Improving transaction throughput of the Hyperledger Fabric framework

Laurens de Gilde

January 31, 2019

Supervisors:	dr. ir. Marc X. Makkes	University of Amsterdam
	Albert Lakerveld	Oracle
	Huub Rood	Oracle
	Peter Spaanderman	Oracle
Host organisation:	Oracle	https://www.oracle.com

UNIVERSITEIT VAN AMSTERDAM
FACULTEIT DER NATUURWETENSCHAPPEN, WISKUNDE EN INFORMATICA
MASTER SOFTWARE ENGINEERING
http://www.software-engineering-amsterdam.nl

ORACLE CORPORATION
DIVISION: ORACLE HEALTH INSURANCE
http://www.oracle.com

Contents

Abstract

Centralized applications are prone to threats like a single point of failure(SPoF). Decentralizing an application is a resource efficient way of overcoming SPoF but keeping data consistent for a decentralized application can be challenging. A framework that allows for the decentralization of an application while maintaining data consistency is the Hyperledger Fabric framework. However, problems arise for the Hyperledger Fabric framework regarding its transaction throughput. The framework becomes impractical for decentralizing applications where the transaction throughput needs to be at a certain minimum rate. This thesis seeks to improve the transaction throughput such that a minimum transaction throughput is reached. Eventually defining whether the Hyperledger Fabric framework is applicable for decentralizing the *module of reimbursement.* To increase overall transaction throughput of the Hyperledger Fabric framework two key elements are improved: read- and write operation time on the key-value world state database. Read- and write operation time is determined by the data structure of the key-value world state database. The typical LevelDB world state database uses a flat data structure. This thesis shows that using a Patricia Trie, read- and write operation time decreases by a magnitude of 23 according to the amount of read- and write operations that need to be done by the module of reimbursement. The self-made implementation of the world state database makes it feasible to use the Hyperledger Fabric framework to decentralize the module of reimbursement and reach the desired transaction throughput.

Keywords: Hyperledger Fabric framework, Performance, Transaction throughput, World State Database

Chapter 1

Introduction

Companies rely on trusted third-party software which makes primary processes in day to day IT activities possible. An example of such software is: *providing a big data store for reading and writing of customer data*. This software is often deployed in a centralized manner[13]. The centralization of such applications unveils threats[12, 18] like *single point of failure*(SPoF). SPoF can be preserved by applying *replication*[22]; a centralized application is replicated twice on two different nodes. Replication A is active and replication B stays idle. When replication A crashes, replication B takes over until replication A is up again. But, replication inefficiently uses resources; replication B and therefore its node' resources will be idle most of the time. A more resource efficient solution to overcome SPoF is by decentralizing an application. Decentralizing an application is done by scattering multiple instances of the same application over multiple nodes and make these accessible to end-users. A decentralized application overcomes SPoF and utilizes resources efficiently; each instance of the application ran on different nodes is accessible to users. However, decentralized applications can be challenging regarding consistency of data[27]. A framework that allows for decentralization of an application, and has a built-in mechanism for data consistency is the Hyperledger Fabric framework [8].

The Hyperledger Fabric framework is used to build permissioned blockchain networks. It adds extra functionality on top of the standard blockchain semantics. This makes Hyperledger Fabric networks; permissioned blockchain networks build with the Hyperledger Fabric framework, more appropriate for business cases. A Hyperledger Fabric network consists of peers which deploy- and make accessible an application onto their stack. By scattering the peers of a Hyperledger Fabric network geographically over different nodes, an application can be decentralized. A client sends transactions to the network to interact with the application. Transaction throughput is defined as a rate at which transactions are sent to the network, processed and sent back by the network to the respective sender. It is broken down into two segments: the transaction *round-trip* time; how long does it take for a transaction to reach the network and, for the network to send back a response to the respective sender? And transaction *processing* time; how long does it take to execute the computations of the application at the network? The core characteristic of the Hyperledger Fabric framework that is utilized in this thesis is the decentralization of an application and overcoming SPoF of a centralized application. Although the following papers [33, 29, 1] published numbers about the transaction throughput of a Hyperledger Fabric network, they do not utilize the decentralization characteristic and keep the Hyperledger Fabric network in a centralized setup by deploying all the peers of the network on a single node. It is important that the transaction throughput of a decentralized Hyperledger Fabric network is researched where peers are scattered over different nodes. The problem with the Hyperledger Fabric framework is that there are currently no representative numbers available about the transaction throughput of a Hyperledger Fabric network when deployed in a decentralized manner on commodity hardware. This thesis analyzes the transaction throughput of a decentralized Hyperledger Fabric network deployed on commodity hardware and investigates how the transaction throughput can be improved.

Oracle Health Insurance provides a back office application for insurance companies, allowing for the appliance- and enforcement of the health care regulations obligated by the Dutch government. The health care domain and how its use cases can be applied with blockchain in general are studied extensively in [21, 2, 35]. The papers present pitfalls in health care and how these pitfalls can be preserved with blockchain. Oracle Health Insurance is therefore interested in investigating how the back office application fits in the blockchain technology in general. The back office application is separated into multiple stand-alone modules. One of these modules is the *module of reimbursement*, it incorporates a well-defined application logic which is used to process reimbursements in the form of *claims*. The module of reimbursement is centralized and the aforementioned threat arises for this module. The centralized implementation processes 50 claims per second at peak load. A detailed description of the module of reimbursement is as follows: by law, a Dutch citizen(insured individual) needs health insurance with an insurance company. The insured individual receives a treatment(health care action) from a hospital(health care provider). In the Dutch health care system, an insured individual first pays for the health care action after which he can reimburse the costs of the health care action if it is insured. The insured individual sends a *claim* of the reimbursement to the *module of reimbursement* facilitated by the insurance company. The module of reimbursement than processes this claim, i.e., execute the incoming claim against the well-defined application logic, which incorporates the needed computations to process the claim successfully. After processing, the module sends a response with the result of the reimbursement; will the health care action be reimbursed or not. The module of reimbursement is maintained by the insurance company. It is deployed in one facility and made accessible via a web portal. The module of reimbursement is a centralized application and SPoF arises in such conditions. With the analysis of the default transaction throughput and its improvement, this thesis eventually states whether a transaction throughput can be reached such that it becomes feasible to decentralize the module of reimbursement with the Hyperledger Fabric framework.

Hypothesis

A decentralized Hyperledger Fabric network is set up and empty transactions are sent to the network. An empty transaction is defined as *a transaction send from a client to the network(no transaction processing happens in the network, i.e., no computation steps) and receive a response from the network.* Each empty transaction is measured according to its time taken. These measurements define the transaction round-trip time. To investigate how the transaction processing time can be improved, we analyze the operations, that the well-defined application logic of the module of reimbursement executes, to process an incoming claim successfully. Each operation in the centralized environment is mapped to the decentralized environment, a Hyperledger Fabric network. The results of this analysis stipulate that the well-defined application logic does circa 40 read- and 10 write operations to a database to process a claim successfully. Data that needs to be used by a Hyperledger Fabric network is stored in an embedded database called the *world state database*. The read- and write operations time on the world state database is directly embedded in the transaction processing time and therefore affects the transaction throughput directly. This thesis investigates how to decrease the read- and write operations time on the world state database such that it decreases the transaction processing time and increases transaction throughput. The goal for this research may be formulated as follows: *Improve the read- and write operations time on the world state database such that the transaction processing time decreases and the transaction throughput increases for a Hyperledger Fabric network.*

1.1 Research questions

To offer a concise statement about the purpose of this thesis, research questions are proposed. The end goal is to stipulate whether a transaction throughput can be reached such that it becomes feasible to decentralize the module of reimbursement the Hyperledger Fabric framework. To answer this question related to *feasibility*, three *technical* sub-questions are proposed which enables us to answer the primary *feasibility* research question at the end of this thesis.

Research question 1: *Can a transaction throughput be reached such that it becomes feasible to decentralize the module of reimbursement with the Hyperledger Fabric framework?*

The Hyperledger Fabric framework is used to decentralize the module of reimbursement and overcome SPoF. But, there are currently no representative numbers about the transaction throughput of a decentralized Hyperledger Fabric network. This primary research question allows us to conclude, at the end of this thesis, whether a transaction throughput can be reached such that it becomes feasible to decentralize the module of reimbursement with the Hyperledger Fabric framework.

Research question 1.1: *What is the transaction round-trip time of a decentralized Hyperledger Fabric network?*

The transaction round-trip time is established for a Hyperledger Fabric network. This means: what is the time-frame for an empty transaction to be sent to a decentralized Hyperledger Fabric network and for the network to send a response to the respective sender? The total transaction time is defined; how long may a transaction take such that a transaction throughput of 50 is reached. Subtracting the transaction round-trip time from the total transaction time leaves us with a remainder. This remainder is the time-frame that can be used for the transaction processing time.

Research question 1.2: *What is the read- and write operation time on the LevelDB world state database of a Hyperledger Fabric network?*

The read- and write operation time on the LevelDB world state database is defined. An analysis is done to state if the computations and the read- and write operations that need to be done by the module of reimbursement to process a claim successfully fit in the remaining time-frame established in RQ 1.1. If this is not the case, research is done to investigate how the read- and write operation time can be improved.

Research question 1.3: *How to improve read- and write operation time on the world state database of a Hyperledger Fabric network?*

In this thesis, we improve upon read- and write operation time on the world state database. With this improvement, is it possible to do the computations and the read- and write operations within the remaining time-frame established in RQ1.1? Eventually reaching a transaction throughput that makes it feasible to decentralize the module of reimbursement with the Hyperledger Fabric framework.

1.2 Contributions

To improve read- and write operations on the world state database, a self-made world state database called PatriciaDB is build. It incorporates a key-value data structure: the Patricia Trie. This data structure ensues from applying the PATRICIA algorithm for read- and write operations on an object. The self-made world state database shows to outperform the default LevelDB world state database that ships with the Hyperledger Fabric framework. With the improved read- and write operation time due to a self-made world state database it becomes feasible to decentralize the module of reimbursement with the Hyperledger Fabric framework while reaching the desired transaction throughput. This thesis contributes the following:

1. Implement a self-made world state database for the Hyperledger Fabric framework such that the transaction throughput improves.
2. Analyze the performance improvement of a Hyperledger Fabric network with the self-made world state database as opposed to the default LevelDB world state database.

3. Stipulate whether the module of reimbursement can be decentralized with the Hyperledger Fabric framework.

The remainder of this thesis is structured as follows. In Chapter 2 the related work on the subject of performance and Hyperledger Fabric in combination with the health care industry is presented. Chapter 3 describes the background information about the blockchain technology and the Hyperledger Fabric framework. Chapter 4 presents the Patricia Trie and how it is implemented in a Java REST API such that it funds as the underlying data structure for the world state database. Furthermore, in Chapter 5 a detailed description of our experimental setup and how we will measure the performance of LevelDB and PatriciaDB is presented. Chapter 6 presents a qualitative and quantitative evaluation of the research. Finally, in Chapter 7 answers to the research questions and future work are given.

Chapter 2

Related work

This Chapter describes the related work of this thesis. Papers related to performance improvements of the Hyperledger Fabric framework are presented. The core contributions of these papers are presented and argued how these differ from the ones presented in this thesis.

Well known organizations like IBM, contribute to Hyperledger Fabric and try to find ways to improve the framework. [26, 23, 4, 32] propose solutions to improve the performance of a Hyperledger Fabric network. The papers state that the core performance bottleneck lays within the consensus that needs to be reached in a Hyperledger Fabric network and blockchain networks in general. Each paper proposes a different protocol to improve the performance regrading the consensus time. The papers give an intention that there are still improvements to be made to the Hyperledger Fabric framework. Although these papers are related to this thesis since the focus of these papers is on improving the performance of the Hyperledger Fabric framework. They differ since the focus is on a different component as the one of this thesis.

