingTime": {"$lt": “2018-03-15T06:25:59.000Z”},
    ]
  },
  "fields": [“id", “parkingspotName", “location“, “price"],
  "sort": [{“price": "asc"}],
  "limit": 10,
  "skip": 0
}
```

Figure 8 Complex Query using timestamps

3.6 GIS support

Parking spot application is dependent on users finding parking spots using location based searches. This requires the data store to support spatial persistence and queries for finding locations in a given map window. These are called range and radius queries.

Hyperledger Fabric does not support spatial queries or does not a dedicated data structure. In the context of this applications, parking spot can be modelled as a point on map.

To find a parking spot a we propose a solution that uses GeoHash [10]. It is enables simple bounding box search using string based approach. This solution works with longitude and latitude coordinates, more exactly using projection called WGS 84 (also known as WGS 1984, EPSG:4326). For every coordinate, it is possible to provide precision to what accuracy it is mapped. Using GeoHash length of 9 results in accuracy < 5 meters and length 11 in accuracy of 15cm. This is accurate enough for parking spot application and we can use GeoHash as spatial index.

When adding parking spot to the blockchain, we provide latitude and longitude. Smart contract uses will use built in parameter to decide the length of GeoHash. We chose 11 because it will provide accuracy enough to describe the location of smallest parking spots, for instance motorcycle. Calculating the hash is deterministic and can be in the smart contract. Data is persisted in a triple: latitude, longitude, hash.

Searching using GeoHash indexes is a matter of comparing strings. For example, searching area of 600m x 600m means you search spatial index that matched the 6 first characters.

Example area of 600m x 600m in the city of Tartu, Estonia: https://www.movable-type.co.uk/scripts/geohash.html?geohash=ud6upe This means all points starting with ud6upe are guaranteed to be inside that area.

Search algorithm was included inside of the smart contract, client software can query it with triple: latitude, longitude and zoom. This was intentional design decision, since the window

of interest can cover multiple GeoHash areas. They have to be calculated and separately queried.

The secondary filter is applied by map component in client user interface, basically all the locations that are outside of window of interest are just not shown.

4 Evaluation

In this chapter, we describe framework and implement a comparison of conventional system and Hyperledger Fabric. This platform enables the usage of generic purpose programming languages and vast possibilities for using it in enterprise markets. There are no good existing frameworks for comparing blockchain solution to a conventional solution. This thesis will create a framework, combining case study of using it as shared platform with for product centric information management [9] and approach developed by a case study describing blockchain as Software Connector [4]. We propose to add several properties that are considered essential in conventional systems.

Hyperledger Fabric is initially compared as a data store and we study the properties of as a connector, how concurrent writes are enabled and how the consensus of the data store is compares to conventional. Then we discuss the coordination service properties with a conventional system. We compare system properties, like performance, reliability, security, migration and development support. We add the properties of continuous integration and delivery.

4.1 Evaluation framework

4.1.1 Atomicity, consistency, isolation and durability

There is a need for multiple concurrent modifications in a shared system. In any software system, the ACID properties of a transaction are desirable. Atomicity is requires that transaction is accepted entirely or if any part of transaction fails, the whole transactions fails. Allows the client to build on assumption, that if the server did not return error during the transaction then it is guaranteed to persisted into the system. Consistency guarantees that the transaction takes system from one valid state to another. Isolation is property that all concurrent transactions is the same as applying them sequentially. Durability means that in the case of disaster, the data is persisted even if the system crashes.

4.1.1.1 Transaction validation

Transaction is an operation that takes the data from initial valid state to a new valid state. It is desirable that transactions support data consistency and data integrity. The possibilities that a platform provides are crucial.

4.1.2 Scalability

A software system has to grow with the growth of the users. This requires more computational power. Ideally, adding resources would be linear to computational power. Modern software must handle short spikes and short periods of high demand. If the software system does not respond in sensible time, the client will take their business elsewhere. Able to scale with usage is a desirable property of software systems.

4.1.3 Consistency, availability and partition tolerance

These three properties are all desirable in a modern software systems. CAP theorem [11] says if you lose one of the properties for the system, then you must choose between the rest of two properties. The balance between them is considered when designing a new software system architecture. Consistency, also called linearizability, is a property to allow an operation to see all the previous operations that have successfully completed before it starts. This means that the operation is getting the latest version of data. If this property is not available, the data could be stale. Availability is the property to guarantee that if a node is reached, then it will return a non-erroneous response. Client software can be certain that if transaction is delivered then it will not get lost. Partition tolerance is a property to tolerate individual server failures or network outages in a distributed system.

4.1.4 Security

High availability systems span multiple global networks. There are firewalls and physical security measures, but it is essential that those networks are secured and do not allow an attacker to gain access to server. Compromised server cannot be used to compromise other parts of systems. Thus, some schema is needed to create trust between the nodes in the network.

Most software systems deal with sensitive data like personal and business information. This data must be protected from unauthorized user access and is an essential part of software systems. Authentication is an answer to question: who are you and are you present? Without having a certainty of the users identity, there cannot be a secure system. Authorization tells the system what are the permissions for a user. What operations can be executed and what data can be accessed.

