UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Olev Abel

Case Study: Analysing Parking Solution using Corda DLT Service

Master thesis (30 ECTS)

Supervisors: Luciano Garcia Banuelos, PhD
Fredrik Payman Milani, PhD

Tartu 2018

Case Study: Analysing Parking Solution using Corda DLT Service

Abstract:

Since the proposal of public ledger system, Blockchain, back in 2008, there has been rapid development of ledger systems.

Original Blockchain, that worked inside Bitcoin cryptocurrency platform, have been moderated by several different groups in order to make it usable in various environments. Soon after the release of Blockchain as a part of cryptocurrency mining process, people realized, that Blockchain's potential is much bigger. With different researches and implementation to support other functions than cryptocurrency, researchers found out that the architectural specification of Blockchain, namely that every transaction in the system is public, is not suitable for various real life usecases, like medical data or stocks.

Then distributed ledger technology(DLT) was introduced. This technology made private transactions, without mediator, possible. This meant that vast potential, that public ledger system had, was now possible to brought into real life usecases, without sacrificing the immutability property of public ledger. Although theoretical background for DLTs has gone a long way there are only few studies on the analyses of DLT in everyday applications. This thesis provides overview of one particular DLT system named Corda. In addition, thesis analyses architectural differences between application using DLT and traditional database type approach. The analyses cover fields like, programming paradigm, architectural design, functionality and usability of the two approaches. As an outcome of this thesis, two different case study applications are analysed in depth. In addition, their key differences are brought out and reasoned about. Furthermore, the discussion of benefits and drawbacks of each key aspect is brought out.

Keywords:

Blockchain, ledger, DLT, Corda

CERCS: P175

Juhtumiuuring: Parkimislahenduse analüüs, kasutades jagatud andmebaasi teenus Corda.

Lühikokkuvõte:

Blockchaini avaldamisest saadik, aastast 2008, on avalike jagatud andmebaaside kasutamine kogunud järjest rohkem populaarsust. Algselt krüptoraha platvormis Bitcoin kasutusel olnud Blockchaini on osapoolte poolt muudetud, et uurida selle kasutust eri valdkondades. Kiiresti pärast Bitcoin'i avaldamist saadi aru, et sellel on potentsiaali rohkemaks kui ainult krüptorahaks. Erinevate uurimisrühmade ja arenduste tulemusena jõuti välja tõdemuseni, et Blockchain oma algsel, avalikul kujul paljudesse valdkondadesse ei sobi. Näiteks ei ole see oma algsel kujul kasutatav meditsiiniandmete või fintantstehingute juures, kus privaatsus on kriitilise tähtsusega. Nii arendati välja privaatne jagatud andmebaaside tehnoloogia. Selle tehnoloogia suurim erinevus standardsest Blockchainist on, et transaktsioonid osapoolte vahel ei ole avalikud, vaid need jagatakse ainult osapoolte vahel. Seetõttu sai nüüd võimalikuks selliste valdkondade, nagu meditsiin, rahandus ja sõjandus, areng jagatud andmebaaside suunas. Kuigi teoreetilised alused ja kontseptsioon on privaatsetel jagatud andmebaasidel üsna lai, on nende rakendamine igapäevaelus hetkel pinnapealne. Käesolev lõputöö annab ülevaate ühest privaatselt jagatud andmebaasi teenusest Corda'st. Lisaks sellele analüüsitakse lõputöös privaatse jagatud andmebaasi ja hetkel laialt kasutusel oleva traditsioonilise andmebaasi tehnoloogiate erinevusi. Võrreldakse paradigmasid, tarkvara arhitektuuri, funktsionaalsusi ja kasutatavust erinevatest aspektidest. Lõputöö tulemusena valmis kaht eri paradigmat kasutava rakenduse arhitektuuri ja funktsionaalsuste analüüs. Toodi välja nende erinevuste, tugevuste ja nõrkuste loetelu ning kirjeldused.

Võtmesõnad:

Blockchain, ledger, DLT, Corda

CERCS: P175

Table of Contents

1 Introduction

Distributed ledger technology (DLT) has been around since 2008, when a scientist or a group of scientists named Satoshi Nakamoto introduced Bitcoin. Bitcoin was introduced as a peer-to-peer electronic cash system, that required no mediator between two parties exchanging cash for goods.[1]

Since the proposal of the DLT, there has been a swift development of the systems that use this technology. Recently there has been a separation in the DLT. Traditional distributed ledger systems are built on top of the publicly distributed ledgers, meaning all of the transactions are broadcasted onto the ledger. New form on distributed ledger systems are also being developed and verified for different use cases. Those are the private distributed ledger systems. Next paragraph will explain those two types of ledgers in more detail.

Recent developments in public ledger systems have led into discussion about potential uses of those systems in more ways than just cryptocurrency. The key concept of public ledger systems is to allow different parties, without some mediator service, to make agreements and transactions that cannot be altered. Although public ledger systems allow participants to form different kind of transactions, the system itself is public, meaning everybody can see every transaction made in the ledger. In other words, once the transaction is verified it is broadcasted to all of the nodes in the ledger. To cryptocurrency it is necessary, because without transparent, publicly accessible knowledge base, the system is not much different than a regular bank. That brings us to crucial drawback of those systems. Everybody in the systems can learn any transactions ever made in the system. If talking about agreements between different parties, the discretion is often desired for. If one wins in a lottery, then he/she wants to stay anonymous, which is impossible with public ledger systems, since all of the transactions are verified by other participants. Same goes with sensible data like medical records, which should be visible to only patient and doctor. This, often underrated drawback of the public ledger systems, made way to another form of ledgers.

Recently a new form of ledgers was introduced. Private distributed ledger technology brings a solution to the privacy issue in public ledger systems.

In private distributed ledger systems data is shared on a need-to-know basis, meaning that only parties of a certain transaction see the details of the transactions and that this transaction ever took place. Other participants of the ledger do not know anything about the transaction between those parties. This will add the much-needed privacy function to a system of no

mediator. All of the core concepts of public ledger systems will remain, like the most important, transaction immutability. Since the evolution of the private distributed ledger services began only recently, there are few paper on how they work and what are the benefits and drawbacks of such services. Because the development of private ledger services is relatively young, there are many different forms of private DLT services. This thesis analyses one of them, named Corda.

Corda was developed to be used in financial institutions and designed for the field of finance. Author of the thesis believes that it can be used in other fields as well and as a part of this thesis, a case study of parking solution application will be conducted to verify, if in fact, Corda can be used inside case study application.[2]

Thesis shares some light on the paradigm and architectural concepts of Corda - private distributed ledger technology service. Moreover, as a part of the thesis, two applications are analysed to showcase technical, functional and usability differences between application that uses private distributed ledger service and the one which relies on traditional database type of implementation. Since distributed ledger service differs from traditional centralised approach in many aspects, this thesis is trying to find those key differences. Thus, the research question of the thesis is the following:

1. What are the key differences between Corda DLT service paradigm and traditional database paradigm?
2. What are the benefits and drawbacks of Corda DLT and traditional database systems?

Outcome of this thesis is an in-depth analysis of two applications with different architectures, answering the above-mentioned question.

2 Background

This section will give an overview of history and development of ledger systems and how the Corda system is used in our case study application.

2.1 Blockchain

Blockchain originally block chain, is a concept proposed by Satoshi Nakamoto, in year 2008 as a part of the Bitcoin cryptocurrency system. It is a form of distributed ledger system. The systems main feature is that it allows mutually distrusting parties to make transactions. To make transactions there needs to be some change of equity or mediator, which in Bitcoin case is defined as coin. Coin consists of chained digital signatures. In Blockchain the transaction consists of a hash of previous transaction and the public key of next owner of the coin. Using this type of transaction, a payee can verify the signatures to ensure the chain of ownership. The problem of double-spending a coin, is resolved with publicly broadcasting the transaction to the network and let majority voting decide on the chronological order of the transactions. Those transactions must be immutable to prevent forgery or withdrawal of the transaction. Since parties are mutually distrusting there needs to be a mechanism to verify the transaction. In Blockchain this is done via public verification. The underlying idea is that once two parties have agreed to make the transaction, they sign the transaction then it is hashed and publicly broadcasted to the ledger. To verify a transaction, nodes must perform proof-of-work, which involves computationally expensive operations to find required transaction value. Since proof-of-work involves hashing using publicly known hashing algorithm, in Blockchain namely SHA-256, the validation of one's proof-of-work is only one hashing. This means that once a proof-of-work is done, it is broadcasted to the network. Acceptance of proof-of-work is straight forward. Upon receiving proof-of-work, node will verify it and if it is accepted will add it to his block and begin to calculate next proof-of-work. Using majority voting the transaction is either accepted or rejected.[1], [3]

Public broadcasting serves multiple vital purpose here. Firstly, it will prevent coin to be double-spent, meaning there is always one chronological order of the transactions and same transaction cannot occur multiple times. This was previously done by a trusted party known as mint, which means that only mint knows all the transactions, leading the system to be basically the same as traditional bank, with one centralized authority of proof. Secondly the public broadcasting serves a purpose of unforgeability of the transaction. Since the mutation

of one transaction in the chain will affect the hashes of all of the transactions that are coming after the modified one, the attacker must redo all proof-of-work that comes after the infested transaction. To make matters worse, the attacker needs to do it on majority of the nodes simultaneously, to not be rejected due to majority voting.

As an extra security mechanism, Bitcoin has built in parameter to determine the proof-of-work difficulty. Since the computational power and runtime is increasing on a day-to-day basis, the ledger system needs a way to protect itself from having one node or group of nodes reaching and preserving the majority of computation power. The achieve that the proof-of-works difficulty is determined by the average number of block per hour, meaning that if blocks are calculated faster than threshold average the proof-of-work difficulty will increase. This ensures that the more powerful the nodes get the more difficult the proof-of-work will get, remaining the average blocks added to chain per hour within the certain threshold.

Since Blockchain is relaying on majority voting, it is vulnerable to the attacks, where attacker gets the hold of majority of the nodes. In early phase of ledger, where there is only few nodes, this type of attack is very likely to happen, but as the ledger gets bigger and more powerful this type of attack becomes impractical. In Bitcoin, as of 21. October 2017 at 18:20 there are 17 834 066 nodes, which makes this type of attack impractical.[4]

Although Blockchain is originally used as part of the crypto currency and exchanging it, the potential of it has caught much attention world-wide. There are many different ledger systems that are using Blockchain as the underlying technology and adding some advanced feature to deal with real-life problems other than currency. But unlike the simple transaction that is made via interaction with the ledger, researchers found a way to automate and add functionality to those simple transactions.

2.2 Distributed Ledger Technology with public ledger using Ethereum as example

Distributed ledger technology (herewith may be referred as DLT) is a common name for all the distributed ledgers, that are trying to solve the consensus replication, sharing and synchronisation of digital data across multiple sites, countries of institutions, without the centralised database or central administration.[5] In this paper those are referred as the alternatives to original Blockchain, meaning, if talking about distributed ledger technology or distributed ledgers original Blockchain is excluded as it was discussed in detail above.

Distributed ledger technology marks another era of ledgers without centralization. In addition to the simple transaction, which is started by human interacting the system, DLT provides different ways to add code to the transaction making it self-performing, when certain requirements are met. The most used DLT system today is Ethereum. It features a Smart Contract notion. Smart Contract is an Ethereum way to add functionalities to regular Blockchain transaction. For example, using Smart Contracts, one can automate the transaction of money to other party from a moment when money is transferred to his account. This means that transactions can be much more powerful. Because of their nature of customization, Smart Contracts can be used to transfer not just crypto currency or money, but an asset, or a membership or virtually anything that can be represented as a form of token. Since the token is an abstraction for the real-world equity, the exchange of it can be combined into really sophisticated processes. Simple example being that the token does not change its ownership until a certain requirement is filled. Whether it is a transaction of money, a signing of physical document or signing a legally binding digital document different from the Smart contract. Using the ability to program the Smart Contract gives much more opportunities to form transaction than a simple one way transaction.[6]

Since every node of the system can make up his own token to use the Smart Contracts are custom as well. Ethereum uses a way to make the users of the ledger pay for the service. This is done using *gas*. Gas is a unit of measure to set a price on the transaction being forwarded to a system. So, for every transaction made and published, a creator of the transaction needs to pay Ether to the system in order to get the transaction to the other party. Since transactions can be very computationally expensive, each of the transaction has its own price determined by the ledger. The more complex the transaction is the more it will cost to publish. There is also an upper bound mechanism, to avoid programmatic errors to deplete ones Ether wallet. As of 22. October 2017 the Ether stands at 295$ per Ether. Of course, the fee of a transaction is much lower than 1 Ether, to encourage users to publish transaction, to have miners to mine them. The Smart Contracts lead the way to a fundamental change on Blockchain from being solely used for cryptocurrency transactions. The vast potential of Smart Contracts also provides more ways to attack the system. Using the weaknesses inside the contracts, one can alter the additional functionally of the transaction, but not the transaction itself. Nevertheless, the introduction of Smart Contract provides a new layer for system to deal with and secure. Smart Contracts bring an extension to the transaction of the Blockchain, but Ethereum is not suitable for the fields, where privacy is

critical. Due to its nature of public broadcasting of the transactions, the privacy of the transaction is impossible.[6]

2.3 Corda

As the Blockchain and ledger systems in general evolved, the need for ledger that can operate on highly privatised fields grew. Fields like medicine and medical data, finance, defensive industry, etc. needed the same kind of ledger system to bring mutually distrusting parties into consensus, but with a subtle difference, that the transactions between parties need to remain private. Private in a sense that they are distributed on a need to know basis. This means that only the parties involved in the transaction has the copy of it and no third party can see the transaction. Distributed ledger technology spread into two different branches. One that is already discussed about – publicly distributed ledger systems and the other one that is the main motivation in our paper – privately distributed ledger systems. Privately distributed ledger systems need a way to bring different parties into consensus without publicly sharing the transaction. As the paper is about the case study or one particular privately distributed ledger system, Corda was chosen for the study.

Corda is a distributed ledger system developed and published by company R3. Corda was developed for financial services industry, but can be applied on other fields as well. The main goal for Corda is to provide frictionless financial agreements, without error, or unnecessary data duplication. Corda aim is to provide platform for legally well-regulated financial institutions. Since banks nowadays rely on many different platforms, data needs to be synced between them. This creates unnecessary duplicates and data parsing errors, resulting hours of manual work to verify the data. In addition since human is error-prone, there are still errors in the data resulting lost in revenue and friction in dealing with financial institutions.[7]

Corda is trying to solve the above-mentioned issues, by introducing privately distributed ledger system. It is different from Ethereum and traditional Blockchain in a way that transaction is never publicly broadcasted. Instead it is shared on a need to know basis with the appropriate parties. With this the state of the ledger needs to be kept somewhere.[7]

States are immutable objects that represent shared facts, at the specific point in time. This means that there needs to be a way to move from one state to another. This is done using transactions. Transactions can have input and output state, it must have signatures of appropriate parties and usually it also has referenced contract code, although since the nature

of automation, the contract code is linked with the state. Since the public broadcasting system does not exist, there needs to be a way to reach into consensus with transactions and also make sure that transactions are unique, meaning no other transactions have been previously triggered with exact same input and output.[8]

For transaction consensus, Corda uses contracts similar to the Ethereum Smart Contracts to add functionalities to transaction. The only difference is that a contract must always have certain parts in them. In Corda, every contract needs to extend an abstract Contract class in order for system interpret it as contract. As every created contract must be subclass of Contract, then every contract has verified method, which as stated by the name let parties to verify the transaction in which the specific contract is in. If transaction is successfully verified, then nothing is returned to indicate that verification does not mutate the transaction or contract code in any way. If transaction gets verification error, then exception is thrown to indicate that transaction has something invalid in it. In this exception developer can indicate of what went wrong by adding comments to exceptions.[7] Since every party involved in transaction can verify it themselves and states are immutable, then the consensus part of the transactions are covered. The uniqueness part cannot be solved within parties. This introduces a new concept in Corda – notary.[8]

Notary service is a "pluggable" service, meaning it is not the core part of Corda, but can be introduced by the developer of the system. Meaning the notary of really small systems, where there are minimal number of participants can be left out. This way system itself needs to prevent double-spending transaction. Notary can be virtually any independent service, as long as it can sign the transaction. The Notary will take the transaction to mark the time, when did it record the transaction and then signs it if it is unique one and returns error, if the transaction is about to be double-spent. The creators of Corda assume that the Notary is composed of multiple distrusting parties, that agrees on a consensus algorithm only. This algorithm is in no way limited as the ledger will know nothing about it. The notary service will be called, once transaction is finalized, in order to determine if the exact same transaction has occurred before. Note that instead of original Blockchain time-stamping, Corda uses Notary to conform chronological order of the transactions.[8]

2.4 Case Study

In case study a simple real-life use case of renting a parking space for a period of time is taken, to facilitate on the paradigm and software architecture of Corda versus traditional centralized database way.