Paper [11] presents optimization's to the Hyperledger Fabric framework of which one is the use of a different world state database. The paper proposes a more light weight hash table as world state database. The new world state database that is presented in this papers allows for parallel read- and write operations. The new presented world state database shows to improve the transaction throughput. However, the paper does not present any improvements on read- or write operation time but analysis how parallelism of read- and write operation can improve the transaction throughput.

Furthermore, propositions [19, 25] present ideas for a relational database model as the world state database type. Relational database models are not yet supported in Hyperledger Fabric v1.2.0. The authors argue that most centralized applications use relationally structured data. The propositions state that it can be counter-intuitive to denormalise the relationally structured data to a key- value structure such that a centralized application can be decentralized with the Hyperledger Fabric framework. Also, the denormalization can result in performance degrading of the Hyperledger Fabric framework. The proposed solutions are not available in Hyperledger Fabric v1.2.0 and therefore not be compared to this research.

At last, [29] focuses on the world state database performance, the component that is analyzed and optimized in this thesis. It presents the performance of the default world state databases based on different transaction complexities. Although it is only investigating the default implementations LevelDB and CouchDB and not proposing new implementations. It provides insights on how the world state database performs in an advanced hardware environment while this thesis focuses on a more commodity hardware environment.

Chapter 3

Background

The semantics of the blockchain technology is described. Next, the Hyperledger Fabric framework components are described and how the individual components of the Hyperledger Fabric framework together compose a functional Hyperledger Fabric network. Furthermore, a description of how the module of reimbursement is applicable in a Hyperledger Fabric network is given. At last, is described how the relationally structured data used by the module of reimbursement in the centralized environment is denormalized to a key-value structure used in the decentralized environment. This is later used to motivate the argumentation of the Patricia Trie data structure for the self-made world state database.

3.1 Blockchain technology

A blockchain network can be used to exchange information without having the need of an authority(such as a bank). Moreover, peers that compose a blockchain network do not need to trust each other. These characteristics are made disposable by the way how information is exchanged between peers. A blockchain network consists of multiple geographically distributed *peers*. Each peer in a blockchain network has a *digital identity*. This digital identity is pseudo-anonymous and is unique for each peer in a network. Each exchange of information between two peers is done via a transaction. A transaction is an encapsulated package of attributes that is understood by a blockchain network. So if *peer X wants to exchange information with peer Y. Peer X sends a transaction to peer with digital identity Y.* Each peer in a blockchain network has a *balance*. The balance of each peer is stored in the *state database* and defines what information a certain peer holds. The balance of a peer is not always related to financial assets. This means that a blockchain network is not merely applicable in the financial sector but can be used in any sector in which information needs to be exchanged. If a blockchain network consists of peers which are car dealers. The information of each peer and therefore its balance could be *how many cars does peer X have?* The word balance here is used to explain what information a peer in a network possesses.

Each transaction in a blockchain network updates the balance of a peer. Since the peers in a blockchain network are distributed, each peer needs to be notified of each transaction that has happened on the network. A blockchain network uses *gossiping* to notify each peer of a transaction. Each peer in a blockchain network has an address list of a subset of all the peers; called its neighbors. A peer can reach its neighbors, these neighbors can, in turn, reach their neighbors; this is called gossiping and allows for a peer to broadcast a transaction across the network. There are two types of peers in a blockchain network: *light* and *full* peers. Light peers want to exchange information in the network and keep a copy of the networks *ledger* and *state database*. The ledger is a cryptographically hashed list of all the transactions that have ever taken place on the blockchain network. The ledger can only be tampered with via a 51% attack on a blockchain network or by forking the ledger [15]. These actions are unlikely to happen, therefore often a blockchain ledger is labeled as tamper proof. Full peers make sure *consensus* is reached in the network. When a *light peer* wants to exchange a piece of information it gossips the transaction to its neighbors, at a certain point in time the transaction will reach a *full peer*.

Full peers collect transactions broadcasted by light peers and collect them in a block. Once a block is full, the full peer executes a protocol specified by the blockchain network to validate the block. The protocol a full peer has to execute to validate a block is computational heavy. Popular known protocols with heavy computational tasks are *Proof of Work* and *Proof of Elapsed Time*. The full peer must devote a significant amount of time in solving a puzzle before being able to prove that a block is valid. Full peers benefit from executing the protocol because each transaction has a transaction fee. If a full peer validates a block, the accumulated transaction fee of the transactions in the block is received by the full peer. Once the block is validated the full peer distributes this block to an amount of other full peers and these, in turn, check whether the protocol was executed correctly. If this is proposition holds, the block is gossiped across the network. Once a block reaches a light peer it commits the block onto its copy of the ledger and the information that is encapsulated by the transactions in that block is exchanged, i.e., the peers update their state of balances by appending their copy of the state database according to the new block that was added to the ledger. At this stage, there is a new state about the information peers in the network possess and consensus in the network is reached.

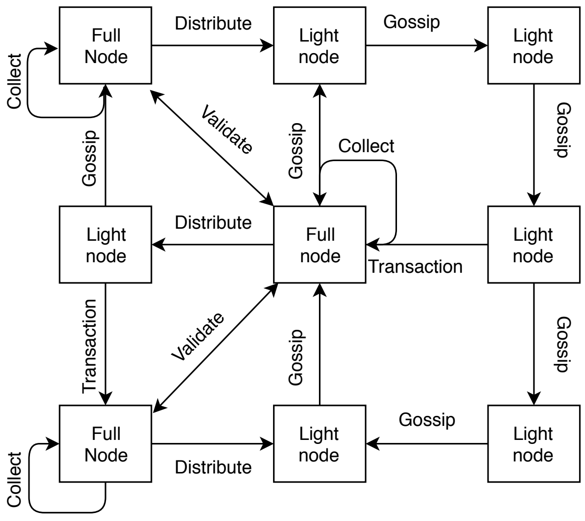

Figure 3.1: Peer to peer blockchain network.

Another mechanism of a blockchain network is *smart contracts*. Smart contracts were introduced in the popular blockchain network Ethereum[34]. A smart contract is application logic that can be deployed, by peers, onto a blockchain network. These smart contracts can incorporate any functionality that can be programmed within an application. When a peer wants to exchange a more complex piece of information, smart contracts are of use. For example, *peer X has invented an algorithm and has programmed it into an application.* peer X can then sell the use of this algorithm by deploying the smart contract on a blockchain network. Peers can than send transactions to that smart contract and if they add enough embedded assets in their transaction they can successfully interact with the smart contract and thus use the incorporate algorithm. With smart contracts, any piece of information can be exchanged that can be implemented in an application. In a broader sense, this enables blockchain networks to decentralize applications. With the possibility to deploy any piece of software onto a network, lots of opportunities arise[31].

There are two types of blockchain 1). an open model where peers can participate without registering (permissionless), and a model where peers have to register, i.e., acquire a certificate (permissioned). Popular permissionless blockchains are Bitcoin[20], Ethereum[34] and Ripple[24]. Permissioned blockchains are a more commonly used ramification of blockchain in business cases as opposed to permissionless blockchains because anyone can interact with a permissionless blockchain, which is often not desired for most business cases. The downside to a permissioned blockchain is that it requires an authority that handles these permissions. This authority can be seen as a third party because it is able to pull the strings in the network. Therefore permissioned blockchains and frameworks that build such networks are better applied when needing to decentralize applications via smart contracts and use the tamper-proof characteristic of the ledger to irrevocable state what has happened on the network.

3.2 Hyperledger Fabric framework

The Hyperledger Fabric framework uses the core blockchain technology semantics: peer, ledger, state database[1], transaction and smart contract[2]. It also introduces new semantics: channel, certificate authority and orderer, that are specific for the Hyperledger Fabric framework.

3.2.1 Components

The Hyperledger Fabric framework has multiple components that each have their own role in a Hyperledger Fabric network. Each Hyperledger Fabric network has a purpose which is simply put: *what is the network build and used for?* Such a purpose is often broken down into smaller services. Notice that *purpose* and *service* here are used as an analogy. It is not directly embedded as a property of a Hyperledger Fabric network but is used to describe the semantics of a Hyperledger Fabric network more easily for the rest of this thesis. A client is denoted as an actor that uses the Hyperledger Fabric network by sending transactions, i.e., wants to make use of a service presented by the network.

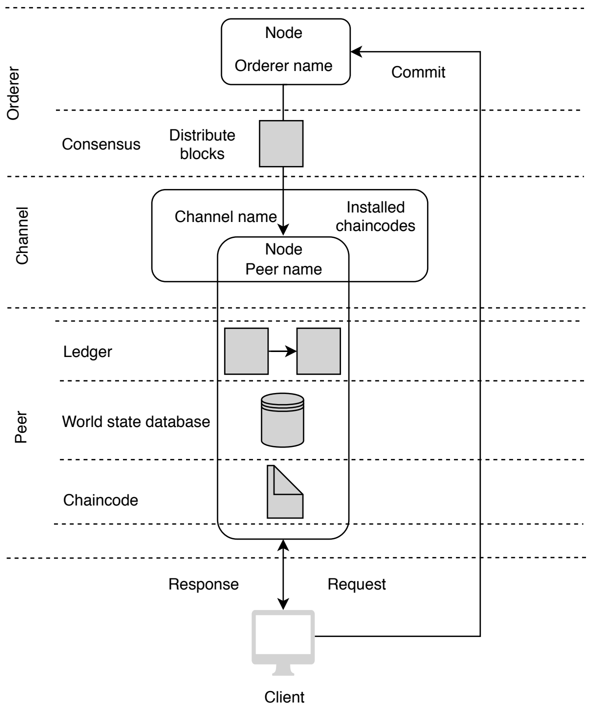

Figure 3.2: Abstract overview of composing Hyperledger Fabric components.

Certificate authority

By default, a Hyperledger Fabric network is a permissioned blockchain. The certificate authority regulates the access to the network. The regulation is achieved by distributed certificates. The certificate authority hands out digital identities to clients that want to interact with the network. Each digital identity has its own access control. Each interaction that is performed needs to be signed which is done by sending the digital identity of the client along with the interaction.

Peer

Each interaction that a client has with a Hyperledger Fabric network is via a peer (peer is a Hyperledger Fabric component and not another client). Peers in a Hyperledger Fabric network are vital, they make a Hyperledger Fabric network accessible by means that they expose an IP address such that clients can send interactions to a peer over gRPCs. Each peer is part of an organization. Organizations are introduced to separate peers, of the Hyperledger Fabric network, in consortia. Benefits arise in a Hyperledger Fabric network with organizations such as permission handling at once for a whole organization. When two or more organizations both have a share in providing a service for the Hyperledger Fabric network, they join a channel with at least one peer of their organization. Each peer in a channel keeps a copy of that channels ledger and its world state database. In addition, peers can install and instantiate chaincodes onto the channel or their stack respectively.

[1]Called world state database in the Hyperledger Fabric framework
[2]Called chaincode in the Hyperledger Fabric framework

Channel

When organizations, and therefore their peers, share a part in providing a service for the Hyperledger Fabric network they both join the same channel. A Hyperledger Fabric network has multiple services which together expose the purpose of the network. It is a common practice to instantiate a channel for each service. This means that one single Hyperledger Fabric network can have multiple channels and a peer can join multiple channels. Channels are mainly used for privacy, to build abstraction and separate concerns in a Hyperledger Fabric network. As said each interaction a client has with a Hyperledger Fabric network is via a peer. Since peers can join multiple channels and each channel has its own service, the client must always specify to which channel the interaction is bound. In more detail this says that *the client wants to make use of the service presented by channel x, the client, therefore, interacts with peer Y which has joined channel x.* Since interaction is bound to a channel, each channel has its own ledger, world state database and installed chaincodes. The channel' ledger and world state database are hosted by each peer that has joined the channel.