From a business point of view, confidentiality is a key property of software systems. Businesses and private people need to certain that their data does not leak. This can be regulated by law to enforce privacy or cause financial losses to businesses.

4.1.5 Migration

Modern software is in a continuous change and migrating the changes of data and upgrading infrastructure is important part of the system. The possibility of upgrading the platform software is vital because of bug fixes and security updates.

4.1.5.1 Continuous integration and delivery

Modern system evolve rapidly and it is desirable for push out new features and bug fixes quickly. Bugs in global software could mean substantial loss of financial resources. Leaving behind with a new feature could mean loss of revenue.

4.1.6 Development support

Deciding to develop on a new platform includes risks. These risks can be mitigated by good development support. These include documentation for the developers and architects. Forums for discussing parts that are not very well documented.

4.2 Comparison of conventional and Hyperledger Fabric

4.2.1 Atomicity, consistency, isolation and durability

Relational databases that support transactions have built in ACID compliancy. They do however differ how isolation is implemented. Most of the databases do agree on read committed isolation implementation. NoSQL databases support ACID properties, but only on single record transactions. There are no multi-record transaction support available. Most modern relational and NoSQL databases support Multiversion Currency Control (MVCC). No locking is needed to achieve ACID properties. MVCC enables increase of the database throughput, since no locking is used and operations do not have to wait for locks. Instead, when data is read the current a snapshot of the data is used. Writing new data will cause the database to create a new version of this snapshot. After the data is committed the new snapshot will be used in next read or write executions.

Execution of **Hyperledger Fabric** smart contract methods are guaranteed to atomic. The smart contract transaction can fail in two different steps. If there is an error during execution of contract, the error will make the endorsing peer fail and error is returned. Even if there is a bug in client and the transaction reaches the Orderer, it will not accept it and error is returned. If the transaction is endorsed, but fails later after Orderer has sent it to peers to be added to blockchain, the transaction will be rejected and logged as failure. The client SDK make this last rejection step available to client code and success of transaction can be checked.

There are no consistency rules in Hyperledger Fabric data store and that means the consistency is left for the smart contract to enforce. If the consistency checks fail inside the contract, then the transaction is not added to ledger and thus making the ledger fully consistent in the sense of ACID.

Fabric data access methods differ significantly from SQL and NoSQL. There are multiple ways of reading data from chain. Serializable isolation is guaranteed for most of the data access methods. There are two exceptions when phantom reads are not detected: GetQueryResult and GetHistoryForKey. The first one executes a rich query from using the world state. This read is not re-executed in the final block writing phase and should not be used during smart contract methods that use a write method PutState, method that adds new value for a key. The second one is used for reading the history from the blockchain for the key. Data is similarly kept in separate database and should not be read during a write.

There are some side effects using PutState. It will add changes to write set, but reading the state for same key using GetState will return the initial value. That means that the changes written to write set are not read during GetState.

The client has to go through many steps to commit data to ledger and the last one is an event that will indicate if the final blockchain block write included the transaction. This is indication that the transactions are fully durable and in the event of crash, the transaction will be written to multiple peers and are not lost.

There is no locking in Hyperledger Fabric. All operations use Multiversion Currency Control with postimage info.

4.2.1.1 Transaction validation

In a **conventional** system, transactions are validated by data store and software business rules. Relational data model dissects data into separate tables and databases use constraints, triggers and foreign keys to accept to enforce data integrity. Constraints check the data nullability, attribute type, allowed range for numbers and that strings match patterns. Foreign keys are used to check that data is composed correctly, as the relational data model describes it. NoSQL databases allow dynamically changing data structures and depends that the business layer manages validation before data is persisted. There are no internal mechanisms to ensure any consistency rules.

Business layer can use domain model specific constraints similar to database constraints, for example Java Validation API. These double the constraints in relational databases and are checked before the data is persisted by implementing framework. If the data is not valid, the persistence layer will raise an error and data never reaches data store. If there is a need for more complex validation, it can be implemented in the business layer and involves custom written domain specific logic.

Transactions management can be divided into two: one changes data in a specific data store and distributed transactions, that are executed on multiple heterogenic resources. These resources can be databases from different networks and types.

X/Open XA architecture [12] provides ACID like properties to distributed transactions. The architecture uses separate transaction processing monitor, which coordinates the transaction. It uses two-phase commit protocol to guarantee that all or none of the transaction succeed. In first phase, data is sent to all resources where the commit is simulated and if any of them return error, the transaction fails. In a happy case all of them return success and the second phase commits the transaction.

There are also software integration patterns that can be implemented in business layer, for example Try-Cancel/Confirm [13], to enable transactions in system does not support any distributed transaction architecture.

Hyperledger Fabric supports creating multiple blockchains, called channels, in the same network infrastructure. Channel is the storage for a smart contract that is called chaincode. Channel can have multiple chaincodes making transactions on it and specific chaincode can be installed on multiple channels. In the latter case, the chaincode installation changes the data only on the channel it is installed on. This is similar to conventional software that can use different instances of database.

