Linköping University | Department of Computer and Information Science
Master thesis, 30 ECTS | Datateknik
2018 | LIU-IDA/LITH-EX-A--2018/043--SE

Permissioned Blockchains and Distributed Databases: A Performance Study

Permissioned Blockkedjor och Distribuerade Databaser: En Prestanda Undersökning

Sara Bergman

Supervisor : Mikael Asplund
Examiner : Simin Nadjm-Tehrani

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

Blockchain technology is a booming new field in both computer science and economics and other use cases than cryptocurrencies are on the rise. Permissioned blockchains are one instance of the blockchain technique. In a permissioned blockchain the nodes which validates new transactions are trusted. Permissioned blockchains and distributed databases are essentially two different ways for storing data, but how do they compare in terms of performance? This thesis compares Hyperledger Fabric to Apache Cassandra in four experiments to investigate their insert and read latency. The experiments are executed using Docker on an Azure virtual machine and the studied systems consist of up to 20 logical nodes. Latency measurements are performed using varying network size and load. For small networks, the insert latency of Cassandra is twice as high as that of Fabric, whereas for larger networks Fabric has almost twice as high insert latencies as Cassandra. Fabric has around 40 ms latency for reading data and Cassandra between 150 ms to 250 ms, thus it scales better for reading. The insert latency of different workloads is heavily affected by the configuration of Fabric and by the Docker overhead for Cassandra. The read latency is not affected by different workloads for either system.

Acknowledgments

First I would like to thank Simin Nadjm-Tehrani and Mikael Asplund, my examiner and supervisor at IDA, for all the feedback and valuable discussions throughout the work. Thanks to Yacov Manevich, IBM, for your patience and expert guidance on Hyperledger Fabric. Thanks to Albin Odervall for excellent feedback on the report. Finally I'd like to thank my friends and family for their support and encouragement during my studies.

Contents

List of Figures

List of Tables

1 Introduction

Blockchain technology is a booming new field in both computer science and economics. It has been called the only truly disruptive new technology of the last few years. Blockchains are essentially a data structure in the form of a chain of cryptographically linked blocks of data stored over a peer-to-peer network. New blocks are continuously added to the chain when new transactions are issued. The key features of blockchains are the immutability of data and full decentralization, and they promise both scalability and anonymity. The first instance of this technique was the cryptocurrency Bitcoin and several other cryptocurrencies emerged shortly after. This cryptocurrency evolution has the economists predicting the end of banks as we know them but there is also a growing concern for the possible downsides[1,2]. Now the world is looking at other areas of possible adoptions for the blockchains. There have been some adoptions of this technique, including dating platforms[3], Singaporean ride sharing services[4] and pork origin tracking in Taiwan[5]. The blockchain technology is in an expansive phase, where more and more companies want to be early adopters and utilize this technology in their own businesses.

Blockchains can be both private and public, public means that anyone can access the blockchain at any given time, and private means that the blockchain only can be accessed by trusted peers. They can also have different permission properties, which refers to the amount of trust placed in the peers. A permissionless blockchain allows anyone to validate transactions whereas a permissioned blockchain only allows peers with permission to validate. The original blockchain, the Bitcoin blockchain, is public and permissionless, meaning that anyone can join and approve transactions. Permissionless blockchains can achieve distributed consensus by several techniques, the most common ones are Proof-of-Work (PoW) and Proof-of-Stake (PoS). PoW is what the Bitcoin blockchain uses and it is a cryptographic puzzle with is very computationally heavy to solve. This makes the

---

[1] https://www.bloomberg.com/news/articles/2017-12-15/what-the-world-s-central-banks-are-saying-about-cryptocurrencies

[2] https://www.imf.org/en/News/Articles/2017/09/28/sp092917-central-banking-and-fintech-a-brave-new-world

[3] https://www.meetluna.com/

[4] https://vulcanpost.com/644448/tada-ride-hailing-app-blockchain-singapore/

[5] https://www.bloomberg.com/news/articles/2018-03-22/ex-googler-wants-to-upend-pigs-and-hotels-with-the-blockchain

permissionless blockchains resilient towards a large number of so called Sybil attacks and provides strong guarantees on integrity [11]. But since PoW is computationally heavy it comes with high energy consumption of the peers and requires participating peers to be hosted on high performance machines. PoS is another option which exists in many variants, but the common feature is that peers with higher stakes e.g. more money are more likely to be selected to validate new blocks. In permissioned blockchains PoW or PoS isn't needed since all peers are trusted, which gives large benefits in terms of energy consumption.

The emergence of these permissioned blockchains has got the blockchain community ask, are they really blockchains [6] [7]? Others think that it is with the permissioned blockchains that we will see the most successful adaptations in the short term[8]. One can go further and ask if we have a trusting environment and want to store data in a decentralized way, do we really need a blockchain? Wouldn't a distributed database also tick the boxes of decentralized and scalability that the blockchain promises? Databases have been in use for many years, and we know what to expect from them since they are a more mature technique. The performance of distributed databases has been researched and benchmarked for years. However, a systematic study of private and permissioned blockchains in terms of their added values with respect to performance has not been done yet.

Blockchain technology is a relatively new technology and the most widely used implementations are public and/or permissionless, for example Bitcoin[9] and Ethereum[10]. The focus of this thesis is blockchains on the other end of the spectrum, the private and permissioned blockchains. When it comes to open-source frameworks there are few options available for permissioned and private blockchains. Distributed databases are an older technique than blockchain technology and there are many open-source options on the market.

1.1 Motivation

Permissioned blockchains are being proposed as a means of making, for example, supply chain transactions and property transactions more transparent. Permissioned blockchains and distributed databases are essentially two different ways of storing data in a distributed system. If a company is at a crossroads and wants to choose either a permissioned blockchain or a distributed database, the performance is a key factor to consider. The two most commonly performed operations on a distributed storage solution, blockchain or database, are reading and inserting data. For distributed databases the Yahoo! Cloud Serving Benchmark (YCSB) has been used since 2010 as a benchmarking tool [8]. Permissioned blockchains are still new but Blockbench is an attempt on a benchmarking tool for them [10]. So far no research has been done to compare the performance of the different technologies, which is what this thesis aims to do.

1.2 Aim

The aim of this thesis is to evaluate and compare the latency of one permissioned blockchain and one distributed database. These two state of the art frameworks are selected based on a number of criteria. The latency in this thesis is defined as the round trip time of either reading a value from the system or inserting a new value to the system. Since the architecture of blockchains and distributed databases are fundamentally different the goal is to see how

---

[6] https://baswisselink.com/articles/permissioned-blockchains-why-they-are-silly/

[7] https://www.theverge.com/2018/3/7/17091766/blockchain-bitcoin-ethereum-cryptocurrency-meaning

[8] https://blocksplain.com/2018/02/07/permissioned-blockchains/

[9] www.bitcoin.com

[10] www.ethereum.org

much the performance of these two basic operations differs. The scalability of the systems is measured as the change in latency when the load is held constant and the number of nodes increases. The changes in latency under different workloads is also a subject of study.

1.3 Research Questions

Given the introduction to the subject the following research questions will be answered in this thesis.

1. Which frameworks are most suitable to choose as representative for each technique?

   How can one framework implementing each technique be chosen in order to make a fair and representative comparison?

2. What is the difference in latency between the two chosen frameworks when inserting new values?

   In particular, will the insert latency depend on the size of the system and how much will the latency differ between two systems of similar size?

3. What is the difference in latency between the two chosen frameworks when reading values?

   In particular, will the read latency depend on the size of the system and how much will the latency differ between two systems of similar size?

4. How does the latency of the two chosen frameworks change under different workloads?

   Will the insert latency and read latency be affected by different mixes of read and insert operations or by different loads on the system?

1.4 Method Overview

The outcome of this thesis is a comparison between the two storage techniques, distributed databases and permissioned blockchains. Only one instance of each technique is compared and these two are selected based on a number of criteria. The metrics are the scalability as a function of the insert latency and read latency. These metrics are measured in a series of four tests conducted on a virtual machine hosted on a virtual platform. The work includes finding a limit of possible network sizes to use for the tests. The scalability is measured for networks of varying numbers of participating nodes. Two of the tests in this thesis measure the latency of both systems as a function of different workloads on networks of constant size.

1.5 Delimitations

Although the selected frameworks are representative, this thesis is not a general comparison between distributed databases and permissioned blockchains. Neither does this thesis compare the application-dependent performance but focuses on the latency of insert and read operations. The thesis only compares relatively small systems of up to 20 nodes due to choice of method, this is discussed further in chapter 4.4.

1.6 Thesis Outline

The outline of this thesis is as follows, chapter 2 covers the relevant theory and terminology. It focuses on the architecture of Fabric and Cassandra and how they do inserts but also covers the related work of benchmarking permissioned blockchains and distributed databases.

Chapter 3 discusses the choice of platforms and frameworks. The experiment design is described in chapter 4 along with the evaluation metrics. Chapter 5 contains the result of the experiments. Chapter 6 discusses the found results, criticizes the method and touches upon the work in a wider context.

2 Background

This chapter covers the theory needed to understand the work in this thesis. Replication of data in distributed systems is the topic of section 2.1. Section 2.2 describes the visualization software Docker. In order to understand this thesis it is important to understand what blockchain technology is and what properties blockchains have, this can be found in section 2.3. Section 2.4 cover the foundations of distributed databases. The last section 2.5 lists and discusses related research on permissioned blockchains and distributed databases.

2.1 Replication of Data in Distributed Systems

Replication of data is a technique widely used in distributed systems, it serves to maintain copies of data on multiple nodes. Blockchains and databases have different approaches to replication of data. In blockchains all nodes in the system keep a copy of the data while in many database systems the replication factor is tunable.

Replication comes with both rewards and challenges and the topic is well covered such as the book by Coulouris et al. [9]. Done properly replication can enhance performance since the workload can be spread out over several nodes according to Coulouris et al. But it can also introduce challenges since making sure that all copies are up-to-date creates overhead and is therefore can have a negative impact on performance. Increased availability can also be gained since a system with replication can tolerate one, or possibly several, node crashes and still provide data to users. However, Coulouris et al. also point out that high availability doesn't necessarily mean that the user will receive up-to-date data, the system might provide stale data. Therefore, it is important to consider how many replicas to use if the system should be fault-tolerant and which fault models the system should be tolerant towards. If the system has a Byzantine fault model and it is assumed that f servers can exhibit Byzantine failures, then $3f+1$ servers are needed in total to maintain a working system [9]. On the other hand, if f nodes in a system of $f+1$ nodes crash, the one still standing is sufficient for fail-stop or crash failures [9]. In conclusion, prioritizing between consistency and availability and considering fault models are both important aspects when choosing the number of replicas.

2.2 Docker - OS-Level Virtualization Software

Docker is a container platform, not to be confused with a virtual machine because it does not contain a guest OS. Docker is a thin layer which runs on the host OS. Containers runs on top of Docker, see figure 2.1. A container is a lightweight, executable version of a program, containing what is needed for it to run on top of Docker as well as the source code and dependencies [1]. Each container is separate from the other containers and multiple containers can be run simultaneously[2]. Docker can be used for quick development, deployment and managing of software. When deploying a distributed system on a local machine it can be very useful since the containers provide different silos for each node to run in and Docker can emulate a network between the nodes.

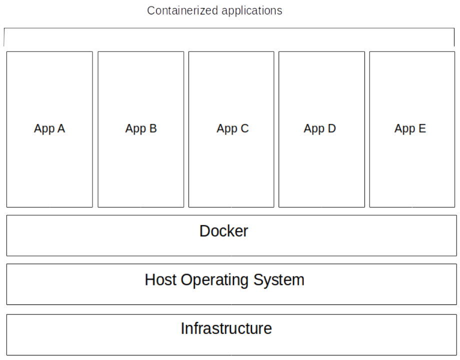

Figure 2.1: Overview of Docker

2.3 Blockchains

The technology of blockchains was first invented as the architecture of the cryptocurrency Bitcoin by the person, or people, behind the name Satoshi Nakamoto[18]. Their paper presents blockchains as distributed, decentralized, immutable and promise both scalability and anonymity of the peers using the technique. Nakamoto presents the blockchain as a decentralized system where each node holds a copy of the ledger, which is an append-only list of blocks which starts with a genesis block. A block added to the chain is never removed. Each block holds a cryptographic hash of the previous block in the chain, a timestamp and a payload specific to the purpose of the blockchain. In the Bitcoin blockchain the payload is transactions of bitcoins but it can also be other types of data such as supply chain information or land rights. A blockchain is typically managed by a peer-to-peer network.

2.3.1 Cryptography

An fundamental buildning block of blockchains is the use of cryptography. The linking of the blocks in the blockchain is done by taking the cryptographic hashes of the previous block and storing them in the block header[18]. This linking of the blocks is what makes the blockchain immutable. If a committed block is altered, the cryptographic hash will change. If the hash of one block changes, the following blocks will no longer link to the correct predecessor and the blockchain will be broken. The information stored in each block on the blockchain is stored in a Merkle tree, which is a hash tree in which every leaf contains the hash of a data block and each non-leaf node contains the cryptographic hash of its children. Only the hash of the root

---

[1] Docker Inc. What is a Container | Docker. 2018. URL: https://www.docker.com/what-container (visited on 07/24/2018).

[2] https://www.docker.com/what-docker

of the Merkle tree is stored in the block header, this makes it possible to compress information in blocks without breaking the chain[18].

2.3.2 Permission Properties and Blockchain Scope

Blockchains can have different permission properties and blockchain scope, this is two of the design decisions discussed by Xu et al. [24]. The permission properties refers to the amount of trust placed in the peers according to Xu et al. The permission properties can either be permissionless or permissioned. Xu et al. defines permissionless blockchain as blockchains where no permission is needed for peers to join and validate transactions. In a permissionless blockchain there can be no trust placed in the peers and the system needs to rely upon techniques such as PoW and PoS to establish trust in a trustless system. Examples of permissionless blockchain frameworks are Bitcoin and Ethereum. Permissioned blockchain are defined by Xu et al. as blockchains where some authority needs to give permission to peers to allow them to participate, the peers with permission are trusted. When the peers can be trusted there are no need for PoW or Pos. Examples of permissioned blockchain are Fabric and Ripple, which is a payment infrastructure. Some blockchain frameworks are permissionable, which means that they can be configured to be either permissionless or permissioned. One example of a permissionable framework is Hyperledger Sawtooth.

Xu et al. [24] defines the scope of the blockchain as the defining factor which decides if the network is public, private or a consortium/community blockchain. The question of private or public is a question of the anonymity of the peers. A blockchain is defined as a blockchain which is public can be accessed anonymously by anyone at any given time by Xu et al., Bitcoin is an example of this. This means that the information on the blockchain is viewable by anyone but it is not open for anyone to be a node in the blockchain network nor approve on transactions. Xu et al. defines a blockchain as private if it can only be accessed by a certain trusted peers, for example like the blockchains Ripple or Eris. The number of these trusted peers may vary over time and they are not anonymous.