Chaincode

Chaincode is a component that represents application logic in a Hyperledger Fabric network. Chaincodes are the key to making the service of a channel within a Hyperledger Fabric network disposable. Peers are on its self a static component, they do not have any functionality other than keeping a copy of the ledger and world state database of the channels that the peer has joined. Although it makes them vital in making the network accessible, chaincodes are the actual application logic/ functionality of the service. The deployment of a chaincode by different peers is what makes the application logic in a Hyperledger Fabric network decentralized. Chaincodes can have two states, installed or instantiated. The installed state is channel bound, instantiated chaincode is peer bound. Chaincode must first be installed onto the channel by a peer. On installation the chaincode is packaged to a *ChaincodeDeploymentSpec(CDS)*. After a chaincode has been installed onto a channel, peers use the CDS to instantiate this chaincode onto their stack and make it run in an isolated environment.

Endorsement policy When a chaincode is instantiated, the peer must set an endorsement policy. endorsement policy is used to control the magnitude of importance of different chaincodes in a channel. On instantiate of a chaincode, the peer needs to specify which peers in the channel need to endorse a request transaction. The purpose of an endorsement policy is related to the peers in a channel not fully trusting each other. When a client sends an interaction to a chaincode it might be so that a malicious peer sends a different response than the other peers that get the same transaction. With the endorsement policy on a chaincode, Hyperledger Fabric handles these malicious responses for the client by returning to the client that the endorsement policy was not met for a transaction and that the client has to resent the transaction. What this also means is that the client is responsible for sending the transaction to the needed peers according to the endorsement policy[3].

Transaction

The client has an *interaction* with the peer by sending an encapsulated package with attributes, this is called a transaction. Each interaction a client has with a Hyperledger Fabric network is via a peer and is channel bound. Thus, *client x wants to make use of the service disposed by channel y, the client, therefore, encapsulates the needed attributes of his interaction in a transaction and sends this to peer z, which has joined channel y.* This is a very strong property of Hyperledger Fabric and the blockchain technology in general because this means each interaction clients have with a channel can be stored and be traversed. With the total set of transactions called the *ledger*, it is possible to replicate and see everything that has happened on that channel irreversible. A transaction passes multiple states during its transaction flow[4].

[3]If the endorsement policy of a chaincode specifies that one peer of each organization one and two need to endorse the request but the client only sends the transaction to the peer of organization 1 the endorsement policy will never be met.

[4]Note that the mentioned "states" here are used as an acronym for explaining the life cycle of a transaction more easily.

Request When the client sends a transaction to the peer, the transaction state is *request.* A peer decodes the transaction and executes the corresponding chaincode with the given arguments specified in the attributes set by the client on the creation of the transaction. During this phase the peer does the following operations

1. Check whether the client has valid permissions according to its digital identity to execute the chaincode.

2. Append the read-write set with the read- and write operations on the world state database that are incorporated in the chaincode.

The client sends the transaction to the necessary endorsing peers to get the required endorsement signatures, according to the chaincode' endorsement policy. After the execution, the peer sends the response to the client. An acronym for the endorsement signature is: *peer X has signed the transaction y with state [approved / disapproved].* The endorsement signature state is vital to what happens to the transaction in its later life cycle. Notice that during this phase the write operations on the world state database are not actually persisted. The actual write operations to the world state database are done at the commit cycle.

Response Once a request is processed by the peer it was sent to, it sends the chaincode response, read-write set and endorsement signature back to the client at which moment the transaction gets the status *response.* The client collects all the *responses* from the peers that it send the request to. The client builds a set of these responses and sends these to the orderer for the request to be committed.

Commit When the client has collected all the responses, it sends the collection of these responses to the orderer. The orderer checks whether the send responses are still valid e.g. the endorsement signatures are not tempered with etc., checks if the endorsement policy has been met for the chaincode that was executed by the initial request with the given collection of responses and checks whether the read-write set is still valid. If this is true the orderer labels the transaction as valid. Once the peers receive a new block of transactions from the orderer they loop over all the transactions in the block and execute only those transactions that have been labeled with valid. At this stage, the peers unpack the read-write set, previously build during the process cycle, and *commit* the write operations to their copy of the world state database. After the loop over the transaction in the block is done, the peer persists the block onto its copy of the ledger. Notice *all* transactions are stored on the ledger thus *valid* and *invalid.* As said the ledger is immutable and thus the transaction will be visible for the rest of the channels life cycle.

Orderer

The orderer collects all the transactions that have status commit send from the client. The orderer orders time chronically the transactions and label' the validity of the transaction. The validity of a transaction depends on whether or not the endorsement policy is met and if the version of the read-keys in the read-write set is correct. After which it adds the transaction to the currently open block. Once the block-limit; maximum amount of transactions in a single block or block-timeout; maximum amount of time for a block to have status open is met, it distributes the newly formed block to all the peers in the channel.

World state database

Each channel in a Hyperledger Fabric network has their own database called the *world state database.* This world state database is used for storing data in a channel. The data in a world state database is accessible from the chaincodes that are instantiated by that channel its peers, via read- and write operations. The world state database can be seen as an indexed view of the final value of each and every known key in a channel. Hyperledger Fabric ships with two default database types, the key-value structured LevelDB[10], and the JSON document structured CouchDB[7].

Double spending We said that when a client executes a chaincode he needs to obtain the number of endorsement signatures from the right peers, specified in the endorsement policy for his transaction to be valid. This process-commit cycle exposes the data in the world state database to the double-spending problem. Explained is what difficulties arise if data is persisted immediately after which is described how these difficulties are accounted for using the *read-write set* semantics.

Immediate persist
client A sends a transaction to chaincode 1. This chaincode incorporates read- and write operations on the world state database. At the same time, that client A has sent a transaction to chaincode 1, client B also sends a transaction to chaincode 1, therefore the transactions of both clients alter the same data. Client A fails to obtain the right amount of endorsement signatures which means: something went wrong during the processing of the transaction and the transaction is invalid. Therefore the integrity of the transaction client A has sent cannot be accounted for. But since the data to the world state database is already persisted, and client B has no notice of the failure of client A, the transaction of client B might have been processed with "corrupted" data. This can get very messy if, for instance, the service of the channel is an electronic cash system and the world state database stores account balances.

Read-write set
It would be a cumbersome process for Hyperledger Fabric to reverse does operations done by client A and notify client B that the integrity of his transaction is not valid. To make this easier, Hyperledger Fabric uses the read-write set semantics. During the processing cycle of a transaction, read- and write operations on the world state database are stored in a *read-write* set. A read operation consists of the key that has been read and the version number of that key which is stored in the world state database. Write operations consist of the key and the value that should be written to the world state database. Give very clear notice here that *write operations are thus not yet persisted to the world state database store on process cycle.* Once a transaction reaches the commit cycle, the orderer will check, next to checking the endorsement signatures, the read-write set. The orderer checks whether the version of the read-keys is the same as the one in his copy of the world state database. The orderer will validate the transaction if this is true. If peers receive a new block they will increment the read-keys version numbers with one and execute the write operations. Since the orderer checks whether the read operations are valid and therefore labels the transaction as valid. And since peers only execute a valid transaction, the integrity of the read and write operations can be accounted for.

Applying this to the example. The read and write operations of client A are stored in the read-write set. The same goes for client B. Since the transaction of client A is invalid, no data in the world state database has been altered yet, with persuasion it can be validated that the transaction of client B has been processed on valid data and therefore the order labels the transaction of client B as valid.

Ledger

Each channel has its own ledger. The ledger is a log of all the transactions that have been sent to a channel. The ledger is separated into blocks, these blocks contain the transactions that have been sent to the peers of that channel. In Figure 3.3, a graphical representation of a ledger is given. As said every interaction a client has with the Hyperledger Fabric network is via a peer with a transaction. That means that the ledger is a track record of every operation that has been done during the life cycle of that channel. Since every read- and write operation is encapsulated in a transaction, the world state database can always be constructed from ledger because all the transactions and their read- and write operations are stored on the ledger. This is a very important property of a blockchain network.

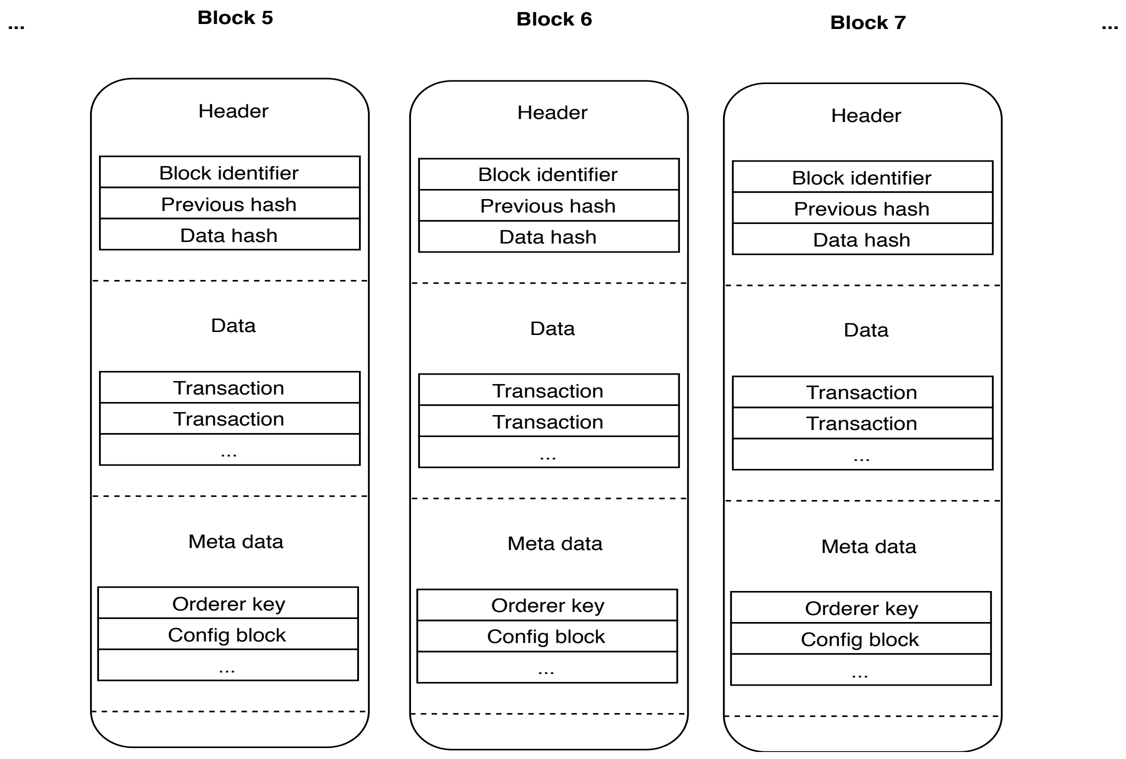

Figure 3.3: Representation of a ledger of the Hyperledger Fabric framework.

3.2.2 Compose

Figure 3.2 presents a composition of the individual components of a Hyperledger Fabric network. The peer has instantiated chaincodes and hosts the world state database and ledger of the channel. The channel has a list of installed chaincodes which peers can instantiate. The client sends a request transaction to one of the peers in a channel and specifies which chaincode the transaction is meant for. After executing the transaction the peer sends a response. Next, the client collects the responses and sends these to the orderer for the transaction to be committed. The orderer checks the validity of the response and embeds it in the current open block, after the block- timeout or size has passed or is full, the orderer distributes the block to the peers in the channel. The peers append the transactions of the block to their ledger and execute the write operations according to the read-write set of each valid transaction.