To measure when to stop the analyses, the criteria of done is needed. This is presented as use case in section 2.6.

This thesis is an exploratory type, meaning the structure of the thesis is different than the traditional thesis. Since the service under discussion is rapidly developing, the exploratory type of the research is the best approach, because it has enough flexibility, to accommodate all the changes done to the service. Although the exploratory type provides the possibility of constant change, in this thesis the author has fixed the version of Corda to be v2.0.

2.5 Business model of case study

Since thesis is about the real-life case study about renting a parking space, business rules and model are needed to be set in order to create a context for the thesis and to better focus on the thesis research questions. Author has selected to introduce business model using Business Model Canvas [9]. As the business part is needed for creating a context, not for proving the case to be profitable, author will give a brief overview of the relevant parts of the business model.

Value proposition is a value that service or a potential business provides. It is important to notice that in the context of value proposition, the source code or asset or even a product being developed is not considered as value. Value is in form of benefit to an end user. If talking about the case study proposed in this thesis, there are many value propositions. As the business model is not the primary goal of this thesis, only some of the potential value propositions are brought out. Most relevant value propositions:

- Renter can rent out parking space. Value to a renter is that he/she can make a revenue on property that is currently unused.
- Tenant can find parking space to rent. Value to tenant is that he/she can find available parking space anywhere in the city, thus there is no need to use parking spaces or parking houses and take a cab to get to the destination.

- Tenant can remotely book and cancel booking to a specific parking space. Value to a tenant is that he/she can book in advance, meaning there is a guaranteed spot for him/her.

Customer segments are different groups of customers of the provided service or product. Customer segments usually have different expectations or needs towards the service. For the case study the author have picked two distinct customer segments.

First one is the individuals. This customer segments refers both to individual renter and individual tenant. As individual renter usually has one parking space then to simplify the business case author assumes that individual renter can manage only one parking space. As for individual tenant, he/she can rent upmost one parking space at any given moment. This is to clearly distinguish different pricing and use cases for individual users and companies.

Second segment is companies. As for companies, the amount of parking spaces is more than one. This means that companies can rent out or use one parking space, but they can have multiple as well. In addition to multiple parking spaces to rent out or to use as tenant, companies have an option to rent out parking spaces as bulk. This means that tenant rents one parking space inside the bulk, but it is not specified by any identifier, meaning it is like in super markets where customer can park to any available parking space.

Author chose those two customer segments, to support bulk renting and to showcase the real-world use case of multiple parking spaces.

Channel is considered as means of providing a service. As for the case study, since author is analysing the architecture, the channel is not set.

Most of the customer relationship is about how to acquire more customers and how to keep them, which is out of the scope for this case study.

Revenue streams is a vital building block for case study as it consists of payments, which is the main aspect of research considering security and usability of the two applications. Revenue streams are basically ways, that service make its revenue. Case study deals with

payments and for payments, the pricing model is needed. To mimic real-life use case author decided to make the following pricing plan:

- For individual renter, he/she can name the price for hour of his/her space.
- To individual renter service has two separate pricing plans similar to public transport one time ticket and monthly fee. Individual renter can either pay fixed % of each rent out to a service, or he/she can pay monthly fee, much like subscription. Therefore, monthly fee makes sense, if renter rents out frequently and one-time fee, when renter rents out maybe once or twice a month.
- For companies the pricing plan is individual for each of the company depending on amount of parking spaces and location of the parking space.

As for case study two distinct pricing plans for individuals are meant to be developed as it showcases how to implement different pricing plans on Corda. Also as a part of case study author is implementing a case when pricing changes whining the time of the day, meaning there are separate rates for day and night-time.

There are many key partners, but as case study focuses of minimal viable product (MVP) author considers key partner as payment mediation service. Since most of the services that uses credit card payment or any kind of payment, do not handle it themselves, due to legal reasons, payment mediation service is needed. Since credit card or any payment related data is considered as highly sensitive, ordinary services cannot store or process it. As a result, case study applications need to rely on one of those services, in order to support some form of payment (as case study focuses on traditional form of finance rather than cryptocurrency, if which case the mediation would be unnecessary). One of the most used service is Stripe. [10]

Key activities, key resources and cost structure are highly business-side parts of the Business Model Canvas and therefore are irrelevant to the case study, as they do not provide useful information for the analyses of the two systems in development. Thus, author considers them as out of the scope for this thesis.

2.6 Use Case

In this chapter, the author of the thesis is giving a detailed description of the use case of the case study, to make it easier to capture the context of the thesis.

The use case of the case study in question is to model a parking lot rental service. User of the application needs to be able to both rent out his/her parking space as well as rent the parking space from another user. In addition to that, user must be able to set the hourly rate for both day and night time, as well as the location and measurements of the specific parking space. One user many have multiple parking spaces, but as the tenants are meant to be persons rather than companies then one user can rent one parking space at any given time. Also, the system should not allow the tenant to re-rent the parking space he/she is currently renting. Only the true owner of the parking space should have the right to rent out parking space. What is more, the system needs to provide a maximum renting time, to avoid depleting users credit, if one forgets to finish the parking session.

User is asked to provide credit card information and also to load the credit to the application before he/she can use the application for rental purposes. This way the problems with funds are not handled in the rental flow, but rather before the rental flow. This makes the application less error-prone. Since the application involves sensitive credit card information, a third- party service for payments – Stripe [10] is used

3 Software architectures of Corda and central database applications

In this chapter, the author of the thesis is focusing on software architectures of both traditional centralised database application approach and private distributed ledger application approach. The author is bringing out the models for both architectures as well as practical implementation on private distributed ledger approach.

3.1 Architecture of central database application

In this section, an architecture of central database application is discussed. As basic use case of the case study is relatively simple both domain model and sequence diagrams are brought out. As central database application is industries traditional way, the author of the thesis is bringing out the architecture without the implementation of the application. Author is focusing the practical implementation to Corda application discussed in section 3.2

3.1.1 Centralised approach domain model

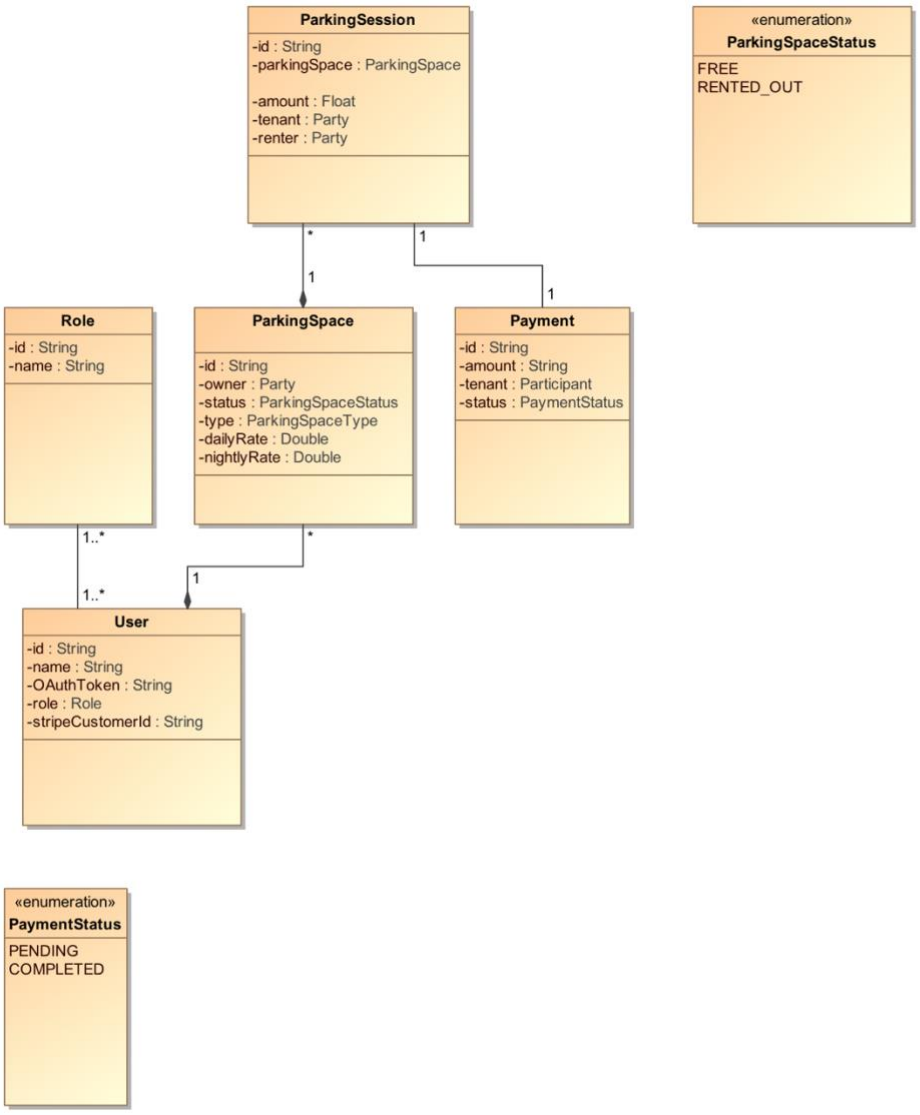


Figure 1 Domain model of centralised solution

For centralised solution user needs to register using third party for authentication. This means system gets only the OAuth 2.0 token from the authorization service and uses it to grant access to system resources. This way there is no need to store the sensitive information about users third party credentials, meaning if system developed by the author has weakness to obtain the OAuth2 token by dishonest party, the credentials for third party service remain a secret. Most frequently Facebook or Google is used as authorization service, for OAuth2 [11]. Concerning payments, since they involve credit card information, which is really sensitive then vast majority of today's applications are using payment services to handle the sensitive data for them. The author of the thesis chose to use Stripe payment service[10].

Stripe service needs info about the credit card and then it creates a Customer entity of its own. In order, not to insert card data every time user accesses the described application, a special token is used to get the information from the Stripe service and use it to make a payment. Since all of the users of the application needs to enter their credit card info in order to use the application, then it is guaranteed, that Stripe service has both renter and tenant credit card information to make payments.

As for parking space, it is modelled as one entity, which has an owner of type Party and status either free or rented out. This will simplify the search of the available parking spaces. As one user can owner several parking spaces, but there is no notion of co-owning a parking space, then the multiplicities between Party and ParkingSpace entities are 1…*. The two different hourly rates are introduced for simulation purposes and to test out the rate change during one session, to evaluate if this kind of process can be implemented on both centralised and decentralised application. Those rates cannot be changed during session. The only modification that can happen during session is the change of ParkingSpace status. This can happen on two occasions. Either tenant finishes the parking or the maximum allowed duration of the session elapses.

Regarding parking session, there exists a ParkingSession entity, which holds information about the current session, such as the start timestamp, tenant, parking space etc. Since the PakingSession is for specific ParkingSpace and obviously there can be multiple ParkingSessions for one ParkingSpace, then the multiplicity between ParkingSpace and ParkingSession is 1…*.

3.1.2 Parking space addition

To add a parking space into the system renter UI will invoke add parking space call to server. The server gets the parking space data from the renter UI and creates parking space object and saves it to database. Once saved server will respond to UI with newly created parking space UUID. Sequence diagram of adding a parking space to central database system is in the Appendix I

3.1.3 Parking session start and update

To start a parking session in central database application, tenant UI will invoke start parking session and sends parking space id on which he would like to start parking to server. Server receives the parking space id, get the parking space information from database and creates

new parking session. After creating new session, a task scheduler is used to make a task to be run, when maximum session time elapses. UUID is then assigned to the task. Then session is saved to database with task UUID and response of successful session start with session UUID is sent to tenant UI. Since central server is the single source of truth, there is no problem with time, as server dictates the actual time. Because of that, an update is not necessary, since server can save both start and end time of parking. Sequence diagram of parking session start and update is in the Appendix II

3.1.4 Finish parking session

In order to finish parking session one of two should happen. Either user finishes parking session from UI, or maximum session time elapses. If user finishes parking session, he will send a finishing call with the session UUID obtained from the parking session start to the server. Server will then cancel the scheduled task using the task UUID, to find which task to cancel. After cancelling the task, server will end the parking session and save the session data to database and marks the parking space as FREE again. Then server will invoke payment, by calculating the amount and sending the amount with stripeCustomerId to Stripe server, using Stripe Java library [12]. After getting response from Stripe server will mark payment as completed and notifies UI of successful ending of parking session. Sequence diagram of parking session finished by user is in the Appendix III

If time maximum parking session time elapses, then server will invoke similar finishing flow as described below, with an exception that parking session will be queried in the database by task UUID. Sequence diagram of parking session finished by maximum time elapsing is in the Appendix IV.

3.2 Architecture of Corda application

In this section, a distributed ledger approach architecture is discussed. As use case is relatively simple, the author of the thesis will bring out both domain model and sequence diagrams for basic use case. Since the basic use case as Corda application is also developed, code snippets to relevant parts are added as well. Payment service is replaced with Corda's Cash state to show how to use on-ledger credit. Author has used multiple example projects [13]–[15] provided by R3 itself to help developers to develop Corda applications. Author has modified them to make use in the case study at hand. The application developed by author to showcase the parking solution can be found in the Bitbucket repository: https://bitbucket.org/olevabel93/cordaparkingsolution/src/master/. The README.md gives

information on how to set up, run and test the application. Release is tagged as v1.3, since Bitbucket is not supporting separate releases.