2.4 Distributed Databases

A database is an organized collection of data which is stored electronically. Databases have a database management system (DBMS) which interacts with end-users and applications as well as manages the data in the database. In recent years DBMS have become synonymous with database [1] and this thesis uses the term database as a synonym to DBMS.

Databases can be classified in different ways, by the data they contain, by their application area or by specific technical aspects they exhibit. One category of databases are distributed database, where both the data and the DBMS are distributed over several computers. This is the type of database that are of interest in this work. According to Özsu and Valduriez [19] a distributed database is defined as "a collection of multiple, logically interrelated databases distributed over a computer network" and a distributed database management system as "the system software that permits the management of the distributed database and makes the distribution transparent to the users". The term distributed database in this thesis refers to both the distributed database and the distributed database management system. Özsu and Valduriez also categories distributed databases in to three classes based on their architecture:

- Peer-to-peer systems
- Client/Server systems
- Multidatabase systems

Databases can store and organize data in different ways. The relation model was introduced by Codd [7] in 1970 in which Codd suggested a data model based on the mathematical relationship between data to prevent disruptive changes in formatted data systems. These are now called relation databases and they store data in tables, where each table described the relationship of the data[1]. Relation databases poor scaling horizontally and this caused the emerge of so called NoSQL system, "Not Only SQL" systems, which provide good horizontal scaling for simple read and write databases [6].

2.5 Related work

This section lists and discusses related research in the field. To the best of our knowledge there has been no comparison between the latency of permissioned blockchains and distributed databases yet. Therefore, this chapter is divided into two parts, one for the related work on permissioned blockchain and one for the related work on distributed database. The related work presented has influenced the choices of metrics and measurement methodology. The presented work also acts as a source of checking validity of values obtained in this thesis and will be discussed in more detail in chapter 6.

2.5.1 Permissioned Blockchains

Dinh et al.[10] construct a framework for benchmarking private blockchains, called Blockbench. They evaluate three different private blockchains, Ethereum, Parity and Hyperledger Fabric. One of the metrics is latency as the response time per transaction, which is similar to this thesis. They also evaluated scalability with respect to changes in throughput and latency. So far no standard for benchmarking permissioned blockchains has emerged, but this is an attempt to create a standardized benchmarking tool for permissioned blockchains.

Androulaki et al. [3] present the Hyperledger Fabric architecture and perform some benchmarking. The experiments presented measure six different aspects of Fabric to see how they affected the performance in terms of throughput and end-to-end latency.

2.5.2 Distributed Databases

When it comes to distributed databases several studies on benchmarking them have been conducted. Below is a list of some studies on benchmarking or evaluating the latency of distributed databases.

- Cooper et al. [8] introduce the The Yahoo! Cloud Serving Benchmark, YCSB, which includes several workloads. This benchmark is often used in research.
- Kuhlenkamp et al. [15] compare Cassandra and Hbase based on YSCB.
- Abramova et al. [1] compare Cassandra and MongoDB by using workloads from YCSB.
- Abramova et al. [2] evaluate the scalability of Cassandra using YCSB.
- Wada et al. [23] evaluate the eventual consistency of 4 different NoSQL databases, including Cassandra

For this thesis only two related works have been chosen as representatives for the work in this area. The first is the work by Cooper et al. [8] since it introduces the YSCB which is frequently used for benchmarking. The second is the work by Kuhlenkamp et al. [15] since the authors based their tests on previous research and compare their own results to the results of others.

Cooper et al. [8] introduces the YCSB with some experiments on performance and scaling on four different database systems, Cassandra, HBase, PNUTS and sharded MySQL. One of the scaling tests measures the latency of a workload which only consists of read operations when increasing the number of nodes. They have used clusters up to 12 nodes for their work.

Kuhlenkamp et al. [15] compares scalability and elasticity of Cassandra and HBase. The authors base their test on the YCSB benchmarking tools and replicated the workloads. The authors used three different cluster sizes in all their tests, 4, 8 and 12 nodes. One of the workloads are read intense and the result was the latency of performing read operations. Another workload used was write intense and the result was the latency of performing write operations.

3 Choice of Platforms

This chapter describes the method and the requirements of choosing the two frameworks to compare. The goal is to choose one state of the art permissioned blockchain framework and one state of the art distributed database which can be compared to each other as closely as possible. Both the blockchain and the database had to be open-source and published under a license which allowed them to be used in an educational context. There are fewer blockchain frameworks available at this stage than distributed databases, simply because blockchain is a much newer technique. Therefore, the blockchain framework was chosen first and limitations or features of the blockchain framework were used to choose distributed database as well.

Section 3.1 described the selection of the blockchain framework and section 3.2 describes the selection of the database frameworks. Section 3.3 contains the necessary theory for chosen the permissioned blockchain framework and section 3.4 covers the theory of the chosen distributed database. This chapter also lists and evaluated the different cloud solutions available and the support they have for the frameworks, see section 3.5.

3.1 Permissioned Blockchain Frameworks

This section describes and compares a selection of permissioned blockchain frameworks. When it comes to open-source frameworks there are few options available for permissioned and private blockchains. For this thesis the choice was that the framework for the blockchain should not be specific to cryptocurrency applications. This is because this thesis aims to make a more general comparison between the two storage techniques and not focus on specific applications. The desired properties of the blockchain framework for this thesis are:

- The permission property should preferably be permissioned, or at the very least permissionable.
- The framework should preferably be benchmarked in literature to make it easier to evaluate the validity of the work.
- The blockchain scope should be private or possible to configure to private.
- The underlying architecture should be peer-to-peer.

- There should be documentation of the framework publicly available to make deployment easier.
- The project should be active in the sense that new updates have been released under 2018.

The items in this list are the criteria for the choice of the blockchain framework. Below are subsections describing a selection of blockchain frameworks that are permissioned, private, and not specific to cryptocurrency.

3.1.1 MultiChain

MultiChain is a framework for private and permissionable blockchains presented in a white paper by Greenspan [13]. The source code was forked from Bitcoin and was then extended. MultiChain is configurable in many ways, for example the permission property and level of consensus. One of the key features presented by Greenspan is the mining diversity, a round robin schedule which selects the validator of each block. In version 2.0, which is still in development as of August 2018 but available as a preview, MultiChain will support applications other than cryptocurrency. It is primarily intended as a framework for private blockchain within or between organizations according to Greenspan.

3.1.2 Hyperledger Fabric

Hyperledger is a collection of open-source blockchain frameworks developed in the context of an initiative from the Linux Foundation. In the Hyperledger family there are several frameworks for blockchains and the project called Fabric is highly modular and permissioned. An instance of the Fabric blockchain framework consists of a peer-to-peer network, which contains nodes, a membership service provider (MSP), an ordering service, smart contracts or chaincode, and the ledger [3].

3.1.3 OpenChain

OpenChain is an open-source framework for distributed ledgers which leverages blockchain technology. OpenChain is a framework for permissioned distributed ledgers which runs on a client-server architecture with configurable blockchain scope. According to OpenChain documentation OpenChain is not strictly a blockchain but rather it cryptographically links each transaction to the previous transaction instead of bundling transactions into blocks that are linked[1]. OpenChain supports running smart contracts and is therefore not specific to cryptocurrency.

3.1.4 HydraChain

HydraChain is a framework for permissioned and private blockchains, and it is an extension of Ethereum. It is fully compatible with Ethereum protocols and smart contracts. Developers can also write their own smart contracts in Python. HydraChain requires a quorum of the validators to sign each block as its consensus mechanism [2].

3.1.5 Hyperledger Sawtooth

Sawtooth is another open-source project under the Hyperledger umbrella. It is a framework for running permissionable distributed ledgers. Since it is permissionable it can be configured to be either permissioned or permissionless. Sawtooth provides a clear separation between

---

[1] https://docs.openchain.org/en/latest/general/overview.html#what-is-openchain

[2] https://github.com/HydraChain/hydrachain

the platform on which the application is running and the application, the smart contracts, itself [3]. Smart contracts can be written by the developer in Python, JavaScript, Go, C++, Java, or Rust. Sawtooth also enables transactions to be executed in parallel and is compatible with Ethereum.

3.1.6 Choosing a Permissioned Blockchain Framework

Table 3.1 lists the chosen blockchain frameworks together with their compatibility with the desired properties.

Table 3.1: Overview of blockchain frameworks

Name	Permission properties	Benchmarked	Blockchain scope	Architecture	Documentation	Active project
MultiChain	Permissionable	No	Configurable	P2P	Limited	Yes
Fabric	Permissioned	Yes	Private	P2P	Yes	Yes
OpenChain	Permissioned	No	Configurable	Client - Server	Yes	No
HydraChain	Permissioned	No	Private	P2P	Limited	No
Sawtooth	Permissionable	No	Private	P2P	Yes	Yes

OpenChain has the wrong architecture and HydraChain isn't an active project which makes both of them disqualified. As can be seen in table 3.1, both MultiChain and Sawtooth match all criteria, except being benchmarked in published literature. Fabric matches all requirements and is featured in several published papers. The latency of Fabric is benchmarked in papers by both Androulaki et al. [3] and Dinh et al. [10]. The consensus process is also benchmarked by Sukhwani et al. [21]. For these reasons Hyperledger Fabric was chosen as the permissioned blockchain.

3.2 Distributed Database Frameworks

This section describes and compares a selection of distributed database frameworks that are considered for this thesis. In order to make a fair comparison the two system needs to resemble each other as much as possible. This means that since the developers of Fabric strongly recommends to run it within Docker, this is a requirement for the database. Fabric runs on a peer-to-peer network, which makes it preferable for the database to do the same but it isn't a strict requirement. The requirements of the database in this thesis are:

- The framework should preferably be benchmarked in literature to make it easier to evaluate the validity of the work.
- The underlying architecture should preferably be peer-to-peer.
- The framework needs to run on Docker.
- There should be documentation of the framework publicly available to make deployment easier.
- The project should be active in the sense that new updates have been released under 2018.

Below subsections lists and describes a selection of open-source distributed database frameworks.

---

[3] https ://sawtooth.hyperledger.org/docs/core/releases/latest /introduction.html

3.2.1 MongoDB

MongoDB is a distributed NoSQL database which stores data in JSON-like documents, not in tables[4]. This database uses replica set as a way of categorizing their replicas. A replica set is a group of nodes that maintain the same dataset[5]. In a replica set there are one primary node which receives all the writes and the other nodes are secondary. MongoDB is open-source and supports over 10 programming languages.

3.2.2 Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is an open-source distributed file system under the Apache Hadoop project. HDFS is tuned to support large datasets and is optimal for batch processing rather than interactive sessions [6]. HDFS has a master-slave architecture where master nodes control the namespace and file access and the slave nodes manage storage.

3.2.3 HBase

HBase is an open-source NoSQL distributed database from The Apache Software Foundation. This database is tuned for very large data sets, preferably over hundreds of millions of rows[7]. HBase is an extension of HDFS and therefore also runs of a master-slave architecture.

3.2.4 Apache Cassandra

Apache Cassandra is a NoSQL distributed database originally developed by Facebook to accommodate its growing storage need [16]. Every node is identical in Cassandra and it is a full distributed system running on a peer to peer network.

3.2.5 Choosing a Distributed Database Framework

The investigated frameworks and their compatibility to the requirements are listed in table 3.2.

Table 3.2: Overview of frameworks for distributed databases

Name	Benchmarked in literature	Architecture	Runs on Docker	Documentation available	Active project
MongoDB	Yes	P2P	Yes	Yes	Yes
HDFS	Yes	Master-slave	Yes	Yes	Yes
HBase	Yes	Master-slave	No	Yes	Yes
Cassandra	Yes	P2P	Yes	Yes	Yes

Both HBase and HDFS are tuned for very large datasets and better for batch processing, since this thesis will have rather small datasets neither of them are well suited. MongoDB matches all the given criteria, however the replication and consistency model are more easily tuned in Cassandra. This is important since the distributed database needs to be configurable to work as similarly to Fabric as possible. For this reason and since it matches all given criteria and had a well-known consensus protocol, Paxos, Cassandra was chosen as the distributed database.

---

[4] https://www.mongodb.com/what-is-mongodb

[5] https://docs.mongodb.com/v3.4/replication/

[6] https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[7] http://hbase.apache.org/book.html7B%5C#%7Darch.overview

3.3 Hyperledger Fabric

This section describes the architecture of Fabric and its transaction flow. The operations of Fabric follow a paradigm for the transaction flow called execute-order-validate paradigm. This is a new type of transaction flow for blockchain frameworks and it consists of three phases: the execution phase, the ordering phase and the validation phase[3].

3.3.1 Roles for Nodes

The nodes which form the network can have one of three different roles, described by Androulaki et al.[3]:

- Client - Clients submit transaction proposals to the endorser and broadcast the transaction proposal to the orderer. Clients only take part in the execution phase of the transaction flow. Clients can be viewed as the user which interacts with Fabric.
- Peers - Peers validate transactions from the ordering service and maintain both the state and a copy of the ledger. All peers belong to an organization, a Fabric network can have at most as many organizations as peers and at least one organization. Peers can also take the special role of endorsement peer which performs the simulation of the transaction proposal in the execution phase. The number of endorsement peers is determined by the endorsement policy, which is set by the developer.
- Orderer - All the orderer nodes collectivity run the ordering service and uses a shared communication channel between clients and peers. These peers only take part of the ordering phase of the transaction flow. The number of ordering nodes is small compared to the number of peers.

3.3.2 Ledger

There is one copy of the ledger per channel kept locally at each peer, this is covered in the Fabric documentation [8]. The ledger consists of two parts, the block store and the state database. The first part, the block store, is a temper-resistant append-only hash chain which records all transactions[9]. The block store is the actual blockchain. Each block on the chain contains a header and a payload of transactions. The header of the block contains the hash of the previous blocks header, cryptographically linking the blocks together in a chain, and the combined hash value of the transaction payload[18]. The second part of the ledger is the state database which holds the current state of the system as a set of key-value-version pairs[10]. Each key is unique, the value is the most recent value for the specific key and the version is the latest version of the key. All the keys to ever be featured in a transaction exists in the state database.

3.3.3 Transaction Flow

The transaction flow of committing a transaction to Fabric consists of three phases. Committing a transaction can be seen as either an insert operation if new values are being written to the system or an update operation if an existing value is updated. The flow of reading data from the blockchain is covered in a separate section, see section 3.3.4. Since the objective of this thesis is to compare the insert latency to Fabric and Cassandra it is important to understand how both systems do this. The phases of the transaction flow in Fabric are the execution phase, the ordering phase and the validation phase, and they involve all the nodes in the network. All the steps can be seen in figure 3.1. The different components involved

---

[8] http://hyperledger-fabric.readthedocs.io/en/release- 1.1/ledger.html

[9] http://hyperledger-fabric.readthedocs.io/en/release- 1.1/ledger.html

[10] http://hyperledger-fabric.readthedocs.io/en/release- 1.1/ledger.html

are further described after this section. The transaction flow is described in detail both in the documentation for Fabric [14] and by Androulaki et al. [3], the developers of Fabric.