Containerization

To deploy a Hyperledger Fabric network, Hyperledger Fabric utilizes Docker. Docker is a containerization platform that containerizes prefabricated images into an isolated environment. Each component in a Hyperledger Fabric network is containerized via Docker. Via a YAML specification, different properties can be set on top of the prefabricated image to configure the image as desired. In this research, Docker v18.0.0 Community Edition[14] is used. To distribute Hyperledger Fabric components over multiple nodes and establish communication between the components, the following techniques are used. With Docker swarm, an overlay network is built to establish a communication line between the nodes via their IP addresses. Next, in the YAML specification, a constraint is set such that a certain component will only be deployed on a specific node that is part of the swarm. These constraints guarantee that the components of the peers in organization one are deployed on node A. And deploy the peers of organization two onto node B.

3.2.3 Use case

To get a better understanding of the Hyperledger Fabric composition, an example of how the *module of reimbursement* can be decentralized with a Hyperledger Fabric network is given. Figure 3.4 presents a possible architecture of a Hyperledger Fabric network which incorporates the module of reimbursement[5]. The purpose of this Hyperledger Fabric network consists of multiple modules of which one is the *module of reimbursement*. Each service gets its own channel to obtain abstraction. A definition for the service of the channel which is used for the module of reimbursement can thus be *provide the needs for processing claims to the insured individuals affiliated with this insurance company*. Each organization, and therefore their peers, that have a share in providing those needs will join the channel. A need for the module of reimbursement is the application logic to process a claim. Application logic is utilized via chaincodes. Each peer in the channel instantiates the chaincode and therefore the module of reimbursement will be decentralized. Insured individuals then, in turn, encapsulate the attributes, of their claim, in a transaction and send it to the peer for it to process the transaction, and thus the claim, against this application logic. The peer sends the transaction back to the client after executing it with a response of the result of the processing. The client sends this response to the orderer for the transaction to be committed and thus commit the claim.

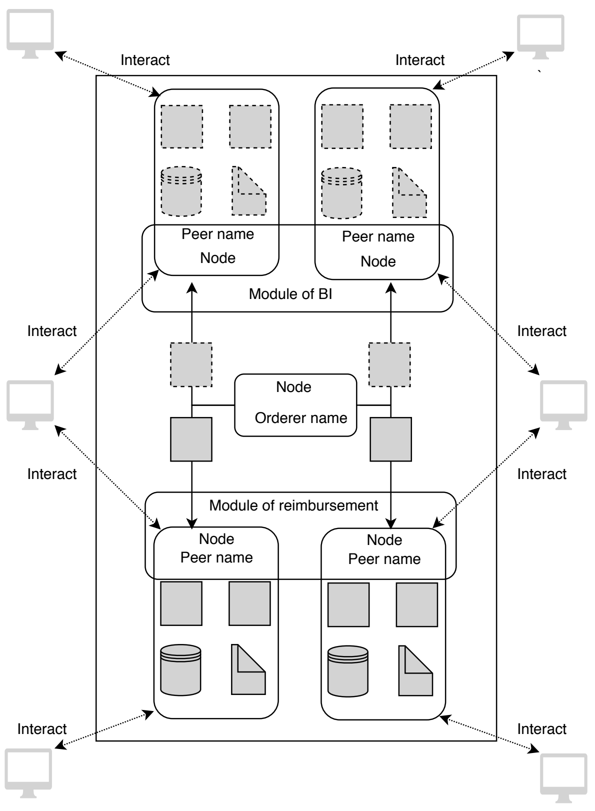

Figure 3.4: Appliance module of reimbursement in a Hyperledger Fabric network.

3.3 Denormalise data

The data that is stored in the centralized version of the module of reimbursement has a relational structure. The Hyperledger Fabric framework only supports either a key-value structure or JSON document structure. In this thesis, the relational structure is mapped to a key-value structure. This is done by identifying every insured individual by a unique code followed by a unique health care action code. The relational structure can be denormalized to a key-value structure via this mapping. Although it will result in a significant amount of key-value entries and overlap between the keys their values. This option is discussed with multiple employees of Oracle Health Insurance and we conclude a numeric key of 12 characters is needed to be able to store all the needed keys for a single health insured individual. The value needs to be a 4 character numeric value.

[5]Note that *interacts* here are the request and response cycle of a transaction. The commit cycle is omitted in this figure for readability.

Chapter 4

PatriciaDB

This Chapter describes how improved read- and write operations on the world state database is achieved. Data structures affect the time of read- and write operations [9]. Therefore a self-made key-value world state database is built. This self-made world state database incorporates a special-optimized data structure for our use case called the Patricia Trie. Theory and motivation behind the Patricia Trie are described and how it theoretically will accomplish improved read- and write operations. At last, the implementation of this data structure in an application is given and shown how it is deployed on each peer' stack such that it funds as the world state database in a Hyperledger Fabric network.

4.1 Patricia Trie

A Patricia Trie is a special type of data structure which ensues from applying the *"Practical Algorithm To Retrieve Information Coded in Alphanumeric"*(PATRICIA) [17] algorithm for read- and write operations to an object. It is a commonly used data structure for *IP routing* and *text auto-complete functions*[30] and it is a ramification of the *Compressed Radix Trie*.

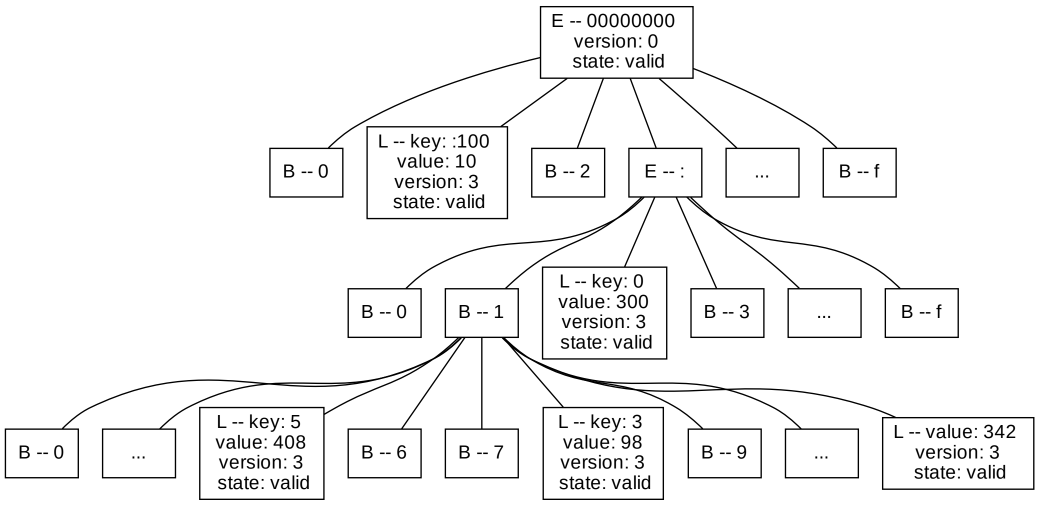

Figure 4.1: Representation of a Patricia Trie for set S.

4.1.1 Theory

A standard implementation of the Patricia Trie consists of nodes and it is constructed from a key-value entry set Each node has a key and a value attribute. The node key is a list of characters which are overlapping between multiple entry keys. The value of a node can either be the terminating end value or, be a node and traversing deeper into the data structure is needed to reach a terminating end value. There is one drawback with the standard implementation of the Patricia Trie. When there is a very small set of overlapping entry keys, it can become inefficient due to the fact that a lot of nodes exist that have node keys with a single character. Ethereum builds a modification of the Patricia Trie [5] to cope with small overlapping entry keys in a set and it is MIT licensed. It is therefore that this code is chosen as the base of the PatriciaDB implementation. We add our own modification to make the Patricia Trie applicable with the Hyperledger Fabric read-write set semantics. The analogy *Patricia Trie* is used with the definition: a Patricia Trie with our self made modification on top of the Ethereum modification. The analogies *slice*, *entry key*, *mutation key*and *node key* are defined as follows.

- Slice[1]: a list of characters.
- Entry key: a key that is part of the entry set from which the Patricia Trie is built.
- Mutation key: the slice that needs to be read- or written to the Patricia Trie.
- Node key: a partial slice which is the key for a Patricia Trie node.

The lead words *overlap*, *single*, *remaining*, *following* and *preceding* are defined with the description.

- Overlap: the correspondence between two slices that have the same character at each position.
- Single: the first character after an overlap.
- Remaining: the remaining slice following after an overlap.
- Following: the remaining slice following after an overlap *minus* the first character of the list.
- Preceding: the preceding slice before a single character or remaining slice.

During a read or write operation on the Patricia Trie, at each traverse; a level deeper down into the structure, the preceding node key from the mutation key is removed. This results in the mutation key becoming smaller and smaller until there are no slices left and the mutation key has either been read or written successfully to the Patricia Trie. A node in a Patricia Trie is one of the following types:

1. Leaf node: an array with $[key, value, version, state]$ attributes where the key is the remaining mutation key up to this stage of the traverse and the value is the 4-byte terminating end value. Version and state will be described later in section 4.1.1
2. Extension node: an array with $[key, value]$ attributes where the key is the preceding mutation key up to this stage of the traverse and the value is the remaining mutation key which is used for the upcoming node key.
3. Branch node: an array with 17 $[key]$ attributes. The keys in the Patricia Trie are hexadecimal encoded and therefore the single mutation key represents the position under which the node with the following node key will be stored.

[1]We use the word *slice* here instead of *string* since the use of *string* suggests a complete string. Here it is described as a part of a string i.e. a *slice*.

Example

$S = \{$ (000000001:100, 10), (000000003:155, 408), (000000003:183, 98), (000000003:1, 342), (000000003:20, 300) $\}$

In Figure 4.1 a Patricia Trie is presented for set S with leaf, extension and branch nodes expressed with the labels L, E, and B respectively. A description can be given as follows. The largest overlapping entry key in the set S is <00000000>. This becomes an extension node with node key <00000000>. The single mutation keys are <1> and <3>, so a branch node is constructed to store the following node key at spot 1 and 3 respectively. The mutation key <000000001:100> is terminated with a leaf node which has a remaining node key <:100> and value 10. At the spot of single node key <3> an extension node is added which holds single node key <:> because there are multiple single overlapping mutation keys after this node namely <1> and <2>. For the single mutation key <1>, there are multiple single mutation keys: <5>, <8> which both terminate at a leaf node with the remaining node key <5> and <3> and the terminating values 408 and 98 respectively. It also terminates at the <f> node key which means the preceding single mutation key <1> itself has a value. The single mutation key <2> terminates at a leaf node with the remaining node key <0> and the terminating value 300.

Self added modification

We implement self-made world state database and the double-spending problem should be prevented in this implementation. Therefore a self added modification is made to the Patricia Trie. In a Patricia Trie only leaf nodes store terminating end values. Each read- and write operation on the Patricia Trie will end at a leaf node[2]. The extension- and branch node are added to utilize the PATRICIA algorithm. Therefore the leaf nodes terminating values are exposed to double-spending. To prevent this from happening, each leaf node gets two extra attributes *version* and *state*.

Version Each leaf node gets a version attribute. The version number is returned by PatriciaDB along with the value at each access to its value by a read operation during the process cycle of a transaction. Once the transaction is in the commit cycle the orderer will check if the version numbers are corresponding. If this is the case, the orderer will validate the transaction, if this is not the case, this means that the processing of the transaction used data that was not consistent, the orderer will invalidate the transaction. This is the same mechanism as Hyperledger Fabric enforces for their default LevelDB implementation. Since PatriciaDB is not yet fully embedded in the Hyperledger Fabric framework due to time limitations of this thesis this is in a hypothetical form. Therefore the endorsement-commit cycle with transactions that read- or write to PatriciaDB is not yet fully enforced but does enable PatriciaDB to become workable with the read-write semantics if it would be fully embedded in Hyperledger Fabric framework.