3.2.1 Distributed ledger approach domain model

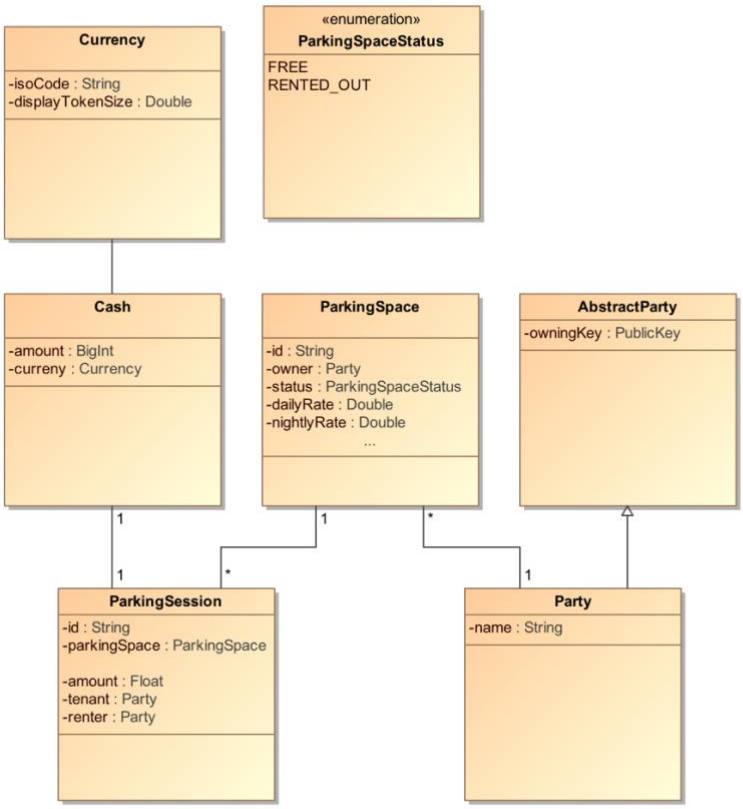


Figure 2 Domain model of distributed ledger solution

Compare to the centralised approach the first notable change is that there is no OAuth 2 token of the user. In fact, in Corda the node owner has a special type named Party, which extends the AbstractParty class. Later ones are used to keep the RSA public key. There is no notion of OAuth 2 token or any kind of credentials for user to enter, but a RSA public-private key pair, which is created at the initialisation of the node. This means that the security of the ledger is guaranteed by the RSA key pair strength. At the time of writing this thesis the minimum strength of the RSA key pair was 3072 bits. Having RSA key pair used to identify, verify and sign the transaction, means that the level of security of the distributed ledger approach is much higher than the centralised one. Of course, one can make up a system where centralised application uses the password as strong as RSA key with length

3072 bits, but it is in no way practical nor feasible. The problem is that this way the password would be 3072 bits long, which is 384 bytes, which in UTF-8 encoding takes 96 - 384 characters. Clearly this is too long for anybody to remember.

Another difference is that there is no start or end timestamp of ParkingSession. This is due to the time conception in Corda, which is discussed in detailed in section 3.2.4.

Since Corda is developed for financial services special state called Cash is built into the system. The author is using this to facilitate the on-ledger credit.

3.2.2 Parking space addition

To add a parking space to Corda ledger, renter UI will invoke a call to add parking space to renter node, by making sending address and daily and nightly rates. This is shown in Figure 3

```
@GET
@Path("create-parkingSpace")
fun createParkingSpace(@QueryParam(value = "address") address: String, @QueryParam(value =
"dailyRate") dailyRate: Double, @QueryParam(value = "nightlyRate") nightlyRate: Double): Response {
    val (status, message) = try {
        val flowHandle = rpcOps.startFlowDynamic(
                CreateParkingSpace::class.java,
                address,
                dailyRate,
                nightlyRate)
        val result = flowHandle.use { it.returnValue.getOrThrow() }
        CREATED to result.tx.outputs.single().data
    } catch (e: Exception) {
        BAD_REQUEST to e.message
    }

    return Response.status(status).entity(message).build()
}
```

Figure 3 Invoke parking space addition

Renter node will create, sign and then verify the transaction of the parking space. Then it stores the pending transaction to the local transaction storage. After saving the transaction to local storage, it will be sent to notary, for notarisation. Notary will verify the transaction data and its uniqueness. If transaction is valid, notary will sign it and sends it back to renter

node. Renter node will store the notarised transaction and broadcasts the parking space to all participants in the ledger. This is shown in Figure 4

```
class CreateParkingSpace(private val address: String, private val dayRate: Double, private val nightRate:
Double) : FlowLogic<SignedTransaction>() {
    @Suspendable
    override fun call(): SignedTransaction {
        val notary: Party = serviceHub.networkMapCache.notaryIdentities.first()

        val parkingSpace = ParkingSpace(owner = ourIdentity, dayRate = dayRate, nightRate = nightRate,
address = address)
        val createCommand = Command(ParkingSpaceContract.Commands.Create(),
listOf(ourIdentity.owningKey))
        val outputState = StateAndContract(parkingSpace, ParkingSpaceContract.CONTRACT_REF)

        val utx = TransactionBuilder(notary = notary).withItems(outputState, createCommand)
        val stx = serviceHub.signInitialTransaction(utx)
        val ftx = subFlow(FinalityFlow(stx))

        subFlow(BroadcastTransaction(ftx))

        return ftx
    }

class ParkingSpaceContract : Contract {
override fun verify(tx: LedgerTransaction) {
    val command = tx.commands.requireSingleCommand<Commands>()
    val setOfSigners = command.signers.toSet()
    when (command.value) {
        is Commands.Create -> verifyCreate(tx, setOfSigners)
        is Commands.SetFree -> verifySetFree(tx, setOfSigners)
        is Commands.SetRentedOut -> verifySetRentedOut(tx, setOfSigners)

        else -> throw IllegalArgumentException("Unrecognised command.")
    }
}
private fun verifyCreate(tx: LedgerTransaction, ofSigners: Set<PublicKey>) {
    "No input states should be consumed when creating parking space." using (tx.inputStates.size == 0)
    "One output state should be created when creating parking space" using (tx.outputStates.size == 1)
    val parkingSpace = tx.outputStates.single() as ParkingSpace
```

```
    "A newly created parking space should be available" using (parkingSpace.status ==
ParkingSpaceStatus.FREE)
    "A newly created parking space must be signed by owner only" using (ofSigners ==
keysFromParticipants(parkingSpace))
}
```

Figure 4 Parking space creation flow

Sequence diagram of parking space addition is in the Appendix V.

3.2.3 Parking session start

To start a parking session, a transaction is needed in order to begin what is known as flow in Corda.

Flow is a sequence of transactions that takes application from one state to the other. Flow can have sub flows. Transaction is created using smart contract and optionally one or many input states. As the parking session is about to start and there is no previous state, then parking session start transaction has no input states. UI will invoke the transaction by giving node an ID of the parking space. Tenant node uses the id to determine which other node is involved in the transaction. Once invoked, transaction needs to be signed by tenant. For signing a built-in method can be used. After signing a transaction tenant node needs to verify the smart contract in the transaction in order to guarantee that the transaction sent to counterparty for signature is a valid one. In case verification fails, an exception is thrown and the flow is halted.

After verification of the partially signed transaction it is stored to local transaction storage to make it repeatable in case of network failure. Only after local saving of transaction locally, transaction is sent to the renter, to get it signed. Once renter node receives the partially signed transaction, renter node will sign it and then verify the contract again using the same code as tenant. If transaction is found to be not valid an exception is thrown and the flow is halted.

Once the transaction is signed and verified by the renter node, it will be saved on renter's local transaction storage. After saving transaction locally, renter will reply to a tenant transaction call with the fully signed transaction.

Tenant will receive the fully signed transaction, saves it to the local transaction storage and passes it to notary service. Notary will both verify the contract code as well as uniqueness

of the transaction. If both criteria are met, notary will sign the transaction as well and sends it back to the tenant.

Tenant now have notarised transaction, which means this transaction is final and should be pushed to ledger. Tenant will save the transaction to local storage and in addition, saves any relevant state to local vault. Then it passes the notarised transaction to renter for saving transaction and state to renter's side as well. This process code is shown in Figure 5

```
@Suspendable
override fun call(): SignedTransaction {



    // Step 1. Initialisation.
    progressTracker.currentStep = INITIALISING
    val parkingSession = ParkingSession(Amount.zero(Currency.getInstance("USD")), renter, ourIdentity,
parkingSpaceReference)
    val ourSigningKey = parkingSession.tenant.owningKey
    val parkingSessionStartCommand = Command(ParkingSessionContract.Commands.Start(),
parkingSession.participants.map { it.owningKey })
    val parkingSessionStateAndContract = StateAndContract(parkingSession,
ParkingSessionContract.PARKING_SESSION_CONTRACT_ID)

    logger.debug("Init")

    // Step 2. Building.
    progressTracker.currentStep = BUILDING
    val utx = TransactionBuilder(notary = firstNotary).withItems(
            parkingSessionStateAndContract,
            parkingSessionStartCommand
    )
    utx.setTimeWindow(serviceHub.clock.instant(), 5.seconds)
    logger.debug("build")
    // Step 3. Sign the transaction.
    progressTracker.currentStep = SIGNING
    val ptx = serviceHub.signInitialTransaction(utx)
    logger.debug("sign")
    // Step 4. Get the counter-party signature.
```

```
    progressTracker.currentStep = COLLECTING
    val renterFlow = initiateFlow(renter)
    val stx = subFlow(CollectSignaturesFlow(
            ptx,
            setOf(renterFlow),
            listOf(ourSigningKey),
            COLLECTING.childProgressTracker())
    )


    // Step 5. Finalise the transaction.
    progressTracker.currentStep = FINALISING
    val ftx = subFlow(FinalityFlow(stx, FINALISING.childProgressTracker()))
    subFlow(ChangeParkingSpaceStatusRentedOut.Initiator(parkingSpaceReference))
    return ftx
}
private fun verifyStart(tx: LedgerTransaction, signers: Set<PublicKey>) = requireThat {
    "No states should be consumed when starting parkingSession." using (tx.inputStates.isEmpty())
    "Only one parkingSession state should be created when starting a parkingSession." using
(tx.outputStates.size == 1)
    val parkingSession = tx.outputsOfType<ParkingSession>().single()
    "A newly started parkingSession must have a positive amount." using (parkingSession.amount.quantity
>= 0)
    "The lender and tenant cannot be the same identity." using (parkingSession.tenant !=
parkingSession.renter)
    "Both lender and tenant together only must sign parkingSession settle transaction." using
            (signers == keysFromParticipants(parkingSession))
}
```

Figure 5 Start parking

Since Corda is a form of distributed system, the notion of time is different than in central database systems. For this reason, updating the parking session is discussed as separate flow in the next section. Sequence diagram of parking session start using Corda is in the Appendix VI and VIII

3.2.4 Concept of time in Corda, continues and scheduled events

Time is often the problematic part of large scale systems. Since the concept of time is thought out and it does not exist in the physical world, people have adopted different properties for the time, such as time zones, or time units. Because of those properties, it is

rather difficult to keep system in sync with all of the different parties being affected by those properties. Namely it is difficult to make such a system, that time related business requirements are fulfilled in correct way. On centralized systems, this can easily be achieved by letting the centralized server dictate the time and use UTC standards Zulu time, for data transfer from client to server and vice versa. Then client can easily convert the received Zulu time to any time zone needed. The key of this solution is that system is based on server's time and is depending only on this time. Same principle cannot be used in DTL systems, because as mentioned before, there is no single source of truth and thus, the whole system can be described as huge desynchronized clock. For the use case analysed in this thesis, there are number of business requirements, that involves time. For example, there needs to be a way to measure the time of one parking session and also, since there are different hourly rates depending on the time of the day, system also needs to be aware of that. Also, there is a safety mechanism for tenant, who forgets to end his/her parking. There is a limited amount of money one can spend on a single parking session. In order to calculate that, time is also needed.

During the first research session, several ideas were proposed to solve the desynchronized clock problem:

- First one was similar to the micropayment strategy. System creates all of the transactions at the invocation of the parking session. This means that for every parking session X / y transactions are created, where X is the maximum possible amount that can be spent on one parking session and y is the rate of the parking space per interval. Then when a tenant starts the parking, system obtains the signatures of both tenant and renter and it saves the tenants transactions into tenant's vault. Then when interval progresses the system will take the corresponding transaction from the tenant vault, gets the renters signature and notarises a transaction. When the tenant wishes to finish the parking session, he will invoke the finish process, which just takes the next uncommitted transaction from the tenant vault, gets renter signature, notarises the transaction and finishes the session. Now when the interval progresses to the already committed transaction, notary will reject it due to double spending and since the transactions are all linked together, then this transaction and all of the transaction coming after it is dismissed.

- Second idea to solve the desynchronized clock issue was to use a method that is familiar in banking. This meant that once the tenant is starting a parking session he/she is making a transaction, which has the maximum amount of time possible for one parking session. Now if tenant wishes to finish the session earlier than the maximum amount expires, he/she makes another transaction stating the early finish of the session. This transaction basically takes in previous transaction and also contains the amount to be paid. The difference of the previous transaction amount and the actual amount is paid back by the renter.
- Third idea was to solve the issue using Corda's Schedulable State [16]. With the schedulable state, the actual time is not relevant, meaning system only needs to know at which interval the next state should be invoked. This means that the desynchronized clock issue, becomes irrelevant, because we don't care about the actual UTC standard time, but rather an interval, which makes it much easier, since the interval counting is build-into the modern computer's hardware. With Schedulable State, it is possible to solve the flow the following way:
    - Tenant starts the parking session by invoking first transaction. This transaction contains the rate and the interval for the specific parking space.
    - Once this transaction is crated, inside it is the above mentioned Schedulable State, which invokes a new transaction periodically at the interval specified by the first transaction.
    - Since the next state takes the previous one as an input, it is easy to increase the amount to be paid for the parking session, by just adding the periodic rate to the previous amount and store this as the amount.
    - Now if a tenant wishes to finish the parking, he/she will invoke the finishing transaction and when this gets committed to the ledger, the system automatically knows that it contains a Scheduled state and will not try to invoke another one [17].