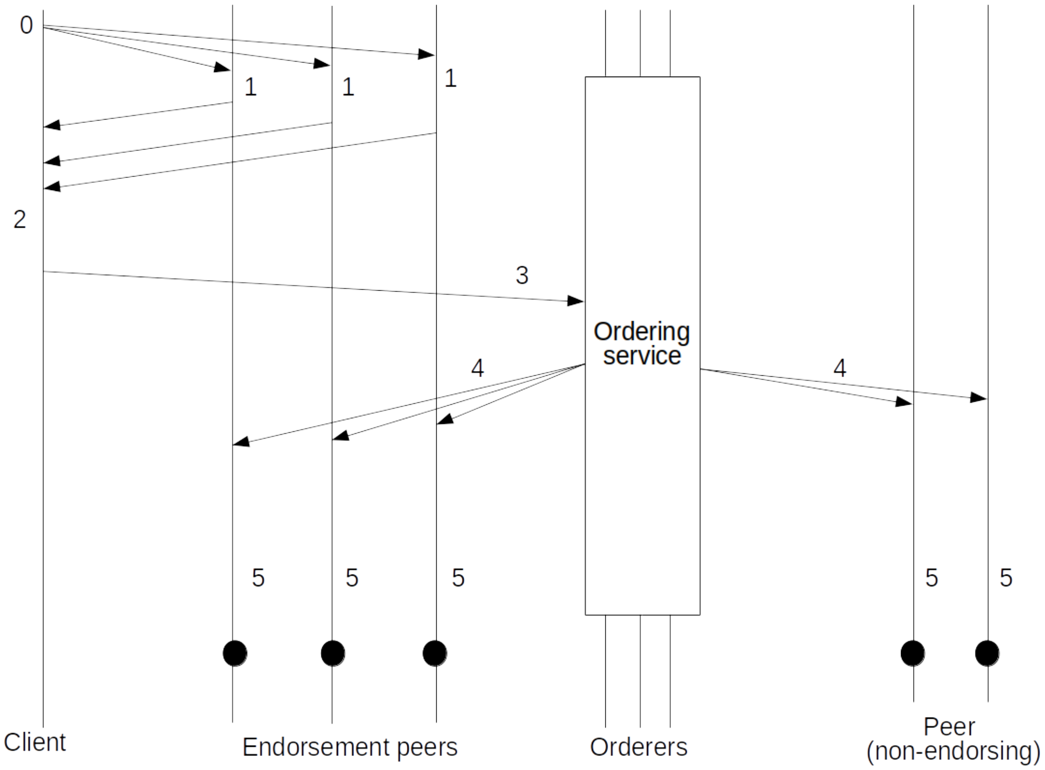

Figure 3.1: The transaction commit flow of Fabric. Source: Androulaki et al.

Below is a description of all the steps in the three phases taken from Androulaki et al. [3] and [14]. The numbers in the description corresponds to the numbers in figure 3.1. The first phase is the execution phase, which comprises of three steps:

0 The client sends a transaction proposal to a set of endorsement peers. This proposal contains the transaction id, the cryptographic identity of the client, the transaction payload, an identifier for the chaincode and a cryptographic number called nonce. The chaincode is the blockchain application.

1 Once an endorsement peer receives a transaction proposal it will simulate the transaction against its own ledger and state. The endorsement peer does not update its own ledger or state but instead produces two sets, a writeset and a readset. The writeset contains all of the state updates in key-value pairs and the readset contains all the keys read during simulation and their version. The two sets are encapsulated into a message that the endorsement peer cryptographically signs before sending it back to the client. The writeset contains the set of key-value pairs that the transaction simulation altered and the readset contains the set of keys that were read during the simulation and their version.

2 The messages sent from the endorsement peers to the client are called endorsements. The client collects endorsements until it has enough to satisfy the endorsement policy. The endorsement policy specifies how many peers that has to respond to the client before it can move on to the next step. The client can send the transaction proposal to more peers than the endorsement policy requires. When the client has successfully collected enough correct endorsements, it creates a transaction which it sends to the ordering service.

This step is the last of the execution phase and the start of the next phase, the ordering phase. The ordering phase contains two steps:

3 The transaction sent by the client contains the transaction payload, the chaincode operation and its parameters, the transaction metadata and a set of the endorsements collected in the previous phase. The ordering service places all the incoming transactions in a total order, thereby establishing consensus on them.

4 The transactions are then bundled into blocks which are appended to each other in a hash chain. The number of transaction in a block is decided by one of two factors; either the number of transactions that arrive before the `batch timeout` or the number of transactions that is equivalent to the `batchSize`[11]. Note that the transaction might be faulty in this step, the ordering service only establishes the order of them. The ordering service then broadcasts the hash chain of blocks to all peers, including the endorsement peers.

This is the last step of the ordering phase and the rest of the transaction flow is a part of the validation phase. The validation contains one step with three smaller sub-steps.

5 All peers receive the hash chain of block from the ordering service. Each block is subject to validation on the peer, since there might be faulty transactions on the blocks.

Evaluation The first step is to evaluate the endorsement policy. This means that all peers make sure that each transaction has collected the correct endorsements according to the endorsement policy. The evaluation of the endorsement policy is done in parallel for all transaction in the same block. If the endorsement policy is not fulfilled, the transaction is considered invalid. All the effects of an invalid transaction are ignored but the transaction isn't removed from the block.

Validation The next step of validation is to check if the version of the keys in the readset for each transaction is an exact match with the version of the keys in the peers' local state. This is done sequentially for all transactions and if the versions don't match the transaction is considered invalid and the effects of the transaction is ignored. Invalid transactions are not removed from the chain.

Commit The last step is to append the block to the peers' local ledger and update the state by writing all key-value pairs to the peers' local state.

After the three phases the client will get an event from one, or several, peers that conveys whether the transaction has been approved or not[12]. This means that all three phases must finalize before an insert or update can be considered successful.

3.3.4 Reading data

Reading data, or querying the ledger, is much simpler than adding a new transaction according to Fabric developer Manevich[13]. He explains that queries can be invoked by a client using chaincode, which is the program code which implements the application logic [3], and that the chaincode communicates with the peer over a secure channel. The peer in turn queries the state database and returns the response to the chaincode. After quering the state database the chaincode executes the chaincode logic and returns the answer to the client via the peer.

---

[11] https://hyperledger-fabric.readthedocs.io/en/release-1.2/config%7B%5C_%7Dupdate.html

[12] https://hyperledger-fabric.readthedocs.io/en/release-1.2/peer%7B%5C_%7Devent%7B%5C_%7Dservices.html

[13] Yacov Manevich, 2018-08-09, e-mail

3.3.5 Membership Service Provider

The membership service provider (MSP) provides identities to all nodes in the system by associating each node with a unique cryptographic identity[3]. The MSP is used to sign the messages sent by the node to verify its identity. These authenticated messages are the only allowed communication. The MSP is the component that gives Fabric its permissioned nature. The MSP is an abstraction and can therefore be instantiated in different ways.

3.3.6 Ordering Service

The ordering service can consist of one or more nodes and it is the communication fabric[3]. The ordering service ensures that transactions are totally ordered on the blockchain. It provides support for multiple channels, which is a feature to logically partition the state of the blockchain[3]. The ordering service enforces the consensus mechanism of Fabric and can be implemented in different ways, which means that Fabric has pluggable consensus.

With version 1.1.0 of Fabric two types of ordering services are provided by Hyperledger. The first is SOLO which is a centralized ordering service, running on one node only and mainly built for testing[14]. SOLO is not intended for production and should therefor not be used when benchmarking. The second is an ordering service which uses Apache Kafka and Apache Zookeeper. Kafka is a distributed, horizontally-scalable, fault-tolerant commit log which uses the publish-subscribe patterns [14]. The fault tolerance comes from replication of Kafka servers on which partition of data is spread out over [15]. Zookeeper is a centralized service which is used for coordinating distributed systems, which Kafka relies upon[16]. The service based on Kafka and Zookeeper is meant to be used in production[14]. Developers can also build their own ordering service.

3.4 Apache Cassandra

This section describes the architecture of Cassandra, the consistency mechanism and the feature called lightweight transactions.

3.4.1 Architecture

Cassandra is a fully distributed system where every node in the system is identical, meaning there is no notion of server or client. Cassandra is built to run on a peer-to-peer network consisting of numerous nodes in several data centers [16]. Cassandra has its own querying language, cql, which is the only way to interact with the system.

The replication factor in Cassandra refers to the number of copies of each data instance that are stored in the Cassandra network. Cassandra has two different replication factor strategies. SimpleStrategy allows a single integer to decide the replication factor without taking the location of the node into account[22]. NetworkTopologyStrategy allows for specification of replication factor per data center and attempts to spread out the replicas over the different racks.

3.4.2 Consistency Mechanism

Cassandra supports tunable consistency by the use of consistency levels. Consistency levels can be set per operation or once for all operations. It specifies the number of replicas needed

---

[14] https://kafka.apache.org/intro

[15] https://kafka.apache.org/intro

[16] https://zookeeper.apache.org/

to respond to the coordinator in order for an operation to be successful and for Cassandra to update or insert a value[22]. These levels are ranging from only requiring one replica to respond to the coordinator to requiring all replicas to respond, see table 3.3. Write operations are always sent to all replicas but the coordinator only waits for the number of responses required by the consistency level before sending its response to the client[22].

Table 3.3: Consistency levels of Cassandra

ALL	Require all replica nodes to respond.
EACH_QUORUM	Require a quorum of all replicas in each data center to respond.
QUORUM	A quorum of all replica nodes across all data centers must respond.
LOCAL_QOURUM	Require a quorum of all replicas in the same data center as the coordinator to respond.
ONE	At least one replica node must respond.
TWO	At least two replica nodes must respond.
THREE	At least three replica nodes must respond.
LOCAL_ONE	At least one replica node in the same data center as the coordinator must respond.
ANY	At least one replica node must respond. If all replica nodes are down, the write can still succeed after a so called hinted handoff has been written.

3.4.3 Lightweight Transactions

Cassandra uses an extended version of Paxos to supports linearizable consistency for a type of operations called lightweight transactions (LWT)[17]. These transactions are applicable to INSERT and UPDATE operations. LWT should be used whenever linearizable consistency is required but is not considered to be needed for most transactions.

Paxos is a consensus algorithm which solves the problem of agreement in a distributed system, explained by Lamport [17]. The algorithm consists of three types of actors and two phases. The actors are the proposers, the acceptors and the learners. Original Paxos has two phases, the prepare phase and the accept phase. In the first phase, which can be seen in figure 3.2, the first step (1) is that a proposer sends a request with a number n to a set of acceptors. If an acceptor receives this request and it has not seen a request with a higher number than n it will accept the proposal and answer the proposer with 1) a promise to never accept any request with a number lower than n and 2) if it has seen any request with a number smaller than n, return the proposal with the highest number. In the second phase, which can be seen in figure 3.4, the proposer waits until it receives a response from a majority of the acceptors. If it gets enough responses, the proposer will send an accept message to all acceptors. After the first two phases the learners, e.g. all the nodes not participating in the first two phases, needs to learn about the accepted proposal. This can be achieved by letting the acceptors message the learners whenever they accept a proposal according to Lamport.

In Cassandra's modified version of Paxos any node can take the role of the proposer and the acceptors are all the participating replicas[18]. The number of acceptors is specified by the serial consistency. If lightweight transactions is used then the consistency level is ignored for that operation and the serial consistency is used instead. Serial consistency has only two levels, LOCAL, which is equivalent to consistency level QUORUM

---

[17] https://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0

[18] https://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0

and SERIAL_LOCAL which is equivalent to consistency level LOCAL_QUORUM in Paxos.

Cassandra's modified Paxos consists of four phases. The first phase is the same prepare-phase as original Paxos, see figure 3.2, but the second is a new phase, called the read phase, see figure 3.3. In this phase the proposer sends a read-request to the acceptors which reads the value of the row which is the target and returns it to the proposer[19]. The third phase is the accept phase of the original Paxos algorithm, see figure 3.4. The last phase of Cassandra's modified version of Paxos is the commit phase, see figure 3.5, in which the accepted value is committed to Cassandra storage[20]. These additions to Paxos costs two extra round-trips, resulting in four round-trips instead of two.

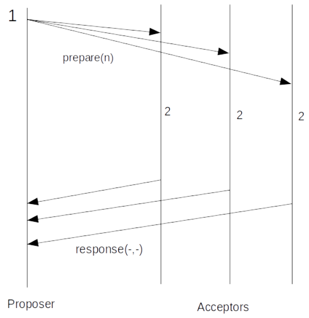

Figure 3.2: Prepare phase of Paxos, n is the request number.

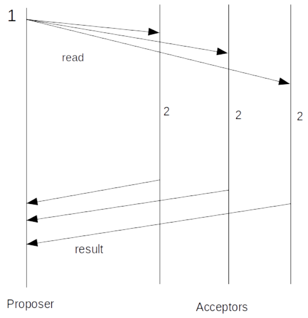

Figure 3.3: Read phase of modified Paxos

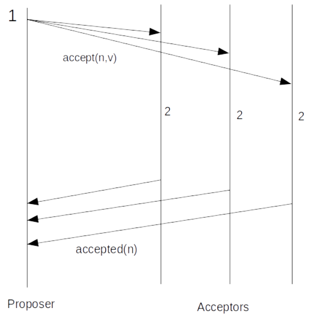

Figure 3.4: Accept phase of Paxos, n is the request number and v is the highest-numbered proposal received from the prepare phase.

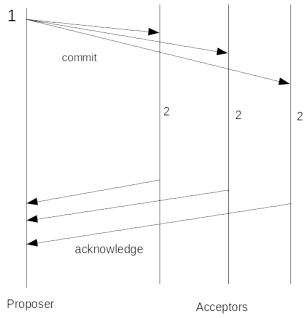

Figure 3.5: Commit phase of modified Paxos

It is important to note that all of the steps of Cassandra's modified version of Paxos takes place "under the hood". A lightweight transaction is invoked in the same manner as any other operation in Cassandra, it is simply the syntax of the operation that differs to the application.

---

[19] https://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0

[20] https://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0

3.4.4 Writing data to Cassandra

The path of writing data to persistent memory in Cassandra is four steps long. The first step is when the client invokes an insert or update operation using cql. This data is then written to a commit log, an append-only log on disk[21]. The same data is also written to the memtable, which is a memory cache stored in memory[22]. There is one memtable per node and table of data. When the data is written to the commit log and the memtable the client gets confirmation that the insert or update is complete. The final destination of data is the SSTable which are the actual datafiles on disk. The data is written to the SSTables by periodical flushes from the memtables to the SSTables[23].