Client software sends a transaction proposal to all endorsing peers. All peers are specified in a policy specified when the smart contract is deployed. The peers validate that the client software is authorized and simulate the transaction. If it succeeds, then a signature is returned with read-write set. Read set consist of the original data that was in ledger and was read during the execution of smart contract. Write set contains the new state of data that was written by the contract. When the client has gotten signatures from all required peers, the signed endorsements are sent to orderer. Orderer collects all transactions that are happening in the system into a block giving them total order and delivers them back to the endorsing peers to be added to the blockchain. Peers run the transactions in the received order, checking that none of the previous transactions have not changed the data that current transaction reads. If the data was changed, then the transaction will be rejected, logged and the rejection of this transaction is notified to client. Client will have to resolve the conflict automatically in code or with a help of end user..

There are no built in constraints or Hyperledger Fabric that would be applied when persisting data to the ledger. Validation must happen in smart contract. There is also no built in validation framework that can be used with chaincode domain classes. Since the smart contract can be written in Go, Java or Node.js, it is possible to choose from myriad of existing frameworks. Similarly, to conventional software system, custom domain specific validation is written into smart contract.

Smart contract can execute another smart contract in the same channel or on an another channel. If it invokes a chaincode on the same channel, the all writes are included in the transaction. The read-write set from the operations in another smart contract will be added to current read-write set and making them part of the transaction. If the chaincode runs on another channel, then only read operations are executed. The other blockchain will not change as a result of this type of execution.

There are no documented built in distributed transaction support for Hyperledger Fabric. The cross channel transaction is planned for the future, indicated by tag *post-v1 feature*. This would allow distributed transactions inside Hyperledger Fabric blockchain. Fabric does not have support for distributed transactions using for X/Open XA architecture in heterogenic systems.

4.2.2 Scalability

Relational databases provide multi-record transactions support and are ACID compliant. These properties mean that they scale vertically very well, meaning the server can have more processors, memory etc. This means that single server must execute complex queries and execute transactions. Adding new clients or vast amount of data will make the vertical scaling reach it limit and get very expensive very quickly.

Unfortunately, they do not scale well horizontally - adding more servers to service requests. There are techniques that relational databases use to scale horizontally, they include:

1. Shared disk architecture – multiple servers are using a shared disk SAN with specially designed file system to able to read and write data with multiple nodes.
2. Data sharding - data is logically partitioned and any parts are written manually to different database node. This can be implemented in software code or by database implementations on a value in a table. This will get complex very quickly and may lose transactional data integrity.
3. Replication - One writer, multiple readers. One database is considered as the master and receives writes. Writes are then replicated to slaves using transaction logs. Read operations use slave and do not burden master. If the master fails, hot standby server takes over.

NoSQL databases are designed to fix the issues with relational database horizontal scaling difficulties. They do this by sacrificing multi-record transactions and consistency of relational data model. Data is aggregated using JSON or similar format and persisted and read by key. There are no complex queries. Horizontal scaling is achieved by

1. Data auto-sharding - data is logically partitioned by database and full aggregates are written automatically to different database node.
2. Replication – similar to the pattern with relational databases, but the data is replicated to the slaves in an already aggregate format and no transaction log is used. It can include master-slave, master-master or quorum-based replication.

Microservice architecture is used to separate business rules into logical groups and deploy them separately. Services are built stateless and that makes it easy to scale by adding new servers and running extra services. Need for more processing power can be even made automatic using auto scaling cloud computing.

Blockchain based systems are never as scalable as conventional systems. Changing the architecture from order-execute to execute-order-validate, IBM has managed to get 3500 transaction per second in Hyperledger Fabric using a peer containing 32 vCPU's [2].

The endorsing peer manages writing of new data. After successful block write, the new block is broadcasted to other peers using gossip protocol. This is known as master-slave replication. If a master peer fails, a new master is elected from the other peers.

Orderers are using highly horizontally scalable Apache Kafka network to achieve consensus and total order.

Sharding can be implemented by persisting partitions to separate channels. There is no automatic-sharding support and thus must be manually implemented. The sharding can happen in two separate locations. In the chaincode or client. As of this moment, invoking other chaincodes is possible, but the write operations are not part of the transaction and not persisted. This support is planned in the future and would give possibility to use sharding in smart contract. Making sharding work on client must work in every organization clients exactly the same and thus making it not feasible.

4.2.3 Consistency, availability and partition tolerance

Conventional systems are built so that two out of three guarantees are chosen for the software system. The two chosen depend on the type of system built.

Relational databases provide consistency (not to be confused by ACID consistency) and availability as their first choice. Consistency is provided by locking the data or using last writer wins Multiversion Concurrency Control. Availability is provided by transaction log. In case of failure, the log is re-read and the database is returned into consistent state.

Achieving partition tolerance is difficult and prone to problems when using relational databases. If master-slave replication is used to leverage the load for the writing master, slave nodes are considered to be eventually consistent. When using data sharding, the consistency cannot be guaranteed since the data is shared between multiple independent servers and multi-record transactions are not ACID anymore.

NoSQL databases were introduced to provide more effective partition tolerance sacrificing either consistency or availability. NoSQL databases like Cassandra and Dynamo like system sacrifice consistency if there is connection lost between servers. MongoDB and Redis sacrifice availability and will return error.