State Each leaf node also gets a state attribute. The Patricia Trie is a tree structure and therefore nodes are dependent on each other. If a new leaf node is inserted into the Patricia Trie during a write operation in the processing cycle, the state attribute is set to *invalid*. Whenever this transaction is sent to the commit cycle, the state will be set to *valid*. This enables for writes to be done during the processing cycle while being able to preserve double-spending. On each read operation, the state attributed is checked whether it is valid if this is the case, a value is returned. If this is not the case, the value is not returned although one was found, because the transaction that did the write operation for the node was not validated. With this mechanism the Patricia Trie object can become flooded with *invalid* nodes. It would require a "cleaning function" to delete the invalid nodes at a certain point in time. This function has yet not been implemented due to time limitations of this thesis.

[2]Except for a read operation with a key that does not exist in the set which will be returned with *null*.

4.1.2 Motivation

The usage of the Patricia Trie as underlying data structure for the self-made world state database is motivated based on the following facts: the entry set will be of significance and the keys will have overlap. The *Patricia Trie* is a data structure that has characteristics that supports such needs.

Entry set significance

As stated in the initial publication of the PATRICIA algorithm by Morrison[17]: *"The amount of computation required to find one occurrence of a key, if there is one, or to find that there is none, has a bound which depends linearly on the length of the key and is independent of the size of the library."* The Patricia Trie has the characteristic that read operations are linearly bound to the maximum length of the key k, in the key entry set. In our implementation, the maximum key length is 12 bytes and keys are hexadecimal encoded. For any size of our entry set, the reading operation worst case scenario will take no more than $O(k)$ computations. The write characteristic of the Patricia Trie is also interesting. It takes $O(k + a)$ operations [16] to insert a key in the Patricia Trie where k is the length of the key and a the size of the alphabet. The alphabet is of size 10 in our use case since keys only consist of numeric values. This means that inserting is also not bound to the size of the entry set.

Key overlap

The maximum amount of nodes is equal to the total entries of the set that the Patricia Trie represents [28]. This only happens when there is no overlap in the keys whatsoever. Which is highly improbable in our use case as established in section 3.3. Also, this would undermine the whole purpose of a Patrica Trie since its main utility is that it is constructed from keys in the entry set. The space complexity for a Patricia Trie worst case scenario is $O(nN + M)$. where n is the number of entries stored in the Patricia Trie. N the size of the alphabet and M the total length of all entries. This means that a large portion, if not the whole Patricia Trie can reside in memory and *swapping* to disk won't take place. Swapping would decrease the operation time significantly.

4.2 The application

The layers that enable the Patricia Trie to fund as underlying data structure for the self-made world state database is described. The analogy Patricia Trie object is used with the following description: a Plain Old Java Object to which read- and write operations are done according to the PATRICIA algorithm. The application is split into three layers: *service*, *application* and *persistence*.

Service

The service layer is used to expose, the procedures *read* and *write* to the outside. Each procedure has its own REST endpoint and is build using the Jersey Framework[3]. The responsibility of the service layer to just expose the functionality to the outside enables the underlying application layer to have different deployment types.

Application

In the application layer, there is a singleton Patricia Trie object. Each execution of either a read- or write operation, the application layer will check whether there is a Patricia Trie object present. If this is not the case it will use the persistence layer to retrieve the latest Patricia Trie object from a data store. This ensures that there is always one instance of the Patricia Trie at run-time. Next, the read- and write procedures according to the PATRICIA algorithm are implemented. A read operation requires just a string object which represents the entry key that needs to be read. A write operation requires a string object which represents the entry key and a string object which represents the value. In Algorithm 1 and 2 the *read* and *write* procedures are described in pseudocode.

Algorithm 1 Pseudocode for a read operation in the PatriciaDB application layer.

```
0: procedure READ(Object node, Byte[] key)
  if key length is 0 then
    return node
  end if
  currentNode ← objectToNode(node)
  if key is not found then
    return null
  end if
  if currentNode is LEAF or EXTENSION then
    if overlap found between the key and currentNode.key then
      return read(currentNode.value, remaining key)
    else
      return null;
    end if
  else
    currentNode is BRANCH
    fromBranchNode ← getNodeFromBranchAtIndex(key[0])
    return read(fromBranchNode, remaining key)
  end if
  end procedure=0
```

Algorithm 2 Pseudocode for a write operation in the application layer

```
0: procedure WRITE(Object node, Byte[] key, Object value)
  if key length is 0 then
    return value
  end if
  if node is not found, build new node then
    Object [] newNode ← new Object[] (key, value)
    return newNode
  end if
  currentNode ← objectToNode(node)
  if currentNode is LEAF or EXTENSION then
    if key already exists, update value then
      Object [] newNode ← new Object[] (hexadecimalToBinary(key), value)
      return newNode
    end if
    if currentKey is overlapping with key then
      newRoot ← write(currentNode.value, remainingKey, value)
    else
      currentKey is overlapping but node needs to be expanded to BRANCH
      Object[] branchNode ← emptyBranchNode(17)
      branchNode[k[matching]] ← write("", remaining currentNode.key, currentNode.value)
      branchNode[key[matching]] ← write("", remaining key, value)
      newRoot ← branchNode
    end if
  else
    the currentNode is of type BRANCH. return newNode[key[0]] = write(currentNode.value, re-
    mainingKey, value)
  end if
```

Persistence

The Patricia Trie object is run-time bound since it runs within a Tomcat server deployed in a Docker container. When the Tomcat server or Docker container fails due to an exception or system crash, the Patricia Trie object is flushed and all data written to it will be lost. Therefore the Patricia Trie object is persisted on each write into a data store. When the Tomcat server or Docker container restarts, the latest Patricia Trie object will be persisted to the data store and all the data that was written in the run-time cycle can be used in the subsequent run-time cycle. The persistence of the Patricia Trie object is done asynchronously and does not affect the performance of write operations on the Patricia Trie object.

4.2.1 Deployment

The application is deployed as a REST API. A WAR file is built which is deployed onto a Tomcat server. Next, the Tomcat server runs in a separate Docker container. Each peer deploys this Docker container alongside the other Docker containers which deploy the Hyperledger Fabric components. At this stage, PatriciaDB can be used as a world state database by each peer. Read- and write operations are done via local HTTP request send from the instantiated chaincode. It is important that each transaction is endorsed by each peer in a channel since this ensures that read- and write operations are done on each copy of PatriciaDB and therefore the same data is stored in the self-made world state database.

Chapter 5

Approach

This Chapter describes how the Hyperledger Fabric network is decentralized over different nodes and what type of hardware is used. Also, the methodology of the measurement phase is described. This research focuses on the request and response cycle of a transaction since this, together, is the transaction processing- and round-trip time. The commit cycle of the transaction will therefore not be taken into account during the measurement phase. The analogies *read transaction* and *write transaction* are defined with the following description: a transaction that is sent to the network which executes a chaincode that incorporates a read- or write operation on the specified world state database.

5.1 Experimental setup

The following two facts are important in this thesis: *the hardware must be commodity* and *the Hyperledger Fabric network needs to be decentralized.* The specifications of the system hardware are described and how the Hyperledger Fabric network is set up such that these facts are accounted for.

Commodity system hardware

Oracle Health Insurance has different customers that vary in different sizes, therefore commodity system hardware on which the network is deployed must be used such that the results of this thesis are applicable to each and every customer of Oracle Health Insurance. Oracle provides three nodes within different data centers based in Salt-Lake, Denver and Atlanta. The client is physically based in Utrecht. The nodes are only accessible via the corporate proxy of Oracle. Therefore each interaction with the nodes will pass through the proxy server which is based in Romania. The client is connected to the network with an Ethernet cable. Due to the decentralized setup, network delay will come to play. It is an ancillary of decentralized systems and it is taken into account during the analysis.

	OS	CPU	RAM	DISK
Node 1 - Denver	Oracle Linux 7.4	Intel(R) Xeon(R) X5670 @ 2.93GHz Dual Core	8GB	32 GB SSD
Node 2 - Salt-lake	Oracle Linux 7.4	Intel(R) Xeon(R) E5-2699 v3 @ 2.30GHz Dual Core	8GB	32 GB SSD
Node 3 - Atlanta	Oracle Linux 7.4	Intel(R) Xeon(R)E5-2699 v3 @ 2.30GHz Dual Core	8GB	32 GB SSD
Client - Utrecht	Windows 10	Intel(R) Core(TM) i7-7700HQ @ 2.80Ghz Quad Core	8GB	256 GB SSD

Table 5.1: System hardware setup.

Decentralized Hyperledger Fabric network

The core characteristic of the Hyperledger Fabric framework that is utilized in this research is *decentralization*. A graphical representation of the Hyperledger Fabric network and the decentralization of its components over the nodes is shown in Figure 5.1. Two organizations are defined; one is distributed in Salt-lake and one in Atlanta. The orderer in the network is based in Denver. Each organization deploys one peer which both join the same channel. On this channel are two installed chaincodes. Chaincode *CC1* and *CC2* are implemented such that they execute read and write operations on LevelDB and PatriciaDB during read- and write transactions respectively. The endorsement policy enforces that a transaction has to be endorsed by both organizations. Both peers, therefore, instantiate the chaincodes. This setup gives us all the needs to fully test Hyperledger Fabric in a decentralized manner.

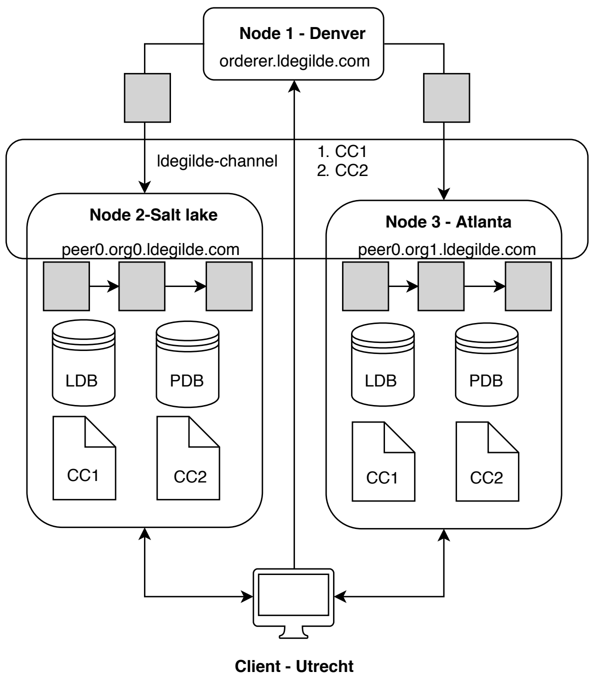

Figure 5.1: Hyperledger Fabric components scattered over the nodes.

The mandatory configurations *block limit* and *transaction time-out* are set as follows. The block limit- is 50 transactions and the transaction time-out is 60 seconds. A block limit of 50 transactions is chosen since the client application sends 50 transactions at a time. Setting the block limit to a particular value is important because this will reduce the block latency. Block latency comes to play during the *commit* cycle of transactions, on which this thesis does not focus. Though it will help speed up the processes of committing transactions and eventually speed up the overall measurement phase. Notice that it has no effect on the results of our measurements. The transaction time-out time is 30 seconds, this is standard practice in Hyperledger Fabric.