3.4.5 Reading data from Cassandra

Reading data from Cassandra is more complicated than writing. Data can reside in three places, the memtable, the row cache, which is a special cache in Cassandra which contains a subsection of the data in the SSTable, or the SSTable [24]. First the memtable in memory is consulted, if the data is present in the memtable, it is read and merged with data from the SSTable. If the data isn't in the memtable the row cache is read, the row cache keeps the most frequently requested data and if the requested data of a query is present here it yields the fastest reads[25].

If the data is not in the row cache nor the memtable, Cassandra proceeds with the following steps to reach the correct SSTable, see figure 3.4.5. Data in Cassandra are grouped into partitions, and in order to find any data, Cassandra needs to find out which partition the data belongs to. Each SSTable can contain one or more partition of data. First Cassandra consults the bloom filter. The bloom filter is a probabilistic function that can help point out which SSTable keeps the requested data[26]. Next step is to read the partition key cache, which a small memory of configurable size which stores an updated mapping between some partition keys and SSTables[27]. A hit in the partition key cache saves one step in the reading flow and the search goes directly to the compression offset map. A miss in the partition key cache means that Cassandra will search the partition summary, which stores a sampling of all the partition keys as well as the location of these keys in the partition index. After getting a range to search in the partition summary, the partition index in searched. The partition index stores all the partition keys. Once the partition key is found, Cassandra proceeds to the compression offset map[28]. The compression offset map locates the SSTable in memory and returns the result.

---

[21] http://cassandra.apache.org/doc/latest/architecture/storage%7B%5C_%7Dengine.html

[22] http://cassandra.apache.org/doc/latest/architecture/storage%7B%5C_%7Dengine.html

[23] http://cassandra.apache.org/doc/latest/architecture/storage%7B%5C_%7Dengine.html

[24] https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlAboutReads.html

[25] https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlAboutReads.html

[26] http://cassandra.apache.org/doc/latest/operating/bloom%7B%5C_%7Dfilters.html

[27] https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlAboutReads.html

[28] https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlAboutReads.html

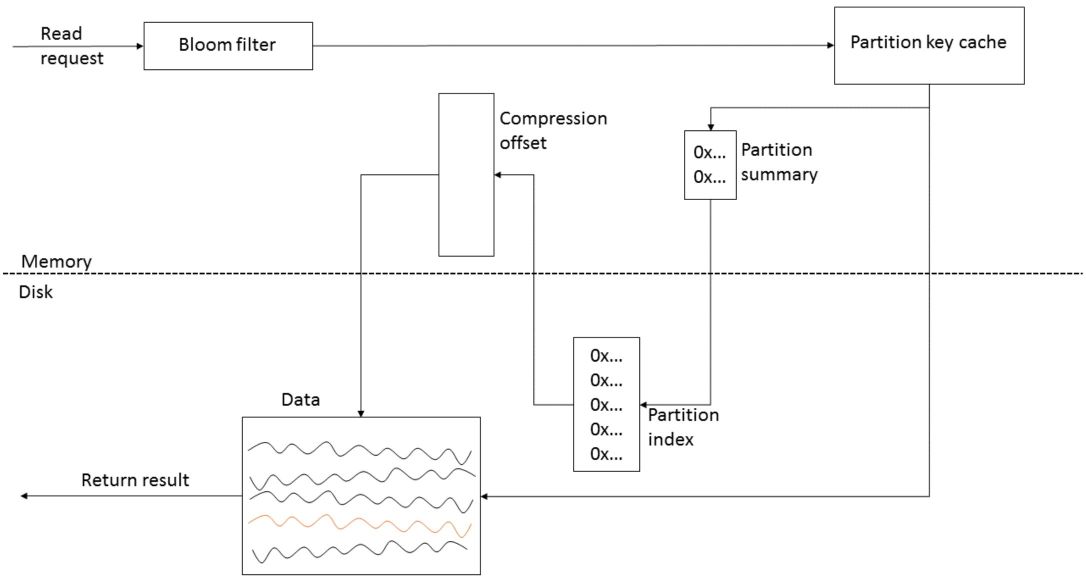

Figure 3.6: The read flow of Cassandra. Source [a].

---

[a] https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlAboutReads.html

3.5 Cloud Solutions

Previous work in the area has successfully utilized cloud solutions to deploy Fabric and Cassandra networks. For example in a paper by Sukhwani et al. [21] IBM Bluemix was used to deploy Fabric and in another paper by Androulaki et al. [3] IBM Cloud was used. In some papers the authors have chosen to build their own infrastructure using servers for setting up virtual machines, for example in the paper by Sousa et al. where they built their own ordering service [20]. For Cassandra Amazon EC2 has been used in one paper by Kuhlenkamp et al.[15]. There are also examples of when the authors built their own solution, for example the work by Cooper et al. [8]. A commercial cloud platform was also the preferred choice as the experimental platform in this thesis since this provides realistic performance for a wide range of use cases.

This section analyses the suitability of four major cloud solutions on the market, Amazon EC2, Microsoft Azure, IBM Cloud and Google Cloud. Each cloud solution is evaluated based on the range of out-of-the-box support for the two chosen storage techniques, Fabric and Cassandra.

3.5.1 Amazon EC2

Amazon EC2 is Amazon's cloud solution where users only pay for what they use, and they have a free tier. EC2 supports both Cassandra and Hyperledger as preinstalled software. Currently they only support running Fabric on a single instance [29]. This resulted in capacity problems since the free tier only supports their smallest instance t2.micro [30] which has very

---

[29] https://docs.aws.amazon.com/blockchain-templates/latest/developerguide/blockchain-templates-hyperledger.html

[30] https://aws.amazon.com/ free/

limited capacity with 1 GiB RAM memory and 1 vCPU [31] . The low capacity could only support very small Fabric networks of less than 10 nodes using Docker.

3.5.2 Microsoft Azure

Azure is Microsoft's cloud solution where users only pay for what they use, and they have a free tier [32]. Azure supports deploying Fabric as a multi-node network through the Azure portal. In the multi-node network all nodes, including one node running the MSP and one node running the ordering service, are deployed on separate virtual machines [33]. However, currently Azure only supports up to 9 peers[34], which is too small for the purpose of this thesis. Azure's free tier is not limited to one instance but the free credit can be used to buy the instances needed [35].

3.5.3 IBM Cloud

IBM Cloud is IBM's cloud solution which supports a range of services [36]. One of their services is IBM Blockchain in which it is possible to deploy a distributed Hyperledger Fabric network but Cassandra is not supported as a plug-and-play type of service [37].

3.5.4 Google Cloud

Google cloud supports Cassandra but not Fabric in their marketplace.

3.5.5 Choice of Cloud Solution

None of the cloud services' out-of-the-box solutions could be used in this thesis to run the systems distributed over several nodes. The networks could have been deployed by manually configuring the systems with one node per virtual machine on any one of the cloud services. But the platform that were chosen for this work was to run the experiments with all the nodes co-located on a single instance. This is not the preferred choice since it might have a negative impact on the result but it was the platform most suited for the budget of this thesis. Microsoft Azure and Amazon EC2 both offer Linux machines of varying sizes and specifications but Azure was chosen because of their free tier. The specifications of the virtual machine used in the tests can be seen in table 3.4. Both frameworks are setup using Docker with one node per container and all tests are run in the Docker-environment.

Table 3.4: Specification of machine running the tests

Azure instance	D4s_v3
Processor (CPU)	4 vCPUs
System memory (RAM)	16 GB
Storage	32 GB Managed Premium SSD disk
Operating system	Ubuntu Server 18.04 LTS
Azure region	West US

---

[31] https://aws.amazon.com/ec2/instance-types/

[32] https://azure.microsoft.com/en-us/free/

[33] https://azuremarketplace.microsoft.com/en-us/marketplace/apps/microsoft-azure-blockchain.azure-blockchain-hyperledger-fabric?tab=Overview

[34] https://techcommunity.microsoft.com/gxcuf89792/attachments/ gxcuf89792/AzureBlockchain/221/1/

[35] https://azure.microsoft.com/en-us/free/

[36] https://www.ibm.com/blockchain

[37] https://www.ibm.com/cloud/

4 Experiment Design

This chapter describes the design and methodology of the experiments conducted in this thesis. Section 4.1 covers the configurations of the frameworks. Section 4.2 describes the application and test scripts used. Section 4.3 motivates the choice of evaluation metrics and section 4.4 describes the experiments.

4.1 Configuration of Chosen Frameworks

This section describes how both Fabric and Cassandra are configured in the experiments.

4.1.1 Hyperledger Fabric

All experiments use version 1.1.0 of Hyperledger Fabric, since the latest (v1.2.0) was released during the writing of this thesis [1]. Each organization has one peer, one CA client and one MSP, meaning that in a network of N peers, there are N organizations. All organizations were connected using a single channel. The policy for endorsement of transactions was a quorum of organization, in order to mimic the consistency level of Cassandra. The chaincode used for the experiments is written in Golang and it is a key-value store with functions for querying the ledger and committing transactions.

As discussed in chapter 2 there are two different ordering services which comes with Fabric in version 1.1.0, SOLO and a Kafka-based ordering service. SOLO is only meant for testing and not built for a production environment, for this reason it was not used in this thesis. The ordering service is pluggable and it is possible to build one, for example Sousa et al. [20] present a Byzantine fault-tolerant ordering service for Fabric. The ordering service used in all experiments in this thesis is the Kafka-based ordering service. This ordering service consists of a variable number of Kafka servers and Zookeeper nodes. There needs to be an odd number of Zookeeper nodes to avoid split-head-decisions. Four Kafka servers is the recommended minimum in order to have fault tolerance. In this work four Kafka servers were used together with three Zookeeper nodes. Unless otherwise stated the `batchSize` is set to 1 message and the `batch timeout` to 1 second.

---

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

4.1.2 Cassandra

All experiments in this thesis use Cassandra version 3.11 and cql version 3.4.4. Cassandra has a tunable replication factor and in this thesis Cassandra was configured to use N as replication factor, where N is the number of nodes. This is the maximum number of replicas and is it not always the best choice. However Fabric always uses one replica per peer and setting the replication factor to N configures Cassandra to resemble Fabric as much as possible. Cassandra consumes a lot of RAM, so each node was restricted to only 64 MB of RAM in order to be able to run larger Cassandra networks on a single virtual machine. For all experiments LWT was used in order to utilize the Paxos consensus protocol. The serial consistency was set to SERIAL, which means that $(N/2 + 1)$ of the replicas must respond to each proposal. The choice of serial consistency level is only between SERIAL and LOCAL_SERIAL, which both are the same if the Cassandra nodes are all in the same data center, or machine as in this thesis. The serial consistency is used for all LWT operations and overrides the ordinary consistency level.

4.2 Application

The application used for the experiments is a key-value store. The test scripts are written in bash by the author of this thesis. The key-value pairs stored consists of the key which is a string and the value which is an integer. There are only two operations available in the application:

- `insert(key, value)` - inserts a new key-value pair to storage
- `read(key)` - reads a value given a key from storage

The insert operation starts a new transaction flow in Fabric and when executed the key-value pair resides in the ledger of each peer. The insert operation in Cassandra uses LWT and when executed the key-value pair resides in all replicas e.g. all nodes of Cassandra given the replication factor chosen. The read operation in Fabric reads a value from the ledger and the read operation in Cassandra follows the read flow described in chapter 2. If the application tries to read a key which isn't in storage an error will be returned, however the experiments are designed so that this never happens since these operations have higher latency.

In figure 4.1 it can be seen how the tests work on a component-level for Cassandra. The application, written in bash, uses the `docker exec` command to access one Cassandra node. Note that the application has to go through Docker and that each node runs in their own container on Docker. The `docker exec` takes the cql-command as an argument. The cql-command is either an `INSERT` for inserting or `SELECT` for reading.

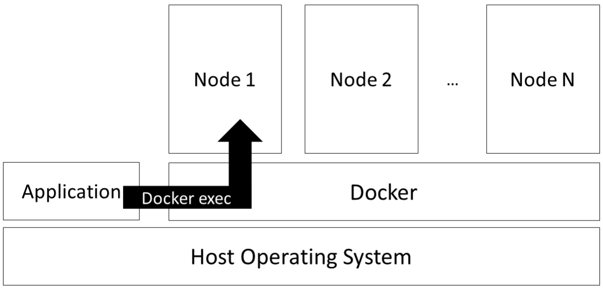

Figure 4.1: Overview of the test setup of Cassandra

In figure 4.2 it can be seen how tests for Fabric work on a component-level. The figure has been simplified in the way that the box called Fabric represents all the the sub-components of Fabric, a more detailed picture of Fabric can be seen in chapter 2, figure 3.1. Each node, e.g. both the peers and ordering service nodes, run within their own container on Docker. The tests are different from Cassandra in the way that the application, written in bash, can directly access the chaincode installed on the peers. The application invokes the chaincode on all endorsing peers, illustrated in figure 4.2 as peer 1 and peer 2.

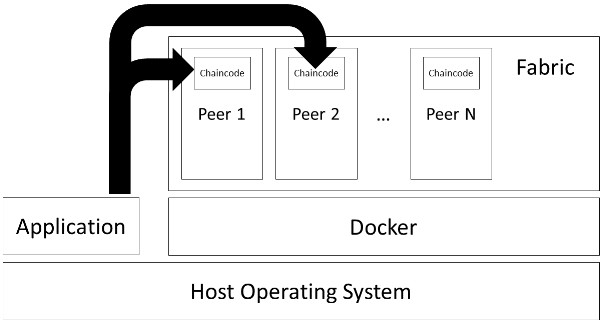

Figure 4.2: Overview of the test setup of Fabric

4.3 Evaluation Metrics

This section covers the choice of the evaluation metrics and how they are measured in the experiments.

4.3.1 Choice of Latency Metrics

The latency of a distributed system can be both measured and defined in different ways. Wada et al. [23] measure the eventual consistency of NoSQL databases from a consumer's perspective. The eventual consistency is measured in two ways, 1), as the time it takes for a client to read fresh data after an insert and 2), as the probability of reading fresh data as a function of the elapsed time since the insert. This estimates how long time the client is expected to have to wait for fresh data. The expected waiting time can be interpreted as the latency, but it is not a common way of measuring it.

The latency could also be measured as the time it takes for the client to send a request to the system or the time it takes for a system to answer the client. These two ways are both problematic in distributed systems since all nodes use different clocks. With different clocks there is always a risk of clock skew, which is hard to estimate and therefore any metric which relies on the time of two different clocks is unreliable[9]. The problem with clock skew is avoided when measuring the latency as the round trip time.

The performance metrics considered in this thesis are the insert latency as the round trip time and read latency as the round trip time.