Software systems, similar to parking applications, are designed keeping availability and partition tolerance in mind. Eventual consistency is acceptable in these kind of systems. If the system is not available then the client takes their business elsewhere. It is acceptable that during network outage inconsistencies will happen. If there is conflict, for example two client buying the same last and only item, then one of them is notified later that the item was already sold out.

In **Hyperledger Fabr**ic the smart contract policy is used to decide what are the system properties. Let us consider two organizations that are using a channel with two configurations:

1. AND ('Organization1.member', 'Organization2.member') – both organizations are required to sign the transaction
2. OR ('Organization1.member', 'Organization2.member') – one of the organizations is required to sign the transaction

When both organizations decide to sign all transactions, it should be considered a strongly consistent system. Peers divide into endorsing peers and peers. Endorsing peer is master type node and accepts writes. Other peers are read only peers and are synchronized by using gossip protocol.

Client contacts all needed endorsing peers in both organizations. If a transaction proposal succeeds, it will be signed and returned to client. Client sends all of the endorsed signatures to orderer. Orderer receives all the transactions from multiple applications and uses Apache Kafka and Apache Zookeeper to provide crash tolerance and total order to transactions. Kafka guarantees atomic delivery and that both organizations receives the transactions in the same order and generate the same blocks. Current implementation does not provide implementation that is Byzantine fault tolerant, but it is planned for the future. These blocks are then sent to endorsing peer and appended to channel. Then the gossip protocol is used to update the read only peers.

When OR notation is used, the client has to get endorsement only from one of the organization peers. The orderer with Kafka implementation applies strong consistency even when OR notation of policy is used. If read only queries from client go to the non-endorsing peers, then an eventual consistency guarantee is given.

Availability means that if a can be node contacted then it should not return error. If an endorsing peer goes down, a new peer is selected and availability is restored. Orderers are crash tolerant by design. When the transaction blocks are delivered by orderer to endorsing peers, the read set is compared with current state and if they do not match then the transaction is rejected. This makes the system not compliant with availability in the sense of CAP theorem.

4.2.4 Security

In **conventional** system the nodes running in internal network are considered secure and no extra trust is needed. In more secure software systems, a X.509 based identities like TLS certificates, can be used to provide authentication to each member of infrastructure. Most of well-known databases support TLS authentication. Commonly known programming languages have support to enable TLS based authentication for API and when communicating with internal services. This can be even automatically enabled by virtualization software (for example Docker Swarm) using a central service that provides identities to all services.

Hyperledger Fabric is permissioned network and membership identity service is provided with the platform – it is called Membership Service Provider (MSP). All nodes are authenticated, no anonymous access is allowed. Roles are peer, orderer and client. Each participant has a certificate associated with identity. Well known Public Key Infrastructure is used with X.509 digital certificate, same that is used to authenticate and encrypt HTTPS protocol. To provide trusted Certificate Authority, Fabric CA is introduced. Fabric also support revoking

existing participants certificate using Certificate Revocation Lists. Public and private keys can be in file system and Hardware Security Module is also supported using PKCS11 API.

Fabric CA can have multiple back ends: LDAP, MySQL, PostgreSQL and SQLite. Authentication to back ends can use username/password or TLS. Latter authentication scheme does not use Fabric infrastructure and has to be set up manually.

Users are **authenticated in a conventional system** by knowledge, ownership or inheritance factor. For example username and password, PIN, biometric information, fingerprint. There are availablesystems that can be used to provide this: libraries, services like OpenID, Public Key Infrastructure based solutions. Latter includes software based applications like SmartID and or physical like Estonian ID card. Schemas are divided into single, two- or multi-factor authentication. Latter two meaning that more than one is required for the user to authenticate.

OAuth 2.0 is considered to be industry-standard protocol for **authorization**. It supports the notion of permissions and roles. It is implemented as a separate authorization server which the client app accesses in the end of authentication step.

Hyperledger Fabric supports single factor **authentication** schema. TLS certificates that are kept in a software wallet in the client, accessed by username. Each request to the other nodes requires credentials. There are no anonymous users. They are grouped into roles administrators and users. Authorization uses Hyperledger Fabric built in permission based solution, called Attribute-Based Access Control. The attributes can have a value. Attributes are not possible to assign to roles (assigning role would assign all attributes). The attributes are set inside an X509 certificate using an extension called Abstract Syntax Notation Object Identifier.

There are two main approaches of providing **confidentiality in conventional systems**. Database encryption can encrypt the whole database, table or a column. This enables the execution of queries on encrypted tables or columns. The encryption key is kept in near database instance. The administrator will still have access the unencrypted data.

Another option is to encrypt data within end user software. The data is received encrypted and when data needs to be read, the software decrypts the data. This separates the key from data. Unfortunately, the encrypted data column cannot be used for querying data. This approach poses limits to the usability. If the holder of end user software is a third party, then the full confidentiality cannot be guaranteed.

Hyperledger Fabric is built with confidentiality in mind. A separate channel can be created for data that needs to be separated from the main channel. The channel can configure to run on subset of Hyperledger Fabric member nodes. This will physically separate private data from other member nodes and allows strong privacy properties.