5.2 Methodology

We set up the Hyperledger Fabric network as described in Figure 5.1. The goal is to reach a total of 100.000 entries for each world state database to be able to do analysis on a proper measurement set. Each read- or write transaction is send to *both* peers. A single read- or write transaction will result in two measurements(one for peer 1 and one for peer 2). This is necessary due to the endorsement policy that is set for the chaincodes. Due to the endorsement policy, a transaction requires to be endorsed by both peers. Also, this enables us to analyze the performance of both peers since this data is generated. The following steps are taken to generate measurements which are used for our analysis.

1. Send 100.000 write transactions on LevelDB to both peers.
2. Send 3.000 read transactions on LevelDB to both peers.
3. Rebuild the Hyperledger Fabric network to clean all previously persisted data.
4. Send 100.000 write transactions on PatriciaDB to both peers.
5. Send 3.00 read transactions on PatriciaDB to both peers.

After the LevelDB measurements, the network will be cleaned such that the hardware and Hyperledger Fabric network setup is the same for both measurement phases. A multi-threaded client application is build to send read- and write transactions on LevelDB and PatriciaDB. The client application is built in Java and the communication with the Hyperledger Fabric network is build using the Hyperledger Fabric Java SDK[6]. In the client application, the thread count is set to 50. This means that a batch of 50 read- or write transactions is sent in parallel to the network at a time. Transactions are sent until the desired amount are reached according to Table 5.2. Write transactions are done as is. The key-value data is generated and marshaled in a transaction. Once the transaction arrives at the chaincode, it executes a write operation to the respective world state database and the key-value will be appended to the read-write set in case of LevelDB and persisted in the case of PatriciaDB. For read transactions, a random key is selected that has been written to the respective world state database and is used as the read key. The value that has been returned by the network is checked against the value that was written to the key ensuring data correctness. Each key consists of an 8 character numeric value followed by delimiter <:> and ended with a 3 character numeric value. Each value consists of a 4 character numeric value. The keys stored in LevelDB are automatically hashed by the Hyperledger Fabric framework via SHA2. Therefore the key and value size accumulated will be of size 68 bytes per entry. The data that is stored in the PatriciaDB won't be encrypted and therefore the key and value size accumulated will be of size 16 bytes per entry.

	Amount of Entries	Accumulated Quantity	Write operations	Read operations
Step 1	1000	1000	1000	1000
Step 2	9000	10.000	9000	1000
Step 3	90.000	100.000	90.000	1000

Table 5.2: Quantity of read- and write operations per step

Transaction timing

In Figure 5.2 the timing steps are shown for a transaction' life cycle during the measurement phase. Each start and end of an arrow is a timing point. A notice needs to be given that more steps can be measured in PatriciaDB as opposed to LevelDB. Since LevelDB is embedded in the Hyperledger Fabric platform and PatriciaDB an application that funds as a world state database but is not fully integrated into the Hyperledger Fabric platform. The light blue columns represent mutual measuring points for LevelDB and PatriciaDB measurements. The dark blue columns are only measured during the PatriciaDB tests. Although more can be measured in PatriciaDB, during the analysis of the results it is made certain that the same semantics are compared.

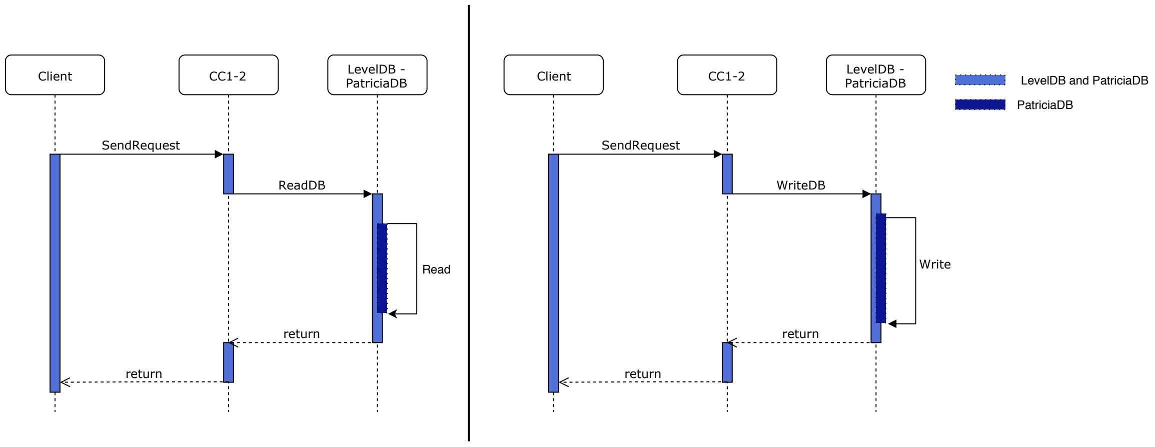

Figure 5.2: Timing diagram for a single read- and write operation.

1. (LevelDB and PatriciaDB): *SendRequest*: how long does it take for a read- or write transaction that is sent from the client to reach the network and for the network to send a response back

to the client?

2. (LevelDB): *ReadDB*: read operation time on LevelDB
3. (LevelDB): *WriteDB*: write operation time on LevelDB
4. (PatriciaDB): *Read*: read operation time on PatriciaDB *without* the HTTP overhead?
5. (PatriciaDB): *Write*: write operation time on PatricaDB *without* the HTTP overhead?
6. (PatriciaDB): *(ReadDB + WriteDB) / 2*: what is the HTTP overhead for reaching the PatricaDB REST API?

We have and will use the analogy write operation on LevelDB to describe a write operation on LevelDB. But a more detailed description of what happens at a write operation on LevelDB during the processing cycle of a transaction needs to be given to correctly understand what is measured here. This research focuses on the processing cycle of a transaction. During the processing cycle, the read-write set is appended and no write operation on LevelDB is done*yet.* Thus if a write operation is done on LevelDB, the measurement tells us how long the appending of the read-write set takes. This is the same semantic step as when a value is written to PatriciaDB and therefore a correct measure for this research.

Time-frame calculations

We have said that at least 50 transaction per second throughput needs to be reached to reach feasibility for the module of reimbursement to be decentralized with the Hyperledger Fabric framework. The client application sends a batch of 50 transactions in parallel at a time(due to the multi-threaded setup). If this batch gets back to the client within one second, a feasible transaction throughput is reached. Therefore we establish that the total transaction time is $1000ms$ when sending a batch of 50 transactions in parallel to the Hyperledger Fabric network.

The chaincodes do not fully incorporate the well-defined application logic normally needed to process a claim successfully by the module of reimbursement. But a time-frame needs to be reserved for the computations that the computations of the well-defined application logic would use if it would be implemented in a chaincode. The centralized version of the application logic is measured and concluded that on average it takes $150ms$ to do the necessary computations. Thus, with the information that a batch of 50 transactions sent in parallel should not take more than $1000ms$ and the reserved time-frame for the well-defined application logic is $150ms$: the remainder time-frame for the transaction round-trip time and 40 read- and 10 write operations on the world state database, is $850ms$.

Chapter 6

Results

The results of the measurement phase are analyzed. A quantitative and qualitative analysis of the transaction round-trip time, transaction processing time and total transaction time is presented.

6.1 Transaction round-trip time

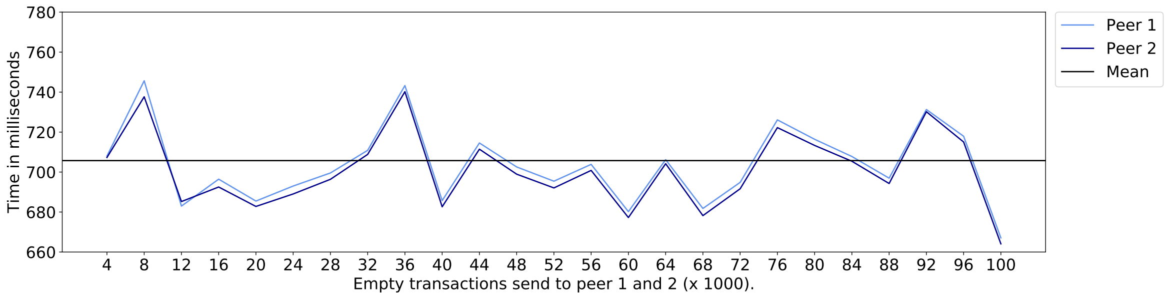

Figure 6.1: Tranasction round-trip time per node.

Figure 6.1 presents the transaction round-trip time for peer 1 and 2. Transaction round-trip time is defined as follows: the time an empty transaction takes[1], that is sent from the client, to reach the network and for each peer in the network to send a response back to the client. Recall that the client has to collect the transaction responses from the peers(Section 3.2). This means that the transaction round-trip time is dependent on the slowest peer since the client has to wait until all peers have sent back a response before being able to pack the responses in a set and send them to the orderer. The measurements are separated per peer such that the influence of the geographical location of the nodes(on which the peers are deployed) on the transaction round-trip time can be studied. The difference of the transaction round-trip time between the peers is 0.584%. It falls within the 5% statically significance which makes the difference negligible. Since the difference is negligible, we observe that the average transaction round-trip time imperatively of which peer is 706.555ms.

The transactions are sent over the network and therefore one could argue that the transaction round-trip time is nothing more than network delay. To neglect this argumentation, a ping is sent to both nodes in the network from the client node. This measurement represents the network delay and is set to 134ms. Subtracting the network delay from the transaction round-trip time leaves us with a time-frame of 572.555ms. This is the time-frame for the client to marshal the attributes of a transaction by the SDK, unpack the transaction by the peers on arrival and for the peer to marshal the response attributes and send the response back to the client. Network delay is an ancillary of a decentralized application and therefore it is kept within the transaction round-trip time.

[1] A transaction that is sent to a chaincode that does not incorporate any computations

The total transaction time should not take more than 1000ms for a batch of 50 transactions send in parallel to reach a feasible transaction throughput for the module of reimbursement to be decentralized with the Hyperledger Fabric framework. The transaction round-trip time is set to 706.555ms, therefore the time-frame for the transaction processing time for a batch of 50 transaction send in parallel is 1000ms - 706.555ms = 293.444ms.

6.2 Transaction processing time

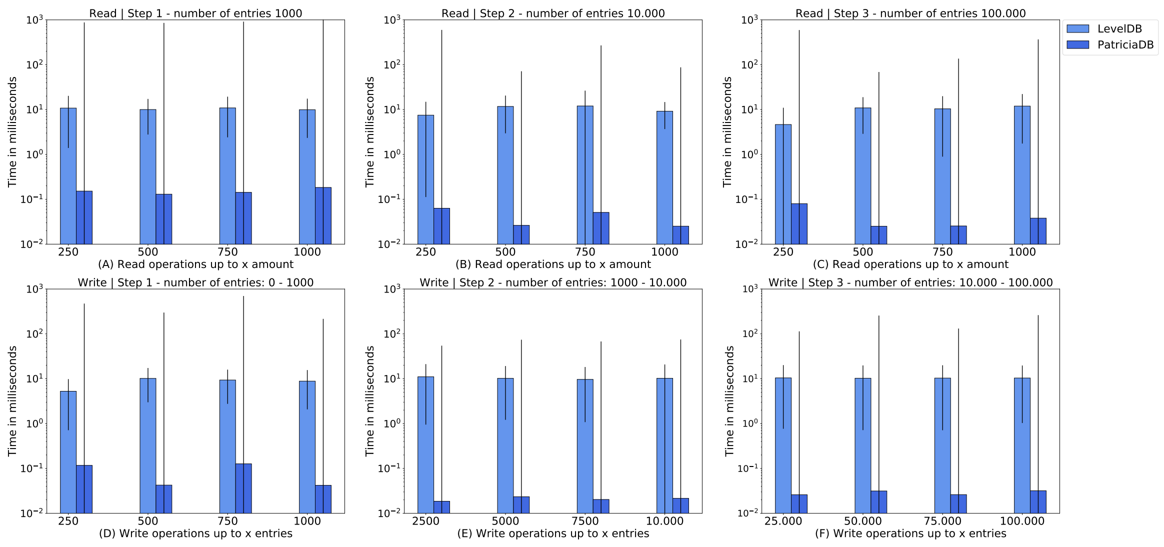

Figure 6.2: Read- and write operation time for LevelDB and PatriciaDB per step.