4.3.2 How Latency as Round Trip Time is Measured

In this thesis two instances of latency are measured. The first is the insert latency as the round trip time, T_{insert}. The second is the read latency as the round trip time, T_{read}. Neither of these can be measured directly, instead T_{io} and T_{ro} are measured. T_{io} is the insert latency plus the overhead and T_{ro} is the read latency plus the overhead. Since the experiments are performed using Docker, it is the overhead of Docker that is of interest. T_{io} and T_{ro} are measured from inside the application e.g. at the start of the arrow in figure 4.1 and 4.2. T_{io} and T_{ro} can be modeled as in equation 4.1 and 4.2. The variable $T_{overhead}$ is measured in experiment 1, which is described further in section 4.4.

$$T_{io} = T_{insert} + T_{overhead} \tag{4.1}$$

$$T_{ro} = T_{read} + T_{overhead} \tag{4.2}$$

The work done when receiving an insert request before sending confirmation to the client, T_{insert}, is different on Cassandra and Fabric. When using LWT in Cassandra, the modified Paxos with four phases needs to finalize before sending confirmation. This means that the value is committed to a number of nodes, how many depends on the consistency level, when the insert operation is finalized. For Fabric all three phases of the transaction flow make up the insert operation, meaning that the value is committed to all peers. The experiments below explains how this thesis measures T_{io}, T_{ro} and $T_{overhead}$ in order to find T_{read} and T_{insert}.

4.4 Description of Experiments

There are five experiments in this thesis, each with a different goal as follows:

1. Estimating the latency overhead caused by Docker
2. Insert latency as a function of network size
3. Read latency as a function of network size
4. Insert latency as a function of increasing load
5. Latency for different mixes of insert and read operations

This section describes the experiments conducted. Each experiment was conducted on both Fabric and Cassandra, configured according to section 4.1.1 and 4.1.2 respectively. All experiments consisted of 50 individual measurement and each experiment were conducted twice and the networks were brought down between runs. This resulted in 100 measurements for each experiment.

Experiment 1, 2 and 3 uses 6 different network sizes; 2, 4, 8, 12, 16, 20 logical nodes or logical peers. Henceforth in this report the logical nodes and logical peers will be called nodes or peer, even though they are not different physical nodes or peers. The decision for these specific network sizes is both based on related work in the area and on limitations imposed by co-locating all nodes on one machine. For example Cooper et al. presents YCSB and in their benchmarking they used 2, 4, 6, 8, 10 and 12 nodes[8]. Dinh et al. presents the benchmarking tool Blockbench for permissioned blockchain and they use networks of sizes; 1, 2, 4, 8, 12, 16, 20, 24, 28, 32 nodes for their experiments[10]. Abramova et al. measures the scalability of Cassandra with YCSB, and they use 1, 3 and 6 nodes for the experiments[2]. Androulaki et al. presents the architecture of Hyperledger Fabric for their experiments they use up to 110 peers [3]. Since Cassandra requires a lot of RAM and the experiments are conducted on the same machine, only 20 nodes could be run at the same time.

4.4.1 Experiment 1 - Estimating the Latency Overhead Caused by Docker

The overhead of the network, $T_{overhead}$ needs to be estimated. Since the experiments are performed using Docker, it is the overhead of Docker that is of interest. Since all nodes are co-located in the same machine the overhead of the network, $T_{overhead}$, is supposedly small in comparison to the overall time consumption of operations, T_{ro} and T_{io}.

In Fabric it is possible to take timestamps in the chaincode and derive the overhead imposed by Docker. The tests were repeated 50 times for each network size. The timestamps in the chaincode were subtracted from the one in the test script to get an estimation of the overhead.

As can be seen in figure 4.1 the only way to execute a cql-command, for example an `INSERT` or `SELECT` statement, is to go through Docker. There are two ways to execute cql-commands. The first way is by connecting to the cql shell for an interactive session. The second way, which is used in this thesis, is to use the `docker exec` command to run a script or command from inside the Docker container. The `docker exec` command connects to the specified node and opens the cql shell, in which it runs a script or command if specified. It isn't possible to take timestamps or use any type of control structure in the querying language cql. For this reason the test scripts are written in bash and the `docker exec` command is used to run cql-scripts on Cassandra, an overview can be seen in figure 4.3. This means that there is a significant overhead from connecting to the Docker container which needs to be measured and subtracted from the timestamps recorded in the bash file. The test to measure the overhead consisted of measuring the time of executing an empty cql-script using the `docker exec` command from a bash-script, see figure 4.4. The test was repeated 50 times for each network size.

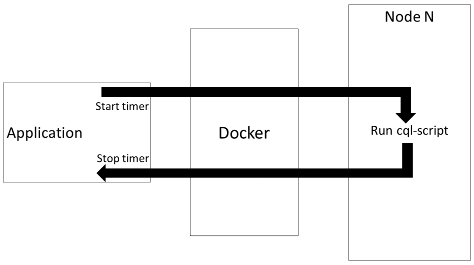

Figure 4.3: Schematic overview of experiment 2-5

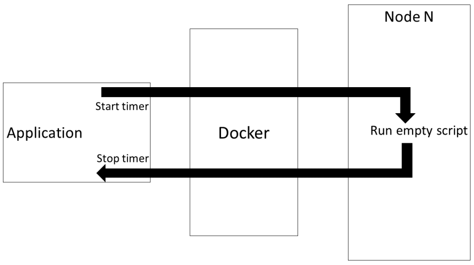

Figure 4.4: Schematic overview of experiment 1

4.4.2 Experiment 2 - Insert Latency as a Function of Network Size

The purpose of this experiment is to identify how the insert latency, T_{insert} is affected by the size of the system. This is achieved by measuring T_{io} and subtracting $T_{overhead}$, found in experiment 1. To measure the time, T_{io}, of the insert operation a timestamp was created when the operation was initialized and another timestamp when the operation finalized. The difference between these timestamps was recorded. The experiments consisted of inserting 50 new objects in the blockchain, or in the database. These operations were made with 10 second intervals. Since the preliminary tests showed latencies over 3 seconds 10 seconds was considered sufficiently large to avoid interference between consecutive operations.

4.4.3 Experiment 3 - Read Latency as a Function of Network Size

The purpose of this experiment is to identify the read latency, T_{read} and how it is affected by the size of the system. This is achieved by measuring T_{ro} and subtracting $T_{overhead}$, found in experiment 1. Since both systems use N replicas in a system of N nodes or peer, ideally the time consumption should not be heavily affected by an increase of network size. To

Table 4.1: Workloads for experiment 5

Name	Percentage of read operations	Percentage of insert operations
Read-intense workload	95%	5%
Balanced workload	50%	50%
Insert-intense workload	5%	95%

measure the time of a read operation, T_{ro}, a timestamp was created when the read command was issued and another timestamp when the read operation finalized, the difference between these timestamps was recorded. The experiments consisted of making 50 consecutive reads from one node or peer in the network and record the time. The reads were conducted once every 10 seconds for the same reason as state in previous section. This was repeated for all nodes or peers in the system.

4.4.4 Experiment 4 - Insert Latency as a Function of Load

The purpose of this experiment is to measure the effect on insert latency when the system size is constant but the load varies. The network in these tests has constant size 20 nodes or peers. The experiment consists of making 1, 5, 10, 15 and 20 concurrent insert operations to the system, repeating each burst of inserts 50 times. Each insert operation is executed on its own thread. The latency, T_{io} was recorded and $T_{overhead}$ subtracted. This experiment was repeated twice, resulting in 100, 500, 1 000, 1 500 and 2 000 reads respectively. To estimate the overhead of Docker when using Cassandra the same experiment was carried out but with empty cql-scripts. The overhead of Docker is expected to differ for this test since multiple containers will be accessed at the same time, which is the reason for repeating this test.

For Fabric the ordering service is configured differently for this experiment compared to the others. The `batchSize` is set to 10 messages and the `batch timeout` to 2 second, which are the recommended values. The reason for the different setup in the different experiments is that for the other experiment only one transaction will be submitted at the time. This makes setting the `batchSize` to 1 message the most favorable for the ordering service. However, for this experiment 10 messages are the median number of concurrent transactions and setting the `batchSize` to 10 messages will show how much this parameter affects the overall latency.

4.4.5 Experiment 5 - Latency for Different Mixes of Insert and Read Operations

The purpose of this experiment is to see how both of the system performs under different mixes of insert and read operations, called workloads. Three different workloads were used in this test, which can be seen in table 4.1. The network in these tests has constant size 20 nodes or peers. All workloads consisted of 100 operations of the least frequently performed operation and were repeated twice. For example in the first row of table 4.1 the read-intense workload is specified, it is made up of 95% read operations and 5% insert operations. For the read-intense workload 100 insert operations were performed and 1900 read operation.

5 Results

This chapter describes the results of the experiments presented in chapter 4. Overall, Cassandra has lower latencies when inserting values and Fabric has lower latencies when reading. Both scale well but the latency of Cassandra is more affected by an increase in load. The result is not discussed and analyzed in this chapter, this is done in chapter 6.

In several cases, the results in this chapter are presented using box plots, in which the vertical lines represent the minimum and maximum values. The bottom of the box represents the 25th percentile, the line in the box represents the median and the top of the box represents the 75th percentile.

5.1 Estimating the Latency Overhead Caused by Docker

The overhead, called $T_{overhead}$, of using the `docker exec` command to run cql-scripts on Cassandra can be found in graph 5.1. As can be seen there is a significant overhead imposed by Docker on Cassandra, from 500 ms for smaller network to almost 800 ms for 20 nodes. The overhead of Docker when using Fabric, called $T_{overhead}$, can also be seen in figure 5.1 and it is around 20 ms for all network sizes. The overhead is relatively small in comparison to the insert latency, see section 5.2 but rather large compared to the read latency, see section 5.3.

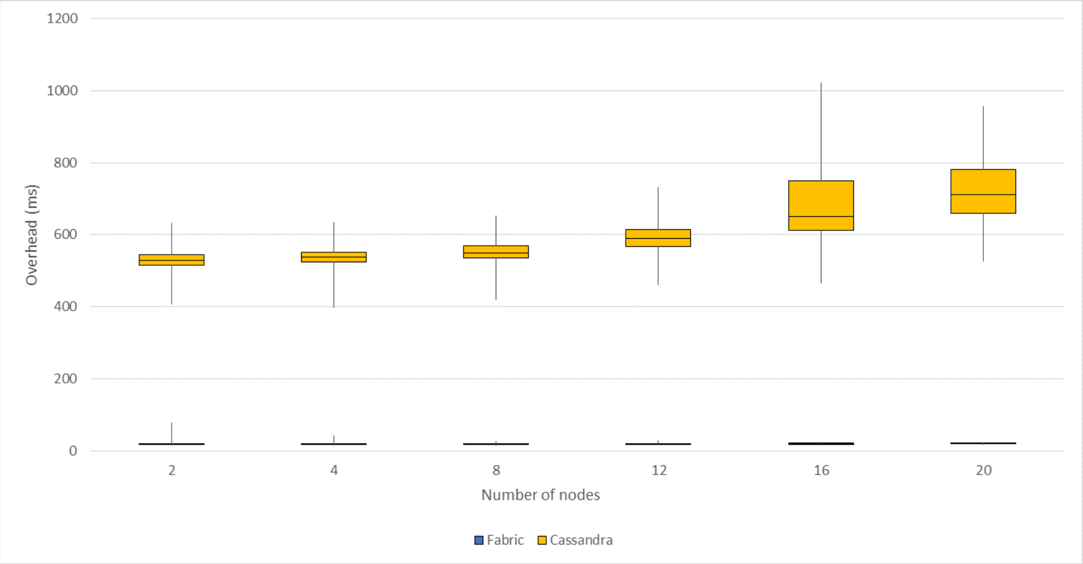

Figure 5.1: The overhead caused by network and system software when using Cassandra and Fabric respectively

The lowest overhead found in both experiments is presented in table 5.1. This overhead is subtracted from the latency found in the other four experiments when presenting the data in this chapter. This way we provide a conservative estimate of the latency caused by the Fabric and Cassandra components. The actual latency of the other experiments could therefore be lower than what is shown in the graphs for experiment 2-5. In theory the highest or median overhead could have been used instead to give the two frameworks the most favorable outcome. However in practice many of the measurement use the lowest overhead and using the median or higher overhead yielded negative measurement. These negative measurements could perhaps have been trimmed but that would have resulted in fewer data point as well as the removal of the most favorable data points. The result could also have been divided into "buckets" and different overhead values could have been applied to each bucket, but the fairness and correctness of this approach can not be guaranteed. For these reasons the choice was made to use the lowest overhead.

Table 5.1: Lowest overhead found

Framework	2 nodes	4 nodes	8 nodes	12 nodes	16 nodes	20 nodes
Cassandra	400 ms	400 ms	420 ms	460 ms	465 ms	520 ms
Fabric	15,6 ms	15,5 ms	15,5 ms	16,3 ms	16,5 ms	17,2 ms

The reason for the large differences between Fabric and Cassandra is discussed in chapter 6, section 6.1.1.

5.2 Insert Latency as a Function of Network Size

The effect of network size on the insert latency for Cassandra and Fabric can be seen in figure 5.2. There are some differences worth pointing out. Cassandra has approximately the same latency as Fabric for smaller networks but is faster for larger networks. Cassandra scales very well with only small increases in latency for larger networks. Fabric is a bit worse but still achieves good scalability for networks of this size. Cassandra has a median latency of around 180-240 ms for 2, 4 and 8 nodes and a median latency of 250-280 ms for 12, 16 and 20 nodes. The majority of the measurement data is in a tight range but there are some outliers with significantly higher latency for Cassandra. Fabric has a median latency of 140-160 ms for 2 and 4 peers, 220-270 ms for 8 and 12 peers and 340-420 ms for 16 and 20 peers. Just like Cassandra the data points for the 25th to the 75th percentile are in a close range but there are some outliers with significantly higher latency.