Latest version of Fabric introduced end to end encryption. This allows to encrypt private data and persist it on shared blockchain The key is kept in the client and sent to server when data is written to ledger. New version supports building complex queries on encrypted data by providing the encryption key as parameter.

4.2.5 Migration

In **conventional** systems migrating data must executed using data migration patterns. There is a pattern for doing migrations in continuous delivery called Evolutionary Database Design and Expand and Contract. Short description of the steps:

1. Deploy new code and migrate all data. This is simple in SQL where you can iterate over the entire table using single update clause, but not very suitable in NoSQL since there are no transactions and you have to iterate over documents one by one. Thou NoSQL databases do not have strict structure, but it depends on the programming language that is used, that applies strictness to data store.
2. Deploy new code and include the update into the software code. Code can be written so it is backwards and forwards compatible. Data is migrated during read operation and can support multiple old versions.

In the first case, conventional relational databases use SQL scripts or special tools like Flyway or Liquibase. The latter allows the migration scripts to be written without using specific database syntax. Tools convert the configuration to a specific database SQL and execute them. NoSQL are considered schemaless and support migrating data using the second approach.

During the life-cycle of software, updating the platform software is required. This is driven by bug fixes and new features. During the upgrade, data must be migrated to the new version. This can happen automatically by database engine converting the data to new version or by using export import process. Latter requires of exporting a data dump and importing it to the new version.

There is no tool for data migration for **Hyperledger Fabric** chaincode. Although the world state is kept in a separate database RocksDB or CaouchDB, it is not possible to use the data migration tools meant for those specific databases. The peers synchronize the world state from their ledgers. Ledgers use custom data structures and are not modular. The tools must be incorporated to chaincode.

There is no documentation how to migrate data during smart contract updates. Thus it must be done using conventional data migration patterns. Short description of the steps:

1. Deploy new code and migrate all data
2. Deploy new code and include the update in read operation

Init method in smart contract is executed during the upgrade stage. Although there are no explicit documentation, it hinted in source code comments that this could be used to migrate the ledger to new.

Installing new version of chaincode is manual labor and needs to be executed on all endorsing nodes. The new version will be in use when all endorsing peers get the new version. If any of the peers do not have the latest version then the endorsement will fail. After new version of chaincode is deployed, the previous one is not removed and is still executable. The removal operation must be done manually on all endorsing peers as there is no automatic way.

When researching how upgrade from Hyperledger Fabric version 0.6 to 1.0, we failed to find any data migration guides. Version 1.0 added support of CouchDB and we cannot find any migration guide from LevelDB. This makes us question if the blockchain data migration is at all possible.

Adding a new organization to Fabric network consists of updating the configuration of the network topology and client software configuration. Trust for new servers must be added to MSP. Orderers configuration has to updated to allow correct ordering of the blocks coming from new member.

4.2.5.1 Continuous integration and delivery

Continuous integration is considered to be a best practice and it purpose to integrate the code changes as frequently as possible. This enables frequent automated testing and automated delivery of software. The keyword here is automatic. Manual work can be error prone and can cause downtime for the service.

Conventional systems have a separate build server that creates versioned software packages and pushes them to artifact repository. Using artifact enables the automated delivery of software to live systems.

In **Hyperledger Fabric** the build is executed by internal commands. Successful build will create a versioned Docker image and this pushed to a registry. Deployment of new smart contract version includes the consent of all parties. One party alone cannot deploy a new smart contract. Deploying a new version of smart contract includes signing the deployment by administrators in all of the organizations. This must be executed manually by the administrators using command line interface of the peer. Steps includes package, install, instantiate and upgrade.

Purpose of packaging is to create a deployment that all parties agree upon. This requires all endorsing parties to sign the deployment in sequential order. Result is called SignedChaincodeDeploymentSpec and it includes:

1. Name of the chaincode (this has to be same during the life of smart contract)
2. Version of the chaincode
3. Source code of the deployment
4. List of owners of the chaincode (who are the owners, will sign and deploy the chaincode)
5. An instantiation policy for the chaincode (used when transactions are endorsed by peers)

One of the parties must initially create a package that other will sign in sequential order.

After signature is provided by all organizations, the chaincode is installed to all endorsing peers. This means that separate administrators for all organizations will manually install the signed package to required peers.

After the binary package has been installed, it must be binded to a channel (blockchain). One smart contract can be used on one or multiple channels. This step is mandatory only in first deployment of chaincode and is called instantiation.

After a new version of chaincode is delivered to all endorsing peers, upgrade step can be used to activate the smart contract.

4.2.6 Development support

Conventional systems have online documentation that describe the architectural aspects of the platform. These include information how a system should be built, the properties of the system. For example how to achieve ACID or CAP properties in live systems. There is online documents and example for developers and if the documentation is vague, forums can be used.

Hyperledger Fabric is an open-source project documentation is seems to be upheld by community[1]. There is online documentation[2].. There are no forums or other kind of interactive support.

[1] https://www.hyperledger.org/community

[2] http://hyperledger-fabric.readthedocs.io/en/release-1.0/

5 Evaluation of results

In the following chapter, the results of comparison are presented and a discussion of benefits and disadvantages is conducted.