Figure 6.2 (A), (B) and (C) show the amount of time the read operations take for both LevelDB and PatriciaDB. Figure 6.2 (D), (E) and (F) show the amount of time it takes to complete a write operation. Each graph is a representation of a step as described in Section 5.2. Each bar has a standard deviation line showing the differentiation between the measurements. To analyze PatriciaDB and LevelDB, three characteristics are introduced.

a) *Performance:* is defined by the time a read- or write operation completes.

b) *Scalability:* is the entry set size of influence for the performance in PatriciaDB and LevelDB regarding the read- and write operations time.

c) *Stability:* investigate if the read- and write operation time have variance.

Performance and scalability are intuitive characteristics to compare LevelDB and PatriciaDB. Stability is introduced as a characteristic here since Figure 6.2 is plotted with the average of a grouped set of measurements. This visually flattens out the variance that occurs in the measurements for both LevelDB and PatriciaDB. To thoroughly compare LevelDB and PatriciaDB, the variance of both world state databases is also taken into account. These three characteristics are the most vital characteristics of a world state database for our use case. Comparing the differences between LevelDB and PatriciaDB according to these characteristics gives us simple metrics to state if PatriciaDB is more sufficient than LevelDB for our use case. The characteristics are addressed according to Figure 6.2.

Performance The performance of LevelDB and PatriciaDB is stipulated according to the read- and write operation time. With the answer to this characteristic, we want to establish whether PatriciaDB has faster read- and write operation times as opposed to LevelDB. According to the measurements on LevelDB and PatriciaDB, the following can be said.

a) *LevelDB:* average read- and write operation time is 10.084ms and 10.342ms respectively

b) *PatriciaDB:* average read- and write operation time is 0.076ms and 0.042ms respectively

Regarding the read- and write operations time, PatriciaDB is 136 and 246 times faster as opposed to LevelDB respectively. In our opinion, this means that PatriciaDB is significantly faster in read- and write operations than LevelDB.

Scalability Each world state database is analyzed individually. This analysis is used to discuss whether LevelDB or PatriciaDB scales better over a larger entry set for both read- and write operations. To analyze each world state database individually, graphs (A), (B) and (C) that depict step 1,2 and 3 for read operations are used. And (D), (E) and (F) which depict step 1, 2 and 3 for write operations are used. The largest fluctuation between two steps is calculated and focused to argue the scalability of LevelDB and PatriciaDB.

We analyze the read-operations of LevelDB according to the graphs in the first row of Figure 6.2. Read operations on LevelDB are stable and the largest fluctuation(11%) between the steps is for step two and three. Read operations on LevelDB are faster in step three as opposed to step two. A counter-intuitive observation since a larger entry set would suggest a slower read- operation time. No grounded argumentation why this happens is found. Although such argumentation might not be necessary since the fluctuation is rather small and it does not influence the analysis. The write operations on LevelDB are stable and the largest fluctuation(3%) between the steps is for step one and two. The difference is statically significant which makes it negligible.

The read operations on PatriciaDB are analyzed. The largest fluctuation(367%) between the steps is for step one and two. Here we see that step two is faster than step one which is counter-intuitive since a larger entry set would suggest a slower read operation time. We suspect that the read operations in step one required more look-ups over nodes in the Patricia Trie as opposed to step two. If for example, a read operation with a certain key requires the lookup of nine nodes in step one. And a read operation with a certain key in step two requires the lookup of three nodes. Although the entry set size does not influence the read operation time. The look-up of three times the extra nodes due to the different key substantiates why the read operation in step two is three times as fast as in step one. The largest fluctuation(405%) of write operations on PatriciaDB between the steps is for step one and two. In step one the Patricia Trie object is at its initial state, there are yet no nodes in the Patricia Trie object. Therefore the first writes to the object require for a lot of node creations which takes time. It is for this fact that the write operations are the slowest in step one.

Although PatriciaDB has larger fluctuation between the steps than LevelDB, the performance in PatriciaDB does not downgrade when the entry set size increases. This also goes for LevelDB. Therefore LevelDB and PatriciaDB are equally scalable regarding the entry set size.

Stability the analysis between LevelDB and PatriciaDB is done based on the average of a grouped set of measurements. This allows for analysis over the large measurement set but also visually flattens out the deviation in the measurements. It is therefore that the stability of LevelDB and PatriciaDB is analyzed. Stability here thus means: are there heavy outliers in the measurements? The standard deviation line for the LevelDB measurements shows a marginal fluctuation for read- and write operation time. Therefore LevelDB is labeled as stable. The standard deviation line for PatriciaDB shows a much larger standard deviation for both read- and write operation time. A more close analysis of the raw measurements shows that these deviations happen on a non-frequent base (2.5% of measurements). We suspect that since PatriciaDB is written in Java, a garbage collection cycle happens to clean the heap space. This action is a blocking call making all operations having to wait until it is done. Thus LevelDB is a more stable solution regarding time variance as opposed to PatriciaDB. Although PatriciaDB is less stable in its measurements, it does not influence the performance of PatriciaDB.

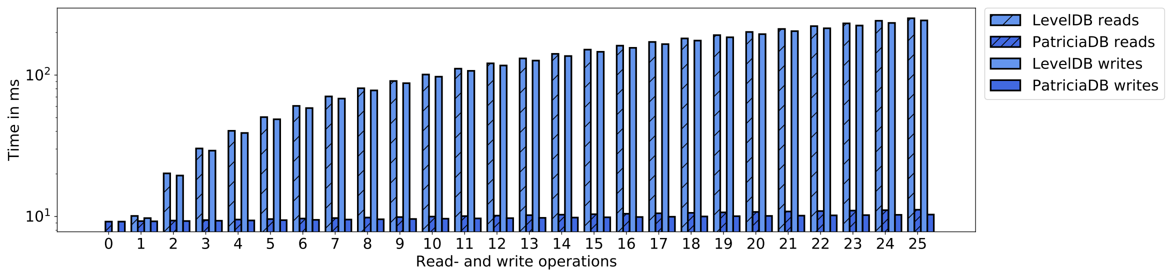

Figure 6.3: Batch read- and write operations on LevelDB and PatriciaDB.

The performance of PatriciaDB is significantly better than LevelDB. PatriciaDB equally scales as opposed to LevelDB. At last PatriciaDB varies more than LevelDB, but it does not influence the performance of PatriciaDB. PatriciaDB is more favorable over LevelDB as a world state database since performance is more important in this thesis than stability.

Batch

To utilize the PatriciaDB and its implementation of the Patricia Trie, it requires for means to access it. PatriciaDB is deployed as a REST API and therefore HTTP requests need to be sent to perform read- and write operations. In Figure 6.2 the HTTP overhead is omitted. Including the HTTP overhead in our measurements would give a distorted representation of its performance because a large portion of time would be the HTTP overhead. The average HTTP overhead is set to $9.201ms$. It is unfair to not include the HTTP overhead in our measurements because it is an ancillary that comes with our implementation of the solution. Therefore a solution to utilize the read- and write operation speed of PatriciaDB need to find a solution to utilize the read and write operations speed of PatriciaDB and cope with the HTTP overhead. Our use case performs multiple read operations before write operations. In addition, a typical transaction holds the inequality of performing more read operations than write operations as well as specific order: first all the read operations than one or multiple write operations. Each batch of read- or write operations to PatriciaDB is sent in a single HTTP request, thus only needing two HTTP request for all the read- and write operations. In Figure 6.3 the timings for batch read- and write operations are presented. This solution utilizes the read- and write operation time of PatriciaDB while coping with the HTTP overhead.

The transaction processing time can be established for the module of reimbursement from this analysis. A time-frame($150ms$) is reserved for the computations of the well-defined application logic. The transaction processing time; the reserved time-frame, 40 read- and 10 write operations and 2 HTTP overheads would therefore take

1. LevelDB: $150ms + (40 \times 10.084ms) + (10 \times 10.342ms) = 656.780ms$
2. PatriciaDB: $150ms + (40 \times 0.076ms) + (10 \times 0.042ms) + (2 \times 9.201ms) = 171.862ms$

6.3 Total transaction time

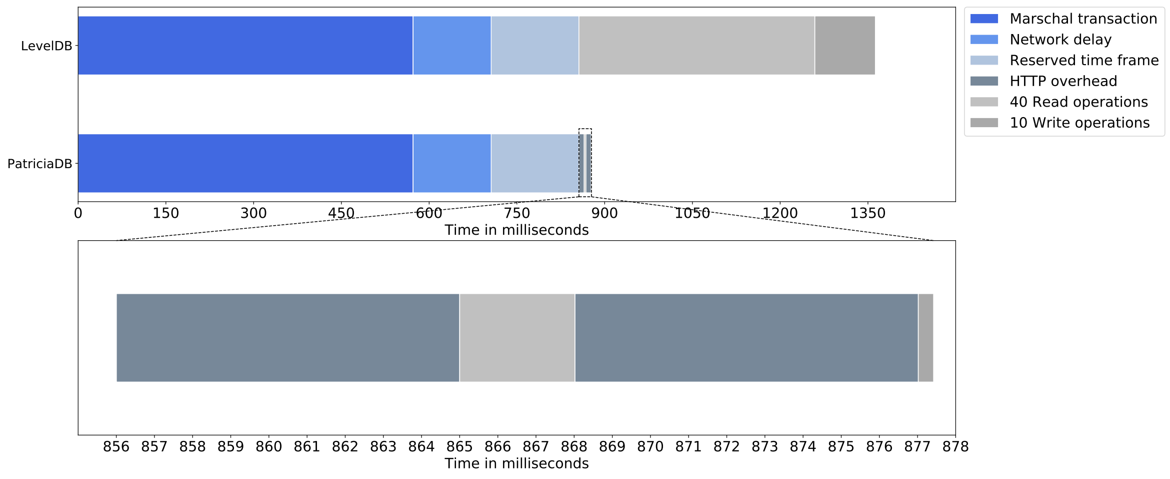

Figure 6.4: Total transaction time for LevelDB and PatriciaDB.

Figure 6.4 shows the total transaction time for LevelDB and PatriciaDB. Depicted are: *marshalling transaction*, *network delay*, *reserved time-frame*, HTTP overhead, *40 read operations*, *10 write operations*. The total transaction time for LevelDB and PatriciaDB is 1363.335ms and 878.417ms respectively. This implies that PatriciaDB, used as world state database, will reach the desired/required 50 transactions within 878.417ms for the module of reimbursement to be decentralized in a Hyperledger Fabric network. The default implementation with LevelDB would not suffice in the case of the module of reimbursement due to the read- and write operation time being to long.

Chapter 7

Conclusion

The primary objective of this thesis is to find ways to improve the transaction throughput for the Hyperledger Fabric framework. The transaction throughput becomes problematic for the module of reimbursement to be decentralized with a Hyperledger Fabric network. The module of reimbursement requires a transaction throughput of 50 transactions per second each incorporating 40 read- and 10 write operations on the world state database. This thesis introduces PatriciaDB which funds as a self-made world state database from which data is read- and written by the module of reimbursement. This world state database incorporates a special data structure called the Patricia Trie. With PatriciaDB embedded in a Hyperledger Fabric network, it can process 50 transactions in 878.417ms when send in parallel while doing the necessary computations and read- and write operations of the module of reimbursement.