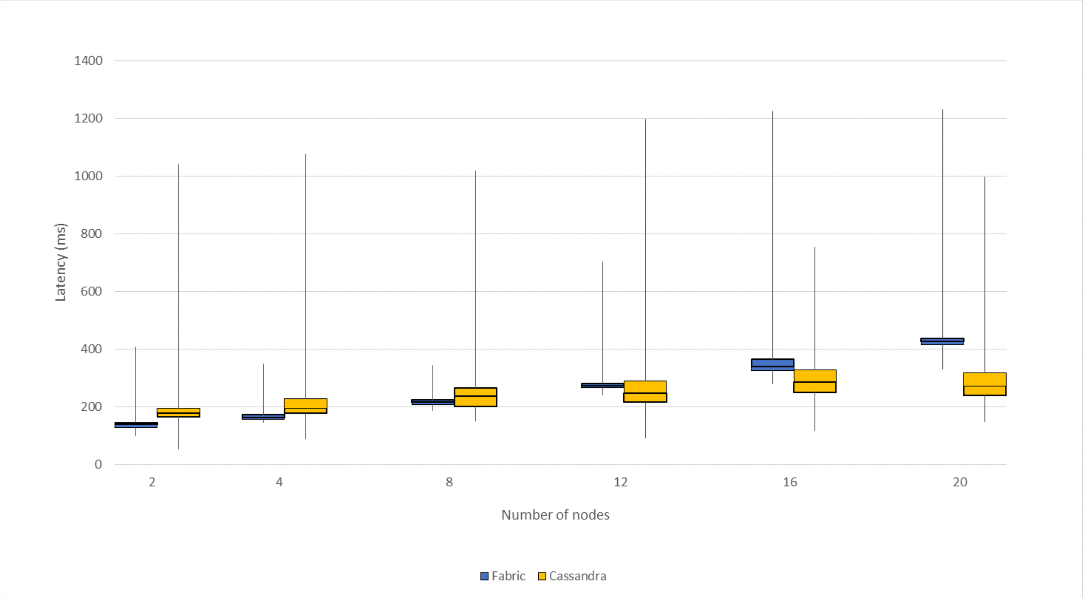

Figure 5.2: The insert latency

5.3 Read Latency as a Function of Network Size

How the read latency is affected by the network size can be seen in figure 5.3 for both Fabric and Cassandra. The results are very similar in terms of scalability, there is very little increase in latency of larger clusters. Fabric has much lower latency than Cassandra. The read latency in Cassandra is similar to the insert latency with 150-180 ms as the median latency with 2 and 4 nodes, 220-250 for 8 and 12 nodes and 230-270 for 16 to 20 nodes. The data points are more scattered for reading than inserting for Cassandra. In Fabric the median read latency is around 30 ms for smaller networks of 2, 4 and 8 peers and between 35-40 ms for larger networks of 12, 16 and 20 nodes. All the data points for Fabric are in a close range. There was no difference in read latency between which node or peer the data was read from, as expected with the given replication factor.

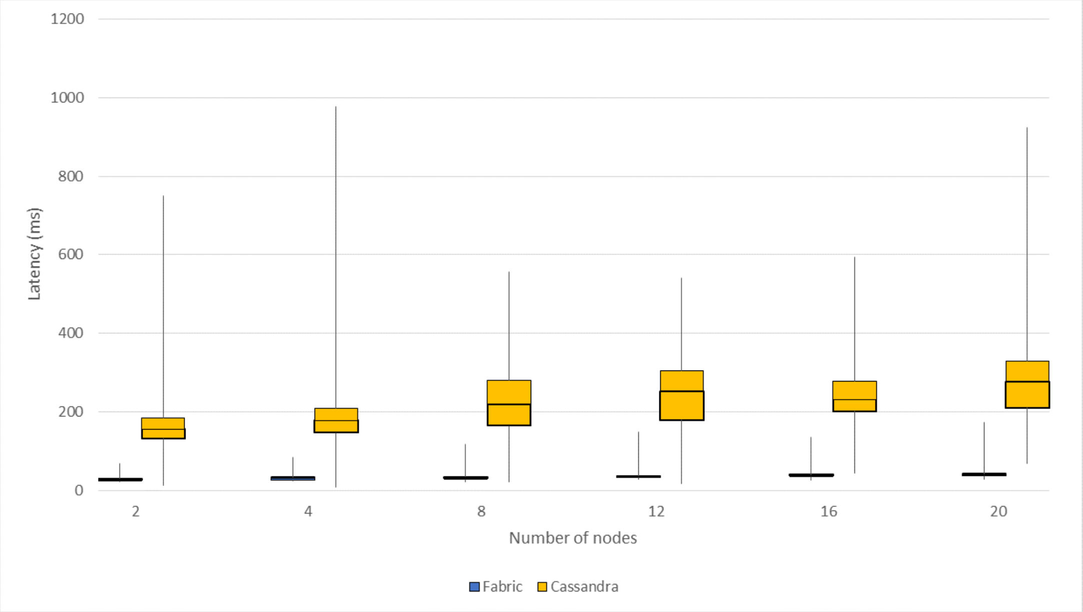

Figure 5.3: The read latency of Cassandra and Fabric

5.4 Insert Latency as a Function of Load

This experiment investigates how the insert latency is affected by the load on the system. The result for Fabric and Cassandra can both be seen in figure 5.4. Surprisingly the latency of Cassandra is now much higher than the latency found in experiment 2. The latency of Cassandra is also higher than the latency of Fabric for higher loads, which also is different from experiment 2. What this figure doesn't tell is that the overhead for Docker is larger for this experiment than the overhead found in experiment 1. The overhead of Docker for this experiment can be seen in figure 5.5. Now it is clear that Cassandra has a better scale up, but the latency for higher throughput is almost the same for both systems. For loads of 15 and 20 writes per second Cassandra sometimes fails to receive confirmation from a sufficient number of acceptors, which leads to timeouts.

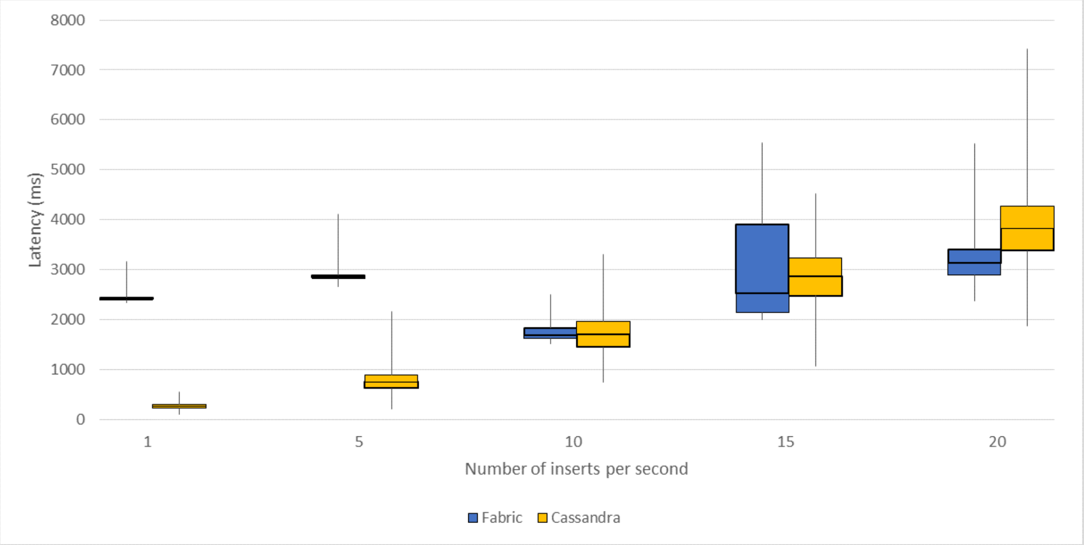

Figure 5.4: The insert latency of Fabric and Cassandra under increasing load

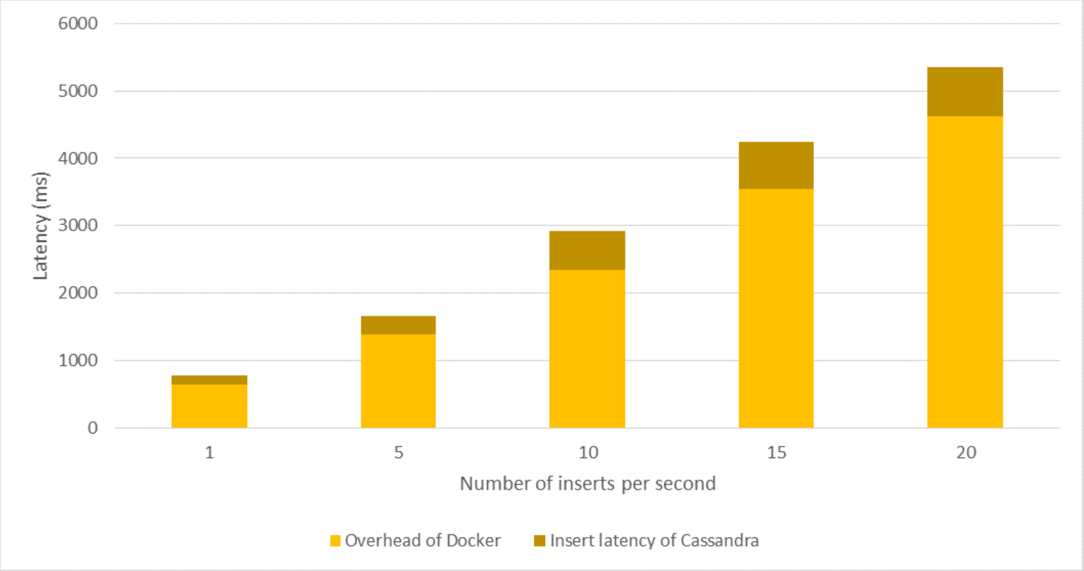

Figure 5.5: Insert latency of Cassandra and overhead of Docker under increasing load

The results for Fabric is interesting since the latency drops significantly between 5 and 10 inserts per second. This behavior can be seen even more clearly in figure 5.6 which contains more data points. In this figure it is clear that at 10 writes and 20 writes per second the latency drops suddenly, and then increase again. This behavior is discussed in chapter 6.

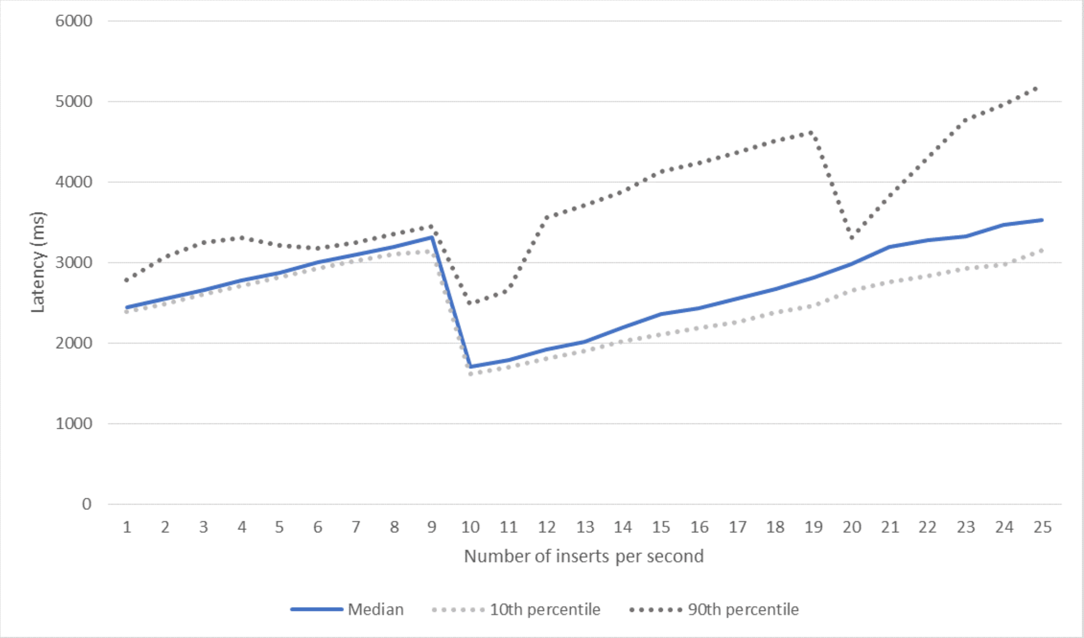

Figure 5.6: The insert latency of Fabric under increasing load - extended

5.5 Latency for Different Mixes of Insert and Read Operations

In this section the result from the experiments on different workloads is presented. The workloads are repeated again in table 5.2.

Table 5.2: Workloads for experiment 5

Name	Percentage of read operations	Percentage of insert operations
Read-intense workload	95%	5%
Balanced workload	50%	50%
Insert-intense workload 3	5%	95%

For Cassandra there are only small differences between the workloads. The insert latency decreases with more write-intense workloads, see figure 5.7. The read latency is almost not affected at all, see figure 5.8, even the outliers have almost the same maximum value.

For Fabric the largest difference is between the read-intense workload and other two workloads, see figure 5.7 and 5.8. Between the insert-intense workload and the balanced workload there are almost no differences neither in insert latency nor read latency. The insert-intense workload and the balanced workload has lower insert latency than the read-intense workload and smaller variance as well. For read latency all workloads have the same median latency. For the read-intense workload, with the highest percentage of read operations, there are more data points above the median for both read and insert latency.

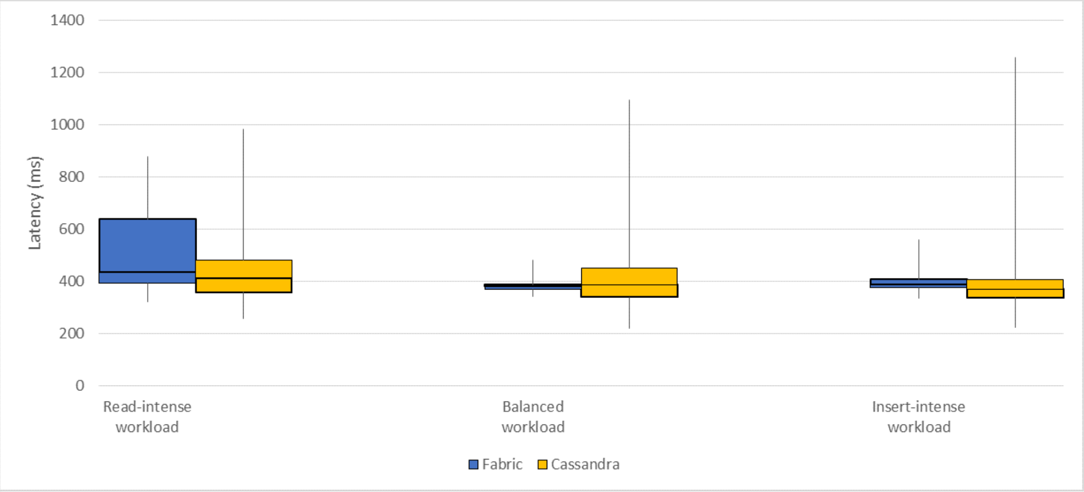

Figure 5.7: The insert latency of different workloads

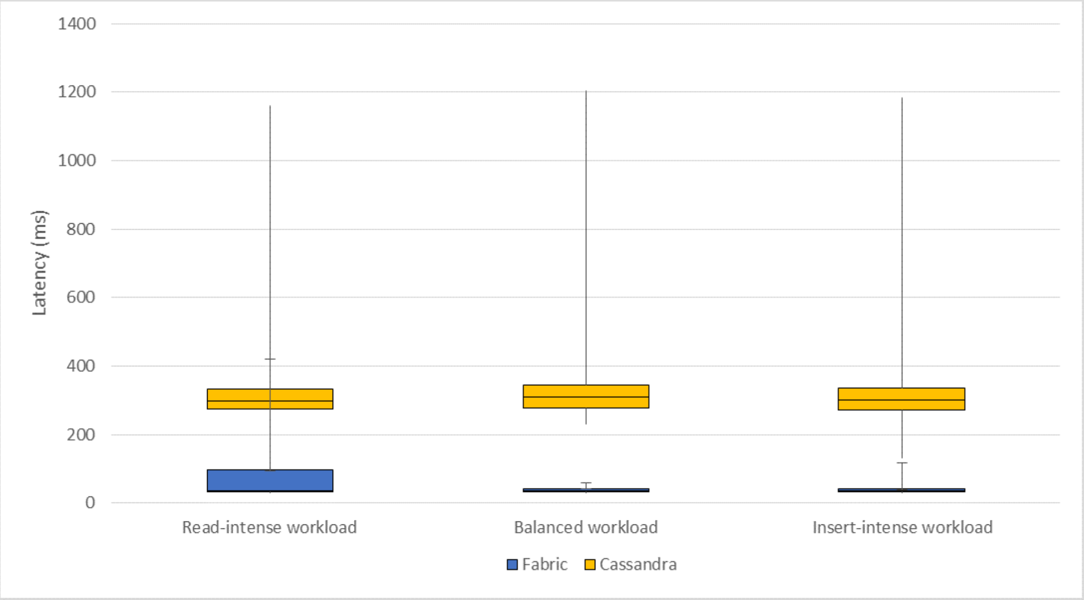

Figure 5.8: The read latency of Cassandra and Fabric of different workloads

6 Discussion

This chapter is divided into three parts, the first, found in section 6.1, discusses the result presented in chapter 5. The second part can be found in section 6.2 where the methodology will be discussed, more specifically how the choice of experimental platform can impact the results, and the difference in the maturity of both systems. Section 6.3 goes on to discuss the work in a wider context, what the benefits of blockchain technology can be and what the risks are.