5.1 Executing nondeterministic functions

In conventional systems, we can run non-deterministic functions, for example getting the current timestamp. This is not the case for smart contracts. All the functions must return a exact value in short time span. Otherwise, some of the peers write-set will differ from others and the transaction will never be committed. We have shown in use cases chapter that using conventional programming languages makes it easy to find alternative solutions to nondeterministic operations. In bitcoin and in Ethereum all the solutions would not be possible or extremely time consuming. Compared to conventional solution designing alternative solutions is time-consuming since some sort of validation oracle must be used.

5.2 Atomicity, consistency, isolation and durability

Hyperledger Fabric supports ACID properties on transactions. There are two exceptions. One of them is the complex query method that is a key feature and separates Hyperledger Fabric from other blockchain applications. This had a side effect for the parking application. Concurrent invocations could create duplicate record. For example, we have a contract function that books a free slot for a parking spot. During the operation, the availability of the spot is verified. This validation uses the complex query method to search bookings for given time period and parking spot. Because the complex query method allows phantom reads, concurrent execution does not see the other one and the Fabric will allow creating a double booking of parking spot. The problem is also evident in other method, for reading history of a key, and has the same effect on transactions that write state to ledger.

To mend this issue, complex query results should be added to the read-set. If transaction depends on multiple values in the database then there is no other way of guaranteeing that transaction is valid. The functionality is already implemented on other read operations. Chaincode has support of getting keys for the values returned by complex query.

5.2.1 Transaction validation

There are no built in data constraints in Hyperledger Fabric data store, like there are in relational databases. There are no built in validation framework that could be used as Java Validation API. The Smart contract can be written in conventional programming languages like Go, Node.js and Java. This enables us to use same data validation libraries that are used in a conventional solution.

There is no support for distributed transactions. Considering that one of the key features of Fabric is to provide privacy using physically separated channels, the lack of support for cross-chain transaction is a surprise. That means the client must commit to two or more separate channels. Let us consider a contract where the price of a contract is kept in a separate private channel. What will happen to the contract, if setting the price transaction fails? Data will be in an inconsistent state. There is no rollback functionality built in for already

committed transaction. We have executed a contract, but there is no price. This issue enforces us to keep the pricing information in the same channel and use other techniques to provide privacy. The only way is to use client side data encryption.

Although distributed transactions are not supported, we were testing the SDK to see if a two-phase commit could be implemented using client SDK. The simulation for transactions would be a good candidate for two-phase commit implementation. First phase would send the transaction to the endorsing peer. This happens for all channels that are part of the distributed transaction. On successful endorsement of all transactions, the second phase would consist of sending all it to orderer. The transaction can still fail after sending to orderer. This can happen when peer applies the transactions to the ledger. Transaction will be rejected, if it changes a value that some other transaction in a block had already changed. This approach was not giving us the results we were expecting.

Software patterns, like Try-Cancel/Confirm [13], can still be implemented in the client, but would require that transaction logic be kept outside of smart contract. The distributed transaction logic should be inside the smart contract and enabling atomic and durable transactions.

Using validation and business rules in smart contract that provides strong consistency enables strong trust toward the correctness of data and the entire system.

5.3 Scalability

Hyperledger Fabric was built with scalability in mind. This was the driving force to change the conventional blockchain architecture execute-verify to execute-order-validate [2].

Having replicating copies of peer enable horizontal scaling for read operations of large scale.

Fabric is far from the transaction throughput that conventional systems have, but it is still big leap from Ethereum and Bitcoin transaction throughput of 3-20 transactions per second [4]. Hyperledger Fabric version 0.6 has been measured by Fujitsu Laboratories Ltd. to reach 1350 transactions per seconds using four servers [14]. IBM tested version 1.0 and reached 3500 transactions per seconds [2]. According to study, VISA is able to execute on average 2000 payment transactions per second [4].

5.3.1 Consistency, availability and partition tolerance

Hyperledger Fabric uses strong consistency leaving the system implementer to decide between partition tolerance or availability. These limitations to the system must be considered before choosing this as platform for a software. Documents do not cover the possibility that one of the organizations network is not accessible for longer period of time. It is not clear if that means that hard fork could happen in those two separated infrastructures.

5.4 Security

The Hyperledger Fabric platform is built on solid framework with security on mind. The client software and the user has to authenticate for each access of the system. Peers have to authenticate themselves to be part of the network. No authorized server can access the network even if attacker can take hold of some other server in network. It just does not have the credentials needed to attack Fabric network.

Although Hyperledger Fabric does not support conventional authentication schemas, they can be combined in the client. Any authentication provider can be used if it results in getting the username. Using this, the wallet can be accessed and for the Fabric point of view, the user is authenticated.

Fabric introduces attributes in the latest version. They can be used to provide permission based authorization. That is not enough; keeping track of which permission is enabled per group of people is hard to manage. Roles must be made available., Roles aggregate permissions for user group. For example, regulators have specific set permissions and thus regulator role. Adding new regulator would mean that assigning the new user regulator role would give all necessary permissions with one operation.