7.1 Answers on the research questions

To form a concise statement for this thesis, a primary research question and sub-research questions are presented. Answers to these sub-research questions and at last the primary research question are given.

What is the transaction round-trip time of a decentralized Hyperledger Fabric network?

This thesis focuses on transaction throughput and one of the segments of transaction throughput is the transaction round-trip time. Therefore we need a research-based number of what the transaction round-trip time is for a decentralized Hyperledger Fabric network. Subtracting this time-frame from the total transaction time leaves us with the remaining time-frame that can be used for the transaction processing time. From the results that are gathered during the measurement phase, we establish that the transaction round-trip time for a decentralized Hyperledger Fabric network is 706.555ms.

What is the read- and write operation time on the LevelDB world state database of a Hyperledger Fabric network?

This sub-question is introduced to establish the default read- and write operation time on the LevelDB world state database. This is of concern since we need these numbers as our base case. The base case is used to define the improvement of the presented solution to improve the transaction throughput. From research, we can say that the read- and write operation time for LevelDB in a Hyperledger Fabric network is 10.084ms and 10.342ms respectively.

How to improve read- and write operation time on the world state database of a Hyperledger Fabric network?

We can improve the transaction processing time of a Hyperledger Fabric network by implementing a self-made world state database called PatriciaDB. PatriciaDB has a different underlying data structure but utilizes the same core semantics of the default world state databases LevelDB and CouchDB. Read operations improve with a magnitude of 132 and write operations improve with a magnitude of 241 without the HTTP overhead as opposed to the default LevelDB implementation.

The module of reimbursement needs to execute 40 read- and 10 write operations on the world state database for the well-defined application logic to process transactions successfully. This would take 506.780ms with LevelDB. With PatriciaDB this would take 21.862ms when using two HTTP request to batch the read- and write operations. The read- and write operation time on the world state database is improved by a magnitude of 23.

Can a transaction throughput be reached such that it becomes feasible to decentralize the module of reimbursement with the Hyperledger Fabric framework?

After extensive quantitative and qualitative research we can for sure say that, with the new implementation of a world state database called PatriciaDB improved read- and write operations time to the world state database is realized. This improves the overall transaction throughput to such an extent that the transaction throughput for Hyperledger Fabric becomes feasible for the module of reimbursement to be implemented in a decentralized Hyperledger Fabric network. The new implementation makes it possible to reach a transaction throughput rate of minimum 50 transactions per second while doing the necessary computations and read- and write operations to process a transaction(claim) successfully. This means at peak load the network can handle the number of claims that would be sent to the network and therefore required feasibility is reached regarding transaction throughput.

7.2 Future work

The Hyperledger Fabric network is set up in such a way that the chaincodes, that incorporate read- and write operations on the world state databases, need to be endorsed by both peers. This means that a single transaction needs to be processed by both peers in our network. The network can process 50 transactions per second with the self-made implementation of the world state database: PatriciaDB when a transaction needs to be endorsed by two peers. This means that a *single* peer is able to processes 50 transactions per second. Moreover, this means that if the endorsement policy of our research network would have been that only one peer needs to endorse a transaction that, with two peers, we would be able to process 100 transactions per second since we can distribute transactions over more than one peer. To even discuss this further: for each peer that we add to the network, we are able to process 50 transactions extra. Although this would undermine the purpose of the endorsement policy(if a transaction only needs to be endorsed by one peer). Since endorsement policies are enforced such that malicious peers can be refuted. In addition, it has not yet been tested if adding more peers would result in other issues like larger network delay. It would still be interesting to see if this proposition holds since if this is true: we can scale the Hyperledger Fabric network in linearly by adding more peers.

We notice that the stability of the PatriciaDB application, Tomcat or Docker can be of concern. Since PatriciaDB is written in Java as REST API which is exposed in a Tomcat that is in turn deployed in a Docker container, there are a lot of points of failure. If either one of the three breaks due to an exception it can become quite a problem regarding the stability of the application. Although it is stable for this set size, future research can be done to establish its stability for larger set sizes with millions of key-value entries.

Finally, the performance of PatriciaDB is significantly better as LevelDB. Therefore it would be interesting to fully incorporate PatriciaDB within the Hyperledger Fabric framework and make it an option when building a Hyperledger Fabric network.

Chapter 8

Acknowledgements

In this Chapter, I would like to take the opportunity to show my gratitude to the individuals that helped me during this master thesis project in any way.

I would like to thank Marc Makkes for supervising me during this project. The excellent technical guides helped a lot during the research and really opened my mind on how interesting distributed/ decentralized systems can be. Also, the firm guidance during the writing of this thesis really helped in become better at scientific writing. Over the course of writing this thesis, I noticed that I became better and better due to this guidance.

Next, I would like to thank Albert Lakerveld, Huub Rood and Peter Spaanderman for the trust in my skill of writing my thesis for Oracle Health Insurance and sharing needed information in the very complicated domain of health care- and insurance regulations in the Netherlands. I would like to thank Albert Lakerveld for helping me in acquiring the needed server nodes to do the research and for his knowledge on how to structure the research approach to come to a positive result. Furthermore, I would like to thank Huub Rood for his management guidance and providing me with the needed accessories to be able to work at Oracle. Furthermore, I am also very grateful for introducing me to other interested blockchain colleagues at Oracle and making it possible to join certain meetings related to the Oracle Autonomous Blockchain Cloud solutions. At last, I would like to thank Peter Spaanderman for making me part of the Interface Team. Although this thesis was performed individually, it definitely didn't feel like it. I was allowed the talk about my thesis in the daily stand-ups which made me feel part of the team. The daily stand-ups and sprint reviews also gave a glimpse of the work that is done by the Interface Team and how their Scrum process is implemented.

List of Figures

List of Tables

Bibliography

[1] Elli Androulaki, Vita Bortnikov, Christian Cachin, Angelo De Caro, David Enyeart, Srinivasan Muralidharan, Chet Murthy, Keith Smith, Alessandro Sorniotti, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric : A Distributed Operating System for Permissioned Blockchains. arXiv:arXiv:1801.10228v2.

[2] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. MedRec: Using blockchain for medical data access and permission management. *Proceedings - 2016 2nd International Conference on Open and Big Data, OBD 2016*, pages 25–30, 2016. doi:10.1109/OBD.2016.11.

[3] Oracle Corporation. Jersey - restful web services in java. URL: https://jersey.github.io/.

[4] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. BLOCKBENCH: A framework for analyzing private blockchains. *CoRR*, abs/1703.04057, 2017. URL: http://arxiv.org/abs/1703.04057, arXiv:1703.04057.

[5] Ethereum. Modified merkle patricia trie specification. URL: https://github.com/ethereum/wiki/wiki/Patricia-Tree.

[6] Hyperledger Fabric. Hyperledger fabric java sdk. URL: https://github.com/hyperledger/fabric-sdk-java.

[7] Apache Foundation. Apache couchdb: The definitive guide. 5(06):568, 2009. URL: http://couchdb.apache.org/.

[8] Hyperledger Foundation. Hyperledger Fabric Documentation. 2018. URL: https://www.hyperledger.org/projects/fabric.

[9] Omar El Garby. Data structures a quick comparison. URL: https://medium.com/omarelgabrys-blog/data-structures-a-quick-comparison-6689d725b3b0.

[10] Google. Leveldb documentation. URL: https://github.com/google/leveldb/blob/master/doc/index.md.

[11] Christian Gorenflo, Stephen Lee, Lukasz Golab, and S Keshav. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. *arXiv preprint arXiv:1901.00910*, 2019.

[12] Saurabh Goyal. Centralized vs decentralized vs distributed. URL: https://medium.com/@bbc4468/centralized-vs-decentralized-vs-distributed-41d92d463868.

[13] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. The Cost of a Cloud : Research Problems in Data Center Networks. 39(1):68–73, 2009.

[14] Docker Inc. Docker: Enterprise container platform. URL: https://docs.docker.com/edge/.

[15] Liao Tzu-Chun Lin, Iuon-Chang. A survey of blockchain security issues and challenges. *IJ Network Security*, 19(5):653–659, 2017.

[16] Patrick Morin. Data Structures for Strings. pages 43–63. URL: http://cglab.ca/~morin/teaching/5408/notes/strings.pdf.

[17] Donald R. Morrison. Patricia&mdash;practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968. URL: http://doi.acm.org/10.1145/321479.321481, doi:10.1145/321479.321481.

[18] Timothy Morrow. Risks, Threats and Vulnerabilities in Moving to the Cloud . URL: https://insights.sei.cmu.edu/sei_blog/2018/03/12-risks-threats-vulnerabilities-in-moving-to-the-cloud.html.

[19] Senthilnathan N. Postgresql as fabric's state database. URL: https://jira.hyperledger.org/browse/FAB-8031.

[20] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. page 9, 2008. URL: https://bitcoin.org/bitcoin.pdf, arXiv:43543534534v343453, doi:10.1007/s10838-008-9062-0.

[21] Mayank Raikwar, Subhra Mazumdar, Sushmita Ruj, Sourav Sen Gupta, Anupam Chattopadhyay, and Kwok-Yan Lam. A Blockchain Framework for Insurance Processes. *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–4, 2018. URL: http://ieeexplore.ieee.org/document/8328731/, doi:10.1109/NTMS.2018.8328731.

[22] Margaret Rouse. Single point of failure (spof), how can it be prevented? URL: https://searchdatacenter.techtarget.com/definition/Single-point-of-failure-SPOF.

[23] Lakshmi Siva Sankar, M. Sindhu, and M. Sethumadhavan. Survey of consensus protocols on blockchain applications. *2017 4th International Conference on Advanced Computing and Communication Systems, ICACCS 2017*, 2017. doi:10.1109/ICACCS.2017.8014672.

[24] Britto A. Schwartz D., Youngs N. The Ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, pages 1–8, 2014. URL: http://www.naation.com/ripple-consensus-whitepaper.pdf.

[25] Nathan Senthil. Postgresql as fabrics state database. URL: https://blockchain-fabric.blogspot.com/2018/06/postgresql-as-fabrics-state-database.html.

[26] Harish Sukhwani, M Mart, Xiaolin Chang, Kishor S Trivedi, and Andy Rindos. Performance Modeling of Blockchain Consensus Process (Hyperledger Fabric). pages 7–9. doi:10.1109/SRDS.2017.36.

[27] Steen M. Tanenbaum A. S. *Distributed Systems.* 2018.

[28] Yufei Tao. Tries , Patricia Tries , and Suffix Trees.

[29] Parth Thakkar, Senthil Nathan, and Balaji Vishwanathan. Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform. 2018. URL: http://arxiv.org/abs/1805.11390, arXiv:1805.11390.

[30] Fahim ul Haq. The top data structures you should know for your next coding interview. URL: https://medium.freecodecamp.org/the-top-data-structures-you-should-know-for-your-next-coding-interview-36af0831f5e3.

[31] Alex Vikati. Ranking ethereum smart contracts. URL: https://medium.com/@vikati/ranking-ethereum-smart-contracts-a27e6f622ac6.

[32] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9591:112–125, 2016. doi:10.1007/978-3-319-39028-4_9.

[33] Marko Vukolić. Rethinking Permissioned Blockchains. *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts - BCC '17*, pages 3–7, 2017. URL: http://dl.acm.org/citation.cfm?doid=3055518.3055526, doi:10.1145/3055518.3055526.

[34] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[35] Xiao Yue, Huiju Wang, Dawei Jin, Mingqiang Li, and Wei Jiang. Healthcare Data Gateways: Found Healthcare Intelligence on Blockchain with Novel Privacy Risk Control. *Journal of Medical Systems*, 40(10), 2016. URL: http://dx.doi.org/10.1007/s10916-016-0574-6, doi:10.1007/s10916-016-0574-6.