6.1 Results

In the following sections the results of the experiments, presented in chapter 5, are discussed.

6.1.1 Estimating the Latency Overhead Caused by Docker

The overhead of Docker is large for Cassandra because the commands have to be issued from inside the container. This means that the command `docker exec` has to be used to start a cqlsh shell and as shown in experiment 1, this has a large overhead. This is not a fundamental feature of Cassandra but rather a result of the choice of implementation in this thesis. For Fabric the overhead is very small compared to the insert latency but very large compared to the read latency. Even though Fabric also uses Docker it is structured differently and issuing operations on the blockchain doesn't require `docker exec` to start any new shells. Since Fabric is built to run inside of Docker it is optimized to utilize Docker in a more effective way. Because of this the overhead is smaller for Fabric.

6.1.2 Insert Latency as a Function of Network Size

The insert latency Cassandra found in this work is higher than the one found in related work, a big reason for this is that this thesis uses Lightweight transactions (LWT). LWT is more time-consuming than the standard insert and update operations since it uses Paxos with 4 round trips and involves a quorum of nodes, read more in chapter 2, section 3.4.3. The reason for using LWT is to try to make the two system work as similarly as possible and Fabric always uses an ordering service to enforce ordering. Using LWT and Paxos is also the reason for why the insert latency of Cassandra scales the way it does. Paxos uses serial consistency of a quorum of nodes, which means that with a larger network more nodes have to reply in order to see the insert as successful. The scalability of LWT is good since the increase of latency is small for larger networks.

The insert latency for Fabric is on average 20% lower than the insert latency of Cassandra for up to 12 nodes. For 16 and 20 nodes Cassandra has the lower latency. Fabric performs well in terms of scalability but not as well as Cassandra. In this experiment Fabric uses a quorum of peers in the first step of the transaction flow, which means that for larger network the execution step is performed by more peers. This is the main reason why the insert latency of Fabric scales the way it does. When the size of the network grow the ordering service also has to communicate with more peers in the ordering step. Another contributing factor is that the validation step is performed by all peers in parallel and when only using one machine the work isn't spread out across multiple machines.

Androulaki et al. [3] found higher insert latency in their benchmarking of Fabric, on average the latency in their work was 542 ms. Most likely this discrepancy comes from the different setting of the ordering service, as seen in chapter 5 different setting can greatly effect the latency. The difference can also come from the fact that this thesis uses one machine for the entire network which means that the network cost of inter-peer communication is much smaller than if all peers are located on different machines. Androulaki et al. use more dedicated virtual machines and run the peers on separate machines, but they also use more CPU-power. The endorsement policy is not specified either which may lead to different conclusions. Since the transaction flow of Fabric includes a simulation of the chaincode function used, it can be hard to compare results with different chaincode applications.

Kuhlenkamp et al. [15] compare scalability and elasticity of Cassandra and Hbase. One of their workloads was write intense and the result was the latency of performing write operations. The average write latency for the 4 node cluster is approximately 20 ms, and decreases to 10 – 15 ms for the 8 and 12 node clusters. The numbers are lower than those found in this thesis, only their reported 90th percentile has around the same latency as in this work. One reason for the gap is probably how the experiments are set up, this thesis puts all the nodes on the same machine which will lead to bottlenecks such as CPU-time. Another is that this thesis uses the LWT instead of the standard inserts, LWT is a lot more time-consuming because it establishes linear consistency.

6.1.3 Read Latency as a Function of Network Size

When it comes to reading data, Fabric performs much better than Cassandra. According to previous research, presented in the section 2.5, Cassandra can be tuned for lower latency than the one found in this work. However, since the setup in these papers are different they can not be directly compared to the results in this work. In Fabric it is relatively quick to read a value since all peers always have the same copy of the world state and only the state database is consulted. This can also be seen in how the outliers are not so far from the other data points.

In Cassandra the read latency is sacrificed for quicker writes. As explained in chapter 2 data can be found in different places yielding different latencies in Cassandra. This is clear in figure 5.3. Both the data points in the 25th to 75th percentile are scattered more compared to the insert latency, which are all in a much closer range. The outliers are also further away from the other data points.

As expected with the given replication factor of one copy per node or peer, both systems had good scalability with only minor increases in latency for larger networks.

A for Cassandra Cooper et al. [8] measure the latency of a workload which only consists of read operations, when increasing the number of nodes. They have only used clusters up to 12 nodes but the increase of latency is similar to the one found in this thesis. However, the latency is much lower in their work than in this thesis. They disabled all replication and did not use LWT, which is probably what caused the biggest difference. Another reason for this is that they used six machines, instead of one, and allocated 3GB of heap instead of the 65MB. Kuhlenkamp et al. [15] also measure the latency of a read intense workload and the result was the latency of performing read operations. The average found was 35 ms for the 4 node cluster, and lower latencies of 25 – 30 m for the 8 and 12 node clusters. The numbers are lower compared to those found in this thesis, their 95th percentile matches the median of this work. Kuhlenkamp et al. didn't use LWT either and they did not use co-location of the nodes, which most likely are the reason for the difference.

6.1.4 Insert Latency as a Function of Load

In this experiment the results for Fabric were surprising because of the drop in latency for 10 writes per second. There could be seen a similar, but smaller, drop at 20 writes per second. This is because of how the ordering service of Fabric works in the ordering phase of the transaction flow. If several transactions arrive in within a small enough time interval, called `batch timeout`, they are clustered together in the same block. Unless the block is full, e.g. the `batchSize` is reached, then the block is sent to the peers immediately and the next transactions have to "wait" until the ordering service creates the next block. This goes the other way around too, if the `batchSize` isn't reached the ordering service will wait for the `batch timeout`. For this application one block can hold 10 transactions, but this is specific for this application and both the `batchSize` and `batch timeout` can be adjusted per channel. Adjusting the `batchSize` and `batch timeout` is what causes the difference in latency between experiment 2 and this experiment, this was done intentional to better optimize the latency for each test scenario.

Recall that the `batch timeout` was set to 2 seconds in this experiment and the `batchSize` to 10 transactions.This explains the latency drops at 10 inserts per second. For all the other loads before the ordering service waits for the `batch timeout` before sending the block. The fact that the 90th percentile is a lot higher for loads of 10 inserts per second and higher can also be explained by this. The 90th percentile is the latency of the transactions that had to wait for the ordering service because the first 10 transactions filled up the `batchSize`. For 20 inserts per second all transactions fit into 2 blocks exactly and the 90th percentile is therefore low for only this load variation. The linear increase of the median latency is the increase of the execution and validation steps in the transaction flow, which is expected.

Dinh et al.[10] evaluated three different private blockchains, including Fabric. One of the metrics evaluated is scalability as changes in throughput and latency. Interestingly they found that Fabric did not scale beyond 16 nodes. As for latency, their findings are similar to those of this thesis. For loads under 200 requests per second the latency started at 1 seconds to increase only a little with more peers. For higher loads the latency increased to over 10 seconds. This thesis does not research loads of the same magnitude but the rate of increases are similar to the one found in the paper.

For Cassandra, it is clear that Docker has a big effect on this experiment. However when the overhead of Docker is removed, the results found in this experiment are similar to the result found in related work by Cooper et al.[8]. In both this work and in the work by Cooper et al. the trend is that the insert latency increases significantly when the throughput was increased . However, the exact latency are not similar, this work found higher latencies that Cooper et al. When it comes to exact measurements and not trends it it important to note that their work does not use LWT and should therefore not be compared directly.

There are problems with running each insert on its own thread on a single machine with only 4 vCPUs. It is hard to separate the overhead of thread switching from actual increase in latency from increased throughput.

6.1.5 Latency for Different Mixes of Insert and Read Operations

For Fabric the largest difference of the insert and read latency could be seen for the read-intense workload 1 compared to the others. For workload 1 there was a large variance in the result for both metrics. The implication of this is that more reads to the system leads to higher latencies. This also implies that the bottleneck of the transaction flow are the peers since this is the only overlap between the transaction flow and the read flow. This result is the consequence of the co-locating of peers in one machine. The execution and validation phases involves a quorum of and all peers restrictively and the work of these phases are done in parallel on the involved peers. If the peers were spread out on different machines, then the workload could be spread out too which would reduce latency in this case. With hindsight, the decision to run all peers on one machine was not a good decision.

In Cassandra a small improvement in latency could be seen with more write-intense workloads. This was expected since Cassandra is more tuned for reads. For the read latency in Cassandra, no real difference could be seen, which is also expected given the architecture of Cassandra.

6.1.6 Overview of Test Results

In comparison to each other it is clear that Fabric is providing much shorter latencies for reading than Cassandra. The read latency of both systems are not affected by neither size of the network nor the mix of read and insert operation performed on the system. This makes Fabric the clear winner in the read latency category. Given the architecture of both systems this was expected but, after these experiments it is clear just how much better Fabric performs compared to Cassandra. It is also important to note here that the transaction flow of Fabric includes the execution phase in which each transaction proposal gathers endorsements to be eligible to change the state of the system. Cassandra does not include a similar step, which means that the latency of Cassandra would be even higher if both systems included the same steps.

For inserting, the insert latency of Cassandra scales better with the size of the network than the insert latency of Fabric. This gives us a hint that for larger systems, Cassandra will outperform Fabric and provide lower insert latencies. However, for the small systems that are the topic of this thesis both systems have almost the same latency. Additionally, with a higher throughput Fabric can outperform Cassandra if the ordering service is configured correctly.

When it comes to overhead of using Docker, as expected it is clear that Fabric is better optimized than Cassandra. The fairness of choosing to use Docker for both systems can be discussed.

6.2 Method

This section covers two aspects of the method. Firstly the choice of frameworks and maturity of Cassandra and Fabric is discussed. Secondly the experiment design and choice of method is covered.

6.2.1 Choice of Frameworks

For this thesis it would have been interesting to compare several frameworks of both technique to get a better understanding of the current state of permissioned blockchains and distributed databases. However, only one framework of each technique was chosen. When it comes to permissioned blockchain there aren't a lot of options available on the market. Developing an application with Fabric was cumbersome and challenging since it is a brand new framework, but compared to the other available options it was still the most mature choice. For distributed databases there were more options available and generally the different frameworks are more mature. In related work comparing Cassandra to other databases, such as Cooper et al. [8] and Abramova et al. [1], in workloads similar to the ones of this thesis Cassandra performs better or equivalent to its competitors. For these reasons, Cassandra was a good choice as a representative for distributed databases.

Fabric is still a very new framework, it is currently on version 1.2.0, with the first version, v.0.6.1, released in October 2016. Cassandra on the other hand was first released in 2009. Since the developers of Cassandra has had a longer time to fix performance issues in the code, one might ask if it is fair to compare the performance of Fabric and Cassandra and then draw conclusions regarding the performance of blockchain-based systems in general. This is a fair argument and perhaps this work should have been done later in time to give Fabric more time to catch up performance wise. This is the stand-point of the developers of Fabric, which they state in the paper by Androulaki et al. from 2018 [3]. However at the time this thesis work started Fabric was the best choice as a representative of the permissioned blockchain frameworks.

The CAP-theorem, originally coined by Eric Brewer in 2000 [4], states that is is not possible for a distributed system which shares data, to have consistency (C), availability (A) and partition tolerance (P) simultaneously. All three aspects are desirable, and users of distributed system have come to expect all of them. The CAP-theorem provides a way of categorizing distributed systems into CA, AP and CP systems. Cassandra is typically classified as an AP-system but with our chosen replication factor and by using QUORUM, a high consistency level, it is more tuned to be a CP-system. Although the classic definition of the CAP-theorem does not declare any connection to latency, they are still connected [5]. It may be unfair towards Cassandra to enforce the chosen replication factor. Preferably this work should have included a configuration section where multiple replication factors were tested. This way the most favorable replication factor could have been chosen for this thesis.

Using LWT instead of regular insert operations is not recommended for Cassandra since the modified Paxos used is time-consuming. Not using LWT would have shown better performance for Cassandra. However, using Paxos makes the process of inserting values more like the transaction flow of Fabric, since it includes an ordering step, which makes the comparison fairer. The chosen method illustrates that using distributed databases as a P2P system has penalties for linearizing consistency and this highlight the need for blockchains.

Overall the goal was to make the two systems behave as similar to each other as possible. The idea is that this will yield the most comparable and fair result. However, another possible approach to this is to tune both framework to meet their most optimal settings. For this report that would mean some changes for Cassandra but not too many for Fabric. This decision come down to what is considered most fair.

6.2.2 Experiment Design

The choice of co-locating all the nodes in one single virtual machine has most likely affected the results negatively. Co-locating means that all the resources are shared which could lead to bottlenecks, for example the CPU or the RAM. Therefore, it would have been preferable to use one node per machine, which is also the real-world scenario of distributed systems. Co-locating could have affected the validity of this thesis, especially since related work in the area have almost always used one node per machine. Abramova et al. [1] evaluate Cassandra and MongoDB by running them on the same machine and for this very reason they chose not to measure latency. However, the machine used in this thesis had 16 GB of RAM and 4 vCPUs and neither worked with full utilization during any test. The use of Docker is also a good infrastructure since it simulates a network between the containers, which helps to cancel out the effect of co-locating to some extent.