Hyperledger Fabric has built in support for confidentiality. This allows the member to keep it confidential data physically separated. This is not possible with conventional systems, where the third party will always have access to your data. Even if encryption is used for the entire database.

The separated channels in Hyperledger Fabric can be used for granting access for a regulator. Thus making all data available in one place and providing transparency.

Until distributed transactions are enabled for this platform, I do not see a way to guarantee public and private data consistency. I consider this to be a priority since it is claimed to be one of the most desirable properties of this product.

Hyperledger Fabric can use attribute encryption on the main channel. The key is kept in the client. This means that the data is encrypted and written to ledger by a peer running on the members own infrastructure. This data can also be used in queries. These queries do not leave the members infrastructure.

We do not know any NoSQL or relational database that can query data that uses end user encryption. Encrypting the data on platform still enables access by the third party. The encryption keys are available on the server and used when the server starts. High confidentiality guarantees are the most strongest part of this platform.

5.5 Migration

Conventional relational databases use separated scripts to migrate the data to next version. We propose a separate chaincode for doing upgrades. It should be deployed before and executed by administrators before the business value chaincode is made available. Chaincode supports this by using three-step deployment: install, upgrade and initialize.

During building the prototype, we used conventional NoSQL patterns to migrate the data between versions. Similarities to the conventional development will make this easy for existing developers to learn. We found that blockchain technology gives advantage of implicit auditing of the migration steps. In case of software error, the previous state can be queried from the immutable history.

We could not find any documentation how to migrate ledger data and world state from version 0.6 to 1.0. We concluded that, considering the change of architecture, this means it is not possible. This is very dangerous for the future of this platform. This would mean that any created software system that uses Hyperledger Fabric would be forever tied to specific version. Eventually the support for that version would be dropped and no bugs would be fixed. This is not acceptable and there should always be a way to migrate old data to new version of Fabric.

In a conventional solution, all software and infrastructure belongs to the third party. This makes **adding a new member** to existing solution easy. It could mean even that it is automatic, provided by the software.

This is not the case for Hyperledger Fabric. There is a long and technically difficult tutorial about adding a new organization to the channel, it consists of extracting the internal state of the peers using command line, extracting information in Protobuf format, converting it JSON , manually adding new organization to JSON file and then converting it back and updating the configuration. There really should be a simple command or a tool to make this easy. This is a limiting factor for future projects.

Documentation does not specify how the chaincode should keep up during these operations. We tested adding new organization into policy consisting of n mandatory peers. This was the hardest scenario we could think of.

Adding additional mandatory organization to the chain. All APIs and applications using this particular chain must change their software configuration (may include changing the software). For that migration to work seamlessly two step migration is proposed: Migration from initial policy

$$(X_1 \text{ AND } X_2 \text{ AND } \dots \text{ Xn})$$

to new policy

$$(X_1 \text{ AND } X_2 \text{ AND } \dots \text{ AND } X_n \text{ AND } X_{n+1})$$

needs to be performed

1. Inetmediate policy is implemented for migrating the chain to new policy. Adding one more organization and declaring it as optional

   $(X_1 \text{ AND } X_2 \text{ AND } \dots \text{ AND } X_n)$ OR $(X_1 \text{ AND } X_2 \text{ AND } \dots \text{ AND } X_n \text{ AND } X_{n+1})$.

   This enables organizations to test their code and gives time to migrate software to include the new soon to be mandatory policy.
2. Migration to final policy $(X_1 \text{ AND } X_2 \text{ AND } \dots \text{ AND } X_n \text{ AND } X_{n+1})$

Removing additional mandatory organization to the chain. All APIs and applications using this particular chain must change their software configuration (may include changinging the software). For that migration to work seamlessly two step migration is proposed: Migration from initial policy

$$(X_1 \text{ AND } X_2 \text{ AND } \dots \text{ AND } X_{n-1} \text{ AND } X_n)$$

to new policy

$$(X_1 \text{ AND } X_2 \text{ AND } \dots \text{ AND } X_{n-1})$$

needs to be performed

1. Intermediate policy is implemented for migrating the chain to new policy. Adding one more organization and declaring it as optional

   (X_1 AND X_2 AND … AND X_{n-1} AND X_n) OR (X_1 AND X_2 AND … AND X_{n-1}).

   This enables organizations to test their code and gives time to migrate software the removal of mandatory policy. This state is already acceptable as the final state and the third one a cleanup step.
2. Migration to final policy (X_1 AND X_2 AND … AND X_{n-1}).

5.5.1 Continuous integration and deployment

Deployment procedures that have been documented by Hyperledger Fabric are manual and time consuming. Considering that the deployment is file based and requires the signatures in sequential order and manual sending of signed file, we consider that it not a possible to use in automated continuous integration. The effects could be minimized by using file sharing services or even Git to orchestrate the sharing of the SignedChaincodeDeploymentSpec file. Nevertheless, it does not seem to be suit the needs of automated deployment that continuous integration requires.

We propose using an existing technology to create consensus on source and binary files – it is called smart contract multisignature. This can be done using chaincode. To make this work, we can use already suggested patterns for including big binary files to blockchain. This consist of creating a hash of the file(s) and committing the hash to blockchain. All parties can automatically verify that their own built binary hash matches the one in contract.