3.2.5 Parking session update

Updating parking session is done as follows. Initial parking session will produce a Schedulable State [17] that will have a time interval of one hour. Upon the interval elapse, a method nextScheduledActivity is called, which will create a new flow. In this flow, a previously produced state is taken as input, then parking fee amount is recalculated and new transaction is created with updated fee amount. After creating new transaction, the following

flow is the same as starting session. Transaction is signed and verified by tenant, then counterparty signature is asked. After getting signature the transaction is sent to notary for verification. After getting the notarised transaction from notary it is sent to renter as well. If another hour elapses the same update flow is invoked again by Schedulable State stored in tenant's vault. Updating continues until one of the following happens, either the maximum fee is met or user invokes finishing the parking session. This is show in Figure 6

```
@Suspendable
override fun call(): String {
    logger.debug("update called")
    progressTracker.currentStep = GENERATING_TRANSACTION
    val input = serviceHub.toStateAndRef<ParkingSession>(stateRef)
    val inputParkingSession = input.state.data
    val inputParkingSpace = getParkingSpace(inputParkingSession.parkingSpaceReference, serviceHub)
    val ourSigningKey = inputParkingSession.tenant.owningKey
    val amount =
input.state.data.amount.plus(calculateAmount(getRateBaseOnCurrentTime(inputParkingSpace.dayRate,
inputParkingSpace.nightRate)))
    val output = ParkingSession(amount, inputParkingSession.renter, inputParkingSession.tenant,
inputParkingSession.parkingSpaceReference)
    val updateCommand = Command(ParkingSessionContract.Commands.Update(),
input.state.data.participants.map { it.owningKey })
    val txBuilder = TransactionBuilder(serviceHub.networkMapCache.notaryIdentities.first())
            .addInputState(input)
            .addOutputState(output, PARKING_SESSION_CONTRACT_ID)
            .addCommand(updateCommand)
            .setTimeWindow(serviceHub.clock.instant(), 5.seconds)
    progressTracker.currentStep = SIGNING_TRANSACTION
    val ptx = serviceHub.signInitialTransaction(txBuilder, ourSigningKey)



    progressTracker.currentStep = COLLECT_SIGNATURE
    val renterFlow = initiateFlow(inputParkingSession.renter)
    subFlow(IdentitySyncFlow.Send(renterFlow, ptx.tx))
    val stx = subFlow(CollectSignaturesFlow(
            ptx,
            setOf(renterFlow),
            listOf(ourSigningKey),
            COLLECT_SIGNATURE.childProgressTracker())
```

```
    )

    progressTracker.currentStep = FINALISING_TRANSACTION
    subFlow(FinalityFlow(stx, FINALISING_TRANSACTION.childProgressTracker()))

    if (amount.quantity + getRateBaseOnCurrentTime(inputParkingSpace.dayRate,
inputParkingSpace.nightRate) * 100  >= 20 * 100) {
        val test = subFlow(FinishParkingSession.Initiator(output.linearId))

    }
    return "Update!"
}
private fun verifyUpdate(tx: LedgerTransaction, signers: Set<PublicKey>) = requireThat {
    "Consume previous parking session state." using (tx.inputStates.size == 1)
    "Only one parking session state should be created when updating." using (tx.outputStates.size == 1)
    val inputParkingSession = tx.inputsOfType<ParkingSession>().single()
    val outputParkingSession = tx.outputsOfType<ParkingSession>().single()
    "An updated parking session must have positive amount." using (outputParkingSession.amount.quantity
> 0)
    "An updated parking session must have bigger amount than input one." using
(inputParkingSession.amount < outputParkingSession.amount)
    "The tenant and lender cannot be the same identity." using (outputParkingSession.tenant !=
outputParkingSession.renter)
    "Both lender and tenant together only must sign updating transaction." using
        (signers == keysFromParticipants(inputParkingSession))
}
```

Figure 6 Upadate parking

Finishing parking session is discussed in the next section. Sequence diagram of parking update using Corda is in the Appendix VI and VIII

3.2.6 Finish parking session

Finishing the parking session whether it happens due to maximum fee reached or user invoked finish, is the following. Tenant creates a new transaction which has one input state. Input state is the last pending parking state in the vault. Then tenant's node signs the transaction and verifies it. After verifying, tenant's node will send the partially signed transaction to the renter node for signing. Renter node signs and verifies the transaction. If transaction is valid, renter node will send it back to tenant node. Once signed transaction has both tenant and renter signature it is sent to notary for verification and approval. Notary

will verify the transaction, sign it and sends it back to tenant node. Tenant node will send the notarised transaction to renter node. The outcome of this transaction is that tenant will send Cash to renter using Corda's own Cash state. Finishing session code is shown in Figure 7

```
@Suspendable
override fun call(): SignedTransaction {
    // Stage 1. Retrieve obligation specified by linearId from the vault.
    progressTracker.currentStep = PREPARATION
    val parkingSessionToFinish = getParkingSession(parkingSessionReference)
    val inputParkingSession = parkingSessionToFinish.state.data

    // Stage 2. Resolve the lender and tenant identity if the obligation is anonymous.
    val tenantIdentity = resolveIdentity(inputParkingSession.tenant)
    val renterIdentity = resolveIdentity(inputParkingSession.renter)

    // Stage 3. This flow can only be initiated by the current recipient.
    check(tenantIdentity == ourIdentity) {
        throw FlowException("Finish parking session flow must be initiated by the tenant.")
    }

    // Stage 4. Check we have enough cash to settle the requested amount.
    val cashBalance = serviceHub.getCashBalance(inputParkingSession.amount.token)
    check(cashBalance.quantity > 0L) {
        throw FlowException("Tenant has no ${inputParkingSession.amount.token} to pay.")
    }
    check(cashBalance >= inputParkingSession.amount) {
        throw FlowException("Tenant has only $cashBalance but needs ${inputParkingSession.amount} to
settle.")
    }

    // Stage 5. Create a finish command.
    val finishCommand = Command(
            ParkingSessionContract.Commands.Finish(),
            inputParkingSession.participants.map { it.owningKey })

    // Stage 6. Create a transaction builder. Add the finish command and input parking session.
    progressTracker.currentStep = BUILDING
    val builder = TransactionBuilder(firstNotary)
```

```
            .addInputState(parkingSessionToFinish)
            .addCommand(finishCommand)

    // Stage 7. Get some cash from the vault and add a spend to our transaction builder.
    // We pay cash to the renters payment key.
    val renterPaymentKey = inputParkingSession.renter
    val (_, cashSigningKeys) = Cash.generateSpend(serviceHub, builder, inputParkingSession.amount,
renterPaymentKey)

    // Stage 9. Verify and sign the transaction.
    progressTracker.currentStep = SIGNING
    builder.verify(serviceHub)
    val ptx = serviceHub.signInitialTransaction(builder, cashSigningKeys +
inputParkingSession.tenant.owningKey)

    // Stage 10. Get counterparty signature.
    progressTracker.currentStep = COLLECTING
    val session = initiateFlow(renterIdentity)
    subFlow(IdentitySyncFlow.Send(session, ptx.tx))
    val stx = subFlow(CollectSignaturesFlow(
            ptx,
            setOf(session),
            cashSigningKeys + inputParkingSession.tenant.owningKey,
            COLLECTING.childProgressTracker())
    )

    // Stage 11. Finalize the transaction.
    progressTracker.currentStep = FINALISING
    val ftx = subFlow(FinalityFlow(stx, FINALISING.childProgressTracker()))
    subFlow(ChangeParkingSpaceStatusFree.Initiator(inputParkingSession.parkingSpaceReference))
    return ftx
}


private fun verifyFinish(tx: LedgerTransaction, signers: Set<PublicKey>) = requireThat {
    // Check for the presence of an input parking session state.
    val parkingSessions = tx.inputsOfType<ParkingSession>()
    "There must be one input parking session." using (parkingSessions.size == 1)
```

```
    // Check there are output cash states.
    // We don't care about cash inputs, the Cash contract handles those.
    val cash = tx.outputsOfType<Cash.State>()
    "There must be output cash." using (cash.isNotEmpty())

    // Check that the cash is being assigned to renter.
    val inputParkingSession = parkingSessions.single()
    val acceptableCash = cash.filter { it.owner == inputParkingSession.renter }
    "There must be output cash paid to the recipient." using (acceptableCash.isNotEmpty())

    // Sum the cash being sent to renter (we don't care about the issuer).
    val sumAcceptableCash = acceptableCash.sumCash().withoutIssuer()
    val amountOutstanding = inputParkingSession.amount
    "The amount paid has to be the amount outstanding." using (amountOutstanding ==
sumAcceptableCash)

    // Checks the required parties have signed.
    "Both lender and tenant together only must sign finish transaction." using
        (signers == keysFromParticipants(inputParkingSession))
}
```

Figure 7 Finish parking session

Sequence diagram of finishing parking session using Corda is in the Appendix VIIII

3.2.7 Parking space entity and ownership

Parking space is the central entity or asset in the use case this thesis is discussing. Since parking space has a strict ownership and it is a real object, there needs to be a way to represent all of the ownership aspects that comes with renting it out. Normal, on-paper renting process involves a paper contract which state both tenant and renter rights about the rental object. This can be attached to the state in Corda, but as the system provides an interface for the user to search for the available parking spaces, it needs to know, which parking spaces are already occupied, and which are not. In addition to the status of the parking space, system also needs to know to whom the specific parking space belongs to. Those two properties of the specified space need to be fulfilled with each of the parking spaces, meaning in every state that the system can be, the status and the owner of the parking space can be distinguished.

State and ownership are vital properties of the parking space, because the determine, if a space can be rented out or not.

For example, the application would use only ownership property, then the possible rental process, is conducted as short-term ownership of the specific parking space. This can then be adapted into the on-paper contract that is attach to the state in the ledger, stating that this short-term ownership does not grant any legal authority for selling or modifying the space in any way. Since ledger has the digital representation of the real-world asset, the author proposes the term "token" to point to the occupant of the parking space at given time. The main issue in the ledger system would be that if only the ownership property is considered then there is no limitation on further renting the space to third party. This means that it is possible for a party to earn a renting fee for the space this party does not own. Also, it possesses another problem, regarding maximum period of parking. Since true owner of the parking space is only seeing the rental state between him and the first renter, he may end up in a state, where the maximum rental period elapses and the parking space "token" is not returned to him, since the party that should return the "token", does not have it. This situation can be avoided, using sophisticated mechanisms for limiting the rental period for second, third, etc... renters, to guarantee that when the maximum rental period for initial renting process elapses the renter has the possession of the "token" that is then returned to the true owner of the parking space. Since this mechanism can be very complex and difficult to reason about in real life, the author of the thesis will introduce a business rule that disallows further renting the parking space.

As for keeping solely the state of the parking space, there is no way to determine the owner of that particular parking space, meaning this causes problems regarding self-renting one's own parking space and also the above-mentioned further renting problem.

The easiest and the most intuitive way for the author is to solve those issues by having both state and ownership property of the parking space in the system. That way one can only rent out a parking space which he owns and which is not rented out. Now it is not possible to further rent the space, since initial renter is not the owner of the parking space, the "token" can never be possessed anyone else than true owner of the parking space or the initial renter. With such two-dimensional authorization, the valid states of one parking space are shown in Table 1. To depict two different parties, abstract party names A and B is used. For parking space states, the two states FREE and RENTED OUT (party name, to whom it is rented out) are used.

Owner of the parking lot	State of the parking lot	Is state valid in ledger
A	FREE	VALID
A	RENTED OUT(A)	INVALID
A	RENTED OUT(B)	VALID

Table 1 Valid states of parking space in ledger

4 Framework for applications comparison

In this chapter, a framework for comparison of the two applications is developed and reasoned about. The author of the thesis will give motivation of why the framework is chosen as it is. In addition, comparison of the two approaches is given and discussion about benefits and drawbacks is presented.

4.1 Motivation of framework

Since the private distributed ledger technology is relatively new, there is no accepted framework developed to reason, if and when to use private distribute ledger over traditional centralised approach. One of the contribution of the thesis is to give one form of such framework. As a guideline, the author is using the approach used in one of the researches about using Blockchain solution in industries dealing with capital and durable goods [18].

4.1.1 Authentication and Authorization

When talking about any software system one key aspect is the user authentication and authorization. Since Blockchain is said to provide strong authentication and authorization [19], [20], it is justified to compare it to the traditional centralised solution. Although the centralised solution's user authentication and authorization can be done several ways, the author of this thesis is proposing to compare one of the easiest and most used authentication and authorization patterns – Oauth2 token for authentication and role attribute of user data entity for authorization. In addition to the patterns, thesis will bring out one possible way for each pattern to communicate the issues regarding either authentication or authorization to the end user.

4.1.2 Conflict Resolution

One crucial part of any system that has multiple users is how to maintain accurate, and consistent data. Also since there are multiple users using the system at any given time, there will happen at some point, that multiple users are trying to access same resource and modify it. When that kind of race is occurring, the systems needs a way to ensure that only one of the mutation will prevail and the other one gets rejected as invalid mutation. This is often named as data integrity, but the author of the thesis is naming it conflict resolution, not to bring in term data integrity, as it is used in database theory and in there means different thing than mentioned here.

Since conflict resolution can be done in multiple ways it and this is one of the core parts of distributed ledger technologies, the author of the thesis finds it justified to compare the approaches regarding how they are dealing with data manipulation and mutation conflicts.

4.1.3 Trust Between Parties

Since Blockchain was initially designed to provide trust between mutually distrusting parties without the mediator between them, this aspect is clearly one to keep in mind, when comparing it with centralised solution. [1] Main thing to compare here is how each of the approaches is guaranteeing trust between different parties. In addition to the method, used to gain trust one interesting aspect is where or not this trust is formal in a sense that, can some data be verified by other party.

4.1.4 Intermediation

One of the most important aspect of the comparison is involving the intermediation between the parties and whether or not is it necessary. What is more, can the intermediation involve any security threat. Also, what are the drawbacks and the gains of using intermediation in terms of speed and efficiency. As intermediation is tightly connected and discussed in all of the other aspects, it will not be discussed as a separate section. Intermediation will contribute to benefits and drawbacks in each separate aspect.

4.1.5 Scalability and Upgrade

Speaking about scalability, various Blockchain solutions have been proven not to scale, meaning the more nodes there are the more unusable the system gets. The issue of scalability is important, when speaking large scale systems and also, since Corda is private distributed ledger, are the same bottlenecks in there as well, compared to the public distributed ledgers and their scalability. [21]

Another aspect of maintaining a system is providing and enforcing upgrades of the system. This is also one of the key issues with public distributed ledger systems. The author of the thesis is proposing to analyse the way how to impose upgrades on Corda and compare it with centralised system to see if there are similar issues as in public distributed ledger systems.

4.1.6 Development Support

When new service or a framework is developed, number of systems adopting it will depend on how good is the development support. Development support consists of several parts, but the author of the thesis is going to discuss maturity, documentation and use of libraries. Thesis will compare those three components in Spring Boot application and in Corda.

4.2 Comparison of centralised and distributed application

In this section, the author of the thesis is comparing the traditional centralised approach with private distributed ledger approach using the above-mentioned framework. Since the private distributed ledgers are relatively new, there are no such frameworks for comparison developed yet, hence the author will provide one possible framework. The comparison is structured as mentioned in the chapter above. After comparing each of the aspects in framework a separate section about discussion will follow, in order to facilitate on the differences and reason about what is there to gain from each of the approach.

4.2.1 Authentication and authorization

One of the key aspects of modern systems is user and resource management. As the system grows the developer needs to take into account the possible risks, when choosing or implementing user authentication and authorization.

As for the centralised approach the author of the thesis is proposing to use OAuth2 authorization framework to grant access to resources. On high level the OAuth2 framework is working as a service that provides access tokens to certain user resources.

OAuth 2 framework will need the authorization to access user resource. This authorization is provided by the end user itself. This is normally obtained by client application. After user is agreed to authorise the framework to use certain resource, the framework will contact predefined Authorization Server (usually some HTTP service, like Facebook, Google, Twitter etc.), to get access token for the resource server to obtain the resource. Authorization Server will validate the authorization grant and if grant is valid, will respond with access token. [11], [22]

There are four main grant types: Authorization code, Implicit, Resource Owner Password Credentials, Client Credentials. As the types vary with the common usage, the author will describe in detail the most relevant type for mobile and web applications – Implicit.