Using relatively small networks of up to 20 nodes/peers is a direct consequence of using only one machine. This is small compared to actual network used in the real world. However, as discussed in chapter 4 many papers benchmarking both distributed databases and blockchains use networks of similar size. Extending the experiments to include more peers for Fabric would have been relatively simple. This is because Fabric doesn't require as much RAM as Cassandra and which means that larger Fabric network could be run on the same machine. The test-script are also extensible. Extending the experiments to include more nodes for Cassandra would require another machine but the test-scripts are extensible, meaning that it would not have required a large amount of work. However, spending time on deploying the frameworks on a fully distributed system is a better investment since this would increase the validity of the work more than adding more nodes or peers.

The documentation for Fabric is still in its early stages and a lot of knowledge only consists in the Fabric community. I've been in contact with people there, most importantly two of the developers at IBM, Yacov Manevich and Konstantinos Christidis, who have provided valuable feedback and insights. There is limited research in the field of benchmarking blockchain since it is a new research topic, which makes it harder to assess the validity of the result. But this also makes this thesis a more interesting contribution to research. Blockchains are still new and especially permissioned blockchains, therefor it is important to conduct research which benchmark them, both towards each other but also towards other possible competitors.

6.3 The Work in a Wider Context

The permissioned blockchain technology with its decentralized structure and immutable record has many possible adaptations which can have ethical benefits. Property rights are a good example of records that need to be immutable but also have few transactions over time, compared to currency. Not everyone should be able to validate property ownership changes, this makes it an excellent use case for permissioned blockchains. For land and property ownership the Swedish Lantmäteriet is one early adopter of the blockchain technology[1]. Lantmäteriet has partnered up with government agencies, banks, telecommunications provider, and other partners, but most importantly ChromaWay. ChromaWay is the technical partner which has provided the blockchain solution. The blockchain solution is ChromaWays' product Postchain, which actually isn't a blockchain but what ChromaWays call a consortium database. Postchain is said to leverage on the desired blockchain properties like linked timestamping and being decentralized yet working together with existing relational database and using SQL. Swedish Lantmäteriet are using

---

[1] https://www.lantmateriet.se/sv/nyheter-och-press/nyheter/2018/blockkedjan-testad-live–kan-spara-miljarder-at-bostadskopare-och-bolanekunder/

Postchain to create what Graglia et al. [12] calls a smart workflow, which is the second step out of eight towards a fully adopting blockchain technology to property registration. In Dubai the work has come two steps further towards full adaptation, and they have successfully changed their centralized database to a permissioned blockchain.

Since blockchains never throw away data the blockchain will eventually get large and require a lot of storage space. Therefore, applications that only have few transactions are better than applications which requires many transactions. If the system only has a few transactions per time unit, then maybe slow inserts isn't a big problem, which we could see in the result that Fabric has for larger networks. There are ethical benefits of putting property and land transactions on a blockchain, if the blockchain is permissioned. This means that the history of a property or piece of land is immutable and can be accessed, which could help with issues such as illegal settlements, land frauds or disputes between neighbors of farming rights[2].

The blockchain technology is currently very hot, with many start-ups emerging[3] as well as older companies that are making a switch to blockchain[4]. More and more research is being published but so far it is still not a well-studied area. Is there an over-optimism towards the blockchain technology and will this lead to the creation of systems which are throw-away systems? Will companies implement them only to realize it was the wrong use-case and lose money and time? As always with new technology this risk exists. For example with the Bitcoin blockchain we can see that it is praised for being fully decentralized. But because of its low performance, off-chain transactions are becoming more and more common. Off-chain transactions are essentially performed by a third party, thus removing the fully decentralized feature. This means that instead of trusting a large group of unknown peers, or an institution like a bank, the trust is now shifted to other parties. As seen in this thesis, the performance of blockchains is still an issue and before this can be solved the effective use-cases for blockchain are few.

---

[2] https://www.reuters.com/article/us-africa-landrights-blockchain/african-startups-bet-on-blockchain-to-tackle-land-fraud-idUSKCN1G00YK

[3] https://www.forbes.com/sites/andrewrossow/2018/07/10/top-10-new-blockchain-companies-to-watch-for-in-2018/#6a66c2095600

[4] https://www.forbes.com/sites/michaeldelcastillo/2018/06/06/the-10-largest-companies-exploring-blockchain/#2856a4b91343

7 Conclusion

When comparing permissioned blockchains to distributed databases it is clear that at this moment in time distributed databases are the more mature technique. There are more options of distributed database frameworks available compared to the number of permissioned blockchain frameworks. The available frameworks for permissioned blockchains are all in the early stages of development, meaning that most likely there will more options available as well as more mature options for permissioned blockchains. For build-in support in cloud solutions it is also clear that there is more support for databases. However, permissioned blockchains are on the rise and both Amazon EC2 and Microsoft Azure are starting to support Hyperledger Fabric, even though it is still only on a very small scale. This thesis work fills some gaps in the knowledge needed before mature blockchain technologies become mainstream.

This area of research is new, to the best of our knowledge this thesis is the first work which compares the latency of permissioned blockchains and distributed databases. This makes the results of this work even more interesting and an important contribution to the field of distributed systems.

7.1 Performance

Permissioned blockchains are new compared to distributed databases, yet this thesis shows that they are comparable to older techniques in terms on latency. When it comes to creating linearizable consistency they perform better than older techniques. This clearly shows that there will be several meaningful use-cases in the near future where a permissioned blockchain will be a better choice than a distributed database. This is a surprising conclusion of this thesis since blockchains in general are celebrated for their potential and not their actual performance.

Based on the experiments performed, we can see that Fabric pays for quick reads with a slow transaction flow which gives a higher insert latency. Cassandra on the other hand is more tuned to provide low insert latency at the cost of having a higher latency when reading data. The insert latency for Fabric is lower than the insert latency of Cassandra for the smallest networks. However, the insert latency of Fabric increases more for larger networks and Cassandra has lower latencies for 16 and 20 nodes. Therefore, it can be assumed that Cassandra will continue outperform Fabric for larger networks.

The read latency has large variations for Cassandra because of the techniques, such as caching, that Cassandra uses. There are small variations in read latency for Fabric because the data is always found in the same place. For inserting both systems have similar variations in latency, which are small up to the 75th percentile but some outliers can have significantly higher latency. What causes these outliers is not clear, it is possible that it is fully circumstantial.

When it comes to different loads Cassandra has lower insert latencies than Fabric but Fabric scales better. When it comes to latency of different throughput Fabric can be tuned by specifying the `batchSize` and `batch timeout` after what is the expected load on the system. Since these adjustments can give large differences in latency it is important that the system architect thoroughly investigates which values gives the best latency for the application.

For the different workloads no difference in latency could be seen for Cassandra except that the insert and read latency are a bit lower for workloads with a lower frequency of reads. This is because the read is a more complex operation than the insert and uses more resources. For Fabric read-intense workload 1 yielded higher latencies for read and insert operations. This leads to the conclusion that the peers are the bottleneck and not the ordering service, which is also one of the conclusion of Androulaki et al. [3]. This result might be affected by the co-locating of peers.

The result presented in this thesis does not necessarily apply to all permissioned blockchains and distributed databases, future work should be extended over more instances of both techniques to determine this. When choosing between a permissioned blockchain and a distributed database the most important aspects to consider is the application that should run on top of it. Some things to consider are:

- If the user will need to fetch data quickly then Fabric might be a good choice.
- If the data is going to be updated and/or inserted frequently and accessed more infrequently, then Cassandra is preferred.
- Does the application require linearized consistency? In that case Fabric is a good choice since consensus is pluggable and always integrated in the transaction flow. Cassandra does support it but is not optimized for it.
- Although it is not covered specifically in this thesis the number of data objects and the number of insert and/or update operations on these data objects is an important factor. Blockchains never throw away data and can therefore grow large quickly if the application is not tuned after this fact.

7.2 Hybrid Solutions

Since permissioned blockchains still comes with some limitations in terms of performance and maturity new hybrid solutions have emerged. Postchain is what the company ChromaWay calls a consortium database[1]. Postchain is said to combine the benefits of blockchains with the maturity of distributed databases by leveraging on the desired blockchain properties like linked timestamping and being decentralized yet working together with existing relational database and using SQL[12]. Another example of a similar product is BigchainDB

---

[1] https://chromaway.com/products/postchain/

which is a distributed database with added blockchain characteristics like immutability, decentralization and the option to chose permission property per transaction [2]. This shows that the lines are getting blurred between databases and blockchains and that some companies prefer to cherry-pick the desired features of both technologies. Since neither a strict blockchain-solution nor a strict database-solution is the best option for all problems this is good news. It also illustrates the current gap between how popular the blockchain technology is and how far the technology has actually come.

7.3 Future Work

For future work in this field there are several extensions possible. One possible extension to this thesis is to perform the same experiments on a distributed network with one machine per node. Several network solutions are possible, Microsoft Azure and Amazon EC2 or building it with local servers. This would mean that the results would not have the same type of overhead from the network and this would have to be estimated differently. Using latency as the round trip time is still feasible since it only relies on one clock. The work could also be extended to include more databases and permissioned blockchain frameworks, for example MongoDB, HBase, Parity and Ethereum, in the same experiment. When choosing frameworks, it would be important to consider the properties of them to ensure that the study will get a representative mix of both technologies. Hybrid solutions such as BigchainDB or Postchain could also be included. Another option is to use the benchmark tool Blockbench presented in the paper by Dinh et al. [10] and modify it to handle databases as well. Then Fabric and Cassandra could be compared again, or more frameworks could be included. Another option is to take workloads from YCSB and build new a benchmarking framework which works for both permissioned blockchains and distributed databases.

---

[2] https://www.bigchaindb.com/

Bibliography

[1] Veronika Abramova and Jorge Bernardino. “NoSQL databases: MongoDB vs Cassandra”. In: *Proceedings of the International C* Conference on Computer Science and Software Engineering - C3S2E '13*. ACM Press, 2013, pp. 14–22. ISBN: 9781450319768. DOI: `10.1145/2494444.2494447`.

[2] Veronika Abramova, Jorge Bernardino, and Pedro Furtado. “Evaluating Cassandra Scalability with YCSB”. In: *International Conference on Database and Expert Systems Applications*. Springer, Cham, 2014, pp. 199–207. DOI: `10.1007/978-3-319-10085-2_18`.

[3] Elli Androulaki, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Artem Barger, Sharon Weed Cocco, Jason Yellick, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, and Gennady Laventman. “Hyperledger fabric: A Distributed Operating System for Permissioned Blockchains”. In: *Proceedings of the Thirteenth EuroSys Conference on - EuroSys '18*. ACM Press, 2018, pp. 1–15. ISBN: 9781450355841. DOI: `10.1145/3190508.3190538`.

[4] Eric Brewer. “A certain freedom”. In: *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing - PODC '10*. ACM Press, 2010, pp. 335–335. ISBN: 9781605588889. DOI: `10.1145/1835698.1835701`.

[5] Eric Brewer. “CAP twelve years later: How the "rules" have changed”. In: *IEEE Computer* 45.2 (Feb. 2012), pp. 23–29. DOI: `10.1109/MC.2012.37`.

[6] Rick Cattell. “Scalable SQL and NoSQL data stores”. In: *ACM SIGMOD Record* 39.4 (May 2011), p. 12. DOI: `10.1145/1978915.1978919`.

[7] Edgar Frank Codd. “A relational model of data for large shared data banks”. In: *Communications of the ACM* 13.6 (June 1970), pp. 377–387. ISSN: 00010782. DOI: `10.1145/362384.362685`.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*. ACM Press, 2010, p. 143. ISBN: 9781450300360. DOI: `10.1145/1807128.1807152`.

[9] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed systems - Concept and Design*. Fifth edition. Addison Wesley Publishing Comp., 2012. ISBN: 978-0-13-214301-1.

[10] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. "BLOCKBENCH". In: *Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD '17*. ACM Press, 2017, pp. 1085–1100. ISBN: 9781450341974. DOI: `10.1145/3035918.3064033`.

[11] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. "The Bitcoin Backbone Protocol: Analysis and Applications". In: Springer, Berlin, Heidelberg, 2015, pp. 281–310. DOI: `10.1007/978-3-662-46803-6_10`.

[12] J. Michael Graglia and Christopher Mellon. "Blockchain and Property in 2018: At the End of the Beginning". In: *Innovations: Technology, Governance, Globalization* 12.1-2 (July 2018), pp. 90–116. ISSN: 1558-2477. DOI: `10.1162/inov_a_00270`.

[13] Gideon Greenspan. *MultiChain Private Blockchain-White Paper*. Tech. rep. URL: `https://www.multichain.com/download/MultiChain-White-Paper.pdf`.

[14] Hyperledger. *Architecture Explained — hyperledger-fabricdocs master documentation*. 2017. URL: `https://hyperledger-fabric.readthedocs.io/en/release-1.1/arch-deep-dive.html` (visited on 06/25/2018).

[15] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. "Benchmarking scalability and elasticity of distributed database systems". In: *Proceedings of the VLDB Endowment* 7.12 (Aug. 2014), pp. 1219–1230. ISSN: 21508097. DOI: `10.14778/2732977.2732995`.

[16] Avinash Lakshman and Prashant Malik. "Cassandra". In: *ACM SIGOPS Operating Systems Review* 44.2 (Apr. 2010), p. 35. DOI: `10.1145/1773912.1773922`.

[17] Leslie Lamport. "Paxos Made Simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25. DOI: `doi:10.1145/568425.568433`.

[18] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: (2009). URL: `www.bitcoin.org`.

[19] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[20] Joao Sousa, Alysson Bessani, and Marko Vukolic. "A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, June 2018, pp. 51–58. ISBN: 978-1-5386-5596-2. DOI: `10.1109/DSN.2018.00018`.

[21] Harish Sukhwani, Jose M. Martinez, Xiaolin Chang, Kishor S. Trivedi, and Andy Rindos. "Performance Modeling of PBFT Consensus Process for Permissioned Blockchain Network (Hyperledger Fabric)". In: *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, Sept. 2017, pp. 253–255. ISBN: 978-1-5386-1679-6. DOI: `10.1109/SRDS.2017.36`.

[22] The Apache Software Foundation. *Documentation*. URL: `http://cassandra.apache.org/doc/latest/architecture/dynamo.html` (visited on 06/21/2018).

[23] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. "Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: the Consumers' Perspective". In: *Proc. Conference on Innovative Data Systems Research (CIDR)*. Vol. 11. 2011, pp. 134–143.

[24] Xiwei Xu, Ingo Weber, Mark Staples, Liming Zhu, Jan Bosch, Len Bass, Cesare Pautasso, and Paul Rimba. "A Taxonomy of Blockchain-Based Systems for Architecture Design". In: *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, Apr. 2017, pp. 243–252. ISBN: 978-1-5090-5729-0. DOI: `10.1109/ICSA.2017.33`.