Proof of correct source code can be the commit hash of distributed version control system. For example, Git supports the model of multiple remotes. Every organization can have a separate version control system and organizations can pull changes from each other. The commits hashes are unique and shared between organizations. Another option is to calculate Merkle tree from all source files and use the root hash. If any source file differs, then the root has will differ.

The procedure that Hyperledger Fabric has introduced is not bad, it just not suitable for continuous automated deployments and feel out of sync from the rest of the Hyperledger Fabric approach. We propose that the signing functionality would be built as plug-and-play, to match the rest of the platform.

5.5.2 Development support

Documentation covers the basic terminology and how to set up development instance. When trying to get a detailed architectural view of the system and capabilities, then the document lacks details. We had to search multiple locations before we got answers to topics discussed in this thesis. There is no forums provided where a discussion could be started. For example, there is no detailed information how orders guarantee that there is no fork in blockchain if network of the one of the members goes down. Details if the orderer hangs or returns error. These are difficult properties to test without good knowledge how the system was meant to work. We consider this to be a serious disadvantage to conventional.

6 Conclusions

To enable software platform to be used without a third trusted party, one of the possibilities is to use blockchain and smart contracts. Open-source Hyperledger Fabric is the latest modular blockchain based system that uses conventional programming languages for smart contracts. This opens up vast possibilities for using it product centric enterprise systems and bringing blockchain technology to the masses.

We wanted to find out how it compared to a conventional solution. We created a proto-type of a parking application that had all of the properties of a connectional solution. We implemented user stories to study the platform. We created a comparison of a traditional solution to the Hyperledger Fabric and discussed the benefits and disadvantages of the platform.

Hyperledger Fabric has clear benefits enabling trust between parties, providing strong confidentiality for data and communication. It is built to be secure and have strong consistency by design. It enables to remove a trusted third party and be the owners of their own data.

Clear disadvantages are lack of distributed cross-chain transactions, support and documentation and inability to support today's fast delivery pace. Most of all, missing process to migrate data from older version to new version and difficulties of adding and removing new organizations from network. These problems are all mendable and allow never versions to exclude disadvantages.

For the future a software patterns could be studied, that describe non-deterministic functions in conventional solutions and how provide specific patters to be used in smart con-tracts. When the distributed transactions feature is enabled on newer version of Hyperledger Fabric, the study of supporting more complex transaction scenarios could be executed.

7 References

[1] N. Atzei, M. Bartoletti and T. Cimoli, "A survey of attacks on Ethereum smart contracts," 2016.

[2] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen and M. Sethi, "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," pp. 1-15, 2018.

[3] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[4] X. Xu, C. Pautasso, L. Zhu, V. Gramoli and A. Ponomarev, "The Blockchain as a Software Connector," pp. 182-191, 2016.

[5] R. Maull, . P. Godsiff, C. Mulligan, A. Brown and B. Kewell, "Distributed ledger technology: Applications and implications," 2017.

[6] K. Delmolino, M. Arnett, A. Kosba, A. Miller and E. Shi, "Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab," 2016.

[7] "Hyperledger Fabric licence," 2017. [Online]. Available: https://github.com/hyperledger/fabric.

[8] И. Ванков, "How exactly Hyperledger Fabric works. Basic workflow of transaction endorsement," 4 09 2017. [Online]. Available: https://www.youtube.com/watch?v=2_RgCfjunEU. [Accessed 21 12 2017].

[9] J. Mattila, T. Seppäla and J. Holmström, "Product-centric Information Management: A Case Study of a Shared Platform with Blockchain Technology," 2016.

[10] G. Niemeyer, "geohash.org," 02 2008. [Online]. Available: http://geohash.org/. [Accessed 01 05 2018].

[11] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," *ACM SIGACT News*, vol. 33, no. 2, p. 51, 2002.

[12] Open Group, "Distributed TP: The XA Specification," 1992.

[13] G. Pardon and C. Pautasso, Towards Distributed Atomic Transactions over RESTful Services, 2011.

[14] Fujitsu Laboratories Ltd. , "Fujitsu Speeds Up Transaction Processing on the Blockchain," Fujitsu Laboratories Ltd., 31 July 2017. [Online]. Available: http://www.fujitsu.com/global/about/resources/news/press-releases/2017/0731-01.html. [Accessed 21 12 2017].

[15] "Hyperledger Fabric documentation," 2017. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release/blockchain.html.

[16] D. Magazzeni, P. McBurney and W. Nash, "Validation and Verification of Smart Contracts: A Research Agenda," *COMPUTER*, no. september, pp. 50-57, 2017.

[17] A. Bieniusa, M. Zawirski , . N. Preguiça, M. Shapiro , . C. Baquero, . V. Balegas and . S. Duarte , "An optimized conflict-free replicated set," 2012.

Appendix

I. Glossary

II. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Sven Mitt**,

(*author's name*)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Blockchain Application - Case Study on Hyperledger Fabric,

(title of thesis)

supervised by Luciano García-Bañuelos, Fredrik Milani,

(supervisor's name)

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **24.05.2018**