Implicit type authorization grant means that no client secret is used, due to highly insecure environment like mobile applications and web. Using the implicit grant type, the user is presented with a link, that is used to make the request to the Authorization Server to obtain access token. Once user is redirecting the application to the specified link, he/she must log in to the service providing the Authorization Server (Facebook, Google, Twitter) or if he/she is already logged in then is prompted to either authorize or deny an application to access to the user account. This prompting for log in and/or authorization is provided by a user – agent, most commonly a web browser. Once the acceptance of the user if gained, the Authorization Server will give a redirect URL containing access token back to user-agent, which then gives it to application.[22]

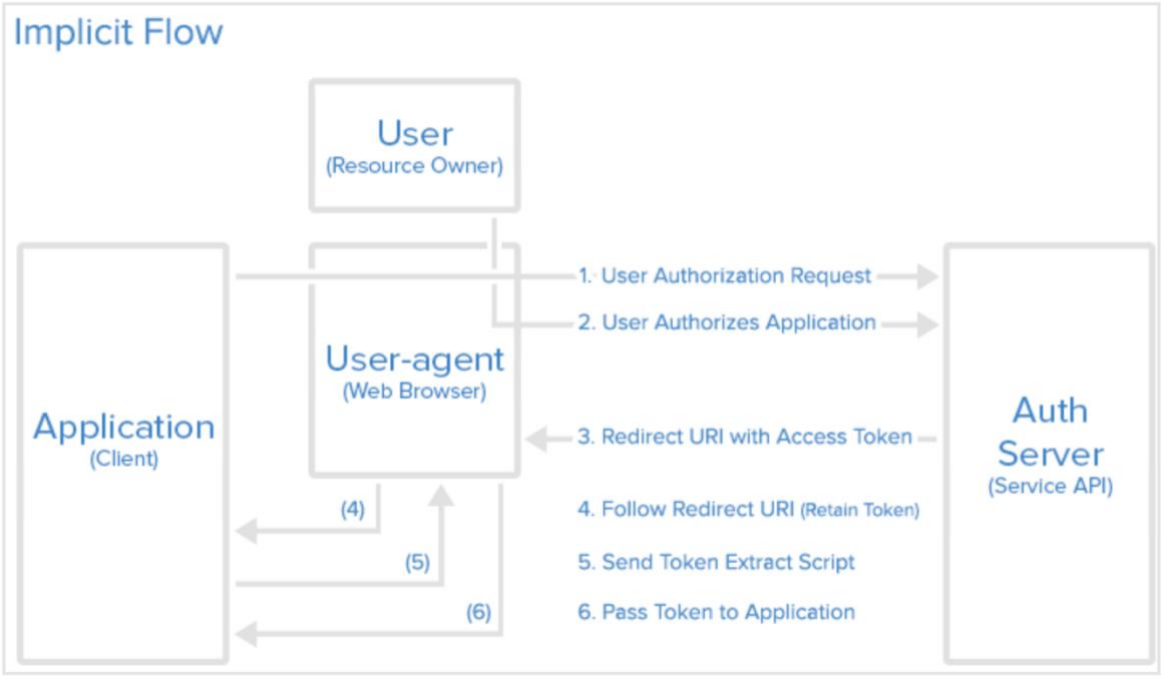


Figure 8 OAuth 2.0 Implicit grant type flow[23]

This access token is used to obtain protected resources from the main application resource server (usually REST API is used). In order for authorization server to know, which application is asking for access token, the application needs to be registered within the authorization server prior to getting the access token. Once registered each application will receive unique credentials to communicate with the server. This layer of security is needed to ensure that only the application that is registered can access the resources.[22]

After obtaining the access token user can navigate within the application and he/she will be provided with the protected recourses the user has access to.

In terms of private distributed ledger and user authentication and authorization, Corda uses public key cryptography for use authentication and authorization.[8]

Since in private distributed ledger's approach every machine is one node and therefore on user, there is no need for login. As long as the environment in which the node is running is protected, there is no need to implement login for the application. Another difference therefore is that distributed ledger approach means that one device is reserved for one user, meaning there cannot be multiple user on the same node. Of course, one can initialize multiple nodes on one machine, but each of the node is isolated, so that it does not have access to any of the node's resources on the same machine. The principle is similar to the UNIX operating system process management. Corda is developed in a way that each node is isolated and runs on separate process with its own sandbox. This is making the application relying on operating system and its way of process management rather than putting this the large part of the application's security on the developer of the application.[8]

What is more the user resources that can be accessed are also cryptographic key based, meaning all the on-ledger resources that user can access are duplicated inside user's vault, which can be accessed only with the user private key. This will make application's user authentication and authorization relying on operating system as well as cryptographic keys, which strength can be proven and will then in terms, provide better security over OAuth 2.[8]

4.2.2 Conflict Resolution

As discussed in the motivation of the framework, conflict resolution is important part of maintaining application's data consistency. There are two main ways how to resolve conflicts in central systems using relational database. Both of them involves locking. Locking means that one user will prevent others from manipulating the data entity until the lock owner releases the lock. [24]

Lock are mainly two types, depending on how strong locking is needed. A read lock is a lock, that the lock holder will set if it wants not to let other users update or delete the entity, but they can read it. [24]

For example, with a case study business use case, one user is browsing the available parking spaces, others can read the data about the parking spaces, but since a user is about to book something, the system prevents others for changing the price or delete the space from the system.

A write lock is a stronger lock, which means that if a user takes a write lock, he/she will prevent others from accessing, updating or deleting the entity that is locked. [24]

Write lock is used for example to update the parking space pricing, because pricing will cause entity to update and system is holding the write lock, to prevent other users from reading the old price.

In addition to lock types, there are two main locking types.

First method to use is pessimistic locking. With pessimistic locking the database locks the entity as soon as one client claims it and it is locked the entire time, that this entity is in the memory of the application. This kind of locking can be set to whole database, only one table or only one row. Pessimistic locking was the major conflict resolution mechanism back in the 1990s. It is still very widely used and mainly because it is simple to implement and changes made in database are consistent and safe. Main drawback of this locking type is the fact that it limits the amount of user a system can have, without the delays reaching above critical point. Since pessimistic locking will lock at least entire entity, other users must wait until the lock is released and if the number of users accessing the same entity at the same time grows too large, the system is unusable for some of them for a very long time. So, pessimistic locking is preventing scalability of the system.[24]

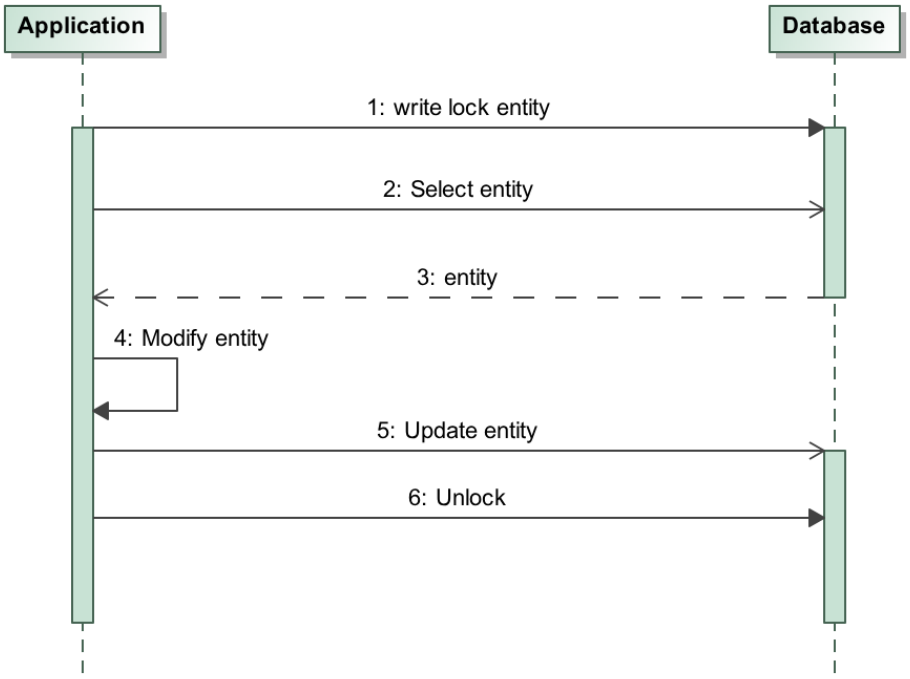


Figure 9 Pessimistic locking general overview

Second method is to use optimistic locking. With optimistic locking the entity is locked only for as long as it is necessary, meaning a read lock is obtained only until a user receives the entity from the database and then the lock is released. Same goes with the write lock, it will only be taken until entity is updated and once update completes the lock is released. Optimistic locking is used in multi-user systems, where conflicts are bound to happen. The way how the optimistic locking is resolving those conflicts is the following. Since the locks are released once the transaction to the database is completed there can be intermediate reads and writes, done by other users. In order to maintain consistency and keep the changes of the database safe, a unique identifier is used to mark the source every time somebody changes it. The conflicts happen only when someone is writing an entity to the database, so to overcome the conflict, once the user is triggering the write operation, a read operation is carried out prior to the write operation and the system checks, if the unique identifier of the entity is changes from the previous read of the entity. If it is not then there is no conflict and a write operation is carried out and the lock is released. If the unique identifier has changed then there are numerous possible ways to resolve the conflict. The easiest is to give up the writing operation and communicate to the user that somebody changed the entity in the meantime. Another way is to let user decide which update to keep and which one to discard.

Third and the most difficult way is to try and merge the updates and if conflicts still persist then let user decide what happens. There is, also systems, which does not take is as conflict, but rather override the previous update. Ignoring the conflict and simply overriding is the least desired action and is mostly just a part of legacy systems, rather than vital behaviour of the system.[24]

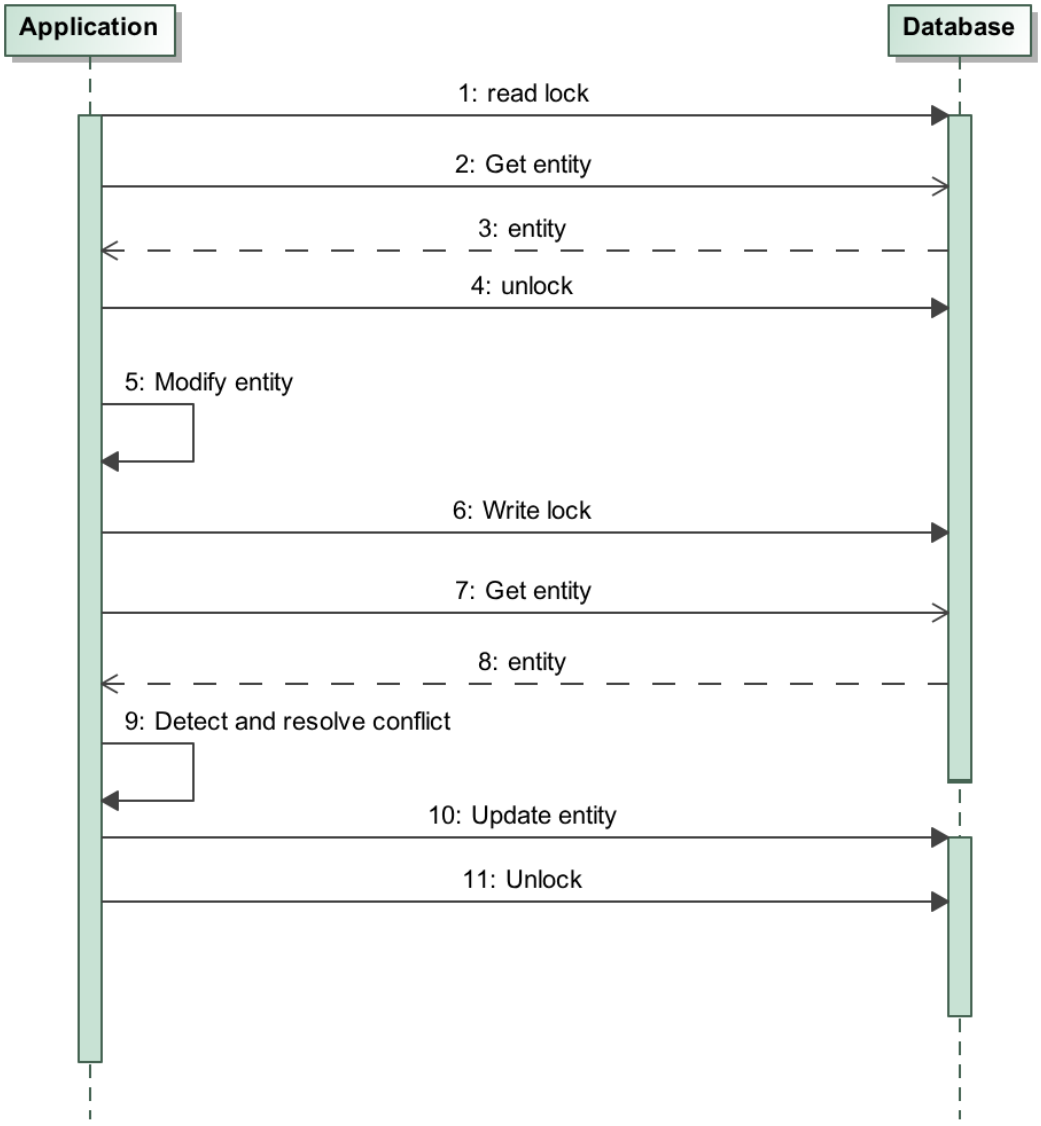


Figure 10 Optimistic locking general overview

One good example of optimistic locking system is versioning control system called Git. Git is free open source versioning control system, used to manage software projects. Git uses

optimistic locking to operate. In git, multiple above-mentioned conflict resolving techniques are used. The simplest git flow uses the easiest give up approach. For more advanced use cases, the user decision and merging are also used.[25]

The simplest git flow uses give up strategy when conflict happens. One user will pull the remote repository version. Then user can update/delete files in local copy of the remote repository. When user wants to push the changes to remote, he/she will need to commit the changes first, which among other process creates a commit hash – unique identifier of each change in git tree. Then user can push local changes to remote. Once push command is called git service will verify that commit hash in the remote repository hasn't changed since user's last pull. If it hasn't, the push will proceed and changes will be pushed to remote, with a new commit hash. If someone has pushed new commit in the meantime the commit hashes of remote and last pull will not match and git will give an error stating that user needs to pull first to get the latest changes and only then can he/she push the new ones. Since git will manage the commits, no commit is lost, so that once some changes are locally committed there can be multiple pulls before the local commit is pushed. More advanced use case that shows both user decision and merging is flow called branch merging in git. Since git is a tree, it can have several braches and from time to time those branches needs to be merged, in order to release a new version of software. With branch merging, user tries to merge different updates, when a conflict happens, meaning some parts of software are changed by multiple commits, then those conflicts are displayed to the user to decide, what to do with them. User can either choose one or he/she can manually manipulate the file to combine both of the changes into it.

Optimistic locking is useful for multi-user systems, as it is more scalable than pessimistic locking and it tries to lock as few as possible. The downside of optimistic locking is that is relatively difficult to implement to facilitate the optimised locking strategy.

Compared to central database approach, the conflict resolution in private distributed ledger Corda, is built inside the service itself. This means that the application developer does not need to implement it by him/herself, but rather call the necessary methods the right time. Corda uses similar approach as optimistic locking, but does it in terms of immutable transactions or states. In Corda. there is a "pluggable" service called notary. Notary is the basic mechanism for conflict resolution. At the time, the author is writing this thesis, Corda uses a simple notary in a form of dictionary. This dictionary has transaction hashes as key and transactions as values. Once the transaction is signed by all parties, it moves through

notary, which checks if the input transaction of current transaction has been recorded to the dictionary or not. If it is not notary will make a new key-value pair for input state and saves it in the dictionary and proceeds with publishing the transaction to the ledger. If the input state hash is present in the notary it will reject the current transaction as double spending and will throw an exception stating that input of the current transaction has already been published to the ledger. Then it is system developer to decide what to do. Usually system will just state that this transaction cannot be completed due to someone has already done transaction with same input state. The problem here is that Corda is using the same "chain" as Blockchain, meaning that every flow is chained. Only difference is that since each flow is isolated, different flows are not chained together. Thus, it is possible that a transaction might not have input state. This transaction is treated as unseen one in notary and the flow will proceed.[8]

4.2.3 Trust between parties

As trust between parties is one of the key aspects in software systems, it makes sense to compare it in the two approaches.

Central database solution is handling the trust between parties very easily. It will combine it to the authentication and authorization. Usually the central database approach assumes that parties do not trust each other at all and the mediator in form of a centralized database is used to bring parties to trust. This means that most of the central database systems developed today, is developed in a way that server is the single source of truth. This means that client applications will not try to facilitate any complex mechanism to achieve trust between each other. They rather both trust the server and thus will see it as the source of truth.

In distributed ledger systems, since there is no mediator the trust between different parties needs to be achieved in another way. In Corda, the trust between parties are achieved, by digital signatures and smart contracts. Since both parties needs to sign a transaction before it is committed to the ledger, a smart contract, which is a piece of code written inside the transaction will run to verify that transaction is valid. Only then the party will sign the transaction. Since smart contracts for each type of transaction is same for each party and they cannot be modified on runtime, this mechanism will guarantee that each party will verify the transactions using same criteria. Combined with digital signatures, smart contracts will provide sufficient mechanism for achieving trust between parties.

4.2.4 Intermediation

Intermediation is an aspect which is often frowned upon, but yet all central database systems need to use it. In central database system, the intermediation is providing the stability and the correctness of the system. Traditional central database system involves one central database and server and multiple clients. In these kinds of systems, all the interactions between clients usually go through server. This is necessary for the central database to keep track on the entity changes and system state.

In distributed ledger systems, intermediation is not required. In fact, distributed ledger systems were invented to prevent from using the mediator. The main purpose of ledger is to let clients communicate to each other and keep entities state in sync, without needing some mediator between them.

4.2.5 Scalability and upgrade

Scalability is one of the most important feature of modern software systems. Scalability is the ability of a system to remain operable and behave the same, when user base is rapidly growing. Modern software systems are built keeping in mind the issues of scalability and to cope with them.

In central database systems, the scalability is not an issue, with multiple services are providing both vertical and horizontal scalability. The most important link to achieve scalable system, using central database is the choice of technologies as well as how well they are meant to work with each other. Different technology stacks have different pros and cons when it comes to scalability, but all in all the problems of scalability in central database systems are resolved by various services and methods throughout the software industry.

Distributed ledger systems on the other hand, has fundamental scalability issues built inside them. For example, Bitcoin has to broadcast all of the transaction to all the nodes in the network. When a network grows, more and more transactions are made, which means more and more of them needs to be broadcasted as well. This is a fundamental scalability issue in Bitcoin that cannot be resolved by any services as it involves changing the architecture of the whole Blockhain.

Like Bitcoin Etherium faces a similar scalability problem. This problem can be abstract away to any public distributed ledger system, no matter the exact details. Until distribution meaning broadcasting to whole network, the issue remains.

Unlike public distributed ledgers, Corda has no public broadcasting. This means that transaction is shared only to parties involved in this transaction. Unless the whole network is involved in a transaction, which is highly unlikely, the system will not broadcast the transaction to whole network but a small subset of the network. Since the nature of the Corda is financial, it is believed that the amount of parties involved in one transaction is very small, compared to the potential number of nodes in the network.

Other big part of any software system is how to provide updates of the system. In addition to support updates, how to make updating process to disturb the working system as less as possible.

In central database systems, clones are used to provide updates. This means that a new version of server and database is initialized on another machine. All new requests to the server is redirected to new version of the server and once all of the interactions with the old server are completed. The old server is shut down and the new one will remain as the running server. Such pattern means that no client needs to approve any updates, since server will guarantee entity state synchronization and redirect patterns. This is making upgrades not dependable on clients, meaning server provider can upgrade system whenever is necessary.

With distributed ledger systems, since there is no single source of truth, the upgrade flows are much more complex. Since with Blockchain, there is a chronological order of the chain, all the nodes in the network needs to agree on the update. In other words, in order to introduce new rules to system using Blockchain, nodes must update their software. In order to introduce new rules, there needs to be a forking of the Blockchain. There are two different kind of forks. Soft forks and hard forks. Soft work as so called backwards-compatible, meaning they introduce additional features, but do not seize the old rules to work in the chain. This means nodes that do not update their software can produce valid block to Blockchain, but will not have access to new features. Hard forks are the ones that do not allow old rules to prevail in the chain. This means that once a hard fork is issued only those nodes, who updated the software can push to ledger and contribute to verifying the transactions.

Whether talking about soft or hard forks, the issue with both is that every node in the network must at to be able to continue as up to date.[26]

Corda uses similar system as distributed ledgers using Blockchain technology. Once an upgrade is proposed on the smart contract, each participant in the state that in containing the

contract being updated needs to authorize the update. Once all of the participants have authorised the upgrade one of them will initiate the upgrade. The initiation crates new transaction consisting of an old state and the new one. Then all of the participants need to verify and sign the transaction. Once initiator has collected all signatures the transaction is notarised and push to ledger. [16][8]

4.2.6 Development Support

Spring is widely used Java framework, that is mainly used to develop large scaled enterprise applications. There are numerous sub projects being developed under Spring, for example Spring Boot, Spring Security and Spring HATEOAS to name a few. Spring initial release was on the October 1st, 2002. Since framework has been there for 15 years, it has matured a lot during that time. At the time of this thesis Spring is on version 5.0.5. Taking a look on the release schedule, one can see that the release cycle is 3-4 years. There is an extensive documentation in Spring official website [27] as well as vast amount of questions and threads in Stack Overflow. Documentation is well structured according to developer's needs. There are 7 main sections for documentation. Documentation presents code examples as well as detailed description of the functionalities.

In Corda, since it was open-sourced on November 30th, 2016, there is much less maturity. Platform is rapidly developing. This can be seen, if looking at the release cycle. At the time of the thesis Corda has been open-sourced about 1.5 years, and there is already version 3.1 out. In addition to short release cycle, the documentation contains exploratory features, as well as the platform itself. Due to those reasons, the author of the thesis needed to pick one specific Corda version into discussion, since during the work of this thesis there has been 2 major version releases, that is changing both API's and adding features to the platform.

Documentation contains some code examples and gives detailed description to features in the service. In addition, all classes in source code are documented. Since Corda is much younger technology than Spring, there is much less questions and threads in Stack Overflow.

4.3 Discussion

In this chapter, the author of the thesis will discuss about the gains and drawbacks of each framework section regarding both central database and private distributed ledger approach. The aim of this chapter is to analyse the strengths and weaknesses of both approaches regarding the above-mentioned framework.

4.3.1 Authentication and authorization

As discussed in comparison chapter 4.2, centralised database and private distributed ledger approaches are using different methods for user authentication and authorization.

Key difference is that while central database approach with OAuth 2 token uses some third-party service for authentication, Corda uses public key cryptography. Since public key cryptography, RSA to be more precise, has mathematical proof behind it, it is considered more secure than OAuth 2, with heavily depends on implementation and cannot be mathematically proven. What is more the misuse of OAuth 2 will lead to exposing sensible user data and credentials for third-party service. If the credentials leak, user access to Authorization Server will be compromised and not only the current system's user will fall into the hands of adversary, but also every other user, that is connected with Authorization server user, which credentials leaked. So, the risk of using OAuth 2 token, is that if the credentials to Authorization Server leaks, then access to all of the systems, using the same Authorization Server will leak as well.

Corda uses isolated RSA key pair to authenticate and authorize user. What it means, is that every node in Corda network, will initialize fresh RSA key pair once it is initialised. RSA public key is but into networks, key vault, for everybody to see, so that when needed, one node can mark other node as participant in the transaction state. RSA private key is stored in a key chain inside the nodes machine. The key chain is inside the nodes vault and is protected with application sandbox and process isolation. Although, Corda uses public key cryptography, in the event of key leakage, the only system affected by the leak is the current one. This means that the system is isolated and will not compromise any other systems.[8]

The main advantage of using OAuth 2 tokens over the RSA keys is the fact, that since OAuth 2 is using Authorization Server, the integration to external systems are seamless, and the identity matching is done via service. For the RSA keys the integration to external systems is complicated, since systems need to match the identities of the RSA keys and they also

need to share some common Certificate Authority in order to verify each other's identity. This will make integrations to external systems cumbersome. This is why using Corda in real-life everyday systems is not as seamless as using OAuth 2 authentication. Corda's advantage is the high level on security and in the financial industry, where Corda is meant to be used, the security is more desired than the ease of integration.

4.3.2 Conflict Resolution

As to conflict resolution, both central database and Corda uses quite similar approaches. Both of them have optimistic locking strategy as preferred one to use in multi-user system. Central database approach needs to take care of the conflict resolution on its own, meaning it is not built into the service, rather there are multiple database integration servers to choose from. It is developer's responsibility to choose the most optimal one for the given application.

In Corda, developer cannot choose which conflict resolving strategy to use, as it is built in the service notarization layer. Once the transaction is singed by all of the participants, the notarization service will take care of the conflict resolution. This will guarantee, that the conflict resolution is done is similar manner, with every transaction. Although, the consistency of the conflict resolution is present, the lack of customization means that the application might not be optimal.

In central database application, developer can modify the conflict resolution, meaning there can be database tables that is not necessary to try and merge the conflicting changes, but just override with the last change, while other tables might strictly need merging. With customization, the application can be made to work more efficiently and thus the scalability and speed of the system might benefit from those customizations. But, as with all customizations there is a risk of ruining the consistency of the conflict resolution.

To conclude, the conflict resolution in both of the approaches are somewhat similar, but the level of modification is different. This means that the software architect must choose between consistency and efficiency of the system.

4.3.3 Trust between parties

Trust between parties are a desired feature in software systems. Since central database approach and Corda uses different ways to bring parties into trusting each other, there is both benefits and drawbacks for both approaches.

In central database approach, the trust between parties is achieved using mediator, which has access to all of the system and can single-handedly manage all the data inside the application.

The benefit of this is that one part of the application is handling all the trust issues and it is easier to model the system, since all the heavy-lifting of the trust and management of data can be programmed inside one component of the system and all the others will use this component to communicate with each other. As mentioned before, the component that is responsible of achieving trust is the server. Since with central database approach all the communication between different clients are going through the server, it can verify and either allow or reject certain actions. Ultimately, this will resolve the trust issues between different clients.

Downside of such trust model, is the fact that all eggs are but into one basket, meaning all of the data is controlled by server. If server falls into the hands of the adversary, this means that clients has no control over the trust and they will continue to trust the server, leading clients to continue with their actions after the attack. Another drawback of such system is that it is impossible to isolate the infected server, meaning once infected the system must be taken down for diagnostic and fixing the security issue that led to adversary gaining access to the server. This means the all this time the service is unusable.

In Corda, the trust is achieved inside each node, meaning the service provides mechanism for each node to verify the data sent from and to it. Also, since there is no mediation of data, each node must gain trust of the other before doing any action with the other node. This kind of a trust model is common in distributed ledgers, but in public distributed ledgers, trust between two parties are not achieved solely by those two parties, but rather by the whole network, meaning the privacy of the nodes can be compromised. In Corda gaining trust between parties are done using only those parties and trusted ledger mechanisms, that cannot be altered by any node in the ledger.

The benefit of this kind of trust model is that the trust is achieved using only those parties involved in the action. This means that the privacy of transactions is guaranteed until none

of the parties involved in the transactions, have fallen into hands of an adversary. In addition to privacy, since there is no mediator in the system, there is no single point of failure. This means that there is no way for the adversary to gain access to all of the system, by gaining access to one of the nodes. Also since all of the nodes are isolated from each other the damage control is easily achieved by notifying honest nodes about the infected node, so they would disregard any transaction in progress with this node. Then this nodes certificate can be revoked by ledger's Certificate Authority, meaning this nodes signatures are invalid and node cannot participate in any of the transactions.

The drawback of this system is that it is more difficult to develop and the system is more error-prone, since in case of one node anomaly, none of the others can communicate it and/or can it participate in transactions. Also, since all of the parties in a transaction must gain trust first, in case of several parties, it can take significant amount of time, affected by network latency, verifying the identity of other nodes etc. This means that starting a transaction can take more time in Corda than in central database system.

4.3.4 Scalability and upgrade

Scalability and upgrade are necessary features of most software systems today. Multiuser systems are expected to tolerate user-base growth and provide a way to upgrade the system.

In central database approach, since all of the system is controlled by server, the scalability is achieved by either provide horizontal or vertical scaling of the server. There are numerous services, providing automation of scaling, in order for a software system to withstand the increasing load.

Benefit of central database system in terms of scalability is that only one component will be needed to scale in order to take one the additional load. Also, since central database approach has been out there for a long time, there are services to support the on-demand performance on the server.

The drawback of central database system scaling is that it needs to be carefully though through, since wrong scaling technique, would result in either a limit in scaling, meaning that the scaling technique would not provide sufficient effect, or scaling would become financially too expensive.

Since Corda is not using public notification, as it is private distributed ledger, the scaling issues of public distributed ledgers are not affecting Corda. One might say that notary

service node can be a problem of scaling, but since notary service is “pluggable”, the scaling of notary service is simply monitoring the number of nodes in ledger and adding more notary services when necessary. This will impose some synchronization issues, between different notaries, with Corda has built in flows to overcome them and they will happen automatically for the node.

The benefit of Corda is that it is said to scale by itself as long as the transactions have reasonable number of participants. Since in corda the transaction result is broadcasted on a need-to-know basis, the number of participants in transaction will affect to how many nodes the result is broadcasted to. Since it is reasonable to assume, that most of transactions are between two to four parties, the scalability is not an issue in Corda.

In central database system upgrading the system is as simple as restarting the server with updated code and if needed updating client applications. The later one is needed if server provides a non-backwards compatible upgrade.

The benefit of central database system in terms of upgrading is, since all of the entity and data management happens through the server then update is simple, since only server is needed to be updated. The cost and complexity of usual upgrade is low.

In Corda, upgrade means either providing new features or changing smart contracts that are already in place. The upgrade flow of Corda is discussed in section 4.2.5.

The benefit of such system is that upgrades can be done in patches, meaning developer can update only one smart contract and only a part of the ledger would be affected by this update, meaning with update there is no need to bring down the entire ledger, just the nodes affected by this update.

The drawback of such upgrade flow is that every node, issuing the old state, including the faulty contract, must agree on upgrade before it can take place. This cannot be seen done in certain period of time, meaning upgrades can take vast amount of time, before they reach into the ledger. Also, some nodes can reject the upgrade, meaning this upgrade will never be carried out into the ledger. All in all, the upgrade flow of Corda is both expensive and difficult to manage, with numerous parties involved.

4.3.5 Development Support

Development support is important part of any software framework or service. If a framework or service is lacking development support, it is much harder to integrate into software systems, which is leading to avoidance by system developers. One key aspect of development support is how mature is the framework or service being used.

In Spring, since the framework has been out there for 15 years, it is matured and there are structured API's in place. In addition, since the framework is vastly used in the industry, there is also huge community that is constantly applying fixes and proposing improvements to the framework. In addition to the community, there is also great number of threads in Stack Overflow, making the development using Spring much easier, since there are code examples for specific problems.

One drawback of matured system is that there is no good way for extensive new features, since the user base is so big and systems, where Spring is used as enterprise systems, most of the improvements to the framework needs to be backwards compatible. Otherwise vast amount of systems would be broken if new version of Spring comes out.

In Corda, since it has been out there for only 1.5 years, it is immature. This is shown by the numbers of releases within this period. There has been already 3 major releases and API's has been constantly changing. In addition, both documentation as well as release notes states that there are experimental features in new version of Corda, which is a clear sign of immature platform. Documentation is in place with some code examples, but since the user base is limited, Stack Overflow does not have nearly the amount of threads compared to Spring.

On benefit for the Corda is that since it is a young platform, there is not many systems using it, the platform can be changed and improved quite rapidly without worrying about the existing users and backwards compatibility.

In addition, Corda has many code example projects designed to help developers to cope with Corda. Furthermore, there is a Slack workspace for Corda to ask technical questions and they also have technical office hours on Wednesday. Author used both Slack and technical office hours to get answers to questions and problems related to development of Corda application. The answer where usually given within one working day.

5 Conclusion

Thesis brought out some of the key differences between traditional central database systems and private distributed ledger system Corda. As private distributed ledgers are relatively young technology there are not many research conducted in this field. This let author of the thesis to develop new framework for comparison. Furthermore, thesis also developed a framework to compare 6 aspects in the architectures of both systems. Also, motivation of why those 6 aspects where chosen was reasoned about and discussion about benefits and drawbacks related to the aspects was conducted. All of the above-mentioned was done to answer two research questions, stated in the introduction of the thesis. Since without knowing the architecture of both systems, it is hard to reason and discuss about them, the author of the thesis analysed the architecture of both systems and made some models to illustrate both systems and their differences.

Since private distributed ledger is young research field, many more research papers are to come. Also, since this thesis focuses on the analyses of the systems, implementing them in practise and integrating them on existing systems are out of the scope of this research and are left as future work. In addition, some of the topic not covered by the thesis are brought out in the next chapter.

5.1 Outlook

This chapter will bring out some of the future works regarding the analysis of the private distributed ledger approach as oppose to central database approach.

This thesis is analysing some aspects of software design, while leaving others to be analysed by other researchers.

One big topic that left out in this thesis, is setup of the system and how to gain access in the system using private distributed ledger Corda. In addition, what are the differences between traditional central database system setup and access and Corda's approach.

Another topic that is not covered in the thesis is how to support offline features and if Corda has some limitations to it or if its supported at all in Corda. What is more, are Corda's offline support any different than central database one.

Third topic not covered is to which platforms Corda can be applied to. Since there is analyses in the thesis, but no practical implementation one aspect not covered is if and how easily can Corda be ported to different platforms like desktop, IoT devices, mobiles, etc.

6 References

[1] N. Satoshi, “Bitcoin: A Peer-to-Peer Electronic Cash System,” 2009. https://bitcoin.org/bitcoin.pdf (08.11.2017)

[2] R3,Corda - Frictionless Commerce, https://www.corda.net/ (08.11.2017)

[3] A. Back, Hashcash - A Denial of Service Counter-Measure, 2002, https://www.researchgate.net/publication/2482110_Hashcash_-_A_Denial_of_Service_Counter-Measure (27.11.2017)

[4] Blockchain Luxembourg S.A., Blockchain Data Charts, 2017, https://blockchain.info/charts (03.12.2017)

[5] UK Government Chief Scientific Adviser, Distributed Ledger Technology: beyond block chain, 2016, https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/492972/gs-16-1-distributed-ledger-technology.pdf (03.12.2017)

[6] Ethereum Foundation, Ethereum, 2017, https://www.ethereum.org/ (05.12.2017)

[7] R. Gendal Brown, J. Carlyle, I. Grigg, and M. Hearn, “Corda: An Introduction,” 2016, https://docs.corda.net/_static/corda-introductory-whitepaper.pdf (10.12.2017)

[8] M. Hearn, “Corda: A distributed ledger,” 2016, https://docs.corda.net/_static/corda-technical-whitepaper.pdf (10.12.2017)

[9] A. Osterwalder, *The Business Model Ontology – A Proposition in a Design Science Approach*. 2004 http://www.hec.unil.ch/aosterwa/PhD/Osterwalder_PhD_BM_Ontology.pdf (12.12.2017)

[10] Stripe, The new standard in online payments, 2017, https://stripe.com/ (15.12.2017)

[11] Okta, OAuth 2.0, 2018, https://oauth.net/2/ (15.12.2017)

[12] Stripe, Java library for the Stripe API, https://github.com/stripe/stripe-java (15.12.2017)

[13] R3, Heartbeat CorDapp, 2017, https://github.com/joeldudleyr3/heartbeat (18.05.2018)

[14] R3, The Obligation CorDapp, 2017, https://github.com/roger3cev/obligation-

cordapp (18.05.2018)

[15] R3, Crowd Funding Demo (Observable States), 2017, https://github.com/corda/observable-states (18.05.2018)

[16] R3, Corda DLT service Kotlin documentation, 2018, https://docs.corda.net/api/kotlin/corda/ (16.05.2018)

[17] R3, Event Scheduling Corda DLT documentation, 2018, https://docs.corda.net/event-scheduling.html (15.12.2017)

[18] J. Mattila, T. Seppälä, and J. Holmström, *Product Centric Information Management: A Case Study of a Shared Platform with Blockchain Technology*, 2016, https://escholarship.org/

[19] D. Shrier, W. Wu, and A. Pentland, Blockchain & infrastructure (identity, data security) part 3, *Massachusetts Inst. Technol.*, 2016, http://www2.caict.ac.cn/zscp/qqzkgz/ljyd/201609/P020160923325936264935.pdf (18.04.2018)

[20] M. Thakur, Authentication, Authorization and Accounting with Ethereum Blockchain, University of Helsinki, 2017, https://helda.helsinki.fi/bitstream/handle/10138/228842/aaa-ethereum-blockchain.pdf?sequence=2 (18.04.2017)

[21] M. Vukolić, The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication BT - Open Problems in Network Security, 2016, pp. 112–125.

[22] IETF Tools, The OAuth 2.0 Authorization Framework, 2012, https://tools.ietf.org/html/rfc6749#section-1.3.2 (18.05.2018)

[23] M. Anicas, An introduction to OAuth 2, 2014, https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2 (08.04.2018)

[24] S. Ambler, *Agile database techniques: effective strategies for the agile software developer*, 2004.

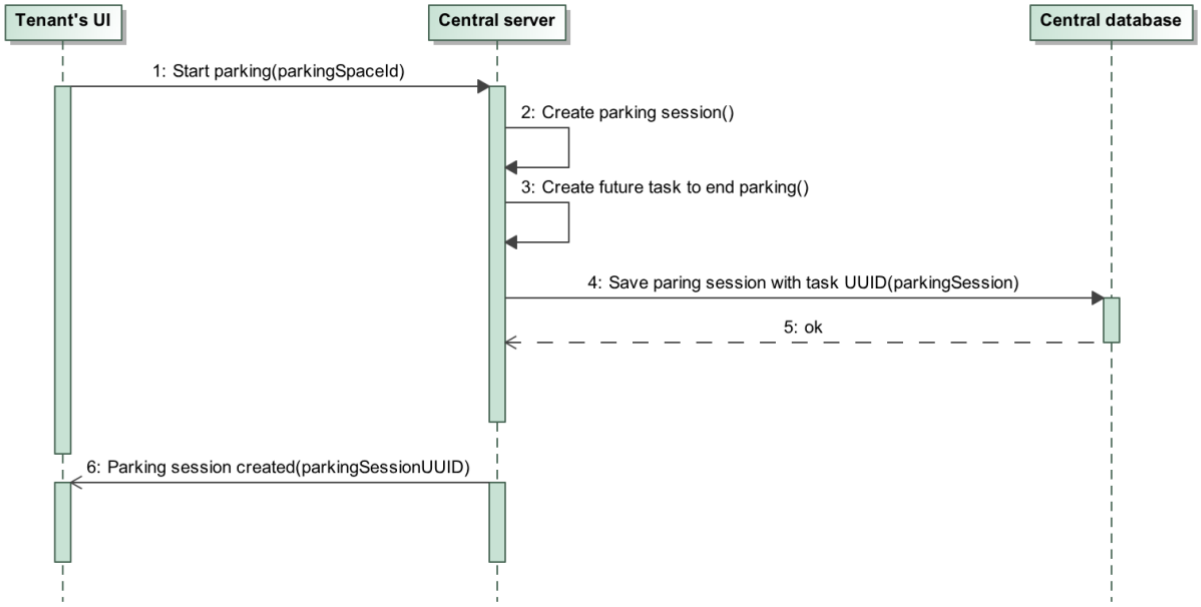
[25] Git --distributed-even-if-your-workflow-isnt, 2018, https://git-scm.com/ (18.05.2018)

[26] C. Decker and R. Wattenhofer, Information propagation in the Bitcoin network,

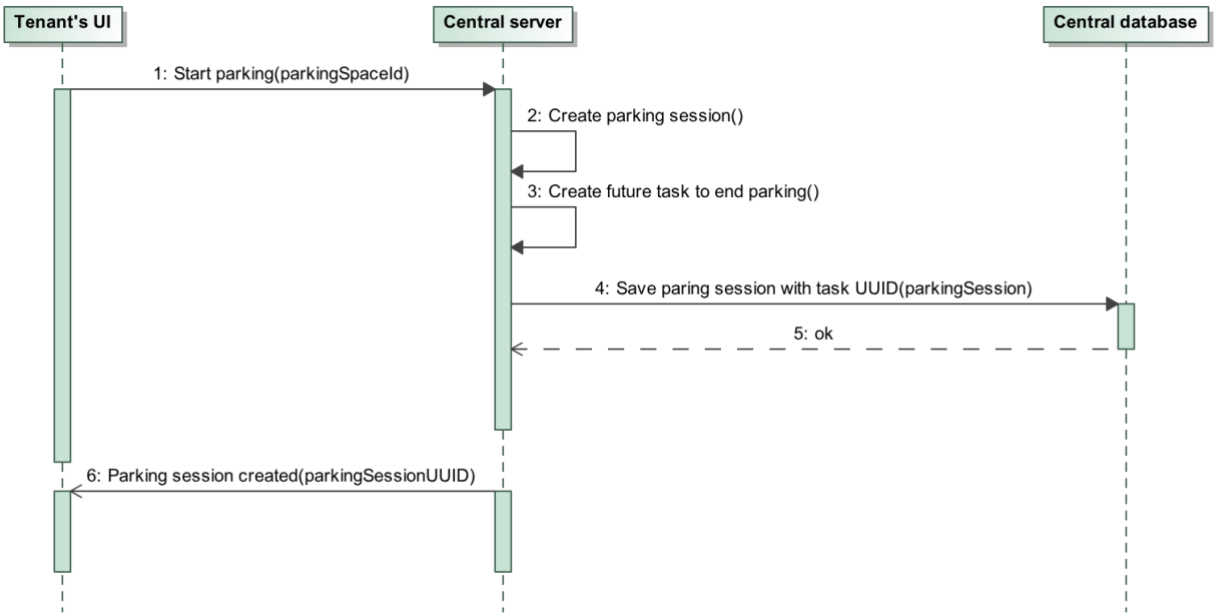
IEEE P2P 2013 Proceedings, 2013, pp. 1–10.

[27] Pivotal Software, Spring, 2018, https://spring.io/ (18.05.2018)

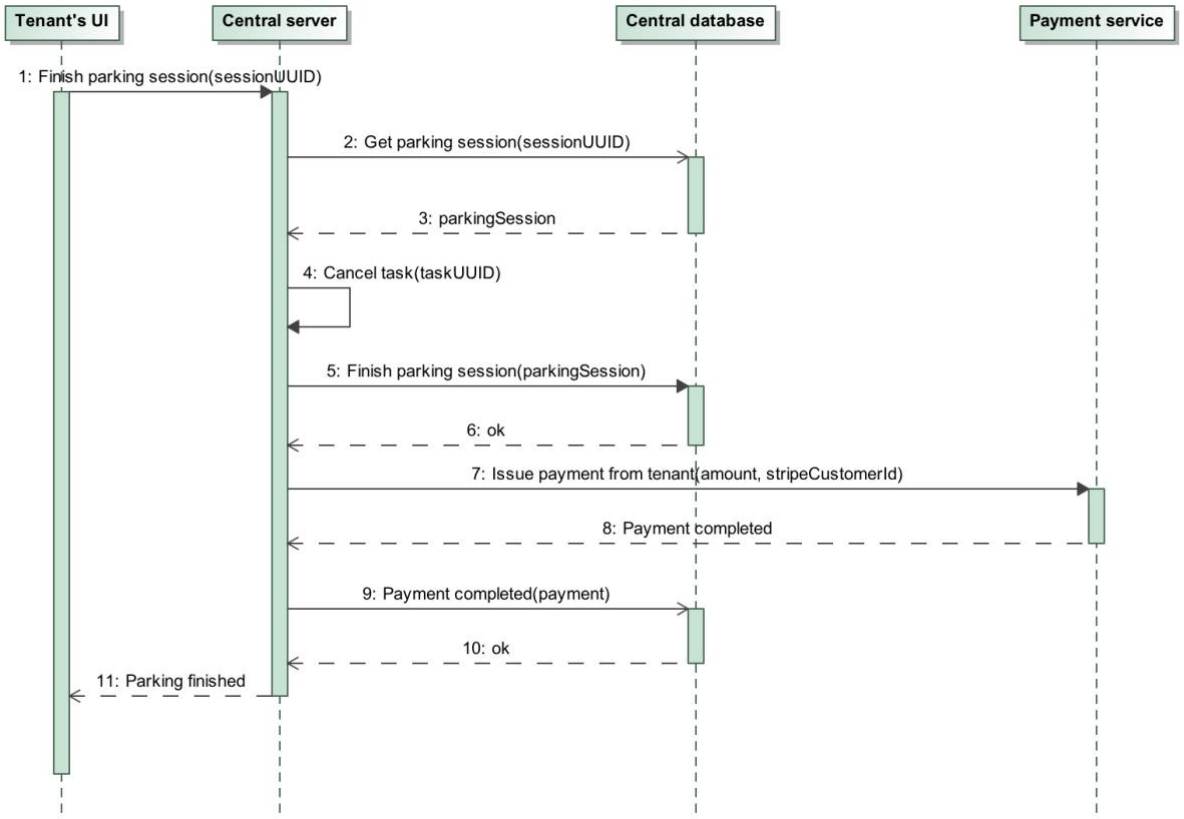
I. Sequence diagram of adding a parking space in central database system



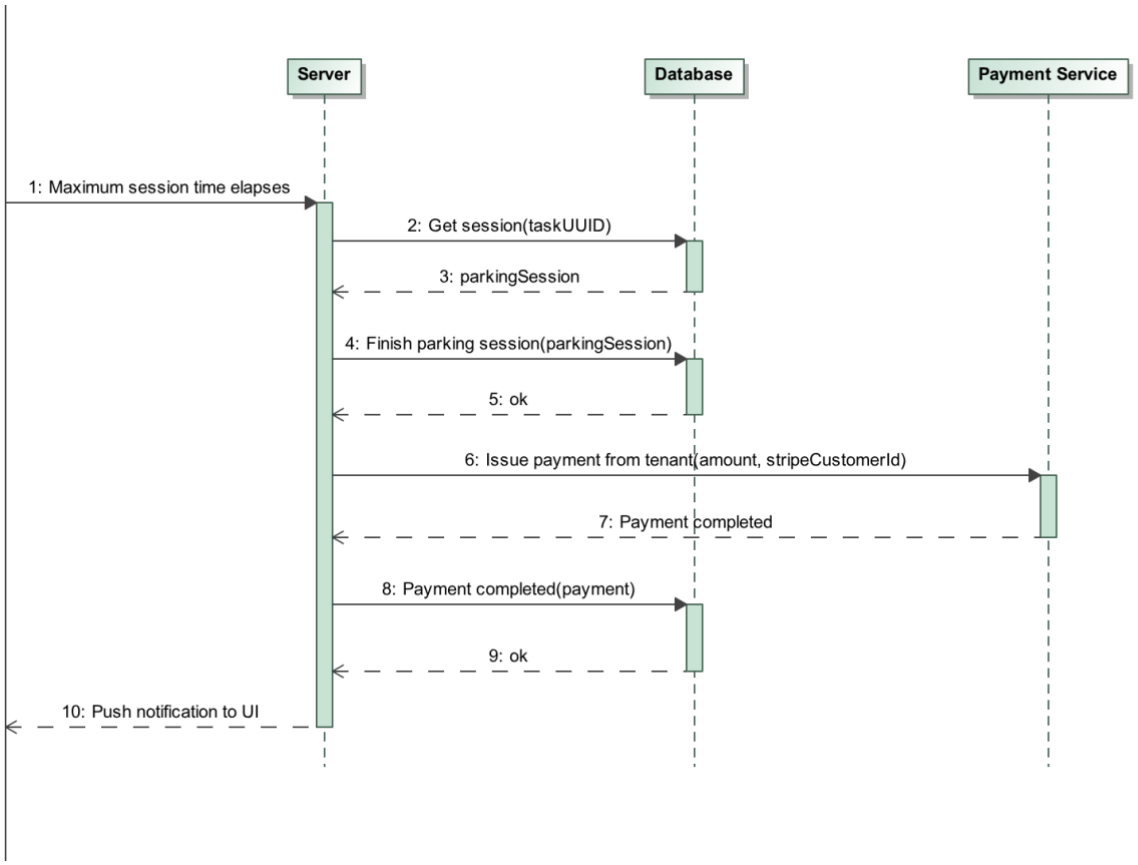
II. Sequence diagram of parking session start in centralised system



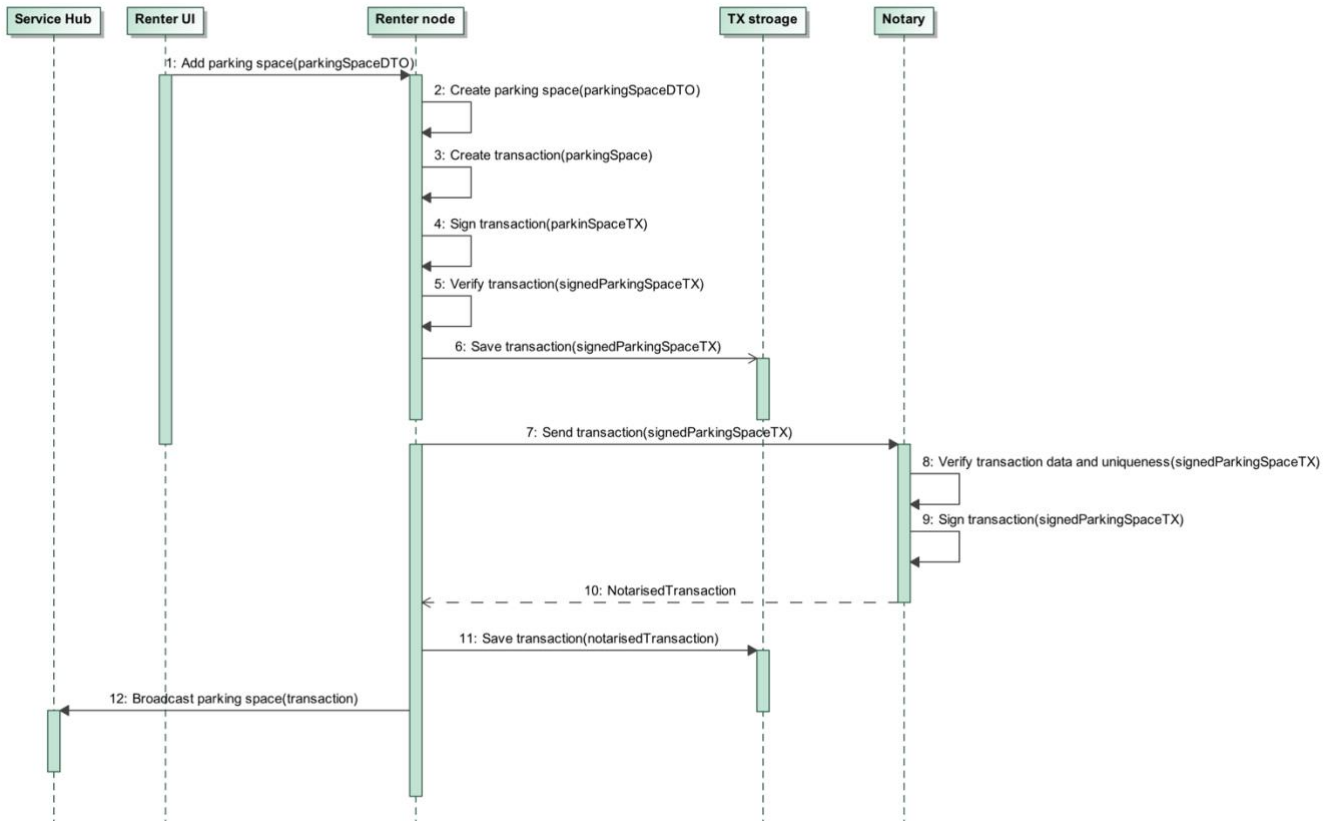
III. Sequence diagram of user finishing parking flow in central system



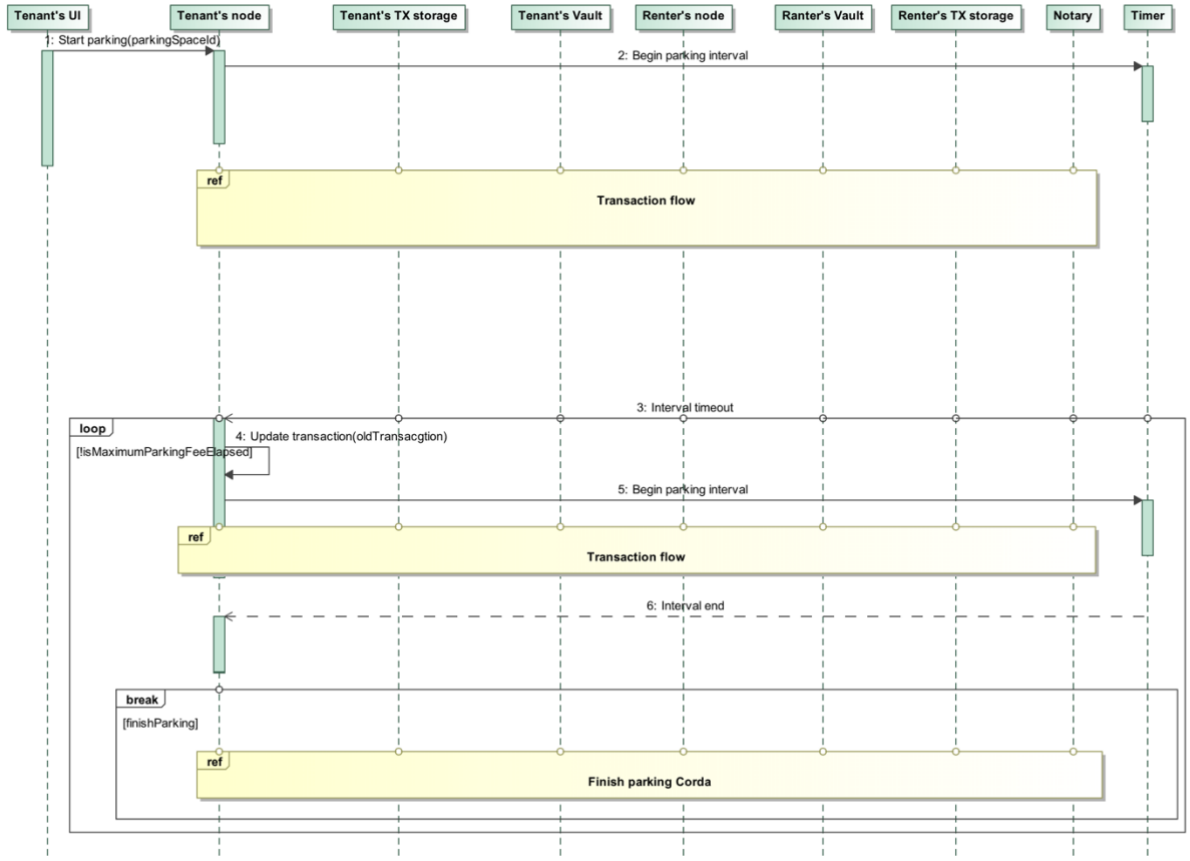
IV. Sequence diagram of maximum parking session time elapses in central system



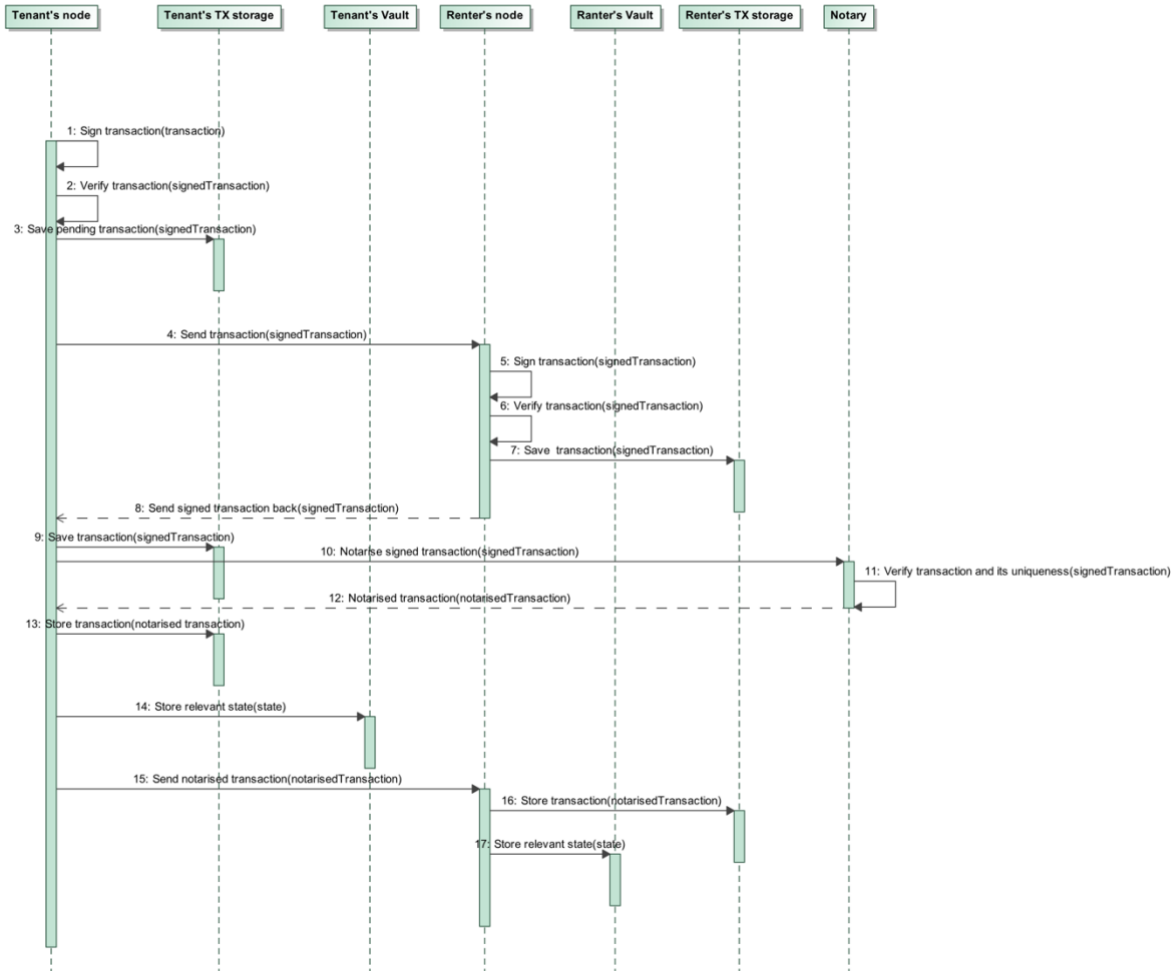
V. Sequence diagram of parking space addition in Corda system



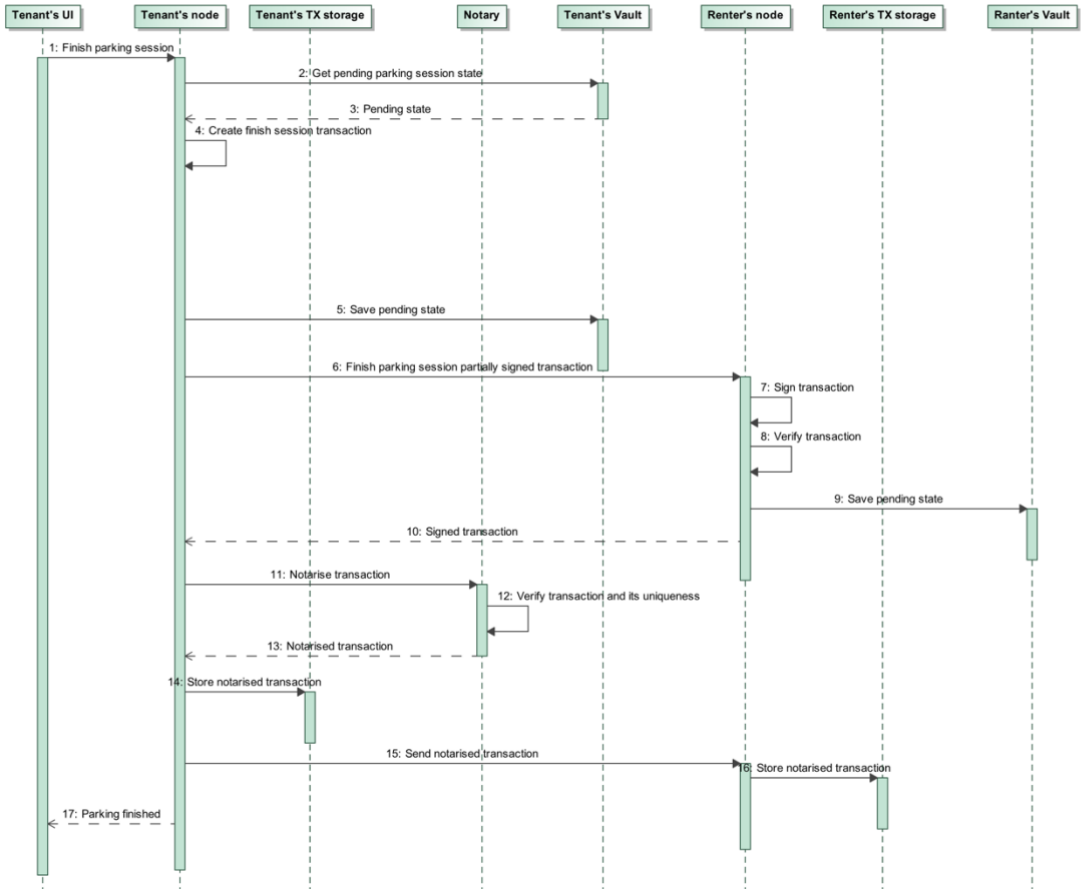
VI. Sequence diagram of parking session overall flow in Corda



VII. Sequence diagram of transaction flow in Corda



VIII. Sequence diagram of finishing parking session in Corda



IX. Definitions and Acronyms

Coin - A form of mediator in Bitcoin cryptocurrency system made up from digital signatures.

Corda - Service based on Distributed Ledger Technology.

Distributed Ledger – Consensus of replicated, shared and synchronized digital data, geographically spread across multiple sites, countries or institutions [5].

DLT – Distributed Ledger Technology

Mining – Performing computationally expensive operation, to find partial hash collision in order to verify a transaction in Blockchain.

Node – One participant of the network of ledger system.

X. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Olev Abel**,

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
   Case Study: Analysing Parking Solution using Corda DLT Service,
   supervised by Luciano Garcia Banuelos and Fredrik Payman Milani.

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, **22/05/2018**